Spring 2017
DD2457 Program Semantics and Analysis
Lab Assignment 1: Abstract Machines

D. Gurov A. Lundblad

KTH Royal Institute of Technology

1 Introduction

This lab assignment is about implementing an interpreter for the **While** language based on the notion of *abstract machines* developed in Chapter 4 of the course book. The attractiveness of abstract machines derives from their conceptual simplicity, allowing both efficient implementations and formal treatment such as formal proofs of language implementation correctness.

As part of the assignment, the **While** language is extended with an arithmetic integer division operation giving rise to possible run–time errors in the case of division by zero, and a try–catch statement for handling exceptions of this kind. The extension is treated both formally and implemented in the interpreter.

The purpose of the lab assignment is to deepen the understanding of the approach to the correct implementation of high–level programming languages discussed in the course book. In this approach, the language is specified through a natural semantics. The implementation is given by means of a low–level abstract machine language, for which a structural operational semantics is given, and a translation of statements of the source language to sequences of instructions of the abstract machine. Correctness of the implementation is established as a formal proof that the translation of any statement, when executed by the abstract machine from any start state, results in the same final state (if there is one) as specified by the natural semantics.

This assignment also prepares the ground for the next assignment, where the abstract machine is adapted to run with abstract values (properties) instead of with concrete ones, supporting various program analyses.

The assignment is carried out in teams of (at most) two.

2 An Interpreter for While

Following the approach advocated in the course book, an interpreter for **While** can be built that consists of two parts:

a) a compiler from **While** to the abstract machine (assembly) language **AM** implementing the clauses in tables 4.2 and 4.3 in the book, and

b) a virtual machine for executing the rules of the operational semantics of **AM** given in Table 4.1.

An important aspect of this approach is that the two transformations have been *proved* to be correct with respect to the natural semantics of **While** (see Theorem 4.20), the latter providing a formal specification for the language. Thus, to produce a correct interpreter one has merely to implement correctly the transformations. The idea is that this should not pose great difficulty, due to the simple and straightforward nature of these transformations.

When extending **While** one should strive to design the **AM** extension so as to be able to reuse the existing proof (structure) as much as possible, such as for instance validity of Lemma 4.18.

3 Exceptions

Let us add division a_1/a_2 to the operations in the arithmetic expressions of our **While** language. Division by zero raises an *exception*; we therefore extend the statements with a new exception–handling construct

$$\textbf{try } S_1 \textbf{ catch } S_2$$

with the expected behaviour: if execution of S_1 terminates normally in a state, then the whole try–catch statement terminates normally in that state; if execution of S_1 terminates with an exception in a state, then statement S_2

takes over from that state.

To adapt the operational semantics of **While**, we add the special error value $\perp$ to the set of integer values, letting $Z_\perp \stackrel{\mathsf{def}}{=} Z \cup \{\perp\}$, and re-define the evaluation function $\mathcal{A} : \mathbf{AExp} \to (\mathbf{State} \to Z_\perp)$ to capture division by zero as the source of producing an exception, and propagation of the exception value.

Evaluation of Boolean expressions is handled similarly, by adding $\perp$ to the set of truth values, letting $\mathbf{T}_\perp \stackrel{\mathsf{def}}{=} \mathbf{T} \cup \{\perp\}$, and re-defining the evaluation function $\mathcal{B} : \mathbf{BExp} \to (\mathbf{State} \to \mathbf{T}_\perp)$ so that the exception value propagates.

To distinguish between normal and exceptional termination, we introduce the set of extended states $\mathbf{EState} \stackrel{\mathsf{def}}{=} \mathbf{State} \times \{\top, \perp\}$, where an extended state $(s, \top)$ is normal and $(s, \perp)$ is exceptional. By abuse of notation, we shall use s to denote the normal state $(s, \top)$, and $\hat{s}$ to denote the exceptional state $(s, \perp)$. This allows re–using much of the existing formalizations when extending the semantics.

Of course, you are welcome to propose alternative formalizations.

4 Implementing the Interpreter

The interpreter you will implement should work roughly as follows.

1. A **While** program is loaded from an external file and parsed into an abstract syntax tree (AST).
2. The AST is transformed into an **AM** program.
3. The resulting **AM** program is executed instruction by instruction in a virtual machine, starting from an initial state provided by the user. The complete trace of configurations is printed on standard output. You can also provide an optional *debugger* mode for stepping through the execution instruction by instruction, possibly with an indication of where in the **While** program the execution is currently.

