Action Semantics of Specimen

Hermano Perrelli de Moura
Federal University of Pernambuco
Department of Informatics
Recife, Brazil
<hermano@di.ufpe.br>

October 10, 1996

Abstract

This document gives the definition of microSpecimen, a simple imperative language designed to serve as an example language in our research on semantics-directed compiler generation.

Contents

1 Informal Description

SPECIMEN is a simple imperative programming language. It has two types of abstractions: procedures and functions, which can be recursive. Functions are higher-order and free of side effects, although they can access non-local variables. Integers, booleans, and arrays are the only data types in the language.

1.1 Programs

Syntax

Program = ⟦ "`program`" Identifier "`is`" Declaration "`in`" Command "`end`" ⟧ .

Semantics

- A SPECIMEN program is a declaration followed by a command. The execution of a program consists of the elaboration of the declaration, followed by the execution of the command using the bindings produced by the declaration. These bindings are visible throughout the command (global bindings); holes in the scope of global bindings can occur by local declarations of the same global identifier.

Examples

Figure 1 and 2 show two examples of SPECIMEN programs.

1.2 Type Denoters

Syntax

(1) ValueType = PrimitiveType | FunctionType .

(2) PrimitiveType = "`bool`" | "`int`" .

(3) FunctionType = ⟦ ValueType "`->`" ValueType ⟧ |
⟦ "`(`" TupleType "`)`" "`->`" ValueType ⟧ .

(4) TupleType = ⟦ ⟧ | ⟦ ValueType ⟧ | ⟦ ValueType "`,`" TupleType ⟧ .

(5) ArrayType = ⟦ "`array`" "`[`" Numeral "`]`" "`of`" PrimitiveType ⟧ .

```
program bindings is
    const ca : bool = true;
    const cb : int = 346;
    const cc : int = 3;
    var ba : bool := true;
    var bb : bool := true;
    var bc : bool := true;
    var bd : bool := true;
    var be : bool := true;
    var bf : bool := true;
    var bg : bool := true;
    var bh : bool := true;
    var bi : bool := true;
    var bj : bool := true;
    var bk : bool := true;
    var bl : bool := true;
    var bm : bool := true;
    var bn : bool := true;
    var ia : int := 56;
    var ib : int := 0;
    var ic : int := 3;
    var id : int := 20;
    var ie : int := 406;
    var iF : int := 78;
    var ix : int := 4;
    var iy : int := 45;
    var iz : int := 4;
    var counter : int := 1000
in
    while (counter > 0) do
        ix := ix + cc;
        iz := ia + ib + ic + id + ie;
        bb := bc and bf and bh and bn;
        iy := ie + (ia + ib + ic + iF);
        counter := counter - 1
    end
end
```

Figure 1: The *bindings* program.

```
program block is
    const c : int = 3;
    var x : int := 4;
    var counter : int := 0;
    proc block () =
            local
                var y : int := 6
            in
                y := 20 + x
            end
in
    while (counter < 500000)
        do
            call block ();
            counter := counter + 1
        end
end
```

Figure 2: The *block* program.

Semantics

Types of constants, variables, procedures, functions and arrays must be explicitly declared. A SPECIMEN expression always evaluate to a value of type ValueType. Booleans and integers are primitive values; they have type PrimitiveValue. Array components are primitive values.

Examples

- *int* is the type of integer values.
- **array** `[20]` **of** *bool* is the type of an array of 20 components of type *bool*.
- `(`*int*`,`*int*`) ->` *bool* is the type of a function which expects two arguments of type *int* and returns a result of type *bool*.
- *bool* `-> (`*int* `->` *int*`)` is the type of a function which expects one argument of type *bool* and returns a function of type *int* `->` *int* as its result.

1.3 Declarations

Syntax

(1) Declaration = ⟦ "`const`" Identifier "`:`" PrimitiveType "`=`" Expression ⟧ **|**
⟦ "`var`" Identifier "`:`" PrimitiveType "`:=`" Expression ⟧ **|**
⟦ "`varray`" Identifier "`:`" ArrayType "`:=`" "`[`" ArrayComps "`]`" ⟧ **|**
⟦ "`proc`" Identifier "`(`" ProcFormals "`)`" "`=`" Command ⟧ **|**
⟦ "`fun`" Identifier "`(`" FunFormals "`)`" "`:`" ValueType "`=`" Expression ⟧ **|**
⟨ Declaration ⟨ "`;`" Declaration ⟩* ⟩ .

