SIMULATION LANGUAGE

specification, verification and implementation

E1-209

Lau Bech Lauritzen *Jasper Kjersgaard Juhl*
Michael Gade Nielsen *Anders Rune Jensen*
Ole Laursen *Janne Larsen*

May 2003

AALBORG UNIVERSITY

Aalborg University
Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

Title:

Simulation language – specification, verification and implementation

Project period:

DAT2, Feb. 3th – May 30th, 2003

Project group:

E1-209

Members of the group:

Anders Rune Jensen
Jasper Kjersgaard Juhl
Janne Larsen
Lau Bech Lauritzen
Ole Laursen
Michael Gade Nielsen

Supervisor:

Igor Timko

Abstract:

A simulation language with a syntax similar to the C family is designed and its formal semantics defi ned. A translation to Java virtual machine assembler is then specifi ed, and proved to preserve the high-level semantics. Finally, an actual implementation of the compiler is presented.

Number of copies: 8

Report – number of pages: 89

Appendix – number of pages: 2

Total amount of pages: 91

Preface

This report documents the development of a simulation language and a compiler for the language with the Java virtual machine as the target architecture. The language and compiler have been developed as part of the DAT2 semester at Department of Computer Science, Aalborg University.

Source code for the compiler implementation is available at:

`http://www.cs.auc.dk/~jasper/dat2/`

Aalborg, May 2003,

Lau Bech Lauritzen	*Jasper Kjersgaard Juhl*
Michael Gade Nielsen	*Anders Rune Jensen*
Ole Laursen	*Janne Larsen*

Contents

1 Introduction **5**

1.1 Modeling behaviour 5
1.2 Specification of models 6
1.3 Basic model characteristics 6
1.4 A railroad company case 7
1.4.1 The three models 8
1.4.2 Using the models for simulation 11
1.4.3 Organising the models and the simulation setup 12
1.5 Further language support 14
1.5.1 Loops 14
1.5.2 Type conversion 15
1.5.3 Symbolic constants 16
1.6 Comparison with general-purpose languages 16

2 Syntax **18**

2.1 Introduction 18
2.2 Context-free specification 18
2.2.1 Abstract syntax 18
2.2.2 Concrete syntax 21
2.3 Contextual specification 21
2.3.1 Create statement requirement 21
2.3.2 Identifier binding 22
2.3.3 Type rules 23

3 Semantics **27**

3.1 Introduction 27
3.2 Definitions 27
3.2.1 Syntactic categories 27
3.2.2 Environment 27

3.2.3 Creation list . . . 29
3.2.4 Conversion of literals . . . 29
3.3 Semantic rules for the language . . . 30
3.3.1 Expressions . . . 30
3.3.2 Statements . . . 33
3.3.3 Toplevel constructs . . . 36
3.4 Built-in functions . . . 38

4 Translation to JVM code 40
4.1 The Java VM . . . 40
4.1.1 The basic model . . . 41
4.1.2 Types . . . 41
4.1.3 Variables and functions . . . 42
4.1.4 Stack manipulation instructions . . . 42
4.1.5 Arithmetic instructions . . . 42
4.1.6 Jump instructions . . . 43
4.1.7 Type conversion instructions . . . 44
4.2 Translation . . . 44
4.2.1 Expressions . . . 45
4.2.2 Statements . . . 47
4.3 Correctness of the translation . . . 48
4.3.1 One way . . . 49
4.3.2 The other way . . . 58
4.3.3 Correctness . . . 66

5 Implementation 67
5.1 Architecture . . . 67
5.2 Syntactic analysis . . . 69
5.2.1 `rdp` . . . 69
5.2.2 Transforming the grammar . . . 69
5.2.3 Representation of the abstract syntax . . . 71
5.2.4 Creating the abstract syntax tree . . . 73
5.3 Contextual analysis . . . 74
5.3.1 Binding identifiers . . . 74
5.3.2 Type checking . . . 76
5.3.3 Importing files . . . 77
5.3.4 Declarations . . . 78
5.4 Error reporting . . . 79

5.5 Code generation 79
5.5.1 Overall design of the generated code 80
5.5.2 Type definitions 80
5.5.3 Variable definitions 81
5.5.4 Variable lookup 82
5.5.5 Function definitions 82
5.5.6 Function call 83
5.5.7 Enumerations 84
5.5.8 Import 85
5.5.9 Create statements 85
5.5.10 Declarations 86
5.6 Connecting the phases 86
5.7 Conclusion 87

Bibliography **89**

A `rdp` grammar **90**

Chapter 1

Introduction

This chapter gives an introduction to the problem that our language is intended to help solve, and presents the various constructs that the language supports.

1.1 Modeling behaviour

Most large company decisions involve cost/benefit analysis based on estimates of various parameters [4], such as how willing the customers are to buy a certain product. But estimation of these parameters may require very complex considerations and considerable experience with the specific domain in order to be correct [5].

For example, a transportation service provider may wish to find out whether a reduction of the ticket prices would increase or decrease his income. Experimenting with changing the ticket prices directly may, however, be expensive and also difficult since it can harm the reputation of the service provider.

Instead, one can try to model the behaviour of the customers. The service provider might conjecture that families with low income travel by bus if it is cheaper than traveling by car, whereas families with high income will travel by bus if it is more convenient no matter what. The service provider can then find the number of low-income and high-income families and examine the prices for gasoline and the average travel durations to calculate an estimate.

The introduction of a model shifts the burden of determining parameters for the decision itself to determining the parameters of the model, which in some situations may be much easier and cheaper. It does also introduce the problem of ensuring that the model is valid in itself, however.

1.2 Specification of models

So how can one specify a model? The goal of this project is to help that.

One approach is to express the relations between the objects in terms of mathematical functions. For instance, the transportation service provider may assume that the fraction of low-income families which travel by bus depends linearly on the ticket price between the two endpoints, everyone traveling by bus and everyone traveling by car.

A mathematical description is in so far good enough, but the specification of a model is only part of the process. Another part is that of retrieving data from the model. If the amount of data to retrieve is large, this retrieval is best done automatically which requires the model to be specified with a strictly formal language. The model can then be run through a simulator that processes the description and outputs the needed data.

1.3 Basic model characteristics

Some fundamentally different options exist for constructing the model. One can choose a continuous time approach or model time with discrete events so that each state is calculated from the previous state. And when modeling many similar objects, one can either actually construct all these objects and simulate their individual behaviour, or only construct one and let it exhibit an average behaviour. The best choice is not obvious and may depend on the situation, but we have chosen to support discrete-event simulation of multiple objects.

One important advantage of formulating the models as discrete events is that it makes it possible to express dependencies on past events directly. For instance, with the transportation model if someone in a family has borrowed the car for a couple of days, the car is not available which affects the decision process. With a discrete model, we can easily keep track of such situations, whereas a continuous model would have to concern itself with how they affect the average case behaviour.

Also using as many objects as modeled instead of only average-case ones lets one focus on modeling a particular object of interest, splitting individual differences into different cases, instead of having to think about all at once to capture the average case behaviour. A more sophisticated analysis of the data generated by the model may also be concerned with more than just the averages in which case we must have data from individual objects.

1.4 A railroad company case

Based on the observations in the previous sections, this section will develop the fundamentals of a formal modeling language for simulation in parallel with presenting a model of some of the customers of a railroad company.

The model assumes that the persons can either go by train or by car and incorporates three kinds of persons who often travel by train: commuters, business men (e.g. marketing or sales persons) and students visiting their family. Commuters need to travel each workday and we will characterize them as wanting low price and short travel times since traveling takes up much of their lives. Business persons do not care much about the price since the company pays the bill and presumably they do not mind a long journey provided they are able to work meanwhile. They will travel only on workdays but not every day. The students usually only travel during the week-ends, i.e. Friday to Monday (we will not consider holidays or vacations). They are concerned about price, but not as much about the duration of the journey.

For specification of the types, we will borrow the class syntax from C++/Java:

```
type Commuter {
  // variable definitions
  // function definitions

  void iterate(int iteration)
  {
    // process object
  }
}
```

In this and the following code snippets, reserved words are in bold face. The iterate method is called once for each iteration and is responsible for updating the state of the object. We can then extend the usual C variable definition syntax with an extra qualifier **watched** which causes the member variable to be output after each iteration:

```
type Commuter {
  watched bool train;  // go by train?
  // ...
}
```

This simple structure realizes the discrete event model and at the same time allows us to easily define what output we want from the simulation.

1.4.1 The three models

So sticking to the notation of the C family of languages, we can define a model of a commuter:

```
1  type Commuter {
2    float income = 50000 ... 500000; // randomly selected
3    watched bool train;
4
5    void iterate(int iteration)
6    {
7      int weekday = iteration % 7;
8      if (weekday < 5) {
9        float prob;
10
11       float price = price_car / (price_train + price_car);
12       float time = time_car / (time_train + time_car);
13
14       if (income < 150000)
15         prob = 0.7 * price + 0.3 * time;
16       else if (income < 300000)
17         prob = 0.3 * price + 0.7 * time;
18       else
19         prob = time;
20
21       float random = 0.0 ... 1.0;
22       train = random < prob;
23     }
24     else
25       train = false;
26   }
27 }
```

On line 2, the model defines an income in the range min_income to max_income; the binary operator ... returns a random number linearly distributed in the range denoted by the two arguments. Then it defines an output boolean variable for indicating whether the person went by train.

The iterate method checks whether it is a workday (line 8); if so, the probability of the person choosing the train is calculated by segmenting the commuters into three groups depending on income (line 14, 16 and 18) and setting weights for the dependency on price and journey duration. Finally a random value is generated and the outcome is chosen from that (line 22).

The calculations depend on some variables that are not defined above, the cost of traveling by train or car and the respective journey durations. These depend on the particular person – we will later discuss how to incorporate them.

A simple model of a business man is:

```
type BusinessMan {
  watched bool train;

  void iterate(int iteration)
  {
    int weekday = iteration % 7;

    if (weekday < 5 && 0.0 ... 1.0 < travel_prob) {
      if (time_car < 0.5)    // less than half an hour
        train = false;
      else if (time_car > 3) // more than three hours
        train = true;
      else {
        float base_prob = (time_car - 0.5) / (3 - 0.5);
        float fraction = time_train / time_car;
        float final_prob = base_prob ^ fraction;

        train = 0.0 ... 1.0 < final_prob;
      }
    }
    else  // do not travel today
      train = false;
  }
}
```

The model assumes that travel_prob has been defined. On line 14, the base probability is linearly interpolated between half an hour (always go by car) and three hours (always go by train). On line 16, the final probability is then skewed by raising it to the power of the relative journey durations between going by train and going by car (^ is the power operator); Figure 1.1 illustrates the effect.

A model of a student is:

```
type Student {
  int max_weeks;
  int departure_day, return_day;
  watched bool train;
  int last_visit = 0;

  void Student()
```

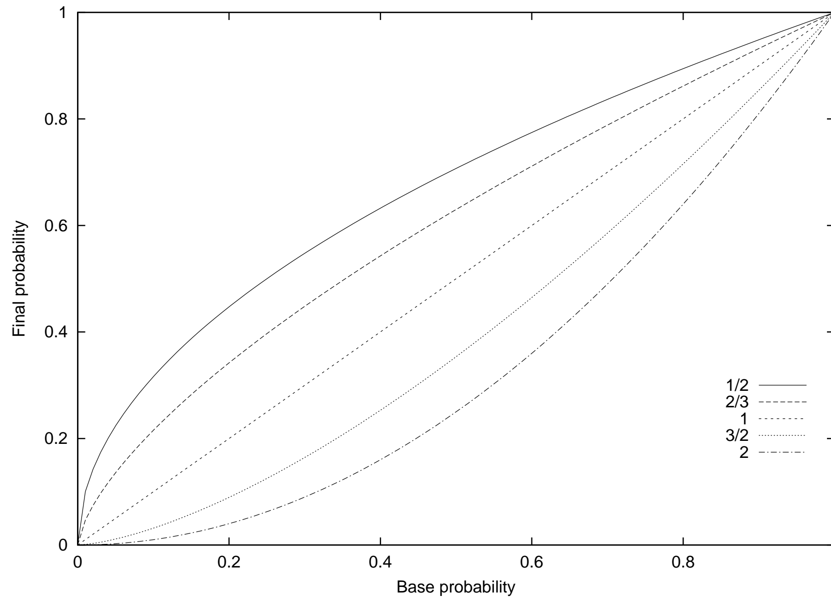

Figure 1.1: Illustration of $p_{final} = p_{base}^{t}$ for some values of t where $t = t_{train}/t_{car}$. When $k > 1$, the probability is skewed towards zero; vice versa for $k < 1$. When $k = 1$, the final probability is simply the base probability.

```
8     {
9       // expensive travel implies staying away longer
10      max_weeks = price_train / 100 + (1 ... 12);
11      // set off Friday, Saturday or Sunday
12      departure_day = 0.6 : 4 | 0.3 : 5 | 0.1 : 6;
13      // return Saturday, Sunday or Monday
14      return_day = ((departure_day + 1) ... 7) % 7;
15    }
16
17    void iterate(int iteration)
18    {
19      int weekday = iteration % 7;
20      float prob = price_car / (price_train + price_car);
21
22      train = false;
23
24      if (weekday == departure_day) {
25        float return_prob = last_visit / max_weeks;
26        if (0.0 ... 1.0 < return_prob) {
27          train = 0.0 ... 1.0 < prob;
28          last_visit = 0;
```

```
29         }
30         else
31           ++last_visit;
32       }
33 
34       if (weekday == return_day && last_visit == 0)
35         train = 0.0 ... 1.0 < prob;
36     }
37 }
```

For the student we introduce a constructor (line 7) which initializes each object with a maximum number of weeks (line 10) before returning home (based on the ticket prices, adding 100 to the ticket price will enlarge the possible maximum by one week) and chooses the week days for departure and return (line 12 and 14).

The decision about whether to depart a specific week is then determined by max_weeks and the amount of time elapsed since last visit (line 25). Whether to go by train is probabilistically determined by the relative prices of traveling by car and by train (line 20).

The constructor uses the special notation

$$e_1 : e_a \quad | \quad e_2 : e_b \quad | \quad \ldots$$

which makes a non-deterministic choice between the expressions $e_a, e_b, \ldots$ using $e_1, e_2, \ldots$ as weights. So the value of the whole expression is either e_a with probability $e_1/(e_1 + e_2 + \ldots)$ or e_b with probability $e_2/(e_1 + e_2 + \ldots)$, etc.

1.4.2 Using the models for simulation

The goal of specifying the above models is to simulate their behaviour for a given period of time, e.g. a month. For this we need instances of each of the models. Thus the basic setup consists of instructing the simulator to realize a number of objects of the different types:

```
create 500000 of Commuter;
create 100000 of BusinessMan;
create 100000 of Student;
```

To actually get a working simulation, we need to specify the times and prices for traveling by train and car in the models, though.

In general, some of the parameters that we wish to control are specific for each instantiated object of a model and as such parameterise that model specification, and some are the same for all objects. For instance, if a model were to take the weather into account, the controlling parameter would be the same for all objects.

We can model this kind of parameters with global variables defined outside the types, e.g.:

```
float weather_factor = 17.3;  // global variable

type WeatherDependent {
  void iterate(int iteration)
  {
    if (weather_factor > 10.0)
      // ...
  }
}
```

The missing variables for the three models we have developed above all fall into the category of parameterising the models, however. Since we have introduced a constructor for **type**s, we can conveniently pass the parameters through that. The constructor for BusinessMan would then be:

```
type BusinessMan {
  float time_train, time_car, travel_probability;
  // ...
  void BusinessMan(float train, float car, float travel)
  {
    time_train = train;
    time_car = car;
    travel_probability = travel;
  }
}
```

Hence, we need to change the create statements:

```
create 20000 of
   BusinessMan(1.0 ... 1.3, 0.8 ... 1.1, 0.1 ... 0.6);
create 20000 of
   BusinessMan(2.0 ... 2.5, 1.8 ... 2.3, 0.1 ... 0.6);
// ...
```

These create statements could easily be generated from collected real-world data.

1.4.3 Organising the models and the simulation setup

The simplest approach for physically organising the code for the models and the simulation setup is to put everything into one file. But this does not scale well:

- The single file may become very large which makes it difficult to manage and navigate.

- Reusing models for different scenarios is only possible by copying and pasting the code.

Instead we can break the file up into smaller files by adding a directive for importing definitions from other files:

```
import "other.model";
```

This means that all definitions of types, functions and variables in "other.model" are parsed and imported into the symbol tables of the processing of the current file. Name clashes are considered errors.

With the import directive, a reasonable way of organising the simulation would be to put the model code in separate ".model" files which are imported in a main ".scenario" file that contains the necessary create statements. The setup is illustrated in Figure 1.2. This way it is also easier to generate the ".scenario" files from an external source with real-world data.

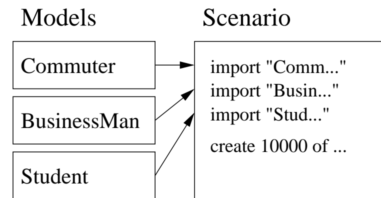

Figure 1.2: The scenario is constructed by importing the model files. Each model file contains a type definition.

As a bonus, the separation into files makes it possible to define small function libraries which can be imported by several files; something that can be very useful for a large modeling project. This requires that all files are only parsed once, though, to support the situation in Figure 1.3.

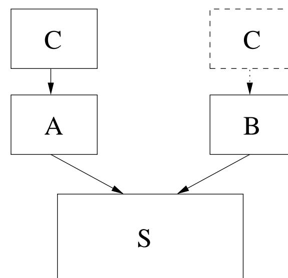

Figure 1.3: S imports A and B which in turn both import C that contains generally useful functions. To support this scenario C must only be parsed once – else there will be name clashes.

Another problem is that global variables in a model need to be known prior to their use. For example, type WeatherDependent cannot use the global variable weather_factor if weather_factor has not been defined before type WeatherDependent. We can remedy this by being careful with ordering the definitions and import directives:

```
float weather_factor = 15.0;
import "weather-dependent.model";
```

However, this makes the modularisation much more fragile. Instead we will introduce declarations which are simply definitions with the body removed and **extern** in front:

```
extern float weather_factor;
extern float max(float x, float y);
```

Putting these in the model files corrects the problem, and also makes it explicit that the identifiers are global. Note that the purpose of a declaration is to make a name (with its type) known to the simulator, whereas a definition also has the actual data associated with it. Thus there is no limit on the number of declarations as long as they all agree on the type of the name they are declaring, but there must be exactly one definition. If a model that refers to a declared, but undefined identifier is created, it is an error.

1.5 Further language support

More language support than presented in the previous section is necessary for a complete modeling language.

1.5.1 Loops

A loop statement may be needed, so we introduce a **while** loop:

```
int i = 0;
while (i < 100)
  ++i;
```

For block structuring we copy the syntax from C++ and Java with {} blocks and local variable definitions allowed everywhere single statements are allowed.

```
int x;
{
  int y;
  {
```

```
    int z;
  }
}
```

1.5.2 Type conversion

Since the modeling language supports two numeral types, floating point numbers and integers, we need a mechanism for converting between them. There are three possibilities:

1. Implicitly and silently convert whenever needed
2. Implicitly convert, but issue a warning
3. Do not convert, issue an error

The first possibility is unfortunately unsafe – consider the following example:

```
int proba = 0.0 ... 0.8;
float probb = 0 ... 1;
```

Here proba has accidentally been defined as an **int** so proba will always be 0, whereas probb is set to a random integer in the interval $[0;1]$, i.e. either 0 or 1. Clearly, both were intended to be continuously distributed.

In most programming languages implicit conversions from **int** to **float** are safe, but this is not so for our language since the ... operator needs to be able to return **int**s as well as **float**s. If it always returned **float**, the expression 1 ... 3 converted to **int** would not yield a uniform distribution (with truncation 1 and 2 would be equally probable, but 3 would almost never occur; with normal rounding 2 would be approximately twice as probable as 1 and 3).

The second and third possibilities are in fact very similar since they both require explicit type conversion. Even when we allow the conversion but issue a warning, we need a mechanism for turning off the warning for that particular statement. Else it is too disturbing, and the programmer may get into the habit of ignoring the warnings.

If we issue an error, we can be sure that the programmer will notice the potential problem. But the problem is only potentially a real problem. Setting up a large simulation just to return later discovering that it has not even started because the simulator halted over something that is in fact correct may be very annoying. Therefore we will just issue a warning.

The syntax for type conversion can be borrowed from function calls, thus:

```
int area = 150;
float radius = (floatify(area) / 3.14) ^ 0.5;
int circumference = intify(2.0 * 3.14 * radius);
```

Syntax similar to this is also allowed in C++. It has the advantage of being very easy to implement.

1.5.3 Symbolic constants

Since the models presumably will be mostly concerned about making decisions, it would be convenient with extended support for multiple-choice decisions. The $e_1 : e_a \mid e_2 : e_b \mid \ldots$ expression is part of this. But to avoid placing magic constants in the code, one has to define constants for the $e_a, e_b, \ldots$, e.g.:

```
int const Bus = 0, Car = 1, Plane = 2, ...;
```

Obviously, this is tedious. It also makes it impossible to type check the type that these constants implicitly define; consider:

```
int const January = 0, February = 1, March = 2, ...;
int day = February + Plane;  // meaning?
```

Instead we can borrow the enumeration syntax from C and C++:

```
enum Transportation {
  Bus, Car, Plane, ...
}
```

Then:

```
bool flying(Transportation t)
{
  return t == Plane;
}
```

Whereas February + Plane can be detected as a type error. Note that the semantics of our **enum** construct is different from that of C and C++ where the **enum** simply defines integer constants and conversions back and forth are allowed.

1.6 Comparison with general-purpose languages

Syntactically, our simulation language is deliberately very close to the C family of languages which is supposed to make it possible to write simulations almost without any training provided some familiarity with C, C++ or Java.

