MP 6 – MiniJava Interpreter

CS 421 – Spring 2013
Revision 1.0

Assigned Thursday, February 21, 2012
Due Tuesday, February 26, 2012, 9:30AM
Extension 48 hours (20% penalty)
Total points 50

1 Change Log

1.0 Initial Release.

2 Objective

You will learn how to write an interpreter for MiniJava. You will start from ASTs generated by an ocamlyacc parser and ocamllex lexer. Your job is to implement, in OCaml, the structured operational semantics (SOS) rules we have written. We discussed how these rules work in lecture 11 (February 21); the rules given in this assignment follow the same idea, but are more extensive.

After completing this MP, you will have learned to:

- interpret structural operational semantics of a programming language
- write an interpreter on an abstract syntax tree
- implement expression evaluation and statement execution

This assignment is difficult. Please start early. Our solution is about 130 non-blank lines, of which about 30 are included in `mp6-skeleton`, so you'll need to write roughly 100 lines.

3 Style Requirements

In this MP, you will be expected to meet the following style requirements. Submissions that do not meet these requirements will not receive a grade until they are resubmitted with correct styling. (Note that these requirements will be checked manually by the grader and will not be enforced by `handin`, so acceptance by `handin` does not indicate correct style.)

- No long lines. Lines are 80 characters long.
- No tabs. Use spaces for indentation.
- Indents should be no more than 4 spaces, and must be used consistently.

4 Syntax and Semantics

Implementing an interpreter for all of MiniJava will take more than one week, so we start this week by limiting ourselves to expressions, `if`, assignment, and block statements, and a single class (called `Main`). We also limit our values to integers, strings, and booleans. Since there are no objects, method calls will ignore the receiver (the expression on the left-hand side of the dot). We will use the same concrete and abstract syntax as in MP2, but we will

```
type value = IntV of int | StringV of string | BoolV of bool | NullV
type varname = string
and binding = (varname * value)
and state = binding list

exception TypeError of string      (* use for errors detectable at compile time *)
exception RuntimeError of string   (* use for "value" errors *)
```

Figure 1: Essential declarations.

ignore parts of it. (Some of the functions you will define this week will produce the OCaml warning "this pattern-matching is not exhaustive," but you can ignore it; if you prefer, you can add `raise NotImplemented` to stop the complaints.)

As with regular Java, when the program is run, the method `main` is invoked; unlike regular Java, `main` returns a value. Furthermore, since we have no print statement in this subset of MiniJava, the value returned from `main` is the only visible result of the program; your solution will be judged correct or incorrect based on this value.

For this MP, we will treat MiniJava as a dynamically-typed language (that is, ignoring the type declarations on all variables) and will instead decide the semantics of operators such as + at runtime. This makes the language in some ways more similar to scripting languages like Python and JavaScript than to Java itself.

4.1 Structured Operational Semantics

The language is specified completely in a set of SOS rules given in section 7. These say exactly how expressions should be evaluated and what statements should do. They may be confusing at first, but you will find that the interpreter is a direct transcription of those rules. SOS is your friend!

5 Testing and handin

- Download `mp6grader.tar.gz`. This tarball contains all the files you need: `minijavaast.ml` contains the abstract syntax, `minijavalex.mll` a lexer, and `minijavaparse.mly` a parser. In other words, you can transform programs to abstract syntax.
- As always, once you extract the tarball, you should rename `mp6-skeleton.ml` to `mp6.ml` and start modifying the file. You will modify only the `mp6.ml` file, and it is the only file that will be submitted.
- Compile your solution with `make`. Run the `./grader` to see how well you do; look in `tests` to see the test cases. As usual, it is a good idea to test your solution on more test cases; do this either by adding test cases to `tests` or using interactive execution.
- *Interactive use:* Once you have finished appropriate sections of your interpreter, you may add programs to the test suite or you can use the the functions `run` and `run_with_args` to test individual programs interactively. These functions are defined in the included `testing.ml` file. After running `make`, run OCaml by typing `ocaml`; then type `#use "testing.ml";;` to load all of the related modules and enable testing. Further specifics on interactive testing are included in that file.
- Submit `mp6.ml` using the `handin` program.