(2) ArrayComps = Expression **|** ⟦ Expression "`,`" ArrayComps ⟧ .

(3) ProcFormals = ⟦ ⟧ **|** ⟦ ProcFormal ⟧ **|** ⟦ ProcFormal "`,`" ProcFormals ⟧ .

(4) ProcFormal = ⟦ Identifier "`:`" ValueType ⟧ **|** ⟦ "`var`" Identifier "`:`" PrimitiveType ⟧ .

(5) FunFormals = ⟦ ⟧ **|** ⟦ FunFormal ⟧ **|** ⟦ FunFormal "`,`" FunFormals ⟧ .

(6) FunFormal = ⟦ Identifier "`:`" ValueType ⟧ .

Semantics

Declarations allow the programmer to make bindings. SPECIMEN has four types of declarations: constant, variable, procedure and function declarations. Procedure and function declarations can be recursive. A SPECIMEN program must have at least one declaration.

- A constant declaration "**const** I`:`T `=` E" binds the constant identifier I to the primitive value yielded by the evaluation of expression E. I and E have primitive type T.
- A variable declaration "**var** I`:`T `:=` E" binds the variable identifier I to a newly allocated cell. The result of the evaluation of the expression E is stored in this cell. I and E have primitive type T.
- An array declaration "**varray** I`:`**array** $[N]$ **of** T `:=` E" binds the array variable identifier I to a list of N newly allocated cell. The expression E, an array aggregate with N components, is evaluated and every resultant component is stored in the correspondent cell. I and E have array type "**array** `[`N`]` **of** T", where T is a primitive type.

- A procedure declaration "`proc` I `(`FS`) =` C" binds the procedure identifier I to a procedure. A procedure is a *command abstraction.* Zero or more formal parameters can be defined in the procedure declaration. Parameters can be passed by value or by reference (var parameters). When passed by value, the argument, a value, is bound to the correspondent formal parameter; when passed by reference the argument, a cell, is bound to the correspondent formal variable. The body of a procedure is just a command. The command is executed in an the same environment of the procedure declaration overlaid by the bindings produced by the elaboration of the procedure and its formal parameters. No result is returned, a procedure works by its side effects only. Procedure can be recursive. Procedures cannot be passed as parameters (of procedures or functions) or return as result of functions.
- A function declaration "`fun` I `(`FS`):`T `=` E" binds the function identifier I to a function. A function is an *expression abstraction.* SPECIMEN functions can have zero or more parameters. Parameters are passed by value only. The body of a function is just an expression which is evaluated in the same environment of the function declaration overlaid by the bindings produced by the elaboration of the function and its formal parameters. An application of function I always returns a value of value type T. Functions can be recursive.
- The sequential declaration "D_1 `;` D_2" is elaborated by elaborating D_1 and then elaborating D_2. D_2 is elaborated in the environment of the sequential declaration overlaid by the bindings produced by D_1. The resultant bindings are the ones produced by D_1 overlaid by those produced by D_2.

Examples

- **const** *year* : *int* = `1991`
- **const** *yearPlusOne* : *int* = *year* + `1`
- **const** *factOfFour* : *int* = *fact* `(4)`
- **var** *x* : *bool* := *false*
- **var** *y* : *bool* := *not* (*x*)
- **varray** *vec* : **array** [4] **of** *int* := `[3,5,9,0]`
- **proc** *inc* (**var** *x* : *int*) = *x*:= *x* + `1`
- **fun** *isGreaterThan* (*a* : *int*, *b* : *int*) : bool = **if** *a* > *b* **then** *true* **else** *false* **end**

1.4 Commands

A command is used to update variables. SPECIMEN has six types of commands: assignment, conditional, while, procedure call, block and sequential.

Syntax

(1) Command = ⟦ Identifier ":=" Expression ⟧ |
⟦ Identifier "[" Expression "]" ":=" Expression ⟧ |
⟦ "if" Expression "then" Command ⟨ "else" Command ⟩? "end" ⟧ |
⟦ "while" Expression "do" Command "end" ⟧ |
⟦ "call" Identifier "(" ProcActuals ")" ⟧ |
⟦ "local" Declaration "in" Command "end" ⟧ |
⟨Command ⟨ ";" Command ⟩* ⟩ .