Although the similarity to these languages forces the semantics of the basic constructs (binary operators, the various kinds of statements and so forth) to be

similar too, there are still quite a lot of important differences between our language and general-purpose languages in general:

- Simplicity. In our language, there is no support for data structures (including OOP), dynamic memory management, interfacing with the rest of the system (e.g. I/O) or low-level bit manipulations. There is not even the concept of a string since it is not needed.

  This makes the language and language processors much simpler and thus easier to learn and construct. It also makes it possible to define much cleaner syntax and semantics in some cases, for instance with **enum**s where the distinction between **enum** variables and **int**s can be unclear in C and C++.

- Models are expressed directly. Since the sole purpose of the language is simulation, there is no need for auxiliary machinery for expressing and setting up the models and the simulation.

- Direct language support for some of the anticipated most common constructs, e.g. the multiple-choice operator. A simpler syntax makes the intended meaning much clearer and easier to write.

Hence, our language is a very simple programming language that supports functions and the basic mathematical tools together with randomised decision-support constructs well, i.e. the things that are needed for specifying models, but not much else.

Chapter 2

Syntax

This chapter defines how a correct program in the language looks like.

2.1 Introduction

For building a scanner to recognise the tokens that a simulation is composed of, a specification with regular expressions is enough, but for recognising complete language constructs, a more powerful specification is needed [10]. With a context-free BNF grammar [11] with production rules on the form $A \rightarrow B_1 B_2 \ldots B_n$ where A is a non-terminal and $B_1, B_2, \ldots, B_n$ may be terminals or non-terminals, most of the syntax rules can be specified easily and precisely. Some rules, e.g. type rules, requires contextual information, though, and are defined separately.

2.2 Context-free specification

We use an extended Backus-Naur form (EBNF) [9] which is BNF augmented with $*$ to denote zero or any number of times and $[\,]$ which denotes zero or once.

2.2.1 Abstract syntax

Starting from the top level, a valid file for the simulator contains a *simulation*:

```
simulation ::= (import-directive
                | variable-declaration
                | function-declaration
                | variable-definition
                | enum-definition
                | function-definition
                | type-definition
                | create-statement)*
```

Hence, at the top level it is possible to have import directives, declarations and definitions, and create statements. Imported files are also parsed starting from a *simulation*.

Further specification yields for the import directive:

```
import-directive ::= import "filename";
```

For the declarations:

```
variable-declaration ::= extern [const] typename identifier (, identifier)*;
function-declaration ::= extern function-header ;
function-header      ::= (typename | void)
                         identifier ( [typename identifier
                                   (, typename identifier)*] )
typename             ::= int | float | bool | identifier
```

Note that we allow declaring multiple variables in one variable declaration. A *typename* can be either of the basic types, or it can be the name of an `enum`.

The definitions:

```
variable-definition ::= [const] typename identifier [= expression]
                              (, identifier [= expression])*;
function-definition ::= function-header { (statement)* }
enum-definition     ::= enum identifier { identifier (, identifier)* }
type-definition     ::= type identifier {
                              ([watched] variable-definition
                               | function-definition
                               | enum-definition)*
                        }
```

Inside type definitions, only variable, enum and function definitions are allowed. There must always be at least one function definition, i.e. the `iterate(int)` function, but specifying that with the EBNF grammar is cumbersome so we will

instead catch the error by a later pass. The constructor function (with the same name as the type) is also special. By letting its return type be `void` we can easily parse it as a normal function. Defining a constructor to be a non-`void` type is an error.

The statement for instantiating objects:

create-statement ::= `create` integer-value `of` function-call **;**

By reusing the function call notation we ban create statements without paranthesis, so even if the type does not specify a constructor it is still necessary to supply an empty pair ().

The definitions of the statements:

```
statement      ::=  { (statement)* }
                    | variable-definition
                    | while ( expression ) statement
                    | if ( expression ) statement [ else statement ]
                    | identifier = expression ;
                    | function-call ;
                    | ++ identifier ; | -- identifier ;
                    | return expression ;
function-call  ::=  identifier ( [ expression (, expression)* ] )
```

Note that the increment and decrement operators (++ and --) are statements – they possess no value. This simplifies the grammar, and also renders postfix increment and decrement operators superfluous.

The expressions:

```
expression  ::=  identifier
                 | value
                 | function-call
                 | unary-op expression
                 | expression binary-op expression
                 | ( expression )
                 | expression : expression (| expression : expression)*
value       ::=  integer-value | float-value | boolean-value
unary-op    ::=  ! | -
binary-op   ::=  && | || | == | != | < | > | <= | >=
                 | + | - | * | / | % | ^ | ...
```

Except the multiple-choice operators and the exponentiation operator ^, all operators have the same meaning as in C, C++ and Java. % is the modulo operator.

2.2.2 Concrete syntax

The abstract syntax presented in the previous section is not precise enough to construct a parser for the language, hence we shall modify it to produce a concrete grammar.

The abstract grammar for expressions is ambiguous since an expression such as $1+2+3$ can be parsed as $((1+2)+3)$ or $(1+(2+3))$. Also, an abstract syntax tree produced from the expression grammar does not necessarily follow the usual operator precedence rules. To remedy this problem we substitute the production rules for *expression* with rules that realize ten expression levels:

expression ::= exp_1
exp_1 ::= exp_2 [(&& | ||) exp_1]
exp_2 ::= exp_3 [(== | !=) exp_2]
exp_3 ::= exp_4 [(< | > | <= | >=) exp_3]
exp_4 ::= exp_5 [: exp_5 (| exp_5 : exp_5)*]
exp_5 ::= exp_6 [(+ | -) exp_5]
exp_6 ::= exp_7 [(* | / | %) exp_6]
exp_7 ::= exp_8 [^ exp_7]
exp_8 ::= exp_9 [. . . exp_8]
exp_9 ::= exp_{10} | ! exp_{10} | - exp_{10}
exp_{10} ::= (exp_1) | identifier | value | function-call

The levels signify the precedence of the operators; the operators on the lowest levels have the highest precedence.

The remaining leaf-level non-terminals are defined as (where digit is any number in `0`–`9` and letter is an alphabetic character, {`a`, . . . , `z`, `A`, . . . , `Z`}):

identifier ::= letter (letter | digit)*
integer-value ::= digit (digit)*
float-value ::= digit (digit)*. digit (digit)*
boolean-value ::= `true` | `false`

2.3 Contextual specification

Some further rules that depend on contextual information are specified in this section.

2.3.1 Create statement requirement

In order to be able to start the simulation, there must be at least one create statement in the program. A consequence of this is that there must also be at least one type

definition.

We could enforce this by means of the EBNF grammar, but that would complicate it unnecessarily.

2.3.2 Identifier binding

The role of the variable, function, enum and type definitions is both to make identifiers known (with their types) and to bind them to some sort of contents, whereas the sole purpose of the variable and function declarations is to make the identifiers known.

We define that multiple declarations of the same identifier are allowed provided they all give the identifer the same type, but require that an identifier must have been defined or declared before the first applied occurrence of it. Furthermore, all declared identifiers must be defined exactly once.

Hence, the following program is a syntactically correct program:

```
extern int x;
extern int x;

void f()
{
  x = 15;
}

int x;
```

Whereas the following listing is not a valid program:

```
int x = f();     // error: f is not declared

int f()
{
  return 15;
}
```

Neither is:

```
extern int f();
int x = f();
// error: f is never defined
```

Furthermore, local variable definitions inside blocks are allowed to have the same name as variables in outer scopes and simply shadow these. The same is not true for function and enum definitions, though.

2.3.3 Type rules

There are three different basic types in the language, **int**, **float** and **bool**, plus an unlimited number of user-defined **enum** types. A syntactically correct program must not have any major type mismatches, but may have any number of minor type mismatches.

Major type mismatches

We define a major mismatch as one of the following three cases:

1. An expected type is **bool**, but the actual type is **int** or **float** or an **enum** type.
2. An expected type is **int** or **float**, but the actual type is **bool** or an **enum** type.
3. An expected type is an **enum** type, but the actual type is **int**, **float** or **bool** or a different **enum** type.

Minor type mismatches

A minor mismatch is one of the following two cases:

1. An expected type is **int**, but the actual type is **float**.
2. An expected type is **float**, but the actual type is **int**.

A minor mismatch causes a warning and an implicit conversion from the actual type to the expected type by means of intify and fbatify .

Expected types

The definition of where expected types occur and what they are is based on the abstract grammar.

A variable definition may contain an initializer expression. The expected type of this expression is the type of the variable being defined.

```
int x = expr;   // expected type of expr is int
```

A **while** statement contains an expression which is tested before each iteration. The expected type of this expression is **bool**.

```
while (expr)  // expected type of expr is bool
   statement;
```

An **if** statement contains a condition expression. The expected type of this expression is **bool**.

```
if (expr)       // expected type of expr is bool
  statement;
else
  statement;
```

An assignment has an expression on the right-hand side. The expected type of this expression is the type of the variable on the left-hand side. Furthermore, the type of the variable may not be **const**.

```
int x;
x = expr;       // expected type of expr is int
```

A function call may have any number of actual parameters which are expressions. The expected type of any such parameter is the expected type of the corresponding formal parameter. Since function overloading is not supported, this is unambiguous.

```
int f(int y)
{
  return y;
}
int x = f(expr); // expected type of expr is int
```

For an increment or decrement statement, the expected type of the identifier is **float** or **int** depending on the actual type and may not be **const**.

```
int identifier = 0;
++identifier; // expected type of identifier is int
--identifier;
```

A **return** statement contains an expression. The expected type of this expression is the return type of the function.

```
int f()
{
  return expr; // expected type of expr is int
}
```

The expected type of the operand to the unary operator ! is **bool**. The expected type of the operand to the unary sign operator – is **float** or **int** depending on the actual type.

The expected type of the operands to the binary operators && and || is **bool**.

The types of the two operands to the operators == and != should be the same. To avoid ambiguity, we specify the following set of rules to determine the expected type where the last rules are considered only in case the first rules do not apply:

1. If the left-hand side of the operator is of type **enum** η, then the expected type of the other operand is also **enum** η.

2. If the left-hand side of the operator is of type **bool**, then the expected type of the other operand is **bool**.

3. If either operand is of type **float**, the expected type of both operands is **float**.

4. Else the left-hand side must be of type **int** and the expected type of the other operand is **int**.

For the rest of the binary operators, $<$, $>$, $<=$, $>=$, $+$, $-$, $*$, $/$, %, ^ and ... , the expected type of the operands is either **int** or **float**. If one of the operands has the type **float**, the expected type of both operands is **float**. Else the expected type of both operands is **int**.

The expected type of the expressions on the left side of the colons in a the multiple-choice operator is **float**. The types of all the right-hand side expressions of the colon pairs should be the same. Thus the type is determined by the following ruleset where the rules again are considered one by one:

1. If the type of the first expression is **enum** η, then the expected type of all expressions is also **enum** η.

2. If the type of the first expression is **bool**, then the expected type of all expressions is **bool**.

3. If any of the expressions is of type **float**, the expected type of all expressions is **float**.

4. Else the first expression must be of type **int** and the expected type of all expressions is **int**.

Type deduction

The definition of the expected types of the language constructs assumes that the actual types of the immediate constituents are known. These can be deduced quite easily by a recursive definition:

- The type of a variable is explicitly and uniquely defined in the program.

- The type of a literal value is deduced from the syntax and is defined with EBNF notation in Section 2.2.2.

- The type of a function call is the return type of the function.

- The type of a unary expression is the type of the operand.
- The type of a binary expression is the type of the two operands (due to the definition in the previous section, the two types are the same; a minor type mismatch would have caused a type conversion).
- The type of a parenthesized expression is the type of the expression.
- The type of a multiple-choice expression is the type of right-hand sides of the colon pairs.

Chapter 3

Semantics

This chapter presents a formal definition of the semantics of the language.

3.1 Introduction

The semantics that we will specify are big-step/natural [2; 8] semantics which work by specifying rules for each of the syntactic constructs. Each rule has a number of premises that must be fulfilled for the rule to be concluded, and the derivation of the conclusion for the rule that covers the whole program is then a specification of the semantics for the program.

3.2 Definitions

3.2.1 Syntactic categories

For the various constructs that are allowed in the language, we define a number of syntactic categories with meta variables that will be used to refer to elements of the categories. The categories are listed in Table 3.1.

3.2.2 Environment

An environment-store model is used to model variables and the mapping from symbolic representations to actual values. A variable environment e_V is a mapping from identifiers to store locations. A store *sto* is a mapping from locations to actual values. A value is a member of the set $\mathbf{Val} = \mathbb{Z} \cup \mathbb{R} \cup \{\texttt{T}, \texttt{F}\} \cup \mathbb{E}$ where $\mathbb{E}$ is the set of all `enum` values.

Since only some identifiers have a defined mapping (i.e. only those that have been defined in the program), and likewise since only some store locations map to

Category	Set	Meta
Toplevel constructs	**Topl**	T
Type members	**Memb**	M
Statements	**Stat**	S
Expressions	**Expr**	E
Variables	**Var**	V
Functions	**Fun**	F
Literal values	**Lit**	C
Typenames	**Typ**	τ

Table 3.1: The syntactic categories for the language.

well-defined values, both mappings are partial. For functions with return value we also need a special location, "retval", for saving the value. Thus the set of variable environments is:

$$\mathbf{Env}_V = \mathbf{Var} \hookrightarrow \mathbf{Loc}$$

And the set of stores:

$$\mathbf{Sto} = \mathbf{Loc} \cup \{\text{retval}\} \hookrightarrow \mathbf{Val}$$

These are updated by choosing new mappings that filters the identifiers. For instance, the environment e'_V which is e_V except that the identifier `dog` is bound to the location l is denoted by $e_V[\texttt{dog} \mapsto l]$. In general, we have

$$e_V[V' \mapsto l](V) = \begin{cases} l & \text{if } V = V' \\ e_V(V) & \text{if } V \neq V' \end{cases}$$

The same notation will be used to decorate stores and function environments. To facilitate variable definitions, the special symbol "next" will always map to the next unused store location.

Function environments are used to keep track of defined functions. The needed information is the function body (a block statement), the formal parameters and the variable and function environments at the time of the definition since the language is statically scoped. Hence the set of function environments is given by

$$\mathbf{Env}_F = \mathbf{Fun} \hookrightarrow \mathbf{Stat}_{\text{block}} \times \mathbf{Parm} \times \mathbf{Env}_F \times \mathbf{Env}_V$$

where $\mathbf{Stat}_{\text{block}}$ is the set of all block statements and **Parm** is the set of all tuples of length ≥ 0 with elements from **Var** as elements.

To support function and variable declarations that allow usage of variables and functions before their definition, two levels of environments are maintained, the global and the local environment. Variable and function lookup is done by first looking in the local environment, and in case it does not contain a binding then in the global environment. For variables, we define L_V to aid this lookup:

$$L_V(V, e_V, e_{V_G}) = \begin{cases} e_V(V) & \text{if } e_V \text{ is defined for } V \\ e_{V_G}(V) & \text{else} \end{cases}$$

$L_F(F, e_F, e_{F_G})$ is defined similarly for functions.

Hence a complete environment $e \in \mathbf{Env}$ is a tuple consisting of the local and global function environment, the local and global variable environment and the store:

$$\mathbf{Env} = \mathbf{Env}_F \times \mathbf{Env}_F \times \mathbf{Env}_V \times \mathbf{Env}_V \times \mathbf{Sto}$$

To make the semantic rules readable, we shall consider e as a short-hand notation for $(e_F, e_{F_G}, e_V, e_{V_G}, sto)$, and for instance usually just write e' instead of $(e_F, e_{F_G}, e_V, e_{V_G}, sto')$.

To model return statements, we further introduce $\mathcal{R}$-tagging of an environment, written $e^{\mathcal{R}}$, to mark an environment returned by a `return` statement transition. Any statement evaluated in an $\mathcal{R}$-tagged environment will then be skipped up to the function calling point where the tag is removed from the environment and everything proceeds as usual.

3.2.3 Creation list

Create statements are collected and eventually used to instantiate objects for a simulation. To model the collection process, we define a creation list to be an ordered sequence of tuples belonging to the set $\mathbf{CL} = \mathbb{N} \times \mathbf{Fun} \times (\mathbf{Expr} \times \ldots \times \mathbf{Expr})$ (number of objects and constructor with actual parameters). The notation $cl :: (x, F, (E_1, \ldots, E_n))$ will be used to append a create directive to the list cl.

3.2.4 Conversion of literals

An auxiliary function $\mathcal{V} : \mathbf{Lit} \rightarrow \mathbf{Val}$ is used to denote conversion of syntactic literals to values, thus $\mathcal{V}(\texttt{12}) = 12$, $\mathcal{V}(\texttt{12.3}) = 12.3$ and $\mathcal{V}(\texttt{true}) = \mathrm{T}$.

3.3 Semantic rules for the language

3.3.1 Expressions

The transition system for expressions is (**Expr** $\times$ **Env** $\cup$ **Val** $\times$ **Env**, $\rightarrow$, **Val** $\times$ **Env**) where $\rightarrow$ is defined in the following subsections.

Literals and identifiers

[lit] $\langle C, e\rangle \rightarrow \langle v, e\rangle$ where $v = \mathcal{V}(C)$

[var] $\langle V, e\rangle \rightarrow \langle v, e\rangle$ where $v = sto(L_V(V, e_V, e_{V_G}))$

Arithmetic operators

The transition rules for the arithmetic operators implicitly define rules for both floating point numbers and integers, except division where the integer rule introduces rounding. $\lfloor x \rfloor$ is assumed to round towards zero.

[plus] $$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\texttt{+}E_2, e\rangle \rightarrow \langle v, e''\rangle}$$ where $v = v_1 + v_2$

[minus] $$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\texttt{-}E_2, e\rangle \rightarrow \langle v, e''\rangle}$$ where $v = v_1 - v_2$

[mult] $$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\texttt{*}E_2, e\rangle \rightarrow \langle v, e''\rangle}$$ where $v = v_1 v_2$

[float-div] $$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\texttt{/}E_2, e\rangle \rightarrow \langle v, e''\rangle}$$ where $v = v_1 / v_2$

[int-div] $$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\texttt{/}E_2, e\rangle \rightarrow \langle v, e''\rangle}$$ where $v = \lfloor v_1 / v_2 \rfloor$

[mod] $$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\texttt{\%}E_2, e\rangle \rightarrow \langle v, e''\rangle}$$ where $v = v_1 - \left\lfloor \frac{v_1}{v_2} \right\rfloor v_2$

[power] $$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\texttt{\^{}}E_2, e\rangle \rightarrow \langle v, e''\rangle}$$ where $v = v_1^{v_2}$

[un-minus] $$\frac{\langle E, e\rangle \rightarrow \langle v', e'\rangle}{\langle -E, e\rangle \rightarrow \langle v, e'\rangle}$$ where $v = -v'$

[paren] $$\frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle}{\langle \texttt{(}E\texttt{)}, e\rangle \rightarrow \langle v, e'\rangle}$$

Note that the rules evaluate the leftmost operand first, which may change the environment (by changing the store) for the rightmost operand.

Probabilistic operators

The outcomes of the probabilistic rules are drawn from linearly distributed sources in the given intervals.

$$\text{[float-rand]} \quad \frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1 \ldots E_2, e\rangle \rightarrow \langle v, e''\rangle} \quad \text{where } v \in [v_1; v_2[$$

$$\text{[int-rand]} \quad \frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1 \ldots E_2, e\rangle \rightarrow \langle v, e''\rangle} \quad \text{where } v \in \{v_1, \ldots, v_2\}$$

The multiple-choice operator evaluates the leftmost operand (the weight) of each colon pair in turn, chooses a pair i with probability p_i and evaluates and returns the value of the rightmost operand of that pair.

$$\text{[mult-choice]} \quad \frac{\langle E_{w_j}, e_{j-1}\rangle \rightarrow \langle w_j, e_j\rangle \text{ for } j = 1, \ldots, n \quad \langle E_{v_i}, e_n\rangle \rightarrow \langle v_i, e_{n+1}\rangle}{\langle E_{w_1} : E_{v_1} \mid \ldots \mid E_{w_n} : E_{v_n}, e_0\rangle \rightarrow \langle v_i, e_{n+1}\rangle} \quad \text{where } p_i = w_i / \sum_{j=1}^{n} w_j$$

Relational operators

$$\text{[equals]} \quad \frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1 \texttt{==} E_2, e\rangle \rightarrow \langle v, e''\rangle} \quad \text{where } v = \begin{cases} \text{T} & \text{if } v_1 = v_2 \\ \text{F} & \text{else} \end{cases}$$

We define the rules [not-equals], [less-than], [greater-than], [less-than-equals] and [greater-than-equals] in the same manner, replacing `==` and $=$ with `!=` and $\neq$, `<` and $<$, etc.

Boolean operators

Since the syntax of our language is close to the C family of languages, we define the semantics of the boolean operators to be short-circuiting to avoid confusion. The [and-sc] and the [or-sc] rules short-circuit the cases where the first operand is F and T, respectively, so that the outcome of the [and] and [or] rules is determined by the second operand (e.g. $\mathrm{T} \wedge \mathrm{F} = \mathrm{F}$ and $\mathrm{T} \wedge \mathrm{T} = \mathrm{T}$).