6 Problems

The main functions in this assignment are `eval` (to evaluate expressions), `applyOp` (to apply binary operations to values), and `execStmt` (to execute statements). These are defined precisely in the SOS rules in section 7. There are also quite a few auxiliary functions that we have provided for your use. We strongly recommend that you define *and test* the functions in the order given here.

The declarations in Figure 1 are basic to the interpreter. Values of type `value` are what are produced by the evaluation of expressions; these are stored in the state, and passed as arguments to functions. The state is a "dictionary" pairing variable names with values.

6.1 Utility functions - provided

The functions in the first part of `mp6-skeleton.ml` are utilities to manipulate abstract syntax trees in simple ways, e.g. to search for a method with a particular name.

For each function, we give its first line, a brief description, and an example. You have also been given the code for these functions in `mp6-skeleton.ml`. These function headers come directly from `mp6-skeleton.ml`. We are using a new notation for function headers, where we state the types of arguments and result explicitly. (Although, as you know, this is not necessary in OCaml, it is good documentation, and can make the type error messages more understandable.)

1. `asgn` changes the value associated with a variable in the state; more precisely, it returns a new state where the value of that variable has been changed. (Raise `TypeError` if the variable isn't in the state; this is a type error because it can only happen if the assignment is to an undeclared variable, which is an error that could have been detected at compile time.)

```
let rec asgn (id:id) (v:value) (sigma:state) : state =
  ...

let sigma1 = [("x", IntV 4); ("y", StringV "abc"); ("z", BoolV true)];;

asgn "y" (IntV 10) sigma1;
- : state = [("x", IntV 4); ("y", IntV 10); ("z", BoolV true)]
```

2. `binds` checks if a variable occurs in a state.

```
let rec binds (id:id) (sigma:state) : bool =
  ...
binds "y" sigma1;;  (* true *)
binds "w" sigma1;;  (* false *)
```

3. `fetch` gets the value of a variable from the state. (Raise `TypeError` if the variable isn't in the state. However, this should not happen because `binds` should always be called before `fetch`.)

```
let rec fetch (id:id) (sigma:state) =
  ...
fetch "y" sigma1;;  (* StringV "abc" *)
```

4. `mklist` creates a list of a given length, containing that many copies of a given value.

```
let rec mklist (i:int) (v:value) : value list =
  ...
mklist 3 NullV;;  (* [NullV; NullV; NullV] *)
```

5. `zip` takes two lists of the same length, and makes a list of pairs of that same length, each pair containing the corresponding elements of the two lists. This is a generic function, but in our case we need it only to pair up lists of ids and values. (Raise `TypeError` if the lists are of different lengths.)

```
let rec zip (lis1:id list) (lis2:value list) : state =
  ...
zip ["a"; "b"] [IntV 10; BoolV true];; (* [("a", IntV 10); ("b", BoolV true)] *)
```

6. `zipscalar` matches up the elements of a list with a single value. (Our definition is not recursive because we defined it using `zip` and `list`, but you can use `let rec` and define it recursively if you prefer.)

```
let zipscalar (lis:id list) (v:value) : state =
  ...
zipscalar ["a"; "b"] NullV;; (* [("a", NullV); ("b", NullV)] *)
```

7. `varnames` extracts the list of names from a list of `var_decls`.

```
let rec varnames (varlis:var_decl list) : id list =
  ...
varnames [Var(IntType, "i"); Var(BoolType, "isEof")] (* ["i"; "isEof"] *)
```

8. `getMethodInClass` finds the method with a given name in a class. (Raise `TypeError` if the class doesn't have a method by that name.) (This is recursive, but our version uses a recursive auxiliary function, so `getMethodInClass` is not recursive itself; hence `let` instead of `let rec`.)

```
let getMethodInClass (id:id) (Class(_, _, _, methlis)) : method_decl =
```

9. `getMethod` finds a method with a given name in a program. In the restricted version of MiniJava we're using here, there is just one class, so it actually just calls `getMethodInClass` on the single class in the program.

```
let getMethod (id:id) (Program classes) : method_decl =
```

6.2 `eval`

The definition of `eval` is the main part of this assignment. First, define `applyOp`, then `eval`. (We will not be able to complete `eval` in this part of the problem, because we can't define method calls until we can interpret statements, which will be in the next section. We recommend testing these using the function `evaluate` in `testing.ml` before moving on to statements and methods.)