(2) ProcActuals = ⟦ ⟧ | ⟦ ProcActual ⟧ | ⟦ ProcActual "," ProcActuals ⟧ .

(3) ProcActual = ⟦ Expression ⟧ | ⟦ "var" Identifier ⟧ .

Semantics

- The assignment "I := E is the simplest form of command. It assigns the value of expression E to variable I. The expression is evaluated and the resultant value is stored in the cell bound to the variable identifier.
- The array assignment "I[E_1] := E_2" updates a component of array I. The component is the one indexed by the result of evalution of E_1 which must be a positive integer. The primitive value resultant from the evaluation of E_2 will be the new component.
- The conditional command "**if** E **then** C_1 **else** C_2 **end**" is executed as follows: the expression E is evaluated; if the evaluation result is true then command C_1 is executed; otherwise command C_2 is executed.
- The while command "**while** E **do** C **end**" is executed as follows: (a) the expression E is evaluated; (b) if its value is true then the subcommand C is executed and then we start from step (a) again; otherwise, the while-command is terminated.
- A procedure call, "**call** I (A)", causes the execution of the procedure bound to identifier I. A parameter can be passed by value or by reference (`var` formal). By value, "I (E)", the actual parameter expression E is evaluated and its value is bound to the formal paramenter; by reference, "I_1 (`var` I_2)", the cell bound to variable I_2 is bound to the formal parameter. In the latter case the actual parameter must be a variable so the value which is bound to the formal paramenter is a cell. Functions can be actual parameters of procedures.
- The block command "**local** D **in** C **end**" is executed as follows. The declaration D is elaborated. The command C is then executed in the environment of the block command overlaid by the bindings produced by D.
- The sequential command "C_1 ; C_2" is executed by first executing C_1 and then executing C_2.

Examples

- i := 314
- x := x + 1
- **if** $x = 0$ **then** x := x + 1 **else** $x := x$ - 1 **end**
- **while** x = *true* **do** x := *true and true* **end**
- *vec*[t] := 345 ; **call** p (3 + 5)
- **local var** x : *int* := 2 **in** x := x + y **end**

1.5 Expressions

The evaluation of an expression yields a value (an integer, a boolean, or a function value). Expressions are completely free of side effects. Functions are first class: they can be passed as arguments to procedures and functions and returned as a result of a function application.

Syntax

(1) Expression = “true” |
“false” |
Numeral |
Identifier |
⟦ Identifier “[” Expression “]” ⟧ |
⟦ “if” Expression “then” Expression ⟨ “else” Expression ⟩$^?$ “end” ⟧ |
⟦ Identifier “(” FunActuals “)” ⟧ |
⟦ Expression Operator Expression ⟧ |
⟦ Operator Expression ⟧
⟦ “let” Declaration “in” Expression “end” ⟧ .

(2) FunActuals = ⟦ ⟧ | ⟦ FunActual ⟧ | ⟦ FunActual “,” FunActuals ⟧ .

(3) FunActual = ⟦ Expression ⟧ .

(4) Operator = “not” | “and” | “or” |
“~” | “+” | “-” | “*” | “/” | “<” | “>” | “<=” | “>=” |
“=” | “<>” .

Semantics

- A literal expression is one that does not need additional evaluation (it is already in a canonical form). SPECIMEN has two types of literal expressions: boolean and integer.
- An identifier is an expression. Its evaluation gives the primitive value bound to it, in the case of a constant; or the function bound to it, in the case of a function identifier, respectively; or the content of the cell bound to it, in the case of a variable.
- The expression “I[E]” is evaluated as follows: expression E is evaluated and its result, a positive integer, say n, is used to access the nth component of array I which is the result of the whole expression.
- The conditional expression “**if** E_1 **then** E_2 **else** E_3 **end**” is evaluates as follows: the expression E_1 is evaluated; if the evaluation result is true then E_2 is evaluate and the result value is the result of the conditional expression; otherwise E_3 is evaluate and the result value is the result of the conditional expression.
- The function application, I ($E_1, \ldots, E_n$), evaluates as follows: each parameter E_i is evaluated and a list of arguments is formed with the resultant values. The function bound to I is then applied to the list of arguments. Each occurrence of the formal parameter inside the function body is replaced by the correspondent resultant value; the function body is evaluated and its result is returned as the result of the function application.
- Constants, variables and functions can be defined locally using a let expression. (Procedures can also be defined locally but they cannot be used, so in practice there is no sense in defining them.)
- Conventional operations on booleans and integers are provided. Equality and inequality between primitive values are present.

