DOGS: Organized Graph Semantics

E3-210

Department of Computer Science
Aalborg University

TITEL:

DOGS: Organized Graph Semantics

Semester:

Dat2
2. Febuary - 28. May, 2004

Group:

E3-210, 2004

Members:

Kristian Ahlmann-Ohlsen
Anders Bennett-Therkildsen
Jeppe Carlsen
Jesper Kristensen
Arne Mejlholm
Søren Pedersen
Jacob Volstrup Pedersen

SUPERVISOR:

Lone Leth Thomsen

Synopsis:

This report documents the process of developing the language DOGS (DOGS: Organised Graph Semantics) designed for students working with graph algorithms, and a compiler `dogsc` that can translate programs written in DOGS. The language contains data types, such as graphs, sets, vertices, and edges made especially for working with these algorithms. One of the main criteria when designing DOGS is readability, meaning that it should be intuitive to write programs in the language and it should be similar to ordinary pseudocode.

The front-end of `dogsc` is built with the help of the SableCC tool and it targets the Java Virtual Machine with bytecode through the Java Assembler Interface (Jasmin).

Copies: 9

Report – pages: 149

Appendix – pages: 74

Appendix – cdrom: 1

Total pages: 241

Preface

The purpose of the DOGS language being developed in this project is to provide an intuitive way of writing graph algorithms from pseudo-code examples and translate these using the `dogsc` compiler also being developed.

Although this project focuses on developing a compiler, we specify the behaviour of DOGS formally in operational semantics to ease the process of developing the compiler.

The compiler is written in Java and it takes advantage of the Visitor pattern, a lexer, and a parser provided by the compiler-compiler tool SableCC. The target machine is the Java Virtual Machine (JVM), through the Java Assembler Interface (Jasmin) creating Java bytecode.

We structure the report in three main phases:

- Analysis: An inspection of already existing languages and graph theory is introduced in order to design the language specifying the design criteria of the language.
- Design of DOGS : Language syntax, type system, and semantics are presented to chart the guidelines for implementing the language..
- Design of `dogsc` : Designing the compiler for DOGS, `dogsc`, involves an elaboration of the type of the DOGS grammar and a discussion of compiler-compiler tools that can aid us in developing it.
- Implementation: In this final part, the implementation of the compiler is accounted for, and an empirical investigation of the correctness of the compiler is conducted.

We include a CD, containing

- The `dogsc` code with Javadoc.
- An executable jar file.
- DOGS source files.
- The necessary library files.
- SableCC grammar file.
- SableCC generated overview of the grammar.

- The complete report in PDF format.

Kristian Ahlmann-Ohlsen	Anders Bennett-Therkildsen
Jeppe Carlsen	Jesper Kristensen
Arne Mejlholm	Søren Pedersen

Jacob Volstrup Pedersen

Contents

Part I

Analysis Document

Introduction

In the first sections of the analysis we will introduce the "problem at hand" of the project and discuss the motivation for working with it. This will lead to a specification of the goals in the Problem Definition. Furthermore, we will discuss possible methods to structure the project.

The next sections of the analysis covers how to address the goals specified in the Problem Definition and what has to be accounted for when reaching the design phase. It is therefore worth noting that the analysis will focus on different considerations so that we are qualified for making design-specific decisions when we start designing the language and compiler.

We will start out introducing a basic part of the graph terminology which is necessary when we discuss graph-related problems as well as a brief inspection of the ways to represent graphs. Dijkstra's Algorithm will be presented in pseudo-code in order to give a feeling of what the goals of the project will be and more importantly what means we need to reach these goals.

From here we will begin pinpointing the requirements of our language which will include the concepts of data types, data structures, control structures, etc.

The last part of the analysis will focus on the language evaluation criteria. These will be discussed and assessed along with a discussion of the four main programming paradigms which will play an important role in the design of the language.

Chapter 1

Problem Analysis

Graphs are some of the most fundamental data structures in the field of computer science and they can be used to model many different systems: roadmaps, electric circuits, tasks to be executed in a particular order, etc. Operations on graphs can be described in very concise functions by a well-defined pseudo-language (such as that used in [CLRS01]) and should be quite easy to use in programming languages.

In the following we will examine the remedies for working with graphs in programming languages. We will inspect some of the commonly used languages[1], which include C, C++, Java, Pascal, Perl, and Python, as well as examine whether or not programming languages constructed with the specific purpose of working with graphs already exist. The languages will in this section be compared on how easy it is to express graphs and graph algorithms in a way that is straightforward and understandable to readers of [CLRS01].

1.1 Programming Graphs

We have gathered that C and Perl have libraries for working with graphs and have experimented a bit with the use of those two. Below are five code examples (Listing 1.1 is in the syntax of the pseudo-code from [CLRS01]) on how to iterate over every vertex in the graph G.

```
1 FOR each vertex ϑ ∈ V[G]
2   Print ϑ
```

Listing 1.1: The `for each` statement in pseudo-code

```
1 for (vertex = list_head(&graph_adjlists(G)); vertex != NULL;
2    vertex = list_next(vertex)) {
3    printf("%s", vertex.name);
4 }
```

Listing 1.2: The `for each` statement in C

[1] The selection of these were based on [Lab]

When considering Listing 1.2 (written in C using `<graph.h>` from [Lou99]) it is not only much harder to type than the first example (with a much higher risk of programming errors), but also much harder to comprehend. In order to easily understand that code example, one would have to be very proficient in the use of C.

```
1 begin
2   v := g.vertices;
3   for i := 1 to length(v) do
4     begin
5       writeln(v[i].name);
6     end;
7 end;
```

Listing 1.3: The `for each` statement in Pascal

Listing 1.3 illustrates an example of a graph in Pascal. We notice that the language has somewhat intuitive syntax, espicially the `begin..end` scope delimiters. In Pascal, however, one cannot define a graph type without the use of records. Using records forces to access each member of the record with the `.` notation, which is rather non-intuitive when working with graphs.

```
1 @v = $G->vertices;
2 foreach(@v) {
3    print $_;
4 }
```

Listing 1.4: The `for each` statement in Perl

Listing 1.4 (written in Perl using the `Graph::Base` library) is a bit more "friendly" but is still somewhat alien to programmers not familiar with Perl.

```
1 Graph = {'A':  ['B', 'D'],
2          'B':  ['C'],
3          'C':  ['D'],
4          'D':  ['A']}
5
6 for vertex in Graph:
7    print vertex
8    for adjacent in Graph[vertex]:
9       print adjacent,
```

Listing 1.5: The `for each` statement in Python

The final example, in Listing 1.5, is Python, which allows constructs like those in line 1-4, for the inherently difficult task of representing graphs in computer languages. For iterating over each element in the graph, the lines 6 and 7 is a clever usage of the built-in data structures of the language (dictionaries and lists). The language has really no grasp of vertices and nodes, just elements in lists. Adding a feature to the program in line 8 and 9 so that it also prints the whole adjacancy list causes us to notice that in order to get the adjacent edges, we have to de-reference

them from the dictionary `Graph[vertex]` which ruins the "illusion" of a graph. Programming graphs in Python is still rather simple compared to C, but there is room for improvement.

It seems that none of the libraries we have found fully live up to our expectation of an easy-to-use and comprehensible syntax. Still, we have to examine if some of the other languages support graphs (or graph-like structures) in a more "natural" way.

Using one of three object-oriented languages (C++, Java, and Python), one could make a package that allows representation of graphs as advanced as necessary. This, however, would still restrict the user of the package into using a somewhat different notation from that of [CLRS01] and as such force the user to concentrate on syntactic details rather than algorithmic details.

Now that we have seen that none of the widely used languages examined here have built-in support for working with graphs, the next step is to examine whether specialized languages customized to work with graphs exist. This, however, has proved somewhat difficult since no such languages known to us are available. Though, we cannot rule out the possibility that a language designed for working specifically with graphs exists.

Following this revelation it is motivating to develop a language that will provide support for working with graphs and ease the process of implementing graph-specific algorithms in an organized manner (hereafter known as). Thus, the goal of this language, which will be referred to as DOGS(DOGS: Organized Graph Semantics) is to allow focusing on the actual algorithm design rather than programming and computer related details.

1.2 Problem Definition

The goal of this project is to develop a compiler, `dogsc`, for a programming language being developed in the process, named DOGS, that will have native support for working with graphs and graph algorithms.

1.3 Focus

Having emphasized the goal of developing a compiler that can translate programs in a programming language also being developed we state that this is a Language and Compiler project. This charts guidelines for the focus of the project.

We design DOGS using techniques from [Hüt04] and [WB00] which comprises the making of both an unambiguous syntax that can be implemented as a LALR[2] grammar and formal semantics that will be utilized to specify the behaviour of DOGS programs.

Developing the `dogsc` compiler entails a choice of virtual machine suitable for handling common language functionality, e.g. control and branching structures, arithmetic and boolean expressions, input/output, etc., as well as the graph-specific part of DOGS.

[2]Discussed in Section 7.3

1.4 Selection of Method

To realize our goals, preliminary considerations of the structure of the project are necessary. The phases of our project will broadly assemble those of the Object-Oriented Analysis and Design method as described in [MMMNS00] in that we will walk through an analysis, design, and implementation phase.

In the analysis phase we will conduct an examination of existing programming languages to help identify the parts of DOGS, which will result in an assessment of language design criteria.

The design criteria from the analysis direct our design choices which will influence the language syntax, type system, and semantics. As a SPO-project, the choice of providing a formal semantics for DOGS is not a requirement, however we judge it more valuable to the implementation than an informal specification, thus aiding us in the development of the compiler.

Developing `dogsc` forces us to consider the choice of virtual machine, compiler-compiler tools, and related tree traversal patterns. The choice of virtual machine depends on considarations regarding portability (i.e. how widespread it is), among others. The choice of compiler-compiler tool is dependant of the classification of the DOGS language which also affects the choice of traversal pattern.

1.5 Target Audience

The target audience of DOGS is people studying (graph) algorithms in general, e.g. students following the course Algorithms and Data Structures. As such, the main purpose of DOGS is to provide these students with an easy-to-use programming language for implementing algorithms from pseudo-code. Although it is the intention that the language will be somewhat specialized, it should still provide enough language orthogonality to allow creation of all-purpose programs.

Chapter 2

Applied Graph Algorithms

Adhering to the Problem Definition, we will in this part of the analysis discuss the requirements for developing a programming language with native graph support. This involves a formal presentation of essential graph terminology as well as a discussion of the ways to represent graphs, followed by a presentation of Dijkstra's Algorithm, which will help us identify required language functionality.

2.1 Introduction to Graphs

A graph $G(V, E)$ is a structure consisting of a set of vertices V and a set of edges E. An edge is a connection between a pair of vertices $u, v \in V$, and we will use the notation (u, v) for such. An edge can have *orientation*, which means that $(u, v) \neq (v, u)$. If an edge does not have an orientation, the two edges (u, v) and (v, u) are considered to be the same edge.

2.1.1 Types of Graphs

It is important that our language supports a fundamental variety of graphs in order to aid the programmer in working with graph-related problems. Therefore we will present the different types and formalize them in definitions based on [Ros03]. Note that since the subject of this project is not the study of graphs, we will not discuss their formal definitions in greater detail. However, we believe it necessary to outline precise definitions rather than just informally describe the different types since they indeed are the subject of our programming language.

The simplest type of graph is the undirected graph, which means that the edges do not have an orientation.

Definition 1 (Undirected graph). *An undirected graph $G(V, E)$ consists of a set vertices V and a set E of unordered pairs of elements of V called edges.*

This type does not suffice when working with problems that require the use of multiple connections or relations between a pair of vertices. Such graphs are known as *undirected multigraphs*. An undirected multigraph is shown in Figure 2.1.

The aforementioned types of graphs will suffice when the programmer does not need other information regarding the relations between vertices, other than knowing

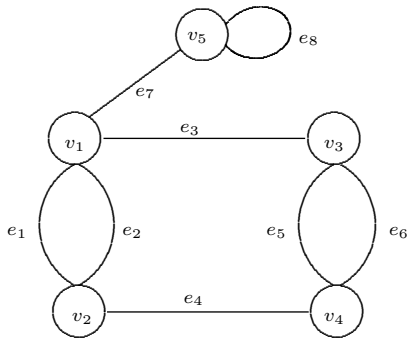

Figure 2.1: An undirected multigraph

which vertices are connected. If he (or she) needs information about how the edges are oriented, *directed graphs* need to be introduced.

Definition 2 (Directed graph). *A directed graph $G(V, E)$ consists of a set of vertices V and a set E of ordered pairs of elements of V called edges.*

A directed graph is shown in Figure 2.2. This type of graph can also be extended to *directed multigraphs*, in which multiple edges with the same orientation are allowed.

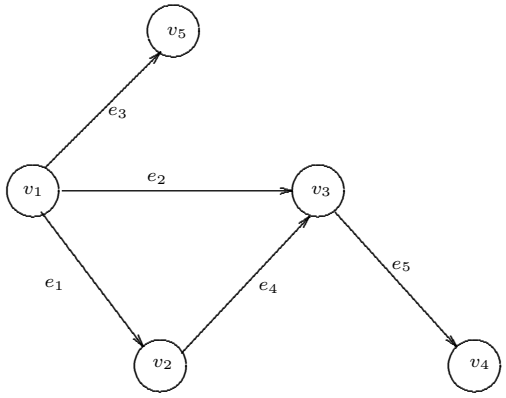

Figure 2.2: A directed graph

In many problems represented by graphs, it is necessary to have a set of values (called *weights*) assigned to each edge in the graph. A graph where all edges are assigned weights is called a *weighted graph*. An undirected, weighted multigraph is shown in Figure 2.3.

Definition 3 (Weighted graph). *A weighted graph $G(V, E)$ can either be an undirected or a directed graph, with the addition of a weight function $w(e)$, that for each edge $e \in E$ assigns a set of values. Each edge must be assigned the same number of weights.*

The values assigned by the weight function are often integer values, but can in theory be any type of value depending on the problem the graph is used to represent.

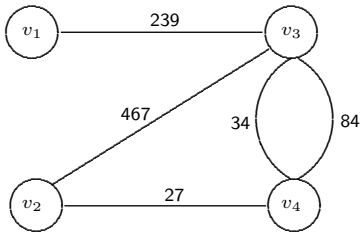

Figure 2.3: An undirected, weighted multigraph

Note that in some graph algorithms, values are also assigned to the vertices of the graph (often referred to as the *state* or *colour* of the vertex).

Graphs can be divided into several more subcategories. For example, we often differ between graphs allowing *loops* and graphs that do not allow loops (that is, edges of the type (v, v)). However, we do not deem it necessary to present more definitions in this section, as those presented are enough to analyse implementation of general types of graphs.

2.1.2 Representation of Graphs

There are several ways to represent graphs [Ros03], two of which are considered to be the standards. One way is as a collection of adjacency lists and the other is as an adjacency matrix.

The adjacency list representation of a graph $G(V, E)$ consists of an array with a list for each vertex in the graph (e.g. an array of $|V|$ elements). Each of these lists contain the adjacent vertices to the respective vertex. The adjacency list representation is often preferred when working with sparse graphs (e.g. a graph $G(V, E)$ for which $|E| < |V|^2$), because it gives a compact representation of these. Compared to many other ways of representing graphs, the adjacency list uses less memory to represent a graph.

The adjacency matrix representation of a graph $G(V, E)$ consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that:

$$a_{ij} = \begin{cases} 1 & \text{if the edge } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

This representation can be preferred, because it can be determined if there is a connection between two vertices in constant time. This property comes at the cost of the additional memory needed for the presentation (compared to adjacency lists).

2.2 Dijkstra's Algorithm

In this section we present Dijkstra's Algorithm. There are several reasons for introducing such an example. One is to illustrate the pseudo-code from [CLRS01], which has motivated for developing a language for working with graph algorithms. Also, the example can be referred to in later sections when pointing out some general characteristics in graph algorithms. Finally, we will use it after the language implementation to test for correctness of the compiler.

Dijkstra's Algorithm is used to find the shortest path in graphs and it works on weighted graphs where a weight function assigns a single positive real number to each edge in the graph. The weights represent the distance between two vertices, thus the problem of finding the shortest path from a vertex a to all other vertices corresponds to finding the sequence of edges from a to the other vertices for which the sum of weights are minimal.

In this context it is not important to discuss why the algorithm always succeeds in finding the shortest paths, but to understand the pseudo-code below, some explanation of central parts of the algorithm will be given.

```
1  Dijkstra(G, w, s)
2    for each vertex v ∈ V[G]
3      do d[v] ← ∞
4         π[v] ← NIL
5    d[s] ← 0
6    S ← ∅
7    Q ← V[G]
8    while Q ≠ ∅
9      do u ← Extract−Min(Q)
10       S ← S ∪ u
11       for each vertex v ∈ Adj[u] and v ∉ S
12         do if d[v] > d[u] + w(u, v)
13               then d[v] ← d[u] + w(u, v)
14                    π[v] ← u
```

Listing 2.1: Dijkstra's Algorithm written in pseudo-code

The algorithm takes the graph G, the weight function w, and the start vertex s as input. The algorithm makes use of a table $d[v]$ to store the distance from s to a vertex v, and a table $\pi[v]$ to store the predecessor to a vertex v. The predecessor u to a vertex v is the vertex from which v can be reached by the distance $d[v]$ at a path from s to v. The algorithm uses a set S to store the vertices for which the distance $d[v]$ is the actual distance of a shortest path from s to v. Furthermore, a min-priority queue of vertices Q is used in which the vertices are prioritized with respect to their d values.

In each iteration of the while loop the vertex with the least value in d is extracted from Q and stored in the variable u. This vertex u is added to the set S, and for all vertices v adjacent with u and not already in S, the values $d[v]$ are updated, if the vertex can be reached by a shorter distance by using the edge (u, v) than the distance value already in $d[v]$.

The loop terminates when Q is empty. In this case all vertices have been added to the set S. Since S contains the vertices for which the value $d[v]$ is the distance of a shortest path from a to v, the algorithm has found the shortest distance from a to all other vertices in the graph.

The shortest paths have also been found and can be constructed by backtracking in the predecessor table π.

2.3 Early View of the Language Requirements

The presentation of Dijkstra's Algorithm gives us the opportunity to reflect on some of the requirements or our language. We can identify the need of *primitives*, i.e. numbers. These are completely essential, confer line 5, for instance. Symbolic values such as *sentinels* from line 3 are also needed. Further, we can see that *conditional branching* (identified in lines 12 to 14) and *loops* (identified in line 8) are necessary in order to make decisions from a series of choices and to iterate over, say, a set, respectively. In this context, *boolean algebra* is also required to maintain loops, to choose between different conditions, to make comparisons, etc. Data structures such as *sets* (cf. line 6), *queues* (cf. line 7), and *lists* are also required. In the next sections, we will analyse the needs of the following requirements more thoroughly:

- Primitive data types
- Composite data structures
- Arithmetic and boolean expressions
- Control structures
- Input and Output structures
- Concurrency
- Mutual exclusion
- Packages/libraries
- Error handling

Summary

This chapter has served multiple purposes. A basic introduction to graph terminology has been the offset to understand what a graph really is and further to discuss how to represent graphs. Dijkstra's Algoritm has been presented in pseudo-code to help illustrating the syntax of our language, and furthermore it has given us the opportunity to reflect on the requirements of our language.

Chapter 3

Analysis of Language Features

In the previous section we ended up with a list of identified requirements that we need to assess the necessity of when designing our language. In this chapter we will walk through features available in the four languages: ANSI C, Turbo Pascal, Perl, and Python[1]. The reason for doing this is to take advantage of the way others have build up a programming language, its features and to take advantage of experiences with those languages.

We will examine how data types are represented and which are provided in the four languages. We will also discuss how the languages provide operations on these data types and the possibility of creating special needed data types, e.g. structures i C.

Furthermore, we will evaluate on how to create control structures and which impact these have on the languages as well as discuss language features like input/output, concurrency, error handling, and language extensions.

3.1 Primitive Data Types

Not all programming languages have primitive data types, in fact both Python and Perl do not have the primitive data types that C has. In C and Pascal, numbers are represented by integers and reals. When working with integers in C the programmer can either have them signed or unsigned in order to represent negative or only positive numbers. This is, however, not possible in Pascal. In C it is also possible to represent integers in different sizes, ranging from byte to long.

To represent real numbers in C the types `float` and `double` are provided, differentiating in how large the representation of the real is.

C does not have a primitive data type to represent strings, it only has representation for a single character, `char`, so in order to work with text strings in C the programmer has to represent these by an array of single characters. This is very different from Pascal where the primitive data type `string` is provided, which has a size limit of 256 characters.

The type `boolean` in Perl and C is represented by zero and one, however C accepts any positive non zero integer as true. Python, however, uses both of the previously mentioned and the use of the keywords `True` and `False`. Additionally,

[1] From now on ANSI C will be referred to as C and Turbo Pascal as Pascal

both Python and Perl allow any data type to be evaluated as a boolean, e.g. an empty string is false, while any non-empty string is true. The boolean in Pascal is manipulated by the keywords `true` and `false`.

The most commonly known representation for sequencing is probably the primitive static sized array as it is known from C. In Pascal, the array works in the same way as in C, but when initializing it the programmer has to specify the size as a range of either chars or integers. However, Perl and Python provide support for sequences on a higher level of abstraction. In both Python and Perl, strings, integers, and reals are represented by only one type, `scalar`, that can contain any of the types.

In fact, both Python and Perl only have few dynamic sized basic types. Besides scalars Perl has arrays and hashes, while Python has lists and dictionaries, which basically are somewhat the same. Python has an in-mutable data type called `tuple` in addition.

Dijkstra's Algorithm uses numbers to represent the cost of a weight and the distance to a given edge. If efficiency is of the essence, many different primitive representations of numbers, like the ones C or Pascal provide, may be useful to cut down on memory usage. However, if efficiency is not very important, perhaps the representation of a number can be minimized to one type, the real, as is the case in Perl and Python.

Similarly, when deciding upon the need of sequence structures, if resources and efficiency is important, a static basic type, like the array, is preferable. If, on the other hand, the language should allow high-level abstraction of the operations on sequences, a dynamic representation may be preferred.

3.2 Composite Data Types

Composite data types are constructs that allow the programmer to form a new data type from primitive data structures and composite ones.

In C, this is done by the keywords `struct` and `union`, which behave in the same way, but union declares a variable, which is composed of primitive data types, instead of a structure. Below is presented a small example of how one could represent an edge with a name (only one character) and a state (an integer) with a `struct`.

```
1  #include <stdio.h>
2
3  struct vertex{
4    char name;
5    int state;
6  };
7
8  int main(void){
9    struct vertex a;
10   a.name = 'A';
11   a.state = 10;
12
13   printf("Name: %c\tState: %d\n", a.name, a.state);
```

```
14 }
```

Listing 3.1: Working with structs in C

In lines 3 to 6 we declare the stucture, in line 9 we initialize it, and in lines 10 and 11 we initialize the members of the structure. Line 12 references each of the members in the structure and prints them. A pendant to this is Pascals `records` and Pythons use of object-oriented classes (also present in Perl to some extent). C and Pascal also support declaration of enumerations with the keywords `enum` and `type`.

Many graph related types cannot be represented with primitive data structures, so in order to allow the programmer to create any composite types, the language must provide some kind of composite data structure declaration. An example could be the representation of a tree, for instance. A tree is a collection of nodes and edges, which cannot be modelled with primitives.

3.3 Arithmetic and Boolean Expressions

Some of the different arithmetic and boolean expressions used in the four languages have been examined and in we have listed the arithmetic operators that three of them share in Table 3.1.

Operators	**Description**
+, −	plus, minus
∗, /	Multiplication, division
%	Modulo
<, <=	Less than, less than or equal
>, >=	Greater than, greater than or equal
=	Assignment
==, ! =	Equal and not equal

Table 3.1: Shared arithmetic operators from the three languages Perl, Python, and C

The arithmetic operators in Pascal are not included in Table 3.1 because some operators in this language differentiate from those of the other languages. In Pascal, assignments are done with := , ! = (not equals) is done with <>, and for the == (equals) expression Pascal provides a single = character. Furthermore, when working with modulo in Pascal, the programmer has to use the `mod` keyword. This also applies when division should return an integer instead of a real. In this case, the programmer has to use the keyword `div`.

Besides from arithmetic expressions the languages also have logical expressions and these are as well implemented in two different ways. Perl, Python, and Pascal provide the written notations `OR` and written `AND` for mathematical $\vee$ and *land*. In C, it is only possible to use the logical expressions && for $\vee$ and || for $\wedge$. This notation is also available in Perl.

The languages also have a set of different bitwise operators, listed in Table 3.2.

Operators	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive or, XOR
<<	shift left
>>	shift right

Table 3.2: Bitwise expressions

We have discovered that both Perl, C, and Pascal have the increment ++ and the decrement −− expressions, but Python has not. This seems a bit strange because this is the only high-level language that we know of that does not support these operators.

All of the arithmetic expressions in this section can be used to make true/false evaluation. To represent the evaluation of this expression the `boolean` data type is provided (described in Section 3.1).

Pascal and Python further provide the keyword `not` instead of the negation symbol !, which is used by the two other languages.

Programming algorithms becomes a tedious task if the language is lacking arithmetic expressions to make calculations and evaluations of these. Conferring Dijkstra's Algorithm from Section 2.2 it is clear that arithmetic expressions are used throughout Dijkstra's Algorithm to do evaluations and calculations.

In order to reach a high level of abstraction, the increment and decrement operators are needed. Also the written `OR` and `AND` operators give a programming language a higher level of abstraction than just using the logical operators, thus increasing readability of the code.

3.4 Control Structures

In the mentioned four programming languages there are three different types of control structures: looping, branching, and recursion. In the following, we will use the notations:

- <**expr**>: is a logical expression.
- <**block**>: is a collection of one or more statements.
- <**iass**>: is an integer assignment.
- <**int**>: is an integer value.

All four languages support conditional branching, using the `if...then...else` structure, which can be nested arbitrarily.

Recursion (a function calling itself within itself) is allowed in all four languages, however in Python it is limited to an implementation dependent recursion depth.

3.4.1 Loops

Loop structures execute code inside a block while an expression is evaluated to either true or false. The loops can be manipulated by the keywords `break` and `continue`/`next`.

C provides the following loop structures:

- while <expr> <block>
- do <block> while <expr>
- for (<init>;<expr>;<block>) <block>

The while structure is common in all four languages and is a common feature in most imperative programming languages.

When working with loop structures in Pascal we have the following unique structures compared to the other three languages:

- repeat <block> until <expr>
- for <iass> to <int> do <block>

Perl provides a very wide range of control structures and many of them work in the same way. All the loop control structures in Perl that are not in C is listed below.

- until <expr> <block>
- do <block> until <expr>
- unless <expr> <block>
- for var (array) <block>
- foreach (array) <block>

The `do...until` loop in Perl works in the same way as the `repeat...until` loop in Pascal. The `for` and `foreach` differentiate from the other loop structures because these iterate over a sequence rather than an evaluation of an expression. The latter was presented in Listing 1.4, page 14.

Besides `while`, Python only provides one looping structure which is the `for <target> in <iterable>` seen in Listing 1.5 in Section 1.5, page 14. It declares a variable, the `<target>`, which can be used inside the loop. To allow this, the `<iterable>` is an object that can be iterated over which is lists and strings. In order to iterate over an indexed array, we would have to create a sequence to iterate over: `for i in range(0,10)`.

Adhering to Dijkstra's Algorithm we can identify the conditional branching and `for each` structures. Conditional branching is widely used in algorithms and a necessity in an orthogonal programming language. The `for each` expression can be represented by a construct similar to Perl's `foreach` and Python's `for...in...`. The latter has a cleaner syntax, though.

There is a wide variety of looping structures available but in order to satisfy minimal needs, all a language really needs is `for...` and `while...` constructs, however specialized constructs like `repeat...until` can sometimes be useful.

A tree is a special, recursive-structured type of graph and when working with it, recursion provides expressive solutions. Further, when working with sets and sequences it is useful to create divide-and-conquer algorithms, which are recursive.

3.5 Input and Output

There are many aspects in the use of input and output, e.g. printing to a console, persisting data, etc.

All useful languages support input and output. C, Perl, Python, and Pascal are no exceptions.

The way to handle input and output is through streams which is a data queue. A stream could contain the keys pressed on the keyboard or the data read from a file. The way a stream is handled is the programmer's responsibility and would often be by characters in the stream. When managing input and output in this way it would be easy to let networking communication work in the same way, letting two applications exchange streams with data.

The usefulness of a programming language is drastically reduced if it does not support input and output. Running, for instance, series of tests of finding the shortest path in a graph would amount to a limited use if the data cannot be persisted. Further, if the results cannot be printed the test would be worth very little.

3.6 Concurrency

When working with concurrency we distinguish between processes and threads. A program normally runs in one process, which can have one or multiple threads. Sometimes it has the need to spawn extra processes, which often need to share ressources.

Python, Perl, and C fully support concurrency, while Pascal does not support it at all. C, Perl, and Python provide two different methodes to handle concurrency: The `fork` statement that spawns a new process, which is a copy of the original process (the new process continues from the state where it was forked) and multiple threads inside a single process, which share resources.

In C, forks from the system library <`unistd.h`> are used to spawn a new process. Threads are provided in the system library file <`pthread.h`>. Python has the same constructs in the packages `os.fork`, `thread`, and `threading` which is built on `thread`. Perl needs the module `Thread.pm` which is to be installed separately.

3.6.1 Mutual Exclusion

Sharing resources between a number of processes and threads is problematic. To avoid two threads modifying the same data, certain controlling mechanisms can be introduced. One commonly known is semaphores which both Python, Perl,

and C support. C provides them in <semaphore.h> and Python has the class semaphore. In Perl semaphores are provided in the IPC::SysV, IPC::Semaphore, and Thread::Semaphore packages.

When working with an application that is performing some kind of calculation and at the same time needs input from the user, the language in question needs to support concurrency, e.g. threads, if the application should continue to calculate while recieving input.

If two or more applications need to communicate and exchange data at runtime, the need for concurrency seems obvious. This could be two computers working with the Travelling Salesman Problem, which exchanges individuals between their populations.

So, concurrency is needed to interact without halting the entire application. When making a new language it is one of the fundamental things to include in order to make the language useful.

3.7 Error Handling

Errors are often handled by the operating system, e.g. division by zero and pointers to invalid or forbidden resources. When designing a programming language it is a matter of design choice whether to let the language handle errors explicitly or leave it to the operating system. The latter is the case in C [KR88].

However, many C compilers check for many errors at compile time in order to avoid these. The Gnu Compiler Collection (gcc), for instance, checks for division by zero if a number is divided with zero. Though, if a variable of type integer is declared and its value is assigned to zero later in the program, it is left to the operating system to handle the situation where division by this variable is executed.

Like the gcc compiler, the Perl interpreter performs some checks before executing the source code at runtime. If an error occurs at runtime it is left to the operating system to handle the situation.

Pascal performs compile time checks and has a mechanism that raises a runtime error description at runtime.

Unlike the other languages being examined Python has a rather large scheme for detecting errors called exceptions (see [Mar03]).

If an error is detected an exception is raised and Python, like Java, allows the programmer to explicitly raise exceptions and handle them. As Python is an interpreter it does not, in contrast to Perl, perform checks before executing the source code.

An error detection scheme is an important part of a language. Debugging C programs, for instance, can be very time consuming, especially when the errors are related to memory access and pointers.

Bearing in mind the target audience of our programming language, it would be worthwhile to consider having graph related runtime error checking. One possible graph-related error could be negative cycles in a weighted graph with negative edges. A solution to this could be to have a feature in the compiler that checks for negative

cycles. Another graph-related error could occur if the programmer defines a weighted graph that has multiple weights on each edge and the amount of weights on an edge differs from another edge. This can also be checked at compile time but would slow down the compile process, so obviously there is a tradeoff.

3.8 Extensions

In large programming tasks, or in distribution of code to other programmers, it is convenient to have some kind of technique to extend and reuse code. To provide this in a programming language, some kind of extension system will be usable.

In C, extensions are created by making files called header files. These are used to link a group of source files into a library. The header file is used to declare all functions from the different source files. Inside each source file the header file is included by using the `#include <package.h>` statement for standard packages and `#include ‘‘package.h’’` statement for user-defined libraries.

The package system in Pascal is similar to C's, except there is no header file for declaration of the functions or procedures because these are declared inside the source file. In top of a file that relies on the use of functions from a library file, the `uses unit-name` statement is used.

In Perl, the programmer has the possibility to make modules as an extension. The modules are very different from the way C libraries are build. A Perl module consists of a single file only, with the possibility to use other modules with the `use <module-name>` statement. This is in contrast to C where a library commonly consists of a collection of files.

The Python module system is similar to Perl's, except from the fact that in Python a package can be made as a collection of modules. The inclusion of these modules is done by using the `import module-name from package` statement.

Another language worth considering is Java, as it has an interesting mechanism for handling language extensions. Like Python it operates with classes, packages and it is executed through an interpreter. The java compiler compiles all needed class files, which allows good modular structure.

When working with a problem and an algorithmic solution has been found, it would be practical to provide an easy way for the programmer to distribute the solution to other programmers and end users. This would be convenient to support directly in the language as it is seen in the four languages from the previous section.

Summary

The inspection of C, Pascal, Perl, and Python has given us insight into what is required in a language being developed. Some of the requirements are more obvious than others, e.g. arithmetic and boolean expressions against packages and error handling, although this is dependant of what the purpose is with the language. Since DOGS is to be a language with native graph support for students there are clearly different priorities in the selection of requirements. Though, before we start

evaluating these we will examine which paradigm to choose, as this also affects our decissions in the design.

Chapter 4

Programming Paradigms

When designing our programming language, we need to decide which programming paradigm to follow. This decision will be based on an analysis of the four main paradigms, which we will present in the next sections. We will discuss the pros and cons of these four paradigms and weigh them against our goals from the Problem Definition as well as make an assessment of the consequences of choosing between them.

4.1 The Four Main Programming Paradigms

There are four main programming paradigms [Nør]: The imperative, the functional, the object-oriented and the logical paradigm, each different from the others. In the following they will be introduced and assessed in relation to the requirements of this project. The following is based on [Nør] and [Wik].

4.1.1 The Imperative Paradigm

Informally, in the imperative programming style a program consists of a sequence of computational steps, or commands. The execution of these commands, which happens in an order governed by control structures, brings the program into a new state defined by the contents of the memory.

This rather abstract description of the imperative paradigm is presented because Dijkstra's Algorithm from Section 2.2 is presented in an imperative pseudo-code. Here we see control structures in form of loops and changes in state in the form of assignments. In fact, all of the graph algorithms from [CLRS01] and [Ros03] are presented in imperative pseudo-code, which is one of the main reasons for considering this paradigm. Also, everyone in the group has had experience in programming in this paradigm before, so we can focus on other project-related problems instead of learning a new paradigm which could be a time-consuming process. Further, the semester course "Programming Languages and Compilers" that are concerned with the development of programming languages and compilers, uses this computation model in concrete design and implementation examples. These are the main reasons for choosing the imperative paradigm.

4.1.2 The Functional Paradigm

In contrast to imperative programming, functional programming is based on evaluation of functional expressions, not on execution of commands.

In our considerations of this paradigm we have to take into account the fact that practically all members of the group have had limited experience in functional programming and absolutely no experience in programming logical, which means that a possibly time-consuming process of learning to program in one of these two paradigms is unavoidable.

According to [Mit03], imperative languages are more commonly used than functional languages. Therefore, it could be argued that due to previous programming experience, chances are that the target audience (cf. Section 1.5) of DOGS will be more familiar with the imperative paradigm than the functional one.

4.1.3 The Logical Paradigm

In short, logical programming is based on axioms, inference rules, and queries. Execution of a program is a systematic search in a set of facts with the use of inference rules.

We have already discussed the logical paradigm to a certain extent in the previous section where some of the argumentation against using this paradigm is presented.

[Mit03] states that this paradigm restricts a language to only one data type. This restriction is unwanted in our programming language so this argument is rather strong against choosing this paradigm.

When considering the fact that logical programming describes to the computer a set of conditions and lets the computer figure out how to satisfy them, it could be argued that this computation model will not be optimal for working with graph-related problems where the programmer often knows what the problem is and how to reach the solution in a number of steps.

4.1.4 The Object-Oriented Paradigm

Finally, one could argue that it would make sense to perceive a graph as an object and that the object-oriented paradigm therefore should be followed. However, there is more to this paradigm. A purely object-oriented approach would be a programming language with native support for inheritance, encapsulation and polymorphism. Without a doubt, all these properties could probably be used with success in DOGS, but conferring the Problem Definition in Section 1.2 yet again, it can be argued that these properties are beyond the scope of this language development. In other words, it could be questioned whether these object-oriented mechanisms will provide indispensible assistance to us or not. At any rate, this will be more clear in Chapter 5 where we present our design criteria. From these, we will be able to make this assessment.

One of the arguments in favor of following the object-oriented paradigm is that all members of the group have relatively much experience in this programming style.

Summary

In this section we have presented and discussed the main programming paradigms. The goal of this is to qualify the group in making a decision about which paradigm to follow when designing the language. This should also be easier when the design criteria will be presented.

Chapter 5

Language Evaluation Criteria

Our brief research of C, Pascal, Java, and Python has shown that none of these languages support graphs in an intuitive fashion. Adhering to the Problem Definition one of the two goals of this project is to develop a programming language that will natively support working with graphs and graph algorithms and further to include a comprehensible and intuitive syntax. This choice reflects a requirement of high readability.

In this chapter we will be concerned with an evaluation of design criteria that applies to our language. It will be based on the research done to this point in the analysis process and form the base of the design.

5.1 Criteria Assessments

Table 5.1 shows the design criteria and how they are rated in relation to one another. The argumentation for the ratings are given from how the criteria are defined (i.e. how we grasp the meaning of them) below.

Criterion	Very important	Important	Less important	Not important
Writability		X		
Readability	X			
Orthogonality		X		
Reliability		X		
Maintainability			X	
Generality		X		
Uniformity		X		
Extensibility			X	
Standardability		X		
Implementability		X		
Efficiency			X	

Table 5.1: Design Criteria for DOGS (the criteria are from [WB00] and [Seb04])

- **Writability:** This criterion describes how easily and quickly the programmer can express algorithms in a language. We have rated it *important* as we want

the user of our language to be able to write all kinds of graph-related algorithms in a concise and intuitive syntax. However, we have decided that readability is more important, and therefore more code to perform a computation is preferred if it can improve readability.

- **Readability:** Readability is a measure of how easy a piece of code in a language is to understand by people other than the original programmer. One of the goals in this project is to design a language with a syntax that leans toward the pseudo-code from [CLRS01]. The purpose of writing pseudo-code is to describe algorithms in an easily understandable way, thus we have rated readability *very important* for our language design. Considering the target audience, readability is also very important since students of algorithms often need to share and discuss their written algorithms and therefore comprehensible code is preferred.

- **Orthogonality:** We have rated orthogonality (that many constructs in the language can be combined in meaningful ways) *important* to support our criterion of designing a language with high writability. However, since readability is most important, our choices regarding orthogonality must not affect readability in a negative way (too much orthogonality reduces the simplicity).

- **Reliability:** This characterizes that the constructs in a language do indeed work as specified in the language definition. It is important in our language design that the language constructs will behave as intended and expected during execution. Therefore, we have rated this criterion *important*.

- **Maintainability:** Our language is not designed for programming industrial applications and therefore maintainability of programs written in DOGS is less important. Hence, we have rated the criterion *less important*.

- **Generality:** A general language provides a few constructs that work in all cases rather than a lot of specialized ones that each works only in a few situations. We have rated generality *important*, as we judge it important to provide general language constructs to cover the programmer's needs. This is preferred as long as the use of these general constructs does not reduce readability of the code, compared to the choice of having several specific constructs instead.

- **Uniformity:** We define uniformity as the degree of similarities between similar constructs in the language. Two constructs with similar purposes should behave in similar ways, thus making the language syntax easier to learn and remember. Since this quality is related to the readability of the language it is *important* to us to address.

- **Extensibility:** This property of a language describes whether the language, by design, is easy to extend by either the original designer or the users. We rated this as *less important* for two reasons, the first being that the language has a well defined problem domain and we believe it to be reasonable that the constructs provided will be adequate. The second reason is that the target audience of our language is students with limited experience in the field of

programming. Such people are unlikely to want to customize and extend their language.

- **Standardability:** The standardability of a language is described as the ease with which programs written in it on one platform can be made to run on other platforms. This is a desirable quality in a language such as DOGS in that we cannot expect our audience to be using a single platform and therefore we rate it *important*, even though it is not something we wish to spend a lot of energy on.

- **Implementability:** The degree of implementability that a language exhibits is described as the ease with which it can be implemented to a given platform. This is obviously *important* to us since the goal is to produce a working compiler for a language. However, we have decided that this concern will not be allowed to compromise the readability concern in our design.

- **Efficiency:** This measures efficency both in terms of running times and memory used. Considering that the target audience is students, it is *less important* whether the programs produced is highly effective as more emphasis is placed on learning and understanding.

Summary

The inspection of how to work with graphs in common programming languages such as C, Perl, Python, and Pascal has revealed room for improvements, thereby conveying a motivation for developing a language and compiler that support working with graphs in a more intuitive way. The basis for this is a pseudo-code directed syntax in the language.

Dijkstra's Algorithm has been introduced as a frame of reference for us to identify elements needed in our language which entailed a survey of the aforementioned four languages. Along with a discussion of which programming paradigm to follow this more thorough inspection of C, Perl, Python, and Pascal has led to a table of language criteria, which will be important in the design of DOGS.

Part II

Design Document

Introduction

The discussion of features in already existing languages in Chapter 3 resulted in our design criteria in Table 5.1, page 34. We will begin the design phase by making decisions based on this examination and argue for the choices made. This will happen in Chapter 6.

The rest of the design phase will focus on designing DOGS. Chapter 7 introduces the syntax of our language along with a description of how to understand the production rules. In Chapter 9 we give an extract of the formal semantics of our language. Before this, however, we discuss the type system of DOGS as it is required in order for us to justify the semantics.

An abstract syntax is introduced in Section 8.4.1 on which the semantics is based, followed by a discussion of our environment-store-model in Section 9.1. Subsequently the semantics is introduced in Chapter 9 in form of a selection of transition rules from different parts of the abstract syntax. The entire semantics can be found in Appendix E.

Chapter 6

Design Choices

The idea of the language analysis from the analysis phase was to qualify us in making reasonable and conscious design decisions. In the following sections we will present our choices and account for these in relation to the analysis as well as to the language design criteria from Section 5. This will document our language syntax which will be presented in the following chapter along with an informal explanation of it, which will prepare for the language semantics.

6.1 Programming Paradigm

The discussion from Section 4.1 concerning which programming paradigm to follow when we design our language insinuates that the imperative paradigm is reasonable to choose for our language. The reasons for designing in this paradigm included already existing programming experience in it for everyone in the group and the fact that the semester course "Programming Languages and Compilers" instructs in language and compiler design in the imperative paradigm. Further, all the graph-related algorithms we have been presented with are imperative which clearly aids our target audience in converting the pseudo-code algorithms from paper to concrete algorithms in our programming language. Any other paradigm would force a "re-designing" of the graph algorithms. Thus, our choice of programming paradigm is the imperative.

As has been stated in the paradigm discussion both the functional and object-oriented paradigm do hold some attraction but using one of these will force the programmer into a different mindset than that of graph-algorithm design. Basically, the choice depends on our language design criteria (cf. Chapter 5). Recall that our highest rated criteria is readability which is a measure of intuitive understanding of the code. This intuitive understanding is based on the imperative pseudo-code from the books.

6.2 Primitive Data Types

In Section 3.1, page 22, we identified the data types in the four languages being analysed. The conclusion was that a range of different primitives was preferable if

efficiency is of the essence. Adhering to our language criteria evaluation, efficiency is not something we have rated high.

If we address readability in relation to data types it would make sense to include only a few in order to keep it simple and intuitive since efficiency is not an issue. We have chosen the types `integer` and `float` to represent numbers, `boolean` to represent truth values (denoted by `true` or `false`), and `string` to represent strings. As regards graphs the types `vertex` and `edge` represent vertices and edges respectively.

Integers and floats are suitable because they make out the range of numbers necessary when working with graph algorithms in particular. Standard arithmetic operations append on both types (i.e. addition, subtraction, multiplication and division) but also modulo and integer division on integers. The two types can enter into the same expression without having a type clash, the responsibility for converting between the types resides in the language (although it is not possible to perform integer division on an integer and float). We will discuss the type system of DOGS in Chapter 8.

In boolean expressions integers are always converted to floats before performing comparisons. Again, the reason for doing this is that it levitates the programmer in expressing pseudo-code from worrying about a strict type system.

We need primitives to represent graph-related types. `vertex` and `edge` make out the foundation when working with graphs, hence they do not apply to arithmetic or boolean algebra. Listing 6.1 illustrates how to create a graph in DOGS.

```
1  program graphConstruction;
2
3  import graphToolkit;
4
5  procedure makeGraph()
6  let
7    graph G;
8    vertex v1; vertex v2; vertex v3;
9    edge e1; edge e2; edge e3;
10 in
11 begin
12   addVertex(G, "a");
13   addVertex(G, "b");
14   addVertex(G, "c");
15   v1 := getVertex(G, "a");
16   v2 := getVertex(G, "b");
17   v3 := getVertex(G, "c");
18   e1 := (v1, v2);
19   e2 := (v1, v3);
20   e3 := (v2, v3);
21   addEdge(G, e1);
22   addEdge(G, e2);
23   addEdge(G, e3);
24   .
25   .
```

```
26 end
```

Listing 6.1: Constructing a graph from vertices and edges in DOGS

In the `let-in` block we declare a graph G followed by the declaration of vertices and edges. A set containing strings are then declared and the standard function (i.e. from the Standard Environment, cf. Section 7.4) `addToSet(graph G, string identifier)` is invoked in order to add three strings to the set.