1. `applyOp` is used to define "strict" binary operations — operations that first evaluate both of their arguments. Hence, it applies to values:

```
let applyOp (bop:binary_operation) (v1:value) (v2:value) : value =
```

Here, you need to refer to the SOS rules. The strict operations are: `Plus`, `Minus`, `Multiplication`, `Division`, `LessThan`, and `Equal`; we are going to ignore `GreaterThan`, `LessThanEq`, and `GreaterThanEq` because they add no interesting concepts. (The non-strict operations, which are handled in `eval`, are the boolean operators `And` and `Or`.) As in Java and Python, `Plus` is "overloaded," including not only arithmetic addition but also string concatenation; the other operations are more narrowly defined. You should raise a value error where that is indicated in the SOS rules; **in cases where the SOS rules do not say what to do at all — for example, how to add an integer and a bool — you should raise a type error.**

2. `eval` is the heart of the assignment. Again, you should follow the SOS rules closely; report type errors whenever there is a case that the SOS rules do not cover.

```
let rec eval (e:exp) (sigma:state) (prog:program) : value =
```

The application of strict binary operations is just a matter of recursively evaluating the arguments and then calling `applyOp`. Constants are easily evaluated (but be careful of the difference between a literal (like `Integer 5`) and the value to which it evaluates (`IntV 5`). The integer case has been implemented for you. Variables are just looked up in the state (using `fetch`).

The `prog` argument to `eval` is used only for method calls, to find the method definition, so it will not come into play until section 6.4, but it still needs to be passed in recursive calls.

The boolean operations are special in that they aren't strict. For example, to evaluate `e1 && e2` you start by recursively evaluating `e1`; if it returns `BoolV false`, then you don't have to, and shouldn't, evaluate `e2`. Again, this is shown in the SOS rules.

Our solutions for `applyOp` and `eval` are each about thirty lines.

6.3 Statements

This week's version of MiniJava includes simple statements: assignment and `if`, as well as statement sequences. The function of a statement (in this interpreter as in any other language with statements) is to *change the state*. Accordingly, you will define a function `execStmt` that has a state as its argument and returns a new state. The function to execute a *sequence* of statements is just a simple recursive application of `execStmt`:

```
and execstmt (s:statement) (sigma:state) (prog:program) : state =

and execstmtlis (sl:statement list) (sigma:state) (prog:program) : state =
```

(Make these mutually-recursive with the definition of `eval`; it is not actually necessary now, but will be in the next section.)

`execStmt` is a `match` expression with three clauses, corresponding to the statements we are including this week: `Assignment(id,e)`, `Block sl`, and `If(e,s1,s2)`. For `Assignment`, call `eval` to evaluate the right-hand side, and then use `asgn` defined above (which correctly throws a type error if the assignment is to a variable that has not been declared). `If` calls `eval` for the condition, and then recursively calls `execStmt` for *either* the true or false statement. Statement sequences are executed by calling `execstmtlis`; that function is straightforward, but just remember to feed the state produced by each statement into the recursive call for the remaining statements. (And, again, the SOS rules cover all of this.)

6.4 Method calls

In MiniJava, method calls appear only as expressions. They have the form e.f($args$), but in this week's version, since we don't have objects, the "receiver" object e is ignored. (It still has to be included because we don't want to alter the syntax; in all of our test cases, we just write `null.f(...)`.)

So we need to go back to `eval` and add a new clause for method calls. Because methods include statements, we will have to call `execstmt` at some point when evaluating a method call, and since statements contain expressions, we need to call `eval` from `execstmt`, so `eval` and `execStmt` are mutually-recursive.

The method-call clause in `eval` does the following (again shown in the SOS rule): (1) Find the method (this is what the `prog` argument is for). Suppose it has arguments $x_1, \ldots, x_n$ and local variables $y_1, \ldots, y_m$ (the types of these variables is ignored; only the names matter). (2) evaluate the arguments in this call, producing values $v_1, \ldots, v_n$. (3) create a state in which each x_i is bound to v_i, and each y_i is bound to `NullV`. (4) Call `evalMethodCall`, passing the body and return expression of the method, and the state constructed in step (3):

```
evalMethodCall (stms:statement list) (retval:exp) (sigma:state) (prog:program) : value
```

`evalMethodCall` is simple: call `execStmt` to execute the method body and then `eval` to evaluate the return value.

