The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 3 MODULE, AUTUMN SEMESTER 2013–2014

COMPILERS

ANSWERS

Time allowed TWO hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced.

Answer ALL THREE questions

No calculators are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject-specific translation directories are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

Note: ANSWERS
Note on Book Work: None of these questions constitute “book work” in the meaning that the answers can be found verbatim in one of the reference texts (there is no main textbook) or the lecture slides. However, a number of the questions are closely related to the coursework. This is intentional and as advertised to the students; the coursework is a central aspect of the module and as such partly examined under exam conditions.

Question 1

(a) Explain what a Scanner is in the context of compiler construction. Illustrate by a small example. (5)

Answer: A scanner groups consecutive individual characters into tokens (or lexemes) according to the lexical syntax for a language, thus transforming the representation of the program being compiled from a sequence of characters into a sequence of tokens (or lexemes). A scanner usually also discards white space and comments. For example, a scanner might transform the character sequence

```
v a r x : = 1 2 3
```

into the tokens:

- Keyword `var`
- Identifier `x`
- Symbol `:=`
- Integer literal `123`

(The symbol ` ` in the input sequence denotes a space.)
(b) Draw the abstract syntax tree for the following MiniTriangle program:

```
let
    proc p(var x : Integer) x := x * 2;
    var y : Integer := 21
in
    p(y)
```

See Appendix A for the MiniTriangle grammar relevant to this question. Start from the production for `Command`. (10)

**Answer:** Abstract syntax tree. Variable spelling terminals have been annotated with their spellings. These annotations are typeset in typewriter font.

![Abstract Syntax Tree](image_url)
(c) Consider the following context-free grammar (CFG):

\[
S \rightarrow aAb \mid cS \mid B \\
A \rightarrow eAB \mid Af \mid g \\
B \rightarrow d
\]

\(S, A\) and \(B\) are nonterminal symbols, \(S\) is the start symbol, and \(a, b, c, d, e, f,\) and \(g\) are terminal symbols.

The DFA below recognizes the viable prefixes for this CFG:

Show how an LR(0) shift-reduce parser parses the string \(cccaeegffddb\) by completing the following table (copy it to your answer book; do not write on the examination paper):

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>Input</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>(\epsilon)</td>
<td>(cccaeegffddb)</td>
<td>Shift</td>
</tr>
<tr>
<td>17</td>
<td>(c)</td>
<td>(cccaeegffddb)</td>
<td>Shift</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>\vdots</td>
</tr>
<tr>
<td>(S)</td>
<td>(\epsilon)</td>
<td></td>
<td>Done</td>
</tr>
</tbody>
</table>
Answer:

<table>
<thead>
<tr>
<th>State</th>
<th>Stack</th>
<th>Input</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>ε</td>
<td>cccaeegffddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I7</td>
<td>c</td>
<td>cccaeegffddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I7</td>
<td>cc</td>
<td>cccaeegffddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I7</td>
<td>cccc</td>
<td>cccaeegffddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I1</td>
<td>cccca</td>
<td>eegffddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I6</td>
<td>cccae</td>
<td>egffddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I6</td>
<td>cccae</td>
<td>gffddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I2</td>
<td>cccaeeg</td>
<td>ffddb</td>
<td>Reduce by $A \rightarrow g$</td>
</tr>
<tr>
<td>I10</td>
<td>cccae.A</td>
<td>ffddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I9</td>
<td>cccae.Af</td>
<td>fddb</td>
<td>Reduce by $A \rightarrow Af$</td>
</tr>
<tr>
<td>I10</td>
<td>cccae.Af</td>
<td>fddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I9</td>
<td>cccae.Af</td>
<td>ddb</td>
<td>Reduce by $A \rightarrow Af$</td>
</tr>
<tr>
<td>I10</td>
<td>cccae.A</td>
<td>ddb</td>
<td>Shift</td>
</tr>
<tr>
<td>I4</td>
<td>cccae.Ad</td>
<td>db</td>
<td>Reduce by $B \rightarrow d$</td>
</tr>
<tr>
<td>I12</td>
<td>cccae.AB</td>
<td>db</td>
<td>Reduce by $A \rightarrow eAB$</td>
</tr>
<tr>
<td>I10</td>
<td>cccae.A</td>
<td>db</td>
<td>Shift</td>
</tr>
<tr>
<td>I4</td>
<td>cccae.Ad</td>
<td>b</td>
<td>Reduce by $B \rightarrow d$</td>
</tr>
<tr>
<td>I12</td>
<td>cccae.AB</td>
<td>b</td>
<td>Reduce by $A \rightarrow eAB$</td>
</tr>
<tr>
<td>I5</td>
<td>cccA</td>
<td>b</td>
<td>Shift</td>
</tr>
<tr>
<td>I8</td>
<td>cccA.b</td>
<td>ε</td>
<td>Reduce by $S \rightarrow aAb$</td>
</tr>
<tr>
<td>I11</td>
<td>cccS</td>
<td>ε</td>
<td>Reduce by $S \rightarrow cS$</td>
</tr>
<tr>
<td>I11</td>
<td>ccS</td>
<td>ε</td>
<td>Reduce by $S \rightarrow cS$</td>
</tr>
<tr>
<td>I11</td>
<td>cS</td>
<td>ε</td>
<td>Reduce by $S \rightarrow cS$</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td>ε</td>
<td>Done</td>
</tr>
</tbody>
</table>
Question 2
See Appendix A for the MiniTriangle grammars relevant to this question.