Examples

- `true`
- `4563738`

- x
- **if** *tag* **then** *process* (*tag*) **else** *unprocess* (*tag*) **end**
- g (x + 3) - f (x - 3)
- `~3`
- `3 + 4 - 4 * 3 - 9`
- `(3 + 4) - (4 * 3 - 9)`

2 Abstract Syntax

needs: Lexical Syntax .

closed

grammar:

2.1 Programs

(1) Program = ⟦ "`program`" Identifier "`is`" Declaration "`in`" Command "`end`" ⟧ .

2.2 Declarations

(1) Declaration = ⟦ "`const`" Identifier "`:`" PrimitiveType "`=`" Expression ⟧ **|**
⟦ "`var`" Identifier "`:`" PrimitiveType "`:=`" Expression ⟧ **|**
⟦ "`varray`" Identifier "`:`" ArrayType "`:=`" "`[`" ArrayComps "`]`" ⟧ **|**
⟦ "`proc`" Identifier "`(`" ProcFormals "`)`" "`=`" Command ⟧ **|**
⟦ "`fun`" Identifier "`(`" FunFormals "`)`" "`:`" ValueType "`=`"
Expression ⟧ **|**
⟨ Declaration ⟨ "`;`" Declaration ⟩* ⟩ .

(2) ArrayComps = Expression **|** ⟦ Expression "`,`" ArrayComps ⟧ .

(3) ProcFormals = ⟦ ⟧ **|** ⟦ ProcFormal ⟧ **|** ⟦ ProcFormal "`,`" ProcFormals ⟧ .

(4) ProcFormal = ⟦ Identifier "`:`" ValueType ⟧ **|** ⟦ "`var`" Identifier "`:`" PrimitiveType ⟧ .

(5) FunFormals = ⟦ ⟧ **|** ⟦ FunFormal ⟧ **|** ⟦ FunFormal "`,`" FunFormals ⟧ .

(6) FunFormal = ⟦ Identifier "`:`" ValueType ⟧ .

2.3 Commands

(1) Command = ⟦ Identifier "`:=`" Expression ⟧ **|**
⟦ Identifier "`[`" Expression "`]`" "`:=`" Expression ⟧ **|**
⟦ "`if`" Expression "`then`" Command ⟨ "else" Command ⟩? "`end`" ⟧ **|**
⟦ "`while`" Expression "`do`" Command "`end`" ⟧ **|**
⟦ "`call`" Identifier "`(`" ProcActuals "`)`" ⟧ **|**
⟦ "`local`" Declaration "`in`" Command "`end`" ⟧ **|**
⟨ Command ⟨ "`;`" Command ⟩* ⟩ .

(2) ProcActuals = ⟦ ⟧ **|** ⟦ ProcActual ⟧ **|** ⟦ ProcActual "`,`" ProcActuals ⟧ .

(3) ProcActual = ⟦ Expression ⟧ **|** ⟦ "`var`" Identifier ⟧ .

2.4 Expressions

(1) Expression = "true" | "false" | Numeral | Identifier | ⟦ Identifier "[" Expression "]" ⟧ | ⟦ "if" Expression "then" Expression ⟨ "else" Expression ⟩? "end" ⟧ | ⟦ Identifier "(" FunActuals ")" ⟧ | ⟦ Expression Operator Expression ⟧ | ⟦ Operator Expression ⟧ ⟦ "let" Declaration "in" Expression "end" ⟧ .

(2) FunActuals = ⟦ ⟧ | ⟦ FunActual ⟧ | ⟦ FunActual "," FunActuals ⟧ .

(3) FunActual = ⟦ Expression ⟧ .

(4) Operator = "not" | "and" | "or" | "~" | "+" | "-" | "*" | "/" | "<" | ">" | "<=" | ">=" | "=" | "<>" .

2.5 Type Denoters

(1) ValueType = PrimitiveType | FunctionType .

(2) PrimitiveType = "bool" | "int" .