[and-sc] $$\frac{\langle E_1, e\rangle \to \langle v_1, e'\rangle}{\langle E_1 \mathtt{\&\&} E_2, e\rangle \to \langle \mathrm{F}, e'\rangle} \quad \text{if } v_1 = \mathrm{F}$$

[and] $$\frac{\langle E_1, e\rangle \to \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \to \langle v_2, e''\rangle}{\langle E_1 \mathtt{\&\&} E_2, e\rangle \to \langle v_2, e''\rangle} \quad \text{if } v_1 = \mathrm{T}$$

[or-sc] $$\frac{\langle E_1, e\rangle \to \langle v_1, e'\rangle}{\langle E_1 \mathtt{||} E_2, e\rangle \to \langle \mathrm{T}, e'\rangle} \quad \text{if } v_1 = \mathrm{T}$$

[or] $$\frac{\langle E_1, e\rangle \to \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \to \langle v_2, e''\rangle}{\langle E_1 \mathtt{||} E_2, e\rangle \to \langle v_2, e''\rangle} \quad \text{if } v_1 = \mathrm{F}$$

[not] $$\frac{\langle E, e\rangle \to \langle v', e'\rangle}{\langle \mathtt{!}E, e\rangle \to \langle v, e'\rangle} \quad \text{where } v = \neg v'$$

Function calls

Function call expressions are computed by evaluating each of the actual parameters (which are expressions, not variables since parameters are passed by value) in turn from left to right, possibly modifying the store. Then the function body and the function and variable environments at the time of the definition of the function are looked up, and the function body is executed with slight modifications of these environments and the store from the evaluation of the last actual parameter.

The function environment is modified to include the definition of the called function to allow recursion, and the variable environment and the store are modified to bind the formal parameters to the values of the actual parameter expressions. The locations to bind these to are found by repeated use of next.

The value of the function call expression is finally found by looking up retval in the store returned from the computation of the function body – the transition system for computing statements is defined in Section 3.3.2.

$$[\text{fcall-}\mathcal{R}] \quad \frac{\langle E_1, e_1\rangle \to \langle v_1, e_2\rangle \quad \ldots \quad \langle E_n, e_n\rangle \to \langle v_n, e_{n+1}\rangle \quad \langle S, e'_{n+1}\rangle \to e_r^{\mathcal{R}}}{\langle F\,(\,E_1, \ldots, E_n\,)\,;\, , e_1\rangle \to \langle v, e_f\rangle}$$

where $c = (S, (V_1, \ldots, V_n), e'_F, e'_V) = L_F(F, e_{F_1}, e_{F_G})$

and $e'_{n+1} = (e'_F[F \mapsto c], e_{F_G}, e'_V[V_1 \mapsto l_1, \ldots, V_n \mapsto l_n], e_{V_G}, sto_{n+1}[l_1 \mapsto v_1, \ldots, l_n \mapsto v_n])$

where $l_1 = \text{next}, \ldots, l_n = \text{next}$

and $e_f = (e_{F_1}, e_{F_G}, e_{V_1}, e_{V_G}, sto_r)$ and $v = sto_r(\text{retval})$

3.3.2 Statements

The transition system for statements is (**Stat** $\times$ **Env** $\cup$ **Env**, $\to$, **Env**) where $\to$ is defined in the following subsections.

Variable modifi cation

$$[\text{assign}] \quad \frac{\langle E, e\rangle \to \langle v, (e_F, e_{F_G}, e_V, e_{V_G}, sto')\rangle}{\langle V\texttt{=}E\,;\, , e\rangle \to (e_F, e_{F_G}, e_V, e_{V_G}, sto'[l \mapsto v])}$$

where $l = L_V(V, e_V, e_{V_G})$

$[\text{increment}] \quad \langle \texttt{++}V\,;\, , e\rangle \to (e_F, e_{F_G}, e_V, e_{V_G}, sto[l \mapsto v])$

where $l = L_V(V, e_V, e_{V_G})$ and $v = sto(l) + 1$

$[\text{decrement}] \quad \langle \texttt{--}V\,;\, , e\rangle \to (e_F, e_{F_G}, e_V, e_{V_G}, sto[l \mapsto v])$

where $l = L_V(V, e_V, e_{V_G})$ and $v = sto(l) - 1$

Conditionals

$$[\text{if-true}] \quad \frac{\langle E, e\rangle \to \langle v, e'\rangle \quad \langle S, e'\rangle \to e''}{\langle \texttt{if}\,(\,E\,)\,S, e\rangle \to e''} \quad \text{where } v = \text{T}$$

$$[\text{if-false}] \quad \frac{\langle E, e\rangle \to \langle v, e'\rangle}{\langle \texttt{if}\,(\,E\,)\,S, e\rangle \to e'} \quad \text{where } v = \text{F}$$

$$[\text{if-else-true}] \quad \frac{\langle E, e\rangle \to \langle v, e'\rangle \quad \langle S_1, e'\rangle \to e''}{\langle \texttt{if}\,(\,E\,)\,S_1\,\texttt{else}\,S_2, e\rangle \to e''} \quad \text{where } v = \text{T}$$

$$[\text{if-else-false}] \quad \frac{\langle E, e\rangle \to \langle v, e'\rangle \quad \langle S_2, e'\rangle \to e''}{\langle \texttt{if}\,(\,E\,)\,S_1\,\texttt{else}\,S_2, e\rangle \to e''} \quad \text{where } v = \text{F}$$

Looping

$$\text{[while-true]} \quad \frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle \quad \langle S, e'\rangle \rightarrow e'' \quad \langle \texttt{while(}E\texttt{)}S, e''\rangle \rightarrow e'''}{\langle \texttt{while(}E\texttt{)}S, e\rangle \rightarrow e'''}$$

where $v = \text{T}$

$$\text{[while-false]} \quad \frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle}{\langle \texttt{while(}E\texttt{)}S, e\rangle \rightarrow e'}$$

where $v = \text{F}$

Blocks

A block statement is computed by evaluating each of the contained statements in turn, possibly modifying the variable environment and the store, and at last recovering the original variable environment.

$$\text{[block]} \quad \frac{\langle S_1, e_1\rangle \rightarrow e_2 \quad \langle S_2, e_2\rangle \rightarrow e_3 \quad \ldots \quad \langle S_n, e_n\rangle \rightarrow e_{n+1}}{\langle \texttt{\{}S_1 S_2 \ldots S_n\texttt{\}}, e_1\rangle \rightarrow (e_{F_1}, e_{F_G}, e_{V_1}, e_{V_G}, sto_{n+1})}$$

Variable definitions

To simplify the semantic specification, we consider only variable definitions on the form:

variable-definition ::= [`const`] typename identifier [= expression]**;**

Definitions of multiple variables in one variable definition, i.e. definitions on the form

variable-definition ::= [`const`] typename identifier [= expression]
(**,** identifier [= expression])***;**

are simply considered shorthand for a series of definition on the first form. Hence, a statement such as

```
int a = 12, b, c = 87;
```

shall be rewritten to

```
int a = 12;
int b;
int c = 87;
```

Thus variable definition statements are covered by the following rules.

[var-stat] $\langle \texttt{[const]}\ \tau\ V\texttt{;}, e\rangle \rightarrow (e_F, e_{F_G}, e_V[V \mapsto l], e_{V_G}, sto)$

where $l = \text{next}$

[var-stat-E]
$$\frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle}{\langle \texttt{[const]}\ \tau\ V \texttt{ = } E\texttt{;}, e\rangle \rightarrow (e_F, e_{F_G}, e_V[V \mapsto l], e_{V_G}, sto'[l \mapsto v])}$$

where $l = \text{next}$

Function calls

Function call statements are executed in the same manner as function call expressions, except the return value is ignored.

[fstat-$\mathcal{R}$]
$$\frac{\langle E_1, e_1\rangle \rightarrow \langle v_1, e_2\rangle \quad \ldots \quad \langle E_n, e_n\rangle \rightarrow \langle v_n, e_{n+1}\rangle \quad \langle S, e'_{n+1}\rangle \rightarrow e_r^{\mathcal{R}}}{\langle F\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{);}, e_1\rangle \rightarrow e_f}$$

where $c = (S, (V_1, \ldots, V_n), e'_F, e'_V) = L_F(F, e_{F_1}, e_{F_G})$

and $e'_{n+1} = (e'_F[F \mapsto c], e_{F_G}, e'_V[V_1 \mapsto l_1, \ldots, V_n \mapsto l_n], e_{V_G},$
$sto_{n+1}[l_1 \mapsto v_1, \ldots, l_n \mapsto v_n])$

where $l_1 = \text{next}, \ldots, l_n = \text{next}$

and $e_f = (e_{F_1}, e_{F_G}, e_{V_1}, e_{V_G}, sto_r)$

For a **void** function the body will not be $\mathcal{R}$-tagged by a **return** statement, but simply stop executing when the last statement is finished. Consequently, we define an extra rule [fstat-no-$\mathcal{R}$] which is exactly the same as [fstat-$\mathcal{R}$], except $\langle S, e'_{n+1}\rangle \rightarrow e_r^{\mathcal{R}}$ is replaced by $\langle S, e'_{n+1}\rangle \rightarrow e_r$.

Return statements

To execute a **return** statement, the returned expression is first evaluated. The value of that expression is then bound to the "retval" symbol, and the resulting environment is $\mathcal{R}$-tagged.

[return]
$$\frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle}{\langle \texttt{return}\ E\texttt{;}, e\rangle \rightarrow (e_F, e_{F_G}, e_V, e_{V_G}, sto'[\text{retval} \mapsto v])^{\mathcal{R}}}$$

$\mathcal{R}$-transitions

A transition rule for all statements covers the case in which a statement is evaluated in an $\mathcal{R}$-tagged environment:

$$[\mathcal{R}\text{-statement}] \qquad \langle S, e^{\mathcal{R}} \rangle \rightarrow e^{\mathcal{R}}$$

3.3.3 Toplevel constructs

The transition system for toplevel constructs is (**Topl** $\times$ **Env** $\times$ **CL** $\cup$ **Env** $\times$ **CL**, $\rightarrow$, **Env** $\times$ **CL**) where $\rightarrow$ is defined in the following subsections.

Import directive

$$[\text{import}] \quad \langle \texttt{import "filename";}, e, cl \rangle \rightarrow \langle e', cl' \rangle$$
where e' and cl' are obtained by applying the semantic rules to the contents of the file `filename`

Create statements

$$[\text{create}] \quad \langle \texttt{create}\ C\ \texttt{of}\ F\texttt{(}E_1, \ldots, E_n\texttt{);}, e, cl \rangle \rightarrow \langle e, cl' \rangle$$
where $x = \mathcal{V}(C)$ and $cl' = cl :: (x, F, (E_1, \ldots, E_n))$

Declarations

Declaration of variables and functions is a syntactic tool for circumventing the usual definition-before-usage rule, and the declarations have no meaning after the contextual syntactic analysis of the code where identifiers are associated with definitions.

Furthermore, our distinction in the semantic specification between local and global environments ensures that it is possible to use a global identifier in a function body even if the identifier is defined after the definition of the function.

Hence, the transition rules will simply skip the declarations (here $\tau_v = \tau \cup \{\texttt{void}\}$):

[var-decl] $\langle \texttt{extern}\,[\texttt{const}]\,\tau\; V\;[= E]\texttt{;}, e, cl\rangle \rightarrow \langle e, cl\rangle$

[fun-decl] $\langle \texttt{extern}\,\tau_{\text{v}}\,F(V_1,\ldots,V_n)\texttt{;}, e, cl\rangle \rightarrow \langle e, cl\rangle$

Variable definitions

Toplevel variable definitions shall be rewritten in the same manner as the variable definition statements. Thus they are covered by the following rules.

[var-top] $\langle [\texttt{const}]\,\tau\; V\texttt{;}, e, cl\rangle \rightarrow \langle (e_F, e_{F_G}, e_V, e_{V_G}[V \mapsto l], sto), cl\rangle$

where $l = \text{next}$

[var-top-E]
$$\frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle}{\langle [\texttt{const}]\,\tau\; V \texttt{ = } E\texttt{;}, e, cl\rangle \rightarrow \langle (e_F, e_{F_G}, e_V, e_{V_G}[V \mapsto l], sto'[l \mapsto v]), cl\rangle}$$

where $l = \text{next}$

Function definitions

Although the distinction between local and global environments makes it possible to use any global identifier inside any function body, one problem with initialization of global variables persists. Consider the following example:

```
extern int f();
int x = f();
int f() { return 15; }
```

The function f needs to be known prior to the initialization of x. Consequently, we define that the program shall be reordered so that the semantic rules are applied to the function definitions before they are applied to the variable definitions.

The transition rule for function definitions simply inserts the function body (S_{block} is a block statement) and its context into the global function environment. Then f will be available in the variable definition above because of the reordering requirement.

[fun-top] $\langle \tau_{\text{v}}\,F(V_1,\ldots,V_n)\,S_{\text{block}}, e, cl\rangle \rightarrow \langle (e_F, e'_{F_G}, e_V, e_{V_G}, sto), cl\rangle$

where $e'_{F_G} = e_{F_G}[F \mapsto (S_{\text{block}}, (V_1,\ldots,V_n), e_F, e_V)]$

Enum definitions

Since **enum**s exist for the sole purpose of type checking, they have no semantic meaning and the transition rule simply skips the definition.

[enum-top] $\langle \texttt{enum}\, \eta\ \{\sigma_1, \ldots, \sigma_n\}, e, cl\rangle \rightarrow \langle e, cl\rangle$

Type definitions

From a semantic viewpoint, type definitions are scopes with function, variable and **enum** definition members.

[type-def] $$\frac{\langle M_1, e_1\rangle \rightarrow e_2 \quad \langle M_2, e_2\rangle \rightarrow e_3 \quad \ldots \quad \langle M_n, e_n\rangle \rightarrow e_{n+1}}{\langle \texttt{type}\ \ \eta\ \ \{M_1 M_2 \ldots M_n\}, e_1, cl\rangle \rightarrow \langle (e_{F_1}, e_{F_G}, e_{V_1}, e_{V_G}, sto_{n+1}), cl\rangle}$$

The transition system for the members $M_1, \ldots, M_n$ is (**Memb** $\times$ **Env** $\cup$ **Env**, $\rightarrow$, **Env**)) where $\rightarrow$ is defined by the following transition rules.

[fun-type] $\langle \tau_{\mathrm{v}}\ F(V_1, \ldots, V_n)\ S_{\mathrm{block}}, e\rangle \rightarrow (e'_F, e_{F_G}, e_V, e_{V_G}, sto)$

where $e'_F = e_F[F \mapsto (S_{\mathrm{block}}, (V_1, \ldots, V_n), e_F, e_V)]$

[var-type] $\langle [\texttt{watched}]\ [\texttt{const}]\ \tau\ V\,\texttt{;}, e\rangle \rightarrow e'$

where $e' = (e_F, e_{F_G}, e_V[V \mapsto l], e_{V_G}, sto)$ and $l = \mathrm{next}$

[var-type-E] $$\frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle}{\langle [\texttt{watched}]\ [\texttt{const}]\ \tau\ V\ \texttt{=}\ E\,\texttt{;}, e\rangle \rightarrow e'}$$

where $e' = (e_F, e_{F_G}, e_V[V \mapsto l], e_{V_G}, sto'[l \mapsto v])$ and $l = \mathrm{next}$

[enum-type] $\langle \texttt{enum}\, \eta\ \{\sigma_1, \ldots, \sigma_n\}, e\rangle \rightarrow e$

3.4 Built-in functions

Before the contextual analysis is started, the function environment is equipped with the built-in functions intify and floatify .

The function intify is defined as

```
int intify(float x)
{
```

```
    return I(x);
}
```

where $\mathcal{I}$ is an expression with a value defined by the following transition rule:

$$\text{[intify]} \qquad \frac{\langle V, e\rangle \rightarrow \langle v', e'\rangle}{\langle \mathcal{I}(V), e\rangle \rightarrow \langle v, e'\rangle} \qquad \text{where } v = \lfloor v' \rfloor \text{ and the type of } v \text{ is } \textbf{int}$$

The function fbatify is defined as

```
float floatify(int x)
{
    return F(x);
}
```

where $\mathcal{F}$ is defined as a normal variable evaluation except the type of the resulting value of the evaluation is **float**.

Chapter 4

Translation to JVM code

This chapter defines the syntax and semantics for a selected part of the Java virtual machine, then defines a translation function $\mathcal{T}$ for converting strings in our language to JVM strings, and finally proves that the semantics of a program before and after application of $\mathcal{T}$ are the same.

4.1 The Java VM

The Java virtual machine is a stack-based abstract machine [7] for executing a low-level language. It is not the only abstract machine that we could have chosen for the target language, but some of the benefits of the JVM are:

- It is completely stack based, thus avoiding the problem of register allocation.
- It is a real-world machine designed for practical use.
- It is very widely deployed.
- It is stable and known to work.

Instead of specifying a translation to the binary code that implementations of the JVM execute (from `.class` files), we will specify a translation to an intermediate human-readable assembler language, the *Java assembler interface* [6], Jasmin. To reduce the complexity of the formal model, we also skip some of the minor JVM details, especially those that stem from its object-oriented ISA parts, and instead focus on the central building blocks, expressions and statements.

4.1.1 The basic model

Our formal model of the JVM is inspired from the model of an abstract machine in [8] and processes an array of instructions c, keeping track of the current location $pc \in \mathbb{N}_0$ within the array, the stack $s \in \textbf{Stack}$ and the free store $sto \in \textbf{Sto}_m$. The store is defined as a mapping from locations to values as in Section 3.2.2 except the values are in $\mathbb{Z} \cup \mathbb{R}$, whereas a stack is an ordered sequence of values from $\mathbb{Z} \cup \mathbb{R}$ that supports pushing and popping of elements. We use the notation $v :: s$ to denote pushing or popping of the element v from stack s.

Since we keep the instruction array c fixed throughout the execution of a program, the configurations belong to the set $\mathbb{N}_0 \times \textbf{Stack} \times \textbf{Sto}_m$ where the first component is the program counter pc. A terminal configuration is a configuration where $pc = |c|$, i.e. there are no more instructions left. We denote the transition relation by $\Rightarrow$ and define it with the rules in the following sections.

4.1.2 Types

In the specification of the semantics for our high-level language, it was possible to ignore the types of the variables almost completely since most operators have the same syntax and similar semantics no matter what the type of the operands is (and from the syntax specification we are assured that the operands have types that are meaningful for the particular operator).

For most of the JVM instructions, the situation is different. The virtual machine has two basic types, integer and floating point numbers, with different instructions for manipulating them. Most of the instructions are semantically the same, though, so in the following we will not repeat the specification for all of them. In general, the integer-specific instructions begin with I and the floating point instructions with F.

Since there is no concept of boolean and enumerated values, we map them to integers with the function $\mathcal{C} : \textbf{Val} \to \mathbb{Z} \cup \mathbb{R}$ which is defined as

$$\mathcal{C}(v) = \begin{cases} 1 & \text{if } v = \text{T} \\ 0 & \text{if } v = \text{F} \\ \mathcal{E}(v) & \text{if } v \in \mathbb{E} \\ v & \text{else} \end{cases}$$

where $\mathcal{E}$ within a given enum type assigns a unique integer to each of the symbols.

4.1.3 Variables and functions

Since variables are allocated at specific points in the JVM (e.g. at the beginning of the enclosing routine), we shall not model the semantics of variable and function definitions directly, but instead introduce an abstraction; we simply define $\nu(V)$ to be the location of the variable V. Likewise we define $\varphi(F)$ to be the code address of the body of the function F.

4.1.4 Stack manipulation instructions

[LDC v]	$\langle pc, s, sto_m \rangle \Rightarrow \langle pc + 1, v :: s, sto_m \rangle$
[ILOAD l]	$\langle pc, s, sto_m \rangle \Rightarrow \langle pc + 1, sto_m(l) :: s, sto_m \rangle$
[ISTORE l]	$\langle pc, v :: s, sto_m \rangle \Rightarrow \langle pc + 1, s, sto_m[l \mapsto v] \rangle$
[POP]	$\langle pc, v :: s, sto_m \rangle \Rightarrow \langle pc + 1, s, sto_m \rangle$
[DUP]	$\langle pc, v :: s, sto_m \rangle \Rightarrow \langle pc + 1, v :: v :: s, sto_m \rangle$
[DUP_X1]	$\langle pc, v_1 :: v_2 :: s, sto_m \rangle \Rightarrow \langle pc + 1, v_1 :: v_2 :: v_1 :: s, sto_m \rangle$
[NOP]	$\langle pc, s, sto_m \rangle \Rightarrow \langle pc + 1, s, sto_m \rangle$
[SWAP]	$\langle pc, v_1 :: v_2 :: s, sto_m \rangle \Rightarrow \langle pc + 1, v_2 :: v_1 :: s, sto_m \rangle$

FLOAD and FSTORE, are defined in the same way as ILOAD and ISTORE for floating point values. The instructions ALOAD and ASTORE are similarly used for loading and storing object references.

4.1.5 Arithmetic instructions

[IADD]	$\langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc + 1, (v_l + v_r) :: s, sto_m \rangle$
[ISUB]	$\langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc + 1, (v_l - v_r) :: s, sto_m \rangle$
[IMUL]	$\langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc + 1, (v_l v_r) :: s, sto_m \rangle$
[IDIV]	$\langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc + 1, (\lfloor v_l / v_r \rfloor) :: s, sto_m \rangle$
[FDIV]	$\langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc + 1, (v_l / v_r) :: s, sto_m \rangle$
[IREM]	$\langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc + 1, (v_l - \lfloor v_l / v_r \rfloor v_r) :: s, sto_m \rangle$
[INEG]	$\langle pc, v :: s, sto_m \rangle \Rightarrow \langle pc + 1, (-v) :: s, sto_m \rangle$

FADD, FSUB, FMUL, FREM, FINC and FNEG are defined similarly for floating point numbers.