(a) The following is a Haskell datatype definition for representing the abstract syntax of a selection of MiniTriangle commands. The type `Expression` represents the abstract syntax of expressions.

```haskell
data Command = CmdAssign Expression Expression
  | CmdIf Expression Command Command
  | CmdWhile Expression Command
  | CmdSeq [Command]
```

A Happy parser specification dealing with commands and sequences of commands is given below. The semantic actions for constructing an abstract syntax tree (AST) have been left out (indicated by a boxed number, like \[3\]). Complete the specification by providing semantic actions for constructing an AST. The type of the semantic values of the non-terminals `var_expression` and `expression` is `Expression`.

```haskell
commands :: { [Command] }
commands : command { 1 } 
  | command ';' commands { 2 }

command :: { Command }
command :
  var_expression ':'= expression { 3 }
  | IF expression THEN command ELSE command { 4 }
  | WHILE expression DO command { 5 }
  | BEGIN commands END { 6 }

Answer:

\[ \begin{align*}
1 &= \text{[$1]} \\
2 &= \$1 : \$3 \\
3 &= \text{CmdAssign $1 $3} \\
4 &= \text{CmdIf $2 $4 $6} \\
5 &= \text{CmdWhile $2 $4} \\
6 &= \text{CmdSeq $2} \\
\end{align*} \]

Marking: 1 marks for each semantic action. \((6 \times 1 = 6)\)
(b) Suppose we wish to extend MiniTriangle with a for-loop (a new command). The following two code fragments illustrate the idea:

- for i from 1 to 10 do \( x[i] := i \times i \)
- for j from 2 * m to n step -2 do \( \text{sum} := \text{sum} + j \)

The for-loop has the following semantics. The expressions defining the start, end, and step are evaluated exactly once. The loop variable is initialised to the value given by the expression following the keyword from. The loop body is then repeated 0 or more times, incrementing (if positive step size) or decrementing (if negative step size) the loop variable after each execution of the body until the value of the loop variable is greater (positive step size) or smaller (negative step size) than the value of the expression following the keyword to. Note that the step size is optional. If left out, it should default to 1. Thus, in the first example, \( i \) will assume the values 1, 2, ..., 10 in that order, with the loop body \( x[i] := i \times i \) executed once for each assignment.

(i) Extend the MiniTriangle lexical and concrete syntax with new productions defining the syntax of the for-loop. Pick the syntactic categories for the constituent parts with care: your extended grammars should be reasonably general, and in particular general enough to accept both examples above.

Answer: The following productions need to be added to the lexical grammar:

- **Keyword** → for | from | step | to

And the following is one way to extend the concrete grammar:

- **Command** → for **VarExpression** from **Expression**
- **to** **Expression** **OptStep** do **Command**

(iii) Extend the type Command with a new constructor for representing for-loops. Then show how to extend the Happy parser specification so that the new construct is accepted and a corresponding AST gets constructed. You may assume that all extensions related to the lexical syntax, including extending the scanner, have already been carried out.

Answer: Abstract syntax extension:

- data Command = ...
- | CmdFor **Expression** **Expression** **Expression**

Extension of the parser specification:

Turn Over
command ::= { Command }

command
   : ...
   | FOR var_expression FROM expression TO expression
     opt_step DO command
     { CmdFor $2 $4 $6 $7 $9 }

opt_step ::= { Maybe Expression }

opt_step
   : { - epsilon - } { Nothing }
   | STEP expression { (Just $2) }

An alternative, as we know that the default of an omitted STEP is 1, is to represent the for-loop without making use of the maybe type:

data Command = ...
   | CmdFor Expression Expression Expression Expression Command

The parser is extended as before, except that the productions for opt_step instead are defined as follows:

opt_step ::= { Expression }

opt_step
   : { - epsilon - } { ExpLitInt 1 }
   | STEP expression { $2 }  

(Only one variant is needed for full marks, of course)

(c) Write the case(s) of a code-generation function execute for generating code for the for-loop, targetting the Triangle Abstract Machine (TAM). See appendix B for a specification of the TAM instructions. The code generation function should be specified through code templates in the style used in the lectures. Thus, for the case without the optional step size, something along the lines

\[
\text{execute } n \left[ \text{for } E_x \text{ from } E_l \text{ to } E_t \text{ do } C \right] = \ldots
\]

where \( n \) is the current stack depth.

Assume a code-generation function evaluate (which does not need the current stack depth as expressions do not introduce new variables) for generating code for expressions, leaving the value of the expression on the top of the stack. Assume further that calling evaluate on the expression corresponding to the loop variable generates code that leaves the address of the variable on the stack (for use by instructions such
as \texttt{LOADI} and \texttt{STOREI}). Call \textit{execute} recursively for commands. Generation of fresh labels need not be considered; it suffices that labels are distinct within each case of the code function. (Also, there is no need to consider environments for mapping identifiers to addresses etc.) Take care to only generate code for the body once. (10)
**Answer:** The following cases generate code for the *for*-loop:

execute \( n \) \([\text{for } E_x \text{ from } E_t \text{ to } E_i \text{ do } C]\) =

execute \( n \) \([\text{for } E_x \text{ from } E_1 \text{ to } E_2 \text{ step } 1 \text{ do } C]\)

execute \( n \) \([\text{for } E_x \text{ from } E_t \text{ to } E_i \text{ step } E_s \text{ do } C]\) =

\[
\begin{align*}
\text{evaluate } [E_x] \\
\text{evaluate } [E_t] \\
\text{LOAD } [\text{ST} - 2] \\
\text{STOREI } 0 \\
\text{evaluate } [E_i] \\
\text{evaluate } [E_s]
\end{align*}
\]

loop:

\[
\begin{align*}
\text{LOAD } [\text{ST} - 3] \\
\text{LOADI } 0 \\
\text{LOAD } [\text{ST} - 3] \\
\text{LOAD } [\text{ST} - 3] \\
\text{LOADL } 0 \\
\text{LSS} \\
\text{JUMPIFNZ} \text{ negstep} \\
\text{GTR} \\
\text{JUMPIFNZ} \text{ out} \\
\text{JUMP} \text{ body}
\end{align*}
\]

negstep:

\[
\begin{align*}
\text{LSS} \\
\text{JUMPIFNZ} \text{ out}
\end{align*}
\]

body:

\[
\begin{align*}
\text{execute } (n + 3) [C] \\
\text{LOAD } [\text{ST} - 3] \\
\text{LOADI } 0 \\
\text{LOAD } [\text{ST} - 2] \\
\text{ADD} \\
\text{LOAD } [\text{ST} - 4] \\
\text{STOREI } 0 \\
\text{JUMP} \text{ loop}
\end{align*}
\]

out:

\[
\begin{align*}
\text{POP } 0 3
\end{align*}
\]
Question 3
This questions concerns types and scope: both how they are captured formally in a type system, and how they might be implemented.

(a) Consider the following expression language:

\[
\begin{align*}
e & \rightarrow \quad \text{expressions:} \\
| & n \quad \text{natural numbers, } n \in \mathbb{N} \\
| & x \quad \text{variables, } x \in \text{Name} \\
| & e = e \quad \text{equality test} \\
| & \text{if } e \text{ then } e \text{ else } e \quad \text{conditional}
\end{align*}
\]

where Name is the set of variable names. The types are given by the following grammar:

\[
\begin{align*}
t & \rightarrow \quad \text{types:} \\
| & \text{Nat} \quad \text{natural numbers} \\
| & \text{Bool} \quad \text{Booleans}
\end{align*}
\]

The ternary relation \( \Gamma \vdash e : t \) says that expression \( e \) has type \( t \) in the typing context \( \Gamma \). It is defined by the following typing rules:

\[