In the body of the procedure the set V is connected to the graph G. Subsequently, the three declared vertices get connected to G and identified each by one of the strings from the set V using the standard function `getVertex(graph G, string identifier)`. This is followed by the initialization of the edges which are then connected to the graph G by invoking `addEdge(graph G, edge e)`.

6.3 Operators

In Section 3.3 we inspected a range of arithmetic and boolean operators from C, Perl, Python, and Pascal. Judging by the goal of DOGS being able to express a wide range of (graph) algorithms there is a need for a range of both arithmetic and boolean operators in DOGS, and in this section we will present the operators that are to be a part of the DOGS language.

A list of these operators is provided in Table 6.1 along with their respective associativity and their relative precedence.

Operators	Associativity
( )	Left to right
&	Right to left
* / div mod	Left to right
+ -	Left to right
< <= > >=	Left to right
= <>	Left to right
not	Left to right
and	Left to right
or xor	Left to right

Table 6.1: Precedence of operators in DOGS

It may be noted that the precedence rules are mainly based on the rules for C and Perl. The operators in the top of the table have the highest precedence and the operators in the bottom have the lowest precedence.

6.4 Composite Data Types

Consulting the summary in Section 3.1 we have to decide upon the need for static sequence structures (arrays, that is) and dynamic sequence structures (e.g. sets). Of course, considerations can be based on efficiency but in our case it is relevant to consider the need from a graph-point-of-view and from a more general programming view. Basically, both arrays and sets provide a convenient way to keep a collection

of data for different purposes, for instance to iterate over. We have included both types of sequence structures.

Sets allow for a high-level abstraction on the operations on sequences, thus fulfilling our readability criterion. Although dynamic, they are limited to hold data of one type only. This is a choice we have made based on the fact that mathematical sets contain a certain type and we have judged this restriction meaningful in regards to graphs. Further, sets only allow for one instance of each value (i.e. no sequence with a representation of a value, say, 5 or vertex v_1 more than once) which also makes sense because they primarily will be used to collect vertices and edges (cf. Listing 6.1 in the previous section), the former will typically be used to iterate over:

```
foreach vertex in V do ... //V is a set
```

To allow for more advanced constructs we will include records with the `record` construct. The record types are constructed from a number of fields of other types (which can be both primitive types, graphs and other records, but not functions or procedures). Together with the primitive types we judge that these types suffice in that the selection of primitive types is easily comprehensible and as such facilitates readable DOGS code, whereas the option to use structures gives programmers increased power in both expressibility and abstraction. Together with reference types (which will be described shortly) the inclusion of structures also allows for the construction of abstract data structures such as queues, trees, and lists.

A record can be declared in the following way:

```
1  program records;
2
3  record rec integer x := 0,
4             boolean b := false,
5             float f := 0.0;
6
7
8  procedure makeRecord()
9  let
10   rec r := {x := 2, b := true};
11 in
12 begin
13
14 end
```

Listing 6.2: Records in DOGS

Note that records are declared outside a `let-in` block. This is reasonable because records represent types that may need to be globally accessible to the programmer and it increases clarity and readability to restrict the declaration of user-defined types in the global scope. Instances of already declared record types can then be declared in a local scope (cf. Section 6.6 for declaration of variables and constants and Section 6.9 for scope rules).

6.5 Variable References

In addition to the types mentioned in the previous sections, DOGS will also support reference variables, however only as parameters to functions and procedures. The reason for this limitation is that we find the use of pointers only in the call-by-reference mechanism the most safe and intuitive way to include them. For instance, if we pass a reference a to a procedure and this reference is assigned to a variable b in the body of the procedure, how should we then decide what a refers to? The value of the variable b, or will the value of b be stored in whatever a was referring to before the assignment?

Again, if we confer the readability criterion we want to let procedures and functions work the same way every time used. This is done by explicitely noting in the procedure- or function declaration block that a variable is just a reference to the variable used when calling, which is denoted by the `ref` keyword. This is in contrast to C where the programmer also has to note the parameter as a pointer, e.g. when calling a function. This way there will be no difference in the way a procedure or function is working on the given variables from time to time, hence welcoming our readability criterion.

6.6 Declaration of Variables and Constants

Before using a variable in DOGS it has to be declared in a block, just before the body of any function or procedure. The idea of having a block where declarations are collected originates from pseudo-code where declarations are somewhat secondary and almost never presented. In order to make the code resemble pseudo-code we have decided to force the programmer to declare variables in a `let-in` block similar to Pascal's and Triangle's [WB00].

```
1  program variableDec;
2
3  procedure emptyProc()
4  let
5    boolean listening = true;
6    integer x := 0
7    float y := 0.0;
8  in
9  begin
10
11 end
```

Listing 6.3: Variable declarations in DOGS

Another reason for making this the only place to declare variables is to make it easy to get an overview of those variables available inside the function or procedure. Thus, it is not possible to declare variables inside any nested block inside a function or procedure body because this would make it harder to get an accumulative survey of all the variables.

In some ways constants are similar to variables, although it is not possible to change their value once set. Furthermore, the constants are not limited to be declared just for a function, but can be declared in every file for use in every function. This could be used advantageously for a package with mathematical constants, for instance. We use the keyword `constant` to denote a constant in DOGS. They can be applied globally by declaring them outside the `let-in` blocks in the top of the file or locally inside a `let-in` block:

```
1  program constantDec;
2
3  constant int speedOfLight := 299792458;
4  constant string homerSimpson := ``doh'';
5
6  procedure emptyProc(int argNumber)
7  let
8    constant float pi = 3.14159;
9  in
10 begin
11
12 end
```

Listing 6.4: Constant declarations in DOGS

6.7 Control Structures

In Section 3.4 we considered three kinds of control structures: recursion, branching, and iteration. We have decided that the exclusion of any of those would severely hamper the writability of programs written in DOGS. Also, many graph related algrorithms yield for a recursive solution. Following this argument we have opted to include recursion in our language as a means to make functions able to call themselves. This will be discussed in Section 6.8.

In regards to branching we have chosen to include the common `if...then...else...` construct. This construct is a part of each of the languages we have examined and we find it easily understandable. Hence, this choice is in accordance with our goal of readable programs. As for iteration we have selected six constructs, two of them similar to C's:

- `do...while...`
- `while...do...`

These two constructs will execute a command while a condition is true, the difference being that the first construct will repeat the command at least once whereas the second will not execute the command if the condition is initially false. The next two merely provide a simple way to execute some code a given number of times. Although we have chosen the `for` keyword to represent this structure, it is quite different from the `for` construct in C. Whereas C's construct is very similar to the

`while` construct we have decided to make it somewhat simpler by just making the code execute a number of times while increasing or decreasing a variable, instead of making the stop condition dependent of the evaluation of a boolean expression. The constructs look like this:

- `for...to...do....`
- `for...downto...do....`

The last two constructs work rather different from what is commonly used in languages like C and Java, but are similar to constructs in Python and Perl:

- `foreach...in...do....`
- `foreach...in...where...do....`

This type of construct is made for iterations over sets of a certain type, where the regular `for`-loop would be inconvenient to use, thus seeking to fulfill our goal of high readability. The first variant of this loop iterates over all elements of the set and makes them available to the programmer. The second only makes those elements available where a condition is true. When working with vertices and edges from graphs, these will be made available through sets and therefore the `foreach`-loop will be very useful. Furthermore, this loop-construct will be the only way to iterate through a set or array of elements.

It could be argued that the choice of six different loop-constructs violates our relatively high rating of generality because the programmer is limited in the choice of loop for a certain purpose. Ultimately, however, the goal is to ease a conversion from pseudo-code algorithms on paper to a running program in the way that the code should reflect and support pseudo-code expressions. We could end up straying from this objective if we were to choose fewer and more flexible ways to solve different tasks (as an example, confer Section 1.1 on page 13 for the way a program in C iterates over every vertex in a graph) and thereby violating the goal of readability. In other words, we have argued that orthogonality should be of relatively high importance in our language as long as it does not affect readability by reducing the simplicity, which could be the result of fewer and more flexible loops.

6.8 Functions and Procedures

Functions and procedures are needed when a subroutine needs to be invoked from a main program. They provide code abstraction and thereby increase the readability of the program.

In DOGS we differentiate from functions and procedures. A function is a subroutine that returns a value, whereas a procedure causes side-effects and has no return value. They are further separated by keywords, a function is declared using the `function` keyword followed by the type of the return value and a given number of arguments, and a procedure is declared using the `procedure` keyword followed by a given number of arguments. In Listing 6.5 we have exemplified how to write a (meaningless!) function in DOGS.

```
1 program DoNothing;
2
3 function integer returnParam(integer argNumber)
4 let
5
6 in
7 begin
8   return argNumber;
9 end
```

Listing 6.5: An meaningless function written in DOGS

As mentioned earlier, our language should support recursion, i.e. the possibility for a function or procedure to invoke itself. Some graph algorithms use this exensively, for instance the Merge Sort algorithm (described in [CLRS01], page 28-36) which sorts a sequence of elements (e.g. an array of integers) by the divide-and-conquer approach. It divides the problem of sorting the sequence into subproblems which it then solves recursively and combines the solutions. The Merge Sort algorithm is presented in DOGS code in Listing 6.6.

```
1  program MergeSort;
2
3  procedure mergeSort(array ref A, integer p, integer r)
4  let
5    integer q;
6  in
7  begin
8    if p < r then
9          q := (p + r) div 2;
10
11         mergeSort(A, p, q);
12         mergeSort(A, q + 1, r);
13         merge(A, p, q, r);
14 end
```

Listing 6.6: The Merge Sort algorithm written in DOGS

The `merge` algorithm does all the calculation and number permutation but is not relevant in this context. What is relevant is that `mergeSort` invokes itself twice, once on each of the left and right split of the sequence.

Recursion increases readability to some extent; it is somewhat intuitive to grasp that due to some condition an algorithm keeps calling itself to perform the exact same operations on changed data, but when trying to really understand recursion it is far more complex. However, excluding recursion from our language would severely limit the programmer in writing algorithms in our language from pseudo-code examples, due to the fact that many graph algorithms (and algorithms in general) use it.

6.9 Scope Rules

Having considered the various options on scope rules (as descriped in [Seb04, ch. 5]) the decision has been made that the DOGS language is to be statically scoped, i.e. variables, functions, and procedures are statically bound. This choice is made since readability of statically scoped code is better than that of dynamically scoped. This is supported by [Seb04]:

> Programs in static-scoped languages are easier to read, more reliable, and execute faster than equivalent programs in dynamic-scoped languages. [Seb04, p. 219]

However, [Seb04] also states that nested subprograms can cause problems in a statically scoped language. This is not an issue in DOGS, however, since nested declarations of procedures and functions are not allowed. The scope rules in DOGS dictate that in the global scope only functions, procedures, global constants, and record types can be declared. Variable declarations (of both simple types and record types) take place in local scopes (defined by the `let-in` construct) inside the declaration of a procedure or function. The variables and constants declared inside these blocks are available only within that local scope, whereas the identifiers declared in the global scope are available everywhere in the program. In effect, this means that constants, record types, procedures, and functions are global within a specific program, even if shared across several files (packages). Parameters to functions and procedures are the only way to transfer data between different local scopes, with the `ref`-keyword allowing for call-by-reference as previously described. In the interest of readable and comprehensible code we have chosen that labels, weights and graphs is always called by reference. This is done for two reasons: One is that references would be the most common way to employ graphs, labels and weights in parameters (resulting in a lot of ref-keywords, thus decreasing readability). The other reason is more sinister: if a label connected to a graph was passed as a parameter together with the graph, to a function (or procedure). What would happen is that they would both be copied to new locations in memory. This would result in the very bizare situation that the label passed to a function would in fact still be connected to the graph outside the function, and *not* to the graph that was passed along with it.

6.10 Error handling

As described in Section 3.7, page 28, there are different ways of handling errors in a programming language. Bearing in mind that the main goal of our language is to make graph-related problems easy to solve, it could be argued that the need for specific error handling is less important. Unlike companies that develop heavy applications it is not crucial for a person of our target audience to rely on a program that is "fail-safe", hence justifying our choice of excluding error handling as a part of DOGS. This means that it will not be possible to throw an error from inside the program by the programmer and it will not be possible to catch exceptions, they will simply halt the running program.

Some errors will be caught during compilation, e.g. type errors (the type system in DOGS will be discussed in Chapter 8, and some will be possible to catch during runtime only, e.g. division by zero. Of course, as to errors that can be caught during compilation it is our responsibility as compiler-designers to design a somewhat helpful compiler that will aid the programmer in correcting these errors, for instance by pointing out the lines in which the errors reside.

6.11 Concurrency

Our discussion in Section 3.6, page 27, of the need for concurrency in DOGS concluded it to be a fundamental part in a programming language. We have, however, decided not to include it in DOGS.

True, as to different graph problems such as the Travelling Salesman Problem it does make sense to let the language support concurrency. This is indeed a powerful way to solve resource demanding tasks. However, in our case of language it is important to keep in mind our target audience and their programming needs, for instance the students following the Algorithms and Data Structures course. Their need is limited to a programming language in which common pseudo-code graph algorithms can be expressed and tested and as such we do not intend to develop a heavy-application-oriented programming language, hence making concurrency minor important.

Another argument for including concurrency is the feasibility of waiting for input from the user while concurrent threads perform tasks in the background. In general this is powerful, but as regards graph-specific tasks it could be argumented that it would make less sense. A typical situation where input from the user is relevant could be that the program needs to unite a graph from an unknown media or source (e.g. from a floppy or alternative disk location) with one it is currently working on. In this situation the program is dependant on the input and cannot continue before action has been taken from the user.

Consider the Merge Sort Algorithm from Listing 6.6 in Section 6.8. This algorithm (and others that use the divide-and-conquer approach) could possibly be implemented using multiple threads to work on different subproblems. In this case we simply judge it to be more readable to use recursion to solve the task.

If simple threading were to be introduced in DOGS one way to do so would be to allow threads to be declared in the same way as procedures (using a different keyword) with parameters, some of which could be reference parameters to share data between the parallel threads. These threads would be started in a manner resembling usual procedure calls, except that the program would proceed to the next command and the call would return an integer used to identify the thread to procedures such as `sleep()`, `kill()`, and `awake()`.

It is inherently more difficult to understand and code programs that use threads and if we decide to focus on making our language support concurrency we can end up contradicting ourselves in the choice of high readability.

6.12 Input and Output

In the analysis we assessed it necessary to include input/output functionality in our language (Section 3.5, page 27) and consequently DOGS will support it. Listing 6.7 exemplifies how to read from a file and write it to another file.

```
1 program inputOutput;
2
3 procedure main(array of string args)
4 begin
5   appendToFile(``output.text'', readFile(``dogsFile.dogs''))
        ;
6 end
```

Listing 6.7: Input/output in DOGS

`appendToFile` takes two string arguments; the first being the file to which it should append the second string. In the example the second argument is the return value of the `readFile` standard function (found in the Standard Environment, cf. Section 7.4) which takes the string name of an existing file and returns the content of the file in form of a string.

As stated in Section 3.5 input/output becomes relevant for DOGS in relation to persisting results from graphs. For instance, one could run Dijkstra's Algorithm with different arguments from the command line and append the results to a file. That way the data is at one's disposal for analysis. Another example is parsing a file name from the command line and let the algorithm in question read a graph from this file.

6.13 Graphs in DOGS

In the language we obviously need some way to represent graphs. In the analysis we have examined different ways to store graphs, here we investigate which parts should be supported in the syntax and which should be made as functions or procedures not written in DOGS. First, however, it is necessary to review the types of graphs directly supported by DOGS.

The decision has been made to support only two types: Directed graphs (using the keyword `digraph`) and undirected graphs (with the `graph` keyword) and not any variant of multigraphs. This decision was motivated primarily by the jugdement that the added complexity (in both design, semantics, and implementation) would not correspond to the functionality that multigraphs represent. Another worthwile consideration would be that most multigraphs can be emulated with ordinary graphs. In the case of multigraphs without weights, one could employ a weight function to indicate the number of edges, and in most cases even multigraphs with weights can be converted to ordinary graphs that the language supports.

6.13.1 Vertices and Edges

Listing 6.1 in Section 6.2 illustrates an example of how to construct a graph in our language. The example clearly demonstrates that we have decided to mix between

pre-defined, standard functions from a standard package and DOGS syntax to represent graphs in the language. The main reason for this choice is that this mixture resembles pseudo-code, thus complies with the readability criterion.

Vertices are represented as strings in a specific graph. This is seen in `addVertices(G, V)` where `V` is a set of strings. "Outside" the graph, a vertex variable contains nothing but a reference to this string: `v1 := getVertex(G, ‘‘a’’)`.

An edge connects two vertices by using references to them. This is where the necessity of the `getVertex(graph G, string identifier)` function is clear. Without a way to access the vertics via references we cannot work with edges. The `edge` primitive contains information on the location of the two vertices it connects to and once the vertex variables have been assigned references to their respective string representations in the graph, an edge can be assigned to a pair of these references using DOGS syntax. However, similar to vertices, edges are connected to a graph using a pre-defined function that is responsible for the underlying edge representation in the graph.

6.13.2 Labels and Weights

Conferring the example of Dijkstra's Algorithm in Section 2.2, page 19, labels and weights are set up using the syntax `weight/label in G of type w` where `type` can be integer or string, for instance, and `w` is the name of the label/weight function. An important argument about this way of labelling or weighing a graph is that the specific label/weight function is associated to a certain graph. This means that if a vertex is removed from a graph it will consequently no longer be possible to access the associated label (similar with edges).

It could be argued that since a graph and its associated label and weight functions are so strongly connected, why not just let these informations reside in the graph variable? We have decided not to do so because we want to reflect which weights and labels are being passed as parameters in the algorithms. This increases readability. Otherwise, the reader would not be able to see whether a graph passed to an algorithm is weighted/labelled or not, how many weights/labels of what type exists in the graph, and so forth.

Summary

The design decisions in this chapter concludes the survey of the four languages C, Perl, Python, and Pascal from the analysis. Our choices are based on the design criteria from Table 5.1 in Section 5.1, page 34, and it should be noted that the reason the readability criterion dominates the decisions is the goal of making an implementation of pseudo-code algorithms a relatively intuitive process without to much ado, thus applying the pseudo-code style from [CLRS01] to DOGS' syntax.

Our code examples in DOGS have illustrated a part of the DOGS syntax. In the following chapter we present the appearance of our language, i.e. its syntax, the classification of this, and its type system. Suffice it to say that our choices are influenced by each of the aforementioned languages, however our syntax is strongly affected by Pascal's. This comes as no surprise since Pascal is a language originally made for educational purposes [Seb04, page 80] with a rather intuitive syntax.

Chapter 7

The DOGS Language

Having classified important decisions we will now turn to the basic elements of the DOGS language, i.e. the syntax. The purpose is to describe how programs may be written, and to some extent how they are intended to be written. Initially, we will describe some preliminary considerations that were resolved during the development of the DOGS grammar, followed by the exact syntax presented as a grammar written in Backus-Naur Form. In part from the development of the grammar of DOGS , a standard environment has been derived, which contains functions, procedures, and types that are a part of the language but cannot be written in DOGS. Within these limits of the language, Dijkstras algorithm will be presented DOGS code in order to illustrate basic parts of the DOGS grammar.

7.1 Syntax Considerations

The DOGS grammar is developed based on the design choices (cf. Section 6), keeping the actual implementation in mind. Due to the latter, some specific steps had to be taken in order to make the grammar "implementation friendly" which basically means that steps were taken so that an implementation of DOGS would not require any specific modifications to be applied to the compiler. The steps included resolving an ambiguity that, to some extent, could have been ignored, and embedding rules of precedence for different expressions into the grammar. These steps are described in the following sections.

7.1.1 The "Dangling Else" Problem

A common problem in programming languages is the so-called "Danglig Else" problem [App99]. Consider the following sample grammar (note that the '. . . ' represent other commands in the grammar):

```
<Command>      ::=  If-Command
                  |  ...

<If-Command>   ::=  if Expression then Command
                  |  if Expression then Command else Command
```

This is a rather standard way of expressing syntax for if-else commands [Sof].

However, this is an ambiguous grammar and the ambiguity is best illustrated with an example:

Example 1. *`if` Expression$_1$ `then` `if` Expression$_2$ `then` Command$_1$ `else` Command$_2$*

The example can be interpreted in two ways; either with `if` Expression$_2$ `then` Command$_1$ `else` Commmand$_2$ as the Command in an If-Then Command, or with `if` Expression$_2$ `then` Command$_1$ as the first Command in an If-Then-Else Command.

In order to overcome this ambiguity parsers have traditionally been modified to accept the first of the two interpretations [Sof], i.e. an interpretation that matches an 'else' clause with the nearest previous unmatched 'if'. This is also the interpretation that DOGS will apply. However, an implementation of DOGS should not rely on a modification of the parser, and as such this ambiguity should be resolved and removed from the grammar.

The basic idea in resolving the dangling-else ambiguity is to split the If-Commands into two categories; open If-Commands, that contain at least one If-Command with no 'else' clause, and closed If-Commands, that does not contain an If-Command unless it is followed by an 'else' clause [Seb04]. According to this we can revise the sample grammar:

<Command> ::= Open-Command
| Closed-Command

<Open-Command> ::= `if` Expression `then` Command
| `if` Expression `then` Closed-Command `else` Open-Command

<Closed-Command> ::= `if` Expression `then` Closed-Command `else` Closed-Command
| ...

A similar approach was adopted in the transformation of the DOGS grammar, the result of which is visible in Section 7.2.4

7.1.2 Precedence Rules

In Section 6.3 we presented the operators of DOGS . These operators are part of the DOGS language and therefore a part of the syntax. One possible approach to embed them into the syntax would be to let them appear in a single production. However, this would not enable a compiler to separate the different rules from each other, as no precedence information would be available. Not unlike the problem of the "dangling else", this could be overcome by embedding these rules in the compiler itself [Inc]. However, as stated in the previous section, a DOGS implementation should not rely on certain specific modifications of a compiler and it is therefore desirable to embed both the operators and the precedence rules into the syntax. A way to embed precedence rules of operators along with associativity rules in a grammar is given in [Seb04]:

- Precedence Rules:
  If operator `opA`, has higher precedence than `opB` this can be implemented by

the sample grammar:

```
<Expression>        ::=  opB-Expression

<opB-Expression>    ::=  opA-Expression
                       |  opA-Expression opB opB-Expression

<opA-Expression>    ::=  Some-Expression
                       |  Some-Expression opA opA-Expression
```

It is clear that any Expression in the above grammar must evaluate any expression with operator `opA` before an expression with operator `opB`, as the former is part of the latter. This principle may be applied to any number of operators, creating new productions for each level of precedence.

- Associativity Rules:
  These rules are applied in extension to the precedence rules. Consider the grammar used to illustrate the precedence rules, but this time `opA` has left to right associativity, and `opB` has right to left associativity. This can be implemented by the sample grammar:

```
<Expression>        ::=  opB-Expression

<opB-Expression>    ::=  opA-Expression
                       |  opA-Expression opB opB-Expression

<opA-Expression>    ::=  Some-Expression
                       |  opA-Expression opA Some-Expression
                       ...
```

  Basically, this illustrates that right to left associativity can be implemented using right recursion and vise versa [Seb04].

This same approach was applied to the operator expressions in the DOGS grammar, the result of which is described in Section 7.2.3.

7.2 DOGS Syntax in BNF

In our language evaluation criteria we have rated readability as very important. As mentioned, this choice, among others, has an impact on how we choose to design the syntax of our programming language. In this chapter we will present our syntax and informally describe how it is to be understood in order to ease the understanding of the formal semantics in Chapter 9.

The syntax of DOGS is somewhat inspired by the syntax of the Triangle language, which is found in Appendix B in [WB00]. We will present the grammar in BNF in the following sections and present the informal semantics and contextual constraints related to the different parts of the language. It should be noted that we use regular expressions such as the $*$-operator to express concatenation when presenting our lexicon (i.e. it is presented in EBNF).

The presentation of the different parts of the language is done in a bottom-up approach in the sense that the actual program part, consisting of a number of

subparts, is the top-most part of the language and as such will be presented last. However, the lexicon, consisting mainly of the alphabet of our language as well as a few very basic production rules, is considered to be inherently different from the other parts of the language and therefore it is presented at the end of this chapter.

7.2.1 V-Names

In any programming language it is necessary to be able to uniquely identify constant values and variables. In DOGS, this is done through the V-name production.

```
1  <V-Name>   ::=  Identifier
2                |  Identifier . V-Name
3                |  Identifier [Expression]
```

1. The simple V-Name *Identifier*, I, identifies the value or variable bound to I. The type of the V-name is the type of that value or variable.

2. The V-Name *I.V-Name*, $I.V$, identifies the field of a record value or variable identified by I. I must be of type `record` and contain a field V.

3. The indexed V-Name *I[Expression]*, $I[E]$, identifies the component of array I located at the index yielded by the evaluation of E. If no such index exists in the array, the index is said to be out of bounce and the program will fail. The evaluation of E must yield an integer value, I must be an array of a given type, and the component at the given index must be of that same type.

7.2.2 Expressions

Expressions are usually statements from which we can retrieve a value. In DOGS, this includes a number of different statements ranging from an edge-construct to the name of a variable.

An expression is evaluated to start the identification of the different elements of the expression. This is done by evaluating through the different operator productions until a primary-expression is identified. A primary-expression is evaluated to yield either a value or a new expression. A comma-expression is evaluated to construct either an array value or record value.

1. The Expression *Assign-Expr*, *Ae*, is evaluated as an *Ae* ensuring that any sub-expressions in E will be evaluated in the right order and eventually yielding a Primary-Expression. This is further explained in Section 7.2.3.

2. The Primary-Expression *Integer-Literal*, *Il* is evaluated to the integer value of *Il*. This is a number expression.

3. The Primary-Expression *String-Literal*, *Sl*, is evaluated to the string value of *Sl*. This is a string expression.

```
1  <Expression>            ::=  Assign-Expr
2  <Primary-Expression>    ::=  Integer-Literal
3                           |    String-Literal
4                           |    Boolean-Literal
5                           |    Float-Literal
6                           |    Infty
7                           |    V-Name
8                           |    Parameter-Call
9                           |    (Identifier, Identifier)
10                          |    (Expression)
11                          |    {Expression Comma-Expression}
12 <Comma-Expression>       ::=  ϵ
13                          |    ,Expression Comma-Expression
```

4. The Primary-Expression *Boolean-Literal*, Bl, is evaluated to the boolean value of Bl. This is a boolean expression.

5. The Primary-Expression *Float-Literal*, Fl, is evaluated to the float value of Fl. This is a number expression.

6. The Primary-Expression *Infty*, *If*, is evaluated as an infinite value and may be compared, or assigned, to variables of either integer type or float type. Other operations, such as multiplication, are not applicable for *If*. This is a number expression.

7. The Primary-Expression *V-name*, Vn, is evaluated to the value identified by Vn, or the value of the variable identified by Vn (when Vn is a reference to another variable).

8. The Primary-Expression *(Identifier, identifier)*, (I_1, I_2), yields the edge that connects the vertices identified by I_1 and I_2. The expression is of type `edge` and I_1 and I_2 must both be of type `vertex`.

9. The Primary-Expression *(Expression)*, (E), is part of the "precedence cycle" and is a parenthesized expression. E is then sent through the "precedence-cycle" once more.

10. The Primary-Expression {*Expression Comma-Expression* }, EcE, may be used to assign values to an array or a record - note that when evaluating cE it may yield zero or more expressions. In case of an array, each following expression will be assigned to a component at the next index of the array, assigning the value of the first expression to index 1. All expressions must evaluate to a value of same type as the array. If there are more expressions than indeces in the array, or if there is a type mismatch, the program will fail.

    In case of a record, each expression must be a specific assignment of an expression to a field contained in the record.

11. The Comma-Expression ϵ, is the empty string and is, as such, not evaluated.

12. The Comma-Expression *, Expression Comma-Expression*, $,EcE$, evaluates as a comma separated list of arbitrarily many expressions (minimum 1).

7.2.3 Precedence Rules

When working with an expression that may be divided into several subexpressions, it is important to know how to make such a division. This is something that is determined by rules of precedence (see Section 6.3) which are build into the syntax of DOGS by creating different productions for each level of precedence. Each of these productions is evaluated to yield expressions containing operators of higher precedence, until a primary-expression is reached (cf. Section 7.1.2).

```
1  <Assign-Expr>           ::=  Or-Expr
2                           |   Or-Expr := Assign-Expr
3  <Or-Expr>               ::=  And-Expr
4                           |   Or-Expr or And-Expr
5                           |   Or-Expr xor And-Expr
6  <And-Expr>              ::=  Not-Expr
7                           |   And-Expr and Not-Expr
8  <Not-Expr>              ::=  Compare-Expr
9                           |   not Not-Expr
10 <Compare-Expr>          ::=  Less-Greater-Expr
11                          |   Compare-Expr = Less-Greater-Expr
12                          |   Compare-Expr <> Less-Greater-Expr
13 <Less-Greater-Expr>     ::=  Plus-Minus-Expr
14                          |   Less-Greater-Expr < Plus-Minus-Expr
15                          |   Less-Greater-Expr > Plus-Minus-Expr
16                          |   Less-Greater-Expr <= Plus-Minus-Expr
17                          |   Less-Greater-Expr >= Plus-Minus-Expr
18 <Plus-Minus-Expr>       ::=  Multiplication-Expr
19                          |   Plus-Minus-Expr + Multiplication-Expr
20                          |   Plus-Minus-Expr - Multiplication-Expr
21                          |   + Multiplication-Expr
22                          |   - Multiplication-Expr
23 <Multiplication-Expr>   ::=  Concatenation-Expr
24                          |   Multiplication-Expr * Concatenation-Expr
25                          |   Multiplication-Expr / Concatenation-Expr
26                          |   Multiplication-Expr div Concatenation-Expr
27                          |   Multiplication-Expr % Concatenation-Expr
28 <Concatenation-Expr>    ::=  Primary-Expression
29                          |   Primary-Expression & Concatenation-Expr;
```

Generally, each production involved with precedence rules may evaluate to the following production, or itself, an operator, and the following production. Only the last production, *Concatenation-Expr*, may evaluate directly to a primary expression.

1 The Assign-Expr may evaluate to an *Or-Expr*, or to *Or-Expr ::= Assign-Expr*, $oE ::= aE$,

3 The Or-Expr yields *true* if either either side of the expression yields *true*, otherwise it yields *false* (i.e. both sides of the operator must be boolean expressions). This is a boolean expression.

6 The And-Expr is evaluated as *true* if both sides of the *And*-operator yields

a truth value (i.e. both sides of the operator must be boolean expressions). Otherwise, the expression yields *false*. This is a boolean expression.

8 The Not-Expr negates a boolean value (i.e. a boolean expression that yields *false* using the *Not*-operator will yield *true*). The expression following the *Not*-operator must be a boolean expression, i.e. an expression that may evaluate to a truth value. This is a boolean expression.

10 A Compare-Expr determines if two expressions evaluate to the same value (*true*) or not (*false*). Expressions on both sides of the operator must be of the same type, otherwise the program will fail. This is a boolean expression.

13 A Less-Greater-Expr compares the values of two expressions and yields a boolean value depending on the operators and the expressions on either side. Expressions on both sides of the operator must be of either a number type (i.e. integer or float) or a string type.

18 A Plus-Minus-Expr evaluates to a number value and as such both sides of the expression must be number expressions and of the same type, otherwise the program will fail. The expressions are then evaluated according to the rules of arithmetic. This is a number expression.

23 A Multiplication-Expr evaluates to a number value and as such both sides of the expression must be number expressions and of the same type. Expressions are evaluated according to the rules of arithmetic. This is a number expression.

28 The Concatenation-Expr yields a string, and as such both sides of the expression must yield strings. The string yielded by this expression consists of the string on the left side of the expression immediately followed by the string on the right side. This is a string expression.

7.2.4 Commands

Commands are statements that update values of variables and otherwise control the order of execution in a program. Therefore, commands consist of all control structures as well as Basic-Commands, allowing new values to be assigned to variables. The syntax for writing commands in DOGS is divided into two main categories; open and closed commands. This is done entirely to ensure that the DOGS grammar is unambiguous (cf. Section 7.1.1) and therefore both types of commands are quite similar, which is why we will only describe the semantics for the general "if" statements, the different loop constructs, and our Basic-Commands from the closed and open commands. Furthermore, the term: "Single-Command" will be used in cases where two productions differentiate only in whether they are closed or open commands.

```
1  <Command>          ::= let Declaration in begin Multi-Command end
2                       | begin Multi-Command end
3  <Multi-Command>    ::= ϵ
4                       | Single-Command Multi-Command
5  <Single-Command>   ::= Open-Command
6                       | Closed-Command
7  <Open-Command>     ::= if Expression then Single-Command
8                       | if Expression then Closed-Command else Open-
                          Command
9                       | Loop-Headers Open-Command
10                      | do Open-Command while Expression
11 <Closed-Command>   ::= Basic-Commands
12                      | if Expression then Closed-Command else Closed-
                          Command
13                      | Loop-Headers Closed-Command
14                      | do Closed-Command while Expression
15 <Loop-Headers>     ::= while Expression do
16                      | for V-Name := Expression (to | downto) Expres-
                          sion do
17                      | foreach Single-Declaration in V-Name do
18                      | foreach Single-Declaration in V-Name where Ex-
                          pression do
19 <Basic-Commands>   ::= V-Name := Expression;
20                      | Parameter-Call ;
21                      | Parameter-Call := Expression ;
22                      | begin Multi-Command end
23                      | switch V-Name Case-Item Default-Item endswitch
24                      | break;
25                      | return Expression;
26                      | V-Name ++;
27                      | V-Name --;
28 <Case-Item>         ::= ϵ
29                      | case Integer-Literal : Single-Command Case-Item
30                      | case Integer-Literal .. Integer-Literal : Single-
                          Command Case-Item
31 <Default-Item>      ::= ϵ
32                      ::= default: Single-Command
33 <Parameter-Call>    ::= Identifier ( Actual-Parameter-Sequence )
```

The basic construction of the commands has been devised to avoid ambiguities in the language (to avoid the "dangling else" problem).

1 The Command production ensures that each function or procedure will be declared using either a *let-in begin-end* construction (if variables are used) or just a *begin-end* block (if no variables are used). In case of the first construction the declaration part of the *let-in* block is elaborated before the command part of the *begin-end* block is executed. In the other case the command part is simply executed.

3 The Multi-Command may evaluate to arbitrarily many Closed-Commands and Open-Commands (through the Single-Command production). The execution order of the commands is from left to right.

7 The Open-Command *if Expression then Single-Command*, *if E then sC*, executes *sC* iff[1] *E* evaluates to *true*. *E* must be a boolean expression.

8, 12 The Command *if E then Closed-Command else Single-Command*, *if E then cC else sC*, executes *cC* iff *E* evaluates to *true*, otherwise it executes *sC*. *E* must be a boolean expression.

10, 14 The Command *do sC while E* is a loop construct that forces *sC* to be executed at least once. When execution of *sC* has finished, it is executed again iff (and as long as) *E* evaluates to *true*. *E* must be a boolean expression.

15 The Command *while E do sC* executes *sC* iff *E* evaluates to *true*. When the execution of *sC* has finished, *E* is evaluated again, and only if it evaluates to *true* is *sC* executed again. This continues until *E* evaluates to *false*. *E* must be a boolean expression.

16 The Command *for vN* := E_1 (*to*|*downto*) E_2 *do sC* updates the variable identified by *vN* with the value of E_1, then the value of *vN* is compared to the value of E_2. Iff this comparison yields *true*, *sC* is executed. When the execution finishes, the value of the variable identified by *vN* is decremented, or incremented, by 1, according to whether the Command is written with *downto* or with *to*. Subsequently, this value is compared to the value of E_2. Iff the comparison yields *true sC* is executed. This cycle is iterated until the value of the *vN* is, in case of *downto*, less than the value of E_2 or, in case of *to*, greater than the value of E_2. Both E_1 and E_2 must be number expressions and the variable identified by *vN* must accordingly be of the same type.

17, 18 The Command *foreach Single-Declaration in vN where E do*, *foreach sD in vN where E do*, iterates through each item of same type as given by *sD* contained in the set identified by *vN*. In case the *where E* clause is applied, an item contained in a set is part of the iteration iff *E* evaluates to *true*. *sD* must not be assigned anything as part of the declaration, *vN* must identify a collection, being a set, an array, a graph, a weight function, a label function, or a record. Finally, *E* must be a boolean expression.

19 The Basic-Command *vN := E;* updates the variable identified by *vN* with the value of *E*. Both *vN* and the value yielded by *E* must be of the same type.

21 In this context the value yielded by *pC* is updated with the value yielded by *E*. However, *pC* must be either a weight or a label function and *E* must evaluate to the same type as the declared label or weight function.

22 The Basic-Command *begin Multi-Command end* is executed simply by executing the Multi-Command.

23 The Basic-Command *switch vN Case-Item Default-Item*, *switch vN cI dI* is a construct that tries to match the value or the value of the variable identified by *vN* with a *cI*. Iff this is not successful, *dI* is entered. The value of the variable identified by *vN* must be of type integer.

[1] if and only if

24 The Basic-Command *break;* may be used only inside loop structures in which it has the effect of breaking out of the loop in question.

25 The Basic-Command *return E;* must be present within the body of a declaration of a function. It indicates that the function is executed and is ready to return with the value yielded by E to the callee. The type of the value yielded by E and the return type of the function must be the same.

26, 27 The Basic-Command vN`++` or vN`--` increments or decrements the value identified by vN by 1 respectively. The value identified by vN must be of a number type.

28 The Case-Item may consist of arbitrarily many case constructs. A value is matched with a case and the following sC is executed iff the value is the same as that of the case (which is given by the *Integer-Literal* in line 29), or if the value is contained within the range of the case (which is given by the *Integer-Literal .. Integer-Literal* in line 30).

31 The Default-Item may, or may not, contain an actual default case construction *default: sC*. A default case, if reached, is executed by simply executing sC.

33 The Parameter-Call, *I(Actual-Parameter-Sequence)*, $I(APS)$, is evaluated as follows: The APS is evaluated to yield an argument list; then the function bound to I is called with that argument list. (I must be bound to a function. APS must be compatible with that function's formal-parameter-sequence.)

7.2.5 Parameters

Functions and procedures can both have parameters and a way to parameterize the two is to use the formal-parameter-sequence. The formal-parameter-sequence allows procedures and functions to have arbitrarily many parameters. The actual-parameter-sequence is used when calling a parameterized function or procedure and allows the use of expressions as input.

```
1  <Formal-Parameter-Sequence>    ::=  ϵ
2                                 |    Type-Denoter (ref | ϵ) Identifier
                                       Comma-Formal-Parameter
3  <Comma-Formal-Parameter>       ::=  ϵ
4                                 |    ,Type-Denoter (ref | ϵ) Identifier
5  <Actual-Parameter-Sequence>    ::=  ϵ
6                                 |    Expression Comma-Expression
```

1 The Formal-Parameter-Sequence may consist of arbitrarily many, comma-separated, formal parameters and is used when declaring a procedure or function. A formal parameter consists of a Type-Denoter followed by an optional keyword, *ref*, that determines whether or not the parameter should be passed to the function or procedure by reference. The Identifier I of the formal parameter is bound as a vN to a variable with the value that is passed as argument to the funtion or procedure, and can be used directly within the body of these.

5 The Actual-Parameter-Sequence is a comma separated list of expressions and is used when calling a procedure or function. Each expression is evaluated and the yielded values are then passed along as arguments to whatever is called. The Actual-Parameter-Sequence must match the corresponding Formal-Parameter-Sequence, i.e. the number of actual parameters must match the number of formal parameters and each actual parameter must be of the same type as its corresponding formal parameter.

7.2.6 Type-denoters

The type-denoters are used to denote data types for Formal-Parameter-Sequences (Section 7.2.5) and for reference declarations (Section 7.2.7). Types are used to determine what a programming language is dealing with. There are only very few types in DOGS that require specific syntax, and as such these are denoted while all others, such as `string` and `boolean`, are included in the standard environment (cf. Section 7.4.

```
1  <Type-Denoter>   ::=   Identifier
2                     |   array of Identifier
3                     |   weight of Identifier
4                     |   label of Identifier
5                     |   set of Identifier
```

1 The Type-Denoter I denotes the type bound to the identifier I.

2 The Type-Denoter *array of* I denotes a type whose values are arrays. Each array value of this type has an index range whose lower bound is 1 and whose upper bound is an integer. This integer is set when the array is declared. Each array value has one component of type I for each value in its index range.

3 The Type-Denoter *weight of* I denotes a weight function with elements of the type bound by identifier I.

4 The Type-Denoter *Label of* I denotes a label function with elements of the type bound by identifier I.

5 The Type-Denoter *set of* T denotes a set whose values are restricted to the type bound by identifier I.

7.2.7 Declarations

DOGS has quite a strict typing system (defined in Section 8 that ensures that everything has to be declared before use. By using declarations, identifiers are bound to a specific type which enable DOGS to avoid type-clash errors. Furthermore, declarations may update variables.

```
1  <Declaration>                      ::=  ϵ
2                                     |    Single-Declaration-Const-Type Declaration
3                                     |    Single-Declaration; Declaration
4  <Single-Declaration>               ::=  SAV-Help-Declarations (ϵ | Declaration-Assignment)
5                                     |    weight in V-Name of Identifier Identifier
6                                     |    label in V-Name of Identifier Identifier
7  <Single-Declaration-Const-Type>    ::=  record Identifier (Const-Declaration — SAV-
                                          Help-Declarations) Declaration-Assignment Type-
                                          Declaration-Helper;
8                                     |    Const-Declaration Declaration-Assignment;
9  <Type-Declaration-Helper>          ::=  ϵ
10                                    |    , (Const-Declaration — SAV-Help-Declarations)
                                          Declaration-Assignment Type-Declaration-Helper
11 <Multi-Declaration-Const-Type>     ::=  ϵ
12                                    |    Single-Declaration-Const-Type Multi-Declaration-
                                          Const-Type
13 <Single-Secondary-Declaration>     ::=  ϵ
14                                    |    procedure Identifier (Formal-Parameter-Sequence)
                                          Command Single-Secondary-Declaration
15                                    |    function Type-Denoter Identifier (Formal-
                                          Parameter-Sequence) Command Single-Secondary-
                                          Declaration
16 <Declaration-Assignment>           ::=  ϵ
17                                    |    ::= Assign-Expr
18 <Const-Declaration>                ::=  constant SAV-Help-Declarations
19 <Variable-Declaration              ::=  Identifier Identifier
20 <Set-Declaration>                  ::=  set of Identifier Identifier
21 <Array-Declaration>                ::=  array of Identifier Array-Size-Denoter Identifier
22 <Array-Size-Denoter>               ::=  ϵ
23                                    |    [ Integer-Literal ]
24                                    |    [ Integer-Literal ] Array-Size-Denoter
25 <Sav-Help-Declarations>            ::=  Set-Declaration
26                                    |    Array-Declaration
27                                    |    Variable-Declaration
```

1 A Declaration may consist of arbitrarily many *Single-Declaration-Const-Type* or *Single-Declaration* productions.

4 The Single-Declaration *SAV-Help-Declarations*, *SHD* may consist of a Set-Declaration, an Array-Declaration, or a Variable-Declaration followed by an optional assignment of an expression, the Declaration-Assignment, *dA*.

5 The Single-Declaration *weight in vN of* I_1 I_2 is a declaration of a weight function. *vN* must be of type `graph` or `diGraph`. I_1 denotes type of the weight and I_2 is the variable name, to which the weight function is bound.

6 The Single-Declaration *label in vN of* I_1 I_2 is a declaration of a label function. *vN* must be of type `graph` or `diGraph`. I_1 denotes the type of the label and I_2 is the variable name, to which the label function is bound.

7 The Single-Declaration-Const-Type may either be a declaration of a new type, i.e. a record, or a declaration of a constant. The record declaration is of the

form *record I* followed by arbitrarily many (at least one), comma-separated, *SHD*'s or *Const-Declaration*'s, *cD*, each followed by *dA*. *I* is the identifier to which the record structure is bound and the different declarations make up the structure of the record. In case of *cD* it should be noted that *dA* is non-optional, i.e. *dA* must not be the empty string, ϵ.

The constant declaration is specified in item 20 and must be followed by a non-optional *dA*.

11 A Multi-Declaration-Const-Type may consist of arbitrarily many Single-Declaration-Const-Type's.

14 The Single-Secondary-Declaration *procedure I (Formal-Parameter-Sequence) C Single-Secondary-Declaration*, *procedure I (FPS) C SSD*, is a procedure declaration. I is bound to a procedure whose formal-parameter-sequence is FPS and whose body is the command C. When I is called with an argument list, FPS is associated with that list, then C is executed in the environment of the procedure declaration overlaid by the bindings of the formal-parameters.

15 The Single-Secondary-Declaration *function tD I (FPS) C SSD* is a function declaration. I is bound to a function whose formal-parameter-sequence is FPS and whose body is the command C. When I is called with an argument list, FPS is associated with that list, then C is executed, in the environment of the function declaration overlaid by the bindings of the formal-parameters. tD is a Type-Denoter that specifies the return type of the expression that I evaluates and returns to the callee.

16 The Declaration-Assignment may become an assignment of the form *::= Assign-Expr*, or the empty string. This is evaluated as any other assignment.