To evaluate the argument list of a method (step (2) above), you need a function to evaluate a list of expressions; this is a simple recursion on the first argument (note that the state cannot change during an expression evaluation):

```
and evallist (el:exp list) (sigma:state) (prog:program) : value list =
```

You're done! Remember, you can test the entire interpreter using the `run` and `run_with_args` functions - examples of using these are in `testing.ml`. You can (and should!) add more test cases to the rubric by editing the `tests` file. Just follow the pattern for the existing test cases.

7 Formal specification

We provide a concrete syntax for this week's subset of MiniJava in figure 2. We also repeat the definition of the abstract syntax, which you have seen before.

The SOS rules themselves are given in two parts, the rules for expression evaluation and those for statement execution. The syntax for expression evaluation judgments is:

$$e, \sigma, \pi \Downarrow v$$

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{ClassDecl} \\
\mathit{ClassDecl} & ::= & \texttt{class Main}\ \{\ \mathit{MethodDecl}^{*}\ \} \\
\mathit{MethodDecl} & ::= & \texttt{public}\ \mathit{Type}\ \mathit{Id}\ \texttt{(}\ (\mathit{Type}\ \mathit{Id}\ (\texttt{,}\ \mathit{Type}\ \mathit{Id}\ )^{*})^{?}\texttt{)}\ \{\ \mathit{VarDecl}^{*}\ \mathit{Statement}^{*}\ \texttt{return}\ \mathit{Expression}\ \texttt{;}\ \ \} \\
\mathit{VarDecl} & ::= & \mathit{Type}\ \mathit{Id}\ \texttt{;} \\
\mathit{Type} & ::= & \texttt{int}\ |\ \texttt{string}\ |\ \texttt{boolean} \\
\mathit{Statement} & ::= & \{\ \mathit{Statement}^{*}\ \} \\
 & | & \texttt{if}\ \texttt{(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Statement}\ \texttt{else}\ \mathit{Statement} \\
 & | & \mathit{Id} = \mathit{Expression}\ \texttt{;} \\
\mathit{Expression} & ::= & \texttt{(}\ \mathit{Expression}\ \texttt{)} \\
 & | & \mathit{int}\ |\ \mathit{string}\ |\ \texttt{true}\ |\ \texttt{false} \\
 & | & \mathit{Id} \\
 & | & \mathit{Expression}\ \texttt{.}\ \mathit{Id}\ \texttt{(}\ (\mathit{Expression}\ (\texttt{,}\ \mathit{Expression}\ )^{*})^{?}\texttt{)} \\
 & | & \mathit{Expression}\ (+\ |\ -\ |\ \star\ |\ /\ |\ ==\ |\ <\ |\ \texttt{\&\&}\ |\ \texttt{||}\ )\ \mathit{Expression} \\
 & | & \texttt{!}\ \mathit{Expression} \\
\mathit{Id} & ::= & \langle\text{identifier}\rangle
\end{array}
$$

Figure 2: MiniJava concrete syntax.

where e is an `exp` (Figure 3), σ is a state (Figure 1), π a `program` (Figure 3), and v a value (Figure 1). This asserts that e will evaluate to v in the given state and program (unless it raises some kind of exception). We are using Greek letters, but otherwise these items exactly match the arguments and result of `eval`, so that implementing these rules should be straightforward. Some of the rules will be used directly in `eval`; for strict (i.e. non-boolean) operations, they will be implemented in `applyOp`.

The judgments for statement execution have the form

$$s, \sigma, \pi \Rightarrow \sigma'$$

where s is a statement, σ and σ' states, and π the program. This asserts that s, if executed in state σ, will change that state to σ'. Again, this matches the arguments and result of `execStmt`.