\begin{align*}
\Gamma \vdash n : \text{Nat} & \quad (\text{T-NAT}) \\
\frac{x : t \in \Gamma}{\Gamma \vdash x : t} & \quad (\text{T-VAR}) \\
\frac{\Gamma \vdash e_1 : \text{Nat} \quad \Gamma \vdash e_2 : \text{Nat}}{\Gamma \vdash e_1 = e_2 : \text{Bool}} & \quad (\text{T-EQ}) \\
\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t} & \quad (\text{T-COND})
\end{align*}
\]

A typing context, \( \Gamma \) in the rules above, is a comma-separated sequence of variable-name and type pairs, such as

\[x : \text{Nat}, y : \text{Bool}, z : \text{Nat}\]

or empty, denoted \( \emptyset \). Typing contexts are extended on the right, e.g. \( \Gamma, z : \text{Nat} \), the membership predicate is denoted by \( \in \), and lookup is from right to left, ensuring recent bindings hide earlier ones.
(i) Use the typing rules given above to formally prove that the expression

\[ \text{if } x = 5 \text{ then } a \text{ else } b \]

has type \( \text{Bool} \) in the typing context

\[ \Gamma_1 = a : \text{Bool}, b : \text{Bool}, x : \text{Nat} \]

The proof should be given as a proof tree. (5)

**Answer:**

\[
\begin{array}{c}
\Gamma_1 \vdash x = 5 : \text{Bool} \\
\hline
a : \text{Bool} \in \Gamma_1 \quad b : \text{Bool} \in \Gamma_1 \\
\hline
\Gamma_1 \vdash \text{if } x = 5 \text{ then } a \text{ else } b : \text{Bool}
\end{array}
\]

\[
\begin{array}{c}
x : \text{Nat} \in \Gamma_1 \\
\hline
\Gamma_1 \vdash x : \text{Nat}
\end{array}
\]

\[
\begin{array}{c}
\Gamma_1 \vdash 5 : \text{Nat} \\
\hline
\Gamma_1 \vdash x = 5 : \text{Bool}
\end{array}
\]

(ii) The expression language defined above is to be extended with let-bound variables; definition of named, possibly recursive, functions; and function application as follows:

\[
e \rightarrow \quad \text{expressions:}
\]

\[
\begin{array}{c}
\ldots \ldots \\
| \text{let var } x = e \text{ in } e \quad \text{variable definition} \\
| \text{let fun } f(x : t) : t = e \text{ in } e \quad \text{function definition} \\
| e(e) \quad \text{function application}
\end{array}
\]

\[
t \rightarrow \quad \text{types:}
\]

\[
\begin{array}{c}
\ldots \ldots \\
| t \rightarrow t \quad \text{function (arrow) type}
\end{array}
\]

Here, \( f \) is the syntactic category of function names (\( f \in \text{Name} \)). Variable definition is not recursive: the let-bound variable is only in scope in the body of the let-expression, not in its defining expression. In contrast, the named function being defined is in scope, along with the named formal argument, in the expression defining the function, thus allowing for recursive functions.

For example, if we assume that the expression language has been extended with basic arithmetic operations as well, the following is a definition of the factorial function:

\[
\text{let fun fac(n : Nat) : Nat =}
\]

\[
\quad \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fac}(n - 1)
\]

\[
\text{in}
\]

\[
\ldots
\]
Provide a typing rule for each of the new expression constructs, in the same style as the existing rules, reflecting the standard notions of typed let-expressions and function application augmented by the additional requirements set forth in the text above. (8)

Answer:

\[
\begin{align*}
\Gamma \vdash e_1 : t_1, \quad & \quad \Gamma, x : t_1 \vdash e_2 : t_2 \\
\Gamma \vdash \text{let} \ var \ x = e_1 \ \text{in} \ e_2 : t_2 & \quad \text{(T-LETVAR)} \\
\Gamma, f : t_{11} \to t_{12}, \quad & \quad \Gamma, f : t_{11} \to t_{12} \vdash e_2 : t_2 \\
\Gamma \vdash e_1 : t_{11}, \quad & \quad \Gamma \vdash e_2 : t_2 \\
\Gamma \vdash \text{let} \ \text{fun} \ f \ ((x : t_{11}) : t_{12} = e_1 \ \text{in} \ e_2 : t_2 \\
\Gamma \vdash e_1 : t_2 \to t_1, \quad & \quad \Gamma \vdash e_2 : t_2 \\
\Gamma \vdash e_1 \ e_2 : t_1 & \quad \text{(T-LETFUN)} \\
\end{align*}
\]
(b) Consider the following code skeleton (note: nested procedures):