18 The Const-Declaration is either a Set-Declaration, an Array-Declaration, or a Variable-Declaration with the word *constant* prepended. This is always followed by a non-optional *dA*. Once I is declared to be a constant of any given type allowable and initialized, it cannot change value.

19 The Variable-Declaration I_1 I_2 is a declaration of a variable. I_2 denotes a variable that is going to be of the type identified by I_1. After initialization I_2 may not be redeclared or used as any other type than what was denoted by I_1.

20 The Set-Declaration *set of* I_1 I_2 declares a set containing an arbitrary number of elements of the type denoted by I_1. The set is bound to, and thereby identified by, I_2.

21 The Array-Declaration *array of* I_1 *Array-Size-Denoter* I_2, is a declaration of an array identified by I_2. The array is of the type denoted by I_1 and may only contain elements of this type. The array is indexed, with its lower bound being 1 and its upper bound being defined by the *Array-Size-Denoter*, *ASD*. If the *ASD* is omitted when declaring the array, it may only be assigned with another array of same type, and it shall then inherit all properties of said array. However, there may be arbitrarily many *ASD*'s and if there are more than one,

the array is said to be *multidimensional*. Basically, this means that each index, defined by the first *ASD*, contains an array of size defined by the second *ASD* and so forth - all of same type as denoted by I_1. An array is static, and may not change size after it has been initialized.

7.2.8 Program

1	<Program>	::=	`program` Identifier `;` Package-Import Multi-Declaration-Const-Type Single-Secondary-Declaration
2		|	`package` Identifier `;` Package-Import Multi-Declaration-Const-Type Single-Secondary-Declaration
3	<Package-Import>	::=	ϵ
4		|	`import` V-Name`;` Package-Import

1 The Program is the topmost part of the DOGS language. The program may be either declared as *program*, in which case it has to have a *main* procedure, or *package*, in which case it must not have a *main* procedure, both identified by an identifier *I*. This is followed by first an arbitrary number of *Package-Import*'s, *pI*, then an arbitrary number of *Multi-Declaration-Const-Type*'s, *MDCT*, and finally a number of *Single-Secondary-Declarations*'s, *SSD*.

4 The Package-Import *import vN;*, imports the package identified by vN and includes references to functions, procedures, and globally declared records and constants. Following all of these elements are accessible from the current DOGS program. *vN* must be the name of *package*.

7.2.9 Lexicon

The DOGS lexicon is presented below:

<Token>	::=	Integer-Literal | String-Literal | Float-Literal | Boolean-Literal | Identifier | Operator | `array` | `ref` | `begin` | `constant` | `do` | `else` | `end` | `function` | `procedure` | `if` | `in` | `let` | `of` | `record` | `then` | `type` | `while` | `for` | `foreach` | `to` | `downto` | `where` | `switch` | `endswitch` | `return` | `case` | `weight` | `label` | `edge` | `vertex` | `program` | `package` | `set` | `++` | `--` | `break` | `infty` | `default` | `import` | `.` | `:` | `;` | `,` | `:=` | `(` | `)` | `[` | `]` | `{` | `}`
<Integer-Literal>	::=	Digit Digit*
<Float-Literal>	::=	Digit Digit* `.` Digit Digit*
<Boolean-Literal>	::=	`true` | `false`
<String-Literal>	::=	‘‘ Escape-Literal* Letter* Escape-Literal* ’’
<Escape-Literal>	::=	\n | \t | \\
<Identifier>	::=	Letter (Letter | Digit)*
<Comment>	::=	`//` Graphic* end-of-line
	|	`/*` Graphic* `*/`
<Blank>	::=	space | tab | end-of-line
<Graphic>	::=	Letter | Digit | Operator | space | tab | `.` | `:` | `;` | `,`
	|	| `(` | `)` | `[` | `]` | `{` | `}` | `_` | | | `!` | ’ | ‘ | ‘‘ | `#` | `$`
<Letter>	::=	`a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` | `i` | `j` | `k` | `l` | `m` | `n` | `o` | `p` | `q` | `r` | `s` | `t` | `u` | `v` | `w` | `x` | `y` | `z` | `A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` | `P` | `Q` | `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z`
<Digit>	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
<Operator>	::=	`+` | `-` | `*` | `/` | `&` | `=` | `<=` | `>=` | `<` | `>` | `%` | `div` | `<>` | `and` | `or` | `not` | `xor`

7.3 Classification of the DOGS Grammar

For parsing purposes grammars are classified according to how they may be recognized, i.e. they are classified according to the main principles of algorithms that recognize them. Therefore, it is important to know the classification of a grammar before a strategy for parsing is chosen. Generally, algorithms that are used to recognize languages follow two main strategies that may be applied for parsing; a top-down strategy or a bottom-up strategy [Seb04].

According to [WB00] the top-down approach takes an input string, examines the terminal symbols from left to right, and creates a syntax tree from these terminal symbols towards the topmost node, the root node (the start production of the grammar). A well-known top-down algorithm is the *recursive descent* algorithm.

The bottom-up approach also takes an input string, examines the terminal symbols from left to right, and creates a syntax tree from the top node towards the terminal symbols. An example of a bottom-up algorithm is an *LR* algorithm.

In Figure 7.1 different classifications of grammars and their relations are shown. The LR algorithm recognizes the LR set of grammars and the *recursive descent* algorithm recognizes the *LL* set of grammars. It is clear that the set of LL grammars is a subset of the LR grammars.

In the following we will give a brief description of some main charateristics of the LL and LR sets of grammars, in order to give a classification of the DOGS grammar.

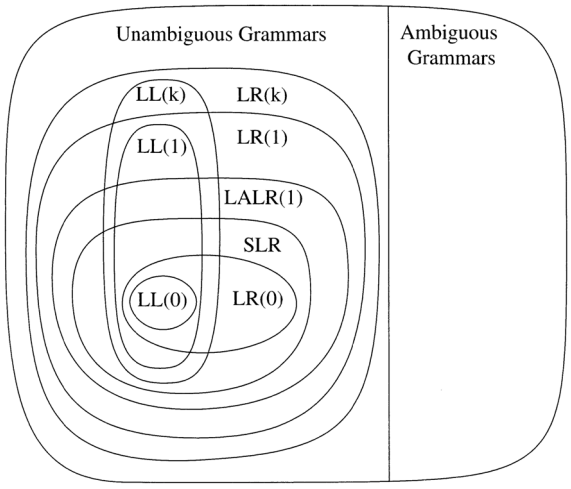

Figure 7.1: Hierarchy of grammars [Tho]

7.3.1 LL Grammars

The term "LL grammar" stems from what the algorithm, that is able to parse it, does: It reads the terminal symbols from **L**eft to right and it makes a **L**eftmost derivation [Seb04]. Furthermore, they may be numbered, as in Figure 7.1, according to the look-ahead needed to parse them, i.e. LL(1) needs a look-ahead of one terminal, LL(2) needs a look-ahead of two terminals, and so on.

A more formal definition of an LL grammar, or more specifically, an LL(1) grammar is [WB00]:

- If the grammar contains a production in which the right side is of the form $X|Y$ the terminal that may start an X production must be different from the terminal that may start a Y production.
- If the grammar contains a production in which the right side is of the form $X*$, the terminal that may start an X production must be inherently different from the terminal that may follow the $X*$.

This may be expanded to a formal definition of an LL(k) grammar, by substituting the terminal with the k first terminals in the above definition.

Also, it is worth noting that the following properties of LL-grammars can be proved [Tho]:

- No grammar with productions that are left-recursive is an LL(1) grammar.
- No grammar that is ambiguous is an LL(1) grammar (cf. Figure 7.1).
- Although many languages can be expressed with an LL(1) grammar, this is not the case for all languages.
- If a grammar contains no empty strings, then if each production starts with a unique terminal, the grammar is an LL(1) grammar.

From Figure 7.1 it is clear that the set of LL(1) grammars is rather small and it turns out that it can be difficult to express common programming constructs as an LL(1) grammar [Gagb]. However, for unknown reasons, many programming language developers still design the syntax of their programming languages such that they are suitable for recursive descent parsing, i.e. they express the grammar as LL(1) grammars [WB00].

7.3.2 LR Grammars

An LR grammar is classified as an LR grammar because it may be parsed with an LR algorithm. An LR algorithm is, as previously mentioned, a bottom-up parsing algorithm that relies on parse-tables to make parse decisions. The algorithm is called an LR algorithm because it reads terminals from **L**eft to right and makes a **R**ightmost derivation [Seb04].

LR algorithms are widely used for parsing purposes because they are very flexible as they work for most grammars that describe programming languages. However, LR parsers do take up a lot space and are difficult to write [Seb04].

As a response, several different LR algorithms have been developed, all being able to parse a subset of the grammars and differentiate only in the way they construct their parse-tables. Amongst these are [Tho]:

- LR0 is the simplest algorithm and although being important for theoretical purposes it is rather weak, thus not very useful in the "real" world.
- SLR, or Simple LR, is an improved version of LR0 and although it, too, is rather weak, it is able to parse more grammars and is therefore more practical.
- LR1 is essentially an LR0 algorithm, but because it is enhanced with an extra look-ahead token it is a very powerful algorithm. Though, in spite of its power it is not a very used algorithm since it takes up too much memory when used.
- LALR is a modified version of the LR1 algorithm in which the memory requirements have been greatly reduced, at a relatively small cost of power. This makes the LALR algorithm the most commonly used parse algorithm today [Tho].

Being that the LALR parse algorithm is the most commonly used algorithm for parsing, it seems reasonable to elaborate on what is known about LALR algorithms.

7.3.2.1 LALR Grammars

Parsers that recognize LALR grammars, or Look-Ahead LR grammars, are obviously called LALR parsers. A grammar is called a LALR grammar if a LALR parse table can be constucted without any conflicts. A quick check, is to use a LALR parser generator tool and observe if it is able to generate this table [Tho]. As seen in Figure 7.1, the set of LALR grammars is greater than the sets of LR0 and SLR grammars. When it is stated that the memory requirements for LALR parsers are less than those for LR grammars it is because a LALR algorithm constructs relatively smaller parse tables [WB00].

However, the most important statement, probably, of LALR grammars is that:

Basically all programming languages have LALR grammars [Tho].

This is why LALR parsing methods, in particular, are commonly used for automatic tools today [Tho].

7.3.3 The DOGS Grammar

The main issue that we have to resolve for the DOGS grammar is whether a top-down parsing approach or a bottom-up parsing approach to parse the grammar should be taken. Based on the previous section, it follows that we may determine if a top-down approach is suitable by determining if the grammar is an LL grammar.

Consider the following part of the DOGS grammar:

<Single-Command>	::=	Open-Command
	|	Closed-Command
<Open-Command>	::=	`if` Expression `then` Single-Command
	|	`if` Expression `then` Closed-Command `else` Open-Command
		...
<Closed-Command>	::=	Basic-Commands
	|	`if` Expression `then` Closed-Command `else` Closed-Command
		...

It is clear that this grammar does not oblige to the first condition in the formal definition of an LL(1) grammar as presented in Section 7.3.1. This is because both the Open-Command and the Closed-Command may start with same terminal `if` (furthermore, this grammar does not even oblige to second condition of the definition). In extension, it can even be shown that the grammar is not an LL grammar at all, since the Open-Command and Closed-Command, may start with arbitrarily many of the same terminals (`if Expression then Closed-Command else`). Hence, top-down parsing of the DOGS grammar is not possible and the parsing strategy applied for the DOGS language must be the bottom-up strategy.

However, as previously implied, it is a tedious task to classify a grammar within the grammars that may be recognized by a bottom-up parsing approach. Though, based on what is known about LALR grammars, it seems plausible to suspect the DOGS grammar to be a LALR grammar. This, however, is easily shown by constructing a LALR parse table, but since this is a rather difficult task [Tho], it will not be clarified until a parser has been constructed.

7.4 Standard Environment

The DOGS syntax does not cover all functionality in the language. Some of it cannot be expressed in DOGS , thus claiming an environment in which this functionality can reside. This is the standard environment in DOGS. It contains a collection of predefined types, constants, and procedures or functions. Before analyzing a source program the contextual analyzer initializes the identification table with entries for the identifiers in the standard environment. With this initialized identification table it is possible to do type checking on the standard types.

The types in the standard environment are the primitive types such as `integer` and `string` and the graph related standard types such as `vertex` and `diGraph` (directed graph). The constants are the boolean values `true` and `false`.

The most interesting part of the standard environment in DOGS is the selection of standard functions and procedures. In the analysis we concluded that a language for working with algorithms would be of no use if input/output is not possible and therefore DOGS must support it. Since it would not provide any new interesting information to include input/output keywords in the syntax, we have chosen (as in most programming languages) to support input/output by the use of functions and procedures. These have been placed in the standard environment. We have included a rather small selection of input/output functions and procedures, since more advanced ones can be created by using the fundamentals included. For example, one can wish to read a whole text file into a single string variable. Such a function can be programmed by using the `readLine` function from the standard environment until `end-of-file` has been reached. Libraries can be used as collections of such more advanced input/output.

In addition to input/output the standard environment in DOGS also contains a selection of functions and procedures for working with graphs. These provide functionality for creating a graph, creating a vertex with a unique identifier, adding vertices or edges to a graph, removing vertices or edges from a graph, and retrieve information from the graph representation.

As mentioned the functionality of the functions and procedures in the standard environment cannot be written in DOGS, thus it must be implemented as a part of the compiler. Functions and procedures that are somehow standard, but can be written in DOGS will be placed in standard libraries instead.

The complete standard environment with comments can be found in Appendix A.

7.5 Dijkstras Algorithm in DOGS

Having introduced the DOGS syntax we will briefly clarify it by an example. We return to Dijksta's Algorithm, only this time we write it in DOGS code. It will not go through a thorough review (a scrutiny is found in Section 2.2, page 19) but merely illustrate a way of implementing it in DOGS.

7.5.1 Presentation of Dijkstra's Algorithm in DOGS

Listing 7.1 shows Dijkstra's Algorithm in DOGS. A package, `graphToolkit`, has been imported which consists of functions and procedures written in DOGS that are useful graph "tools".

In `main` we declare the variables necessary for Dijkstra to run, i.e. a graph G, vertices, edges, a set of strings V, and a weight function that relates to G. In the body of `main` we call the standard functions `addVertices` for connecting V to G and `addToSet` for adding strings to V. We then assign the vertices declared to a value in the graph, i.e. they are now part of G. Correspondingly the edges are initialized and connected to G using `addEdge`. Finally, the weight function weights the edges

and `dijkstra` is invoked with G, w, and a start vertex a passed. This constitutes the preliminary work.

Similarly, the `let-in` block in `dijkstra` does all the groundwork, i.e. declaring two sets of vertices V and S, two label functions to the passed graph G, d and pi, an instance of a set *Queue*, and the start vertex u. From this point the actual Dijkstra's Algorithm initiates. The DOGS code can be compared to the pseudo-code example from Section 2.2 to see the resemblance in appearance.

```
1  program DijkstraInDOGS;
2
3  import graphToolkit;
4
5  procedure main(array of string args)
6  let
7    graph G;
8    vertex v1; vertex v2; vertex v3;
9    edge e1; edge e2; edge e3;
10   set of string V;
11   weight in G of integer w;
12 in
13 begin
14   addVertices(G,V);  /*resides in a standard package*/
15   addToSet(V, "a");
16   addToSet(V, "b");
17   addToSet(V, "c");
18   v1 := getVertex(G, "a");
19   v2 := getVertex(G, "b");
20   v3 := getVertex(G, "c");
21   e1 := (v1, v2);
22   e2 := (v1, v3);
23   e3 := (v2, v3);
24   addEdge(G, e1); addEdge(G, e2); addEdge(G, e3);
25   //An alternative would be addEdge(G, (v2, v3));
26   w(e1) := 5;
27   w(e2) := 3;
28   w(e3) := 7;
29   dijkstra(G, w, a);
30 end
31
32
33 procedure dijkstra(graph G, weight of integer w, vertex s)
34 let
35   set of vertex V;
36   label in G of integer d;
37   label in G of vertex pi;
38   set of vertex S;
39   set of vertex Queue;
```

```
40    vertex u;
41  in
42  begin
43    V := vertices(G);
44    foreach vertex v in V do
45      begin
46        d(v) := infty;
47        addToSet(Queue, v);
48      end
49    d(s) := 0;
50    while sizeOfSet(Queue) > 0 do
51      begin
52        u := extractMinLabel(Queue, d);
53        addToSet(S, u);
54        foreach vertex v in V where (isEdge(u, v) and not
              inSet(S, v)) do
55          if (d(v) > d(u) + w((u, v))) then
56            begin
57              d(v) := d(u) + w((u ,v));
58              pi(v) := u;
59            end
60       end
61  end
```

Listing 7.1: Dijkstra's Algorithm written in DOGS

Chapter 8

Type System in DOGS

In this chapter we will present the formal type system in DOGS in order to justify leaving aside the type checking in the semantics, in the assumption that it has already been done at this stage. In other words, we will assume that DOGS programs are *well typed*. This claim requires explanation. A brief introduction as to what type systems, are along with a discussion of the need for a formal type system, account for introducing a selection of type rules in DOGS. The entire DOGS type system can be found in Appendix D. The chapter is based on [Car04].

8.1 Introducing Type Systems

According to [Car04] the purpose of a type system is to prevent the occurence of runtime errors when running a program. Succeeding in doing so makes a language *type sound*, a property that can be obtained by formalizing the type system of the language and prove the *type soundness* theorem stating that *well-typed programs are well behaved* [Car04]. Whether the language in question can be proved to be type sound, formal type systems are more powerful than informal language descriptions as they characterize the type structure of a language in a concise way, that allows for an unambiguous implementation.

8.1.1 Well-behaved Programs

Type systems are used to determine *well behaviour* of programs. A program is said to be well behaved if it avoids causing a certain type of execution errors to occur, the *forbidden* errors. This class of errors is a subset of the possible execution errors, including *trapped* errors (i.e. errors that causes the computation to stop immediately) and a subset of the *untrapped* errors (i.e. errors that go unnoticed for a while and later in execution cause arbitrary behaviour [Car04]).

By performing compile time checks maintained by the typechecker, typed languages (i.e. languages where variables can be given nontrivial types [Car04]) can enforce good behaviour and prevent ill-behaved programs from ever running. A program that passes the typechecker is said to be *well-typed*.

8.2 Formalizing Type Systems

Having identified the meaning of well-typed programs, we now describe the strategy of formalizing the type system of a language, as it explains the steps in setting up the DOGS type system. The steps are:

- Describe the syntax of types and terms (statements, expressions, declarations, etc) of the language.
- Define the scope rules of the language.
- Define typing environments.
- Identify type relations.

The scope rules of a language need to be static in the sense that binding location of identifiers has to be determined before runtime [Car04].

Identifying type relations involves describing the relation *has-type* denoted by $M : A$ between terms M and types A. Further, if the language has subtypes, a relation *subtype-of* has to be defined, denoted by $A <: B$ between types A and B. This also applies for type equivalence between two types A and B, denoted by $A = B$

Static typing environments are closely related to the symbol table of a compiler in the typechecking phase in that they record the types of free variables. The notation $\Gamma \vdash M : A$ expresses that the term M has type A in the type environment Γ.

8.3 Typing Judgments

The notation introduced in the previous section, $\Gamma \vdash M : A$, meaning that M has type A in Γ is called a *typing judgment*. Generally, judgments are on the form:

$$\Gamma \vdash \Im$$

where $\Im$ is an assertion with free variables declared in Γ. Variables declared in Γ is indicated by $dom(\Gamma)$. A judgment we will use is

$$\Gamma \vdash \diamond$$

which asserts that Γ is well-formed, meaning that it has been properly constructed [Car04].

8.4 Defining the Type Rules in DOGS

The type rules in DOGS will be outlined in the following sections. According to the steps in formalizing the DOGS type system, we introduce an abstract syntax from which we can specify the rules.

DOGS is a statically scoped language (cf. Section 6.9, page 49) and as such the identifiers are bound before runtime.

8.4.1 Abstract Syntax

The DOGS syntax presented in Chapter 7.2, page 56, is in ways too concrete to use in the formal semantics, since it provides unnecessary information, e.g. operator precedence. Instead, we need a syntax with a somewhat more simplified overview of our language and with a classification of production rules that is more "form oriented", i.e. a syntax that solely describes the structure of the different language constructions, regardless of the fact, that it allows for syntactical constructs that makes no sense in DOGS.

8.4.1.1 Syntactical Categories

Table 8.1 lists the syntactical categories and meta variables representing elements in these categories. As mentioned, elements fall into the same category when they share characteristics (e.g. elements from IntegerExpression can enter into arithmetic expressions, whereas elements from Command cannot).

Pro	$\in$	DOGSProgram
S	$\in$	Command
i	$\in$	IntegerExpression
f	$\in$	FloatExpression
n	$\in$	NumberExpression
op	$\in$	Operators
b	$\in$	BooleanExpression
w	$\in$	StringExpression
exp	$\in$	Expression
c	$\in$	Constant
v	$\in$	Value
int	$\in$	Integer
float	$\in$	Float
bool	$\in$	Boolean
str	$\in$	String
x	$\in$	Var
X	$\in$	generalizedVar
D_V	$\in$	DecVar
D_{VB}	$\in$	DecVarBlock
D_{VR}	$\in$	DecVarRec
D_C	$\in$	DecConst
D_R	$\in$	DecRec
D_B	$\in$	DecFuncProcBlock
$Proc_N$	$\in$	ProcedureName
$Func_N$	$\in$	FunctionName
Rec_N	$\in$	RecordName
$type_N$	$\in$	TypeName
$Rtype_N$	$\in$	RecordType

Table 8.1: The syntactical categories in DOGS

8.4.1.2 Production Rules

The abstract syntax is broadly divided in declarations, commands, and expressions. We will not explain the production rules for the different language constructs here, since a walkthrough is found in Chapter 7.2, starting on page 56.

8.4.1.3 Declarations

Pro ::=	`program` Imp $D_R D_C D_B$ | `package` Imp $D_R D_C D_B$
Imp ::=	`import` str`;` | ImpImp | ϵ
D_{VB}::=	D_{VB} D $_{VB}$ | D_V `;` | ϵ
D_{VR}::=	D_{VR}`,` D_{VR} | D_V | ϵ
D_V ::=	$type_N$ x | $type_N$ x `:=` exp | `array of` $type_N$ x ArrayDim | `set of` $type_N$ x | `weight in` X `of` $type_N$ x | `label in` X `of` $type_N$ x
D_C ::=	D_C D_C | `constant` $type_N$ c := exp`;` | ϵ
D_B ::=	$D_B D_B$ | `function` type $Func_N(Par_F)$ LetIn | `procedure` $Proc_N(Par_F)$ LetIn | ϵ
D_R ::=	`record` $Rtype_N$ D_{VR}`;` | D_R D_R | ϵ
LetIn ::=	let D_{VB} in S | S

8.4.1.4 Commands and Expressions

S ::=	$S_1 S_2$ | X `:=` exp`;` | X `:=` {RecAggr} `;` | X`(` exp_1 `)` := exp_2; | X ArrayDim `:=` exp`;`| `if` b `then` S | `if` b `then` S_1 `else` S_2 | `while` b `do` S | `do` S `while` b`;` | `for` x `:=` i_1 `to` i_2 `do` S | `for` x `:=` i_1 `downto` i_2 `do` S | `foreach` $type_N$ x `in` X `do` S | `foreach` $type_N$ x `in` X `where` b `do` S | `switch` X S `endswitch` | `return` exp`;` | `break;` | X++`;` | X−−`;`| $Proc_N$`(`Par_A`);` | `begin` S `end` | `case` int`:` S | `case` int_1`..`int_2`:` S | `default:` S | ϵ
X ::=	x | Rec_N`.`X
RecAggr ::=	RecAggr, S | S | ϵ
b ::=	bool | X | n_1 `=` n_2 | n_1 `<` n_2 | n_1 `>` n_2 | n_1 `<=` n_2 | n_1 `>=` n_2 | n_1 `<>` n_2 | `not` b_1 | b_1 `=` b_2 | b_1 `and` b_2 | b_1 `or` b_2 | b_1 `xor` b_2 | `(`b`)` | X ArrayDim | $Func_N$`(`Par_A`)`| X`(` exp `)`
op :=	`+` | `-` | `*` | `/` | `div` | `mod`
i ::=	int | X | i_1 `op` i_2 | `(`i`)` | X ArrayDim | $Func_N$`(`Par_A`)` | X`(` exp `)` | `infty`| −`infty`
f ::=	float | X | f_1 `op` f_2 | `(`f`)` | X ArrayDim | $Func_N$`(`Par_A`)`| X`(` exp `)` | `infty`| −`infty`
n ::=	i | f
w ::=	str | X | exp_1 & exp_2 | $Func_N(Par_A)$
exp ::=	n | n op n | b | w | X | $Func_N(Par_A)$ | (X$_1$`,` X$_2$) | X`(` exp `)` | X ArrayDim
Par_A ::=	exp_1 | exp_1, Par_A | ϵ
Par_F ::=	$type_N$ x | Par_F $type_N$ x | $type_N$ `ref` x | Par_F $type_N$ `ref` x | ϵ
ArrayDim ::=	[i]ArrayDim |ϵ

8.4.2 Types and Judgements in DOGS

In Table D.1 the different types in DOGS are listed and categorized. They are grouped in sets for reasons of readability and comprehension which will be apparent when exploring the different type rules. Notice that the `record` type of DOGS is nowhere to be found. It has deliberately been left out of the type system due to the fact that it is rather complex within the DOGS language and will therefore be dealt with seperately in a more informal manner.

The types in Table D.1 form the foundation for the type system. Notice that types

`boolean`	boolean type	$\in basic \in primitiveType$
`string`	string type	$\in basic \in primitiveType$
`float`	float type	$\in basic \in primitiveType$
`integer`	integer type	$\in basic \in primitiveType$
`infty`	infty type	
`vertex`	vertex type	$\in primitiveType$
`edge`	edge type	$\in primitiveType$
`array of` primitive	array type	
`set of` primitive	set type	
`weight of` primitive	weight type	
`label of` primitive	label type	
$type_{N_1} \rightarrow type_{N_2}$	function type	
`proc` $type_N$	procedure type	
`prog`	program type	
`pack`	package type	
$type_N$ *constant*	constant type	

Table 8.2: Types in the DOGS type system

for functions, procedures, programs, and packages are amongst the types. Intuitively these are not considered to be data types in DOGS, however, in order to ensure that either of them is well-typed, it makes sense to include them among the data types. Another noticeable thing is the introduction of the sets *primimtiveTypes* and *basic* that contain primitive types of DOGS and types that are commonly regarded as primitives, respectively.

The DOGS type system has a series of judgements listed in Table D.2. One judgement that may need some explanation is $\Gamma \vdash D \therefore A$, this judgement asserts that the declaration D is well-formed and with the signature A. The signature of such an declaration is, essentially the type of the declaration [Car04], however, in DOGS the signature may consist of list elements, very similar to environments.

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash type_N$	$type_N$ is a well-formed type in Γ
$\Gamma \vdash S$	S is a well-formed command in Γ
$\Gamma \vdash exp : type_N$	exp is a well-formed expression of type $type_N$ in Γ
$\Gamma \vdash D \therefore A$	D is a well-formed declaration of signature A in Γ
$\Gamma \vdash type_{N_1} <: type_{N_2}$	$type_{N_1}$ is a subtype of $type_{N_2}$ in Γ

Table 8.3: Judgements for the DOGS type system

Now that the types and judgments in DOGS have been presented, we may begin to state actual type rules. This includes rules that ensure that types of the language are well-formed, rules that ensure that declarations are well-formed and so on. As previously mentioned we will now present a few of these rules.

The first rule that is introduced in DOGS is actually the axiom that states that the empty set is well-formed. Along with this axiom, a few important rules that

ensure that any variable can only be declared once in any environment and that a variable is only available if it has already been declared are presented. The rules are stated in Table 8.4.

[env-empty]	[env-extension]	[env-var-exists]
$\dfrac{}{\emptyset \vdash \diamond}$	$\dfrac{\Gamma \vdash type_N \qquad x \notin dom(\Gamma)}{\Gamma, x : type_N \vdash \diamond}$	$\dfrac{\Gamma, x : type_N \vdash \diamond}{\Gamma, x : type_N \vdash x : type_N}$

Table 8.4: Environment rules

The DOGS type system differentiates between declarations that declare new variables, expressions that have a specific type, and commands that do not have types. The first rule is the [label] rule in Table 8.5 182.

[label]

$$\frac{\Gamma \vdash x : \texttt{labelof}type_N \qquad \Gamma \vdash exp : \texttt{vertex}}{\Gamma \vdash x(exp) : type_N}$$

Table 8.5: Expression rule, found in Table D.18, page 182

The rule states that the expression $x(exp)$ is a type correct label iff x has been declared as a variable, i.e. if it exists in the environment Γ and expression exp is a type correct vertex.

Commands may consist of other commands and expressions and the next rule that will be presented is the [LetIn] command rule in Table 8.6. It considers the function or procedure body, that may also contain declarations. The above rule states that the command `let` D_{VB} `in` S is type correct iff the block of declarations D_{VB} is well-formed under environment Γ and the command S is well-formed under Γ combined with the signature of D_{VB}, A, i.e. the different declarations of D_{VB}

The final rule that is presented in this section is the [dec-var-init] rule in 8.7, which is a declaration that is assigned a value. The above rule states that a declaration of variable x is well-formed iff $type_N$ is a well-formed type in DOGS and

[LetIn]

$$\frac{\Gamma \vdash D_{VB} \therefore (A) \qquad \Gamma, A \vdash S}{\Gamma \vdash (\texttt{let}\ D_{VB}\ \texttt{in}\ S)}$$

Table 8.6: Command rule, found in Table D.3, page 179

[dec-var-init]
$$\frac{\Gamma \vdash type_N \quad \Gamma \vdash exp : type_N}{\Gamma \vdash (type_N\ x{:=}exp) \therefore (x : type_N)}$$

Table 8.7: Declaration rule, found in Table D.9, page 174

expression *exp* is of the same type as the declaration, i.e. the expression must be of type $type_N$. Furthermore, it implicitly follows from rules [env-extension] and [env-var-exists] (Table 8.4) that a variable of the same name as x must not have been previously declared, within the environment Γ.

8.5 Records

As previously mentioned, the record type of the DOGS language was intentionally left out, mainly because it is a type type, i.e. it is a type that is used to create new types with. In the following we will give brief descriptions of legal uses for the record type, as well as account for its evaluation. By the end of this section it is our intention that it will be possible to check the record type along with the formal DOGS type system, of which the record is not a part. In order to cover the wide range of applications for a record type, we will have to walk through the three main areas in which record types may appear. These are declarations, expressions, and assignments. Before these situations are explored, some basic use of the record structure is explained.

A record is used to declare a record type. This record type may then contain declarations of variables that will only be available through the record. These variables are known as fields and are accessed through a "dot"-notation, i.e. the field f of a record r should be accessed using the notation $r.f$.

8.5.1 Declarations

A record is involved in two types of declarations; one is the declaration of a new record type, the other is the declaration of a new variable with the previously declared record type.

The declaration of a new record type consists of an identifier for this new type, a number of standard variable declarations, that may even be other, previously declared, record types. These variables may be assigned with value that will be the default value when a new variable is declared with the record type.

When a record type is declared, new variables may be declared of that type. When doing this, the fields of the record type become available solely through the variable of the record type. In case of multiple variables of the same record type, they all have distinct fields. Also, when declaring a new variable of the record type, it may initially be assigned another variable of the same record type or a record-aggregate, both described in Section 8.5.3.

8.5.2 Expressions

Generally, an expression of a record type may evaluate to either the record type or the type of a field contained within the record type, i.e. if a field contained the record type is addressed (through the "dot"-notation), the evaluated type is that of the field, otherwise the evaluated type is the record type itself.

8.5.3 Assignments

When a record type appears in an assignment command it may, as with other types, appear on either the left side or on the right side of the command. The right side is redundant, as the record will evaluate as an expression, and this case is described in the previous section. The case where the record type appears on the left side, on the other hand, is a bit more interesting. Depending on whether or not it is pointing at a field within the record (assuming that this field is not a record type itself), the right-side of the assignment may be an expression that evaluates to a specific type, a record type, or the special *record aggregate*. In the case that the type of a record type variable differs from any record type, the left side must be a field and the right side of the assignment command must be an expression of the same type as that field.

In the case that the type of a record type variable is of a record type, it may either not address any field, or it may be adressing a field that is actually of a record type itself. In this case, the type of the right side of the expression must be of the same record type, or it may be a record aggregate that is well-formed according to the respective record-type.

A record-aggregate is a collection of assignment commands. These assignments are each a specific assignment of an expression to a field of the record. The record-aggregate is considered well-formed iff it does not contain assignments to other fields than those contained in the record and all assignments are legal. In case of a record type field within a record type variable, record-aggregates may be nested.

Summary

Following this section and Appendix D, we deem that it is reasonable to assume that DOGS programs are well typed. As mentioned in the beginning of this section, this assumption is valuable when developing the semantics, given in the next chapter.

Chapter 9

DOGS Operational Semantics

In [Hüt04] it is stated that (translated from Danish)

> Only an exact semantic definition can provide an exhaustive, implementation-independent and totally precise definition of all aspects of the language.

In this chapter we will provide such a semantic definition for DOGS. The purpose is to provide the language implementors with a complete reference to how programs written in our language will behave on execution.

Several different semantic standards are available, however according to [Hüt04] only operational semantics expresses *how* a program is executed (as changes in states within a real or virtual machine). Because of this distinction we have found that operational semantics is the better choice for a formal description of DOGS.

There are two main types of operational semantics: a big-step-semantics and a small-step-semantics. The former describes the entire calculation from one configuration γ to another γ' (denoted by $\gamma \rightarrow \gamma'$), whereas the latter describes the calculation from γ to γ' through a series of configurations (denoted by $\gamma \Rightarrow \gamma'' \Rightarrow \ldots \Rightarrow \gamma'$. The fact that DOGS will not support concurrency (cf. Section 6.11, page 50) has an impact on the choice of the type of operational semantics since a big-step-semantics renders modeling of parallelism impossible as commands in two concurrently executing threads can be interleaved. However, the exclusion of concurrency in DOGS allows for an operational big-step-semantics which is the type we have chosen.

We have already presented an abstract syntax in Section 8.4.1 when defining the type system in DOGS. This syntax will be important for the semantics. In the following we will present

- An environment-store-model, including
  - Help functions
  - Definitions of environments and stores
- Definitions of every transition systems, including
  - Definitions of configurations and end configurations
  - The transition relation defined by transition rules

We will, however, not present our formal semantics in its complete form as it will provide redundant information. Instead we have selected some transition rules that represent interesting charateristica of our semantics. The semantics for DOGS can be found in its entire form in Appendix E.

9.1 The Environment-Store-Model

The operational semantics in this chapter will make use of an environment-store-model based on the model from [Hüt04], but is expanded to handle more than one primitive type, multidimensional arrays, sets, graphs, and weights and labels.

9.1.1 Mathematical Shortcuts

Before explaining the appearance of our model we will introduce a notation and help functions that will prove useful in the semantics.

The elements in the *primitiveTypes* category are the names of the primitive types and the elements in the *compositeTypes* category are the names of the composite types.

$$\begin{aligned} primitiveTypes &= \{integer, string, float, boolean, vertex, edge\} \\ compositeTypes &= \{graph, digraph, array, set, label, weight\} \end{aligned}$$

In the abstract syntax we have the syntactical categories: int, str, float, and bool. Elements of int are, for instance, **-2**, **5**, **17**, etc. Elements of str can be **"abc"**, etc. Elements of float are **3.20**, etc, and elements of bool are limited to `true` and `false`. In addition we have the syntactical `infty` and `-infty`, and the literal type of these will be denoted $infty$ and $-infty$, respectively. We assume it possible to store these literals in our stores, thus leaving it the implementor of the semantics to make a suitable representation of these "values".

The underlined elements in the *primitives* category are the actual data representations of the types that are stored in the store tables.

$$primitives = \{\mathbb{Z}, \underline{string}, \underline{float}, \underline{bool}, Loc, Loc \times Loc\}$$

We will assume that there is a semantic function defined for int, str, float, and bool:

$$LittType : \{int, str, float, bool\} \rightarrow \{\mathbb{Z}, \underline{string}, \underline{float}, \underline{bool}\}$$

and correspondingly an inverse function:

$$LittType : \{\mathbb{Z}, \underline{string}, \underline{float}, \underline{bool}\} \rightarrow \{int, str, float, bool\}$$

For example we can use the $LittType$ function to get the actual integer number 10 from the syntax **10**: $LittType(\mathbf{10}) = 10$. Elements from $\underline{bool}$ (the values of the syntax `true` and `false`) are denoted tt and ff.

Likewise we assume that there is a semantic function $floatValue$ for converting a number (integer or float) value to a float value.

$$floatValue : \{\underline{integer}, \underline{float}\} \rightarrow \underline{float}$$

The function $new : sto \rightarrow Loc$ returns the free location following the last used location in the store sto.

$$new(type) = size(sto_{type}) + 1$$
where $type = primitiveTypes \cup compositeTypes \cup \{graphProp\}$

sto is a collection of all type-specific stores in our semantics:

$$\begin{aligned} sto \quad = \quad & \{sto_{integer}, sto_{float}, sto_{string}, sto_{boolean}, sto_{vertex}, sto_{edge}, sto_{set}, \\ & sto_{array}, sto_{graph}, sto_{graphProp}, sto_{label}, sto_{weight}\} \end{aligned}$$

To express that a specific store has been changed while all other stores in sto remain unchanged, we use the notation

$$sto[sto_{type}[\ldots]]$$
where $type = primitiveTypes \cup compositeTypes \cup \{graphProp\}$

The recursive function $\Gamma(sto, Loc, int) = Loc'$ is used when searching for a location of an element in the stores $Sto_{graph}, Sto_{graphProp}, Sto_{set}, Sto_{label}$, and Sto_{weight}. The sto parameter is one of these stores, Loc is the start location of the search, and int is the number of locations to seek forward.

$$\Gamma(sto, Loc, int) = \begin{cases} Loc & \text{if } int = 0 \\ \Gamma(sto, Loc'', int - 1) & \text{where } (-, Loc'') = sto(Loc) \text{if } int > 0 \end{cases}$$

Sto_{graph}, Sto_{set}, Sto_{label}, and Sto_{weight} return a 2-tuple (v, l) where v is the value in the store and l is the location of the next value. Often we only need to change one of the two elements in the 2-tuple and keep the other unchanged. This will be denoted by $(v, -)$, or $(-, l)$ as seen in the definition of Γ, meaning that l will remain unchanged or v will remain unchanged respectively.

9.1.2 Environments

The most simple environment in our environment-store-model is the variable environment Env_V. In this environment variables are bound to a type and a memory location (Loc). These locations will be considered as integers in our semantics, which means we can add and substract locations, as if they were integers, to get new locations.

$$Env_V = Var \rightharpoonup (type_N \times Loc) \cup Loc$$

where $type_N = primitiveTypes \cup compositeTypes$. Note that the set Env_V of variable-environments is the set of partial functions from variables to $(type, l)$. The reason for it being partial functions is that we abstract from actual storage limitations in the model and deal with an unlimited number of locations for a finite number of variables. This argument goes for the rest of the environments as well. We will denote an element from Env_V with env_V.

The Env_C environment is the environment of global constants. We need this environment in order to seperate the global constants with the local variables and constants in Env_V.

$$Env_C = Var \rightharpoonup (type_N \times Loc) \cup Loc$$

We have defined two sets of environments for handling records in our semantics: Env_{RT} for the defined record types and Env_{RV} for the actual record variables. A record can contain variables and nested records and the following definitions cover this:

$$\begin{aligned} Env_{RT} &= RecordType \rightharpoonup Env_V \times Env_{RV} \\ Env_{RV} &= RecordName \rightharpoonup Env_V \times Env_{RV} \end{aligned}$$

A record-type contains which variables and records a specific type consists of. This is represented by Env_V and Env_{RV}, which also bind the content of the record type to some default values. When a record variable is declared it uses Env_{RT} to find out what it must consist of. Actually, the only thing needed to create a record variable is a copy of Env_V and Env_{RV} from Env_{RT}, followed by a rebinding of all the variables in Env_V to new memory locations (so the record and the record-type content do not share locations, which would result in conflicts). Elements from Env_{RT} and Env_{RV} are denoted by env_{RT} and env_{RV} respectively.

Finally, we have the function and the procedure environments which only differentiate from each other in the fact that we need to store the return type of a function. This return type is either a primitive, a set, an array, a graph, or a record type. In accordance with our selected scope rules (cf. Section 6.9, page 49) we need to store which variables and record variables a function and a procedure have access to. In addition we have a parameter environment (Env_{PAR}) that stores information about the formal parameters of a function or a procedure.

$$\begin{aligned} Env_F &= Func_N \rightharpoonup S \times DecVar \times \\ &\quad (type_N \cup RecordType) \times Env_{PAR} \\ Env_P &= Proc_N \rightharpoonup S \times DecVar \times Env_{PAR} \\ Env_{PAR} &= \mathbb{N} \rightharpoonup Var \times type_N \times \underline{bool} \end{aligned}$$

where $type = primitiveTypes \cup compositeTypes$. The formal parameters of a function or a procedure is ordered, thus the parameter environment uses $\mathbb{N}$ to index the parameters. The Env_{PAR} stores the variable name of the parameter, its type, and whether it is a reference parameter (in that case the value of the boolean is) or not (the boolean is then set to *ff*). The variable declaration block is stored in *DecVar* so

that on every function or procedure call the variables are declared and their values are stored in locations in the proper stores as the first thing. Elements from Env_P and Env_F are denoted by env_P and env_F respectively, and elements from Env_{PAR} are denoted by env_{PAR}.

9.1.3 Stores

Our environment-store-model makes use of several stores; one store for each primitive type and composite type and in addition $sto_{graphProp}$ which links a graph with its weights and labels. Each store has its own set of locations (represented by non-negative numbers in $\mathbb{Z}$) and each store automatically keeps track of its size, i.e. the number of locations that are used to store information. This integer value is represented by $size(sto_{string})$, for example. We ensure it possible to retrieve the size of a specific store by making the domains of the stores finite. This is denoted by $\rightharpoonup_{fin}$. The functions are partial because the number of locations used is finite but can theoretically hold any value (e.g. $sto_{integer}$ can return any value in $\mathbb{Z}$).

The store for a primitive type simply stores its primitive values in one location for each value. Note that Loc and $Loc \times Loc$ are considered as primitive values (the values representing vertex and edge respectively).

$$\begin{aligned} Sto_{integer} &= Loc \rightharpoonup_{fin} \mathbb{Z} \\ Sto_{float} &= Loc \rightharpoonup_{fin} \underline{float} \\ Sto_{boolean} &= Loc \rightharpoonup_{fin} \underline{boolean} \\ Sto_{string} &= Loc \rightharpoonup_{fin} \underline{string} \\ Sto_{vertex} &= Loc \rightharpoonup_{fin} Loc \\ Sto_{edge} &= Loc \rightharpoonup_{fin} Loc \times Loc \end{aligned}$$

Elements from these stores are denoted in the same way, only non-capitalized (e.g. an element from Sto_{vertex} is denoted by sto_{vertex}).

To store arrays of primitives we use Sto_{array}, in which the primitive values are stored directly in the locations. In the first location a reference to the first location in the array block is stored. This reference location has been found necessary because array variables can be bound to new array blocks by assigning one array to another.

In the array block the first location stores the number of dimensions ($dims$), followed by $dims$ locations with the sizes of the dimensions ($size_1$, $size_2, ..., size_{dims}$). The actual content of the array is stored in the following $size_1 \cdot size_2 \cdot ... \cdot size_{dims}$ locations. The store function is defined as follows.

$$Sto_{array} = Loc \rightharpoonup_{fin} primitives$$

Elements from Sto_{array} are denoted by sto_{array}.

Sets differentiate from arrays as they are dynamic sized. They can be considered as kinds of linked lists and are stored by storing in the first location the size of the set and a pointer to the location storing the first value in the set. Similarly, in this location a pointer to the location of the next value is stored in addition to the value,

etc. In the location of the last value in the set a *nil* pointer is stored besides the value itself.

$$Sto_{set} = Loc \rightharpoonup_{fin} primitives \times Loc$$

Elements from Sto_{set} are denoted by sto_{set}.

In resemblance to a set, a graph is stored as an integer representing the number of vertices (the *size*), a location to make a reference to the graph-properties store (the store that connects a graph with labels and weights), and a pointer to the location of the first vertex where the name of the vertex is stored as a string. In this location a pointer to the location of the next vertex is stored in addition to the string name of the vertex, etc. In the location of the last vertex the pointer points to the location that contains the first entry in the $size^2$ adjencency matrix representation (described in Section 2.1.2, page 19) of the graph (the matrix consists of $0's$ and $1's$ depending on connections between vertices). Similarly, besides the value in this location a pointer is stored which points to the location containing the next entry in the matrix, etc. The same store is used for both non-directed graphs and directed graphs and this store can be referred to by Sto_{graph} or $Sto_{digraph}$.

As mentioned the $Sto_{graphProp}$ is used for connecting the graph with its labels and weights and is organized with the number of properties (that is, labels and weights) in the first location, a *type* denoting which of the two types *label* and *weight* is in question, and additionally a pointer to the first location storing references to the label and weight stores (stored as *Loc*). In this location a pointer to the next location storing a reference to the label or weight store is stored along with the actual label and weight store references.

$$\begin{aligned} Sto_{graph} &= Sto_{digraph} = Loc \rightharpoonup_{fin} \{\mathbb{Z}, string, Loc\} \times Loc \\ Sto_{graphProp} &= Loc \rightharpoonup_{fin} \{\mathbb{Z}, Loc\} \times type_N \times Loc \end{aligned}$$

Elements from Sto_{graph} and $Sto_{digraph}$ are denoted by sto_{graph} and $sto_{digraph}$ respectively, and elements from $Sto_{graphProp}$ are denoted by $sto_{graphProp}$.