(3) FunctionType = ⟦ ValueType "->" ValueType ⟧ | ⟦ "(" TupleType ")" "->" ValueType ⟧ .

(4) TupleType = ⟦ ⟧ | ⟦ ValueType ⟧ | ⟦ ValueType "," TupleType ⟧ .

(5) ArrayType = ⟦ "array" "[" Numeral "]" "of" PrimitiveType ⟧ .

3 Semantic Entities

includes: **Action Notation** .

3.1 Values

introduces: primitive-value , value .

(1) primitive-value = truth-value | integer .

(2) value = primitive-value | function .

3.2 Bindings

(1) alpha = lowercase-letter | uppercase-letter .

(2) token = string-of (alpha,(alpha | digit)*) .

(3) bindable = value | procedure | cell | array .

3.3 Storage

(1) cell = cell [truth-value] | cell [integer] .

(2) storable = primitive-value .

3.4 Procedures and Functions

introduces: procedure , proc-argument , proc-argument-list , function , fun-argument , fun-argument-list , fun-result .

(1) procedure = abstraction .

(2) proc-argument = value | cell .

(3) proc-argument-list = list [proc-argument] .

(4) function = abstraction .

(5) fun-argument = value .

(6) fun-argument-list = list [fun-argument] .

(7) fun-result = value .

3.5 Arrays

introduces: array .

(1) array = list [cell] .

4 Semantic Functions

needs: **Abstract Syntax** , **Semantic Entities** .

4.1 Programs

introduces: run _ .

- run _ :: Program → action .

(1) run ⟦ "`program`" I:Identifier "`is`" D:Declaration "`in`" C:Command "`end`" ⟧ =
  | elaborate D
  hence
  | execute C .

4.2 Declarations

introduces: elaborate _ , elaborate-formals _ , elaborate-formal _ , evaluate-array-components _ .

4.2.1 Elaborating Declarations

- elaborate _ :: Declaration → action .

(1) elaborate ⟦ "`const`" I:Identifier "`:`" T:PrimitiveType "`=`" E:Expression ⟧ =
  evaluate E then bind token-of I to the primitive-value .

(2) elaborate ⟦ "`var`" I:Identifier "`:`" T:PrimitiveType "`:=`" E:Expression ⟧ =
  | | evaluate E then give the primitive-value label #1
  | and
  | | allocate-for-primitive-value T then give the cell label #2
  then
  | | bind token-of I to the cell#2
  | and
  | | store the primitive-value#1 in the cell#2 .

(3) elaborate ⟦ "varray" I:Identifier ":" T:ArrayType ":=" A:ArrayComps ⟧ =
| | evaluate-array-components A then give the list label #1
| and
| | allocate-for-array T then give the list label #2
then
| | unfolding
| | | | check (the list#1 is empty-list)
| | | and then
| | | | complete
| | or
| | | | check not (the list#1 is empty-list)
| | | and then
| | | | | store head-of (the list#1) in head-of (the list#2)
| | | | and then
| | | | | | | give tail-of (the list#1) label #1
| | | | | | and
| | | | | | | give tail-of (the list#2) label #2
| | | | | then
| | | | | | unfold
| and
| | bind token-of I to the list#2 .

(4) elaborate ⟦ "proc" I:Identifier "(" FS:ProcFormals ")" "=" C:Command ⟧ =
recursively bind token-of I to closure abstraction
| | furthermore elaborate-proc-formals FS
| hence
| | execute C .

(5) elaborate ⟦ "fun" I:Identifier "(" FS:FunFormals ")" ":" T:ValueType "=" E:Expression ⟧ =
recursively bind token-of I to closure abstraction
| | furthermore elaborate-fun–formals FS
| hence
| | evaluate E then give the fun-result .

(6) elaborate ⟦ D_1:Declaration ";" D_2:Declaration ⟧ =
| elaborate D_1
before
| elaborate D_2 .

4.2.2 Evaluating Array Components

- evaluate-array-components _ :: ArrayComps → action .

(1) evaluate-array-components ⟦ E:Expression ⟧ =
evaluate E then give list (primitive-value) .

(2) evaluate-array-components ⟦ E:Expression , A:ArrayComps ⟧ =
| | evaluate E then give list (primitive-value) label #1
| and
| | evaluate-array-components A then give the list label #2
then
| give concatenation (the list#1,the list#2) .