4.1.6 Jump instructions

Jump instructions are modelled with labels. To facilitate this, we define a label x instruction to do nothing and $\lambda(x)$ to be the code address where the label instruction x is located.

$$
\begin{array}{ll}
[x:] & \langle pc, s, sto_m \rangle \Rightarrow \langle pc+1, s, sto_m \rangle \\
[\textsc{goto}\ x] & \langle pc, s, sto_m \rangle \Rightarrow \langle \lambda(x), s, sto_m \rangle
\end{array}
$$

For integers, there are a number of instructions that jump conditionally depending on how the top of the stack compares to 0.

$$
\begin{array}{lll}
[\textsc{ifeq}\ x] & \langle pc, v :: s, sto_m \rangle \Rightarrow \langle \lambda(x), s, sto_m \rangle & \text{if } v = 0 \\
[\textsc{ifeq}\ x] & \langle pc, v :: s, sto_m \rangle \Rightarrow \langle pc+1, s, sto_m \rangle & \text{else}
\end{array}
$$

The instruction IFNE is defined similarly for $v \neq 0$, IFGT is defined for $v > 0$, IFGE for $v \geq 0$, IFLT for $v < 0$ and IFLE for $v \leq 0$. For floating point comparisons, there is the FCMPG instruction.

$$
\begin{array}{lll}
[\textsc{fcmpg}\ x] & \langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc+1, 0 :: s, sto_m \rangle & \text{if } v_l = v_r \\
[\textsc{fcmpg}\ x] & \langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc+1, -1 :: s, sto_m \rangle & \text{if } v_l < v_r \\
[\textsc{fcmpg}\ x] & \langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc+1, 1 :: s, sto_m \rangle & \text{if } v_l > v_r
\end{array}
$$

The JVM also defines instructions for performing jumps depending on a comparison of the two top elements of the stack. Below is the semantics for [IF_ICMPEQ x]; we define the instruction [IF_ICMPNEQ x] similarly for $\neq$, [IF_ICMPGT x] for $>$, [IF_ICMPGE x] for $\geq$, [IF_ICMPNLT x] for $<$ and [IF_ICMPLE x] for $\leq$.

$$
\begin{array}{lll}
[\textsc{if_icmpeq}\ x] & \langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle \lambda(x), s, sto_m \rangle & \text{if } v_l = v_r \\
[\textsc{if_icmpeq}\ x] & \langle pc, v_r :: v_l :: s, sto_m \rangle \Rightarrow \langle pc+1, s, sto_m \rangle & \text{else}
\end{array}
$$

A function call consists of popping the parameters from the stack and transfering them to the store, saving the return address on the stack and finally jumping to

the function block statement. Here, $V_1, \ldots, V_n$ are the formal parameters:

[INVOKESTATIC F] $\langle pc, v_n :: \ldots :: v_1 :: s, sto_m \rangle \Rightarrow \langle \varphi(F), pc :: s, sto'_m \rangle$
where $sto'_m = sto_m[\nu(V_1) \mapsto v_1, \ldots, \nu(V_n) \mapsto v_n]$

[INVOKEVIRTUAL F] $\langle pc, v_n :: \ldots :: v_1 :: o :: s, sto_m \rangle \Rightarrow \langle \varphi(F), pc :: s, sto'_m \rangle$
where $sto'_m = sto_m[\nu(V_1) \mapsto v_1, \ldots, \nu(V_n) \mapsto v_n]$
and o is an object reference for finding the function

A return jumps to the saved return address, leaving the returned value on top of the stack (FRETURN is defined similarly):

[IRETURN] $\langle pc, v :: pc_r :: s, sto_m \rangle \Rightarrow \langle pc_r + 1, v :: s, sto_m \rangle$

For **void** procedures, we need RETURN instead:

[RETURN] $\langle pc, pc_r :: s, sto_m \rangle \Rightarrow \langle pc_r + 1, s, sto_m \rangle$

4.1.7 Type conversion instructions

The JVM defines several instructions for converting between the different basic types. Below are the specifications for integer-floating point conversions; I2D, D2I, F2D and D2F are defined in the same manner for conversions to and from double-precision floating point numbers.

[I2F] $\langle pc, v_i :: s, sto_m \rangle \Rightarrow \langle pc + 1, v_f :: s, sto_m \rangle$
where v_i is an integer and v_f is v_i in the floating point representation

[F2I] $\langle pc, v_f :: s, sto_m \rangle \Rightarrow \langle pc + 1, v_i :: s, sto_m \rangle$
where v_f is a floating point number and $v_i = \lfloor v_f \rfloor$ is an integer

4.2 Translation

The translation of programs from our language to JVM code is performed by the recursive function $\mathcal{T}$ defined in the following sections. Since our model of the JVM does not take variable and function environments into account, defining a useful

semantic equivalence relation for the toplevel transitions would be difficult. Hence, we shall focus on the expressions and statements, and not specify the translation step for the toplevel constructs.

For improved readability, a semicolon will denote sequencing of instructions, and whenever we define a label, it is assumed that it is uniquely defined so that name clashes do not occur. The function $\mathcal{V}$ is reused from Section 3.2.4.

4.2.1 Expressions

$$\mathcal{T}(C) = \text{LDC } v \qquad \text{where } v = \mathcal{C}(\mathcal{V}(C))$$
$$\mathcal{T}(\texttt{(}E\texttt{)}) = \mathcal{T}(E); \text{NOP}$$

The following expressions assume that the type of the immediate constituents is integer. The similar translations for floating point numbers are constructed by replacing I with F.

$$\mathcal{T}(V) = \text{ILOAD } l \qquad \text{where } l = \nu(V)$$
$$\mathcal{T}(E_1\texttt{+}E_2) = \mathcal{T}(E_1); \mathcal{T}(E_2); \text{IADD}$$
$$\mathcal{T}(E_1\texttt{-}E_2) = \mathcal{T}(E_1); \mathcal{T}(E_2); \text{ISUB}$$
$$\mathcal{T}(E_1\texttt{*}E_2) = \mathcal{T}(E_1); \mathcal{T}(E_2); \text{IMUL}$$
$$\mathcal{T}(E_1\texttt{/}E_2) = \mathcal{T}(E_1); \mathcal{T}(E_2); \text{IDIV}$$
$$\mathcal{T}(E_1\texttt{\%}E_2) = \mathcal{T}(E_1); \mathcal{T}(E_2); \text{IREM}$$
$$\mathcal{T}(\texttt{-}E) = \mathcal{T}(E); \text{INEG}$$

The boolean operators need only be defined for integers (i.e. converted booleans):

$$\mathcal{T}(E_1\texttt{\&\&}E_2) = \mathcal{T}(E_1); \text{IFEQ } \mathit{false}; \mathcal{T}(E_2); \text{IFEQ } \mathit{false}; \text{LDC } \mathcal{C}(\text{T}); \text{GOTO } \mathit{end}; \mathit{false}\text{:}; \text{LDC } \mathcal{C}(\text{F}); \mathit{end}\text{:}$$
$$\mathcal{T}(E_1\texttt{||}E_2) = \mathcal{T}(E_1); \text{IFNE } \mathit{true}; \mathcal{T}(E_2); \text{IFNE } \mathit{true}; \text{LDC } \mathcal{C}(\text{F}); \text{GOTO } \mathit{end}; \mathit{true}\text{:}; \text{LDC } \mathcal{C}(\text{T}); \mathit{end}\text{:}$$
$$\mathcal{T}(\texttt{!}E) = \mathcal{T}(E); \text{IFEQ } \mathit{true}; \text{LDC } \mathcal{C}(\text{F}); \text{GOTO } \mathit{end}; \mathit{true}\text{:}; \text{LDC } \mathcal{C}(\text{T}); \mathit{end}\text{:}$$

The translation of $E_1\texttt{==}E_2$ for integers is

$$\mathcal{T}(E_1\texttt{==}E_2) = \mathcal{T}(E_1); \mathcal{T}(E_2); \text{IF_ICMPEQ } \mathit{true}; \text{LDC } \mathcal{C}(\text{F}); \text{GOTO } \mathit{end}; \mathit{true}\text{:}; \text{LDC } \mathcal{C}(\text{T}); \mathit{end}\text{:}$$

$\mathcal{T}(E_1\,\texttt{!=}E_2)$ is defined by replacing IF_ICMPEQ with IF_ICMPNE, for $\mathcal{T}(E_1\texttt{<}E_2)$ replacing with IF_ICMPLT, $\mathcal{T}(E_1\texttt{>}E_2)$ with IF_ICMPGT, $\mathcal{T}(E_1\texttt{<=}E_2)$ with IF_ICMPLE and $\mathcal{T}(E_1\texttt{>=}E_2)$ with IF_ICMPGE.

For floating point values, it is

$$\begin{aligned}\mathcal{T}(E_1\texttt{==}E_2) = {} & \mathcal{T}(E_1); \mathcal{T}(E_2); \text{FCMPG}; \text{IFEQ } \mathit{true}; \text{LDC } \mathcal{C}(\text{F}); \text{GOTO } \mathit{end};\\ & \mathit{true}{:}; \text{LDC } \mathcal{C}(\text{T}); \mathit{end}{:}\end{aligned}$$

and $\mathcal{T}(E_1\,\texttt{!=}E_2)$ is defined by replacing IFEQ with IFNE, $\mathcal{T}(E_1\texttt{<}E_2)$ is defined by replacing with IFLT, $\mathcal{T}(E_1\texttt{>}E_2)$ with IFGT, $\mathcal{T}(E_1\texttt{<=}E_2)$ with IFLE and $\mathcal{T}(E_1\texttt{>=}E_2)$ with IFGE.

Function parameters are transfered on the stack. Thus a function call consists of evaluating the parameters and transfering control:

$$\mathcal{T}(F(E_1, \ldots, E_n)) = \mathcal{T}(E_1); \ldots; \mathcal{T}(E_n); \text{INVOKESTATIC } F$$

The function will then leave the computed value on top of the stack upon returning. This translation actually only covers calls of one kind of functions; in some cases, an INVOKEVIRTUAL instruction is needed instead of INVOKESTATIC, but the details are beyond the abstraction level of our JVM model – Section 5.5 discusses them for the actual implementation.

The power and the random operator have no direct equivalent in the JVM instruction set. Instead we translate them to function calls. For integers, they are translated to:

$$\begin{aligned}\mathcal{T}(E_1\,\hat{}\,E_2) = {} & \mathcal{T}(E_1); \text{I2D}; \mathcal{T}(E_2); \text{I2D};\\ & \text{INVOKESTATIC java/lang/Math/pow(DD)}; \text{D2I}\\ \mathcal{T}(E_1 \ldots E_2) = {} & \mathcal{T}(E_1); \text{DUP}; \mathcal{T}(E_2); \text{SWAP}; \text{ISUB}; \text{LDC } 1; \text{IADD}; \text{I2F};\\ & \text{GETSTATIC Scenario/random Ljava/util/Random;};\\ & \text{INVOKEVIRTUAL java/util/Random/nextFloat()F};\\ & \text{FMUL}; \text{F2I}; \text{IADD}\end{aligned}$$

For floating point numbers, we translate them to:

$$\begin{aligned}\mathcal{T}(E_1\,\hat{}\,E_2) = {} & \mathcal{T}(E_1); \text{F2D}; \mathcal{T}(E_2); \text{F2D};\\ & \text{INVOKESTATIC java/lang/Math/pow(DD)}; \text{D2F}\\ \mathcal{T}(E_1 \ldots E_2) = {} & \mathcal{T}(E_1); \text{DUP}; \mathcal{T}(E_2); \text{SWAP}; \text{FSUB};\\ & \text{GETSTATIC Scenario/random Ljava/util/Random;};\\ & \text{INVOKEVIRTUAL java/util/Random/nextFloat()F};\\ & \text{FMUL}; \text{FADD}\end{aligned}$$

Hence, regarding the random operator, we could obviously offer the same functionality by providing a function that returns a random number between 0 and 1. But although the ... construct is not more powerful than a random number generator function, it is much more convenient and conveys the intended meaning directly.

The same observation is even more true for the multiple-choice construct. It is translated to first computing the sums $E_{w_1}, E_{w_1} + E_{w_2}, \ldots, E_{w_1} + E_{w_2} + \ldots + E_{w_n}$, putting them on the stack. The last sum is then multiplied by a random number between 0 and 1, and compared with the sums, one by one. If it is larger than a given sum, the corresponding right side expression is evaluated and the computation stops. So basically, each colon pair is assigned an interval with a size that depends on the left side value, a random number is generated and the pair in which interval the number falls in is chosen.

$$
\begin{aligned}
&\mathcal{T}(E_{w_1} : E_{v_1} \mid \ldots \mid E_{w_n} : E_{v_n}) = \\
&\quad \mathcal{T}(E_{w_1});\quad \text{DUP}; \mathcal{T}(E_{w_2}); \text{FADD}; \ldots; \text{DUP}; \mathcal{T}(E_{w_n}); \text{FADD} \\
&\quad \text{ALOAD } 1; \text{INVOKEVIRTUAL } \text{java/util/Random/nextFloat()F}; \text{FMUL}; \\
&\quad \text{DUP_X1}; \text{FCMPG}; \text{IFGT } \mathit{next}_{n-2}; \{\text{POP};\}^{n-1} \mathcal{T}(E_{v_n}); \text{GOTO } \mathit{end}; \mathit{next}_{n-2}: \\
&\quad \text{DUP_X1}; \text{FCMPG}; \text{IFGT } \mathit{next}_{n-3}; \{\text{POP};\}^{n-2} \mathcal{T}(E_{v_{n-1}}); \text{GOTO } \mathit{end}; \mathit{next}_{n-3}: \\
&\quad \ldots \\
&\quad \text{DUP_X1}; \text{FCMPG}; \text{IFGT } \mathit{next}_0; \text{POP}; \mathcal{T}(E_{v_2}); \text{GOTO } \mathit{end}; \mathit{next}_0: \\
&\quad \text{POP}; \mathcal{T}(E_{v_1}); \mathit{end}:
\end{aligned}
$$

The DUP_X1 ensures that a copy of the random number is always available on the stack below the top sum. The sequences of POP statements are needed to clear the stack in those cases where not all of the sums are needed, e.g. for the first case (E_{v_n}) we need to issue $n-1$ POPs.

4.2.2 Statements

The statements that changes variable values depend on the type of the variable. The translation for integers follows, for floating point variables we substitute F for I and 1 for 1.0:

$$
\begin{aligned}
\mathcal{T}(V = E\,;) &= \mathcal{T}(E); \text{ISTORE } l && \text{where } l = \nu(V) \\
\mathcal{T}(\texttt{++}V\,;) &= \text{ILOAD } l; \text{LDC } 1; \text{IADD}; \text{ISTORE } l && \text{where } l = \nu(V) \\
\mathcal{T}(\texttt{--}V\,;) &= \text{ILOAD } l; \text{LDC } 1; \text{ISUB}; \text{ISTORE } l && \text{where } l = \nu(V) \\
\mathcal{T}(\texttt{return}\,E\,;) &= \mathcal{T}(E); \text{IRETURN}
\end{aligned}
$$

The rest of the statements do not depend on the type of the expressions involved.

$$
\begin{aligned}
\mathcal{T}(\{S_1,\ldots,S_n\}) &= \mathcal{T}(S_1);\ldots;\mathcal{T}(S_n)\\
\mathcal{T}(\texttt{[const]}\ \tau\ V\texttt{;}) &= \text{NOP}\\
\mathcal{T}(\texttt{[const]}\ \tau\ V = E\texttt{;}) &= \mathcal{T}(V = E\texttt{;})\\
\mathcal{T}(\texttt{while(}E\texttt{)}S) &= \mathit{start}{:};\mathcal{T}(E);\text{IFEQ}\ \mathit{done};\mathcal{T}(S);\text{GOTO}\ \mathit{start};\\
&\quad\ \mathit{done}{:}\\
\mathcal{T}(\texttt{if(}E\texttt{)}S) &= \mathcal{T}(E);\text{IFEQ}\ \mathit{done};\mathcal{T}(S);\mathit{done}{:}\\
\mathcal{T}(\texttt{if(}E\texttt{)}S_1\ \text{else}\ S_2) &= \mathcal{T}(E);\text{IFEQ}\ \mathit{else};\mathcal{T}(S_1);\text{GOTO}\ \mathit{done};\\
&\quad\ \mathit{else}{:};\mathcal{T}(S_2);\mathit{done}{:}
\end{aligned}
$$

The translation of a function call statement depends on whether the function is a **void** function. If not, the returned value must be popped of the stack.

$$
\begin{aligned}
\mathcal{T}(F(E_1,\ldots,E_n)) &= \mathcal{T}(E_1);\ldots;\mathcal{T}(E_n);\text{INVOKESTATIC}\ F\\
&\quad \text{if the return type of } F \text{ is } \textbf{void}\\
\mathcal{T}(F(E_1,\ldots,E_n)) &= \mathcal{T}(E_1);\ldots;\mathcal{T}(E_n);\text{INVOKESTATIC}\ F;\text{POP}\\
&\quad \text{else}
\end{aligned}
$$

4.3 Correctness of the translation

To prove that the translation $\mathcal{T}$ is correct we need to define what semantic equivalence means. First we define that $sto \overset{\mathcal{C}}{=} sto_m$ if and only if sto is defined for the same locations as sto_m (except for retval which is not needed in the JVM) and $\mathcal{C}(sto(l)) = sto_m(l)$ for any such location l.

Definition 1 *An expression E is semantically equivalent with the code* $c[pc,\ldots,pc+k-1]$ *when a transition*

$$\langle E,(e_F,e_{F_G},e_V,e_{V_G},sto)\rangle \rightarrow \langle v,(e_F,e_{F_G},e_V,e_{V_G},sto')\rangle$$

exists if and only if there is a finite computation sequence

$$\langle pc,s,sto_m\rangle \Rightarrow^* \langle pc+k,\mathcal{C}(v) :: s,sto'_m\rangle$$

where $sto \overset{\mathcal{C}}{=} sto_m$ *and* $sto' \overset{\mathcal{C}}{=} sto'_m$*. A statement S is semantically equivalent with the code* $c[pc,\ldots,pc+k-1]$ *when a transition*

$$\langle S,(e_F,e_{F_G},e_V,e_{V_G},sto)\rangle \rightarrow (e_F,e_{F_G},e'_V,e_{V_G},sto'),$$

exists if and only if there is a finite computation sequence

$$\langle pc, s, sto_m \rangle \Rightarrow^* \langle pc + k, s, sto'_m \rangle$$

where $sto \stackrel{\mathcal{C}}{=} sto_m$ *and* $sto' \stackrel{\mathcal{C}}{=} sto'_m$.

So a piece of machine code is equivalent to an expression provided both terminate and both end up with the same value and same store, and equivalent to a statement if both terminate and end up with the same store. The correctness proof is then based on showing that expressions and statements are semantically equivalent with the machine code that $\mathcal{T}$ produces.

Since the generated machine code for integers and floating point numbers is the same barring the replacement of I with F in most cases, we shall only prove for integers except where completely different instructions are generated.

4.3.1 One way

First we shall prove that if the untranslated code terminates, then the translated code terminates too, and with the same result. We will not prove this for the power, random and multiple choice operators since they are mostly an exercise in assembly programming.

Lemma 1 *For all expressions* E*, if a transition for* E *exists*

$$\langle E, (e_F, e_{F_G}, e_V, e_{V_G}, sto) \rangle \rightarrow \langle v, (e_F, e_{F_G}, e_V, e_{V_G}, sto') \rangle,$$

then there is a finite computation sequence

$$\langle pc, s, sto_m \rangle \Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m \rangle$$

where $c[pc, \ldots, pc + |\mathcal{T}(E)| - 1] = \mathcal{T}(E)$ *and* $sto \stackrel{\mathcal{C}}{=} sto_m$ *and* $sto' \stackrel{\mathcal{C}}{=} sto'_m$.

And for all statements S*, if a transition for* S *exists*

$$\langle S, (e_F, e_{F_G}, e_V, e_{V_G}, sto) \rangle \rightarrow (e_F, e_{F_G}, e'_V, e_{V_G}, sto'),$$

then there is a finite computation sequence

$$\langle pc, s, sto_m \rangle \Rightarrow^* \langle pc + |\mathcal{T}(S)|, s, sto'_m \rangle$$

where $c[pc, \ldots, pc + |\mathcal{T}(S)| - 1] = \mathcal{T}(S)$ *and* $sto \stackrel{\mathcal{C}}{=} sto_m$ *and* $sto' \stackrel{\mathcal{C}}{=} sto'_m$.

We prove the lemma by induction on the height of the derivation trees of the transitions.

The case C: Using the rule [lit], we have $\langle C, e\rangle \rightarrow \langle v, e\rangle$ where $v = \mathcal{V}(C)$. Now $\mathcal{T}(C) =$ LDC $\mathcal{C}(v)$ which results in the transition $\langle pc, s, sto_m\rangle \Rightarrow \langle pc+1, \mathcal{C}(v) :: s, sto_m\rangle$.

The case (E): If there is a transition for this expression, the rule [paren]

$$\frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle}{\langle\, (\,E\,)\, , e\rangle \rightarrow \langle v, e'\rangle}$$

must have been used. Now $\mathcal{T}(\,(\,E\,)\,) = \mathcal{T}(E)$; NOP. By the premise of the transition rule, since the derivation tree for E is of height less than the derivation tree for the whole expression the following computation sequence must exist by the inductive hypothesis

$$\langle pc, s, sto_m\rangle \Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m\rangle.$$

Then the NOP computation

$$\langle pc + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m\rangle \Rightarrow \langle pc + |\mathcal{T}(E)| + 1, \mathcal{C}(v) :: s, sto'_m\rangle$$

concludes the proof of the case.

The case V: Using the rule [var], we have $\langle V, e\rangle \rightarrow \langle v, e\rangle$ where $v = sto(L_V(V, e_V, e_{V_G}))$. Now $\mathcal{T}(V) =$ ILOAD l where $l = \nu(V)$ which results in the transition $\langle pc, s, sto_m\rangle \Rightarrow \langle pc+1, sto_m(l) :: s, sto_m\rangle$. Since $\nu(V)$ gives the location of V, this proves the case.