Finally, we have the label and weight stores. The Sto_{label} is similar to Sto_{set}, the difference being that it holds a reference to a graph instead of the number of values stored (this number of values equals the number of vertices in the graph) along with a pointer to the location of the label of the first vertex in the graph. In the location of this label a pointer which points to the location of the label of the next vertex is also stored, etc.

Sto_{weight} is used to store information related to the edges and is therefore organized so that the reference to a graph is stored in the first location ($graphLoc$) along with a pointer to the location of the first weight. In this location a pointer to the location of the next weight is stored and so forth ($size^2$ locations with the actual weights ($size = sto_{graph}(graphLoc)$)).

$$\begin{aligned} Sto_{label} &= Loc \rightharpoonup_{fin} primitives \times Loc \\ Sto_{weight} &= Loc \rightharpoonup_{fin} primitives \times Loc \end{aligned}$$

9.1.3.1 Usage of stores

In this section we briefly describe how we will make use of our defined stores in our semantics, i.e. how we can use our stores to access specific data, such as an element in an array, or a connection between two vertices in a graph.

Graphs: A graph is organized with its number of vertices in the first location. A graph variable is bound to such a location and the number of vertices in the graph can therefore be accessed simply by looking up the value of the variable-location in the graph store. The graph properties reference is stored in the following location (i.e. the location that the pointer in the first location points to) and can therefore be accessed via the pointer to this location.

The following list shows how information can be retrieved from the graph store (l is the first location in the graph and the location that a graph variable is bound to):

- The number of vertices in the graph (n), and the location of the graph's properties *propLoc*:

$$(n, propLoc) = Sto_{graph}(l)$$

- The vertices of the graph:

$$(vertex_i, -) = \Gamma(Sto_{graph}, propLoc, i) \qquad \text{where } i = 1, 2, ..., n$$

- A connection c (1 or 0 depending on whether there is a connection or not, respectively) between two vertices v_1 and v_2 in the graph (v_1 is stored in location l_1 and v_2 in l_2):

$$\begin{gathered} i \text{ for which } \Gamma(sto_{graph}, propLoc, i) = l_1 \text{ and } 0 < i < n \\ j \text{ for which } \Gamma(sto_{graph}, propLoc, j) = l_2 \text{ and } 0 < j < n \\ c = \Gamma(sto_{graph}, propLoc, n + (i-1) * n + j) \end{gathered}$$

- If the graph is non-directed the graph matrix is symmetric and the equivalent connection between the two vectices can be accessed by swapping i and j in the connection equation.

Sets: The size of a set is stored in its first location *sLoc*. A set variable will reference to this first location. The data in a set is retrieved by:

- The size n of the set:

$$(n, -) = sto_{set}(sLoc)$$

- The values v in the set:

$$(v_i, -) = \Gamma(sto_{set}, sLoc, i) \qquad \text{where } i = 1, 2, ..., n$$

Labels: The reference to its graph is stored in the first location $lLoc$ of the label function. The number of vertices (elements in the label function) can therefore be retrieved by looking up in sto_{graph} on the location of the graph.

- The location $gLoc$ of the label's graph:

$$(gLoc, -) = sto_{label}(lLoc)$$

- The number of elements n in the label function:

$$(n, -) = sto_{graph}(gLoc)$$

- The values v of the elements in the label function (ordered in the same way as the vertices of the label's graph).

$$(v_i, -) = \Gamma(sto_{label}, lLoc, i) \qquad \text{where } i = 1, 2, ..., n$$

Weights: The reference to its graph is stored in the first location $wLoc$ of the weight function. The number of vertices noV can be retrieved by looking up in sto_{graph} on the location of the graph and the number of elements in the weight function equals noV^2.

- The location $gLoc$ of the weight's graph:

$$(gLoc, -) = sto_{label}(wLoc)$$

- The number of elements n in the weight function:

$$\begin{aligned} (noV, -) &= sto_{graph}(gLoc) \\ n &= noV^2 \end{aligned}$$

- The values v of the elements in the weight function (ordered in the same way as the edges of the weight's graph).

$$(v_i, -) = \Gamma(sto_{weight}, wLoc, i) \qquad \text{where } i = 1, 2, ..., n$$

Arrays: As mentioned, an array variable is bound to a location $aLoc$ that references to the first location $bLoc$ of the block. In this location the number of dimensions is stored which describes the following number of locations that contain the size of the different dimensions. The data in an array is retrieved by:

- The location of the "array block":

$$bLoc = sto_{array}(aLoc)$$

- The number of dimensions dim of the array:

$$dim = sto_{array}(bLoc)$$

- The size of the different dimensions $size_i$:

$$size_i = Sto_{array}(bLoc + i) \qquad \text{where } i = 1, 2, ..., dim$$

- A value v in the array $A[i_1][i_2]...[i_{dim}]$:

$$\begin{aligned} v = \quad & Sto_{array}(bLoc + dim + ((i_1 - 1) \cdot (size_2 \cdot size_3 \cdot ... \cdot size_{dim}) \\ & +(i_2 - 1) \cdot (size_3 \cdot ... \cdot size_{dim}) + ... + i_{dim})) \end{aligned}$$

[D_{VB}-block]

$$\frac{env_C, env_F, env_{RT}, env_P \vdash \langle D_{VB1}, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto'') \quad env_C, env_F, env_{RT}, env_P \vdash \langle D_{VB2}, env''_V, env''_{RV}, sto''\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}{env_C, env_F, env_{RT}, env_P \vdash \langle D_{VB1} D_{VB2}, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}$$

[D_{VR}-rec-block]

$$\frac{env_C, env_F, env_{RT} \vdash \langle D_{VR1}, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto'') \quad env_C, env_F, env_{RT} \vdash \langle D_{VR2}, env''_V, env''_{RV}, sto''\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}{env_C, env_F, env_{RT} \vdash \langle D_{VR1}, D_{VR2}, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}$$

Table 9.1: Transition rules for variable declarations in blocks and records

9.2 Transition Systems in DOGS

The abstract syntax and our environment-store-model have qualified us in presenting a selection of our semantics. We will explain parts from declarations, expressions, and commands.

Conferring [Hüt04, page 38], a transition system is defined in the following way:

Definition 4 (Transition systems). *A transition system is a tripel* $(\Gamma, \rightarrow, T)$ *where* Γ *is a set of configurations,* $\rightarrow$ *is the transition relation, and* $T \subseteq \Gamma$ *is a set of end configurations.*

This definition will be used when defining the transition systems for the syntactical categories.

9.2.1 Declarations

In the following sections we will present transition rules for declarations of non-composite types and composite types.

9.2.1.1 Declarations of Primitives

Our semantics for variable declarations is the transition system $(\Gamma_{DV}, \rightarrow_{DV}, T_{DV})$ where the configurations are given by

$$\begin{aligned} \Gamma_{DV} &= (DecVar \times Env_V \times Env_{RV} \times Store) \cup (Env_V \times Env_{RV} \times Store) \\ T_{DV} &= (Env_V \times Env_{RV} \times Store) \end{aligned}$$

This means that transitions are on the form $\langle D_V, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')$, given bindings in the global constants, function, record-type, and procedure environments.

[Var-dec-init]

$$\frac{env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ x\mathbf{;}\ env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle x\ \mathbf{:=}\ exp\mathbf{;}, sto''\rangle \rightarrow sto'}{env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ x\mathbf{:=}exp\mathbf{;},\ env_V, env_{RV}, sto\ \rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}$$

[RefPar-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ \texttt{ref}\ x\mathbf{:=}X\mathbf{;}\quad env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env_V[x \mapsto (type_N, l)], env_{RV}, sto)$$

where $env_V, env_{RV} \vdash X \rightarrow (type_N, l)$

$type_N \in primitiveTypes \cup \{\text{array of } type'_N, \text{set of } type'_N, \text{label of } type'_N, \text{weight of } type'_N\}$

Table 9.2: Transition rules for variable and reference declarations and initialization

In Table D.18 transition rules for variables in blocks and records are given. [D_{VB}-block] expresses that we declare successive variables by first declaring D_{VB1} which changes the variable environment, record-variable environment, and store, and then declaring D_{VB2} in these changed environments and store resulting in the final variable environments and stores. [D_{VR}-rec-block] expresses the same, the difference being that the variables are a part of a record and are therefore syntactically separated with a comma.

The declaration and initialization of a single variable or reference variable is covered in Table 9.2. [Var-dec-init] says that a variable x of type $type_N$ is declared, and initialized to the value of exp, by first declaring it using the proper declaration rule (depending on $type_N$, e.g. the rule [Var-dec] found in E.4 if $type_N \in primitiveTypes$), and initializing it to exp using the proper assignment command (also depending on the $type_N$).

As to reference declaration and initialization (in relation to the declaration of parameters) [RefPar-dec] expresses that we declare a (non-record) reference variable x of type $type_N$ using the keyword `ref` by binding x to its type and the location l of the variable X that we want x to be a reference to. sto remains the same in the process as no expressions are evaluated.

9.2.1.2 Graph Declarations

Table 9.3 shows the transition rules for declaring an undirected graph and a directed graph.

[Graph-dec] tells us to declare a graph variable x using the keyword `graph` by binding x to a free location l in env_{graph} using new. In sto_{graph} l is bound to $(0, l+1)$, the first value representing the number of vertices in the graph which is 0 in the moment of declaration, the second value being the next free location in

[Graph-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle \texttt{graph}\ x, env_V, env_{RV}, sto \rangle \rightarrow_{DV} (env_V[x \mapsto (graph, l)], env_{RV}, sto[sto_{graph}[l \mapsto (0, l+1)] [l+1 \mapsto (l', nil)], sto_{graphProp}[l' \mapsto (0, integer, nil)]])$$

where $l = new(sto_{graph})$, $l' = new(sto_{graphProp})$

[DiGraph-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle \texttt{diGraph}\ x, env_V, env_{RV}, sto \rangle \rightarrow_{DV} (env_V[x \mapsto (digraph, l)], env_{RV}, sto[sto_{graph}[l \mapsto (0, l+1)] [l+1 \mapsto (l', nil)], sto_{graphProp}[l' \mapsto (0, integer, nil)]])$$

where $l = new(sto_{graph})$, $l' = new(sto_{graphProp})$

Table 9.3: Transition rules for graph declarations

[Equals1]

$$\frac{envC, envF, envV, envRV, envRT, envP \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad envC, envF, envV, envRV, envRT, envP \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{envC, envF, envV, envRV, envRT, envP \vdash \langle n_1 = n_2, sto\rangle \rightarrow (tt, sto')}$$

$$\text{where} \quad v_1' = v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$$

Table 9.4: Transition rule for equality between two floats or integers

sto_{graph}. This value, $l + 1$, is bound to the 2-tuple (l', nil) where l' denotes a free location in $sto_{graphProp}$ and nil denotes that no vertices are connected to the graph yet. In $sto_{graphProp}$ l' is bound to $(0, nil)$ since no labels or weights are related to the graph. Every other store remain unchanged.

The same happens in [DiGraph-dec], the only difference is that we use the `diGraph` keyword and that the type of the variable is bound to *digraph* instead of *graph*.

9.2.2 Expressions

Our semantics for expressions is the transition system $(\Gamma_{exp}, \rightarrow_{exp}, T_{exp})$ where the configurations are given by

$$\begin{aligned} \Gamma_{exp} &= (exp \times Store) \\ T_{exp} &= (Value \times Store) \end{aligned}$$

Transitions are therefore on the form $\langle exp, sto\rangle \rightarrow_{exp} (v, sto')$, given bindings in the global constants, function, variable, record-variable, record-type, and procedure environments.

Table 9.4 shows a single of the transition rules for boolean expressions, [Equals1]. It expresses equality between two numbers covering floats and integers (in accordance with our design decision from Section 6.2). The evaluation of this expression is done by first evaluating n_1 to a value v_1 which changes sto to sto''. In this sto'' n_2 is evaluated to v_2 which changes sto'' to sto'. In the clause the conditions for this transition are that the output of the help function $float$ on v_1 and v_2 are respectively the floats v_1' and v_2' which have to be mathematically equal.

[Label-val] in Table 9.5 expresses how we retrieve a label value bound to a vertex. X represents a label variable which is seen in the clause where X evaluates to the type *label* and a location $lLoc$. In the premise, exp evaluates to $vLoc$ which is a vertex value (a value of type Loc) and a changed store sto'.

Everything happens in the clause in this transition. We use $lLoc$ from the evaluation of X to yield the location of the graph which the label is connected to, denoted

[Label-val]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (vLoc, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X(exp), sto\rangle \rightarrow_{exp} (v, sto')}$$

where
$$\begin{aligned} & env_{RV}, env_V \vdash X \rightarrow (label, lLoc) \\ & (graphLoc, -) = sto'_{label}(lLoc) \\ & (noVertices, propLoc) = sto'_{graph}(graphLoc) \\ & vNo \ for \ which \ \Gamma(sto'_{graph}, propLoc, vNo) = vLoc \\ & \quad and \ 0 < vNo \leq noVertices \\ & (v, -) = sto'_{label}(\Gamma(sto'_{label}, lLoc, vNo)) \end{aligned}$$

Table 9.5: Transition rule for the label value expression

by $graphLoc$. A look-up in sto'_{graph} on the location of the graph, $graphLoc$, yields the number of vertices that the specific graph consists of, and the location $propLoc$. We use Γ to find the number vNo of the vertex at location $vLoc$ we wish to retrieve the label value of, by searching forward in the sto'_{graph} at $propLoc$ until we get to the location of the vertex $vLoc$. vNo is the number of vertices we had to seek forward to get to the location $vLoc$. Further, we condition that vNo has to be a vertex in the graph in question which is taken into account by $0 < vNo \leq noVertices$. Finally, we retrieve the value v by using Γ on sto'_{label} to seek vNo locations forward from the location of the label $lLoc$ and looking up in sto'_{label} on the location from Γ.

[Func-Call-Prim] expresses how we call a function with a return value that is either a primitive, a graph, digraph, array, or set in DOGS (seen in the clause). $Func_N$ is called with the actual parameter sequence Par_A (evaluated in the transition rules [Par$_A$] and [Par$_A$-ref] found in Appendix E, Table E.56 on page 231). The parameter rules are on a speciel form so that the actual parameters are evaluated as expressions in the environments known when the function is called, but are bound to variables in the environments of the function invoked.

A look-up in env_F with $Func_N$ passed gives us $Func_N$'s command S, its declaration block DV, its return type $type_N$, and the parameter environment env_{PAR} bound to it.

In the premise, $temp[i \mapsto 0]$ is used to keep track on the sequence of actual parameters in relation to the formal parameters and is vital when evaluating the actual parameters Par_A (see Section 9.1.1). This creates variable environments env'_V and env'_{RV} and the store sto'', with the formal parameters of the function bound to the values of the actual parameters. In these environments the declaration block DV is evaluated, which results in env''_V and env''_{RV} and the store $sto^{(3)}$.

The command S of the function is executed in these changed environments (the environments after evaluating both the actual parameters and the variables in the `let-in` block), but with the reserved keyword $returnvalue$ bound to $(type_N, l)$, and,

[Func-Call-Prim]

$$\frac{\begin{array}{ll} env_{PAR}, env_C, env_V, env_{RV}, env_{RT}, env_F, temp[i \mapsto 0] \vdash & \langle Par_A, \emptyset, \emptyset, sto\rangle \rightarrow_{ParA} \\ & (env'_V, env'_{RV}, sto'') \\ env_C, env_F, env_{RT}, env_P \vdash \langle DV, env'_V, env'_{RV}, sto''\rangle \rightarrow_{DV} & (env''_V, env''_{RV}, sto^{(3)}) \\ env_C, env_F, env''_V[returnvalue \mapsto (type_N, l)], & env''_{RV}, env_{RT}, env_P \vdash \langle S \\ & , sto^{(3)}[sto_{type_N}[l \mapsto nil]]\rangle \rightarrow sto' \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle Func_N(Par_A), sto\rangle \rightarrow_{exp} (v, sto')}$$

where $env_F(Func_N) = (S, DV, type_N, env_{PAR})$
$type_N \in primitiveTypes \cup \{graph, digraph, array, set\}$
$l = new(sto_{type_N})$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle returnvalue, sto'\rangle \rightarrow_{exp} (v, sto')$

Table 9.6: Transition rules for function calls on primitives

of course, in the changed store. In the clause it is noted that l is the next free location in the store determined by $type_N$ which is the return type looked up in the function environment.

During execution of S *returnvalue* is eventually assigned to a value by the return command (E.26), the type of which can be a primitive, graph, digraph, array, or set, i.e. a non-composite or a composite value. Execution of S results in a changed store sto', which, together with the value v, is the final result of the function call expression. v is retrieved by evaluating the *returnvalue* variable in sto'. If it is a primitive, v is a concrete value stored in the location of *returnvalue*. If it is a composite, v is the location of the *returnvalue* variable. This is accomplished by using the transition rules for [Var-val] and [Var-val-composite] found in Appendix E, Table E.50 on page 225.

9.2.3 Commands

Our semantics for commands is the transition system $(\Gamma, \rightarrow, T)$ where the configurations are given by

$$\begin{aligned} \Gamma &= (S \times Store) \\ T &= Store \end{aligned}$$

This means that transitions are on the form $\langle S, sto\rangle \rightarrow sto'$, given bindings in the global constants, function, variable, record-variable, record-type, and procedure environments.

Table 9.7 illustrates the transition rule for composite commands, [Comp]. It specifies that the composite command $S_1 S_2$ is executed by executing S_1 first, which changes sto to sto'', and then executing S_2 in sto'' which changes to sto'' sto'.

[Comp]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_1, sto\rangle \rightarrow sto'' \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_2, sto''\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_1 S_2, sto\rangle \rightarrow sto'}$$

where $env_V(return) \rightarrow (boolean, l)$
$sto''_{boolean}(l) = ff$
$env_V(break) \rightarrow (boolean, l)$
$sto''_{boolean}(l) = ff$

Table 9.7: Transition rule for composite commands

[While-true]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto''\rangle \rightarrow sto^{(3)} \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, sto^{(3)}\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, sto\rangle \rightarrow sto'}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto\rangle \rightarrow_b (tt, sto'')$
$env_V(break) = (boolean, l)$
$sto^{(3)}_{boolean}(l) = ff$

Table 9.8: Transition rules for a `while..do`-loops

Both sequences of commands can contain the `return` command. However, this rule tells us that the predefined *return* variable in env_V evaluates to a boolean type and a location l and that a look-up in sto'' on l has to yield the value ff in order to execute S_2. This is also the case for the predefined *break* variable.

[While-true] in Table 9.8 specifies how the `while` loop is executed in DOGS. First, b is evaluated to a value and provided that it evaluates to tt this rule is used. S is then executed in sto'' (sto has changed due to the evaluation of b) which changes it to $sto^{(3)}$. The loop is then executed again in this new store.

As with [Comp] it is conditioned that the *break* variable is set to ff in order to use this rule. Note that if S_1 in [Comp], for instance, consists of a loop containing the `break` command, we have to ensure that the value of the *break* variable is set to ff once we leave the loop, otherwise S_2 is never executed (consistent with the transition rule [Comp-return] in Appendix E, Table E.26 on page 206). This can be seen in the rule [While-true-break] in Appendix E, Table E.24, page 204).

[For-each-set] found in Table 9.9 describes what happens in the `for-each` loop in DOGS. Informally, it executes one or several commands on each element in a set.

The clause X is evaluated to the type *set* and a location l. The size of the set is

[For-each-set]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{foreach}\ \ type_N\ x\ \texttt{in}\ X\ \texttt{do}\ S, sto\rangle \rightarrow sto^{(size)}[sto_{boolean}[l' \mapsto ff]]$$

where
$env_{RV}, env_V \vdash X \rightarrow (set, l)$
$env_V(break) = (boolean, l')$
$(size, -) = sto_{set}(l)$
$env_C, env_F, env_V[x \mapsto (type_N, l_i)], env_{RV}, env_{RT}, env_P \vdash \langle \texttt{if}\ (return = \texttt{false}\ \texttt{and}\ break = \texttt{false})\ \texttt{then}\ S, sto^{(i-1)}\rangle \rightarrow sto^{(i)}$
$i = 1, 2, \ldots, size$
$sto^{(0)} = sto$
$l_i = \Gamma(sto_{set}, l_{i-1}, 1)$
$l_0 = l$

Table 9.9: Transition rules for `for-each` loops for sets

retrieved by making a look-up in sto_{set} on l which is used in the following conditions. Given bindings in env_F, env_V, except that each element x in the set is bound to a type determined by $type_N$ and a location l_i which is determined by using Γ on sto_{set}, the location value found in the store of the previous element, and 1 as offset, env_{RV}, env_{RT}, and env_P, we execute S as long as the *return* and *break* variables are set to false. The trick is that S can consist of a return command or a break command which entails that the following loop iterations should not execute S or evalute b.

9.2.4 Standard environment

In the standard environment of DOGS several functions and procedures for working with graphs and sets can be found. Semantics have been worked out for these functions and procedures (expressions and commands respectively), as specifications of these are just as important as the specification of the syntax when having to implement DOGS.

In this section we present the semantics for the standard environment procedures `AddEdge` (graph) and `AddEdge` (diGraph) found in Table 9.10. The purpose of these procedures are to, given an edge value, add 1's in the graph store to represent a connection between to vertices, in undirected and directed graphs respectively.

In [AddEdge-graph] the two expressions exp_1 and exp_2 are the first thing to be evaluated. exp_1 is evaluated in sto to a graph location $gLoc$ and a changed store sto''. exp_2 is evaluated in sto'' to an edge primitive (two vertex locations) $vLoc' \times vLoc''$ and a changed store sto'.

The number of vertices $noVertices$ and the graph properties location $propLoc$ is retrieved by looking up in the graph store on the graph location $gLoc$. The vertex numbers of the two vertices in the edge is found by seeking forward with the Γ function in sto'_{graph} from the location $propLoc$, until the locations $vLoc'$ and $vLoc''$

are reached. The vertex numbers are represented by i and j, and we ensure in the clause that the two vertices are actually contained in the graph located at location $gLoc$ by the statements $0 < i < noVertices$ and $0 < j < noVertices$. Finally, in the clause, we find the proper edge locations $eLoc'$ and $eLoc''$ (locations in the matrix of the graph) by seeking forward using the Γ function and the vertex numbers i and j (as described in Section 9.1.3.1 on page 90).

The [AddEdge-graph] works on undirected graphs and therefore two 1's have to be added to the graph matrix (due to the symmetric nature of the undirected matrix). In the conclusion the bindings of the edge locations to the value 1 happens in the store sto_{graph} in sto'.

The semantics for adding an edge to directed graphs [AddEdge-diGraph] is very similar to the semantics for adding an edge to undirected graphs. The way we differentiate between working with edges with orientation or not is illustrated in the *addEdge* transition rules. When working with undirected graphs the order of the vertex references in an edge is of no interest, since (v_1, v_2) and (v_2, v_1) represent the same edge in the graph. When working with directed graphs, (v_1, v_2) and (v_2, v_1) represent two different edges and therefore we only determine a single edge location $eLoc'$ and bind this location to 1 in the transition rule [AddEdge-diGraph].

Summary

When reflecting on the process of defining semantics for DOGS, we can identify several problems. The environment-store-model and the transition rules are very detailed and low level in nature, which has some drawbacks. First of all, the complete semantics for DOGS is very large and can therefore be a bit confusing and hard to grasp. Some of the more complex operations in DOGS were also very hard to describe in this low level model. Having decided to implement our language in JVM, it can be questioned whether this level of detail is really necessary. Our semantics does indeed define the behaviour of programs written in DOGS, and does therefore work as an implementation guide, but JVM provides a much higher level of abstraction than our environment-store-model (e.g., the use of objects). This can result in difficulties when wanting to ensure that our implementation (code generation) actually corresponds to our semantics. On the other hand we leave the possibility of implementing DOGS to another target machine open.

Another problem with our environment-store-model is that several of our transition rules in the different semantical categories have knowledge of a lot of environments they do not actually make use of. As an example, expressions are evaluated given bindings in $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P$, although most expressions do not use anything but the variable environments. In this case, the reason for having knowledge to all these environments is that a function call is an expression, and that such a call has to make declarations and do commands to evaluate to its return value, thus expressions need to know of these environments. This lowers the readability of parts of our semantics, and a modification of the environment-store-model (e.g., the function environment) might improve it. This could be done by moving some of the environments, that only the function-call expression needs knowledge of,

into the function environment itself, and then do a look-up when these environments are needed.

[AddEdge-graph]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle addEdge(exp_1, exp_2);, sto\rangle \rightarrow sto'[sto_{graph}[eLoc' \mapsto 1][eLoc'' \mapsto 1]]$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc', sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc' \times vLoc'', sto')$
$sto'_{graph}(gLoc) = (noVertices, propLoc)$
i for which $\Gamma(sto'_{graph}, propLoc, i) = vLoc'$
j for which $\Gamma(sto'_{graph}, propLoc, j) = vLoc''$
$0 < i < noVertices$
$0 < j < noVertices$
$eLoc' = \Gamma(sto'_{graph}, propLoc, noVertices + (i-1) \cdot noVertices + j)$
$eLoc'' = \Gamma(sto'_{graph}, propLoc, noVertices + (j-1) \cdot noVertices + i)$

[AddEdge-diGraph]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle addEdge(exp_1, exp_2);, sto\rangle \rightarrow sto'[sto_{graph}[eLoc' \mapsto 1]]$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc', sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc' \times vLoc'', sto')$
$sto'_{graph}(gLoc) = (noVertices, propLoc)$
i for which $\Gamma(sto'_{graph}, propLoc, i) = vLoc'$
j for which $\Gamma(sto'_{graph}, propLoc, j) = vLoc''$
$0 < i < noVertices$
$0 < j < noVertices$
$eLoc' = \Gamma(sto'_{graph}, propLoc, noVertices + (i-1) \cdot noVertices + j)$

Table 9.10: Transition rules for `addEdge` from the Standard Environment

Summary

In this part we have documented a lot of decisions based on the analysis and we have presented an almost complete formal definition of DOGS based on the design criteria. The formal specification of DOGS consists of the three specifications: syntax, type system, and operational semantics. These specifications dictate the course of action in the implementation phase.

Part III

Implementation Document

Introduction

In this document we will discuss and account for choices made regarding the design of the compiler. We have purposely left out any specific documentation regarding the implementation of the compiler and included this in the design wherever necessary. We did this in the opinion that the implementation is contained within the design itself - the design reflects the implementation and vice versa. Yet, this is the implementation part because the development of a compiler is an implementation of a language.

This part consists of three different chapters that contain the choices that have been made specifically in order to be able to develop a compiler, the actual design of the compiler, and tests as well as results of these.

When describing the compiler design choices, we will emphasize the decision to choose SableCC to assist us in creating the front-end of our compiler. This includes discussing the consequences of this choice, while accounting for the framework, we are presented with by using SableCC.

The chapter regarding the actual design, focuses on the three main phases of a compilation and the design of these mechanisms in the compiler.

The tests we have conducted, will be accounted for in the last chapter in the document. We will illustrate them and argue that our compiler is correct, although these tests will be purely empirical.

We have chosen to let part the design document act as the analysis on which the development of the compiler is based. Thus, we are covering all the phases of a normal software development life-cycle, if we include the implementation that is covered by the design, according to [MMMNS00].

Chapter 10

Compiler Design Choices

In this chapter we will outline the different technologies on which the compiler is based, as well as the reasons for choosing these technologies. Choosing the target platform for the DOGS language is rather important, however, this seems more important when generating code than when designing the compiler as a whole. So, although this issue is considered in this chapter, the main focus is to determine how to develop the compiler. This is done by selecting a compiler-compiler tool to assist us in the development of `dogsc` and design the remainder of the compiler based on the "restrictions" imposed on us by the compiler-compiler tool.

10.1 Choosing a Virtual Machine

Choosing a target architecture for a compiler requires some consideration. If efficiency is prioritized, one should consider writing to low level architecture, such as the `x86`.On the other hand, if portability is desired, one should consider adding a level of abstraction to the target language by writing to a virtual machine. A virtual machine provides portability by abstracting from the hardware specific processor, since it is built on top of the operating system. As we give low priority to efficiency and wish to make DOGS available on many different architectures, a virtual machine is the best choice for our target language.

10.1.1 Java Virtual Machine

The Java Virtual Machine (JVM) is a stack based virtual machine, with native support for integers, floats, chars, and objects. It additionally supports input/output which makes it desirable as a target platform for DOGS. JVM is distributed by Sun Microsystems Inc [MI] and is widely used, thus making it a recognized and well tested virtual machine. Furthermore, we all have some experience with Java programming and thereby having some implicit experience with JVM. When using a virtual machine like JVM it is also possible to make use of its classloader library, which makes it easy to implement the parts of the language that can not be coded directly in DOGS, like basic input/output, available at runtime in DOGS. It can be implemented directly (through bytecode) for any kind of virtual machine, but it easily gets very complex, By taking advantage of it, it is possible to implement our

special graph-related types in Java and let the compiled DOGS programs make use of them. This mainly constitutes why we believe that we will be more successful in accomplishing our goals with the JVM as our target machine.

10.1.2 Triangle Abstract Machine

Triangle Abstract Machine [WB00, p. 178] (TAM) is a stack and heap based virtual machine with registers and with support for the data types: integer, boolean value, and character. It supports many standard features, however it is a virtual machine made specifically for educational purposes and supports only limited input/output functionality. In DOGS both the float and string types are included, and neither of them is directly supported by TAM. Strings are possible, though complicated to implement, due to the fact that TAM only provides characters. We do not want to remove the float primitive from our language, thus making TAM a poor target machine. Although the source code for TAM is free and written in Java, thus making it possible to change and extend it, we find it more valuable to spend our resources in other areas, rather than experimenting to extend TAM to fit our needs. All these factors contribute to our decision of choosing the JVM over TAM as the target platform for DOGS.

10.1.3 Common Language Runtime

The Common Language Runtime [Cor] (CLR) is another stack based machine and yet another alternative. In contradiction to JVM it is designed to support other source languages than Java. In CLR, weaknesses, such as JVM's opcodes[1], fixed storage size (32 bit), and only little support for references and pointers in JVM, have been eliminated [MM]. The main drawback of choosing CLR is that we deem that there are no reliable versions available at the moment except for the Windows platform. Keeping this in mind we judge that JVM is preferable over CLR.

The process of the `dogsc` compiler taking a program written in DOGS as input and converting it to JVM binary code is depicted in 11.5. This is done through a number of steps. The first step is to convert DOGS code to Java Assembler Interface (Jasmin) code [Mey] and the second step is to have the Jasmin compiler compile the program to JVM code, which then can be executed on a given machine M, through the JVM interpreter.

10.1.4 Assembler Interface

When compiling for JVM, there are roughly two ways of doing this, either by creating the bytecode for JVM directly, or by using an assembler interface, which also, can be translated directly to bytecode. The compiler will work faster when translating directly to bytecode, which is a tradeoff for the difficult debugging, due to the fact that a lot of opcodes (e.g. `0xC6`) in output from the compiler, can be hard to understand. When using an assembler interface, every instruction for the target platform, in this case JVM, is translated into readable opcodes, like `ifnull`. It is

[1] Instructions with similar functionality but different input types have different opcodes, e.g. iload, fload, aload,etc.

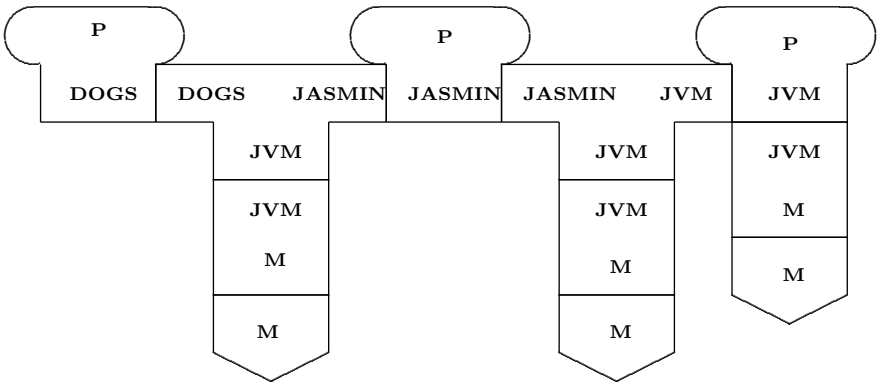

Figure 10.1: Tombstone diagram of the `dogsc` compiler

easier to survey the output from the compiler, thus making it easier when debugging. Since this is our first time making a compiler, we asses it to be an error-prone process generating bytecode, thus making debugging a necessary part of creating `dogsc`, and therefore, we have chosen to use an assembler interface when creating the bytecode for JVM.

There are a few different assembler interfaces for JVM, but they are very poorly documented which makes it very hard to choose. Our choice is Jasmin as the assembly interface used for our compilation, because it has been used since 1996 and came recommended by the lecturer in the "Languages and Compilers" course. Jasmin has some features which makes it easier to jump in code, e.g. when creating looping structures.

10.2 Compiler Passes

One main issue in compiler design is the issue of how many times the compiler should pass through the source code in order to produce an output. Generally, there are two options; either it is all done in a single pass or it is done in multiple passes. This is an important consideration and a well-founded decision needs to be taken before the actual compiler design can be commenced [WB00]. In fact, it is not even possible to choose a front-end tool to assist in the development of the compiler without knowing whether the compiler should be a multi-pass compiler.

Deciding whether to go with a single or a multi-pass compiler is, to a large extent, a tradeoff between different properties of the compiler. For instance, if speed of the compilation process is an issue, a single-pass compilation is definitely preferable, but if flexibility has higher priority, a multi-pass compiler should be used [WB00]. The mentioned properties are all related to the compiler itself, however, even the language, in our case DOGS, may have some properties that restrict the compilation of the language. If this is case there might be no choice but to go with a multi-pass compiler. If the language does not have these properties we can choose freely. Although properties like memory consumption (that also favors single-pass compilation) and speed are not as important to `dogsc` as flexibility, these properties are not determining factors when deciding whether to use a single or multi-pass com-

pilation. This is because DOGS actually do impose restrictions on the choice. In DOGS, functions and procedures can be declared in any arbitrary order, and they can be called from any other function or procedure. This property of DOGS requires the compiler to know of all functions and procedures before any type-checking can be done, i.e. `dogsc` must use one pass just to identify all the properties of each function and procedure, hence `dogsc` must be a multi-pass compiler. This is unlike a programming language like C, in which the programmer has to specify headers for all functions and procedures before the main procedure is declared.

10.3 Syntax Trees

When creating a multi-pass compiler it is necessary to store the structure of the source code being compiled, as it is only parsed once. This is often done as a tree structure, known as a parse tree or syntax tree. In this section two such trees will be presented; the *concrete syntax tree*, CST, and the *abstract syntax tree* AST.

10.3.1 Concrete Syntax Trees

Technically, CSTs are parse trees with exactly one node for each token. Consider the DOGS expression that consist of a simple multiplication (Listing 10.1):

```
1     ... 3 * 3 ...
```

Listing 10.1: An expression in DOGS

A CST for the DOGS arithmetic expression in Listings 10.1 is illustrated in Figure 10.2, and from this example it is clear that many nodes (and thereby tokens) are redundant and convey no useful information. It follows that such a parse tree may be inconvenient to use directly, i.e. in order to evaluate the expression, one has to traverse through more than fifteen nodes!

Often grammars need to be modified in order be able to be parsed as well as they often contain constructs that allow semantics to be part of the grammar (e.g. precedence rules that are part of the grammar). A major reason why so many seemingly useless nodes are traversed in the DOGS expression is exactly that many modifications were applied to the grammar in order to avoid ambiguities and to maintain rules of precedence. However, since the parse tree has already been built, all these extra nodes are not needed in order to perform the contextual analysis [App99].

Alternately, some of these modifications could be left out when constructing the parse tree, thus creating an AST.

10.3.2 Abstract Syntax Trees

As the name implies, an AST is generated from the abstract syntax of a grammar. An abstract syntax provides a clean mapping between the parser and later stages of the compilation by conveying the phrase structure of the program while ignoring the semantics of the language [App99]. An AST of the DOGS expression in Listings 10.1, based on the abstract syntax presented in Section 8.4.1, is shown in Figure 10.3.

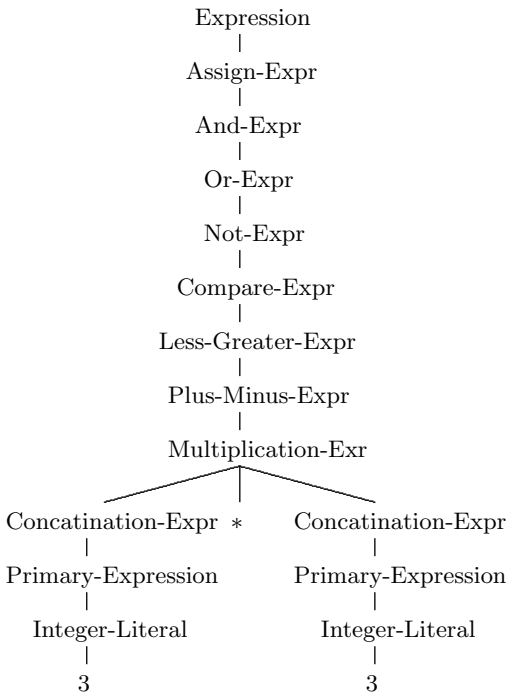

Figure 10.2: A CST for the DOGS example 10.1

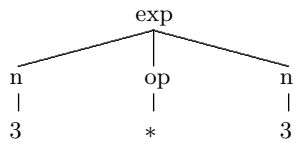

Figure 10.3: An AST for the DOGS example 10.1

The choice of syntax representation through a tree is related to the choice of compiler-compiler tool, which we will discuss in the next section.

10.4 Discussion of Compiler-Compiler Tools

A compiler consists of a front-end as well as a back-end. The front-end consists of a lexer (scanner), that generates tokens from source file, and a parser that checks the order of the tokens. As it appears both the lexer and parser have well-defined purposes and are both, to some extent, based on the grammar of a language.

The task of constructing these parts of the compiler is rather mechanical [Gagb, chapter 1, page 1]. Hence, there are tools that can assist in the generation of lexers and parsers. There are different tools available and although they basically perform the same basic tasks, differences still distinguish them from one another. Therefore it is important to make an aware choice as to what tools are to be used before designing the compiler. In the following we will present JavaCC [Inc], Jlex [BA] and CUP [Hud], and SableCC [Gaga], which are all tools for generating lexers and parsers. These presentations will be with emphasis on the tool of choice that is to

assist us in the development of the compiler for DOGS. We will also argue why this particular tool has been chosen over the other tools.

10.4.1 SableCC

We have chosen to use SableCC to construct the lexer and parser. One of the main reasons for this choice is that it is relatively easy to use due to the fact that the syntax of the input needed in order to generate both the parser as well as lexer is fairly simple compared to other available front-end tools. The input is in the form of a SableCC specification that contains the lexical definitions and the grammar productions (written in Extended Backus-Naur Form, EBNF) of the language to be recognized by the generated compiler framework [Gagb].

Another reason for choosing SableCC is that the grammar of our language is suspected to be a LALR grammar and proven to not be an LL grammar (see Section 7.3.3 on page 71), thus a recursive descent parser will not be able to recognize the DOGS language. SableCC provides a parser with a bottom-up parsing strategy, enabling it to recognize LR0, LR1, SLR, and LALR grammars and thereby making it a choice that is more than acceptable for our front-end [Tho, slides for lecture 5].

On top of all this, SableCC generates an object-oriented framework which builds an AST. In this framework the source code of the lexer and parser is isolated from the code generator or interpreter (that has to be build manually at a later stage). The fact that it provides an object-oriented framework should also give us the ability to easily comprehend the SableCC generated classes, since we all have experience from the object-oriented paradigm.

When using SableCC it is not necessary to include any action code in the grammar file. Instead, an extension of the Visitor pattern (discussed in Section 10.5.1) on the AST (which the parser generates) is used to add action code to the compiler or interpreter [Gagb]. The use of this pattern makes it relatively easy to debug the compiler since what has been written is easily identified, and in case the grammar works the error has to reside in the code. As an additional benefit it would suffice to recompile this one part of the compiler. Alternatively, a tool, in which the action code is placed in the grammar file, will typically implement the code in the source code of the parser. This makes debugging more difficult as it is a tedious task to identify the action code amongst the automatically generated code [Tho, slides for lecture 5].

10.4.1.1 The Sable Generated Syntax Tree

As previously mentioned, the compiler front-end, generated by SableCC, generates an AST in the parsing process. This AST, however, is basically equal to a CST, as it contains a node for each token in the file that is parsed. From section 10.3, it seems clear that this is not the optimal AST, so luckily SableCC provides means for modifying it.

One way to do this modification is to create a new class that inherits from the parser class generated by SableCC, and then let this new class modify the AST as it is constructed [Gaga].

Another way to modify the AST is by using functionality available on all nodes

in the AST. This method allows one subtree to be replaced by another and thereby changing the AST [Gaga].

SableCC has made the design decision that tree nodes can not be modified, only the relations between nodes can be altered. The advantage of this is that it is guaranteed that the tree will not be corrupted, however the drawback is that it is not possible to decorate the tree with references to other nodes or other information.

In short, using SableCC is a matter of knowing the Visitor design pattern, and because the action code is separated from the rest of the code it should be feasible to locate errors and correct them.

10.4.2 JLex and CUP

Alternatively, one tool can be used for building the scanner and another tool for building the parser. JLex and CUP are such tools, they build a lexer and a parser respectively. These tools can be used independently of each other, allowing different combinations of tools. Although CUP generates a parser based on LALR grammars [Hud] as needed by the DOGS language, this combination of tools has some drawbacks. First of all, it is necessary to have different input files for the two tools, both containing different action code. This is not a property that we find alluring. In fact, having two files instead of one makes this approach less desirable compared to a single file solution which, in our opinion, would be more comprehensible, especially because overlaps between the two different input files will occur (both JLex [BA] and Cup [Hud] input files must contain a list of tokens). Furthermore, as stated earlier, the fact that action code is embedded into these files decreases the comprehensibility of the generated code, thus rendering it more complex and difficult to debug.

Another important reason for choosing SableCC over the JLex and CUP solution is that CUP generates a parser, but this parser does not produce an AST, and there are, to our knowledge, no tools available to generate an AST with CUP. This means that CUP is for single-pass compilers and not, as the compiler for DOGS should be, for multi-pass compilers.

It may also be noted that the use of action code within the grammar, as in both CUP and JavaCC, is basically equivalent to using attribute grammars. Generally, attribute grammars have "gone out of fashion" [Tho, slides for lecture 4], which further makes SableCC as the front-end tool for DOGS the more obvious choice.

10.4.3 JavaCC

As to JavaCC we have identified some drawbacks in the tool. The first drawback is that JavaCC uses action code in its grammar file [Inc]. This, as mentioned in Section 10.4.1, makes it harder to debug. A second drawback is that JavaCC does not support automatic creation of an AST to be used as output from the parser. If an AST is wanted, another tool besides JavaCC has to be used [Inc] (unless, of course, one desires to separately write the AST-generating code), whereas this is supported in SableCC.

However, even if these drawbacks had not been rather severe issues, we still would not have used JavaCC due to the important property of DOGS grammar that it is

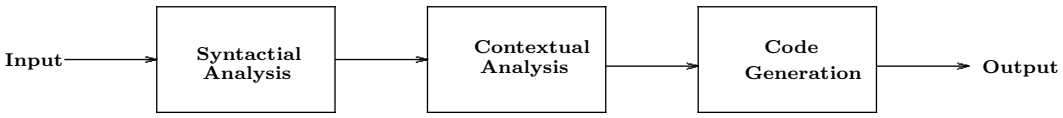

Figure 10.4: Compilation phases

a LALR grammar. Since JavaCC creates a recursive descent parser it would not be possible for JavaCC to create a parser for the DOGS grammar.

10.5 SableCC Framework

Choosing SableCC obviously has some consequences for the design of `dogsc`, because the parser is essential for the compiler and SableCC generates the parser used in `dogsc`. Not only does SableCC generate a parser, it also generates a framework that allows the compiler to pass through source code written in DOGS multiple times. This is done by creating an AST that, although equivalent with the corresponding CST, dictates the design of the rest of the compiler to a large extent. The basic framework created by SableCC is explained in the remainder of this section.

Developing a compiler with SableCC consists in ([Gagb, p. 22-23, ch. 3.2]):

1. Writing a grammar specification file (The DOGS grammar file can be found in Appendix C).
2. Running SableCC on the grammar file.
3. Writing worker classes inheriting from the SableCC generated framework that work on the CST .
4. Writing a main compiler class which uses the lexer, parser, and working classes from step 3.
5. Compiling the classes to compile the executable compiler.

SableCC generates a Deterministic Finite State Automaton (DFA) which recognizes the DOGS language, also known as a lexer, a LALR(1) parser, and a framework to build classes that work while walking the AST, known as workers. The framework generated consists of the four packages: analysis, node, lexer, and parser. These four packages include everything needed to construct a specialized tree walker. In order to explain how a tree walker is constructed we will summarize the Visitor pattern and the rest of the SableCC framework.