```plaintext
var a, b, c: Integer
proc P
    var x, y, z: Integer
    proc Q
        var u, v: Bool
        proc R
            var w: Bool
            begin ... Q() ... end
            begin ... R() ... end
            begin ... Q() ... end
            begin ... P() ... end
begin ...
```

The variables a, b, and c are global. The variables x, y, and z are local to procedure P, as is procedure Q, which in turn has two local variables, u and v, and a local procedure R. The latter has one local variable, w.

The notation P(), Q(), etc. signifies a call to the named procedure. Thus main calls P, P calls Q, Q calls R, and R calls Q (recursively).

Assume stack-based memory allocation with dynamic and static links.

(i) Show the layout of the activation records on the stack after the main program has called procedure P. Explain how global and local variables are accessed from P. (3)

(ii) Show the layout of the activation records on the stack after the call sequence: P, Q, R, Q, R (that is, after main has called P, which in turn has called Q, etc.). Explain how global variables, P’s variables, Q’s variables, and R’s own local variables are accessed from the last activation of R. (9)

**Answer:**

(i) **Activation record layout (stack grows downwards):**

```
Global variables   SB --> a 
                   |   b 
                   |   c 
Frame of P         LB --> 
                   |   static link 
                   |   dynamic link 
                   |   return address 
                   |   x 
                   |   y 
                   |   z 
ST --> 
```

SB is Stack Base, LB is Local Base (or Frame Pointer), ST is Stack Top. The activation record (or frame) of the currently active procedure/function is the one between LB and ST. The solid arrow from the current activation record represents the dynamic link, i.e. it refers to the activation record of the caller and is thus
equal to the previous value of LB. The dashed arrow represents the static link. Global variables are accessed relative to SB. Variables local to P are accessed relative to LB. In both cases, the offsets are statically known.

(ii) Activation record layout (stack grows downwards):

<table>
<thead>
<tr>
<th>Global variables</th>
<th>SB →</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame of P</td>
<td></td>
</tr>
<tr>
<td>Frame of Q (1)</td>
<td></td>
</tr>
<tr>
<td>Frame of R (1)</td>
<td></td>
</tr>
<tr>
<td>Frame of Q (2)</td>
<td></td>
</tr>
<tr>
<td>Frame of R (2)</td>
<td></td>
</tr>
<tr>
<td>ST →</td>
<td></td>
</tr>
</tbody>
</table>

SB is Stack Base, LB is Local Base (or Frame Pointer), ST is Stack Top. The solid arrows represent the dynamic link, the dashed ones is the static link. Global variables are accessed relative to SB. Variables local to R are are accessed relative to LB. Hence the second instance of these variables are being accessed (corresponding to the currently active procedure). As Q directly encloses R, following the static link from R’s activation record takes us the the correct activation record, the most recent activation of Q, and the variables can then be found at statically known offsets within this activation record. As P is two scope levels out, we have to follow two static links: first the static link from R’s activation record to get to the activation record for the immediately enclosing scope, and then the static link from that record to get to the record two levels out. The variables are again found at statically known offsets within this record.
Appendix A: MiniTriangle Grammars