There are several notations used in the SOS rules:

$\sigma(x)$ means `fetch` x σ
$\sigma(x) \neq\perp$ means `binds` x σ = true
$\sigma[v/x]$ means `asgn` x v σ

For reference, we repeat the definition of `value`:

```
type value = IntV of int | StringV of string | BoolV of bool | NullV
```

(Abstract syntax and SOS rules follow.)

```
type program = Program of (class_decl list)

and class_decl = Class of id * id
        * ((var_kind * var_decl) list)
        * (method_decl list)

and method_decl = Method of exp_type * id * (var_decl list)
        * (var_decl list) * (statement list) * exp

and var_decl = Var of exp_type * id

and var_kind = Static | NonStatic

and statement = Block of (statement list)
    | If of exp * statement * statement
    | Assignment of id * exp
      (* the following statement constructors not used this week *)
    | While of exp * statement | Println of exp
    | ArrayAssignment of id * exp * exp
    | Break | Continue

and exp = Operation of exp * binary_operation * exp
    | Integer of int
    | True
    | False
    | Id of id
    | Not of exp
    | Null
    | String of string
    | MethodCall of exp * id * (exp list)
      (* the following exp constructors not used this week *)
    | Array of exp * exp | Length of exp | This
    | NewArray of exp_type * exp | NewId of id | Float of float

and binary_operation = And | Or | LessThan | Plus | Minus
    | Multiplication | Division | Equal

and exp_type = ArrayType of exp_type | BoolType | IntType
    | ObjectType of id | StringType | FloatType

and id = string;;
```

Figure 3: MiniJava abstract syntax.

(INT)	$i, \sigma, \pi \Downarrow \mathsf{IntV}(i)$	
(STRING)	$s, \sigma, \pi \Downarrow \mathsf{StringV}(s)$	
(BOOL-TRUE)	$\texttt{true}, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{true})$	
(BOOL-FALSE)	$\texttt{false}, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{false})$	
(NULL)	$\texttt{null}, \sigma, \pi \Downarrow \mathsf{NullV}$	
(VAR)	$x, \sigma, \pi \Downarrow \sigma(x)$	if $\sigma(x) \neq \bot$
(NOT)	$\texttt{!}\ e, \sigma, \pi \Downarrow \mathsf{BoolV}(\neg b)$	
	$\quad e, \sigma, \pi \Downarrow \mathsf{BoolV}(b)$	
(AND-TRUE)	$e_1 \texttt{\&\&}\ e_2, \sigma, \pi \Downarrow v_2$	
	$\quad e_1, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{true})$	
	$\quad e_2, \sigma, \pi \Downarrow v_2$	
(AND-FALSE)	$e_1 \texttt{\&\&}\ e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{false})$	
	$\quad e_1, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{false})$	
(OR-TRUE)	$e_1 \texttt{\|\|}\ e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{true})$	
	$\quad e_1, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{true})$	
(OR-FALSE)	$e_1 \texttt{\|\|}\ e_2, \sigma, \pi \Downarrow v_2$	
	$\quad e_1, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{false})$	
	$\quad e_2, \sigma, \pi \Downarrow v_2$	
(BINOP-INTEGER)	$e_1\ op\ e_2, \sigma, \pi \Downarrow \mathsf{IntV}(i_1\ op\ i_2)$	where *Op* $=$ `+` , `-` , *or* $\star$
	$\quad e_1, \sigma, \pi \Downarrow \mathsf{IntV}(i_1)$	
	$\quad e_2, \sigma, \pi \Downarrow \mathsf{IntV}(i_2)$	
(BINOP-INTEGER-DIV)	$e_1 \texttt{/}\ e_2, \sigma, \pi \Downarrow \mathsf{IntV}(i_1 \div i_2)$	if $i_2 \neq 0$
	$\quad e_1, \sigma, \pi \Downarrow \mathsf{IntV}(i_1)$	
	$\quad e_2, \sigma, \pi \Downarrow \mathsf{IntV}(i_2)$	
(BINOP-ZERO-DIV)	$e_1 \texttt{/}\ e_2, \sigma, \pi \Downarrow \mathit{RuntimeError}\ \texttt{"DivisionByZero"}$	
	$\quad e_1, \sigma, \pi \Downarrow \mathsf{IntV}(i_1)$	
	$\quad e_2, \sigma, \pi \Downarrow \mathsf{IntV}(0)$	
(BINOP-LESS)	$e_1 \texttt{/}\ e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(i_1 < i_2)$	
	$\quad e_1, \sigma, \pi \Downarrow \mathsf{IntV}(i_1)$	
	$\quad e_2, \sigma, \pi \Downarrow \mathsf{IntV}(i_2)$	