10.5.1 Visitor Pattern

When working on larger data structures, e.g. lists or trees with objects of different types, it can be difficult to gather information from the objects. The Visitor pattern provides a mechanism for gathering information from a number of different objects

without determining the actual type of the objects. Determining the type of a given object however is not a simple task.

A naive approach could be to do this through simple branching statements as illustrated in Code Listing 10.2. This, however, has got worst time complexity equivalent to the number of types and this is where the Visitor pattern comes into the picture.

```
1  void determineType(Type obj){
2    if (obj instanceof Type1){
3      System.out.println(``Type1'');
4    } else {
5      if (obj instanceof Type2){
6        System.out.println(``Type2'');
7      }
8  ...
9      else{
10       System.out.println(``TypeN'');
11     }
12   }
13 }
```

Listing 10.2: Determining types

The main idea is to move some functionality away from the objects and into a Switch interface. This new Visitor interface contains methods for visiting different objects, denoted `visitObject`, and takes an `Object` as parameter is listed in Code Listing 10.3.

```
1  interface Switch{
2    void visitType1(Type1 obj);
3    void visitType2(Type2 obj);
4    ...
5    void visitTypeN(TypeN obj);
6  }
```

Listing 10.3: the Switch Interface

```
1  abstract class Type{
2    abstract void accept(Switch sw);
3  }
4
5  class Type1 extends Type{
6    void accept(Switch sw){
7      sw.visitType1(this);
8    }
9  }
```

Listing 10.4: Classes of types

```
1  class TypeSwitch implements Switch{
2    void visitType1(Type1 obj){
3      System.out.println(``Type1'');
4    }
5    void visitType2(Type2 obj){
6      System.out.println(``Type2'');
7  }
8    ...
9    void visitTypeN(TypeN obj){
10     System.out.println(``TypeN'');
11   }
12 }
```

Listing 10.5: The TypeSwitch

```
1  String determineType(Type obj){
2    obj.accept(new TypeSwitch());
3  }
```

Listing 10.6: Accepting the TypeSwitch

In order to create a Visitor, the interface providing the `visitObject` methods is simply implemented as shown in Code Listing 10.5.

The object classes must have a specialized `accept` method that takes a Visitor as argument. The `apply` method will call the appropriate method on the Visitor object. Thus, it is possible to call the right method without the need to determine the class of the object.

10.5.2 Extended Visitor Pattern

Although the Visitor pattern provides high abstraction on a number of objects, it does have some flaws. Extensibility is expensive, because in order to introduce a new type of object it is urgent to add a new visit method to the `Switch` class in addition to creating the new class. To remedy this, SableCC introduces the Extended Visitor pattern.

The extended Visitor pattern introduces an ancestor interface for all the type classes called `Switchable`. Adding a new type imposes to create a new interface extending `Switchable` which provides a new visit method for the new type. A class can then be defined to implement the new extended interface, hence creating a visitor.

SableCC however has applied other names, than the usual Visitors pattern. `visitObject` methods are called `caseAObject` and the `accept` method is called `apply`.

10.5.3 SableCC Classes

SableCC uses the Extended Visitor pattern to generate the classes shown in Figures 10.5 and 10.5. These classes contain all that is needed to build a worker tree-walker.

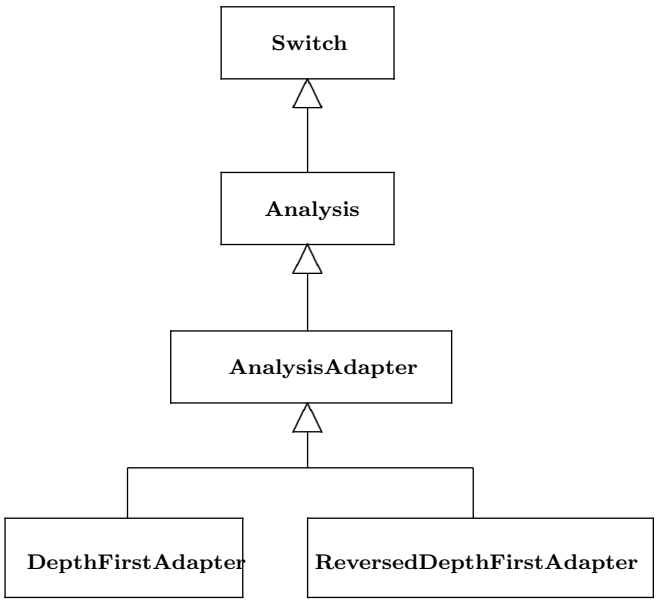

Figure 10.5: SableCC hierarchy for Switch

Besides that, SableCC also builds the packages `lexer` and `parser`, which constitute the syntactical analysis part of the compiler.

Two of the more interesting classes generated by SableCC are the `DepthFirstAdapter` and `ReversedDepthFirstAdapter`, descendants of the `Switch` interface. These are built of a number of `caseANode(ANode node)` methods that ensures that every node in the AST is visited.

To implement a worker tree-walker, you extend either of these two classes and overwrite the `caseANode(ANode node)` methods.

The constructed AST is pieced together by nodes representing grammar rules and tokens, which is pictured in Figure 10.6.

The nodes can be split up into three different categories: T_{token}, $P_{production}$ and $A_{node+production}$. The latter is the kind of node actually visited through the `case` methods and it has Tokens as children. All Productions have more productions or $A_{node+production}$ as their children.

On the nodes, information such as the actual string tokens, line number and references to parents and children are stored on the object.

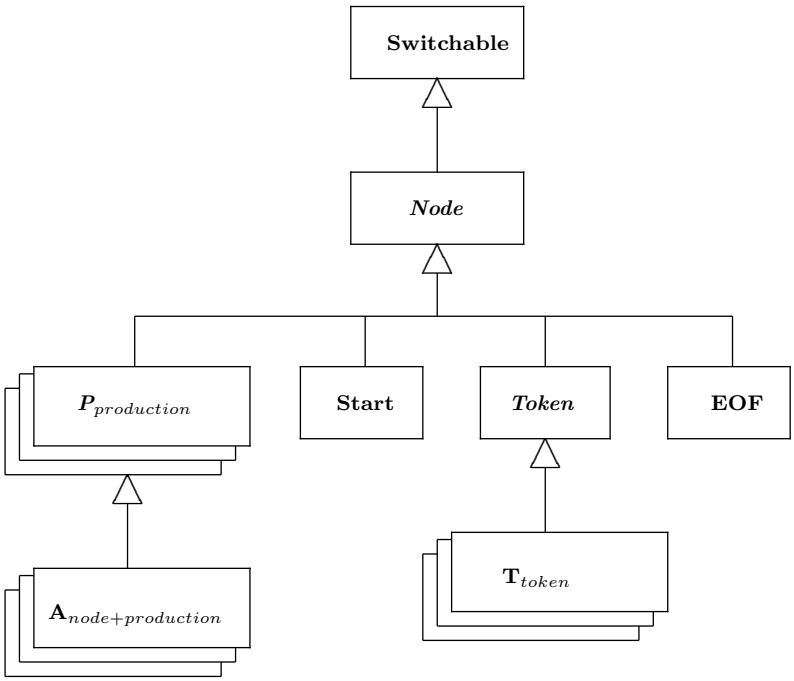

Figure 10.6: SableCC hierarchy for Switchable

Chapter 11

Compiler Design

The job of a compiler consists of three things: it must do a syntactical analysis, a contextual analysis, and code generation. In this chapter we document the design of these three phases as well as the design in general, which is covered in Section 11.1 . In spite of the fact that the design of the lexer and parser is defined by SableCC we will still comment on this design, and although runtime organization is included as a chapter of its own, it is intended to act as a design for the code generation, describing how to overcome issues such as mapping an imperative language to the object-oriented platform JVM.

11.1 Compiler Considerations

Our compiler consists of the classes depicted in Figure 11.2. In general the compiler is split up into several passes, each with their own responsibility. We will look closer at them the following sections. First we will have a inspect the data structures utilized by the compiler, namely `StandardEnvironment`, `Library`, and `ErrorList`. After that we will present the different packages that constitute the `dogsc` compiler.

11.1.1 StandardEnvironment

The `StandardEnvironment` class consists of information about the types available in DOGS. When new record declarations are encountered they are added to an instance of this class at compile time. The instance is resident through the whole compile process. This class is not related to the standard environment descrpied in the design document.

11.1.2 Library

To ensure that imported DOGS packages, both selfdefined and the standard library, are available at compile time, the compiler uses the `Library` class. It uses two `HashMaps` to store the known functions and procedures, as well as their interfaces. The information includes formal parameter sequences, return values, and package names, which is packaged in our self defined `IFace` object.

The class itself makes use of the Java `ClassLoader` and `Class` classes, to query a class-file for information, which is then written to the `HashMaps`. Note that the

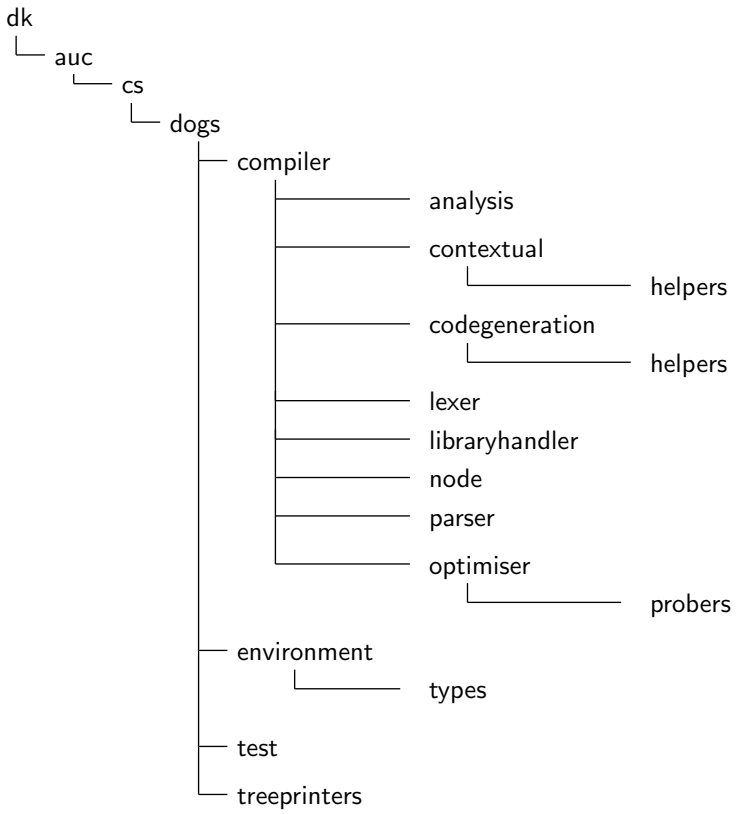

Figure 11.1: Overview of the compiler structure

`Library` is built for the DOGS language only, therefore some restrictions apply. For instance, in order to avoid namespace conflicts, only one instance of a function or procedure name is allowed. It should also be noted that the `Library` can only guarantee the presence of a given class file at compile time. At runtime there is no such guarantee.

11.1.3 ErrorList

The `ErrorList` is a static class that accumulates error messages generated by the checker classes. When an error is encountered, an error description and the appropriate $\texttt{T}_{Token}$ node is stored in the `ErrorList`. After each checker has been applied to the CST, these messages are printed to the screen and the compile process is halted, given any. The output error messages consists of a line number gathered from the $\texttt{T}_{Token}$ and the description.

11.1.4 Packages

Briefly put, present the packages in the `dogsc` are:

The `dk.auc.cs.dogs.compiler.*` contains all parts of the compiler. It can be further divided into smaller parts, in which we find the SableCC generated packages `analysis`, `lexer`, `node`, and `parser`. Our own worker classes are located in `contextual`, `codegeneration`, `libraryhandler`, and `optimiser`.

The `dk.auc.cs.dogs.environment.*` contains the implementation of the standard environment and the DOGS types.

The `dk.auc.cs.dogs.test.*` contains all the unit tests made during the implementation phase.

The `dk.auc.cs.dogs.treeprinters.*` contains handy tools for getting an overview of a given sourcefile.

11.2 Syntactical Analysis

The purpose of the syntactical analysis is to ensure that the source code is syntactically correct. This can be split in two tasks: lexical analysis and parsing.

Lexical analysis, is mainly, scanning the sourcefile in order to create a stream of tokens. This stream is passed on to the parser, which determines the phrase structure of the source program, and ensures that it is correct. The output of the parser is a well stuctured datatype containing the source program, which in modern compilers often is a tree.

The syntactical analyser is basically a DFA (deterministic finite automaton), which has an accept state for every valid construct in the language. When it encounters errors it generates an error report and shifts back to the last known safe state and continue analysis.

The reason for this behaviour is that it tries to collect as many errors as possible in a single analysis of the source code.

Creating a piece of software that implements the above mentioned functionality is rather mechanical, yet lengthy, task, which is exactly why we use SableCC to do this.

11.3 Contextual Analysis

Knowing that the source program is syntactically well formed the contextual analysis can commence. This analysis is a series of checks to see whether the program abides to the contextual rules or not. The structure of the `dogsc` contextual analyzer will be presented in the following.

11.3.1 Optimizer

The `Optimiser` class reduces constant expressions, a term that is also known as *constant folding*. Optimizations are performed on string, arithmetic and boolean expressions. Consider the example in Code Listing 11.1.

```
1     ...
2     let
3       float f := 4 + 8 * 3.14;
4       boolean b := 3 <= 4 and 2 > 1;
5     in
6     begin
7       if true then
```

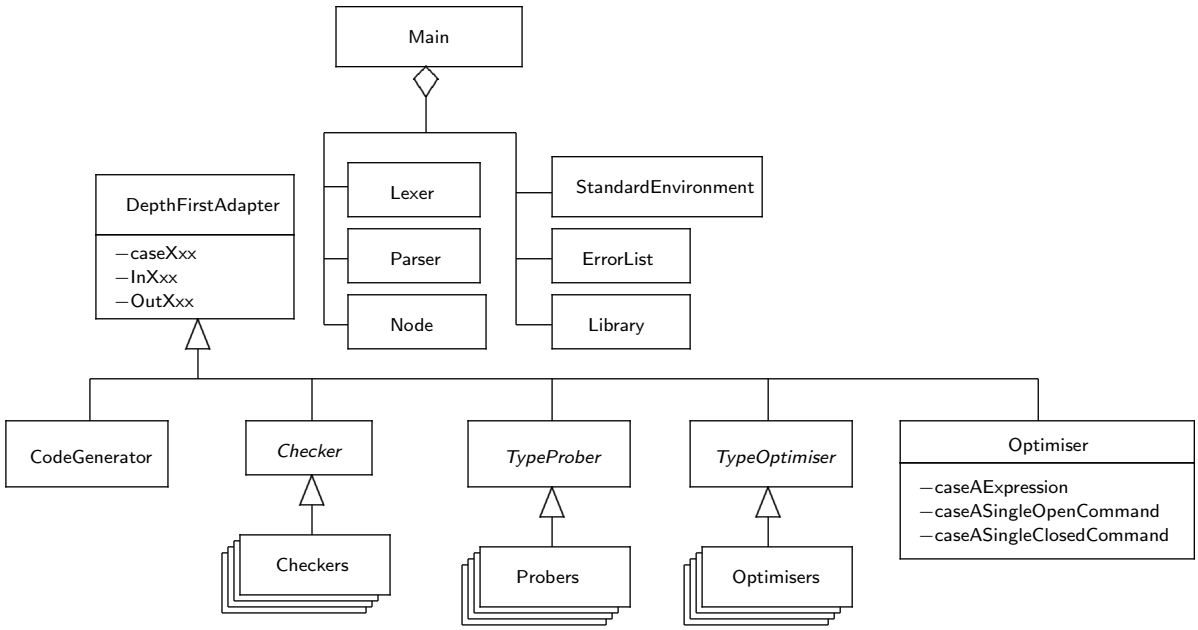

Figure 11.2: Class diagram for the contextual phase

```
8          print("string" & " concatenation");
9       else
10         print("this is obsolete");
11      print(f);
12   end
13   ...
```

Listing 11.1: Constant folding in `dogsc`

In the `let-in` block we notice that both the float and the boolean expressions can be reduced dramatically, by replacing the float expression with `29.12` ($4+8*3.14 = 29.12$) and the boolean expression with `true`. Also the string expression in line 8 is needlessly large. The optimizations are possible since the values in the expressions are known at compile time.

This kind of optimization is precisely what the `Optimiser` class does. It applies the following optimizers in the order given here on expressions:

- StringOptimiser
- ArithmeticOptimiser
- BooleanOptimiser

It also applies a `BranchingOptimiser` to single commands, which tries to determine branching statements that can be determined to be constant or dead code. When successful it replaces `if <b> then` <S_1> `else` <S_2> expressions with either <S_1> or <S_2>, depending on the boolean value `<b>`.

After these transformations to Code Listing 11.1, a number of things have been reduced, which is shown in Code Listing 11.2.

```
1  ...
2  let
3    float f := 29.12;
4    boolean b := true;
5  in
6  begin
7    print("string concatenation");
8    print(f);
9  end
10 ...
```

Listing 11.2: Applied constant folding in `dogsc`

All expressions can, however, not always be optimized, so in order to determine whether an expression can be optimized, the `Optimiser` classes launch `Probers` on the expressions.

The `Prober` classes traverse the subtree on which they are applied and accumulate relevant information, e.g. arithmetic operators, integer and float literals, and parenthesis for arithmetic optimization. These are stored in the `Prober` object. Below we have listed the prober classes in DOGS as well as the abstract superclass they all inherit from.

- *TypeProber*
- StringProber
- ArithmeticProber
- BooleanProber
- InftyProber
- BranchingProber

From the collected data the `Optimiser` can determine wether an expression can be optimized or not. An expression can be optimized only if the `Optimiser` can break the expression down to a subtree with only a case-specific number of pieces of stored information.

11.3.2 Contextual Checks

The abstract class `Checker` extends `DepthFirstAdapter`, thereby making it a tree walker. These properties are inherited by a number of different checkers:

- *Checker*
- BreakChecker
- ReturnChecker

- MainProcedureChecker
- DivisionByZeroChecker
- TypeChecker

`BreakChecker` traverses the tree looking for break commands out of place. `ReturnChecker` checks that defined functions contain return statements. It also asserts that no return commands are placed out of context which includes checking that eventual branches have a valid return statement. The `MainProcedureChecker` assures that declared program files contain a main procedure with the necessary arguments. It also confirms that a file declared to be a package does not contain a main procedure. `DivisionByZeroChecker` traverses the tree, ensuring that there are no divisions, integer divisions, or modulo by zero.

These are all simple classes with little complexity. The `TypeChecker` class, however, is rather complex and will be discussed in the following section.

11.3.3 TypeChecker

Type checking is the activity of enforcing the typing rules of a given language. In this case the rules of Section 8 have to be enforced. DOGS is a statically typed language, and therefore a key property is that a compiler is able to detect any type errors without actually running the program. When compared to dynamic (run time) typechecking, static typechecking is by far more efficient as type errors are discovered sooner and the type checking mechanism does not degrade performance because no run-time type checking is necessary [WB00].

In this section the possibilities for contextual analysis (typechecking, essentially) based on the framework provided by SableCC will be explored. Following this, the typechecking mechanism of `dogsc` will be described which includes an overview of the different classes contained within the typechecker. .

11.3.3.1 SableCC and Typechecking

Since `dogsc` is a multi-pass compiler, the results of the contextual analysis should be recorded for further use. This is often referred to as decoration of an AST [WB00]. There are a number of ways this is usually done, and some methods that have proved useful have in common that they store the information directly in the AST [WB00]. However, as explained, SableCC does not allow storing information on the nodes in the AST, and therefore `dogsc` stores everything in separate tables.

11.3.3.2 The `dogsc` Type Checker

This section describes the inner workings of the `dogsc` typechecker and to some detail how it is implemented. An overview of the different classes that make up the typechecker is presented in Figure 11.3 and will be described here.

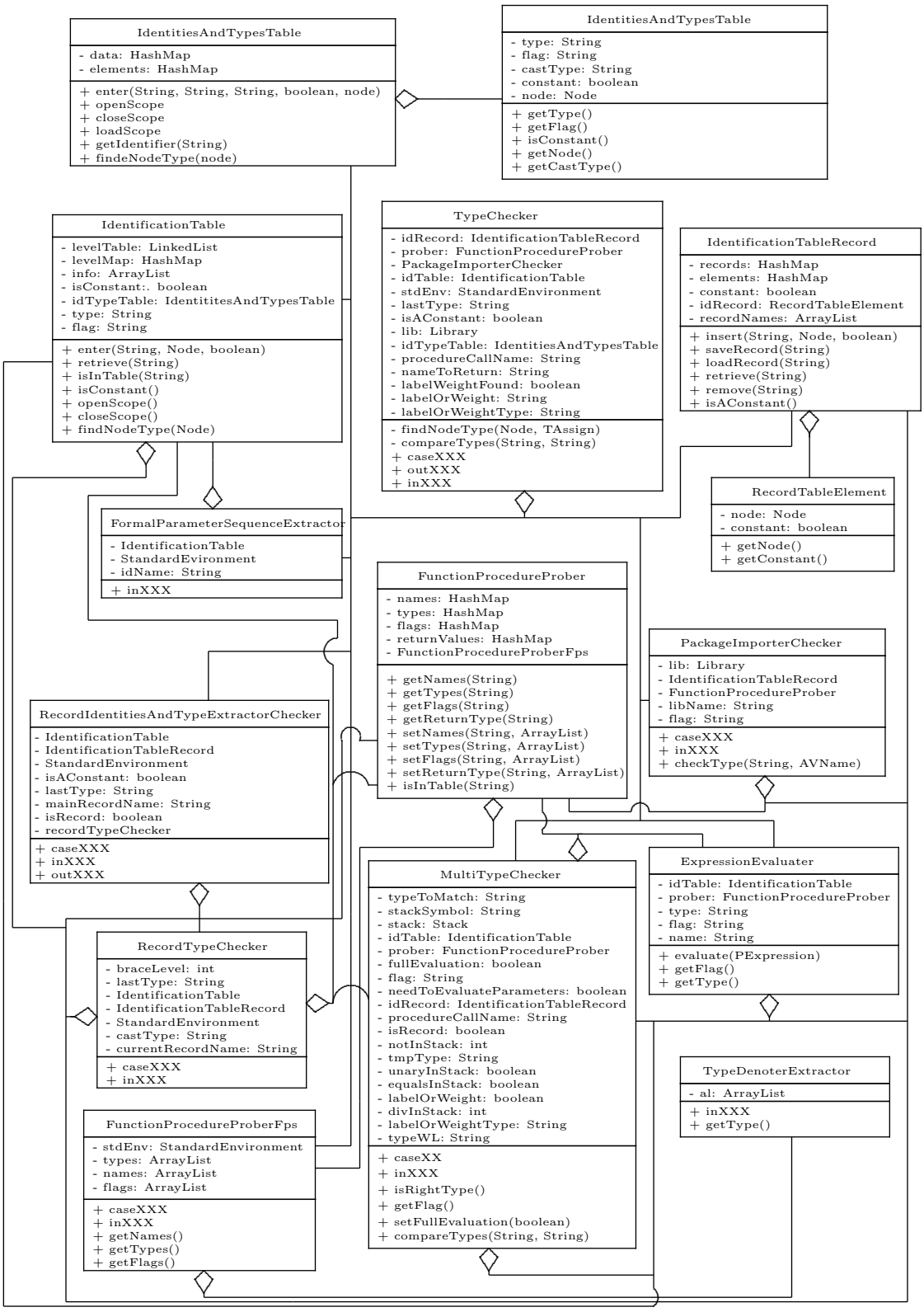

Figure 11.3: TypeChecker class diagram

The Classes

- **TypeChecker**: The `TypeChecker` class is the heart of the typechecker. It is from this class the different subtrees in the CST are evaluated.

  When initiated, the first responsibility of this class is to probe the AST to identify functions and procedures, which is necessary due to the scope rules for functions and procedures (cf. Section 6.9) (functions and procedures need not be declared in any particular order).

  Subsequently, the `PackageImporterChecker` is applied which will import the packages defined in the source code and check that all function and procedure invocations are defined in one of the compiled packages, or in the source code. From this point the real type checking starts following the flow of the source code and makes type checking on every expression, assignment, etc.

- **PackageImporterChecker**: In this class we use the library to import packages defined in the source code and in the standard library. To accomplish this, we use the `FunctionProcedureProber` to traverse the CST to find function and procedure calls. If the calls originate from a package we insert them into the function probe so they can be of use later in the typechecker to evaluate the calls.

- **MultiTypeChecker**: Almost every other class in the type checker relates to this class in form of an aggregation. It is used to evaluate expressions and variable name extensions, for instance, and is made as a universal type matcher. It takes a type in the form of a Java `String` and checks that the node it has been applied to is of the correct type.

  The `MultiTypeChecker` is implemented by a `Stack`. When, for instance, finding a variable during evaluation of an expression and its type matches the `String` passed to the `MultiTypeChecker`, it pushes the type on the stack. If the type of the variable does not match `String`, the symbol "`!!`" will be pushed on the stack.

  When the `MultiTypeChecker` has traversed the expression tree, the method `isRightType` is invoked, which then evaluates the stack to check whether or not the expression was true in relation to the `String` passed.

- **ExpressionEvaluater**: The `ExpressionEvaluater` class is a front-end to the `MultiTypeChecker`. Instead of matching against a single type it uses the `MultiTypeChecker` to try to match against every known type in DOGS and returns this type, should it find a match. This information is then used in the typechecker to make an evaluation of function or procedure calls for instance.

- **FunctionProcedureProber:** The `FunctionProcedureProber` is used to analyse the CST for functions and procedures. The reason for this pre-analysis is that we need some kind of insight into which functions and procedures are available in the source code. If this probing was not performed in the source code, we would not be able to call a function or procedure below the one currently invoked.

The prober uses four `HashMaps` to keep track of the variable names and types of the formal parameters, and the return types of the functions and procedures identified. It also contains a flag which is used to keep track of the type of a composite type, i.e. arrays, sets, etc. The function and procedure names are used as keys in the `HashMaps`.

- **FunctionProcedureProberFps** This class is used by the `FunctionProcedureProber` to extract the formal parameter sequence from a function or procedure. It uses three `ArrayList` into which it stores information about parameter names, types, and flags, e.g. name: A, type: string, flag: array.
- **TypeDenoterExtractor** This is a very little class with only one attribute and a few methods. It is uses by the `FunctionProcedureProberFps`, to help evaluate the formal parameter sequence.
- **FormalParameterSequenceExtractor:** As the name implies, this class is used to extract the formal parameter sequence from a function or procedure. The `FunctionProcedureProber` and the typechecker make calls during function and procedure declarations to extract the variable names in order to put them in the identification table. We do this because we do not want the programmer to be able to overwrite them in the body of the function or procedure.
- **RecordIdentitiesAndTypeExtractorChecker:** This class is used to find the types inside a record during declaration of it and checks that they are assigned to the proper type, should they be assigned to an expression. In collaboration with the `RecordTypeChecker`, it can find the right type of nested records in any depth.
- **RecordTypeChecker:** The `RecordTypeChecker` is used by the `RecordIdentitiesAndTypeExtractorChecker` which uses it to check if the types in nested record declarations are of the right type. It does so by applying them to the different subtrees when it detects a nested record.
- **IdentificationTableRecord:** Similar to the `IdentificationTable`, the `IdentificationTableRecord` is used to keep track of records and its members. It is implemented by using two `HashMaps`; one to save `Record- TableElements`, which are the identifiers or members of a record, and one to save the `HashMap` with the elements in, under the name of the record. Later, when we discover use of records, we can load the records and extract information about the individual members and use this information to do type checking of record usages.
- **RecordTableElement:** Instances of this class is used to save information about the individual identifier or element in a record. It holds a reference to the node in which the record was declared and a boolean to bookkeep if it is a constant or not.
- **IdentificationTable:** As suggested in [WB00, p.136] we have implemented an identification table in the contextual analysis, since, as mentioned, the SableCC generated framework supports no decorating of the CST. This is

where the `IdentificationTable` comes in. It is used to decorate the tree, or more simply put, keep track of declaration of variables, records, arrays, etc. It is implemented using a `HashMap` where we save an array with a reference to the node and a boolean to mark whether the declaration is constant or not. This `HashMap` is then saved in a `LinkedList` that represents the two scope levels. The lower level contains function, procedure, and record names. The upper level contains the declarations made in the `let-in` block in the function or procedure. These declarations are removed from the identification table when the function or procedure is left. Once inside the body of a function or procedure the `IdentificationTable` is used to do type checking when variables are used in expressions.

- **IdentitiesAndTypesTable:** This table is an extension of the `IdentificationTable` and works in the same way with only a few differences. The biggest difference is that it does not remove identifiers when leaving a function or procedure. Also, it is implemented using two `HashMaps`. The second `HashMap` is used as a `LinkedList` to save the different scopes, only now under the names of the functions and procedures. The reason for this table is that it is used during code generation to extract types of identifiers, without having to traverse the tree more times than necessary.

- **IdentitiesAndTypesTableElements:** Objects of this class hold information regarding types of identifiers. They also hold a boolean to indicate if the identifier in question is a constant or not and, of course, the node itself and the type of the node in the form of a string.

Having now introduced the first to phrases of the compiler, the remainder of this document will be concerned with the final phrase, code generation.

11.4 Runtime Organization

In this section we present how types in DOGS are represented in JVM. Furthermore, we will outline some of the problems with representing types in JVM.

11.4.1 Implementing the Types

When working with a virtual machine like JVM there is no direct access to memory locations. Instead, it provides the primitives: boolean, int, float (both with alternatives with more or less precision), char, array, and references to objects [MD96, p. 57]. Taking this into consideration, the most obvious way to implement types from the DOGS language is to use the basic data types provided by JVM.

11.4.1.1 Implementing Infty

In the DOGS language we have the special value signed `infty`, which can be assigned to both `integers` and `floats`. These values make it difficult to implement the DOGS primitives to use JVM primitives directly, as it should be possible to compare values between them. The need for comparison between `integers` and `floats` eliminates

the possibility to let `infty` be implemented as the highest value (or lowest value for negative `infty`) for the primitive data type in JVM. This is due to the fact that the range of values for `integers` and `floats` is not the same, leading to `infty` being represented in two different values for the two different primitives, which makes the comparison of the values rather complicated.

11.4.1.2 Types through Objects

As the JVM only has support for local variables and no direct access to memory, the task of representing the DOGS types becomes somewhat unusual compared to the representation of e.g. types in C. As we deem it absurde to build a front-end to the JVM, just to have a traditional data representation, we have chosen to make use of the JVM's real strength, objects. It may seem somewhat strange to have types of an imperative language mapped to an object oriented platform, but it actually opens some interesting apspects, such as inheritance and polymorphism.

By implementing every type as classes we can take advantage of the many powerful features with object oriented design, as mentioned above with the primitive numbers. It is easy to manipulate the data just by placing methods on these classes. Also, it allows for easy description of common features of different types through inheritance. The aforementioned methods will be the obvious way to make use of when building our standard environment and take advantage of the way JVM is working with objects. The implementation of the standard environment is briefly described in Section 11.4.2.

When implemented with classes, we can take advantage of inheritance, and we have the option to simulate multiple inheritance with interfaces. By doing this we can let our primitives in runtime be specified as types of given interfaces for integers and floats, and let `infty` implement both of these interfaces. This way the simulated multiple inheritance lets `infty` be able to act as both `integer` and `float` in DOGS.

11.4.1.3 DOGS Types as Objects

For making our own hierarchy of types in the most intuitive way, we have chosen to implement an abstract super-class called `Type`, in which we include all the types that exist in a running DOGS program. This includes our primitive types, record types, composite types, and the special types for holding properties for graphs, this is illustrated in figure 11.4.

The first class inheriting from `Type` is the abstract class `Primitive`, which covers the primitive types in the DOGS language. The primitives are implemented as wrapper classes around the basic types in JVM.

Regarding the actual value representation. we have chosen to represent our `integer` as a wrapper class for the 64-bit `long` and the `float` as a `double`, which also is 64-bit. In runtime, `infty` keeps track of whether it is positive or negative by using an internal boolean inside the class.

The abstract class `Number`, from which `Infty` and `Reading` inherits is the superclass to the numbers. Arithmetic and boolean operations are implemented directly on the objects, with the interfaces defined by the `Number` superclass. By doing this,

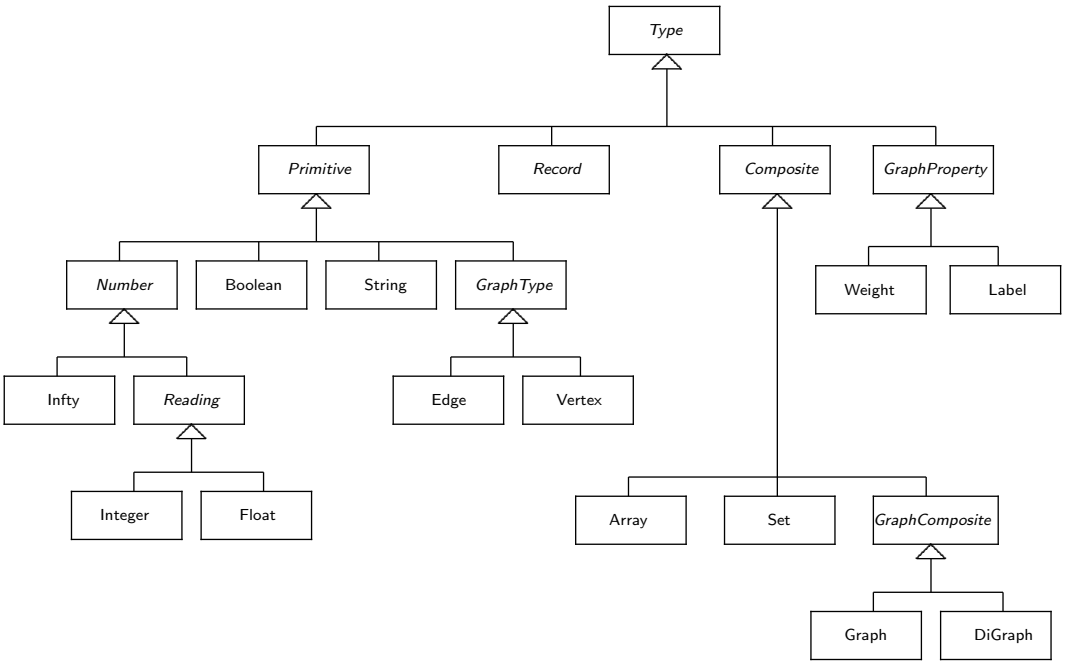

Figure 11.4: Class diagram of the types in DOGS

we take advantage of polymorphism in the sence that we need only specify the superclass when doing code generation as the JVM will call the appropriate operations.

There is however one complication, Java does not support multiple inheritance, which we for representing `infty`, which should inherit from both `integer` and `float`. We can, however in Java, simulate multiple inheritance through a number of interfaces as we mentioned in the last section.

Because we have chosen not to use single characters and only `strings`, the easiest, and most obvious way to implement them, is to use the standard `String` object, which is always available in JVM. But since we have chosen to make our own hierarchy of types, we have also made a `String` wrapper class.

Furthermore, we have the `Boolean` class, for the `boolean` type.

The class `GraphProperty` has two inheriting classes: `Weight` and `Label`. The `Composite` class has quite a few more classes inheriting, because it also covers the two types of graphs that we have implemented, `Graph` and `DiGraph`. These types of graphs are inheriting from the abstract class `GraphComposite`. One of the reasons for indexing the different types of graphs under an extra abstract class is to ease making methods work in a general way, for all kinds of graphs. Also, it can be used to declare some methods which should always be available on graphs, but not on the other composite types. Besides the composite graph types, the two classes `Set` and `Array` also inherit from the `Composite` class, which can both hold primitive types of the DOGS language.

The `Record` class is abstract, and should only be used as superclass for the records made when translating DOGS source code to JVM byte code.

The last group of types inheriting from `Primitive` are the special types for graph

purposes; `Edge` and `Vertex`, organized under the abstract class `GraphType`. As for `vertex` and `edge`, they are essentially just references to graphs.

11.4.2 Standard Environment

The functions and procedures in the DOGS Standard Environment are implemented directly in Java in the class `Standard`. In Code Listing 11.3 one of the implemented functions from `Standard` is shown. This function can be accessed as `addEdge` from a DOGS program.

```
1  // function boolean addEdge(graphComposite G, edge e)
2  public static Boolean addEdge(GraphComposite g, Edge e) {
3    return new Boolean(g.addEdge(e));
4  }
```

Listing 11.3: The implementation of `addEdge`

All functions and procedures in the Standard Environment are listed in Section A.2 on page 151.

11.5 Code Generation

Assured that the program is both syntactically and contextually correct, the code generation phase can commence. The `codegeneration` package consists of several abstract `Encoder` classes, which basically is one long chain of inheritance. The `CodeGenerator` class is actually the only class that is ever instantiated and the `Encoder` classes is a way to group related tree walking cases. All of the classes are listed below.

- *Encoder*
- *TypeEncoder*
- *ExpressionEncoder*
- *DeclarationEncoder*
- *StatementEncoder*
- *BranchingEncoder*
- *LoopingEncoder*
- CodeGenerator

The result of this inheritance is basically just a tree-walker that outputs certain Jasmin code when visiting certain nodes in the AST.

The `Encoder` classes, however, does not only contain code generating methods, they also contain cases that change the flow of the tree walker. This is for instance necessary when visiting the assignment `a := a + 10`. Here it is needed to push a new `Integer` object and it's initial value 10 onto the stack, load the variable `a`'s

location, and finally invoke the addition interface on the object and then store the object. Unless the walkers path is changed , it is not possible to push the right things in the right order. The code generated from visiting an assignment to an arithmetic addition expression in DOGS should look like Code Listing 11.4. In line 3 the object reference is loaded from local variable 1 and in line 4 to 7 a reference to a new DOGS integer object with the value 10 is initialized. To do this operation, first a new object is created with the `new` operator.

Now the stack consists of a reference to the `a` object at the bottom and a new reference to a uninitialized `Integer` object on top. Then the top reference is duplicated (we need the second to initialize the object) and 10 is pushed as a `long` on the stack.The stack now contains: a reference to `a`, two references to an uninitialized `Integer` object and a `long` with the value 10. The `initialize` method of the `Integer` object is then invoked, popping the `long` and the second reference to the uninitialized `Integer` object. This is done because parameters to methods are passed by popping elements from the stack. At the end of line 7, `V` indicates that the method returns void.

Now the stack consists of a reference to `a` at the bottom and a reference to the `integer` object with the value 10. The latter is then casted to a `NumberNumber` in line 9, and the actual `addition` method is invoked through the `NumberNumber` interface in line 10. The method pops the two object references from the stack, thus passing them as parameters to the `addition` method. Finally, in line 11, the result is stored in local variable 1.

```
1  ...
2    ; a := a + 10
3    aload 1
4    new dk/auc/cs/dogs/environment/types/Integer
5    dup
6    ldc2_w 10
7    invokespecial dk/auc/cs/dogs/environment/types/Integer/<
        init>(J)V
8
9    checkcast dk/auc/cs/dogs/environment/types/NumberNumber
10   invokeinterface dk/auc/cs/dogs/environment/types/
        NumberNumber/addition(Ldk/auc/cs/dogs/environment/types
        /NumberNumber;)Ldk/auc/cs/dogs/environment/types/
        NumberNumber; 2
11   astore 1
12 ...
```

Listing 11.4: The assignment `a := a + 10` in Jasmin code

The actual file output is done through the `CodePrinter` class, which basically is a wrapper class for the Java `PrintWriter` with a little more functionality, such as adding indentation and numbered labels in the generated Jasmin code.

The `VariableMap` is utilized in order to manage locations in the JVM correctly. It has functions for querying the location of a variable and vice versa. As shown in Figure 11.5, the class has a pointer, `nextNewLocation`, which always holds the value

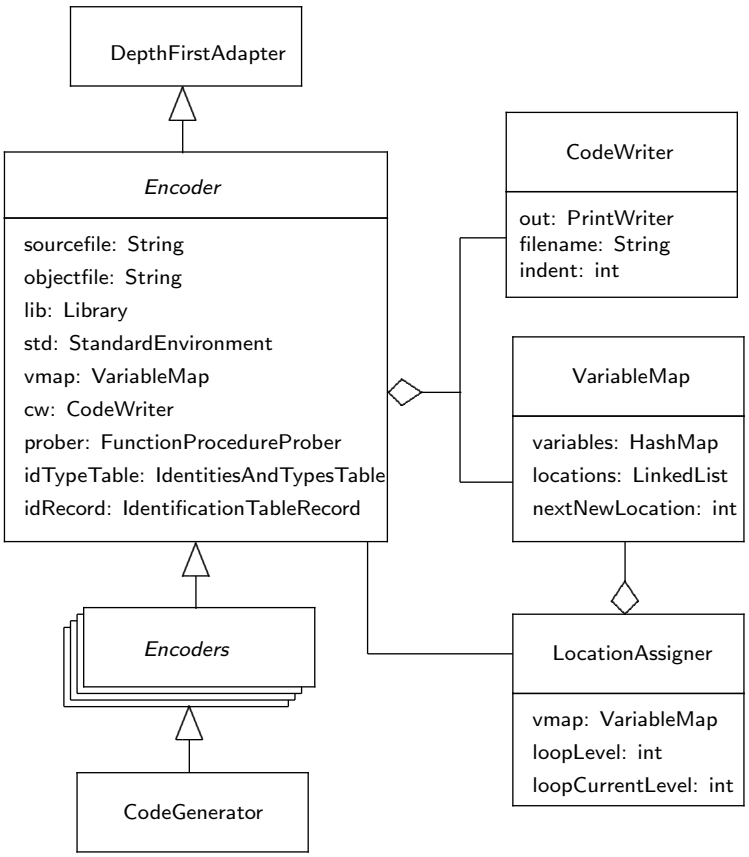

Figure 11.5: The CodeGeneration class diagram

of the next available location. This is easily done because an object in JVM only requires one local variable for references to objects.

The `LocationAssigner` is a small class that traverses subtrees constituting functions and procedures, before the actual `CodeGenerator` enters the subtrees. When encountering a variable it enters this variable name in the `VariableMap`, thereby reserving a location for the variable.

Chapter 12

Testing dogsc

Testing of our compiler, `dogsc`, was done through a number of imperative tests and unit tests. The unit tests can be found in `dk.auc.cs.dogs.test.*`, grouped together by the `TestDocsc` test suite class. These are rather mechanical and hold no real assurance that the compiler is correct. They do, however, provide a relatively good assurance that major parts of the compiler are sound and work as designed.

Besides the unit tests, we have applied the `dogsc` compiler on a number of source programs and monitored both the compiler and the runtime behaviour of the object program. In these sections we will present the results of the test concerning Dijkstra's algorithm for finding the shortest path, which was introduced in Section 7.5, page 72.

12.1 Hello Dogs

The first, and very simple, test is the mandatory `hello world` program, which is listed in Listing 12.1.

```
1 program helloworld;
2
3 procedure main(array of string args)
4 let
5    string str := "Hello Dogs World!!!";
6 in
7 begin
8         print(str);
9 end
```

Listing 12.1: Hello Dogs source code

Although a simple example, it does cover a lot of basic functionality: Declaration, assignment, and calls to the standard environment. The test was successful and `Hello Dogs World!!!` was printed on the screen.

12.2 Testing Dijkstra's Algorithm in DOGS

Now we move to a more complex test example, namely Dijkstra's Algorithm. When testing for some sort of correctness in the executable programs compiled by `dogsc`, the easiest way is to know the answer to the input beforehand. For this test, we have used a slightly modified version of Dijkstra's Algorithm, compared to the one presented in the analysis (cf. Section 2.2) in that the test version is also passed a target vertex. This version finds the shortest path between the two given vertices and prints out the shortest path and the length of the shortest path.

Our example consists of the vertices, edges, and weights shown below:

```
vertex   weight    vertex
a      -   5     -   b
a      -   3     -   c
b      -   9     -   e
c      -   7     -   d
c      -   9     -   e
d      -   16    -   f
e      -   3     -   f
```

The correct path, which the compiled program also has found, was through the nodes: `a,c,e,f` with the length 15.

From these empirical tests, we can conclude that `dogsc` works in both cases. Of course, we did a number of tests while building `dogsc` and found that the compiler, at least on small programs, works. We can, however, not rule out the possibility that there are bugs in the compiler, as discovering bugs in general in compilers is a long and tedious task.

Status of implementation

When reviewing the implementation of the DOGS language, there are parts of the language that we have not achieved to implement correctly, and some few parts that we have not implemented. The parts which we have not implemented correctly are:

- Call-by-reference (currently graphs, labels, weights, sets and arrays will always act as *call-by-reference* when passed as parameter)
- Declaration of records
- Declaration and use of multidimensional arrays
- Declaration of constants

The main reason for the lack of complete implementation of the DOGS language in `dogsc` is that DOGS is not only made for working with graphs, but also provides structures known from common programming languages, thus making the language heavy and complex to implement. Furthermore, the implementation of an imperative language, with the related primitives for such a language, for an object-oriented platform such as JVM, has proved difficult to say the least, not only due to the lack of documentation for SableCC and Jasmin, but also because our semantics cannot be fully used. This is a problem we will discuss in the conclusion.

Although our implementation of DOGS has some flaws, we find that it has been successful in that the parts of the specification that is missing in the implementation is the ones least likely to be employed by our target audience. All the essential constructs for working with graph algorithms are part of the implementation and, as such, the language and the compiler should be useful to those they were intended for. The table below lists the part of the DOGS specification that has been implemented in `dogsc`.