The case $E_1 + E_2$: If there is a transition for this expression, the rule [plus]

$$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\text{+}E_2, e\rangle \rightarrow \langle v, e''\rangle} \quad \text{where } v = v_1 + v_2,$$

must have been used. $\mathcal{T}(E_1\text{+}E_2) = \mathcal{T}(E_1); \mathcal{T}(E_2);$ IADD. With the assumption that the lemma holds for the immediate constituents (the derivation trees of which have height less than the tree of the entire expression), this code will bring the machine through the following transition sequences:

$$\begin{aligned}\langle pc, s, sto_m\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto'_m\rangle \\ &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)| + |\mathcal{T}(E_2)|, \mathcal{C}(v_2) :: \mathcal{C}(v_1) :: s, sto''_m\rangle\end{aligned}$$

The program counter in the last configuration must now point to IADD which by definition results in the transition

$$\begin{aligned}&\langle pc + |\mathcal{T}(E_1)| + |\mathcal{T}(E_2)|, \mathcal{C}(v_2) :: \mathcal{C}(v_1) :: s, sto''_m\rangle \\ &\qquad \Rightarrow \langle pc + |\mathcal{T}(E_1)| + |\mathcal{T}(E_2)| + 1, \mathcal{C}(v) :: s, sto''_m\rangle\end{aligned}$$

where $\mathcal{C}(v) = \mathcal{C}(v_1) + \mathcal{C}(v_2)$.

The cases $E_1 - E_2$, $E_1 * E_2$, E_1 / E_2, $E_1 \% E_2$ and $-E$: These are all similar to the case $E_1 + E_2$.

The case E_1&&E_2: If there is a transition, it must have been concluded with the rule [and-sc] or [and]. We start by splitting the proof into two cases depending on whether E_1 evaluates to F or to T. If E_1 evaluates to F, the rule [and-sc] must have been used:

$$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle}{\langle E_1\&\&E_2, e\rangle \rightarrow \langle \text{F}, e'\rangle} \quad \text{if } v_1 = \text{F}$$

The expression is translated to

$$\begin{aligned}\mathcal{T}(E_1\&\&E_2) = \mathcal{T}(E_1); &\text{IFEQ } \mathit{false}; \mathcal{T}(E_2); \text{IFEQ } \mathit{false}; \text{LDC } \mathcal{C}(\text{T}); \text{GOTO } \mathit{end};\\ &\mathit{false}:; \text{LDC } \mathcal{C}(\text{F}); \mathit{end}:\end{aligned}$$

which results in the following computation sequence

$$\begin{aligned}\langle pc, s, sto_m\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto'_m\rangle && \text{execution of } \mathcal{T}(E_1)\\ &\Rightarrow \langle \lambda(\mathit{false}), s, sto'_m\rangle && \text{execution of IFEQ } \mathit{false}\\ &\Rightarrow \langle \lambda(\mathit{false}) + 1, s, sto'_m\rangle && \text{execution of } \mathit{false}:\\ &\Rightarrow \langle \lambda(\mathit{false}) + 2, \mathcal{C}(\text{F}) :: s, sto'_m\rangle && \text{execution of LDC } \mathcal{C}(\text{F})\\ &\Rightarrow \langle \lambda(\mathit{false}) + 3, \mathcal{C}(\text{F}) :: s, sto'_m\rangle && \text{execution of } \mathit{end}:\end{aligned}$$

The existence of the computation sequence for E_1 follows from the premise and the inductive hypothesis since the derivation tree for E_1 is of height less than the derivation tree for the whole expression. Since $|\mathcal{T}(E_1\&\&E_2)| = \lambda(\mathit{false}) + 3$, this proves the case.

If E_1 evaluates to T, then the rule [and] must have been used:

$$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\&\&E_2, e\rangle \rightarrow \langle v_2, e''\rangle} \quad \text{if } v_1 = \text{T}$$

Then the following computation sequence must exist by the premises and the inductive hypothesis (we let $k = |\mathcal{T}(E_1)| + 1 + |\mathcal{T}(E_2)|$ for convenience):

$$\begin{aligned}\langle pc, s, sto_m\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto'_m\rangle && \text{execution of } \mathcal{T}(E_1)\\ &\Rightarrow \langle pc + |\mathcal{T}(E_1)| + 1, s, sto'_m\rangle && \text{execution of IFEQ } \mathit{false}\\ &\Rightarrow^* \langle pc + k, \mathcal{C}(v_2) :: s, sto''_m\rangle && \text{execution of } \mathcal{T}(E_2)\end{aligned}$$

The computation now proceeds depending on v_2. If $v_2 = \text{F}$, then the computation sequence is

$$
\begin{array}{lll}
\langle pc+k, \mathcal{C}(v_2) :: s, sto''_m \rangle & & \\
& \Rightarrow \langle \lambda(\mathit{false}), s, sto''_m \rangle & \text{execution of IFEQ } \mathit{false} \\
& \Rightarrow \langle \lambda(\mathit{false})+1, s, sto''_m \rangle & \text{execution of } \mathit{false}\text{:} \\
& \Rightarrow \langle \lambda(\mathit{false})+2, \mathcal{C}(\text{F}) :: s, sto''_m \rangle & \text{execution of LDC } \mathcal{C}(\text{F}) \\
& \Rightarrow \langle \lambda(\mathit{false})+3, \mathcal{C}(\text{F}) :: s, sto''_m \rangle & \text{execution of } \mathit{end}\text{:}
\end{array}
$$

If $v_2 = \text{T}$, then the computation sequence is

$$
\begin{array}{lll}
\langle pc+k, \mathcal{C}(v_2) :: s, sto''_m \rangle & & \\
& \Rightarrow \langle pc+k+1, s, sto''_m \rangle & \text{execution of IFEQ } \mathit{false} \\
& \Rightarrow \langle pc+k+2, \mathcal{C}(\text{T}) :: s, sto''_m \rangle & \text{execution of LDC } \mathcal{C}(\text{T}) \\
& \Rightarrow \langle \lambda(\mathit{end}), \mathcal{C}(\text{T}) :: s, sto''_m \rangle & \text{execution of GOTO } \mathit{end} \\
& \Rightarrow \langle \lambda(\mathit{end})+1, \mathcal{C}(\text{T}) :: s, sto''_m \rangle & \text{execution of } \mathit{end}\text{:}
\end{array}
$$

And $|\mathcal{T}(E_1\&\&E_2)| = \lambda(\mathit{end}) + 1$.

The case E_1 || E_2 and !E: Similar to the case $E_1\&\&E_2$.

The case E_1==E_2 for integers: If there is a transition, it must have been concluded with the rule [equals]:

$$
\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\texttt{==}E_2, e\rangle \rightarrow \langle v, e''\rangle} \quad \text{where } v = \begin{cases} \text{T} & \text{if } v_1 = v_2 \\ \text{F} & \text{else} \end{cases}
$$

Since the expression is translated to

$$
\begin{aligned}
\mathcal{T}(E_1\texttt{==}E_2) = \mathcal{T}(E_1); \mathcal{T}(E_2); &\text{IF_ICMPEQ } \mathit{true}; \text{LDC } \mathcal{C}(\text{F}); \text{GOTO } \mathit{end}; \\
&\mathit{true}\text{:}; \text{LDC } \mathcal{C}(\text{T}); \mathit{end}\text{:}
\end{aligned}
$$

the following computation sequences must exist by the premises and the inductive hypothesis (the derivation trees for E_1 and E_2 are of height less than the derivation tree for the whole expression):

$$
\begin{aligned}
\langle pc, s, sto_m \rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto'_m \rangle \\
&\Rightarrow^* \langle pc + |\mathcal{T}(E_1)| + |\mathcal{T}(E_2)|, \mathcal{C}(v_2) :: \mathcal{C}(v_1) :: s, sto''_m \rangle
\end{aligned}
$$

The computation now depends on whether $\mathcal{C}(v_1) = \mathcal{C}(v_2)$. Let for convenience $k = |\mathcal{T}(E_1)| + |\mathcal{T}(E_2)|$, then if this is the case, we have the computation sequence:

$$
\begin{array}{ll}
\langle pc + k, \mathcal{C}(v_2) :: \mathcal{C}(v_1) :: s, sto''_m \rangle & \\
\quad \Rightarrow \langle \lambda(\mathit{true}), s, sto''_m \rangle & \text{execution of IF_ICMPEQ } \mathit{true} \\
\quad \Rightarrow \langle \lambda(\mathit{true}) + 1, s, sto''_m \rangle & \text{execution of } \mathit{true}\text{:} \\
\quad \Rightarrow \langle \lambda(\mathit{true}) + 2, \mathcal{C}(\mathrm{T}) :: s, sto''_m \rangle & \text{execution of LDC } \mathcal{C}(\mathrm{T}) \\
\quad \Rightarrow \langle \lambda(\mathit{true}) + 3, \mathcal{C}(\mathrm{T}) :: s, sto''_m \rangle & \text{execution of } \mathit{end}\text{:}
\end{array}
$$

Since $|\mathcal{T}(E_1\texttt{==}E_2)| = \lambda(\mathit{true}) + 3$, this proves this case. If $\mathcal{C}(v_1) \neq \mathcal{C}(v_2)$ then we have the computation sequence

$$
\begin{array}{ll}
\langle pc + k, \mathcal{C}(v_2) :: \mathcal{C}(v_1) :: s, sto''_m \rangle & \\
\quad \Rightarrow \langle pc + k + 1, s, sto''_m \rangle & \text{execution of IF_ICMPEQ } \mathit{true} \\
\quad \Rightarrow \langle pc + k + 2, \mathcal{C}(\mathrm{F}) :: s, sto''_m \rangle & \text{execution of LDC } \mathcal{C}(\mathrm{F}) \\
\quad \Rightarrow \langle \lambda(\mathit{end}), \mathcal{C}(\mathrm{F}) :: s, sto''_m \rangle & \text{execution of GOTO } \mathit{end} \\
\quad \Rightarrow \langle \lambda(\mathit{end}) + 1, \mathcal{C}(\mathrm{F}) :: s, sto''_m \rangle & \text{execution of } \mathit{end}\text{:}
\end{array}
$$

Since $|\mathcal{T}(E_1\texttt{==}E_2)| = \lambda(\mathit{end}) + 1$, this concludes this case.

The cases E_1!=E_2, E_1<E_2, E_1>E_2, E_1<=E_2 and E_1>=E_2 for integers: Analogous to the case E_1==E_2.

The cases E_1==E_2, E_1!=E_2, E_1<E_2, E_1>E_2, E_1<=E_2 and E_1>=E_2 for floating point: Due to the irregularity of the JVM these are not quite the same as for integer, but still very similar. Hence we shall not present a proof.

The case $F(E_1, \ldots, E_n)$ If there is a transition, it must have been concluded with the rule [fcall-$\mathcal{R}$]. The rule itself is

$$
\frac{\langle E_1, e_1 \rangle \rightarrow \langle v_1, e_2 \rangle \quad \ldots \quad \langle E_n, e_n \rangle \rightarrow \langle v_n, e_{n+1} \rangle \quad \langle S, e'_{n+1} \rangle \rightarrow e_r^{\mathcal{R}}}{\langle F\,(\,E_1, \ldots, E_n\,)\,\texttt{;}, e_1 \rangle \rightarrow \langle v, (e_{F_1}, e_{F_G}, e_{V_1}, e_{V_G}, sto_r) \rangle}
$$

The expression is translated to

$$
\mathcal{T}(F(E_1, \ldots, E_n)) = \mathcal{T}(E_1); \ldots; \mathcal{T}(E_n); \text{INVOKESTATIC } F
$$

From the inductive hypothesis, the computation sequences

$$
\begin{array}{l}
\langle pc, s, sto_{m_1} \rangle \Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto_{m_2} \rangle \\
\qquad \vdots \\
\qquad \Rightarrow^* \langle pc + |\mathcal{T}(E_1)| + \ldots + |\mathcal{T}(E_n)|, \mathcal{C}(v_n) :: \ldots :: \mathcal{C}(v_1) :: s, sto_{m_{n+1}} \rangle
\end{array}
$$

must exist since the derivation trees for the expressions $E_1, \ldots, E_n$ have heights that are less than the height of the derivation tree for the whole expression. Now consider the transition $\langle S, e'_{n+1} \rangle \rightarrow e_r^{\mathcal{R}}$. Since the environment is $\mathcal{R}$-tagged, a **return** statement is evaluated somewhere inside S. Hence, S can be written as

$$\{S_1 \ldots \texttt{return}\, E_r\texttt{;} \ldots S_n\}$$

where the **return** statement may be inside a compound statement. Therefore S must have been translated to

$$\mathcal{T}(S_1); \ldots ; \mathcal{T}(E_r); \text{IRETURN}; \ldots ; \mathcal{T}(S_n)$$

so that the computation proceeds with (where $k = |\mathcal{T}(E_1)| + \ldots + |\mathcal{T}(E_n)|$)

$$\begin{aligned} \langle pc + k, \mathcal{C}(v_n) :: \ldots :: \mathcal{C}(v_1) :: s, sto_{m_{n+1}} \rangle & \\ &\Rightarrow \langle \varphi(F), (pc + k) :: s, sto'_{m_{n+1}} \rangle \\ &\Rightarrow^* \langle \varphi(F) + |\mathcal{T}(S_1); \ldots ; \mathcal{T}(E_r)|, \mathcal{C}(v) :: (pc + k) :: s, sto_{m_r} \rangle \\ &\Rightarrow \langle pc + k + 1, \mathcal{C}(v) :: s, sto_{m_r} \rangle \end{aligned}$$

where $sto'_{m_{n+1}} = sto_{m_{n+1}}[\nu(V_1) \mapsto \mathcal{C}(v_1), \ldots, \nu(V_n) \mapsto \mathcal{C}(v_n)]$. The execution of the middle computation sequence follows from the transition for S and the inductive hypothesis since the derivation tree for S must be of height less than the derivation tree for the whole function call expression.

The proof above was for the case where a function call occurs in an expression – the cases where a function call occurs as a statement are similar so we shall not prove them.

The case $V = E$**;**: If there is a transition, the transition rule [assign] must have been applied:

$$\frac{\langle E, e \rangle \rightarrow \langle v, (e_F, e_{F_G}, e_V, e_{V_G}, sto') \rangle}{\langle V\texttt{=}E\texttt{;}, e \rangle \rightarrow (e_F, e_{F_G}, e_V, e_{V_G}, sto'[l \mapsto v])} \quad \text{where } l = L_V(V, e_V, e_{V_G})$$

Now $\mathcal{T}(V = E\texttt{;}) = \mathcal{T}(E); \text{ISTORE}\ l$ where $l = \nu(V)$. The premise of the transition rule guarantees that a transition for E exists. Therefore from the inductive hypothesis (since the derivation tree for E is of height less than the derivation tree for the whole statement) the machine first goes through the computation sequence

$$\langle pc, s, sto_m \rangle \Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m \rangle.$$

Then the definition of ISTORE yields

$$\langle pc + |\mathcal{T}(E)|, v :: s, sto'_m \rangle \Rightarrow \langle pc + |\mathcal{T}(E)| + 1, s, sto'_m[l \mapsto \mathcal{C}(v)] \rangle.$$

Now from the inductive hypothesis, $sto' \stackrel{\mathcal{C}}{=} sto'_m$; hence $sto'[l \mapsto v] \stackrel{\mathcal{C}}{=} sto'_m[l \mapsto \mathcal{C}(v)]$ since $\mathcal{C}(sto'[l \mapsto v](l)) = \mathcal{C}(v) = sto'_m[l \mapsto \mathcal{C}(v)](l)$.

The case $++V$;: The rule [increment] is

$$\langle \texttt{++}V\texttt{;}, e\rangle \rightarrow (e_F, e_{F_G}, e_V, e_{V_G}, sto[l \mapsto v]) \quad \text{where } v = sto(l) + 1$$

which is translated to $\mathcal{T}(\texttt{++}V\texttt{;}) = \text{ILOAD } l; \text{LDC } 1; \text{IADD}; \text{ISTORE } l$ where $l = \nu(V)$. This gives the following computation sequence

$$\begin{aligned}
\langle pc, s, sto_m\rangle &\Rightarrow \langle pc+1, sto_m(l) :: s, sto_m\rangle && \text{execution of ILOAD } l\\
&\Rightarrow \langle pc+2, 1 :: sto_m(l) :: s, sto_m\rangle && \text{execution of LDC } 1\\
&\Rightarrow \langle pc+3, \mathcal{C}(v) :: s, sto_m\rangle && \text{execution of IADD}\\
&\Rightarrow \langle pc+4, s, sto_m[l \mapsto \mathcal{C}(v)]\rangle && \text{execution of ISTORE } l
\end{aligned}$$

where $\mathcal{C}(v) = sto_m(l) + 1$.

The case $--V$;: Analogous to the case $++V$;.

The case {$S_1, \ldots, S_n$}: If there is a transition, it must have been concluded with the rule [block]:

$$\frac{\langle S_1, e_1\rangle \rightarrow e_2 \quad \langle S_2, e_2\rangle \rightarrow e_3 \quad \ldots \quad \langle S_n, e_n\rangle \rightarrow e_{n+1}}{\langle \{S_1 S_2 \ldots S_n\}, e_1\rangle \rightarrow (e_{F_1}, e_{F_G}, e_{V_1}, e_{V_G}, sto_{n+1}).}$$

Since $\mathcal{T}(\{S_1, \ldots, S_n\}) = \mathcal{T}(S_1); \ldots; \mathcal{T}(S_n)$ and each of the statements $S_1, \ldots, S_n$ has a derivation tree with a height less than the height of the derivation tree for the whole block, we conclude by the inductive hypothesis that the following computation sequences exist:

$$\begin{aligned}
\langle pc, s, sto_{m_1}\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(S_1)|, s, sto_{m_2}\rangle\\
&\Rightarrow^* \langle pc + |\mathcal{T}(S_1)| + |\mathcal{T}(S_2)|, s, sto_{m_3}\rangle\\
&\ \ \vdots\\
&\Rightarrow^* \langle pc + |\mathcal{T}(S_1)| + \ldots + |\mathcal{T}(S_n)|, s, sto_{m_{n+1}}\rangle.
\end{aligned}$$

The case while(E)S: We first consider the case where E evaluates to F. If there is a transition, the rule [while-false]

$$\frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle}{\langle \texttt{while(}E\texttt{)}S, e\rangle \rightarrow e'} \quad \text{where } v = \text{F}$$

must have been used. The statement is translated to

$$\mathcal{T}(\texttt{while(}E\texttt{)}S) = \textit{start}{:}; \mathcal{T}(E); \text{IFEQ } \textit{done}; \mathcal{T}(S); \text{GOTO } \textit{start}; \textit{done}{:}$$

which gives the following computation sequence:

$$
\begin{aligned}
\langle pc, s, sto_m \rangle &\Rightarrow \langle pc + 1, s, sto_m \rangle && \text{execution of } \mathit{start}\text{:} \\
&\Rightarrow^* \langle pc + 1 + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m \rangle && \text{execution of } \mathcal{T}(E) \\
&\Rightarrow \langle \lambda(\mathit{done}), s, sto'_m \rangle && \text{execution of IFEQ } \mathit{done} \\
&\Rightarrow \langle \lambda(\mathit{done}) + 1, s, sto'_m \rangle && \text{execution of } \mathit{done}\text{:}
\end{aligned}
$$

Since $pc + |\mathcal{T}(\texttt{while(}E\texttt{)}S)| = \lambda(\mathit{done}) + 1$, this proves this case. The execution of $\mathcal{T}(E)$ follows from the premise of the [while-false] transition (which states that there is a transition and that it has the value $v = \text{F}$) and the inductive hypothesis since the derivation tree for E must be smaller than the tree for the entire statement.

For the case where E evaluates to T, if there is a transition, the rule [while-true]

$$
\frac{\langle E, e \rangle \to \langle v, e' \rangle \quad \langle S, e' \rangle \to e'' \quad \langle \texttt{while(}E\texttt{)}S, e'' \rangle \to e'''}{\langle \texttt{while(}E\texttt{)}S, e \rangle \to e'''} \quad \text{where } v = \text{T}
$$

must have been used. Here, the computation sequence is:

$$
\begin{aligned}
\langle pc, s, sto_m \rangle &\Rightarrow \langle pc + 1, s, sto_m \rangle && \text{execution of } \mathit{start}\text{:} \\
&\Rightarrow^* \langle pc + 1 + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m \rangle && \text{execution of } \mathcal{T}(E) \\
&\Rightarrow \langle pc + 1 + |\mathcal{T}(E)| + 1, s, sto'_m \rangle && \text{execution of IFEQ } \mathit{done} \\
&\Rightarrow^* \langle pc + 1 + |\mathcal{T}(E)| + 1 + |\mathcal{T}(S)|, s, sto''_m \rangle && \text{execution of } \mathcal{T}(S) \\
&\Rightarrow \langle pc, s, sto''_m \rangle && \text{execution of GOTO } \mathit{start}
\end{aligned}
$$

This completes one iteration of the loop. The execution of $\mathcal{T}(S)$ follows from the inductive hypothesis since the derivation tree for S must be smaller than the tree for the entire loop. Note that the machine is now in a state where it is about to execute $\mathcal{T}(\texttt{while(}E\texttt{)}S)$ with the store sto''_m. Then from the third premise and the inductive hypothesis, we can conclude that there must be a computation sequence which continues from the final tuple above and ends with the correct store:

$$
\langle pc, s, sto''_m \rangle \Rightarrow^* \langle pc + |\mathcal{T}(\texttt{while(}E\texttt{)}S)|, s, sto'''_m \rangle
$$

The case `if(`E`)`S_1 else S_2: We first consider the case where E evaluates to T. If a transition exists, it must have been concluded with the rule [if-else-true]:

$$
\frac{\langle E, e \rangle \to \langle v, e' \rangle \quad \langle S_1, e' \rangle \to e''}{\langle \texttt{if(}E\texttt{)}S_1\,\texttt{else}\,S_2, e \rangle \to e''} \quad \text{where } v = \text{T}
$$

The conditional is translated to

$$
\begin{aligned}
\mathcal{T}(\texttt{if(}E\texttt{)}S_1 \text{ else } S_2) = \mathcal{T}(E); \text{IFEQ } \mathit{else}; \mathcal{T}(S_1); \text{GOTO } \mathit{done}; \\
\mathit{else}\text{:}; \mathcal{T}(S_2); \mathit{done}\text{:}
\end{aligned}
$$

which gives the following computation sequence

$$\begin{aligned}
\langle pc, s, sto_m \rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m \rangle && \text{execution of } \mathcal{T}(E) \\
&\Rightarrow \langle pc + |\mathcal{T}(E)| + 1, s, sto'_m \rangle && \text{execution of IFEQ } \textit{else} \\
&\Rightarrow^* \langle pc + |\mathcal{T}(E)| + 1 + |\mathcal{T}(S_1)|, s, sto''_m \rangle && \text{execution of } \mathcal{T}(S_1) \\
&\Rightarrow \langle \lambda(\textit{done}), s, sto''_m \rangle && \text{execution of GOTO } \textit{done} \\
&\Rightarrow \langle \lambda(\textit{done}) + 1, s, sto''_m \rangle && \text{execution of } \textit{done}\text{:}
\end{aligned}$$

Since $pc + |\mathcal{T}(\texttt{if}\,(E)\,S_1 \text{ else } S_2)| = \lambda(\textit{done}) + 1$, this proves this case. The execution of $\mathcal{T}(E)$ follows from the first premise of the [if-else-true] transition (which states that there is a transition and that it has the value $v = \text{T}$) and the inductive hypothesis. The execution of $\mathcal{T}(S_1)$ follows from second premise and the inductive hypothesis since the derivation tree for S_1 is smaller than the derivation tree for the whole conditional.