This appendix contains the grammars for the MiniTriangle lexical, concrete, and abstract syntax. The following typographical conventions are used to distinguish between terminals and non-terminals:

- nonterminals are written like this
- terminals are written like this
- terminals with variable spelling and special symbols are written like this

**MiniTriangle Lexical Syntax:**

\[
\begin{align*}
\text{Program} & \rightarrow (\text{Token} \mid \text{Separator})^* \\
\text{Token} & \rightarrow \text{Keyword} \mid \text{Identifier} \mid \text{IntegerLiteral} \mid \text{Operator} \\
& \mid , \mid ; \mid : \mid := \mid = \mid ( \mid ) \mid [ \mid ] \mid \text{col} \\
\text{Keyword} & \rightarrow \text{begin} \mid \text{const} \mid \text{do} \mid \text{else} \mid \text{end} \mid \text{fun} \mid \text{if} \mid \text{in} \\
& \mid \text{let} \mid \text{out} \mid \text{proc} \mid \text{then} \mid \text{var} \mid \text{while} \\
\text{Identifier} & \rightarrow \text{Letter} \mid \text{Identifier Letter} \mid \text{Identifier Digit} \\
& \text{except Keyword} \\
\text{IntegerLiteral} & \rightarrow \text{Digit} \mid \text{IntegerLiteral Digit} \\
\text{Operator} & \rightarrow ^ \mid * \mid / \mid + \mid - \mid < \mid <= \mid != \mid >= \mid > \mid \&\& \mid || \mid ! \\
\text{Letter} & \rightarrow \text{A} \mid \text{B} \mid \ldots \mid \text{Z} \mid \text{a} \mid \text{b} \mid \ldots \mid \text{z} \\
\text{Digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\text{Separator} & \rightarrow \text{Comment} \mid \text{space} \mid \text{col} \\
\text{Comment} & \rightarrow // (\text{any character except col})* \text{col}
\end{align*}
\]
MiniTriangle Concrete Syntax:

\[\text{Program} \rightarrow \text{Command}\]

\[\text{Commands} \rightarrow \text{Command} \mid \text{Command} ; \text{Commands}\]

\[\text{Command} \rightarrow \text{VarExpression} := \text{Expression} \mid \text{VarExpression} (\text{Expressions}) \mid \text{if} \ \text{Expression} \ \text{then} \ \text{Command} \ \text{else} \ \text{Command} \mid \text{while} \ \text{Expression} \ \text{do} \ \text{Command} \mid \text{let} \ \text{Declarations} \ \text{in} \ \text{Command} \mid \text{begin} \ \text{Commands} \ \text{end}\]

\[\text{Expressions} \rightarrow \epsilon \mid \text{Expressions}_1\]

\[\text{Expressions}_1 \rightarrow \text{Expression} \mid \text{Expression}, \text{Expressions}_1\]

\[\text{Expression} \rightarrow \text{PrimaryExpression} \mid \text{Expression} \ \text{BinaryOperator} \ \text{Expression}\]

\[\text{PrimaryExpression} \rightarrow \text{IntegerLiteral} \mid \text{VarExpression} \mid \text{UnaryOperator} \ \text{PrimaryExpression} \mid \text{VarExpression} (\text{Expressions}) \mid [\text{Expressions}] \mid (\text{Expression})\]

\[\text{VarExpression} \rightarrow \text{Identifier} \mid \text{VarExpression} [\text{Expression}]\]

\[\text{BinaryOperator} \rightarrow ^| * | / | + | - | < | <= | == | != | >= | > | && | ||\]

\[\text{UnaryOperator} \rightarrow - | !\]
\textbf{Declarations} \rightarrow \textit{Declaration} \\
| \textit{Declaration} ; \textit{Declarations} \\

\textit{Declaration} \rightarrow \textit{const} \textit{Identifier} : \textit{TypeDenoter} = \textit{Expression} \\
| \textit{var} \textit{Identifier} : \textit{TypeDenoter} \\
| \textit{fun} \textit{Identifier} ( \textit{ArgDecls} ) : \textit{TypeDenoter} = \textit{Expression} \\
| \textit{proc} \textit{Identifier} ( \textit{ArgDecls} ) \textit{Command} \\

\textit{Declarations} \rightarrow \epsilon \\
| \textit{ArgDecls}_1 \\