- Looping constructions
- Branching
- Declaration of functions and procedures
- Declaration of variables
- Graph and diGraph
- Weights and labels

- Sets
- Import of packages

We have turned to Dijkstra's Algorithm a number of times in the report, when describing graph algorithms. With a few modifications, this algorithm is possible to implement in DOGS, which is one of our main goals to be able to implement in the language.

Since we have many commonly known constructs in the DOGS language, we deem that it is also possible to create many programs which are not graph related.

Summary

In this chapter we have chosen the target platform for our implementation of DOGS. Though, we have chosen to use an intermediate language, Jasmin, to help during the development phase, this was only done to help debugging. We have also decided to take advantage of SableCC, which helped us in creating and organizing the structure of our compiler.

Our implementation of the DOGS language into the `dogsc` compiler, with all the functionality that it encapsulates, ranging from constant folding over typechecking to outputting of bytecode have been accounted for. All of this have been brought together in a final series of tests, in which we have seen a running of Dijkstra's Algorithm in DOGS.

Part IV

Conclusion

Chapter 13

Conclusion

Having presented the technical part of our report, we will now reflect on the parts we believe have been of the greatest influence in the development of the project. This includes the design criteria, our syntax, semantics, and compiler design.

13.1 Evaluating the Design Criteria

The language design criteria from Section 5.1 are important to consider when reflecting on how well our goals have been achieved, as they can indicate if the project has veered off course during design and implementation of the language. Here, we will discuss some of the criteria deemed important or very important.

Our primary concern was that the language be *readable* (e.g. the programs written in DOGS can be easily understood by novice programmers or programmers unfamiliar with the language). This ease of understanding should keep the programmers focus on the actual algorithm by making it possible to produce code similar to pseudo-code and with similar constructs. It is our belief that DOGS only falls a little short of fullfilling this goal. Even though we have implemented a series of constructs to facilitate this readability, the design of the language has forced us to somewhat limit the expressibility of the language (one example is that linked lists are not possible due to the lack of reference types). Even though the limitations are not directly related to the readability of the algorithms that can be expressed, it could still force the programmer to use rather arcane (and less intuitive) constructs in some places. One example is implementing queues via arrays or sets, which is a valid strategy, but a bit distant from the "usual" understanding (and even if this construct was included in a standard library there would still have to be a queue type for each type in the language). These shortcomings, however, do not change the fact that DOGS does allow for most graph algorithms to be expressed in a way that is similar to pseudo-code and, as such, should be easily comprehensible to our target audience.

As for the *writability* of the language a lot of the same observations can be made as with readability. The inclusion of many common constructs (e.g. `for-to` directly in the syntax of the language) allows for somewhat rapid expression of the programmer's thoughts. The high readability of the language also contributes positively in this category, however the same reservations must be made. The slightly limited pos-

sibilities allowed by the language might force the programmer to do certain things in ways different than the one most appealing. On top of this there are some areas that might be confusing to the programmer (such as the decision that all graph, label, and weight parameters are treated as references, while other types have to be explicitly marked for call-by-reference). These subtle details were a consequence of our focus on readability. The decision was made to make the final code resemble pseudo-code (it was estimated that most algorithms would need reference graph parameters and that `ref graph` would decrease the readability). In the end, however, DOGS remains a quite writable language.

Another criteria we found to be important for a successful design and implementation of the language was the language being *general*, meaning that a few constructs work in many different situations. The condition was that fewer constructs would not lessen the readability of the code, and we have learned that this condition has prevented us from fulfilling this goal for our language. We have included a lot of constructs to do things that could be accomplished with other language constructs. An example is that `switch-case` can be coded as a series of `if-then-else` constructs. Also, our constructs are less general that one could imagine. In some languages, `for` and `while` are more or less equivalent, whereas in DOGS they serve totally different purposes. Altogether these observations must lead to the conclusion that DOGS is not a highly general language, but since the generality of the language should not come at the expense of readability, it is not a disaster for the language.

The *standardability* of DOGS was rated important since it could not be assumed that everyone in our target audience could use the same platform. This concern merited a design which could be described in a standard way and as such be moved across platforms. Our choice in this matter was to simply use JVM as the platform for the language. This choice made our language available on a number of different platforms, but did also have some adverse effects on the implementation process (these will be described shortly). Nonetheless, the language (and as such the programs written) is highly mobile and the criteria is fulfilled.

In considering the *implementability* of the language we noted that it was an important criteria but that this goal would not be allowed to interfere with the readability. This reservation has proved to be something that we were forced to consider on a number of occasions during the design and implementation. Often a construct in the language could be altered slightly (or left out entirely) to ease the implementation, nevertheless these decisions could not be made without carefully evaluating what effect such changes would have on the readability of the language. The language can still be implemented (as shown) but the difficulty of the task could possibly be reduced. As such, we did not produce a highly implementable language, but we did not violate the readability clause either.

Ultimately, the primary concern was to produce a language that could be used to produce readable programs. Although the language might not be as readable as could be desired, the program written will still be quite close to pseudo-code, at least for most algorithms and better than pseudo-code algorithms implemented in common programming languages.

13.2 Operational Semantics and JVM

The work done on the formal specification of DOGS was not influenced by the decision to use JVM as our target architecture. This had a number of peculiar consequences.

In operational semantics it is common to use a somewhat abstract view of the machine details (e.g. the organization of the environment-store-model). Since a simplified model can greatly reduce the complexity of the finished semantics as well the development process. This choice often has the effect that the developed semantics is quite difficult to implement to a low-level (e.g. assembly) machine.

This classic dilemma is, ironically, the opposite of the situation we have experienced, when trying to apply our operational semantics in the compiler development process. Our semantics has a very elaborate model for stores (and environments) that puts it very close (in abstraction) to the architecture of an actual computer. That, however, does not map very well to the object-oriented nature of our target platform, that does not support either sequential allocation in stores (accessible through arithmetic expressions on locations) or pointers. These two restrictions (among others) hamper the implementation of the language in that the implementor has to be somewhat creative to translate the low-level details to higher level object-oriented constructs.

Despite this weakness in the semantics (for this particular platform) it is still a solid foundation upon which to construct the code generation, as it provides the implementor with the workings of all constructs in the language. On top of that, the language could potentially be implemented on a different (lower level) architecture without much difficulty, as a complete formal specification of the semantics and type system in DOGS has been made.

Even though this added difficulty has not been an insurmountable obstacle to the implementation, a simpler model for the semantics might have simplified the development of the semantics

To avoid this problem of mismatched abstraction levels it should have been realized earlier that JVM does not support the lower abstraction levels, and the semantics be developed accordingly.

13.3 Realization of DOGS

As we stated in the last chapter, the implementation of DOGS is not complete. However, DOGS is a large and expressive language, and the part of the language that has been implemented provides a wide range of functionality. It is possible to work with the main features in the language: Graphs, sets, weights, and labels, thus making it usable for the target audience.

The actual design of the compiler is somewhat complicated. Instead of one single datastructure containing all relevant information about the AST, we have several different tables containing the information. This could, and perhaps should, have been combined into one datastructure encapsulating the information. Looking back, it would have been a better design decision to have one pass gathering the information and make it available to the following passes.

The SableCC framework, which is the real backbone of the compiler, is a rather powerful tool. It does, however, have a steep learning curve and the poor documentation does not help. The fact that one cannot decorate the AST, which made it necesary to store information outside the tree, is, after all, still a sound SableCC design choice, as we deem that it has saved us a lot of time debugging.

In respects to the utilization of Jasmin as a front end to JVM bytecode, it has surely helped us quite a lot. Mostly in regards to debugging, as the Jasmin code is easier to comprehend than bytecode would have been. As with SableCC, Jasmin came with a just as bad documentation, which made the process of learning the syntax and opcodes harder.

13.4 Future Course

If we consider the future work with the project, we have a number of things to consider. First of all, a small redesign of the language could be to include pointers which could be beneficial. Another approach could be to implement the previously considered concurrency (cf. Section 6.11, page 50), although this would have a major impact on the semantics of DOGS , due to the choice of defining a big-step semantics.

Much more testing should be applied to make the compiler stable enough to consider performing a bootstrap, i.e. rewriting the compiler in the language itself. A redesign of the compiler would be in place to make the parts of the compiler more independent.

Part V

Bibliography

Bibliography

[App99] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1999.

[BA] Elliot Joel Berk and C. Scott Ananian. Jlex: A lexical analyzer generator for java(tm). http://www.cs.princeton.edu/~appel/modern/java/JLex/. Seen on 08-04-2004.

[Car04] Luca Cardelli. *CRC Handbook of Computer Science and Engineering*, chapter 97. CRC, 2004.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Massachusetts Institute of Technology, 2001.

[Cor] Microsoft Corporation. Microsoft .net framework. http://msdn.microsoft.com/netframework/. Seen on 15-4-2004.

[Gaga] Etienne M. Gagnon. Sablecc parser generator. http://www.sablecc.org/. Seen on 08-04-2004.

[Gagb] Étienne Gagnon. Sablecc, an object-oriented compiler framework. http://www.sablecc.org/thesis/thesis.php. Seen on 10-4-2004.

[Hud] Scott E. Hudson. Cup parser generator for java. http://www.cs.princeton.edu/~appel/modern/java/CUP/. Seen on 08-04-2004.

[Hüt04] Hans. Hüttel. *Pilen ved træets rod*. Aalborg Universitet, 2004.

[Inc] CollabNet Inc. javacc: Javacc home. https://javacc.dev.java.net/. Seen on 08-04-2004.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall, 1988.

[Lab] Francois Labelle. Programming language usage graph. http://www.cs.berkeley.edu/~flab/languages.html. Seen on 25-02-2004.

[Lou99] Kyle Loudon. *Mastering Algorithms in C*. O'Reilly & Associates Inc., 1999.

[Mar03] Alex Martelli. *Python in a Nutshell*. O'Reilly & Associates Inc., 2003.

[MD96] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly & Associates Inc., 1996.

[Mey] Jon Meyer. Java assembler interface. http://mrl.nyu.edu/~meyer/jasmin/. Seen on 30-03-2004.

[MI] Sun Microsystems Inc. Java virtual machine. http://java.sun.com/docs/books/vmspec/. Seen on 30-03-2004.

[Mit03] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[MM] Erik Meijer and Jim Miller. Technical overview of the common language runtime (or why the jvm is not my favorite execution environment). http://docs.msdnaa.net/ark/Webfiles/WhitePapers/CLR.pdf. Seen on 15-4-2004.

[MMMNS00] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object Oriented Analysis and Design*. Marko Publishing APS, Aalborg, Denmark, 2000.

[Nør] Kurt Nørmark. Programming paradigms. http://www.cs.auc.dk/~normark/prog3-02/pdf/paradigms.pdf. Seen on 09-03-2004.

[Ros03] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 2003.

[Seb04] Robert W. Sebesta. *Concepts of Programming Languages Sixth ED.* Addison Wesley, 2004.

[Sof] Parsifal Software. Resolving the general dangling else/if-else ambiguity. http://www.parsifalsoft.com/ifelse.html. Seen on 24-5-2004.

[Tho] Bent Thomsen. Programing languages and compilers - spring 2004. http://www.cs.auc.dk/~bt/SPOF04. Seen on 30-03-2004.

[WB00] David A Watt and Deryck F Brown. *Programming Language Processors in Java*. Pearson Education Limited, 2000.

[Wik] Wikipedia. Imperative programming. http://en.wikipedia.org/wiki/Programming_paradigm. Seen on 09-03-2004.

Part VI

Appendix

Appendix A

Standard Environment

A.1 Types

The types in DOGS include the most common types for working with graphs, and can be indexed into *primitives*, *composite types*, *graph types*, and *graph properties*. A further description of the types are listed here:

A.1.1 Primitives

Boolean: A `boolean` can hold the values `true` and `false`.

String: A `string` can hold an unlimited amount of characters.

Float: A `float` is a number with decimals, both positive and negative.

Integer: An `integer` is a number without decimals, both positive and negative.

Infty: An `infty` is a special constant which cannot be initialized as a variable, but only be assigned to `float` and `integer` variables. `Infty` can be both positive and negative.

Vertex: This primitive is an element in a graph, and can only be initialized by adding a vertex to a graph.

Edge: This primitive is a connection between two vertices.

A.1.2 Composite types

Array: An `array` is an indexed multidimensional list of primitives, which can only hold a limited of elements given when initialized.

Set: A `set` is an unsorted list of primitives, which can hold an unlimited amount of elements.

A.1.3 Graph types

Graph: A `graph` is a simple graph, which means that an `edge` $(u, v) = (v, u)$.

DiGraph: A `diGraph` is a directed graph, which means that an `edge` $(u, v) \neq (v, u)$.

A.1.4 Graph properties

Weight: A `weight` is always connected to a specific graph, and can hold primitive values for every `edge` in the graph.

Label: A `label` is a always connected to a specific graph, and can hold primitive values for every `vertex` in the graph.

A.2 Functions and Procedures

A.2.1 Input / Output

procedure print(string s)a: Prints the given string to the screen.

function string readStringInput(): Reads input from keyboard, until `return` is pressed, and returns this as a string.

function string readIntegerInput(): Reads input from keyboard, until `return` is pressed, and returns this as an integer.

function float readFloatInput(): Reads input from keyboard, until `return` is pressed, and returns this as a float.

function string readFile(string filename): Reads contents from the given file, and returns this as a string. If the file cannot be read the program will terminate with an error message.

function boolean writeFile(string filename, string content): Writes the given contents to the given file and returns a boolean, telling whether the writing went well. If the file does not exist it is created.

function boolean appendToFile(string filename, string content): Appends the given content to the given file and returns a boolean telling whether the writing went well. If the file does not exist it is created.

procedure exit(string message): Terminates the running program and prints the given message.

A.2.2 Conversion

function string integerToString(integer i): Converts the given integer to a string and returns this string.

function string floatToString(float f): Converts the given float to a string and returns this string.

function integer round(float f): Converts the given float to a integer and returns this integer. Any decimals are truncated.

function string booleanToString(boolean b): Converts the given boolean to a string and returns this string.

function integer stringToInteger(string s): Converts the given string to an integer and returns this integer. If the string cannot be converted to an integer the program will terminate.

function integer stringToFloat(string s): Converts the given string to a float and returns this integer. If the string cannot be converted to a float the program will terminate.

function boolean stringToBoolean(string s): Converts the given string to a boolean and returns this boolean. If the string cannot be converted to a boolean the program will terminate.

function array of string explodeString(string s, string pattern): Explodes the given string to an array by the given pattern. The strings in the pattern will be removed while exploding, thus not occur in the array. If the pattern is not found in the given string the array will contain only one element: with the given string.

A.2.3 Sets

Functions and procedures taking a `primitive` as argument can use all primitives in DOGS as argument.

procedure addToSet(set S, primitive p): Functions and procedures taking a `graphComposite` as argument, can use

Adds the given primitive to the given set. If the primitive equals another element in the set it will not be added to the set.

procedure removeFromSet(set S, primitive p): Removes the given primitive from the given set. No error occurs if the primitive is not in the set.

function boolean isInSet(set S, primitive p): Returns a boolean telling whether the given primitive is found in set or not.

function integer sizeOfSet(set S): Returns an integer with the size of the given set.

A.2.4 Arrays

function integer arrayDimensions(array a): Returns the amount of dimensons in the given array.

function integer arrayDimSize(array a, integer dim): Returns the size for the given dimension in the given array.

A.2.5 Graphs

Functions and procedures taking a `graphComposite` as argument can use both `graph` and `diGraph`.

procedure addVertex(graphComposite G, string name): Adds a vertex to the given graph with the given name and returns this vertex. If a vertex with the given name is already in the graph the program will terminate.

procedure removeVertex(graphComposite G, vertex v): Removes the given vertex from the graph. If the vertex is not in the graph the program will terminate.

function set of vertex vertices(graphComposite G): Returns a set containing all vertices in the given graph.

function set of edge edges(graphComposite G): Returns a set containing all edges in the given graph.

function boolean isEdge(vertex v1, vertex v2): Returns a boolean telling whether there is an edge between the two given vertices. If the two vertices are in different graphs the boolean will be set to false.

function vertex getVertex(graphComposite G, string name): Returns the vertex with the given name from the given graph. If no vertex with the given name is found the program will terminate.

function boolean addEdge(graphComposite G, edge e): Adds the given edge to the given graph. Returns a boolean, telling whether it was added. This function will return false if the edge is already in the graph.

function boolean nameOfVertex(vertex v): Returns the name of the vertex, as given when added to the graph.

Appendix B

DOGS Syntax in BNF

Commands

<command>	::=	`let` Declaration `in begin` Multi-Command `end`
	|	`begin` Multi-Command `end`
<Multi-Command>	::=	ϵ
	|	Single-Command Multi-Command
<Single-Command>	::=	Open-Command
	|	Closed-Command
<Open-Command>	::=	`if` Expression `then` Single-Command
	|	`if` Expression `then` Closed-Command `else` Open-Command
	|	Loop-Headers Open-Command
	|	`do` Open-Command `while` Expression
<Closed-Command>	::=	Basic-Commands
	|	`if` Expression Closed-Command `else` Closed-Command
	|	Loop-Headers Closed-Command
	|	`do` Closed-Command `while` Expression
<Loop-Headers>	::=	`while` Expression `do`
	|	`for` V-Name `:=` Expression `(to` | `downto)` Expression `do`
	|	`foreach` Single-Declaration `in` V-Name `do`
	|	`foreach` Single-Declaration `in` V-Name `where` Expression `do`
<Basic-Commands>	::=	V-Name `:=` Expression`;`
	|	Parameter-Call `;`
	|	Parameter-Call `:=` Expression `;`
	|	`begin` Multi-Command `end`
	|	`switch` V-Name Case-Item Default-Item `endswitch`
	|	`break;`
	|	`return` Expression`;`
	|	V-Name `++;`
	|	V-Name `--;`
<Parameter-Call>	::=	Identifier `(` Actual-Parameter-Sequence `)`

Case items

<Case-Item>	::=	ϵ
	|	`case` Integer-Literal `:` Single-Command Case-Item
	|	`case` Integer-Literal `..` Integer-Literal `:` Single-Command Case-Item
<Default-Item>	::=	ϵ
	|	`default:` single-command

Expressions

<Expression>	::=	Assign-Expr
<Primary-Expression>	::=	Integer-Literal
	|	String-Literal
	|	Boolean-Literal
	|	Float-Literal
	|	Infty
	|	V-Name
	|	Parameter-Call
	|	(Identifier, Identifier)
	|	(Expression)
	|	{Expression Comma-Expression}
<Comma-Expression>	::=	ϵ
	|	,Expression Comma-Expression

Precedence

```
<Assign-Expr>          ::=  Or-Expr
                        |   Or-Expr := Assign-Expr
<Or-Expr>              ::=  And-Expr
                        |   Or-Expr or And-Expr
                        |   Or-Expr xor And-Expr
<And-Expr>             ::=  Not-Expr
                        |   And-Expr and Not-Expr
<Not-Expr>             ::=  Compare-Expr
                        |   not Not-Expr
<Compare-Expr>         ::=  Less-Greater-Expr
                        |   Compare-Expr = Less-Greater-Expr
                        |   Compare-Expr <> Less-Greater-Expr
<Less-Greater-Expr>    ::=  Plus-Minus-Expr
                        |   Less-Greater-Expr < Plus-Minus-Expr
                        |   Less-Greater-Expr > Plus-Minus-Expr
                        |   Less-Greater-Expr <= Plus-Minus-Expr
                        |   Less-Greater-Expr >= Plus-Minus-Expr
<Plus-Minus-Expr>      ::=  Multiplication-Expr
                        |   Plus-Minus-Expr + Multiplication-Expr
                        |   Plus-Minus-Expr - Multiplication-Expr
                        |   + Multiplication-Expr
                        |   - Multiplication-Expr
<Multiplication-Expr>  ::=  Concatenation-Expr
                        |   Multiplication-Expr * Concatenation-Expr
                        |   Multiplication-Expr / Concatenation-Expr
                        |   Multiplication-Expr div Concatenation-Expr
                        |   Multiplication-Expr % Concatenation-Expr
<Concatenation-Expr>   ::=  Primary-Expression
                        |   Primary-Expression & Concatenation-Expr;
```

Value-or-variable-names

```
<V-Name>   ::=  Identifier
             |  Identifier . V-Name
             |  Identifier [Expression]
```

Declarations

<declaration>	::=	ϵ
	|	Single-Declaration-Const-Type Declaration
	|	Single-Declaration; Declaration
<Single-Declaration>	::=	SAV-Help-Declarations (ϵ | Declaration-Assignment)
	|	`weight in` V-Name `of` Identifier Identifier
	|	`label in` V-Name `of` Identifier Identifier
<Single-Declaration-Const-Type>	::=	`record` Identifier (Const-Declaration — SAV-Help-Declarations) Declaration-Assignment Type-Declaration-Helper;
	|	Const-Declaration Declaration-Assignment;
<Type-Declaration-Helper>	::=	ϵ
	|	, (Const-Declaration — SAV-Help-Declarations) Declaration-Assignment Type-Declaration-Helper
<Multi-Declaration-Const-Type>	::=	ϵ
	|	Single-Declaration-Const-Type Multi-Declaration-Const-Type
<Single-Secondary-Declaration>	::=	ϵ
	|	`procedure` Identifier `(`Formal-Parameter-Sequence`)` Command Single-Secondary-Declaration
	|	`function` Type-Denoter Identifier `(`Formal-Parameter-Sequence`)` Command Single-Secondary-Declaration
<Declaration-Assignment>	::=	ϵ
	|	`::=` Assign-Expr
<Const-Declaration>	::=	`constant` SAV-Help-Declarations
<Variable-Declaration	::=	Identifier Identifier
<Set-Declaration>	::=	`set of` Identifier Identifier
<Array-Declaration>	::=	`array of` Identifier Array-Size-Denoter Identifier
<Array-Size-Denoter>	::=	ϵ
	|	`[` Integer-Literal `]`
	|	`[` Integer-Literal `]` Array-Size-Denoter
<Sav-Help-Declarations>	::=	Set-Declaration
	|	Array-Declaration
	|	Variable-Declaration

Parameters

<Formal-Parameter-Sequence>	::=	ϵ
	|	Type-Denoter (`ref` | ϵ) Identifier Comma-Formal-Parameter
<Comma-Formal-Parameter>	::=	ϵ
	|	,Type-Denoter (`ref` | ϵ) Identifier
<Actual-Parameter-Sequence>	::=	ϵ
	|	Expression Comma-Expression

Type-denoter

<Type-Denoter>	::=	Identifier
	|	`array of` Identifier
	|	`weight of` Identifier
	|	`label of` Identifier
	|	`set of` Identifier

Program

<Program>	::=	`program` Identifier `;` Package-Import Multi-Declaration-Const-Type Single-Secondary-Declaration
	|	`package` Identifier `;` Package-Import Multi-Declaration-Const-Type Single-Secondary-Declaration
<Package-Import>	::=	ϵ
	|	`import` V-Name`;` Package-Import

Lexicon

<Token>	::=	Integer-Literal | String-Literal | Float-Literal | Boolean-Literal | Identifier | Operator | `array` | `begin` | `constant` | `do` | `else` | `end` | `function` | `procedure` | `if` | `in` | `let` | `of` | `record` | `then` | `while` | `for` | `foreach` | `default` | `break` | `to` | `downto` | `where` | `switch` | `endswitch` | `return` | `case` | `ref` | `weight` | `label` | `edge` | `vertex` | `infty` | `program` | `package` | `++` | `--` | `break` | `infty` | `default` | `import` | `.` | `:` | `;` | `,` | `:=` | `(` | `)` | `[` | `]` | `{` | `}`
<Integer-Literal>	::=	Digit Digit*
<Float-Literal>	::=	Digit Digit* `.` Digit Digit*
<Boolean-Literal>	::=	`true` | `false`
<String-Literal>	::=	‘‘ Escape-Literal* Letter* Escape-Literal* ’’
<Escape-Literal>	::=	`\n`
	|	`\t`
	|	`\\`
<Identifier>	::=	Letter (Letter | Digit)*
<Comment>	::=	`//` Graphic* end-of-line
	|	`/*` Graphic* `*/`
<Blank>	::=	space | tab | end-of-line
<Graphic>	::=	Letter | Digit | Operator | space | tab | `.` | `:` | `;` | `,` | `~` | `(` | `)` | `[` | `]` | `{` | `}` | `_` | `|` | `!` | `’` | `‘` | `‘‘` | `#` | `$`
<Letter>	::=	`a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` | `i` | `j` | `k` | `l` | `m` | `n` | `o` | `p` | `q` | `r` | `s` | `t` | `u` | `v` | `w` | `x` | `y` | `z` | `A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` | `P` | `Q` | `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z`
<Digit>	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
<Operator>	::=	`+` | `&` | `--` | `&` | `*` | `/` | `=` | `<=` | `>=` | `<` | `>` | `%` | `div` | `<>` | `and` | `or` | `not` | `xor`

Appendix C

SableCC grammar for DOGS

```
1
2  /* This is the grammar file for the DOGS compiler */
3
4  Package dk.auc.cs.dogs.compiler;
5
6  Helpers
7   letter = [[['a' .. 'z'] + ['A' .. 'Z']] + '_'];
8   digit = ['0' .. '9'];
9   digit_sequence = digit+;
10  escape_sequence = '\' ''' | '\t' | '\n' | '\\' | '\"';
11  all = [0 .. 127];
12  not_star = [all - '*'];
13  not_star_slash = [not_star - '/'];
14  s_char = [ all - [ '"' + [ '\' + [ 10 + 13 ] ] ] ] |
       escape_sequence;
15  s_char_sequence = s_char+;
16  cr = 13;
17  lf = 10;
18  tab = 9;
19  end_of_line = lf | cr | cr lf;
20  single_line_comment_input = [all - [cr + lf]];
21
22 Tokens
23  /* operators */
24  semicolon = ';';
25  colon = ':';
26  range = '..';
27  dot = '.';
28  comma = ',';
29  assign = ':=';
30  l_par = '(';
31  r_par = ')';
32  l_bracket = '[';
```

```
33  r_bracket = ']';
34  l_brace = '{';
35  r_brace = '}';
36
37  /* Arithmetic tokens */
38  star = '*';
39  plus = '+';
40  plus_plus = '++';
41  minus = '-';
42  minus_minus = '--';
43  div = '/';
44  mod = 'mod';
45  int_div = 'div';
46  concat = '&';
47
48  /* Logical tokens */
49  equal = '=';
50  lteq = '<=';
51  gteq = '>=';
52  lt = '<';
53  gt = '>';
54  neq = '<>';
55  and = 'and';
56  or = 'or';
57  not = 'not';
58  xor = 'xor';
59  infty = 'infty';
60
61  /* Control tokens */
62  begin = 'begin';
63  end = 'end';
64  let = 'let';
65  in = 'in';
66  if = 'if';
67  then = 'then';
68  else = 'else';
69  while = 'while';
70  do = 'do';
71  for = 'for';
72  to = 'to';
73  of = 'of';
74  downto = 'downto';
75  foreach = 'foreach';
76  where = 'where';
77  switch = 'switch';
78  endswitch = 'endswitch';
79  case = 'case';
```

```
80  default = 'default';
81  return = 'return';
82  break = 'break';
83
84  /* Type tokens */
85  ref = 'ref';
86  const = 'constant';
87  array = 'array';
88  record = 'record';
89  label = 'label';
90  weight = 'weight';
91  set = 'set';
92
93  /* Primitive type tokens */
94  integer_literal = digit+;
95  string_literal = '"' s_char_sequence ? '"';
96  boolean_literal = 'true' | 'false';
97  float_literal = digit+ '.' digit+;
98
99  /* Other */
100 procedure = 'procedure';
101 function = 'function';
102 program = 'program';
103 package = 'package';
104 import = 'import';
105 identifier = letter (digit | letter)*;
106
107 /* To be ignored */
108 blank = (cr | lf | tab | ' ')+;
109 comment = '/*' not_star* '*'+ (not_star_slash not_star
       * '*'+)* '/';
110 comment_line = '//' single_line_comment_input* end_of_line
       ?;
111
112 Ignored Tokens
113
114 blank,
115 comment,
116 comment_line;
117
118 Productions
119             /***********
120              * Program *
121              ***********/
122 program =
123  T.program identifier semicolon package_import*
       single_declaration_const_type*
```

```
    single_secondary_declaration* |
124 {package} T.package identifier semicolon package_import*
    single_declaration_const_type*
    single_secondary_declaration*;
125
126 package_import =
127  import [package_name]:v_name semicolon;
128
129            /***************
130             * Declarations *
131             ***************/
132 declaration =
133  declarations*;
134
135 declarations =
136           single_declaration_const_type |
137  {single} single_declaration semicolon;
138
139 single_declaration =
140  {set_array_var} sav_help_declarations
    declaration_assignment? |
141  {weight_label}  weight_label_declaration;
142
143 single_declaration_const_type =
144  {record}  record_declaration semicolon |
145  {const}   const_declaration declaration_assignment
    semicolon;
146
147 single_secondary_declaration =
148  {procedure} procedure identifier l_par
    formal_parameter_sequence? r_par command |
149  {function}  function type_denoter identifier l_par
    formal_parameter_sequence? r_par command;
150
151 declaration_assignment =
152  assign expression;
153
154            /*********
155             * Types *
156             *********/
157 record_declaration =
158  record [type]:identifier csav_help_declarations
    declaration_assignment? type_declaration_helper*;
159
160 const_declaration =
161  const sav_help_declarations;
162
```

```
163 variable_declaration =
164  [type]:identifier [name]:identifier;
165
166 array_declaration =
167  array of [type]:identifier array_size_denoter? [name]:
        identifier;
168
169 set_declaration =
170  set of [type]:identifier [name]:identifier;
171
172 weight_label_declaration =
173  {weight} weight in [graph]:v_name of [type]:identifier [
        name]:identifier |
174  {label}  label in [graph]:v_name of [type]:identifier [name
        ]:identifier;
175
176             /****************
177              * Type helpers *
178              ****************/
179 csav_help_declarations =
180  {const}         const_declaration |
181  {set_array_var} sav_help_declarations;
182
183 sav_help_declarations =
184  {set}      set_declaration |
185  {array}    array_declaration |
186  {variable} variable_declaration;
187
188 type_declaration_helper =
189  comma csav_help_declarations declaration_assignment?;
190
191 array_size_denoter =
192  {single} l_bracket integer_literal r_bracket |
193  {multi}  l_bracket integer_literal r_bracket
        array_size_denoter;
194
195             /****************
196              * Type Denoter *
197              ****************/
198 type_denoter =
199  {identifier} T.identifier |
200  {array}      array of [type]:identifier |
201  {weight}     weight of [type]:identifier |
202  {label}      label of [type]:identifier |
203  {set}        set of [type]:identifier;
204
205             /***********
```

```
206             * Command *
207             **********/
208 command =
209  {let_in_begin_end} let declaration in begin single_command
        * end |
210  {begin_end}        begin single_command* end;
211
212 single_command =
213  {open}   open_command |
214  {closed} closed_command;
215
216 open_command =
217  {if_then}  if expression then single_command |
218  {if_else}  if expression then closed_command else
        open_command |
219  {loop}     loop_headers open_command |
220  {while_do} while expression do open_command|
221  {do_while} do open_command while expression semicolon;
222
223 closed_command =
224  {basic_command} basic_commands |
225  {if_open_else}  if expression then [true]:closed_command
        else [false]:closed_command |
226  {loop}          loop_headers closed_command |
227  {while_do}      while expression do closed_command|
228  {do_while}      do closed_command while expression
        semicolon;
229
230 loop_headers =
231  {for_to}              for v_name assign [from_expr]:
        expression to [to_expr]:expression do |
232  {for_downto}          for v_name assign [from_expr]:
        expression downto [downto_expr]:expression do |
233  {foreach_in_do}       foreach single_declaration in v_name
        do |
234  {foreach_in_where_do} foreach single_declaration in v_name
        where expression do;
235
236 basic_commands =
237  {assign}     v_name assign expression semicolon |
238  {call}       parameter_call semicolon |
239  {lw_ass}     parameter_call assign expression semicolon |
240  {switch}     switch v_name case_item* default_item?
        endswitch |
241  {begin_end}  begin single_command* end |
242  {break}      break semicolon |
243  {return}     return expression semicolon |
```

```
244  {postfix_pp} v_name plus_plus semicolon |
245  {postfix_mm} v_name minus_minus semicolon;
246
247           /******************
248            * Case Definition *
249            ******************/
250 case_item =
251  {single}  case integer_literal colon single_command |
252  {range}   case [from]:integer_literal range [to]:
        integer_literal colon single_command;
253
254 default_item =
255  {case} default colon single_command;
256
257          /***************
258           * Expressions *
259           ***************/
260 expression =
261  assign_expr; /* Start of the precedence ring */
262
263 primary_expression =
264  {integer}  integer_literal |
265  {string}   string_literal |
266  {boolean}  boolean_literal |
267  {float}    float_literal |
268  {infty}    infty |
269  {v_name}   v_name |
270  {call}     parameter_call|
271  {edge}     l_par [first]:identifier comma [last]:identifier
        r_par |
272  {par}      l_par expression r_par |
273  {brace}    l_brace expression primary_expr_comma_expr*
        r_brace;
274
275 primary_expr_comma_expr =
276  comma expression;
277
278          /*******************
279           * Precedence rules *
280           *******************/
281 assign_expr =
282  {or}     or_expr |
283  {assign} or_expr T.assign assign_expr;
284
285 or_expr =
286  {and} and_expr |
287  {or}  or_expr T.or and_expr |
```

```
288  {xor} or_expr T.xor and_expr;
289
290 and_expr =
291  {not} not_expr |
292  {and} and_expr T.and not_expr;
293
294 not_expr =
295  {compare} compare_expr |
296  {not}     T.not not_expr;
297
298 compare_expr =
299  {lt_gt}     less_greater_expr |
300  {equal}     compare_expr T.equal less_greater_expr |
301  {not_equal} compare_expr T.neq less_greater_expr;
302
303 less_greater_expr =
304  {plus_minus} plus_minus_expr |
305  {lt}         less_greater_expr T.lt plus_minus_expr |
306  {gt}         less_greater_expr T.gt plus_minus_expr |
307  {lteq}       less_greater_expr T.lteq plus_minus_expr |
308  {gteq}       less_greater_expr T.gteq plus_minus_expr;
309
310 plus_minus_expr =
311  {multi}       multiplication_expr |
312  {plus}        plus_minus_expr T.plus multiplication_expr |
313  {minus}       plus_minus_expr T.minus multiplication_expr |
314  {unary_plus}  T.plus multiplication_expr |
315  {unary_minus} T.minus multiplication_expr;
316
317 multiplication_expr =
318  {concatination_expr} concatination_expr |
319  {star}               multiplication_expr T.star
        concatination_expr |
320  {div}                multiplication_expr T.div
        concatination_expr |
321  {int_div}            multiplication_expr T.int_div
        concatination_expr |
322  {mod}                multiplication_expr T.mod
        concatination_expr;
323
324 concatination_expr =
325  {unary_pp_mm}   primary_expression |
326                  primary_expression concat
                        concatination_expr;
327
328          /*********************
329           * v-name identifiers *
```

```
330           **********************/
331 v_name =
332  identifier v_name_extension?;
333
334 v_name_extension =
335  {identifier_record} dot v_name |
336  {identifier_array}  l_bracket expression r_bracket;
337
338 parameter_call =
339  [name]:identifier l_par actual_parameter_sequence? r_par;
340
341          /********************
342           * Formal Parameter *
343           ********************/
344 formal_parameter_sequence =
345  type_denoter ref? identifier formal_type_denoter*;
346
347 formal_type_denoter =
348  comma type_denoter ref? identifier;
349
350          /********************
351           * Actual Parameter *
352           ********************/
353 actual_parameter_sequence =
354  expression actual_expression*;
355
356 actual_expression =
357  comma expression;
```

Listing C.1: Gramma for the DOGS language

Appendix D

DOGS Formal Type System

Formal presentation of the type system in DOGS excluding the record type.

D.1 Type Rules

$boolean$	boolean type	$\in basic$, $\in primitiveTypes$
$string$	string type	$\in basic$, $\in primitiveTypes$
$float$	float type	$\in basic$, $\in primitiveTypes$
$integer$	integer type	$\in basic$, $\in primitiveTypes$
$infty$	infty type	
$vertex$	vertex type	$\in primitiveTypes$
$edge$	edge type	$\in primitiveTypes$
$array\ of\ primitive$	array type	
$set\ of\ primitive$	set type	
$weight\ of\ primitive$	weight type	
$label\ of\ primitive$	label type	
$type_{N_1} \rightarrow type_{N_2}$	function type	
$proc\ type_N$	procedure type	
$prog$	program type	
$pack$	package type	
$type_N\ constant$	constant type	

Table D.1: Types in the DOGS type system

D.2 Declaration Rules

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash type_N$	$type_N$ is a well-formed type in Γ
$\Gamma \vdash S$	S is a well-formed command in Γ
$\Gamma \vdash exp : type_N$	exp is a well-formed expression of type $type_N$ in Γ
$\Gamma \vdash D \therefore A$	D is a well-formed expression of signature A in Γ
$\Gamma \vdash type_{N_1} <: type_{N_2}$	$type_{N_1}$ is a subtype of $type_{N_2}$ in Γ

Table D.2: Judgements for the DOGS type system

D.3 Command Rules

[type-boolean]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{boolean}}$$

[type-diGraph]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{diGraph}}$$

[type-func]
$$\frac{\Gamma \vdash D \therefore A \qquad \Gamma \vdash type_N}{\Gamma \vdash \texttt{A} \rightarrow type_N}$$

[type-string]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{string}}$$

[type-boolean1]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{boolean}}$$

[type-proc]
$$\frac{\Gamma \vdash D \therefore A}{\Gamma \vdash \texttt{proc}\ A}$$

[type-integer]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{integer}}$$

[type-infty]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{infty}}$$

[type-prog]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{prog}}$$

[type-float]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{float}}$$

[type-array]
$$\frac{\Gamma \vdash \text{primitiveType}}{\Gamma \vdash \texttt{array of}\ \text{primitiveType}}$$

[type-pack]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{pack}}$$

[type-vertex]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{vertex}}$$

[type-set]
$$\frac{\Gamma \vdash \text{primitiveType}}{\Gamma \vdash \texttt{set of}\ \text{primitiveType}}$$

[type-const]
$$\frac{\Gamma \vdash type_N}{\Gamma \vdash type_N\ \texttt{constant}}$$

[type-edge]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{edge}}$$

[type-weight]
$$\frac{\Gamma \vdash \text{primitiveType}}{\Gamma \vdash \texttt{weight of}\ \text{primitiveType}}$$

[type-graph]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{graph}}$$

[label]
$$\frac{\Gamma \vdash \text{primitiveType}}{\Gamma \vdash \texttt{label of}\ \text{primitiveType}}$$

Table D.3: Basic type rules

D.4 Expression Rules

[sub-type-int-float]

$$\frac{\Gamma \vdash integer \qquad \Gamma \vdash float}{\Gamma \vdash integer <: float}$$

[sub-type-reflex]

$$\frac{\Gamma \vdash type_N}{\Gamma \vdash type_N <: type_N}$$

[sub-type-subsumption]

$$\frac{\Gamma \vdash x : type_{N_2} \qquad \Gamma \vdash type_{N_1} <: type_{N_2}}{\Gamma \vdash x : type_{N_2}}$$

Table D.4: Subtype rules

[env-empty]

$$\frac{}{\emptyset \vdash \diamond}$$

[env-extension]

$$\frac{\Gamma \vdash type_N \qquad x \notin dom(\Gamma)}{\Gamma, x : type_N \vdash \diamond}$$

[env-var-exists]

$$\frac{\Gamma, x : type_N \vdash \diamond}{\Gamma, x : type_N \vdash x : type_N}$$

Table D.5: Environment rules

[dec-prog]

$$\frac{\emptyset \vdash Imp \therefore A_1 \qquad A_1 \vdash D_R \therefore A_2 \qquad A_1, A_2 \vdash D_C \therefore A_3 \qquad A_1, A_2, A_3 \vdash D_B \therefore A_4}{\emptyset \vdash (\texttt{program}\ x\ Imp\ D_R\ D_C\ D_B) \therefore (x : prog)}$$

[dec-pack]

$$\frac{\Gamma \vdash Imp \therefore A_1 \qquad \Gamma, A_1 \vdash D_R \therefore A_2 \qquad \Gamma, A_1, A_2 \vdash D_C \therefore A_3 \qquad \Gamma, A_1, A_2, A_3 \vdash D_B \therefore A_4}{\Gamma \vdash (\texttt{package}\ x\ Imp\ D_R\ D_C\ D_B) \therefore (x : pack, A_1, A_2, A_3, A3)}$$

[dec-imp-block]

$$\frac{\Gamma \vdash Imp_1 \therefore A_1 \qquad \Gamma, A_1 \vdash Imp_2 \therefore A_2}{\Gamma \vdash (Imp_1 Imp_2) \therefore (A_1, A_2}$$

[dec-imp]

$$\frac{\Gamma \vdash x : string}{\Gamma \vdash (Impx) \therefore A}$$

Table D.6: Program and package declarations, Pro

[dec-constant-block]

$$\frac{\Gamma \vdash D_{C_1} \therefore (x_1 : type_{N_1}) \qquad \Gamma \vdash D_{C_2} \therefore (x_2 : typeN_2)}{\Gamma \vdash (D_{C_1} D_{C_2}) \therefore (x_1 : type_{N_1},\ x_2 : type_{N_2})}$$

[dec-constant]

$$\frac{\Gamma \vdash type_N \qquad \Gamma \vdash exp : type_N}{\Gamma \vdash (\texttt{constant}\ type_N\ x{:=}exp) \therefore (x : type_N\ constant)}$$

Table D.7: Constant declarations, D_C

[dec-func]

$$\frac{\Gamma,\ A \vdash Letin : type_N \qquad \Gamma \vdash type_N \qquad \Gamma \vdash par_F \therefore A}{\Gamma \vdash (\texttt{function}\ type_N\ x\ (par_F)\ LetIn) \therefore (x : A \rightarrow type_N}$$

[dec-proc]

$$\frac{\Gamma \vdash par_F \therefore A \qquad \Gamma,\ A \vdash LetIn}{\Gamma \vdash (\texttt{procedure}\ (par_F)\ LetIn) \therefore (x : proc\ A)}$$

Table D.8: Function and procedure declarations, D_B

[dec-var]

$$\frac{\Gamma \vdash type_N \qquad type_N \in primitiveType}{\Gamma \vdash (type_N\ x) \therefore (x : type_N)}$$

[dec-var-init]

$$\frac{\Gamma \vdash type_N \quad \Gamma \vdash exp : type_N}{\Gamma \vdash (type_N\ x := exp) \therefore (x : type_N)}$$

[dec-array]

$$\frac{\Gamma \vdash type_N \qquad \Gamma \vdash ArrayDim \qquad type_N \in primitiveType}{\Gamma \vdash (\texttt{arrayof}\ type_N\ x\ ArrayDim) \therefore (x : array\ of\ type_N)}$$

[dec-array-init]

$$\frac{\Gamma \vdash type_N \qquad \Gamma \vdash ArrayDim \qquad \Gamma \vdash exp : type_N}{\Gamma \vdash (\text{array of } type_N\ x \text{ ArrayDim} := exp) \therefore (x : \text{array of } type_N)}$$

[dec-set]

$$\frac{\Gamma \vdash type_N \qquad type_N \in primitiveType}{\Gamma \vdash (\texttt{set of}\ \ type_N\ x) \therefore (x : \texttt{set of}\ \ type_N)}$$

[decl-weight]

$$\frac{\Gamma \vdash type_N \quad \Gamma \vdash x : \{\texttt{graph}, \texttt{diGraph}\} \qquad type_N \in basic}{\Gamma \vdash (\texttt{weight in}\ \ x\ \ \texttt{of}\ \ type_N\ x) \therefore (x : \texttt{weight of}\ \ type_N)}$$

[decl-label]

$$\frac{\Gamma \vdash type_N \quad \Gamma \vdash x : \{\texttt{graph}, \texttt{diGraph}\} \qquad type_N \in basic}{\Gamma \vdash (\texttt{label in}\ \ x\ \ \texttt{of}\ \ type_N\ x) \therefore (x : \texttt{label of}\ \ type_N)}$$

Table D.9: Variable declarations, D_V

[single-Par_F]

$$\frac{\Gamma \vdash type_N \qquad x \notin dom(\Gamma)}{\Gamma \vdash type_N \;\; x : type_N}$$

[multi-Par_F]

$$\frac{\Gamma \vdash Par_F : A \qquad \Gamma \vdash type_N \qquad x \notin dom(\Gamma)}{\Gamma \vdash Par_F \;\; type_N \;\; x : A, type_N}$$

[single-ref-Par_F]

$$\frac{\Gamma \vdash type_N \qquad x :\notin dom(\Gamma)}{\Gamma \vdash type_N \text{ ref } x : type_N}$$

[multi-ref-Par_F]

$$\frac{\Gamma \vdash Par_F : A \qquad \Gamma \vdash type_N \qquad x \notin dom(\Gamma)}{\Gamma \vdash Par_F \;\; type_N \text{ ref } x : A, type_N}$$

Table D.10: Formal parameter sequence rules

[ass-var]

$$\frac{\Gamma \vdash x : type_N \qquad \Gamma \vdash exp : type_N}{\Gamma \vdash x := exp}$$

[ass-label]

$$\frac{\Gamma \vdash \texttt{label of}\ type_N \qquad \Gamma \vdash exp_2 : type_N \qquad \Gamma \vdash exp_1 : \texttt{vertex} \qquad type_N \in primitiveType}{\Gamma \vdash x\texttt{(}exp_1\texttt{):=}exp2}$$

[ass-weight]

$$\frac{\Gamma \vdash \texttt{weight of}\ type_N \qquad \Gamma \vdash exp_2 : type_N \qquad \Gamma \vdash exp_1 : \texttt{edge} \qquad type_N \in primitiveType}{\Gamma \vdash x\texttt{(}exp_1\texttt{):=}exp2}$$

[ass-array-element]

$$\frac{\Gamma \vdash \texttt{array of}\ type_N \qquad \Gamma \vdash exp : type_N \qquad type_N \in primitiveType \qquad \Gamma \vdash ArrayDim}{\Gamma \vdash x\ ArrayDim\ \texttt{:=}exp}$$

[ass-array]

$$\frac{\Gamma \vdash x : \texttt{arrayof}type_N \qquad \Gamma \vdash exp : type_N}{\Gamma \vdash (x\texttt{arrayDim} := exp)}$$

Table D.11: Assignment type rules

[for-downto-loop]

$$\frac{\Gamma,\ x : \texttt{integer} \vdash \diamond \qquad \Gamma \vdash i_2 : integer \qquad \Gamma \vdash S}{\Gamma \vdash \texttt{for}\ x \texttt{:=}\ i_1\ \texttt{downto}\ i_2\ \texttt{do}\ S}$$

[foreach-loop]

$$\frac{\Gamma \vdash \diamond \qquad \Gamma \vdash type_N \qquad \Gamma \vdash x = \{\texttt{array of}\ type_N,\ \texttt{set of}\ type_N\} \qquad \Gamma \vdash \qquad S \qquad \text{where } type_N \in primitiveType}{\Gamma \vdash \texttt{foreach}\ type_N\ x\ \texttt{in}\ x\ \texttt{do}\ S}$$

[foreach-where-loop]

$$\frac{\Gamma,\ x : type_N \vdash \diamond \qquad \Gamma \vdash type_N \qquad \Gamma \vdash x = \{\texttt{array of}\ type_N,\ \texttt{set of}\ type_N\} \qquad \Gamma \vdash S \qquad \Gamma \vdash b : \texttt{boolean} \qquad \text{where } type_N \in primitiveType}{\Gamma \vdash \texttt{foreach}\ type_N\ x\ \texttt{in}\ x\ \texttt{where}\ b\ \texttt{do}\ S}$$

[while-do-loop]

$$\frac{\Gamma \vdash b : \texttt{boolean} \qquad \Gamma \vdash S}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S}$$

[do-while-loop]

$$\frac{\Gamma \vdash b : \texttt{boolean} \qquad \Gamma \vdash S}{\Gamma \vdash \texttt{do}\ S\ \texttt{while}\ b}$$

[for-to-loop]

$$\frac{\Gamma,\ x : \texttt{integer} \vdash \diamond \qquad \Gamma \vdash i_2 : integer \qquad \Gamma \vdash S}{\Gamma \vdash \texttt{for}\ x \texttt{:=}\ i_1\ \texttt{to}\ i_2\ \texttt{do}\ S}$$

Table D.12: Loop type rules

[if-statement]

$$\frac{\Gamma \vdash b : \texttt{boolean} \qquad \Gamma \vdash S}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S}$$

[if-else-statement]

$$\frac{\Gamma \vdash b : \texttt{boolean} \qquad \Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2}$$

[comm-block]

$$\frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash S_1 S_2}$$

[switch]

$$\frac{\Gamma \vdash x : \texttt{integer} \qquad \Gamma \vdash S}{\Gamma \vdash \texttt{switch}\ x\ S\ \texttt{endswitch}}$$

[increment-decrement]

$$\frac{\Gamma \vdash x : type_N \qquad type_N \in \{float, integer\}}{\Gamma \vdash x \quad \square : type_N}$$

Where $\square = \{++, --\}$

Table D.13: General command type rules

[proc-call]
$$\frac{\Gamma \vdash x : \texttt{proc}\ A \qquad A \vdash Par_A}{\Gamma \vdash x(Par_A)}$$

[begin-end]
$$\frac{\Gamma \vdash S}{\Gamma \vdash \texttt{begin}\ S\ \texttt{end}}$$

[array-dim]
$$\frac{\Gamma \vdash \texttt{ArrayDim} \qquad \Gamma \vdash i : \texttt{integer}}{\Gamma \vdash [i]\ \ \texttt{ArrayDim}}$$

[LetIn]
$$\frac{\Gamma \vdash D_v \therefore (A) \qquad \Gamma, A \vdash S}{\Gamma \vdash (\texttt{let}\ D_v\ \texttt{in}\ S)}$$

Table D.14: General command type rules, cont.