You are free to implement the interpreter in a language of your own choice. However, in case you choose Java, we have prepared some fundamental classes for you. A brief description of the classes can be found in Table 1.

Class	Purpose
`WhileParser`	To parse a **While** program located in file `"prog.while"`, simply invoke `WhileParser.parse("prog.while")`. The method returns the root element (an `Stm` object) of the AST if the program was syntactically correct, and throws an exception otherwise.
`WhileVisitor`	This interface is implemented by classes that wish to traverse a **While** ASTs. It is important that you familiarize yourself with the visitor pattern by inspecting the sources and/or browsing the web.
`Pretty-Printer`	A sample class that pretty-prints a **While** program, using the visitor pattern.
`Compile-Visitor`	An (almost) empty implementation of the visitor interface, for you to complete.
`whilesyntax.*` `amsyntax.*`	These classes are used to represent **While** and **AM** ASTs. There is one class for each syntactical construct of each language.
`Main`	Contains a main method stub that parses and prints a **While** program. A good place for you to put your calls to your compiler and virtual machine.

Table 1: Brief description of the available classes.

The classes are located in the course directory: `/info/DD2457/semant15/lab1`. You should copy these files into your working directory and make sure you can compile and run `Main.java`:

```
> mkdir semanticslab
> cd semanticslab
> cp -r /info/DD2457/semant15/lab1/* .
> javac -cp .:java-cup-v11a.jar semant/Main.java
> java -cp .:java-cup-v11a.jar semant.Main samples/trycatchsample.while
```

If the compilation fails, make sure you are using Java version 1.5 or later (`module add jdk/1.5.0` should do the trick otherwise).

5 Tasks

The present lab assignment consists of the following tasks:

1. Implement an interpreter for **While** as discussed in Section 4 (if your choice of language is Java, simply complete the compiler and virtual machine of the skeleton in the course directory). Test your interpreter on several small but meaningful **While** programs, starting from different states.

2. As a specification, provide a *natural semantics* for **While** extended with exceptions by redefining tables 1.1, 1.2 and 2.1 as discussed above in Section 3. Use your natural semantics to execute the program:

   $$x := 7; \textbf{try } x := x - 7; x := 7/x; x := x + 7 \textbf{ catch } x := x - 7$$

   from an arbitrary state s (hint: the final state should be $s[x \mapsto -7]$).

3. As an implementation, provide an *abstract machine semantics* for **While** extended with exceptions by extending suitably the instruction set of the abstract machine and redefining tables 4.1, 4.2 and 4.3.

   (a) Use your abstract machine semantics to execute the program above, from the same (arbitrary) state s, and compare the resulting final states.

   (b) Argue semi–formally for correctness of your semantics. More specifically, explain how the original correctness proof is affected by the current language extension.

In particular, make sure that your abstract machine handles correctly *nested* tries-and-catches. For instance, in a program of the shape **try** (S_1; **try** S_2 **catch** S_3) **catch** S_4, an exception raised when executing S_1 should be caught in S_4 and not in S_3!

4. Adapt your interpreter to the extended language. Test your interpreter on several small but meaningful programs, including programs with nested tries-and-catches.

5. Write a *report* containing all your results, both theoretical and practical. In particular, the report should contain:

   (a) the added rules of the natural semantics of the extended **While** language, the added **AM** instructions, the added SOS rules of the abstract machine, and the added rules of the translation;

   (b) the derivation tree from Task 2 and execution from Task 3a;

   (c) the adapted semi–formal correctness argument from Task 3b; and

   (d) examples on which you tried your interpreter in Tasks 1 and 4.

6 Tips and Hints

- More information on the visitor pattern can be found at

  `http://en.wikipedia.org/wiki/Visitor_pattern`

- To ease the transition to the next lab, you are advised to let your execution method have a signature similar to the following one:

  `public Configuration step(Configuration conf).`

  Furthermore it is recommended that you let your representation of a configuration be immutable or cloneable (since the state space will be traversed).

- It is up to you to decide how to implement the virtual machine. The `Inst` class has an opcode field so one option would be to perform a switch on this value.

- For stepping through your execution instruction by instruction (say, when in debugger mode), the following line of code suspends the execution until the user presses enter:

  ```
  new BufferedReader(new InputStreamReader(System.in)).readLine();
  ```

- A few useful sample programs can be found in the `samples` directory. `gcd.while` also contains the corresponding **AM** code.
- The error messages from the parser may be a bit cryptic. Just keep in mind that it is not allowed to have a trailing "`;`" after the last statement of a **While** program and that comments (lines starting with `#`) needs to be terminated by a newline.