4.2.3 Elaborating Procedure Formal Parameters

- elaborate-proc-formals _ :: ProcFormals → action .

(1) elaborate-proc-formals ⟦ ⟧ = complete .

(2) elaborate-proc-formals ⟦ F:ProcFormal ⟧ =
give head-of (the proc-argument-list) then elaborate-proc-formal F .

(3) elaborate-proc-formals ⟦ F:ProcFormal "," FS:ProcFormals ⟧ =
| give head-of (the proc-argument-list) then elaborate-proc-formal F
and then
| give tail-of (the proc-argument-list) then elaborate-proc-formals FS .

4.2.4 Elaborating Procedure Formal Parameter

- elaborate-proc-formal _ :: ProcFormal → action .

(1) elaborate-proc-formal ⟦ I:Identifier ":" T:PrimitiveType ⟧ =
| | give the primitive-value label #1
| and
| | allocate-for-primitive-value T then give the cell label #2
then
| bind token-of I to the cell#2 and store the primitive-value#1 in the cell#2 .

(2) elaborate-proc-formal ⟦ I:Identifier ":" T:FunctionType ⟧ =
bind token-of I to the function .

(3) elaborate-proc-formal ⟦ "var" I:Identifier ":" T:Type ⟧ = bind token-of I to the cell .

4.2.5 Elaborating Function Formal Parameters

- elaborate-fun-formals _ :: FunFormals → action .

(1) elaborate-fun-formals ⟦ ⟧ = complete .

(2) elaborate-fun-formals ⟦ F:FunFormal ⟧ =
give head-of (the fun-argument-list) then elaborate-fun-formal F .

(3) elaborate-fun-formals ⟦ F:FunFormal "," FS:FunFormals ⟧ =
| give head-of (the fun-argument-list) then elaborate-fun-formal F
and then
| give tail-of (the fun-argument-list) then elaborate-fun-formals FS .

4.2.6 Elaborating Function Formal Parameter

- elaborate-fun-formal _ :: FunFormal → action .

(1) elaborate-fun-formal ⟦ I:Identifier ":" T:ValueType ⟧ =
bind token-of I to the fun-argument .

4.2.7 Allocating Storage

introduces: allocate-for-primitive-value _ , allocate-for-array _ .

- allocate-for-primitive-value _ :: PrimitiveType → action .

(1) allocate-for-primitive-value ⟦ "bool" ⟧ = allocate a truth-value-cell .

(2) allocate-for-primitive-value ⟦ "int" ⟧ = allocate an integer-cell .

- allocate-for-array _ :: ArrayType → action .

(3) allocate-for-array ⟦ “array” “[” N:Numeral “]” “of” T:PrimitiveType ⟧ =

```
| | give valuation-of N label #1
| and
| | give 1 label #2
| and
| | give empty-list label #3
then
| unfolding
| | | | | check not (the integer#1 is the integer#2)
| | | | and then
| | | | | | give the integer#1 label #1
| | | | | and
| | | | | | give successor (the integer#2) label #2
| | | | | and
| | | | | | | allocate-for-primitive-value T and give the list#3 label #1
| | | | | | then
| | | | | | | give concatenation (the list#1,list (the cell)) label #3
| | | then
| | | | unfold
| | or
| | | | | check (the integer#1 is the integer#2)
| | | | and then
| | | | | allocate-for-primitive-value T and give the list#3 label #1
| | | then
| | | | give concatenation(the list#1,list (the cell)) .
```

4.3 Commands

introduces: execute _ , evaluate-actuals _ , evaluate-actual _.

4.3.1 Executing Commands

- execute _ :: Command → action .