For the case where E evaluates to F, the rule [if-else-false] must have been applied, and through the same reasoning as with the T case we get the following computation sequence:

$$\begin{aligned}
\langle pc, s, sto_m \rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m \rangle && \text{execution of } \mathcal{T}(E) \\
&\Rightarrow \langle \lambda(\textit{else}), s, sto'_m \rangle && \text{execution of IFEQ } \textit{else} \\
&\Rightarrow \langle \lambda(\textit{else}) + 1, s, sto'_m \rangle && \text{execution of } \textit{else}\text{:} \\
&\Rightarrow^* \langle \lambda(\textit{else}) + 1 + |\mathcal{T}(S_2)|, s, sto''_m \rangle && \text{execution of } \mathcal{T}(S_2) \\
&\Rightarrow \langle \lambda(\textit{else}) + 1 + |\mathcal{T}(S_2)| + 1, s, sto''_m \rangle && \text{execution of } \textit{done}\text{:}
\end{aligned}$$

Since $pc + |\mathcal{T}(\texttt{if}\,(E)\,S_1 \text{ else } S_2)| = |\lambda(\textit{else}) + 1 + |\mathcal{T}(S_2)| + 1|$, this proves the case.

The case `if(`E`)`S: Similar to the previous case.

The case `[const]` τ V`;`: The transition for the statement must be concluded with the rule [var-stat] that does not change the store. The statement is translated to $\mathcal{T}(\texttt{[const]}\ \tau\ V\texttt{;}) = \text{NOP}$ which results in the computation sequence

$$\langle pc, s, sto_m \rangle \Rightarrow \langle pc + 1, s, sto_m \rangle.$$

The case `[const]` τ $V = E$`;`: Since the variable environment is modeled implicitly with ν, this case is analogous to the case $V = E\texttt{;}$.

4.3.2 The other way

Next we shall prove that if the translated code terminates, then the untranslated code terminates too, and with the same result. Again, we will not prove this for the power, random and multiple choice operators.

Lemma 2 *For all expressions E, if a finite computation sequence for E exists*

$$\langle pc, s, sto_m \rangle \Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m \rangle$$

where $c[pc, \ldots, pc + |\mathcal{T}(E)| - 1] = \mathcal{T}(E)$, then there is a transition

$$\langle E, (e_F, e_{F_G}, e_V, e_{V_G}, sto)\rangle \rightarrow \langle v, (e_F, e_{F_G}, e_V, e_{V_G}, sto')\rangle$$

where $sto \overset{\mathcal{C}}{=} sto_m$ and $sto' \overset{\mathcal{C}}{=} sto'_m$.

And for all statements S, if a finite computation sequence for S exists

$$\langle pc, s, sto_m \rangle \Rightarrow^* \langle pc + |\mathcal{T}(S)|, s, sto'_m \rangle$$

where $c[pc, \ldots, pc + |\mathcal{T}(S)| - 1] = \mathcal{T}(S)$, then there is a transition

$$\langle S, (e_F, e_{F_G}, e_V, e_{V_G}, sto)\rangle \rightarrow (e_F, e_{F_G}, e'_V, e_{V_G}, sto')$$

where $sto \overset{\mathcal{C}}{=} sto_m$ and $sto' \overset{\mathcal{C}}{=} sto'_m$.

The proof is based on induction on the length of the computation sequences.

The case C: $\mathcal{T}(C) = \text{LDC } \mathcal{C}(v)$ where $v = \mathcal{V}(C)$ which results in the transition $\langle pc, s, sto_m \rangle \Rightarrow \langle pc + 1, \mathcal{C}(v) :: s, sto_m \rangle$. Using the rule [lit], we have $\langle C, e \rangle \rightarrow \langle v, e \rangle$ where $v = \mathcal{V}(C)$.

The case (E): $\mathcal{T}((E)) = \mathcal{T}(E); \text{NOP}$ so if there is a computation sequence for this expression, it must be

$$\begin{aligned} \langle pc, s, sto_m \rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m \rangle \\ &\Rightarrow \langle pc + |\mathcal{T}(E)| + 1, \mathcal{C}(v) :: s, sto'_m \rangle \end{aligned}$$

Since the length of the computation sequence for $\mathcal{T}(E)$ is less than the length of the computation sequence for the entire expression, we know from the inductive hypothesis that the transition

$$\langle E, e \rangle \rightarrow \langle v, e' \rangle$$

exists. Then we can apply the rule [paren]

$$\frac{\langle E, e \rangle \rightarrow \langle v, e' \rangle}{\langle (E), e \rangle \rightarrow \langle v, e' \rangle.}$$

The case V**:** $\mathcal{T}(V) = \text{ILOAD}\ l$ where $l = \nu(V)$ which results in the transition $\langle pc, s, sto_m \rangle \Rightarrow \langle pc + 1, sto_m(l) :: s, sto_m \rangle$. Using the rule [var], we have $\langle V, e \rangle \rightarrow \langle v, e \rangle$ where $v = sto(L_V(V, e_V, e_{V_G}))$. Since $\nu(V)$ gives the location of V the two values must be the same.

The case $E_1 + E_2$**:** $\mathcal{T}(E_1 + E_2) = \mathcal{T}(E_1); \mathcal{T}(E_2); \text{IADD}$. If there is a computation sequence for this expression, it must have brought the machine through the following steps

$$\begin{aligned}\langle pc, s, sto_m \rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto'_m \rangle \\ &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)| + |\mathcal{T}(E_2)|, \mathcal{C}(v_2) :: \mathcal{C}(v_1) :: s, sto''_m \rangle \\ &\Rightarrow \langle pc + |\mathcal{T}(E_1)| + |\mathcal{T}(E_2)| + 1, \mathcal{C}(v) :: s, sto''_m \rangle\end{aligned}$$

where $\mathcal{C}(v) = \mathcal{C}(v_1) + \mathcal{C}(v_2)$.

Using the inductive hypothesis for the immediate constituents (which have shorter computation sequences than the entire expression), the transitions

$$\langle E_1, (e_F, e_{F_G}, e_V, e_{V_G}, sto) \rangle \rightarrow \langle v_1, (e_F, e_{F_G}, e_V, e_{V_G}, sto') \rangle$$

and

$$\langle E_2, (e_F, e_{F_G}, e_V, e_{V_G}, sto') \rangle \rightarrow \langle v_2, (e_F, e_{F_G}, e_V, e_{V_G}, sto'') \rangle$$

must exist. Then we can use the rule [plus] to conclude

$$\frac{\langle E_1, e \rangle \rightarrow \langle v_1, e' \rangle \quad \langle E_2, e' \rangle \rightarrow \langle v_2, e'' \rangle}{\langle E_1 + E_2, e \rangle \rightarrow \langle v, e'' \rangle} \quad \text{where } v = v_1 + v_2$$

where $e'' = (e_F, e_{F_G}, e_V, e_{V_G}, sto'')$.

The cases $E_1 - E_2$**,** $E_1 * E_2$**,** E_1 / E_2**,** $E_1 \% E_2$ **and** $-E$**:** Again, these are all similar to the case $E_1 + E_2$.

The case $E_1 \&\& E_2$**:** The expression is translated to

$$\begin{aligned}\mathcal{T}(E_1 \&\& E_2) = \mathcal{T}(E_1); \text{IFEQ } \mathit{false}; \mathcal{T}(E_2); \text{IFEQ } \mathit{false}; \text{LDC } \mathcal{C}(\text{T}); \text{GOTO } \mathit{end}; \\ \mathit{false}{:}; \text{LDC } \mathcal{C}(\text{F}); \mathit{end}{:}\end{aligned}$$

If there is a computation sequence, then there must be a computation sequence for E_1, and possibly also one for E_2. We split the proof into two cases depending on

whether E_1 evaluates to F or to T. If E_1 evaluates to F, then the following computation sequence must exist

$$\begin{aligned}
\langle pc, s, sto_m\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto'_m\rangle && \text{execution of } \mathcal{T}(E_1)\\
&\Rightarrow \langle \lambda(\mathit{false}), s, sto'_m\rangle && \text{execution of IFEQ } \mathit{false}\\
&\Rightarrow \langle \lambda(\mathit{false}) + 1, s, sto'_m\rangle && \text{execution of } \mathit{false}\text{:}\\
&\Rightarrow \langle \lambda(\mathit{false}) + 2, \mathcal{C}(\mathrm{F}) :: s, sto'_m\rangle && \text{execution of LDC } \mathcal{C}(\mathrm{F})\\
&\Rightarrow \langle \lambda(\mathit{false}) + 3, \mathcal{C}(\mathrm{F}) :: s, sto'_m\rangle && \text{execution of } \mathit{end}\text{:}
\end{aligned}$$

From the inductive hypothesis, the transition

$$\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle$$

exists since the computation sequence for E_1 is shorter than the computation sequence for the whole expression. Then we can apply the rule [and-sc]:

$$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle}{\langle E_1 \&\& E_2, e\rangle \rightarrow \langle \mathrm{F}, e'\rangle} \quad \text{if } v_1 = \mathrm{F}.$$

If E_1 evaluates to T, then the computation sequence must be (where $k = |\mathcal{T}(E_1)| + 1 + |\mathcal{T}(E_2)|$):

$$\begin{aligned}
\langle pc, s, sto_m\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto'_m\rangle && \text{execution of } \mathcal{T}(E_1)\\
&\Rightarrow \langle pc + |\mathcal{T}(E_1)| + 1, s, sto'_m\rangle && \text{execution of IFEQ } \mathit{false}\\
&\Rightarrow^* \langle pc + k, \mathcal{C}(v_2) :: s, sto''_m\rangle && \text{execution of } \mathcal{T}(E_2)
\end{aligned}$$

From this, the transitions

$$\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle$$

and

$$\langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle$$

must exist by the inductive hypothesis. Hence, we use the rule [and] to conclude

$$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1 \&\& E_2, e\rangle \rightarrow \langle v_2, e''\rangle} \quad \text{if } v_1 = \mathrm{T}.$$

The rest of the proof amounts to showing that the computation sequences for the two cases $v_2 = \mathrm{F}$ and $v_2 = \mathrm{T}$ agree with the conclusion of [and]. This has already been done in the proof for Lemma 1 for this case.

The case E_1 || E_2 and !E: Again, similar to the case $E_1 \&\& E_2$.

The case E_1==E_2 for integers: The expression is translated to

$$\mathcal{T}(E_1\texttt{==}E_2) = \mathcal{T}(E_1);\mathcal{T}(E_2);\text{IF_ICMPEQ } \mathit{true};\text{LDC } \mathcal{C}(\text{F});\text{GOTO } \mathit{end};$$
$$\mathit{true}:;\text{LDC } \mathcal{C}(\text{T});\mathit{end}:$$

If there is a computation sequence, it must go through the steps

$$\begin{aligned}\langle pc, s, sto_m\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto'_m\rangle \\ &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)| + |\mathcal{T}(E_2)|, \mathcal{C}(v_2) :: \mathcal{C}(v_1) :: s, sto''_m\rangle\end{aligned}$$

From the inductive hypothesis and the fact that the computation sequences for $\mathcal{T}(E_1)$ and $\mathcal{T}(E_2)$ must be shorter than the computation sequence for the whole expression, the transitions

$$\langle E_1, (e_F, e_{F_G}, e_V, e_{V_G}, sto)\rangle \rightarrow \langle v_1, (e_F, e_{F_G}, e_V, e_{V_G}, sto')\rangle$$

and

$$\langle E_2, (e_F, e_{F_G}, e_V, e_{V_G}, sto')\rangle \rightarrow \langle v_2, (e_F, e_{F_G}, e_V, e_{V_G}, sto'')\rangle$$

must exist. Therefore, we can apply the rule [equals]:

$$\frac{\langle E_1, e\rangle \rightarrow \langle v_1, e'\rangle \quad \langle E_2, e'\rangle \rightarrow \langle v_2, e''\rangle}{\langle E_1\texttt{==}E_2, e\rangle \rightarrow \langle v, e''\rangle} \quad \text{where } v = \begin{cases} \text{T} & \text{if } v_1 = v_2 \\ \text{F} & \text{else} \end{cases}$$

The rest of the proof for this case amounts to showing that the results of the two computation sequences for $\mathcal{C}(v_1) = \mathcal{C}(v_2)$ and $\mathcal{C}(v_1) \neq \mathcal{C}(v_2)$ that remain agree with the result of the [equals] transition. This has already been done in the proof for Lemma 1 for this case.

The cases E_1!=E_2, E_1<E_2, E_1>E_2, E_1<=E_2 and E_1>=E_2 for integers: Analogous to the case E_1==E_2.

The cases E_1==E_2, E_1!=E_2, E_1<E_2, E_1>E_2, E_1<=E_2 and E_1>=E_2 for floating point: Again these are similar to the integer cases.

The case $F(E_1, \ldots, E_n)$ The expression is translated to

$$\mathcal{T}(F(E_1, \ldots, E_n)) = \mathcal{T}(E_1);\ldots;\mathcal{T}(E_n);\text{INVOKESTATIC } F$$

If there is a finite computation sequence for the expression, then the body $\mathcal{T}(S)$ of the function F must contain a translated **return** statement, i.e. it must look like

$$\mathcal{T}(S_1);\ldots;\mathcal{T}(E_r);\text{IRETURN};\ldots;\mathcal{T}(S_n)$$

Therefore, if there is a computation sequence, it must be (where $k = |\mathcal{T}(E_1)| + \ldots + |\mathcal{T}(E_n)|$ for convenience):

$$\begin{aligned}
\langle pc, s, sto_{m_1}\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E_1)|, \mathcal{C}(v_1) :: s, sto_{m_2}\rangle \\
&\vdots \\
&\Rightarrow^* \langle pc + k, \mathcal{C}(v_n) :: \ldots :: \mathcal{C}(v_1) :: s, sto_{m_{n+1}}\rangle \\
&\Rightarrow \langle \varphi(F), (pc + k) :: s, sto'_{m_{n+1}}\rangle \\
&\Rightarrow^* \langle \varphi(F) + |\mathcal{T}(S_1); \ldots ; \mathcal{T}(E_r)|, \mathcal{C}(v) :: (pc + k) :: s, sto_{m_r}\rangle \\
&\Rightarrow \langle pc + k + 1, \mathcal{C}(v) :: s, sto_{m_r}\rangle
\end{aligned}$$

where $sto'_{m_{n+1}} = sto_{m_{n+1}}[\nu(V_1) \mapsto \mathcal{C}(v_1), \ldots, \nu(V_n) \mapsto \mathcal{C}(v_n)]$. Hence, the transitions

$$\begin{gathered}
\langle E_1, e_1\rangle \to \langle v_1, e_2\rangle \\
\vdots \\
\langle E_n, e_n\rangle \to \langle v_n, e_{n+1}\rangle
\end{gathered}$$

must exist by the inductive hypothesis. Since S is

$$\{S_1 \; \ldots \; \texttt{return}\, E_r\texttt{;} \; \ldots \; S_n\}$$

the derivation tree must look like (with the **return** statement inside an arbitrarily complex compound)

$$\frac{S_1 \quad \cdots \quad \dfrac{\dfrac{\dfrac{\langle E_r, e_t\rangle \to \langle v, e_r\rangle}{\langle \texttt{return}\, E_r\texttt{;}, e_t\rangle \to \hat{e}_r^{\mathcal{R}}} \quad \cdots}{\cdots}}{\cdots} \quad \cdots \quad S_n}{\langle S, e'_{n+1}\rangle \to \hat{e}_r^{\mathcal{R}}}$$

where $\hat{e}_r$ is e_r except the store has been augmented with the binding retval $\mapsto v$. The transition from S_1 towards the **return** statement and the transition for E_r follows from the computation sequence above and the inductive hypothesis. Since the **return** statement returns an $\mathcal{R}$-tagged environment, the rest of the transitions towards S_n will simply hand over the $e_r^{\mathcal{R}}$ environment, thus completing the transition for S.

Therefore, we can apply the rule [fcall-$\mathcal{R}$]

$$\frac{\langle E_1, e_1\rangle \to \langle v_1, e_2\rangle \quad \ldots \quad \langle E_n, e_n\rangle \to \langle v_n, e_{n+1}\rangle \quad \langle S, e'_{n+1}\rangle \to \hat{e}_r^{\mathcal{R}}}{\langle F\,(\,E_1, \ldots, E_n), e_1\rangle \to \langle v, (e_{F_1}, e_{F_G}, e_{V_1}, e_{V_G}, \widehat{sto}_r)\rangle}$$

where $v = \widehat{sto}_r(\text{retval})$.

Again, we shall not prove the case where the function call occurs as a statement.

The case $V = E$;: $\mathcal{T}(V = E;) = \mathcal{T}(E);$ ISTORE l where $l = \nu(V)$. If a computation sequence exists, it must bring the machine through the following steps:

$$\begin{aligned}\langle pc, s, sto_m \rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(v) :: s, sto'_m \rangle \\ &\Rightarrow \langle pc + |\mathcal{T}(E)| + 1, s, sto'_m[l \mapsto \mathcal{C}(v)] \rangle\end{aligned}$$

Hence, from the inductive hypothesis the transition

$$\langle E, e \rangle \to \langle v, (e_F, e_{F_G}, e_V, e_{V_G}, sto') \rangle$$

must exist since the computation sequence for E is shorter than the computation sequence for the whole statement. Therefore we can apply the rule [assign] and conclude

$$\frac{\langle E, e \rangle \to \langle v, (e_F, e_{F_G}, e_V, e_{V_G}, sto') \rangle}{\langle V{=}E;, e \rangle \to (e_F, e_{F_G}, e_V, e_{V_G}, sto'[l \mapsto v])} \quad \text{where } l = L_V(V, e_V, e_{V_G}).$$

The case $++V$;: $\mathcal{T}(++V;) =$ ILOAD l; LDC 1; IADD; ISTORE l where $l = \nu(V)$. This gives the computation sequence

$$\begin{aligned}\langle pc, s, sto_m \rangle &\Rightarrow \langle pc + 1, sto_m(l) :: s, sto_m \rangle && \text{execution of ILOAD } l \\ &\Rightarrow \langle pc + 2, 1 :: sto_m(l) :: s, sto_m \rangle && \text{execution of LDC 1} \\ &\Rightarrow \langle pc + 3, \mathcal{C}(v) :: s, sto_m \rangle && \text{execution of IADD} \\ &\Rightarrow \langle pc + 4, s, sto_m[l \mapsto \mathcal{C}(v)] \rangle && \text{execution of ISTORE } l\end{aligned}$$

where $\mathcal{C}(v) = sto_m(l) + 1$. Using the rule [increment] we can conclude

$$\langle ++V;, e \rangle \to (e_F, e_{F_G}, e_V, e_{V_G}, sto[l \mapsto v]) \quad \text{where } v = sto(l) + 1.$$

The case $--V$;: Again, analogous to the case $++V;$.

The case $\{S_1, \ldots, S_n\}$: $\mathcal{T}(\{S_1, \ldots, S_n\}) = \mathcal{T}(S_1); \ldots; \mathcal{T}(S_n)$. If a computation sequence exists, it must be on the form

$$\begin{aligned}\langle pc, s, sto_{m_1} \rangle &\Rightarrow^* \langle pc + |\mathcal{T}(S_1)|, s, sto_{m_2} \rangle \\ &\Rightarrow^* \langle pc + |\mathcal{T}(S_1)| + |\mathcal{T}(S_2)|, s, sto_{m_3} \rangle \\ &\;\;\vdots \\ &\Rightarrow^* \langle pc + |\mathcal{T}(S_1)| + \ldots + |\mathcal{T}(S_n)|, s, sto_{m_{n+1}} \rangle.\end{aligned}$$

Since each of the computation sequences for the statements is shorter than the entire computation sequence, we know from the inductive hypothesis that the transitions

$$\langle S_1, e_1 \rangle \rightarrow e_2$$
$$\langle S_2, e_2 \rangle \rightarrow e_3$$
$$\vdots$$
$$\langle S_n, e_n \rangle \rightarrow e_{n+1}$$

exist. Hence, the rule [block] can be used:

$$\frac{\langle S_1, e_1 \rangle \rightarrow e_2 \quad \langle S_2, e_2 \rangle \rightarrow e_3 \quad \ldots \quad \langle S_n, e_n \rangle \rightarrow e_{n+1}}{\langle \{ S_1 S_2 \ldots S_n \}, e_1 \rangle \rightarrow (e_{F_1}, e_{F_G}, e_{V_1}, e_{V_G}, sto_{n+1}).}$$

The case `while(`E`)`S: The loop is translated to

$$\mathcal{T}(\texttt{while(}E\texttt{)}S) = \textit{start}{:}; \mathcal{T}(E); \text{IFEQ } \textit{done}; \mathcal{T}(S); \text{GOTO } \textit{start}; \textit{done}{:}$$

We first consider the case where E evaluates to F. The computation sequence is then

$$\begin{array}{lll} \langle pc, s, sto_m \rangle & \Rightarrow \langle pc+1, s, sto_m \rangle & \text{execution of } \textit{start}: \\ & \Rightarrow^* \langle pc+1+|\mathcal{T}(E)|, \mathcal{C}(\text{F}) :: s, sto'_m \rangle & \text{execution of } \mathcal{T}(E) \\ & \Rightarrow \langle \lambda(\textit{done}), s, sto'_m \rangle & \text{execution of IFEQ } \textit{done} \\ & \Rightarrow \langle \lambda(\textit{done})+1, s, sto'_m \rangle & \text{execution of } \textit{done}: \end{array}$$

From the inductive hypothesis the transition

$$\langle E, e \rangle \rightarrow \langle v, e' \rangle$$

must exist and v must be F. Then the rule [while-false] can be applied:

$$\frac{\langle E, e \rangle \rightarrow \langle v, e' \rangle}{\langle \texttt{while(}E\texttt{)}S, e \rangle \rightarrow e'} \quad \text{where } v = \text{F}$$

Since the final store is correct (from the evaluation of E), this proves this case.