[Par_A]
$$\frac{\Gamma \vdash exp : type_N}{\Gamma \vdash (exp, Par_A) : type_N, type'_N}$$

Table D.15: Actual parameter rules

[arit-gen-]

$$\frac{\Gamma \vdash exp_1 : \texttt{integer} \quad \Gamma \vdash exp_2 : \texttt{integer}}{\Gamma \vdash exp_1 \ \# \ exp_2 : \texttt{integer}}$$

where $\# = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{div}, \texttt{mod}\}$

[arit-gen]

$$\frac{\Gamma \vdash exp_1 : \texttt{float} \quad \Gamma \vdash exp_2 : \texttt{float}}{\Gamma \vdash exp_1 \ \# \ exp_2 : \texttt{float}}$$

where $\# = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$

[arit-gen-]

$$\frac{\Gamma \vdash exp : type_N \quad type_N \in \{\texttt{integer}, \texttt{float}\}}{\Gamma \vdash \ \# \ exp : type_N}$$

where $\# = \{\texttt{+}, \texttt{-}\}$

Table D.16: Type rules for arithmetic expressions

[concat-string]

$$\frac{\Gamma \vdash exp_1 : \texttt{string} \quad \Gamma \vdash exp_2 : \texttt{string}}{\Gamma \vdash exp_1 \& exp2 : \texttt{string}}$$

[bool-op]

$$\frac{\Gamma \vdash exp_1 : \texttt{boolean} \quad \Gamma \vdash exp2 : \texttt{boolean}}{\Gamma \vdash exp_1 \ \# \ exp_2 : \texttt{boolean}}$$

where $\# = \{\texttt{or}, \texttt{and}, \texttt{xor}\}$

[equal-string]

$$\frac{\Gamma \vdash exp_1 : \texttt{string} \quad \Gamma \vdash exp_2 : \texttt{string}}{\Gamma \vdash exp_1 \ \# \ exp_2 : \texttt{boolean}}$$

where $\# = \{\texttt{<>}, \texttt{=}\}$

[bool-not]

$$\frac{\Gamma \vdash exp : \texttt{boolean}}{\Gamma \vdash \texttt{not}\ exp : \texttt{boolean}}$$

[bool-gen]

$$\frac{\Gamma \vdash exp_1 : type_N \quad \Gamma \vdash exp_2 : type_N}{\Gamma \vdash exp_1 \ \# \ exp_2 : \texttt{boolean}}$$

where $\# = \{\texttt{<>}, \texttt{=}, \texttt{<=}, \texttt{>=}, \texttt{<}, \texttt{>}\}$
and $type_N = \{\texttt{integer}, \texttt{float}, \texttt{boolean}\}$

Table D.17: Type rules booleean expressions

[expr-par]

$$\frac{\Gamma \vdash S}{\Gamma \vdash (S)}$$

[function-call]

$$\frac{\Gamma \vdash x : A \rightarrow type_N \qquad \Gamma, A \vdash Par_A}{\Gamma \vdash (Func_N(Par_A)) : type_N}$$

[Weight]

$$\frac{\Gamma \vdash x : \text{weight of } type_N \qquad \Gamma \vdash exp : \text{edge}}{\Gamma \vdash x(exp) : type_N}$$

[label]

$$\frac{\Gamma \vdash x : \text{label of } type_N \qquad \Gamma \vdash exp : \text{vertex}}{\Gamma \vdash x(exp) : type_N}$$

[edge]

$$\frac{\Gamma \vdash x_1 : \text{vertex} \qquad \Gamma \vdash x_2 : \text{vertex}}{\Gamma \vdash (x_1, x_2) : \text{edge}}$$

Table D.18: General expression rules

Appendix E

Semantics

E.1 Generalized Variables

Generalized variabels are divided in semantics for variables and record variables. The semantics for variables are on the form $env_{RV}, env_V \vdash X \rightarrow (type_N, l)$. The semantics for record variables are on the form $env_{RV}, env_V \vdash X \rightarrow (env'_V, env'_{RV})$.

[Gvar1]

$$\frac{env'_{RV}, env'_V \vdash \langle X \rangle \to (type_N, l)}{env_{RV}, env_V \vdash \langle Rec_N.X \rangle \to (type_N, l)}$$

where $env_{RV}(Rec_N) = (env'_{RV}, env'_V)$

[Gvar2]

$$env_{RV}, env_V \vdash x \to (type_N, l)$$

where $env_V(x) = (type_N, l)$

[Gvar1-Rec]

$$\frac{env'_{RV}, env'_V \vdash \langle X \rangle \to (env''_V, env''_{RV})}{env_{RV}, env_V \vdash \langle Rec_N.X \rangle \to (env''_V, env''_{RV})}$$

where $env_{RV}(Rec_N) = (env'_V, env'_{RV})$

[Gvar2-Rec]

$$env_{RV}, env_V \vdash Rec_N \to (env'_V, env'_{RV})$$

where $env_{RV}(Rec_N) = (env'_V, env'_{RV})$

Table E.1: Transition rules for generalized variables

E.2 Declarations

All declarations are on the form $env_C, env_F, env_{RT}, env_P \vdash \langle DV, env_V, env_{RV}, sto\rangle \rightarrow_{DV} env'_V, env'_{RV}, sto'$.

[D$_{VB}$-block]

$$\frac{env_C, env_F, env_{RT}, env_P \vdash \langle D_{VB1}, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto'') \quad env_C, env_F, env_{RT}, env_P \vdash \langle D_{VB2}, env''_V, env''_{RV}, sto''\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}{env_C, env_F, env_{RT}, env_P \vdash \langle D_{VB1} D_{VB2}, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}$$

[D$_{VR}$-rec-block]

$$\frac{env_C, env_F, env_{RT}, env_P \vdash \langle D_{VR1}, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto'') \quad env_C, env_F, env_{RT}, env_P \vdash \langle D_{VR2}, env''_V, env''_{RV}, sto''\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}{env_C, env_F, env_{RT}, env_P \vdash \langle D_{VR1}, D_{VR2}, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}$$

[Dec-empty]

$$env_C, env_F, env_{RT}, env_P \vdash \langle \epsilon, env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env_V, env_{RV}, sto)$$

Table E.2: Transition rules for variable declarations in blocks and records, and the empty declaration

[Var-dec-init]

$$\frac{env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ x\mathbf{;}\ env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto'') \quad env_C, env_F, env'_V, env'_{RV}, env_{RT}, env_P \vdash \langle x\ \mathbf{:=}\ exp\mathbf{;}, sto''\rangle \rightarrow sto'}{env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ x\mathbf{:=}exp\mathbf{;},\ env_V, env_{RV}, sto\ \rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}$$

[Var-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ x\mathbf{;}\ env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env_V[x \mapsto (type_N, l)], env_{RV}, sto[sto_{type_N}[l \mapsto nil]])$$

where $l = new(sto_{type_N})$
$type_N \in primitiveTypes$

[Local-Const-dec-init]

$$\frac{env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ x\mathbf{:=}exp\mathbf{;}\ env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}{env_C, env_F, env_{RT}, env_P \vdash \langle \texttt{constant}\ type_N\ x\mathbf{:=}exp\mathbf{;}\ env_V, env_{RV}, sto\ \rangle \rightarrow_{DV} (env'_V, env'_{RV}, sto')}$$

Table E.3: Transition rules for variable declarations

[RefPar-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ \texttt{ref}\ x\mathbf{:=}X\mathbf{;}\ \ env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env_V[x \mapsto (type_N, l)], env_{RV}, sto)$$

where $env_V, env_{RV} \vdash X \rightarrow (type_N, l)$
$type_N \in primitiveTypes \cup \{\text{array of } type'_N, \text{set of } type'_N, \text{label of } type'_N, \text{weight of } type'_N, graph, diGraph\}$

[RefPar-rec-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ \texttt{ref}\ Rec_N\ \ \mathbf{:=}X\mathbf{;}env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env_V, env_{RV}[Rec_N \mapsto (env'_V, env'_{RV})], sto)$$

where $env_V, env_{RV} \vdash X \rightarrow (env'_V, env'_{RV})$
$type_N \in RecordType$

Table E.4: Transition rules for reference declarations

[Record-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle\ Rtype_N\ Rec_N \quad ;, env_V, env_{RV}, sto\rangle \rightarrow_{DV} env_V, env_{RV}[Rec_N \mapsto (env'_V, env'_{RV})], sto'$$

where $env_{RT}(Rtype) = (env''_V, env''_{RV})$
$\alpha(env''_V, env''_{RV}, sto) = (env'_V, env'_{RV}, sto')$

[Record-dec-help-function]

$$\alpha(env_V, env_{RV}, sto) = (env'_V, env'_{RV}, sto')$$

for each $x \in env_V$
$env_V(x) = (type_N, l)$
$env'_V[x \mapsto (type_N, l')]$
$l' = new(sto_{type_N})$ and $sto[l' \mapsto nil]$
$\emptyset, \emptyset, env'_V[x' \mapsto (type_N, l)], \emptyset, \emptyset, \emptyset \vdash \langle x{:=}x'{;}, sto''\rangle \rightarrow sto^{(3)}$

for each $Rec_N \in env_{RV}$
$env_{RV}(Rec_N) = (env^{(3)}_V, env^{(3)}_{RV}))$
$\alpha(env^{(3)}_V, env^{(3)}_{RV}, sto^{(3)}) = (env''_V, env''_{RV}, sto')$
$env'_{RV} = env_{RV}[Rec_N \mapsto (env''_V, env''_{RV})]$

Table E.5: Transition rules for declaration of record variables

[Array-dec]

$$\frac{\begin{array}{l} env_C, env_F, env_{RT}, env_P \vdash \\ \langle Arraydim, env_V[temp \mapsto (type_N, l+1)], env_{RV}, sto[sto_{array}[l \mapsto l+1][l+1 \mapsto 0]]\rangle \\ \rightarrow_{DV} (env'_V, env_{RV}, sto') \end{array}}{\begin{array}{rl} env_C, env_F, env_{RT}, env_P & \vdash \langle \texttt{array of } type_N\ x\ Arraydim\texttt{;}\ env_V, env_{RV}, sto\rangle \\ & \rightarrow_{DV} (env_V[x \mapsto (array, l)], env_{RV}, sto') \end{array}}$$

where $l = new(sto_{array})$
and *temp* is a concrete variable name

[Array-dim]

$$\frac{env_C, env_F, env_{RT}, env_P \vdash \langle Arraydim, env_V, env_{RV}, sto''[sto_{array}[l \mapsto x+1][l' \mapsto v]]\rangle \rightarrow_{DV} (env_V, env_{RV}, sto')}{env_C, env_F, env_{RT}, env_P \vdash \langle [i]\ Arraydim,\ env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env_V, env_{RV}, sto')}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i, sto\rangle \rightarrow (v, sto'')$
$env_V(temp) = (-, l)$
$x = sto_{array}(l)$
$l' = new(sto_{array})$

[Array-dim-empty]

$$env_C, env_F, env_{RT}, env_P \vdash \langle \epsilon,\ env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env_V, env_{RV}, sto[sto_{array}[l+s+j \mapsto nil]])$$

where $env_V(temp) = (-, l)$
$s = sto_{array}(l)$
$n_{i=1}, n_{i=2}, ..., n_{i=s} = sto_{array}(l+i)$
$m = n_1 \cdot n_2 \cdot ... \cdot n_s$
$j = 1, 2, ..., m$

Table E.6: Transition rules for array declaration

[Label-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle \texttt{label in } X \texttt{ of } type_N\ x\texttt{;}, \quad env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env_V[x \mapsto (label, l)], env_{RV}, sto[sto_{label}[l \mapsto (l', l+1)] [l+i \mapsto (nil, l+i+1)] [l+noVertices \mapsto (nil, nil)]\ sto_{graphProp}[l^{(4)} \mapsto (l, label, l^{(3)})] [l'' \mapsto (v+1, -, l^{(4)})]]$$

where
$l = new(sto_{label})$
$env_{RV}, env_V \vdash X \rightarrow (graph, l')$
$sto_{graph}(l') = (noVertices, l'')$
$i = 1, 2, \ldots, noVertices - 1$
$sto_{graphProp}(l'') = (v, -, l^{(3)})$
$l^{(4)} = new(sto_{graphProp})$

[Weight-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle \texttt{weight in } X \texttt{ of } type_N\ x\texttt{;}, \quad env_V, env_{RV}, sto\rangle \rightarrow_{DV} (env_V[x \mapsto (weight, l)], env_{RV}, sto[sto_{weight}[l \mapsto (l', l+1)] [l+i \mapsto (nil, l+i+1)] [l+noVertices^2 \mapsto (nil, nil)]\ sto_{graphProp}[l^{(4)} \mapsto (l, weight, l^{(3)})] [l'' \mapsto (v+1, -, l^{(4)})]]$$

where
$l = new(sto_{weight})$
$env_{RV}, env_V \vdash X \rightarrow (graph, l')$
$sto_{graph}(l') = (noVertices, l'')$
$i = 1, 2, \ldots, noVertices^2 - 1$
$sto_{graphProp}(l'') = (v, -, l^{(3)})$
$l^{(4)} = new(sto_{graphProp})$

Table E.7: Transition rules for weight and label declaration

[Graph-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle \texttt{graph}\ x\texttt{;}, env_V, env_{RV}, sto \rangle \rightarrow_{DV} (env_V[x \mapsto (graph, l)], env_{RV}, sto[sto_{graph}[l \mapsto (0, l+1)][l+1 \mapsto (l', nil)], sto_{graphProp}[l' \mapsto (0, integer, nil)]])$$

where $l = new(sto_{graph})$, $l' = new(sto_{graphProp})$

[DiGraph-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle \texttt{diGraph}\ x\texttt{;}, env_V, env_{RV}, sto \rangle \rightarrow_{DV} (env_V[x \mapsto (digraph, l)], env_{RV}, sto[sto_{graph}[l \mapsto (0, l+1)][l+1 \mapsto (l', nil)], sto_{graphProp}[l' \mapsto (0, integer, nil)]])$$

where $l = new(sto_{graph})$, $l' = new(sto_{graphProp})$

Table E.8: Transition rules for graph declarations

[Set-dec]

$$env_C, env_F, env_{RT}, env_P \vdash \langle \texttt{set of}\ type_N\ x\texttt{;}, env_V, env_{RV}, sto \rangle \rightarrow_{DV} (env_V[x \mapsto (set, l)], env_{RV}, sto[sto_{set}[l \mapsto (0, nil)]])$$

where $l = new(sto_{set})$

Table E.9: Transition rules for set declaration

E.3 Record type Declarations

All record type declarations are on the form $\langle DR, env_{RT}, sto\rangle \rightarrow_{DR} (env'_{RT}, sto')$ (can be found in Tabel E.10).

[DR-block]

$$\frac{\langle D_{R1}, env_{RT}, sto\rangle \rightarrow_{DR} (env''_{RT}, sto'') \quad \langle D_{R2}, env''_{RT}, sto''\rangle \rightarrow_{DR} (env'_{RT}, sto')}{\langle D_{R1} D_{R2}, env_C, sto\rangle \rightarrow_{DR} (env'_{RT}, sto')}$$

[Record-type-dec]

$$\frac{\emptyset, \emptyset, env_{RT}, \emptyset \vdash \langle D_{VR}, \emptyset, \emptyset, sto\rangle \rightarrow_{DV} (env_V, env_{RV}, sto'') \quad \langle D_R, env_{RT}[RType_N \mapsto (env_V, env_{RV})], sto''\rangle \rightarrow_{DR} (env'_{RT}, sto')}{\langle \texttt{record}\ RType_N\ V_{VR}\texttt{;}\ D_R, env_{RT}, sto\rangle \rightarrow_{DR} (env'_{RT}, sto')}$$

Table E.10: Transition rules for record type declaration

E.4 Global Constant Declarations

All global constant declarations are on the form $\langle D_C, env_C, sto\rangle \rightarrow_{DC} env'_C, sto'$.

[DC-block]

$$\frac{\langle D_{C1}, env_C, sto\rangle \rightarrow_{DC} (env''_C, sto'') \quad \langle D_{C2}, env''_C, sto''\rangle \rightarrow_{DC} (env'_C, sto')}{\langle D_{C1}D_{C2}, env_C, sto\rangle \rightarrow_{DC} (env'_C, sto')}$$

[Const-dec]

$$\langle type_N\ x\ \mathbf{:=}\ exp\mathbf{;}\ env_C, sto\rangle \rightarrow_{DC} (env_C[x \mapsto (type_N, l)], sto'[sto_{type_N}[l \mapsto v]])$$

where $env_C, env_F, \emptyset, \emptyset, \emptyset, \emptyset \vdash \langle exp, sto\rangle \rightarrow_{exp} (v, sto')$
$l = new(sto_{type_N})$
$type_N \in primitiveTypes$

[Const-dec-empty]

$$\langle \epsilon, env_C, sto\rangle \rightarrow_{DC} (env_C, sto)$$

Table E.11: Transition rules for global constant declarations

E.5 Procedure and Function Declarations

Procedure and function declarations are on the form $env_{RT} \vdash \langle D_B, env_F, env_P \rangle \rightarrow_{DB} (env'_F, env'_P)$.

[D$_B$-block]
$$\frac{env_{RT} \vdash \langle D_{B1}, env_F, env_P \rangle \rightarrow_{DB} (env''_F, env''_P) \quad env_{RT} \vdash \langle D_{B2}, env''_F, env''_P \rangle \rightarrow_{DB} (env'_F, env'_P)}{env_{RT} \vdash \langle D_{B1} D_{B2}, env_F, env_P \rangle \rightarrow_{DB} (env'_F, env'_P)}$$

[D$_B$-empty]
$$env_{RT} \vdash \langle \epsilon, env_F, env_P \rangle \rightarrow_{DB} (env_F, env_P)$$

Table E.12: Transition rules function and procedure declaration block

[Func-dec]

$$\frac{env_{RT}, temp[i \mapsto 0] \vdash \langle Par_F, \emptyset \rangle \rightarrow_{ParF} env_{PAR}}{env_{RT} \vdash \langle \texttt{function}\ type_N\ Func_N(Par_F)\ \texttt{begin}\ S\ \texttt{end}, env_F, env_P \rangle \rightarrow_{DB} (env'_F, env_P)}$$

$$\text{where} \quad env'_F = env_F[Func_N \mapsto (S, \emptyset, type_N, env_{PAR})]$$

[Func-dec-letIn]

$$\frac{env_{RT}, temp[i \mapsto 0] \vdash \langle Par_F, \emptyset \rangle \rightarrow_{ParF} env_{PAR}}{\begin{array}{l} env_{RT} \vdash \langle \texttt{function}\ type_N\ Func_N(Par_F)\ \texttt{let} \quad D_V\ \texttt{in}\ \texttt{begin}\ S\ \texttt{end}, env_F, env_P \rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow_{DB} (env'_F, env_P) \end{array}}$$

$$\text{where} \quad env'_F = env_F[Func_N \mapsto (S, D_V, type_N, env_{PAR})]$$

[Proc-dec]

$$\frac{env_{RT}, temp[i \mapsto 0] \vdash \langle Par_F, \emptyset \rangle \rightarrow_{ParF} env_{PAR}}{env_{RT} \vdash \langle \texttt{procedure}\ Proc_N(Par_F)\ \texttt{begin}\ S\ \texttt{end}, env_F, env_P \rangle \rightarrow_{DB} (env_F, env'_P)}$$

$$\text{where} \quad env'_P = env_P[Proc_N \mapsto (S, \emptyset, env_{PAR})]$$

[Proc-dec-LetIn]

$$\frac{env_{RT}, temp[i \mapsto 0] \vdash \langle Par_F, \emptyset \rangle \rightarrow_{ParF} env_{PAR}}{env_{RT} \vdash \langle \texttt{procedure}\ Proc_N(Par_F)\ \texttt{let}\ D_V\ \texttt{in}\ \texttt{begin}\ S\ \texttt{end}, env_F, env_P \rangle \rightarrow_{DB} (env_F, env'_P)}$$

$$\text{where} \quad env'_P = env_P[Proc_N \mapsto (S, D_V, env_{PAR})]$$

Table E.13: Transition rules declaration of procedures and functions

E.6 Formal-Parameter Declarations

Formal-parameter declarations are on the form $env_{RT}, temp \vdash \langle Par_F, env_{PAR} \rangle \rightarrow_{ParF} env'_{PAR}$.

[Par_F-dec]

$$\frac{env_{RT}, temp[i \mapsto i'] \vdash \langle Par_F, env_{PAR}[i' \mapsto (x, type_N, \mathit{ff})]\rangle \rightarrow_{ParF} env'_{PAR}}{env_{RT}, temp \vdash \langle type_N\ x\texttt{,}Par_F, env_{PAR}\rangle \rightarrow_{ParF} env'_{PAR}}$$

where $i' = temp(i) + 1$
$type_N = primitiveTypes \cup \{\text{set of } type'_N, \text{array of } type'_N\}$
$\cup RecordType$

[Par_F-dec-graph]

$$\frac{env_{RT}, temp[i \mapsto i'] \vdash \langle Par_F, env_{PAR}[i' \mapsto (x, type_N, \#t)]\rangle \rightarrow_{ParF} env'_{PAR}}{env_{RT}, temp \vdash \langle type_N\ x\texttt{,}Par_F, env_{PAR}\rangle \rightarrow_{ParF} env'_{PAR}}$$

where $i' = temp(i) + 1$
$type_N = \{graph, diGraph, \text{label of } type'_N, \text{weight of } type'_N\}$

[Par_F-dec-ref]

$$\frac{env_{RT}, temp[i \mapsto i'] \vdash \langle Par_F, env_{PAR}[i' \mapsto (x, type_N, \#t)]\rangle \rightarrow_{ParF} env'_{PAR}}{env_{RT}, temp \vdash \langle \texttt{ref}\ type_N\ x\texttt{,}Par_F, env_{PAR}\rangle \rightarrow_{ParF} env'_{PAR}}$$

where $i' = temp(i) + 1$
$type_N = primitiveTypes \cup \{\text{set of } type'_N, \text{array of } type'_N\}$
$\cup RecordType$

[Par_F-dec-empty]

$$env_{RT}, temp \vdash \langle \epsilon, env_{PAR}\rangle \rightarrow_{ParF} env_{PAR}$$

Table E.14: Transition rules for formal-parameter declarations

E.7 Program and Import

The program transistion rule is on the form $\langle program\ imp\ D_R\ D_C\ D_B,\ \emptyset\rangle \rightarrow sto$. The import transistion rules is on the form $\langle DR, env_C, env_{RT}, env_F, env_P, sto\rangle \rightarrow_{Imp} (env'_C, env'_{RT}, env'_F, env'_P, sto')$.

[Program]

$$\frac{\begin{array}{l}\langle Imp, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle \rightarrow_{Imp} (env_C, env_{RT}, env_F, env_P, sto)\\ \langle D_R, env_{RT}, sto\rangle \rightarrow_{DR} (env'_{RT}, sto')\\ \langle D_C, env_C, sto'\rangle \rightarrow_{DC} (env'_C, sto'')\\ env'_{RT} \vdash \langle D_B, env_F, env_P\rangle \rightarrow_{DB} (env'_F, env'_P)\\ env'_C, env'_F, env'_V[break \mapsto (boolean, l)][return \mapsto (boolean, l')], env'_{RV}, env'_{RT}, env'_P \vdash\\ \qquad \langle main(input);, sto''[sto_{boolean}[l \mapsto ff][l' \mapsto ff]]\rangle \rightarrow sto^{(3)}\end{array}}{\langle program\ Imp\ D_R\ D_C\ D_B, \emptyset\rangle \rightarrow sto^{(3)}}$$

where $input$ is an array of strings from the console
$new(boolean) = l$
$l + 1 = l'$

[Import]

$$\frac{\begin{array}{l}\langle D_R, env_{RT}, sto\rangle \rightarrow_{DR} (env''_{RT}, sto'')\\ \langle D_C, env_C, sto''\rangle \rightarrow_{DR} (env''_C, sto^{(3)})\\ env''_{RT} \vdash \langle D_B, env_F, env_P\rangle \rightarrow_{DB} (env''_F, env''_P)\\ \langle Imp, env''_C, env''_{RT}, env''_F, env''_P, sto^{(3)}\rangle \rightarrow_{Imp} (env'_C, env'_{RT}, env'_F, env'_P, sto')\end{array}}{\langle \texttt{import}\ str\texttt{;}\ Imp, env_C, env_{RT}, env_F, env_P\rangle \rightarrow_{Imp} (env'_C, env'_{RT}, env'_F, env'_P, sto')}$$

where $fileLoad(str) = (D_R, D_C, D_B)$

[Import-empty]

$$\langle \epsilon, env_C, env_{RT}, env_F, env_P, sto\rangle \rightarrow_{Imp} (env_C, env_{RT}, env_F, env_P, sto)$$

Table E.15: Transition rules for program execution and import

E.8 Commands

All command transitions are on the form $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto \rangle \rightarrow sto'$.

[Ass-primitive]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X{:=}exp;, sto\rangle \rightarrow sto'[sto_{type_N}[l \mapsto v]]$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (v, sto')$
$env_{RV}, env_V \vdash X \rightarrow (type_N, l)$
$type_N \in primitiveTypes \setminus float$

[Ass-float]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X{:=}exp;, sto\rangle \rightarrow sto'[sto_{type_N}[l \mapsto v]]$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (v', sto')$
$v = floatValue(v')$
$env_{RV}, env_V \vdash X \rightarrow (type_N, l)$
$type_N = float$

[Ass-set]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X{:=}exp;, sto\rangle \rightarrow sto'[sto_{set}[sLoc' \mapsto (size, l)][l + i - 1 \mapsto (v_i, l + i)][l + size - 1 \mapsto (-, nil)]]$$

where $env_{RV}, env_V \vdash X \rightarrow (set, sLoc')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (sLoc'', sto')$
$l = new(sto_{set})$
$(size, Loc) = sto_{set}(sLoc'')$
$l_i = Loc \qquad (i = 1)$
$(v_i, l_{i+1}) = sto_{set}(l_i)$
$i = 1, 2, ..., size$

[Ass-array]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X{:=}exp;, sto\rangle \rightarrow sto'[sto_{array}[aLoc' \mapsto l][l + i \mapsto v_i]]$$

where $env_{RV}, env_V \vdash X \rightarrow (array, aLoc')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (aLoc'', sto')$
$aLoc''_2 = sto_{array}(aLoc'')$
$l = new(sto_{array})$
$v_i = sto_{array}(aLoc''_2 + i)$
$i = 0, 1, ..., size$
$dims = sto_{array}(aLoc''_2)$
$n_{j=1}, n_{j=2}, ..., n_{j=dims} = sto_{array}(aLoc''_2 + j)$
$size = dims + n_1 \cdot n_2 \cdot \ldots \cdot n_{dims}$

Table E.16: Transition rules for primitive, set, and array assignments

[Ass-label-val]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle x\texttt{(}exp_1\texttt{) := }exp_2\texttt{;}, sto\rangle \rightarrow sto'[sto_{label}[l \mapsto v]]$$

where
$env_{RV}, env_V \vdash X \rightarrow (label, lLoc)$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (vLoc, sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (v, sto')$
$(graphLoc, -) = sto_{label}(lLoc)$
$(noVertices, propLoc) = sto_{graph}(graphLoc)$
vNo for which $\Gamma(sto'_{graph}, propLoc, vNo) = vLoc$
and $0 < vNo \leq noVertices$
$l = (\Gamma(sto'_{label}, lLoc, vNo))$

[Ass-weight-val]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle x\texttt{(}exp_1\texttt{) := }exp_2\texttt{;}, sto\rangle \rightarrow sto'[sto_{label}[l \mapsto v]]$$

where
$env_{RV}, env_V \vdash X \rightarrow (weight, wLoc)$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (vLoc' \times vLoc'', sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (v, sto')$
$(graphLoc, -) = sto_{weight}(wLoc)$
$(noVertices, propLoc) = sto'_{graph}(graphLoc)$
i for which $\Gamma(sto'_{graph}, propLoc, i) = Loc'$
and $0 < i \leq noVertices$
j for which $\Gamma(sto'_{graph}, propLoc, j) = Loc''$
and $0 < j \leq noVertices$
$eNo = (i-1) \cdot noVertices + j$
$(v, -) = sto_{weight}(\Gamma(sto_{weight}, wLoc, eNo))$
$l = (\Gamma(sto'_{label}, lLoc, vNo))$

Table E.17: Transition rules for weight and label value assignments

[Ass-infty]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X\mathbf{:=}\ infty\mathbf{;}, sto\rangle \rightarrow sto[sto_{type_N}[l \mapsto infty]]$$

where $env_{RV}, env_V \vdash X \rightarrow (type_N, l)$
$type_N \in \{integer, float\}$

[Ass-minus-infty]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X\mathbf{:=}\ -infty\mathbf{;}, sto\rangle \rightarrow sto[sto_{type_N}[l \mapsto -infty]]$$

where $env_{RV}, env_V \vdash X \rightarrow (type_N, l)$
$type_N \in \{integer, float\}$

Table E.18: Transition rules for infty assignments

[Ass-rec-aggregate]

$$\frac{env_C, env_F, env'_V, env'_{RV}, env_{RT}, env_P \vdash \langle RecAggr, sto\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle Rec_N\mathbf{:=\{}RecAggr\mathbf{\};}, sto\rangle \rightarrow sto'}$$

where $env_{RV}(Rec_N) = (env'_V, env'_{RV})$

[Rec-aggregate]

$$\frac{\begin{array}{c} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto\rangle \rightarrow sto'' \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle RecAggr, sto''\rangle \rightarrow sto' \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle SRecAggr, sto\rangle \rightarrow sto'}$$

Table E.19: Transition rules for assignments of record-aggregates

[Ass-graph]

$$
\begin{array}{l}
env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \quad X := exp;, sto \quad \rangle \rightarrow sto'[\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad sto_{graph} \quad [gLoc' \mapsto (size, l)] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [l \mapsto (pl, l+1)] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [l+i \mapsto (v_i, l+1+i)] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [l+length \mapsto (-, nil)], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad sto_{graphProp} \quad [pl \mapsto (psize, integer, pl+1)] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [pl+i \mapsto (pv_i, integer, pl+i+1)] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [pl+psize \mapsto (-, -, nil)]]
\end{array}
$$

where
$env_{RV}, env_V \vdash X \rightarrow (-, gLoc')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto \rangle \rightarrow_{exp} (gLoc'', sto')$
$new(sto_{graph}) = l$
$sto_{graph}(gLoc'') = (size, gpLoc)$
$sto_{graph}(gpLoc) = (gpLoc', vLoc)$
$l_i = vLoc \qquad (i = 1)$
$(v_i, l_{i+1}) = sto_{graph}(l_i)$
$i = 1, 2, \ldots, length$
$length = size + size \cdot size$

and
$new(sto_{graphProp}) = pl$
$sto_{graphProp}(gpLoc') = (psize, pLoc)$
$pl_i = pLoc \qquad (i = 1)$
$(pv_i, type_i, pl_{i+1}) = sto_{graphProp}(pl_i)$
$i = 1, 2, \ldots, psize$

Table E.20: Transition rule for graph assignments

[Ass-record]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X := exp;, sto\rangle \rightarrow sto'$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} ((env''_V, env''_{RV}), sto'')$
$env_{RV}, env_V \vdash X \rightarrow (env'_V, env'_{RV})$
$\beta((env'_V, env'_{RV}), (env''_V, env''_{RV}), sto'') = sto'$

[β]

$$\beta((env_V, env_{RV}), (env'_V, env'_{RV}), sto) = sto'$$

for each $x \in env_V$
$\emptyset, \emptyset, env_V[x' \mapsto (type_N, l)], env_{RV}, \emptyset, \emptyset \vdash \langle x := x', sto\rangle \rightarrow sto''$
$(type_N, l) = env'_V(x)$
for each $Rec_N \in env_{RV}$
$\beta((env''_V, env''_{RV}), (env_V^{(3)}, env_{RV}^{(3)}), sto'') = sto'$
$(env''_V, env''_{RV}) = env_{RV}(Rec_N)$
$(env_V^{(3)}, env_{RV}^{(3)}) = env'_{RV}(Rec_N)$

Table E.21: Transition rule for record assignments

[Ass-array-element]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp[arr \mapsto l''][v \mapsto 0][dimNo \mapsto 0] \vdash \langle ArrayIndex, sto\rangle \rightarrow_{exp} (v, sto'')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X\ ArrayIndex := exp;, sto\rangle \rightarrow sto'[l \mapsto v']}$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto''\rangle \rightarrow_{exp} (v', sto')$
$env_{RV}, env_V \vdash X \rightarrow (array, l')$
$l'' = sto_{array}(l')$
$dims = sto_{array}(l'')$
$l = l'' + dims + v$

Table E.22: Transition rules for array element assignment

[If-true]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto'' \rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{if } b \texttt{ then } S, sto \rangle \rightarrow sto'}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto \rangle \rightarrow_b (tt, sto'')$

[If-false]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{if } b \texttt{ then } S, sto \rangle \rightarrow sto'$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto \rangle \rightarrow_b (ff, sto')$

[If-else-true]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_1, sto'' \rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, sto \rangle \rightarrow sto'}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto \rangle \rightarrow_b (tt, sto'')$

[If-else-false]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_2, sto'' \rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, sto \rangle \rightarrow sto'}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto \rangle \rightarrow_b (ff, sto'')$

Table E.23: Transition rules if-statements

[While-true]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto'' \rangle \to sto^{(3)} \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, sto^{(3)} \rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, sto \rangle \to sto'}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto \rangle \to_b (tt, sto'')$
$env_V(break) = (boolean, l)$
$sto^{(3)}_{boolean}(l) = ff$

[While-true-break]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto'' \rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, sto \rangle \to sto'[sto_{boolean}[l \mapsto ff]]}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto \rangle \to_b (tt, sto'')$
$env_V(break) = (boolean, l)$
$sto'_{boolean}(l) = tt$

[While-false]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, sto \rangle \to sto'$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto \rangle \to_b (ff, sto')$

Table E.24: Transition rules for `while..do` loops

[Do-while-true]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto\rangle \to sto'' \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{do } S \texttt{ while } b, sto^{(3)}\rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{do } S \texttt{ while } b, sto\rangle \to sto'}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto''\rangle \to_b (tt, sto^{(3)})$
$env_V(break) = (boolean, l)$
$sto''_{boolean}(l) = ff$

[Do-while-true-break]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto\rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle\ \texttt{do } S \texttt{ while } b, sto\rangle \to sto'[sto_{boolean}[l \mapsto ff]]}$$

where $env_V(break) = (boolean, l)$
$sto'_{boolean}(l) = tt$

[Do-while-false]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto\rangle \to sto''}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle\ \texttt{do } S \texttt{ while } b, sto\rangle \to sto'}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto''\rangle \to_b (ff, sto')$

Table E.25: Transition rules for `do..while` loops

[Comp]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_1, sto\rangle \rightarrow sto'' \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_2, sto''\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_1 S_2, sto\rangle \rightarrow sto'}$$

where $env_V(return) = (boolean, l)$
$sto''_{boolean}(l) = ff$
and $env_V(break) = (boolean, l)$
$sto''_{boolean}(l) = ff$

[Comp-return]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_1, sto\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_1 S_2, sto\rangle \rightarrow sto'}$$

where $env_V(return) = (boolean, l)$
$sto'_{boolean}(l) = tt$

or $env_V(break) = (boolean, l')$
$sto'_{boolean}(l') = tt$

[Begin-end]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{begin}\ S\ \texttt{end}, sto\rangle \rightarrow sto'}$$

[Return]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle returnvalue\ \text{textbf:=}\ exp\texttt{;}, sto\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{return}\ exp\texttt{;}, sto\rangle \rightarrow sto'[sto_{boolean}[l \mapsto tt]]}$$

where $env_V(return) \rightarrow (boolean, l)$

[Break]

$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{break}\text{textbf;}, sto\rangle \rightarrow sto[sto_{boolean}[l \mapsto tt]]$

where $env_V(break) \rightarrow (boolean, l)$

[Com-empty]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \epsilon, sto\rangle \rightarrow sto$$

Table E.26: Transition rules for composite commands, begin-end, break, and return

[Proc-call]

$$\frac{\begin{array}{c} env_{PAR}, env_C, env_V, env_{RV}, env_{RT}, env_F, temp[i \mapsto 0] \vdash \langle Par_A, \emptyset, \emptyset, sto\rangle \\ \rightarrow_{ParA} (env'_V, env'_{RV}, sto'') \\ env_C, env_F, env_{RT}, env_P \vdash \langle DV, env'_V, env'_{RV}, sto''\rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto^{(3)}) \\ env_C, env_F, env''_V, env''_{RV}, env_{RT}, env_P \vdash \langle S, sto^{(3)}\rangle \rightarrow sto' \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle Proc_N(Par_A);, sto\rangle \rightarrow sto'}$$

where $env_P(Proc_N) = (S, DV, env_{PAR})$

Table E.27: Transition rules for procedure calls

[For-to-true]

$$\frac{env_C, env_F, env_V[x \mapsto (integer, l')], env_{RV}, env_{RT}, env_P \vdash \langle S, sto^{(3)}[sto_{integer}[l' \mapsto v_1]]\rangle \to sto^{(4)} \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{for}\ x := int_1\ \texttt{to}\ int_2\ \texttt{do}\ S, sto^{(4)}\rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{for}\ x := i_1\ \texttt{to}\ i_2\ \texttt{do}\ S, sto\rangle \to sto'}$$

where
$env_V(break) = (boolean, l)$
$sto^{(4)}_{boolean}(l) = ff$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \to_i (v_1, sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \to_i (v_2, sto^{(3)})$
$v_1 \leq v_2$
$int_1 = LittType^{-1}(v_1 + 1)$
$int_2 = LittType^{-1}(v_2)$
$l' = new(sto_{integer})$

[For-to-true-break]

$$\frac{env_C, env_F, env_V[x \mapsto (integer, l')], env_{RV}, env_{RT}, env_P \vdash \langle S, sto^{(3)}[sto_{integer}[l' \mapsto v_1]]\rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{for}\ x := i_1\ \texttt{to}\ i_2\ \texttt{do}\ S, sto\rangle \to sto'[sto_{boolean}[l \mapsto ff]]}$$

where
$env_V(break) = (boolean, l)$
$sto'_{boolean}(l) = tt$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \to_i (v_1, sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \to_i (v_2, sto^{(3)})$
$v_1 \leq v_2$
$l' = new(sto_{integer})$

[For-to-false]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{for}\ x := i_1\ \texttt{to}\ i_2\ \texttt{do}\ \ S, sto\rangle \to sto'$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \to_i (v_1, sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \to_i (v_2, sto')$
$v_1 > v_2$

Table E.28: Transition rules for `for..to` loops

[For-downto-true]

$$\frac{env_C, env_F, env_V[x \mapsto integer, l'], env_{RV}, env_{RT}, env_P \vdash \langle S, sto^{(3)}[sto_{integer}[l' \mapsto v_1]]\rangle \to sto^{(4)} \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{for } x \texttt{ := } int_1 \texttt{ downto } int_2 \texttt{ do } S, sto^{(4)}\rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{for } x \texttt{ := } i_1 \texttt{ downto } i_2 \texttt{ do } S, sto\rangle \to sto'}$$

where
$env_V(break) = (boolean, l)$
$sto^{(4)}_{boolean}(l) = ff$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \to_i (v_1, sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \to_i (v_2, sto^{(3)})$
$v_1 \geq v_2$
$int_1 = LittType^{-1}(v_1 - 1)$
$int_2 = LittType^{-1}(v_2)$
$l' = new(sto_{integer})$

[For-downto-true-break]

$$\frac{env_C, env_F, env_V[x \mapsto type_N, l'], env_{RV}, env_{RT}, env_P \vdash \langle S, sto^{(3)}[sto_{type_N}[l' \mapsto v_1]]\rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{for } x \texttt{ := } i_1 \texttt{ downto } i_2 \texttt{ do } S, sto\rangle \to sto'[sto_{boolean}[l \mapsto ff]]}$$

where
$env_V(break) = (boolean, l)$
$sto'_{boolean}(l) = tt$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \to_i (v_1, sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \to_i (v_2, sto^{(3)})$
$v_1 \geq v_2$
$l' = new(sto_{integer})$

[For-downto-false]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{for } x \texttt{ := } i_1 \texttt{ downto } i_2 \texttt{ do} S, sto\rangle \to sto'$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \to_i (v_1, sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \to_i (v_2, sto')$
$v_1 < v_2$

Table E.29: Transition rules for `for..downto` loops

[For-each-set]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{foreach}\ \ type_N\ x\ \texttt{in}\ X\ \texttt{do}\ S, sto\rangle \rightarrow sto^{(size)}[sto_{boolean}[l' \mapsto \mathit{ff}]]$$

where
$env_{RV}, env_V \vdash X \rightarrow (set, l)$
$env_V(break) = (boolean, l')$
$(size, -) = sto_{set}(l)$
$env_C, env_F, env_V[x \mapsto (type_N, l_i)], env_{RV}, env_{RT}, env_P \vdash \langle \texttt{if}\ (return = \texttt{false}\ \texttt{and}\ break = \texttt{false})\ \texttt{then}\ S, sto^{(i-1)}\rangle \rightarrow sto^{(i)}$
$i = 1, 2, \ldots, size$
$sto^{(0)} = sto$
$l_i = \Gamma(sto_{set}, l_{i-1}, 1)$
$l_0 = l$

[For-each-where-set]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{foreach}\ type_N\ x\ \ \texttt{in}\ X\ \texttt{do}\ S\ \texttt{where}\ b, sto\rangle \rightarrow sto^{(size)}[sto_{boolean}[l' \mapsto \mathit{ff}]]$$

where
$env_{RV}, env_V \vdash X \rightarrow (set, l)$
$env_V(break) = (boolean, l')$
$(size, -) = sto_{set}(l)$
$env_C, env_F, env_V[x \mapsto (type_N, l_i)], env_{RV}, env_{RT}, env_P \vdash \langle \texttt{if}\ (return = \texttt{false}\ \texttt{and}\ break = \texttt{false})\ \texttt{then if}\ b\ \texttt{then}\ S, sto^{(i-1)}\rangle \rightarrow sto^{(i)}$
$i = 1, 2, \ldots, size$
$sto^{(0)} = sto$
$l_i = \Gamma(sto_{set}, l_{i-1}, 1)$
$l_0 = l$

Table E.30: Transition rules for `for..each` loops for sets

[For-each-array]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{foreach}\ type_N\ x\ \texttt{in}\ X\ \texttt{do}\ S, sto\rangle \to sto^{(noelem)}[sto_{boolean}[l' \mapsto ff]]$$

where
$env_{RV}, env_V \vdash X \to (array, l)$
$env_V(break) = (boolean, l')$
$dimLoc = sto_{array}(l)$
$nodims = sto_{array}(dimLoc)$
$noelem = size(1) \cdot size(2) \cdot \ldots \cdot size(nodims)$
$size(y) = sto_{array}(dimLoc + y)$
$i = 1, 2, \ldots, noelem$
$env_C, env_F, env_V[x \mapsto (type_N, l_i)], env_{RV}, env_{RT}, env_P \vdash \langle \texttt{if}\ (return = \texttt{false}\ \texttt{and}\ break = \texttt{false})\ \texttt{then}\ S, sto^{(i-1)}\rangle \to sto^{(i)}$
$sto^{(0)} = sto$
$l_i = dimLoc + nodims + i$

[For-each-where-array]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{foreach}\ type_N\ x\ \texttt{in}\ X\ \texttt{do}\ S\ \texttt{where}\ b, sto\rangle \to sto^{(noelem)}[sto_{boolean}[l' \mapsto ff]]$$

where
$env_{RV}, env_V \vdash X \to (array, l)$
$env_V(break) = (boolean, l')$
$dimLoc = sto_{array}(l)$
$nodims = sto_{array}(dimLoc)$
$noelem = size(1) \cdot size(2) \cdot \ldots \cdot size(nodims)$
$size(y) = sto_{array}(dimLoc + y)$
$i = 1, 2, \ldots, noelem$
$env_C, env_F, env_V[x \mapsto (type_N, l_i)], env_{RV}, env_{RT}, env_P \vdash \langle \texttt{if}\ (return = \texttt{false}\ \texttt{and}\ break = \texttt{false})\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ S, sto^{(i-1)}\rangle \to sto^{(i)}$
$sto^{(0)} = sto$
$l_i = dimLoc + nodims + i$

Table E.31: Transition rules for `for..each` loops for arrays

[Plusplus]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X{+}{+}, sto\rangle \rightarrow sto[sto_{type_N}[l \mapsto v + 1]]$

where $env_{RV}, env_V \vdash X \rightarrow (type_N, l)$
$type_N \in \{integer, float\}$
$v = sto_{type_N}(l)$

[Minusminus]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X{-}{-}, sto\rangle \rightarrow sto[sto_{type_N}[l \mapsto v - 1]]$

where $env_{RV}, env_V \vdash X \rightarrow (type_N, l)$
$type_N \in \{integer, float\}$
$v = sto_{type_N}(l)$

Table E.32: Transition rules for `++` and `--` commands

[Switch]

$$\frac{env_C, env_F, env_V[break \mapsto (boolean, l')], env_{RV}, env_{RT}, env_P, temp[v \mapsto v'] \vdash \langle S, sto[l' \mapsto \mathit{ff}]\rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{switch}\ X\ S\ \texttt{endswitch}\ , sto\rangle \to sto'}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X, sto\rangle \to_{exp} (v', sto)$
$l' = new(sto_{boolean})$

[Switch-case-true]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp \vdash \langle S, sto\rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp \vdash \langle \texttt{case}\ int\text{: } S, sto\rangle \to sto'[sto_{boolean}[l \mapsto \mathit{tt}]]}$$

where $v' = temp(v)$
$v'' = LittType(int)$
$v' = v''$
$(boolean, l) = env_V(break)$

[Switch-case-false]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp \vdash \langle \texttt{case}\ int\text{: } S, sto\rangle \to sto$$

where $v' = temp(v)$
$v'' = LittType(int)$
$v' \neq v''$

[Switch-caseInterval-true]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp \vdash \langle S, sto\rangle \to sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp \vdash \langle \texttt{case}\ int_1..int_2\text{: } S, sto\rangle \to sto'[sto_{boolean}[l \mapsto \mathit{tt}]]}$$

where $v' = temp(v)$
$v'' = LittType(int_1)$
$v^{(3)} = LittType(int_2)$
$v'' \leq v' \leq v^{(3)}$
$(boolean, l) = env_V(break)$

Table E.33: Transition rules for switch

[Switch-caseInterval-false]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp \vdash \langle \texttt{case}\ int_1..int_2\texttt{:}\ S, sto\rangle \rightarrow sto$$

where $v' = temp(v)$
$v'' = LittType(int_1)$
$v^{(3)} = LittType(int_2)$
$(v' < v'') \vee (v' > v^{(3)})$

[Switch-default-breakFalse]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S, sto\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{default}\ \texttt{:}\ S, sto\rangle \rightarrow sto'}$$

where $env_V(break) = (boolean, l)$

$sto_{boolean}(l) = \mathit{ff}$

[Switch-default-breakTrue]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{default}\ \texttt{:}\ S, sto\rangle \rightarrow sto[sto_{boolean}[l \mapsto \mathit{ff}]]$$

where $env_V(break) = (boolean, l)$

$sto_{boolean}(l) = \mathit{tt}$

Table E.34: Transition rules for switch continued

E.9 Procedures and Functions in Standard Environment

The procedures and functions in the standard environment are on the semantic form of commands and expressions respectively.