Figure 4: SOS rules for evaluation, part 1

(STRING-PLUS1) $e_1 + e_2, \sigma, \pi \Downarrow \mathsf{StringV}(s_1 \wedge \text{to-string } v_2)$

$e_1, \sigma, \pi \Downarrow \mathsf{StringV}(s_1)$

$e_2, \sigma, \pi \Downarrow v$

(STRING-PLUS2) $e_1 + e_2, \sigma, \pi \Downarrow \mathsf{StringV}(\text{to-string } v_1 \wedge s_2)$

$e_1, \sigma, \pi \Downarrow v_1$

$e_2, \sigma, \pi \Downarrow \mathsf{StringV}(s_2)$

where to-string (StringV s) = s, to-string (IntV i) = string_of_int i, etc.

(EQUAL-STRING) $e_1 == e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(s_1 = s_2)$

$e_1, \sigma, \pi \Downarrow \mathsf{StringV}(s_1)$

$e_2, \sigma, \pi \Downarrow \mathsf{StringV}(s_2)$

(EQUAL-INT) $e_1 == e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(i_1 = i_2)$

$e_1, \sigma, \pi \Downarrow \mathsf{IntV}(i_1)$

$e_2, \sigma, \pi \Downarrow \mathsf{IntV}(i_2)$

(EQUAL-BOOL) $e_1 == e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(b_1 = b_2)$

$e_1, \sigma, \pi \Downarrow \mathsf{BoolV}(b_1)$

$e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(b_2)$

(EQUAL-NULL) $e_1 == e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(true)$

$e_1, \sigma, \pi \Downarrow \mathsf{NullV}$

$e_2, \sigma, \pi \Downarrow \mathsf{NullV}$

(NOTEQUAL-NULL) $e_1 == e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(false)$ if $v \neq \mathsf{NullV}$

$e_1, \sigma, \pi \Downarrow \mathsf{NullV}$

$e_2, \sigma, \pi \Downarrow v$

(NOTEQUAL-NULL) $e_1 == e_2, \sigma, \pi \Downarrow \mathsf{BoolV}(false)$ if $v \neq \mathsf{NullV}$

$e_2, \sigma, \pi \Downarrow v$

$e_1, \sigma, \pi \Downarrow \mathsf{NullV}$

(METHOD-CALL) $x\,.\,f(e_1, \ldots, e_n), \sigma, \pi \Downarrow v$ if $\pi(Main)(f) =$ Method(_,f,$x_1 \ldots x_n$, $y_1 \ldots y_m$, s, e)

$e_1, \sigma, \pi \Downarrow v_1$

$\vdots$

$e_n, \sigma, \pi \Downarrow v_n$

$s, [(x_1, v_1); \ldots; (y_1, \mathsf{NullV}); \ldots], \pi \Rightarrow \sigma'$

$e, \sigma', \pi \Downarrow v$

Figure 5: SOS rules for evaluation (part 2)

(STMT-LIST) $s_1\texttt{;}\ \ldots\texttt{;}\ s_n\texttt{;}\ , \sigma_1 \Rightarrow \sigma_{n+1}$

$s_1, \sigma_1, \pi \Rightarrow \sigma_2$

$\vdots$

$s_n, \sigma_n, \pi \Rightarrow \sigma_{n+1}$

(IF-TRUE) $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma, \pi \Rightarrow \sigma_1$

$e, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{true})$

$s_1, \sigma, \pi \Rightarrow \sigma_1$

(IF-FALSE) $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2, \sigma, \pi \Rightarrow \sigma_2$

$e, \sigma, \pi \Downarrow \mathsf{BoolV}(\texttt{false})$

$s_2, \sigma, \pi \Rightarrow \sigma_2$

(ASSIGN) $x = e, \sigma, \pi \Rightarrow \sigma[v/x]$ if $\sigma(x) \neq \bot$

$e, \sigma, \pi \Downarrow v$

Figure 6: SOS rules for statement execution