For the case where E evaluates to T, if there is computation sequence, it can be written as:

$$\begin{array}{lll} \langle pc, s, sto_m \rangle & \Rightarrow \langle pc+1, s, sto_m \rangle & \text{execution of } \textit{start}: \\ & \Rightarrow^* \langle pc+1+|\mathcal{T}(E)|, \mathcal{C}(\text{T}) :: s, sto'_m \rangle & \text{execution of } \mathcal{T}(E) \\ & \Rightarrow \langle pc+1+|\mathcal{T}(E)|+1, s, sto'_m \rangle & \text{execution of IFEQ } \textit{done} \\ & \Rightarrow^* \langle pc+1+|\mathcal{T}(E)|+1+|\mathcal{T}(S)|, s, sto''_m \rangle & \text{execution of } \mathcal{T}(S) \\ & \Rightarrow \langle pc, s, sto''_m \rangle & \text{execution of GOTO } \textit{start} \\ & \Rightarrow^* \langle pc+|\mathcal{T}(\texttt{while(}E\texttt{)}S)|, s, sto'''_m \rangle & \text{execution of the rest of the loop} \end{array}$$

From the inductive hypothesis the transition

$$\langle E, e\rangle \rightarrow \langle v, e'\rangle$$

must exist and v must be T, and the transition

$$\langle S, e'\rangle \rightarrow e''$$

must exists since the computation sequence for S is shorter than the computation sequence for the whole loop. Likewise, since the computation sequence for the rest of the loop above is shorter than the computation sequence for the whole loop, there is a transition that starts with store sto'' and ends with sto''':

$$\langle \texttt{while(}E\texttt{)}S, e''\rangle \rightarrow e'''$$

Hence, we can apply the rule [while-true] to conclude that there is a transition and that it ends with the correct store:

$$\frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle \quad \langle S, e'\rangle \rightarrow e'' \quad \langle \texttt{while(}E\texttt{)}S, e''\rangle \rightarrow e'''}{\langle \texttt{while(}E\texttt{)}S, e\rangle \rightarrow e'''} \quad \text{where } v = \mathrm{T}$$

The case $\texttt{if(}E\texttt{)}S_1$ else S_2: The conditional is translated to

$$\begin{aligned}\mathcal{T}(\texttt{if(}E\texttt{)}S_1 \text{ else } S_2) = \mathcal{T}(E); \text{IFEQ } \mathit{else}; \mathcal{T}(S_1); \text{GOTO } \mathit{done};\\ \mathit{else}{:}; \mathcal{T}(S_2); \mathit{done}{:}\end{aligned}$$

We first consider the case where E evaluates to T. If a computation sequence exists, it must be

$$\begin{aligned}\langle pc, s, sto_m\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(\mathrm{T}) :: s, sto'_m\rangle && \text{execution of } \mathcal{T}(E)\\ &\Rightarrow \langle pc + |\mathcal{T}(E)| + 1, s, sto'_m\rangle && \text{execution of IFEQ } \mathit{else}\\ &\Rightarrow^* \langle pc + |\mathcal{T}(E)| + 1 + |\mathcal{T}(S_1)|, s, sto''_m\rangle && \text{execution of } \mathcal{T}(S_1)\\ &\Rightarrow \langle \lambda(\mathit{done}), s, sto''_m\rangle && \text{execution of GOTO } \mathit{done}\\ &\Rightarrow \langle \lambda(\mathit{done}) + 1, s, sto''_m\rangle && \text{execution of } \mathit{done}{:}\end{aligned}$$

From the inductive hypothesis the transition

$$\langle E, e\rangle \rightarrow \langle v, e'\rangle$$

must exist (and v must be T), and from the inductive hypothesis the transition

$$\langle S_1, e'\rangle \rightarrow e''$$

must also exist since the computation sequence for S_1 is shorter than the computation sequence for the whole conditional.

Hence, the rule [if-else-true] can be applied and gives

$$\frac{\langle E, e\rangle \rightarrow \langle v, e'\rangle \quad \langle S_1, e'\rangle \rightarrow e''}{\langle \texttt{if(}E\texttt{)}\,S_1\,\texttt{else}\,S_2, e\rangle \rightarrow e''} \quad \text{where } v = \text{T}.$$

For the case where E evaluates to F, if the computation sequence exists, it must be

$$\begin{aligned}
\langle pc, s, sto_m\rangle &\Rightarrow^* \langle pc + |\mathcal{T}(E)|, \mathcal{C}(\text{F}) :: s, sto'_m\rangle && \text{execution of } \mathcal{T}(E)\\
&\Rightarrow \langle \lambda(else), s, sto'_m\rangle && \text{execution of IFEQ } else\\
&\Rightarrow \langle \lambda(else) + 1, s, sto'_m\rangle && \text{execution of } else\text{:}\\
&\Rightarrow^* \langle \lambda(else) + 1 + |\mathcal{T}(S_2)|, s, sto''_m\rangle && \text{execution of } \mathcal{T}(S_2)\\
&\Rightarrow \langle \lambda(else) + 1 + |\mathcal{T}(S_2)| + 1, s, sto''_m\rangle && \text{execution of } done\text{:}
\end{aligned}$$

Through the same reasoning as the T case, transitions for E and S_2 must exist and we can apply the rule [if-else-false].

The case `if(`E`)`S: Again similar to the previous case.

The case `[const]` τ V`;`: The statement is translated to $\mathcal{T}(\texttt{[const]}\ \tau\ V\texttt{;}) =$ NOP which results in the computation sequence

$$\langle pc, s, sto_m\rangle \Rightarrow \langle pc + 1, s, sto_m\rangle.$$

Since we can apply the rule [var-stat] directly (because it is an axiom) without changing the store, this proves the case.

The case `[const]` τ $V = E$`;`: Again, since the variable environment is modeled implicitly with ν, this case is analogous to the case $V = E$`;`.

4.3.3 Correctness

From Lemma 1 and 2 the correctness of the translation follows immediately:

Theorem 1 *For all expressions E, E is semantically equivalent to $\mathcal{T}(E)$, and for all statements S, S is semantically equivalent to $\mathcal{T}(S)$.*

Chapter 5

Implementation

This chapter describes the implementation of a processor for the language defined in the preceding chapters.

5.1 Architecture

As explained in Chapter 1, a source program consists of a number of text files containing strings from our language. From a user's point of view, running a simulation amounts to initiating a program that processes these files, outputting the values of the watched identifiers for each iteration.

Internally, however, to enhance the modularity of the code the processing is divided into a number of steps where the intermediate representation is an abstract syntax tree. A parsing and a contextual analysis phase is definitely needed, but after these phases there are two fundamentally different possibilities: either interpret the simulation directly from the abstract syntax tree, or translate the tree into a program in another language and continue processing that program.

We have chosen the translation approach, translating the high-level language into JVM assembler, which makes our language processor a compiler. One advantage of this approach is that the simulations will benefit from the speed optimisations which have been put into the JVM implementations.

Thus we structure the compiler into four internal components and one external, as shown in Figure 5.1.

- The syntactic analysis phase is concerned with parsing the source text files and converting them to the internal abstract syntax tree.
- The contextual analysis is concerned with binding identifiers to the various entities in the program and checking that the language rules that cannot be

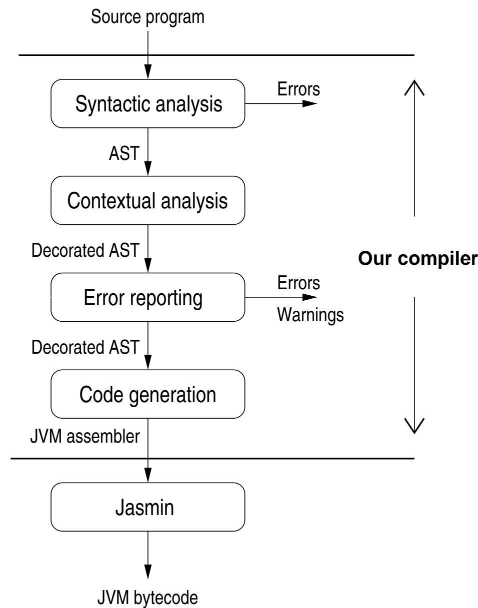

Figure 5.1: The various phases in the compiler.

specified with the EBNF grammar are adhered to.

- The reporting of errors detected in the contextual phase is decoupled from the actual analysis by inserting an extra phase that checks the decorations of the abstract syntax tree and reports any error marked by the contextual phase. A benefit of this is that all errors, including errors which can only be detected at the end of the analysis (e.g. when definitions of declared identifiers are missing), are reported in the order of their appearance in the source.

- When the source code is known to be completely free of errors, the code generation phase operates on the decorated abstract syntax tree, producing Java assembler output.

- Finally, the `jasmin` program translates the assembler code to binary JVM code which is ready to be executed by a JVM.

We proceed by describing each of the internal phases in turn.

5.2 Syntactic analysis

5.2.1 `rdp`

To avoid having to construct a scanner and a parser from scratch we have used the compiler compiler `rdp` [3] which takes as input a grammar written in an EBNF variant decorated with C or C++ statements and outputs a combined scanner and parser in C with the C and C++ statements embedded. The C++ support is essential since the rest of our compiler is written in C++.

With `rdp`, scanning and parsing is interleaved. The scanner is called from the parser whenever it needs a new token. If a syntactic error in the source program is detected, the `rdp`-generated parser formulates and prints an error message. Thus, when the syntactical analysis phase is over, we can be sure that the source program is written in the language defined by the EBNF grammar.

The parser generated by `rdp` uses a recursive-descent technique for parsing the source program which means that the input grammar must be an LL(1) grammar. This in turn means that it must have the properties [3]:

1. If $X ::= Y_1 \,|Y_2\,| \ldots |Y_n$ then $FIRST(Y_i) \cap FIRST(Y_j) = \emptyset$ for all Y_i, Y_j where $1 \leq i \leq n$, $1 \leq j \leq n$, $i \neq j$; that is if the rule for a non-terminal has alternatives then the first sets of these must be disjoint.

2. If $X ::= \ldots |Y| \ldots$ and $X \Rightarrow^{\star} \epsilon$ then $FIRST(Y) \cap FOLLOW(X) = \epsilon$. If a non-terminal X generates the empty string then the first sets of the alternatives in its rule must be disjoint from the follow set of X, except for the empty string ϵ.

The first set of X, written $FIRST(X)$, is the set of tokens that can start a string generated by X. The follow set, written $FOLLOW(X)$, is the set of tokens that can follow X in its context ([3] has a formal definition).

5.2.2 Transforming the grammar

Unfortunately, the grammar from Chapter 2 is not an LL(1) grammar. One problem is the *statement* rule from Section 2.2.1. The first sets are:

- S_1 = FIRST({ (*statement*)* }) = {{}.
- S_2 = FIRST(*variable-definition*) = {`const`, `int`, `float`, `bool`, *identifier*}.
- S_3 = FIRST(`while` `(` *expression* `)` *statement*) = {`while`}.
- S_4 = FIRST(`if` `(` *expression* `)` *statement* [`else` *statement*]) = {`if`}.

- S_5 = FIRST(*identifier* = *expression* **;**) = {*identifier*}.
- S_6 = FIRST(*function-call* **;**) = {*identifier*}.
- S_7 = FIRST(++ *identifier* **;**) = {++}.
- S_8 = FIRST(-- *identifier* **;**) = {--}.
- S_9 = FIRST(`return` *expression* **;**) = {`return`}.

Property 1 is not fulfilled because $S_2 \cap S_5 \cap S_6 = \{identifier\}$. The affected alternatives in the *statement* rule must be left factorized to get rid of the joint token, which also involves changing other rules:

typename	::=	`int` | `float` | `bool`
variable-definition	::=	[= expression] (**,** identifier [= expression])* **;**
function-call	::=	([expression (**,** expression)*])
statement	::=	{ (statement)* }
		| [`const`] typename identifier variable-definition
		| `while` (expression) statement
		| `if` (expression) statement [`else` statement]
		| identifier (= expression | function-call) **;**
		| ++ identifier **;** | -- identifier **;**
		| `return` expression **;**

Of course these changes must in turn be propagated to the rest of the rules to ensure that the grammar generates the same language, and then all rules must be checked for joint first sets again. The final result of a number of these iterations – the input grammar to `rdp` – is shown in Appendix A.

One problem remains, though. The `if-else` statement in our input grammar does not have the second property of a LL(1) grammar, because

$$FIRST([\texttt{else}\ statement]) \cap FOLLOW([\texttt{else}\ statement]) = \texttt{else}$$

This is a classic problem called a *dangling else* which `rdp` can compensate for. To appreciate why this is a problem, consider the derivation of the string `if (true) if (false) x=0; else x=1;`:

`if` (*expression*) *statement* [`else` *statement*]

⇒ `if (true) if` (*expression*) *statement* [`else` *statement*] [`else` *statement*]

⇒ `if (true) if (false) x=0;` [`else` *statement*] [`else` *statement*]

For the final step, either of `else` clauses can be matched with the `else x=1;`, and the decision affects the semantics of the string (either `x` is assigned to 1 or `x` is unchanged). With `rdp`, the innermost `else` is matched, which also is the case in other languages such as C, C++, Pascal and Java.

5.2.3 Representation of the abstract syntax

The abstract syntax tree is modeled as classes derived from the syntax in Chapter 2, cf. Figures 5.2–5.5. Only those aggregations that are not one-to-one are connected with edges, for the others the aggregated object (starting with a capital letter) is written as a member field.

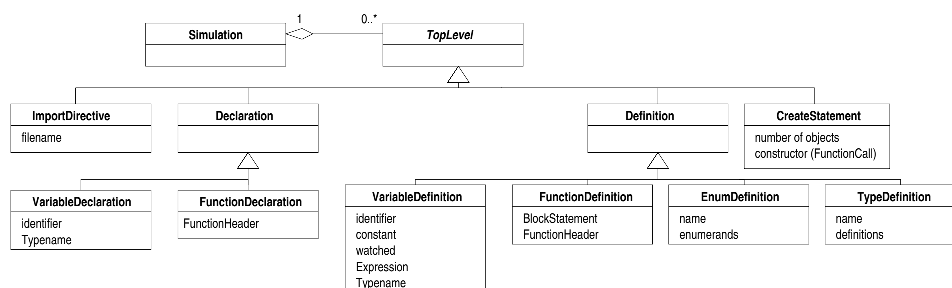

Figure 5.2: The toplevel node classes.

The SIMULATION class represents the whole source program and has a number of TOPLEVEL objects as shown in Figure 5.2. The TOPLEVEL class represents a common class for the alternatives in the *simulation* rule, and is further divided into classes for import directives, create statements and declarations and definitions, which in turn are subdivided and so forth.

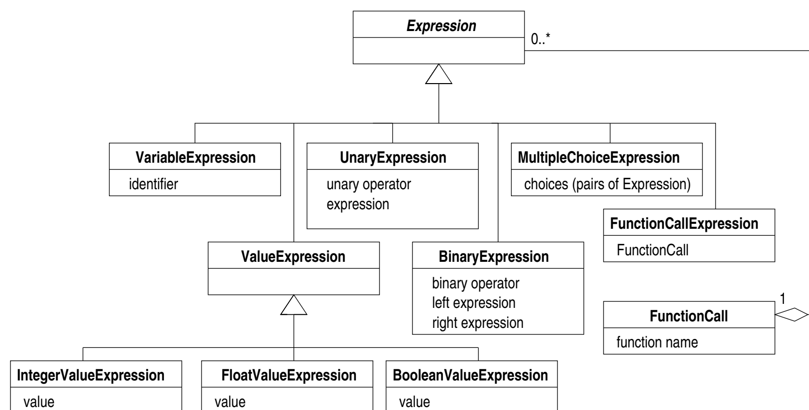

Figure 5.3: Expression classes.

For instance, for the *statement* rule, the class hierarchy is constructed by creating a base class, STATEMENT, and inheriting a class from that for each alternative in the rule since they are all a sort of statement:

statement ::=	
{ (statement)* }	**block**
| variable-definition	**variable definition**
| `while` (expression) statement	**while**
| `if` (expression) statement [`else` statement]	**if**
| identifier = expression **;**	**assignment**
| function-call **;**	**function call**
| ++ identifier **;** | -- identifier **;**	**increment** and **decrement**
| `return` expression **;**	**return**

The **block** alternative references the *statement* rule zero or a number of times, thus BLOCKSTATEMENT has zero or more STATEMENT objects. The **variable definition** is a statement of a variable definition, thus VARIABLEDEFINITIONSTATEMENT has a VARIABLEDEFINITION. The **while** alternate has a condition and a body, hence WHILESTATEMENT has an EXPRESSION and a STATEMENT. The IFSTATEMENT has an EXPRESSION and a then and possibly an else STATEMENT, because the **if** statement has a condition and a statement for when the condition is true and false. The ASSIGNMENTSTATEMENT has an identifier and an EXPRESSION, because an identifier is assigned to an expression in **assignment**. The FUNCTIONCALLSTATEMENT – **function call** – has a FUNCTIONCALL. The INCREMENTSTATEMENT and DECREMENTSTATEMENT both have an identifier, as mandated by **increment** and **decrement**. The RETURNSTATEMENT has an EXPRESSION to evaluate for the **return**.

The result of this construction is the segment of the class diagram illustrated in Figure 5.4.

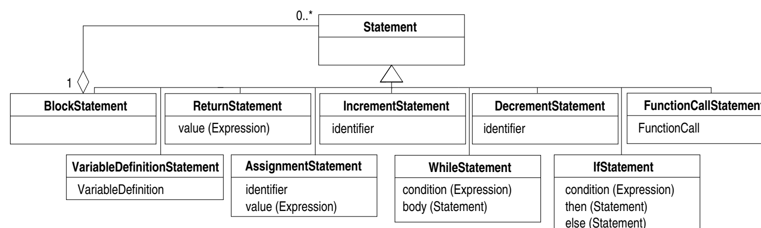

Figure 5.4: Statement classes.

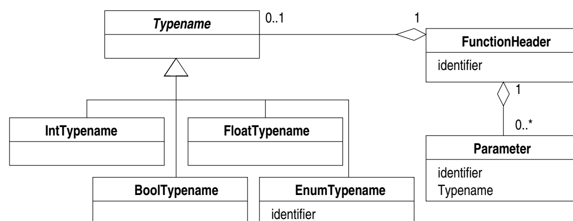

Figure 5.5: The typename and function header classes. These are aggregated by the other classes.

5.2.4 Creating the abstract syntax tree

The abstract syntax tree is built by instantiating objects from the class diagram during the parsing when their syntactical counterparts are encountered. The aggregation connections are then established as the analysis proceeds.

The object instantiation and linking is embedded in `rdp`'s semantic actions which are driven directly by the syntax. For instance, consider the left factorization alternative of the identifier token in *statement* from Section 5.2.2:

```
statement ::= ID ('=' expression | function-call) ';'
            | ... .
```

The semantic actions are defined within a pair of `[*` and `*]` and references to terminals are defined with a trailing colon and the name of the reference:

```
statement ::= ID:id ('=' exp1
  [*
   cstat = new AssignmentStatement(id, cexpr);
   cstat->line = LN;
  *]
                          |
  [*
   cfuncall = new FunctionCall(id);
   cfuncall->line = LN;
   cfuncalls.push(cfuncall);
  *]
                             function_call
  [*
   cstat = new FunctionCallStatement(cfuncalls.top());
   cstat->line = LN;
   cfuncalls.pop();
  *]
                                                  ) ';'
            | ... .
```

Line 1 adds a reference `id` to the identifier and line 2–5 describes the semantic actions for an assignment statement. The first action instantiates an ASSIGNMENTSTATEMENT with the identifier and an EXPRESSION, which is the resulting object from parsing `exp1` and executing its semantic actions. The second action, in line 4, stores the current line number of the source program which caused the object to be instantiated. This is done to facilitate helpful error reports during the error reporting fase. Line 7–11 describes the semantic actions for a function call statement. The first action instantiates a FUNCTIONCALL and the second action stores the current line number in that object. Line 10 pushes the instantiated object onto a stack for later reference. Line 13–17 instantiates a FUNCTIONCALLSTATEMENT with the FUNCTIONCALL as argument, by taking the top of the stack, to establish the aggregation. Lastly the stack is popped to clean things up a bit and ensure that no wrong aggregations are established.

Semantic actions must be defined for every syntactical construct to build the entire abstract syntax tree during parsing.