[getVertex]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (gLoc, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle w, sto''\rangle \rightarrow_{w} (v, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle getVertex(exp, w);, sto\rangle \rightarrow_{exp} (Loc, sto')}$$

where Loc for which $sto_{graph}(\Gamma(sto'_{graph}, propLoc, i)) = (v, -)$
$0 < i \leq noVertices$
$sto'_{graph}(gLoc) = (noVertices, propLoc)$

[vertices]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (gLoc, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle vertices(exp);, sto\rangle \rightarrow_{exp} (sLoc, sto^{(noVertices)})}$$

where $sLoc = new(sto_{set})$
$env_C, env_F, env_V[graph \mapsto (graph, gLoc)][vertex \mapsto (vertex, vLoc_i)]],$
$env_{RV}, env_{RT}, env_P \vdash \langle addToSet(graph, vertex);, sto^{(i-1)}\rangle$
$\rightarrow sto^{(i)}$
$sto^{(0)} = sto'$
$vLoc_i = \Gamma(sto^{(i-1)}_{graph}, vLoc_{i-1}, 1)$
$vLoc_0 = propLoc$
$sto'_{graph}(gLoc) = (noVertices, propLoc)$
$i = 1, 2, \cdots, noVertices$

Table E.35: Transition rules for getVertex and vertices

[nameOfVertex]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (vLoc, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{NameOfVertex}(exp);, sto\rangle \rightarrow_{exp} (v, sto')}$$

where $sto'_{graph}(vLoc) = (v, -)$

[sizeOfSet]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (sLoc, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{sizeOfSet}(exp);, sto\rangle \rightarrow_{exp} (v, sto')}$$

where $sto'_{set}(sLoc) = (s, -)$

Table E.36: Transition rules for nameOfVertexx and sizeOfSet from the Standard Environment

[addVertex]

$$\frac{\begin{array}{l} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_{i=1}S_{i=2}\ldots S_{i=noProp}, sto^{(3)}\rangle \rightarrow sto^{(4)} \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle extend(propLoc, graph, noVertices), sto^{(4)}\rangle \rightarrow sto^{(5)} \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle extendmatrix(newLoc, graph, noVertices), sto^{(5)}\rangle \rightarrow sto' \end{array}}{\begin{array}{l} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle addVertex(exp, w);, sto\rangle \rightarrow \\ sto'[sto_{graph}[gLoc \mapsto (noVertices + 1, -)][newLoc \mapsto (v, -)]] \end{array}}$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (gLoc, sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle w, sto''\rangle \rightarrow_{w} (v, sto^{(3)})$
$sto^{(3)}_{graph}(gLoc) = (noVertices, propLoc)$
$sto^{(3)}_{graphProp}(propLoc) = (noProp, -, -)$
$i = 1, 2, \ldots, noProp$
$l_0 = propLoc$
$(l'_i, type_i, l_i) = sto^{(3)}_{prop}(l_{i-1})$
$S_i = extend(l'_i, label, noVertices) \quad (if\ type_i = label)$
$S_i = extendmatrix(l'_i, weight, noVertices) \quad (if\ type_i = weight)$
$newLoc = \Gamma(sto^{(5)}_{graph}, propLoc, noVertices + 1)$

Table E.37: Transition rule for the addVertex procedure.

[Extend]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle extend(l, type_N, int), sto\rangle \rightarrow sto[sto_{type_N}[l' \mapsto (-, newLoc)][newLoc \mapsto (nil, nextLoc)]]$$

where
$$l' = \Gamma(sto_{type_N}, l, int)$$
$$(-, nextLoc) = sto_{type_N}(l')$$
$$newLoc = new(type_N)$$

Table E.38: Helper transition rules for extending a block of size int at location l in sto_{type_N} with an extra element.

[Extend-matrix]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_{(i=1)}S_{(i=2)} \ldots S_{(i=int)}, sto\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \; extendmatrix(l, type_N, int), sto\rangle \rightarrow sto'[sto_{type_N}[prevLoc \mapsto (-, newLoc)][newLoc + i - 1 \mapsto (nil, newLoc + i)][newLoc + int - 1 \mapsto (nil, nil)]]}$$

where
$$l_i = \Gamma(sto_{type_N}, l_{i-1}, int) \qquad (i > 0)$$
$$l_0 = l$$
$$i = 1, 2, \ldots, int$$
$$newLoc = new(sto'_{type_N})$$
$$S_i = extend(l_i, type_N, int)$$
$$lPrev = \Gamma(sto'_{type_n}, l, (int) \cdot (int + 1))$$

Table E.39: Helper transition rules for extending a matrix of int rows at location l in sto_{type_N} with an extra row and column.

[RemoveVertex]

$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_{i=1} S_{i=2} \ldots S_{i=noProp}, sto^{(3)} \rangle \rightarrow sto^{(4)}$

$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle shrink(propLoc, graph, vNo), sto^{(4)} \rangle \rightarrow sto^{(5)}$

$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle shrinkmatrix(eLocs, graph, vNo, noVertices), \quad sto^{(5)} \rangle \rightarrow sto'$

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle removevertex(exp_1, exp_2);, sto \rangle \rightarrow sto'[sto_{graph}[gLoc \mapsto ((noVertices - 1), -)]]$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto \rangle \rightarrow_{exp} (gLoc, sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto'' \rangle \rightarrow_{exp} (vLoc, sto^{(3)})$
vNo for which $\Gamma(sto_{graph}, propLoc, vNo) = vLoc$
and $0 < vNo < noVertices$
$sto^{(3)}_{graph}(gLoc) = (noVertices, propLoc)$
$sto^{(3)}_{graphprop}(propLoc) = (-, noprop, -)$
$eLocs = \Gamma(sto^{(3)}_{graph}, propLoc, noVertices)$
$i = 1, 2, \ldots, noProp$
$l_0 = propLoc$
$(l'_i, type_i, l_i) = sto^{(3)}_{graphProp}(l_{i-1})$
$S_i = shrink(l'_i, label, noVertices)$ if $type_i = label$
$S_i = shrinkmatrix(l'_i, weight, noVertices)$ if $type_i = weight$

Table E.40: Transition rules for the RemoveVertex procedure.

[Shrink]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle shrink(l, type_N, int), sto \rangle \rightarrow sto[sto_{type_N}[l' \mapsto (-, l^{(3)})]]$$

where
$l' = \Gamma(sto_{type_N}, l, int - 1)$
$sto_{type_N}(l') = (-, l'')$
$sto_{type_N}(l'') = (-, l^{(3)})$

Table E.41: Helper transition rule removing element number int, counting from location l in sto_{type_N}.

[Shrink-matrix]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle S_{(i=1)} S_{(i=2)} \ldots S_{(i=int_2-1)}, sto\rangle \rightarrow sto'}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle shrinkmatrix(l, type_N \quad, int_1, int_2), sto\rangle \rightarrow sto'[sto_{type_N}[l' \mapsto (-, l'')]]}$$

where
$i = 0, 1, \ldots, int_2 - 1$
$l_i = l \qquad (i = 0)$
$l_i = \Gamma(sto_{type_N}, l_{i-1}, int_2)$
$S_i = shrink(l_i, type_N, int_1) \qquad (i \neq int_1 - 1)$
$S_i = \epsilon \qquad (i = int_1 - 1)$
$l' = \Gamma(sto'_{type_N}, l, (int_2 - 1) \cdot (int_1 - 1)$
$l'' = \Gamma(sto'_{type_N}, l', int_2 + 1)$

Table E.42: Helper transition rule for removing element int_1 from the matrix(with int_2 rows) located at location l in sto_{type_N}.

[Is-vertex-in-graph-true]

$$\frac{\begin{array}{c} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc, sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc, sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle isVertexInGraph(exp_1, exp_2);, sto\rangle \rightarrow_b (tt, sto')}$$

where
$(noVertices, propLoc) = sto'_{graph}(gLoc)$
$l_i = \Gamma(sto'_{graph}, l_{i-1}, 1)$
$l_0 = propLoc$
$l_i = vLoc$ for some $i \in \{1, 2, ..., noVertices\}$

[Is-vertex-in-graph-false]

$$\frac{\begin{array}{c} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc, sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc, sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle isVertexInGraph(exp_1, exp_2);, sto\rangle \rightarrow_b (ff, sto')}$$

where
$(noVertices.propLoc) = sto'_{graph}(gLoc)$
$l_i = \Gamma(sto'_{graph}, l_{i-1}, 1)$
$l_0 = propLoc$
$l_i \neq \ vLoc$ for all $i \in \{1, 2, ..., noVertices\}$

Table E.43: Transition rules for isVertex from the standard environment

[isEdge-true]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc' \times vLoc'', sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle isEdge(exp_1, exp_2);, sto\rangle \rightarrow_b (tt, sto')}$$

$$\begin{array}{l} sto'_{graph}(gLoc) = (noVertices, propLoc) \\ i \text{ for which } \Gamma(sto'_{graph}, propLoc, i) = vLoc' \\ j \text{ for which } \Gamma(sto'_{graph}, propLoc, j) = vLoc'' \\ 0 < i < noVertices \\ 0 < j < noVertices \\ eLoc = \Gamma(sto'_{graph}, propLoc, noVertices + (i-1) \cdot noVertices + j) \\ sto'_{graph}(eLoc) = 1 \end{array}$$

[isEdge-false]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc' \times vLoc'', sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle isEdge(exp_1, exp_2)\ ;, sto\rangle \rightarrow_b (ff, sto')}$$

$$\begin{array}{l} sto'_{graph}(gLoc) = (noVertices, propLoc) \\ i \text{ for which } \Gamma(sto'_{graph}, propLoc, i) = vLoc' \\ j \text{ for which } \Gamma(sto'_{graph}, propLoc, j) = vLoc'' \\ 0 < i < noVertices \\ 0 < j < noVertices \\ eLoc = \Gamma(sto'_{graph}, propLoc, noVertices + (i-1) \cdot noVertices + j) \\ sto_{graph}(eLoc) = 0 \end{array}$$

Table E.44: Transition rules for isEdge from the Standard Environment

[AddEdge-graph]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle addEdge(exp_1, exp_2);, sto\rangle \rightarrow sto'[sto_{graph}[eLoc' \mapsto 1][eLoc'' \mapsto 1]]$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc', sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc' \times vLoc'', sto')$
$sto'_{graph}(gLoc) = (noVertices, propLoc)$
i for which $\Gamma(sto'_{graph}, propLoc, i) = vLoc'$
j for which $\Gamma(sto'_{graph}, propLoc, j) = vLoc''$
$0 < i < noVertices$
$0 < j < noVertices$
$eLoc' = \Gamma(sto'_{graph}, propLoc, noVertices + (i-1) \cdot noVertices + j)$
$eLoc'' = \Gamma(sto'_{graph}, propLoc, noVertices + (j-1) \cdot noVertices + i)$

[AddEdge-diGraph]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle addEdge(exp_1, exp_2);, sto\rangle \rightarrow sto'[sto_{graph}[eLoc' \mapsto 1]]$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc', sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc' \times vLoc'', sto')$
$sto'_{graph}(gLoc) = (noVertices, propLoc)$
i for which $\Gamma(sto'_{graph}, propLoc, i) = vLoc'$
j for which $\Gamma(sto'_{graph}, propLoc, j) = vLoc''$
$0 < i < noVertices$
$0 < j < noVertices$
$eLoc' = \Gamma(sto'_{graph}, propLoc, noVertices + (i-1) \cdot noVertices + j)$

Table E.45: Transition rules for addEdge from the Standard Environment

[RemoveEdge-graph]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle removeEdge(exp_1, exp_2);, sto\rangle \rightarrow sto'[sto_{graph}[eLoc' \mapsto 0][eLoc'' \mapsto 0]]$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc', sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc' \times vLoc'', sto')$
$sto'_{graph}(gLoc) = (noVertices, propLoc)$
i for which $\Gamma(sto'_{graph}, propLoc, i) = vLoc'$
j for which $\Gamma(sto'_{graph}, propLoc, j) = vLoc''$
$0 < i < noVertices$
$0 < j < noVertices$
$eLoc' = \Gamma(sto'_{graph}, propLoc, noVertices + (i-1) \cdot noVertices + j)$
$eLoc'' = \Gamma(sto'_{graph}, propLoc, noVertices + (j-1) \cdot noVertices + i)$

[RemoveEdge-diGraph]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle removeEdge(exp_1, exp_2);, sto\rangle \rightarrow sto'[sto_{graph}[eLoc' \mapsto 0]]$$

where
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (gLoc', sto'')$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (vLoc' \times vLoc'', sto')$
$sto'_{graph}(gLoc) = (noVertices, propLoc)$
i for which $\Gamma(sto'_{graph}, propLoc, i) = vLoc'$
j for which $\Gamma(sto'_{graph}, propLoc, j) = vLoc''$
$0 < i < noVertices$
$0 < j < noVertices$
$eLoc' = \Gamma(sto'_{graph}, propLoc, noVertices + (i-1) \cdot noVertices + j)$

Table E.46: Transition rules for removeEdge from the Standard Environment

[Add-to-set-true]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (sLoc, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (v, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle addToSet(exp_1, exp_2), sto\rangle \rightarrow sto'[sto_{set}[sLoc \mapsto (noElements + 1, l)][l \mapsto (v, l')]]}$$

where $l = new(sto_{set})$
$l_0 = sLoc$
$l_i = \Gamma(sto'_{set}, l_{i-1}, 1)$
$v_i = sto_{set}(l_i)$
$v_i \neq v$ for all $i \in \{1, 2, ..., noElements\}$
$sto'_{set}(sLoc) = (noElements, l')$

[Add-to-set-false]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (sLoc, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (v, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle addToSet(exp_1, exp_2), sto\rangle \rightarrow sto'}$$

where $l_0 = sLoc$
$l_i = \Gamma(sto'_{set}, l_{i-1}, 1)$
$v_i = sto'_{set}(l_i)$
$v_i = v$ for some $i \in \{1, 2, ..., noElements\}$
$sto'_{set}(sLoc) = (noElements, -)$

Table E.47: Transition rules for the addToSet procedure

[Remove-from-set]

$$\frac{\begin{array}{l} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (sLoc, sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (v, sto') \end{array}}{\begin{array}{l} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle removeFromSet(exp_1, exp_2), sto\rangle \\ \rightarrow sto'[sto_{set}[prevLoc \mapsto (-, nextLoc)]] \end{array}}$$

$$\begin{array}{l} sto'_{set}(sLoc) = (noElements, -) \\ i \text{ for which } \ sto'_{set}(\Gamma(sto'_{set}, sLoc, i)) = v \\ Loc = \Gamma(sto'_{set}, sLoc, i) \\ sto'_{set}(Loc) = (v, nextLoc) \\ prevLoc = \Gamma(sto'_{set}, sLoc, i-1) \\ 0 < i < noElements \end{array}$$

Table E.48: Transition rules for the removeFromSet procedure

[Is-in-set-true]

$$\frac{\begin{array}{l} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (sLoc, sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (v, sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle isInSet(exp_1, exp_2), sto\rangle \rightarrow_b (tt, sto')}$$

$$\begin{array}{lll} \text{where} & l_0 = sLoc & \\ & l_i = \Gamma(sto'_{set}, l_{i-1}, 1) & \\ & sto'_{set}(sLoc) = (noElements, -) & \\ & v_i = sto_{set}(l_i) & \\ & v_i = v \text{ for some } i \in \{1, 2, ..., noElements\} & \end{array}$$

[Is-in-set-false]

$$\frac{\begin{array}{l} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (sLoc, sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (v, sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle isInSet(exp_1, exp_2), sto\rangle \rightarrow_b (ff, sto')}$$

$$\begin{array}{lll} \text{where} & l_0 = sLoc & \\ & l_i = \Gamma(sto'_{set}, l_{i-1}, 1) & \\ & sto'_{set}(sLoc) = (noElements, -) & \\ & v_i = sto_{set}(l_i) & \\ & v_i \neq v \text{ for all } i \in \{1, 2, ..., noElements\} & \end{array}$$

Table E.49: Transition rules for the addToSet procedure

E.10 Expressions

All expressions are on the form $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto \rangle \rightarrow_{exp} (v, sto')$.

[Var-val]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X, sto \rangle \rightarrow_{exp} (v, sto)$

where $env_{RV}, env_V \vdash X \rightarrow (type_N, l)$
$v = sto_{type_N}(l)$
$type_N \in primitiveTypes$

[Const-val]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle x, sto \rangle \rightarrow_{exp} (v, sto)$

where $env_C(x) = (type_N, l)$
$v = sto_{type_N}(l)$
$type_N \in primitiveTypes$

[Var-val-composite]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X, sto \rangle \rightarrow_{exp} (v, sto)$

where $env_{RV}, env_V \vdash X \rightarrow (type_N, l)$
$v = l$
$type_N \in \{set, array, graph, label, weight\}$

[Record-val]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X, sto \rangle \rightarrow_{exp} (v, sto)$

where $env_{RV}, env_V \vdash X \rightarrow (env'_V, env'_{RV})$
$v = (env'_V, env'_{RV})$

Table E.50: Transition rules for variable values

[Array-val]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp[arr \mapsto l'][v \mapsto 0] \quad [dimNo \mapsto 0] \vdash \langle ArrayIndex, sto\rangle \rightarrow_{exp} (v', sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X\ ArrayIndex, sto\rangle \rightarrow_{exp} (v, sto')}$$

where $env_{RV}, env_V \vdash X \rightarrow (array, l)$
$l' = sto_{array}(l)$
$dims = sto_{array}(l')$
$v = sto_{array}(l' + dims + v')$

[Array-index]

$$\frac{\begin{array}{l} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, \\ temp[dimNo \mapsto dimNo'][v \mapsto v''] \vdash \langle ArrayIndex, sto''\rangle \rightarrow_{exp} (v', sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp \vdash \langle [i]\ ArrayIndex, sto\rangle \rightarrow_{exp} (v', sto')}$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i, sto\rangle \rightarrow_i (v^{(3)}, sto'')$
$l = temp(arr)$
$dims = sto_{array}(l)$
$dimNo' = temp(dimNo) + 1$
$v'' = temp(v) + (v^{(3)} - 1) \cdot (size(dimNo' + 1) \cdot$
$\quad size(dimNo' + 2) \cdot \ldots \cdot size(dimNo' + dims))$
$size(p) = l + p$

[Array-index-final]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P, temp \vdash \langle [i], sto\rangle \rightarrow_{exp} (v', sto')$$

where $env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i, sto\rangle \rightarrow_i (v'', sto')$
$v' = temp(v) + v''$

Table E.51: Transition rules for array values and index

[Label-val]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (vLoc, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X(exp), sto\rangle \rightarrow_{exp} (v, sto')}$$

where
$env_{RV}, env_V \vdash X \rightarrow (label, lLoc)$
$(graphLoc, -) = sto_{label}(lLoc)$
$(noVertices, propLoc) = sto'_{graph}(graphLoc)$
vNo for which $\Gamma(sto'_{graph}, propLoc, vNo) = vLoc$
and $0 < vNo \leq noVertices$
$(v, -) = sto'_{label}(\Gamma(sto'_{label}, lLoc, vNo))$

[Weight-val]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow_{exp} (Loc' \times Loc'', sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle X(exp), sto\rangle \rightarrow_{exp} (v, sto')}$$

where
$env_{RV}, env_V \vdash X \rightarrow (weight, wLoc)$
$(graphLoc, -) = sto'_{weight}(wLoc)$
$(noVertices, propLoc) = sto'_{graph}(graphLoc)$
i for which $\Gamma(sto'_{graph}, propLoc, i) = Loc'$
and $0 < i \leq noVertices$
j for which $\Gamma(sto'_{graph}, propLoc, j) = Loc''$
and $0 < j \leq noVertices$
$eNo = (i-1) \cdot noVertices + j$
$(v, -) = sto'_{weight}(\Gamma(sto'_{weight}, wLoc, eNo))$

[Edge-val]

$$\frac{\begin{array}{c} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_1, sto\rangle \rightarrow_{exp} (Loc', sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp_2, sto''\rangle \rightarrow_{exp} (Loc'', sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle (exp_1, exp_2), sto\rangle \rightarrow_{exp} (Loc' \times Loc'', sto')}$$

Table E.52: Transition rules for label, weight, and edge value expressions

[Func-Call-Prim]

$$\frac{\begin{array}{ll} env_{PAR}, env_C, env_V, env_{RV}, env_{RT}, env_F, temp[i \mapsto 0] \vdash & \langle Par_A, \emptyset, \emptyset, sto\rangle \rightarrow_{ParA} \\ & (env'_V, env'_{RV}, sto'') \\ env_C, env_F, env_{RT}, env_P \vdash \langle DV, env'_V, env'_{RV}, sto''\rangle \rightarrow_{DV} & (env''_V, env''_{RV}, sto^{(3)}) \\ env_C, env_F, env''_V[returnvalue \mapsto (type_N, l)], & env''_{RV}, env_{RT}, env_P \vdash \langle S \\ & , sto^{(3)}[sto_{type_N}[l \mapsto nil]]\rangle \rightarrow sto' \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle Func_N(Par_A), sto\rangle \rightarrow_{exp} (v, sto')}$$

where $env_F(Func_N) = (S, DV, type_N, env_{PAR})$
$type_N \in primitiveTypes \cup \{graph, digraph, array, set\}$
$l = new(sto_{type_N})$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle returnvalue, sto'\rangle \rightarrow_{exp} (v, sto')$

[Func-Call-Rec]

$$\frac{\begin{array}{ll} env_{PAR}, env_C, env_V, env_{RV}, env_{RT}, env_F, temp[i \mapsto 0] \vdash \langle & Par_A, \emptyset, \emptyset, sto\rangle \\ & \rightarrow_{ParA} (env'_V, env'_{RV}, sto'') \\ env_C, env_F, env_{RT}, env_P \vdash \langle DV\ type_N\ returnvalue;, env'_V & , env'_{RV}, sto''\rangle \rightarrow_{DV} \\ & (env''_V, env''_{RV}, sto^{(3)}) \\ env_C, env_F, env''_V, env''_{RV}, env_{RT}, env_P \vdash \langle S, sto^{(3)}\rangle \rightarrow sto' & \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle Func_N(Par_A), sto\rangle \rightarrow_{exp} (v, sto')}$$

where $env_F(Func_N) = (S, DV, type_N, env_{PAR})$
$type_N \in RecordTypes$
$env''_{RV}(returnvalue) = (env_V^{(3)}, env_{RV}^{(4)})$
$v = (env_V^{(3)}, env_{RV}^{(4)})$

Table E.53: Transition rules for function calls

E.11 Actual Parameters

Actual parameters are on the form
$env_{PAR}, env_V, env_{RV}, env_{RT}, env_F, temp \vdash \langle Par_A, env''_V, env''_{RV}, sto\rangle \rightarrow_{ParA} (env'_V, env'_{RV}, sto')$.

[Par$_A$-prim]

$$\frac{\begin{array}{c} env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ var := x', env'_V[x' \mapsto (type_N, l)], env'_{RV}, \\ sto'[sto_{type_N}[l \mapsto v]]\rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto'') \\ env_{PAR}, env_C, env_V, env_{RV}, env_F, env_P, temp[i \mapsto i'] \vdash \langle Par_A, env''_V, env''_{RV}, \\ sto''\rangle \rightarrow_{ParA} (env^{(3)}_V, env^{(3)}_{RV}, sto^{(3)}) \end{array}}{\begin{array}{c} env_{PAR}, env_C, env_V, env_{RV}, env_{RT}, env_F, env_P temp \vdash \langle exp, Par_A, env'_V, env'_{RV}, sto\rangle \\ \rightarrow_{ParA} (env^{(3)}_V, env^{(3)}_{RV}, sto^{(3)}) \end{array}}$$

where
$i' = temp(i) + 1$
$env_{PAR}(i') = (var, type_N, ff)$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow (v, sto')$
$l = new(type_N)$
$type_N \in primitiveTypes$

[Par$_A$-comp]

$$\frac{\begin{array}{c} env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ var := x', env'_V[x' \mapsto (type'_N, l)], env'_{RV}, \\ sto'\rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto'') \\ env_{PAR}, env_C, env_V, env_{RV}, env_F, env_P, temp[i \mapsto i'] \vdash \langle Par_A, env''_V, env''_{RV}, \\ sto''\rangle \rightarrow_{ParA} (env^{(3)}_V, env^{(3)}_{RV}, sto^{(3)}) \end{array}}{\begin{array}{c} env_{PAR}, env_C, env_V, env_{RV}, env_{RT}, env_F, env_P temp \vdash \langle exp, Par_A, env'_V, env'_{RV}, sto\rangle \\ \rightarrow_{ParA} (env^{(3)}_V, env^{(3)}_{RV}, sto^{(3)}) \end{array}}$$

where
$i' = temp(i) + 1$
$env_{PAR}(i') = (var, type_N, ff)$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto\rangle \rightarrow (l, sto')$
$type'_N = set$ if $type_N = set\ of\ type$
$type'_N = array$ if $type_N = array\ of\ type$
$type'_N = label$ if $type_N = label\ of\ type$
$type'_N = weight$ if $type_N = weight\ of\ type$

Table E.54: Transition rules for actual parameters

E.12 String Expressions

String expressions are expressions and therefore use the semantic form of the general expressions (can be found in E.57

[Par$_A$-ref]

$$\frac{\begin{array}{c} env_C, env_F, env_{RT}, env_P \vdash \langle type'_N \texttt{ ref } var \texttt{ := } ref, env'_V[ref \mapsto (type_N, l)], \\ env'_{RV}, sto\rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto') \\ env_{PAR}, env_C, env_V, env_{RV}, env_F, env_P, temp[i \mapsto i'] \vdash \langle Par_A, env''_V, env''_{RV}, sto'\rangle \\ \rightarrow_{ParA} (env^{(3)}_V, env^{(3)}_{RV}, sto'') \end{array}}{\begin{array}{c} env_{PAR}, env_V, env_{RV}, env_{RT}, env_F, env_P, temp \vdash \langle X\texttt{,}Par_A, env'_V, env'_{RV}, sto\rangle \\ \rightarrow_{ParA} (env^{(3)}_V, env^{(3)}_{RV}, sto'') \end{array}}$$

where $i' = temp(i) + 1$
$env_{PAR}(i') = (var, type'_N, \not{t})$
$env_{RV}, env_V \vdash X \rightarrow (type_N, l)$

[Par$_A$-empty]

$$env_{PAR}, env_C, env_V, env_{RV}, env_{RT}, env_F, env_P, temp \vdash \langle \epsilon, env'_V, env'_{RV}, sto\rangle \rightarrow_{ParA} (env'_V, env'_{RV}, sto)$$

Table E.55: Transition rules for actual reference parameters, and the empty parameter

[Par$_A$-rec]

$$\frac{\begin{array}{c} env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ var\ \texttt{:=}\ x', env'_V, env'_{RV}[x' \mapsto (env_V^{(4)}, env_{RV}^{(4)})], \\ sto' \rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto'') \\ env_{PAR}, env_C, env_V, env_{RV}, env_F, env_P, temp[i \mapsto i'] \vdash \langle Par_A, env''_V, env''_{RV}, \\ sto'' \rangle \rightarrow_{ParA} (env_V^{(3)}, env_{RV}^{(3)}, sto^{(3)}) \end{array}}{\begin{array}{c} env_{PAR}, env_C, env_V, env_{RV}, env_{RT}, env_F, env_P temp \vdash \langle exp\texttt{,}Par_A, env'_V, env'_{RV}, sto \rangle \\ \rightarrow_{ParA} (env_V^{(3)}, env_{RV}^{(3)}, sto^{(3)}) \end{array}}$$

where $i' = temp(i) + 1$
$env_{PAR}(i') = (var, type_N, \mathit{ff})$
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle exp, sto \rangle \rightarrow ((env_V^{(4)}, env_{RV}^{(4)}), sto')$
$type_N \in RecordType$

[Par$_A$-rec-ref]

$$\frac{\begin{array}{c} env_C, env_F, env_{RT}, env_P \vdash \langle type_N\ \texttt{ref}\ var\ \texttt{:=}\ x', env'_V, env'_{RV}[x' \mapsto (env_V^{(4)}, env_{RV}^{(4)})], \\ sto \rangle \rightarrow_{DV} (env''_V, env''_{RV}, sto'') \\ env_{PAR}, env_C, env_V, env_{RV}, env_F, env_P, temp[i \mapsto i'] \vdash \langle Par_A, env''_V, env''_{RV}, \\ sto'' \rangle \rightarrow_{ParA} (env_V^{(3)}, env_{RV}^{(3)}, sto') \end{array}}{\begin{array}{c} env_{PAR}, env_C, env_V, env_{RV}, env_{RT}, env_F, env_P temp \vdash \langle exp\texttt{,}Par_A, env'_V, env'_{RV}, sto \rangle \\ \rightarrow_{ParA} (env_V^{(3)}, env_{RV}^{(3)}, sto') \end{array}}$$

where $i' = temp(i) + 1$
$env_{PAR}(i') = (var, type_N, \mathit{ff})$
$env_{RV}, env_V \vdash X \rightarrow (env_V^{(4)}, env_{RV}^{(4)})$
$type_N \in RecordType$

Table E.56: Transition rules for actual record parameters

[Concatenation]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle w_1, sto\rangle \rightarrow_w (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle w_2, sto''\rangle \rightarrow_w (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle w_1 \& w_2, sto\rangle \rightarrow_w (v, sto')}$$

where $v = v_1 \circ v_2$

[Litt-string]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle str, sto\rangle \rightarrow_w (v, sto)$$

where $LittType(str) = v$

Table E.57: Transition rules for actual parameters

E.13 Boolean Expressions

Boolean expressions are expressions and therefore use the semantic form of the general expressions.

$$
\begin{array}{rll}
infty < n & \rightarrow \mathit{ff} & \text{where } n \neq infty \\
infty > n & \rightarrow \mathit{tt} & \text{where } n \neq infty \\
n = infty & \rightarrow \mathit{ff} & \text{where } n \neq infty \\
infty = infty & \rightarrow \mathit{tt} & \\
-infty = -infty & \rightarrow \mathit{tt} & \\
n \neq infty & \rightarrow \mathit{tt} & \text{where } n \neq infty \\
-infty < n & \rightarrow \mathit{tt} & \text{where } n \neq -infty \\
-infty > n & \rightarrow \mathit{ff} & \text{where } n \neq -infty \\
-infty = n & \rightarrow \mathit{ff} & \text{where } n \neq -infty \\
n = -infty & \rightarrow \mathit{ff} & \text{where } n \neq -infty
\end{array}
$$

[Equals1]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1{=}n_2, sto\rangle \rightarrow_b (tt, sto')}$$

where $v_1' = v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

[Equals2]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1{=}n_2, sto\rangle \rightarrow_b (ff, sto')}$$

where $v_1' \neq v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

[Greater-than1]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1{>}n_2, sto\rangle \rightarrow_b (tt, sto')}$$

where $v_1' > v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

[Greater-than2]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1{>}n_2, sto\rangle \rightarrow_b (ff, sto')}$$

where $v_1' \not> v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

[Lesser-than1]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1{<}n_2, sto\rangle \rightarrow_b (tt, sto')}$$

where $v_1' < v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

[Lesser-than2]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1{<}n_2, sto\rangle \rightarrow_b (ff, sto')}$$

where $v_1' \not< v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

Table E.58: Transition rules for boolean expressions

[Greater-equal-1]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1 >= n_2, sto\rangle \rightarrow_b (tt, sto')}$$

where $v_1' > v_2'$ *or* $v_1' = v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

[Lesser-equal1]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1 <= n_2, sto\rangle \rightarrow_b (tt, sto')}$$

where $v_1' < v_2'$ *or* $v_1' = v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

[Not-equal1]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1 <> n_2, sto\rangle \rightarrow_b (ff, sto')}$$

where $v_1' = v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

[Not-equal2]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1, sto\rangle \rightarrow_n (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_2, sto''\rangle \rightarrow_n (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle n_1 <> n_2, sto\rangle \rightarrow_b (tt, sto')}$$

where $v_1' \neq v_2', v_1' = floatValue(v_1), v_2' = floatValue(v_2)$

Table E.59: Transition rules for boolean expressions

[Not1]
$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto\rangle \rightarrow_b (tt, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{not}\ b, sto\rangle \rightarrow_b (ff, sto')}$$

[Not2]
$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto\rangle \rightarrow_b (ff, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{not}\ b, sto\rangle \rightarrow_b (tt, sto')}$$

[Bool-equals1]
$$\frac{\begin{array}{c} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (tt, sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (tt, sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 = b_2, sto\rangle \rightarrow_b (tt, sto')}$$

[Bool-equals2]
$$\frac{\begin{array}{c} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (ff, sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (tt, sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 = b_2, sto\rangle \rightarrow_b (ff, sto')}$$

[Bool-equals3]
$$\frac{\begin{array}{c} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (tt, sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (ff, sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 = b_2, sto\rangle \rightarrow_b (ff, sto')}$$

[Bool-equals4]
$$\frac{\begin{array}{c} env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (ff, sto'') \\ env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (ff, sto') \end{array}}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 = b_2, sto\rangle \rightarrow_b (tt, sto')}$$

Table E.60: Transition rules for boolean expressions

[Bool-and1]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (tt, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (tt, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ and } b_2, sto\rangle \rightarrow_b (tt, sto')}$$

[Bool-and2]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (ff, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ and } b_2, sto\rangle \rightarrow_b (ff, sto')}$$

[Bool-and3]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (tt, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (ff, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ and } b_2, sto\rangle \rightarrow_b (ff, sto')}$$

[Bool-or2]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (tt, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ or } b_2, sto\rangle \rightarrow_b (tt, sto')}$$

Table E.61: Transition rules for boolean expressions

[Bool-or3]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (\mathit{ff}, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (\mathit{tt}, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ or } b_2, sto\rangle \rightarrow_b (\mathit{tt}, sto')}$$

[Bool-or4]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (\mathit{ff}, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (\mathit{ff}, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ or } b_2, sto\rangle \rightarrow_b (\mathit{ff}, sto')}$$

[Bool-xor1]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (\mathit{tt}, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (\mathit{tt}, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ xor } b_2, sto\rangle \rightarrow_b (\mathit{ff}, sto')}$$

[Bool-xor2]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (\mathit{tt}, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (\mathit{ff}, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ xor } b_2, sto\rangle \rightarrow_b (\mathit{tt}, sto')}$$

[Bool-xor3]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (\mathit{ff}, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (\mathit{tt}, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ xor } b_2, sto\rangle \rightarrow_b (\mathit{tt}, sto')}$$

[Bool-xor4]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1, sto\rangle \rightarrow_b (\mathit{ff}, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_2, sto''\rangle \rightarrow_b (\mathit{ff}, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b_1 \texttt{ xor } b_2, sto\rangle \rightarrow_b (\mathit{ff}, sto')}$$

[Bool-parent]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle b, sto\rangle \rightarrow_b (v, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle (b), sto\rangle \rightarrow_b (v, sto')}$$

[Bool-true]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{true}, sto\rangle \rightarrow_b (\mathit{tt}, sto)$$

[Bool-false]

$$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \texttt{false}, sto\rangle \rightarrow_b (\mathit{ff}, sto)$$

Table E.62: Transition rules for boolean expressions

[Convert-integer-1]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i, sto\rangle \rightarrow_a (v', sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle float\ op\ f, sto''\rangle \rightarrow_f (v, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i\ op\ f, sto\rangle \rightarrow_f (v, sto')}$$

where $float = LittType^{-1}(floatValue(v'))$

[Convert-integer-2]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i, sto\rangle \rightarrow_a (v', sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle float\ op\ f, sto''\rangle \rightarrow_f (v, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f\ op\ i, sto\rangle \rightarrow_f (v, sto')}$$

where $float = LittType^{-1}(floatValue(v'))$

[Convert-integer-3]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \rightarrow_a (v', sto^{(3)}) \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto^{(3)}\rangle \rightarrow_a (v'', sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle float_1\ /\ float_2, sto''\rangle \rightarrow_f (v, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1\ /\ i_2, sto\rangle \rightarrow_f (v, sto')}$$

where $float_1 = LittType^{-1}(floatValue(v'))$

where $float_2 = LittType^{-1}(floatValue(v''))$

Table E.63: Transition rules for aritmetic expressions

E.14 Arithmetic Expressions

Arithmetic expressions are expressions and therefore use the semantic form of the general expressions.

[Plus-integer]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \rightarrow_i (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \rightarrow_i (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1{+}i_2, sto\rangle \rightarrow_i (v, sto')}$$

where $v = v_1 + v_2$

[Minus-integer]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \rightarrow_i (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \rightarrow_i (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1{-}i_2, sto\rangle \rightarrow_i (v, sto')}$$

where $v = v_1 - v_2$

[Mult-integer]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \rightarrow_i (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \rightarrow_i (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1{*}i_2, sto\rangle \rightarrow_i (v, sto')}$$

where $v = v_1 \cdot v_2$

[Div-integer]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \rightarrow_i (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \rightarrow_i (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1 \texttt{ div } i_2, sto\rangle \rightarrow_i (v, sto')}$$

where $v = v_1 \div v_2$

[Mod-integer]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1, sto\rangle \rightarrow_i (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_2, sto''\rangle \rightarrow_i (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i_1 \texttt{ mod } i_2, sto\rangle \rightarrow_i (v, sto')}$$

where $v = v_1 \mod v_2$

[Parent-integer]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle i, sto\rangle \rightarrow_i (v, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle (i), sto\rangle \rightarrow_i (v, sto')}$$

Table E.64: Transition rules for integer expressions

[Plus-float]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_1, sto\rangle \rightarrow_f (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_2, sto''\rangle \rightarrow_f (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_1 + f_2, sto\rangle \rightarrow_f (v, sto')}$$

where $v = v_1 + v_2$

[Minus-float]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_1, sto\rangle \rightarrow_f (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_2, sto''\rangle \rightarrow_f (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_1 - f_2, sto\rangle \rightarrow_f (v, sto')}$$

where $v = v_1 - v_2$

[Mult-float]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_1, sto\rangle \rightarrow_f (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_2, sto''\rangle \rightarrow_f (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_1 * f_2, sto\rangle \rightarrow_f (v, sto')}$$

where $v = v_1 \cdot v_2$

[Division-float]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_1, sto\rangle \rightarrow_f (v_1, sto'') \quad env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_2, sto''\rangle \rightarrow_f (v_2, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f_1 / f_2, sto\rangle \rightarrow_f (v, sto')}$$

where $v = v_1 / v_2$

[Parent-float]

$$\frac{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle f, sto\rangle \rightarrow_f (v, sto')}{env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle (f), sto\rangle \rightarrow_f (v, sto')}$$

Table E.65: Transition rules for float expressions

[Litt-float]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle float, sto\rangle \rightarrow_f (v, sto)$

where $LittType(float) = v$

[Litt-integer]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle int, sto\rangle \rightarrow_i (v, sto)$

where $LittType(int) = v$

[Infty]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \textbf{infty}, sto\rangle \rightarrow_f (v, sto)$

where $v = infty$

[-Infty]
$env_C, env_F, env_V, env_{RV}, env_{RT}, env_P \vdash \langle \textbf{-infty}, sto\rangle \rightarrow_f (v, sto)$

where $v = -infty$

Table E.66: Transition rules for Litt-float, Litt-int, infty, and -infty