(1) execute ⟦ I:Identifier “:=” E:Expression ⟧ =

```
| evaluate E
then
| store the primitive-value in the cell bound to token-of I .
```

```
(2) execute ⟦ I:Identifier "[" E1:Expression "]" ":=" E2:Expression ⟧ =
      | | evaluate E1 then give the integer label #1
      | and
      | | evaluate E2 then give the primitive-value label #2
      | and
      | | give the list bound to token-of I label #3
      | and
      | | give 1 label #4
      then
      | unfolding
      | | | | check (the integer#1 is the integer#2)
      | | | and then
      | | | | store the primitive-value#2 in head-of (the list#3)
      | | or
      | | | | check not (the integer#1 is the integer#2)
      | | | and then
      | | | | | | give the integer#1 label #1
      | | | | | and
      | | | | | | give the primitive-value#2 label #2
      | | | | | and
      | | | | | | give tail-of (the list#3) label #3
      | | | | | and
      | | | | | | give successor (the integer#4) label #4
      | | | | then
      | | | | | unfold .

(3) execute ⟦ "if" E:Expression "then" C:Command "end" ⟧ =
      | evaluate E
      then
      | | execute C
      | else
      | | complete .

(4) execute ⟦ "if" E:Expression "then" C1:Command "else" C2:Command "end" ⟧ =
      | evaluate E
      then
      | | execute C1
      | else
      | | execute C2 .

(5) execute ⟦ "while" E:Expression "do" C:Command "end" ⟧ =
      unfolding
      | | evaluate E
      | then
      | | | execute C and then unfold
      | | else
      | | | complete .

(6) execute ⟦ "call" I:Identifier "(" AS:ProcActuals ")" ⟧ =
      | evaluate-proc-actuals AS
      then
      | enact (the procedure bound to token-of I with the proc-argument-list) .
```

(7) execute ⟦ “`local`” D:Declaration “`in`” C:Command “`end`” ⟧ =
| furthermore elaborate D
hence
| execute C .

(8) execute ⟦ C_1:Command “`;`” C_2:Command ⟧ = execute C_1 and then execute C_2 .

4.3.2 Evaluating Procedure Actual Parameters

- evaluate-proc-actuals _ :: ProcActuals → action .

(1) evaluate-proc-actuals ⟦ ⟧ = give empty-list .

(2) evaluate-proc-actuals ⟦ A:ProcActual ⟧ = evaluate-proc-actual A then give list (it) .

(3) evaluate-proc-actuals ⟦ A:ProcActual “`,`” AS:ProcActuals ⟧ =
| | evaluate-proc-actual A then give list (it) label #1
| and
| | evaluate-proc-actuals AS then give it label #2
then
| | give concatenation (the list#1,the list#2) .

4.3.3 Evaluating Procedure Actual Parameter

- evaluate-proc-actual _ :: ProcActual → action .

(1) evaluate-proc-actual ⟦ E:Expression ⟧ = evaluate E .

(2) evaluate-proc-actual ⟦ “`var`” I:Identifier ⟧ = give the cell bound to token-of I .

4.4 Expressions

introduces: evaluate _ , apply-operator _ .

4.4.1 Evaluating Expressions

- evaluate _ :: Expression → action .

(1) evaluate ⟦ “`true`” ⟧ = give true .

(2) evaluate ⟦ “`false`” ⟧ = give false .

(3) evaluate ⟦ N:Numeral ⟧ = give valuation-of N .

(4) evaluate ⟦ I:Identifier ⟧ =
| give the value bound to token-of I
or
| give the primitive-value stored in the cell bound to token-of I .

(5) evaluate ⟦ I:Identifier "[" E:Expression "]" ⟧ =
| | evaluate E then give the integer label #1
| and
| | give the list bound to token-of I label #2
| and
| | give 1 label #3
then
| unfolding
| | | | check (the integer#1 is the integer#3)
| | | and then
| | | | give the primitive-value stored in head-of (the list#2)
| | or
| | | | check not (the integer#1 is the integer#3)
| | | and then
| | | | | | give the integer#1 label #1
| | | | | and
| | | | | | give tail-of (the list#2) label #2
| | | | | and
| | | | | | give successor (the integer#3) label #3
| | | | then
| | | | | unfold .

(6) evaluate ⟦ "`if`" E_1:Expression "`then`" E_2:Expression "`end`" ⟧ =
| evaluate E_1
then
| | evaluate E_2
| else
| | complete .

(7) evaluate ⟦ "`if`" E_1:Expression "`then`" E_2:Expression "`else`" E_3:Expression "`end`" ⟧ =
| evaluate E_1
then
| | evaluate E_2
| else
| | evaluate E_3 .

(8) evaluate ⟦ I:Identifier "(" AS:FunActuals ")" ⟧ =
| evaluate-fun-actuals AS
then
| enact (the function bound to token-of I with the fun-argument-list) .