5.3 Contextual analysis

This phase is implemented through the subclass DECORATEVISITOR of the abstract VISITOR class. The VISITOR class represents the *visitor* pattern [1; 11] and facilitates a common way of visiting each particular type of node in the abstract syntax tree, while avoiding that the contextual analysis ends up as one giant switch statement.

5.3.1 Binding identifi ers

The identifiers in the abstract syntax tree need to be linked to their definitions so that the code generation phase can allocate space and generate code for accessing their contents. Hence, when visiting the tree, it is necessary to keep track of which identifiers are currently available when crossing scope boundaries, registering new identifiers when definitions (and declarations, but we treat them separately in Section 5.3.4) are encountered and utilising the lookup tables when identifiers occur elsewhere.

Scope levels

The block structure of our language is realised with a stack of mappings for identifiers. Each time a scope is entered, a new mapping is placed on the top of the stack, and each time the scope is closed again, the mapping is popped off. Searching for

an identifier then amounts to searching the mappings from the top of the stack. The first, global mapping is created when the contextual analysis starts. For example:

```
int x;          // ([x])
int y;          // ([x, y])

void f()
{               // ([], [x, y])
  int z;        // ([z], [x, y])
  {             // ([], [z], [x, y])
    int x;      // ([x], [z], [x, y])
  }             // ([z], [x, y])
}               // ([x, y])
```

A separate stack of mappings is maintained for each of the different sorts of identifiers, variables, functions and **enum**s. Only one mapping is used for **type**s, though, since they are always defined at the global level.

The mappings are from identifier strings to pointers to the definition of the identifier in the abstract syntax tree, e.g. to a VariableDefinition for a variable. A standard C++ `map` is used for the implementation, which guarantees logarithmic lookup and insertion complexity.

The identification tables are implemented with the IdentificationTable class. Its constructor creates the global scope mapping in the stack for variables, functions and enums by invoking the *open_scope()* function which is also invoked for each block encountered during the node visiting. The methods *retrieve(...)* and *enter(...)* retrieve and enter definitions from and into the corresponding map.

Entering identifiers

When registering an identifier, *enter* checks whether the identifier is already present in the local scope, as redefinition of identifiers in the same scope is not allowed. If the identifier is present, the definition is marked by setting *err_already_defined* to **true**, thus indicating to the error detection phase that this should be reported as an error. The snippet below is the visitor method for variable definitions.

```
void DecorateVisitor::visit(VariableDefinition *vd)
{
  if (ident_table.enter(vd->identifier, vd))
    vd->err_already_defined = true;
  ...
}
```

Binding identifi ers

When an identifier is encountered, its definition is looked up. If it is not present in the local scope, the containing scope is instead considered and so forth until the global scope is reached. If the lookup fails, the examined node is marked as containing an error. Else a link to the the found definition is established by setting a pointer in the node.

The snippet below is from the assignment statement.

```
void DecorateVisitor::visit(AssignmentStatement *as)
{
  ...
  VariableDefinition *vd = lookup_ident(as->identifier);

  if (vd != 0){
    t = vd->type;

    if (vd->constant)
      as->err_declared_as_constant = true;
  }
  ...
  // type checking
  ...
  // binding of occurrence
  if (vd != 0)
    as->occ = vd;
  ...
}
```

If the variable assigned to is constant, the assignment statement is obviously also erratic.

5.3.2 Type checking

Type checking a unary expression is straightforward, as the type check either fails or succeeds. Type checking a binary expression, a multiple choice expression or a statement, however, results in one of these three cases:

- A type match. Everything is fine.
- A major type mismatch. The error is reported in the reporting phase.
- A minor type mismatch which causes an implicit type conversion by inserting a call to either intify or fbatify with the expression as the parameter.

For example, type checking a variable definition is done by visiting its type and expression, and checking that the type of the expression match the type of the definition:

```
void DecorateVisitor::visit(VariableDefinition *vd)
{
  ...
  vd->type->visit(this);
  if (vd->expression) {

    vd->expression->visit(this);

    switch (check_type_match(vd->type, vd->expression->type))
    {
    case TYPE_MATCH:
      break;
    case MAJOR_TYPE_MISMATCH:
      vd->type = 0;
      break;
    case MINOR_TYPE_MISMATCH:
      vd->expression =
        build_convert_function_call(vd->expression, vd->type);
      break;
    }
  }
  ...
}
```

The *check_type_match* helper is defined according to the rules in Section 2.3.3.

5.3.3 Importing files

When an import directive is encountered, the `rdp`-generated parser is invoked on the imported file and a pointer in the import directive node is set to point to the resulting abstract syntax tree. Then the contextual analysis proceeds by traversing the just created abstract syntax tree (with the same visitor object) before leaving the import directive node. Hence, the definitions from the file will eventually be entered into the identification table.

To remedy the problem with files that are included multiple times (cf. Figure 1.3 on page 13), a table of file names imported so far is maintained. Before a file is parsed, it is then checked whether it already has been processed. For example, for Figure 1.3, when C and A have been analysed and the import directive for

C in B is encountered, it is simply skipped since C is already in the table and the contained definitions already parsed and entered into the identification table.

```
void DecorateVisitor::visit(ImportDirective *id)
{
  if (!import_table.exist(id->filename)) {
    id->isim = rdp::parser(const_cast<char *>
                                 (id->filename.c_str()));
    ...
    visit(id->isim);
    ...
  }
}
```

5.3.4 Declarations

Declarations of variables and functions are maintained in a declaration table, which facilitates looking up identifiers and matching declarations with definitions.

Declarations are stored in the table with the identifier as the key, as illustrated in the following code:

```
void DecorateVisitor::visit(VariableDeclaration *vd)
{
  vd->type->visit(this);

  if (!decl_table.match(lookup_ident(vd->identifier), vd))
    decl_table.enter(vd->identifier, vd);
}
```

Since declarations are local to the file they appear in, a stack of mappings is maintained in the declaration table with the mapping at the top of the stack representing the declarations in the current file. Searching for an identifier during the analysis then amounts to searching the definition table first, and then the declaration table for the declarations in the current file if no definition was found.

Matching a declaration with the corresponding definition can occur at two places: when the declaration is encountered or when the definition is encountered. We define a partial match to be when the identifiers are the same and a full match for variables to be when additionally the type (including the `const` qualifier) match, and for functions when additionally the return type and the number of parameters and their types match.

Then when a definition is encountered, if there is a partial match, it is an error since name overloading is not supported. If there is a full match, the declaration

is removed from the table and a pointer to the definition is installed in the declaration so that the nodes that reference it can find the definition. From the variable definition visit method:

```
void DecorateVisitor::visit(VariableDefinition *vd)
{
  ...
  if (ident_table.level == 1) // check scope level
    decl_table.remove(vd);
}
```

Likewise, when a declaration is met, if there is a partial match, it is an error. If there is a full match, the declaration is simply not entered into the declaration table as the previous snippet showed.

When the analysis is complete, any declaration that does not bind itself to a definition is reported as an error by the error reporting phase.

5.4 Error reporting

The error reporting phase is implemented with the BUGFINDERVISITOR class that inherits from VISITOR. Any error mark found in a node from the contextual analysis is reported with a helpful message with the filename and line number. Warnings caused by implicit type conversions are also reported.

For example, the simple division function

```
float div(int x, int y){
  float d = x/y;
}
```

causes the following output from the compiler:

```
ERROR:div:1 - function is missing a return statement 'div'
WARNING:div:2 - implicit type conversion
errors found, now terminating
```

5.5 Code generation

The code generation phase is implemented with the class GENCODEVISITOR that inherits from VISITOR. When the tree is traversed, the code specified with $\mathcal{T}$ in Chapter 4 is simply emitted for each node. Some of the details of the translation were left out of the simple model of the JVM, though, and will instead be discussed here.

5.5.1 Overall design of the generated code

Since the JVM is designed to support classes only, we need to structure the generated code into a number of classes. We follow the pattern outlined in Figure 5.6.

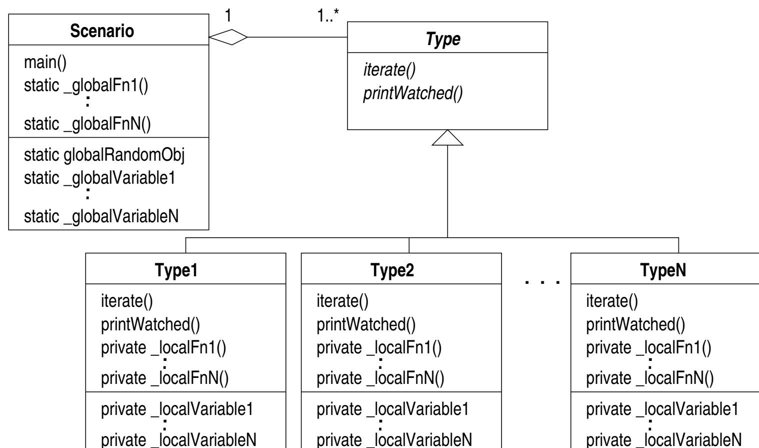

Figure 5.6: Overview of the design of the classes that the code generation phase produces.

One class, the SCENARIO class, contains the *main* method and aggregates a number of **type** objects through the abstract TYPE interface. The *main* method simply enters a loop, invoking *iterate* and *printWatched* on each of the **type** objects for each iteration. All global functions and variables are placed as static members of SCENARIO from where they accessible to the rest of the classes. The RANDOM object that is used as a random source for the probabilistic constructs is also placed here.

Since **type** objects may contain local variables, each **type** is translated to a class that derives from the TYPE interface. Functions and variables in the **type** are then translated to private methods and fields in the class. The *printWatched* method is automatically generated to output the values of the watched variables, if any.

5.5.2 Type defi nitions

When a type definition is encountered, the code generator switches output to a file with the name of the type, emits a class preamble and starts processing the definitions in the type. For example

```
type SomeType {
  int b;
  ...
}
```

is translated into

```
.class public Siml/SomeType
.super Type
.field private b I
```

After having processed the contents of the type definition node, the output is then again redirected to the SCENARIO class file.

5.5.3 Variable definitions

Although the JVM model in Chapter 4 did not take it into account, there are in fact three different kinds of variables that all need different JVM code. The global variables are specified as static members in the SCENARIO class, the variables defined in **type**s are specified as private members in the corresponding **type** class and variables that are local to functions are simply given a unique offset in the corresponding function.

So when the abstract syntax tree is traversed, each variable definition is decorated with the variable kind and either the local variable offset or the class name.

Variable definitions where the variable is assigned an initial value are for the first two kinds translated into assignments in the class constructors. For example, if the previous example is rewritten to

```
type SomeType {
  int b = 100;
  ...
}
```

the constructor for the class is then emitted as

```
.method <init>()V
 .limit stack 2
 .limit locals 1

 aload_0
 invokespecial java/lang/Object/<init>()V
 aload_0
 ldc 100                    ; b = 100
 putfield SomeType/b I
 return
.end method
```

5.5.4 Variable lookup

When a variable is referenced in a statement or an expression, the pointer to the variable definition (or to a variable declaration and then to the definition) is followed, and the output then determined by the kind of variable definition.

Global variables are accessed with the GETSTATIC and PUTSTATIC instructions. Type specific variables are accessed with the GETFIELD and PUTFIELD instructions. Local variables are simply accessed with LOAD and STORE instructions and the computed offset as specified in Chapter 4.

For example, the code

```
int a;
type SomeType {
  int b;
  void f()
  {
    int c;
    a = 1;
    b = 2;
    c = 3;
  }
  ...
}
```

causes the body of f to be translated to

```
ldc 1
putstatic Scenario/a I    ; lookup a in Scenario
ldc 2
aload_0
putfield SomeType/b I     ; lookup b in current object
ldc 3
istore 1                  ; lookup c in local variables
```

5.5.5 Function definitions

A function definition is translated into a header, two strings defining the maximum number of local variables and the maximum height of the stack, and the function body. Also, for **void** functions, a RETURN instruction is placed as the last instruction of the body. For example, the code

```
type SomeType {
  ...
  void f()
```

```
  {
    int b = 200;
  }
  ...
}
```

is translated into

```
.class SomeType
...
.method f()V
 .limit stack 1      ; max height of stack
 .limit locals 1     ; max number of local variables

 ldc 200
 istore 1
 return
.end method
```

5.5.6 Function call

Since global functions reside as static members of the SCENARIO class whereas type-specific functions are in the **type** classes, we need to generate different code for each of the cases. For the static members, INVOKESTATIC is used, e.g.

```
int g(int x)
{
  return x*2;
}

type SomeType {
  ...
  void f()
  {
    int b = g(200);
  }
}
```

where the body of f becomes

```
 ldc 200
 invokestatic SomeType/g(I) I
 istore 1
 return
```

For the others, the specific **type** object reference must first be pushed on the stack and then INVOKEVIRTUAL is used. For instance

```
type SomeType {
  ...
  int g(int x){
    return x*2;
  }

  void f()
  {
    g(200);
  }
}
```

where the body of f becomes

```
aload_0
ldc 200
invokevirtual SomeType/g(I) I
pop
return
```

If the function call is a statement and the function is non-**void**, we must also emit a POP to clear up the stack from the return value.

As a special case, the built-in intify and floatify are translated directly into the instructions F2I and I2F respectly.

5.5.7 Enumerations

When an **enum** definition is encountered, a unique integer value is associated with each enumerand by resetting a counter to one and incrementing it by one for each enumerand.

The contextual analysis phase has already established pointers from the occurences of enumerands in expressions to the **enum** definitions, so when an enumerand is encountered, the pointer is simply followed and the enumerand evaluated to its associated integer. For example, for

```
enum E { A, B };
void f()
{
  E x = A;
  E y = B;
}
```

the body of f is translated to

```
ldc 1       ; A is 1
istore 1
ldc 2       ; B is 2
istore 2
return
```

5.5.8 Import

When an import node is encountered, the extra imported abstract syntax tree which is rooted in the node is traversed before continuing so that code is generated for the functions and types in the imported file and the variable definitions are visited.

5.5.9 Create statements

The **create** statements are translated into loops in the SCENARIO constructor that construct the needed number of **type** objects and insert them into the object list so that *iterate* and *printWatched* can later be invoked on them. For example

```
import "SomeType.model"
...
create 10 of SomeType(1);
```

is translated into the following in the constructor.

```
 ldc 10
start:
 dup
 ifeq done:
 getstatic Scenario/typeList java/util/ArrayList;
 new SomeType
 dup
 ldc 1
 invokespecial SomeType/<init>(I)V
 invokevirtual java/util/ArrayList/add(Ljava/lang/object)Z
 pop
 ldc -1
 isub
 goto start
done:
```

5.5.10 Declarations

Declarations are only relevant in the contextual analysis phase and are simply ignored by the code generator.

5.6 Connecting the phases

The main function of the compiler is shown below, and provides an overview of the phases and how they use and modify the abstract syntax tree.

```
int main(int argc, char* args[])
{
  if (argc <= 1)
    return 1;

  Simulation *csim = rdp::parser(args[1]);

  if (csim == 0)
    return 1;

  DecorateVisitor v(args[1]);
  populate_ident_table(&v);
  v.visit(csim);

  BugfinderVisitor bfv(args[1]);
  bfv.visit(csim);
  if (bfv.halt) {
    std::cerr << "errors found, now terminating" << std::endl;
    return 1;
  }

  GencodeVisitor gcv;
  gcv.visit(csim);

  return 0;
}
```

The compilation is started by running the compiler with the source program as its only parameter, the other parameters are simply ignored. `rdp` is invoked on the source program, which returns a SIMULATION object representing the source program.

If for some reason `rdp` fails to parse the source program, it will return null and the compilation is halted. Before the contextual analysis is started the identification

table is populated with the built-in functions intify (**float**) and fbatify (**int**). After running the contextual analysis, the bug finder is initiated and will find and report any error marked by the contextual analysis. Lastly, if the bug finder did not find any errors, the code generator is started, outputting Jasmin assembler files.

5.7 Conclusion

An overview of how everything fits together is provided in Figure 5.7.

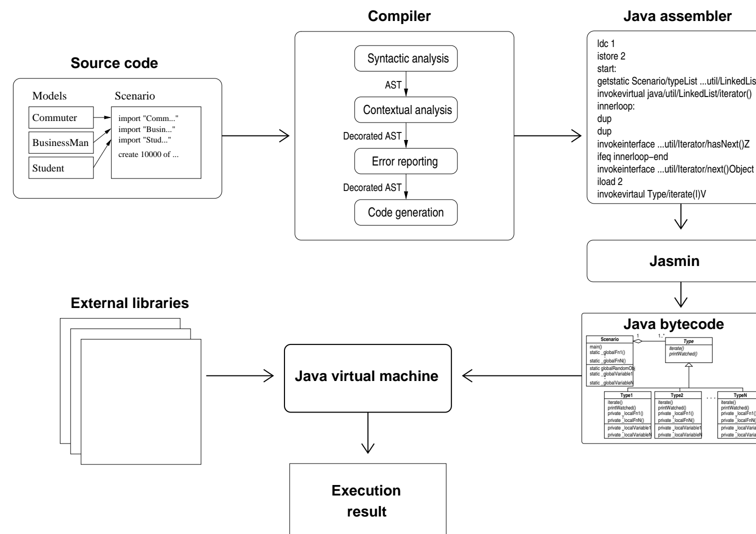

Figure 5.7: From source code to simulation result.

Below is the first part of the output from running the BusinessMan model in Chapter 1 with 20 iterations. The numbers in the parentheses represent the iteration number.

```
$ java Siml/Scenario 20
Siml/BusinessMan/train (1): true
Siml/BusinessMan/train (1): false
Siml/BusinessMan/train (1): false
Siml/BusinessMan/train (1): true
Siml/BusinessMan/train (1): false
Siml/BusinessMan/train (1): true
Siml/BusinessMan/train (1): true
Siml/BusinessMan/train (1): true
Siml/BusinessMan/train (1): false
```

```
Siml/BusinessMan/train (1): true
Siml/BusinessMan/train (1): true
Siml/BusinessMan/train (1): false
Siml/BusinessMan/train (1): false
...
```

Bibliography

[1] Erich Gamma et. al. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1994.

[2] Hans Hüttel. *Pilen ved træets rod – strukturel operationel semantik af programmeringssprog*. Uni-print, 2003.

[3] Adrian Johnstone and Elizabeth Scott. The RDP parser generator, December 1997. `http://www.cs.rhul.ac.uk/research/languages/projects/rdp.shtml`.

[4] Peter Lynggaard. *Driftsøkonomiske eksempler*. Handelshøjskolens forlag, 1997.

[5] Peter Lynggaard. *Driftsøkonomi*. Handelshøjskolens forlag, 2001.

[6] Jon Meyer. A Java assembler interface, March 1997. `http://mrl.nyu.edu/~meyer/jasmin/`.

[7] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, 1997.

[8] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications – a formal introduction*. John Wiley & Sons, 1999. `http://www.daimi.au.dk/~hrn/`.

[9] Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages – Design and Implementation*. Prentice Hall, 4 edition, 2000.

[10] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Co., 1996.

[11] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java*. Prentice Hall, 2000.

Appendix A

rdp grammar

```
(* START *)

simulation ::= { import_directive |
                 'extern ' ( 'const ' type_name_enum ID { ',' ID } ';' |
                          (type_name_enum ID ({ ',' ID } ';' |
                                                 function_header ';') |
                           'void ' ID function_header ';' )) |
                 type_name_enum ID ( variable_definition |
                                     function_definition ) |
                 'void ' ID function_definition |
                 'const ' type_name_enum ID variable_definition |
                 enum_definition |
                 type_definition |
                 create_statement }.

(* IMPORT *)

import_directive ::= 'import ' STRING('"'):filename ';'

(* DECLARATIONS *)

function_header ::= '(' [type_name_enum ID { ',' type_name_enum ID}] ')'.

(* DEFINITIONS *)

variable_definition ::= [ '=' exp1 ] { ',' ID [ '=' exp1 ]} ';'.
function_definition ::= function_header '{' { statement } '}'.
enum_definition ::= 'enum' ID '{' ID { ',' ID}'}'.
type_definition ::= 'type ' ID '{'{type_name_enum ID ( variable_definition |
                                                       function_definition ) |
                                  'void ' ID function_definition |
                                  'const ' type_name_enum ID variable_definition |
                                  'watched ' [ 'const '] type_name_enum ID
                                                        variable_definition |
                                  enum_definition }'}'.
```

```
(* INSTANTIATING *)

create_statement ::= 'create' INTEGER:no 'of' ID function_call ';' .

(* STATEMENTS *)

statement ::= ID ('=' exp1 ';' | function_call ';' | ID variable_definition) |
              '{'{statement} '}' |
              'while' '(' exp1 ')' statement |
              'if' '(' exp1 ')' statement ['else' statement] |
              'const' type_name_enum ID variable_definition |
              type_name ID variable_definition |
              '++'ID ';' |
              '--'ID ';' |
              'return' exp1 ';' .

function_call ::= '(' [exp1 {',' exp1}] ')'.

(* EXPRESSIONS *)

exp1 ::= exp2 [('&&' exp1 | '||' exp1)].
exp2 ::= exp3 [('==' exp2 | '!=' exp2)].
exp3 ::= exp4 [('<' exp3 | '>' exp3 | '<=' exp3 | '>=' exp3)].
exp4 ::= exp5 [':' exp5 {'|' exp5 ':' exp5}].
exp5 ::= exp6 [('+' exp5 | '-' exp5)].
exp6 ::= exp7 [('*' exp6 | '/' exp6 | '%' exp6)].
exp7 ::= exp8 ['^' exp7].
exp8 ::= exp9 ['...' exp8].
exp9 ::= exp10 | '!' exp10 | '-' exp10.
exp10 ::= '(' exp1 ')' | value | ID.

(* VALUES *)

value ::= INTEGER:i | REAL:r | 'true' | 'false'.
type_name ::= 'int' | 'float' | 'bool'.
type_name_enum ::= type_name | ID.

comment ::= COMMENT_LINE('//').
```