(9) evaluate ⟦ E_1:Expression O:Operator E_2:Expression ⟧ =
| | evaluate E_1 then give the value label #1
| and
| | evaluate E_2 then give the value label #2
then
| apply-operator O .

(10) evaluate ⟦ O:Operator E:Expression ⟧ =
| evaluate E then give the value
then
| apply-operator O .

(11) execute ⟦ "let" D:Declaration "in" E:Expression "end" ⟧ =
| furthermore elaborate D
hence
| evaluate E .

4.4.2 Evaluating Function Actual Parameters

- evaluate-fun-actuals _ :: FunActuals → action .

(1) evaluate-fun-actuals ⟦ ⟧ = give empty-list .

(2) evaluate-fun-actuals ⟦ A:Actual ⟧ = evaluate-fun-actual A then give list (it) .

(3) evaluate-fun-actuals ⟦ A:Actual "," AS:Actuals ⟧ =
| | evaluate-fun-actual A then give list (it) label #1
| and
| | evaluate-fun-actuals AS then give it label #2
then
| | give concatenation (the list#1,the list#2) .

4.4.3 Evaluating Function Actual Parameter

- evaluate-fun-actual _ :: FunActual → action .

(1) evaluate-fun-actual ⟦ E:Expression ⟧ = evaluate E .

4.4.4 Applying Operators

- apply-operator _ :: Operator → action .

(1) apply-operator ⟦ "not" ⟧ = give not (the truth-value) .

(2) apply-operator ⟦ "and" ⟧ = give both (the truth-value#1, the truth-value#2) .

(3) apply-operator ⟦ "or" ⟧ = give either (the truth-value#1, the truth-value#2) .

(4) apply-operator ⟦ "~" ⟧ = give negation (the integer) .

(5) apply-operator ⟦ "+" ⟧ = give sum (the integer#1, the integer#2) .

(6) apply-operator ⟦ "-" ⟧ = give difference (the integer#1, the integer#2) .

(7) apply-operator ⟦ "*" ⟧ = give product (the integer#1, the integer#2) .

(8) apply-operator ⟦ "/" ⟧ = give integer-quotient (the integer#1, the integer#2) .

(9) apply-operator ⟦ "<" ⟧ = give is-less-than (the integer#1, the integer#2) .

(10) apply-operator ⟦ ">" ⟧ = give is-greater-than (the integer#1, the integer#2) .

(11) apply-operator ⟦ "<=" ⟧ = give either (is-less-than (the integer#1, the integer#2),
the integer#1 is the integer#2) .

(12) apply-operator ⟦ ">=" ⟧ = give either (is-greater-than (the integer#1, the integer#2),
the integer#1 is the integer#2) .

(13) apply-operator ⟦ "=" ⟧ = give (the primitive-value#1 is the primitive-value#2) .

(14) apply-operator ⟦ "<>" ⟧ = give not (the primitive-value#1 is the primitive-value#2) .

5 Lexical Syntax

introduces: token-of _ , valuation-of _ .

closed

grammar:

(1) Identifier = letter **|** Identifier letter **|** Identifier digit .

(2) Numeral = digit **|** Numeral digit .

(3) digit = 0 **|** 1 **|** ... **|** 9 .

(4) letter = A **|** B **|** ... **|** Z **|** a **|** ... **|** z .

5.1 Identifiers

- token-of _ :: Identifier → token .

(1) token-of $[\![\; I \;]\!] = I$.

5.2 Numerals

- valuation-of _ :: Numeral → integer .

(1) valuation-of $[\![$ 0 $]\!]$ = 0 .

(2) valuation-of $[\![$ 1 $]\!]$ = 1 .

(3) valuation-of $[\![$ 2 $]\!]$ = 2 .

(4) valuation-of $[\![$ 3 $]\!]$ = 3 .

(5) valuation-of $[\![$ 4 $]\!]$ = 4 .

(6) valuation-of $[\![$ 5 $]\!]$ = 5 .

(7) valuation-of $[\![$ 6 $]\!]$ = 6 .

(8) valuation-of $[\![$ 7 $]\!]$ = 7 .

(9) valuation-of $[\![$ 8 $]\!]$ = 8 .

(10) valuation-of $[\![$ 9 $]\!]$ = 9 .

(11) valuation-of $[\![\; N \; D \;]\!]$ = sum (product (valuation-of N, 10), valuation-of D) .