\textit{ArgDecls}_1 \rightarrow \textit{ArgDecl} \\
| \textit{ArgDecl} , \textit{ArgDecls}_1 \\

\textit{ArgDecl} \rightarrow \textit{Identifier} : \textit{TypeDenoter} \\
| \textit{in} \textit{Identifier} : \textit{TypeDenoter} \\
| \textit{out} \textit{Identifier} : \textit{TypeDenoter} \\
| \textit{var} \textit{Identifier} : \textit{TypeDenoter} \\

\textit{TypeDenoter} \rightarrow \textit{Identifier} \\
| \textit{TypeDenoter} [ \textit{IntegerLiteral} ] \\

Note that the productions for \textit{Expression} makes the grammar as stated above ambiguous. Operator precedence and associativity for the binary operators as defined in the following table is used to disambiguate:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>1</td>
<td>right</td>
</tr>
<tr>
<td>* /</td>
<td>2</td>
<td>left</td>
</tr>
<tr>
<td>+ -</td>
<td>3</td>
<td>left</td>
</tr>
<tr>
<td>&lt; &lt;= == != &gt;= &gt;</td>
<td>4</td>
<td>non</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>5</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A precedence level of 1 means the highest precedence, 2 means second highest, and so on.
MiniTriangle Abstract Syntax: \( \text{Name} = \text{Identifier} \cup \text{Operator} \)

\[
\begin{align*}
\text{Program} & \quad \rightarrow \quad \text{Command} \\
\text{Command} & \quad \rightarrow \quad \text{Expression} \; \diamond \; \text{Expression} \quad \text{CmdAssign} \\
& \quad \mid \quad \text{Expression} \; (\; \text{Expression}^* \; ) \quad \text{CmdCall} \\
& \quad \mid \quad \text{begin} \; \text{Command}^* \; \text{end} \quad \text{CmdSeq} \\
& \quad \mid \quad \text{if} \; \text{Expression} \; \text{then} \; \text{Command} \quad \text{CmdIf} \\
& \quad \mid \quad \text{else} \; \text{Command} \quad \text{CmdElse} \\
& \quad \mid \quad \text{while} \; \text{Expression} \; \text{do} \; \text{Command} \quad \text{CmdWhile} \\
& \quad \mid \quad \text{let} \; \text{Declaration}^* \; \text{in} \; \text{Command} \quad \text{CmdLet} \\
\text{Expression} & \quad \rightarrow \quad \text{IntegerLiteral} \quad \text{ExpLitInt} \\
& \mid \quad \text{Name} \quad \text{ExpVar} \\
& \quad \mid \quad \text{Expression} \; (\; \text{Expression}^* \; ) \quad \text{ExpApp} \\
& \quad \mid \quad [\; \text{Expression}^* \; ] \quad \text{ExpAry} \\
& \quad \mid \quad \text{Expression} \; [\; \text{Expression} \; ] \quad \text{ExpIx} \\
\text{Declaration} & \quad \rightarrow \quad \text{const} \; \text{Name} \; : \; \text{TypeDenoter} \quad \text{DeclConst} \\
& \quad \mid \quad \text{var} \; \text{Name} \; : \; \text{TypeDenoter} \quad \text{DeclVar} \\
& \quad \mid \quad (\; := \; \text{Expression} \; | \; \epsilon \; ) \quad \text{DeclVar} \\
& \quad \mid \quad \text{fun} \; \text{Name} \; (\; \text{ArgDecl}^* \; ) \quad \text{DeclFun} \\
& \quad \mid \quad : \; \text{TypeDenoter} \; = \; \text{Expression} \quad \text{DeclFun} \\
& \quad \mid \quad \text{proc} \; \text{Name} \; (\; \text{ArgDecl}^* \; ) \; \text{Command} \quad \text{DeclProc} \\
\text{ArgDecl} & \quad \rightarrow \quad \text{ArgMode} \; \text{Name} \; : \; \text{TypeDenoter} \quad \text{ArgDecl} \\
\text{ArgMode} & \quad \rightarrow \quad \epsilon \quad \text{ByValue} \\
& \mid \quad \text{in} \quad \text{ByRefIn} \\
& \quad \mid \quad \text{out} \quad \text{ByRefOut} \\
& \quad \mid \quad \text{var} \quad \text{ByRefVar} \\
\text{TypeDenoter} & \quad \rightarrow \quad \text{Name} \quad \text{TDBaseType} \\
& \quad \rightarrow \quad \text{TypeDenoter} \; [\; \text{IntegerLiteral} \; ] \quad \text{TDArry}
\end{align*}
\]
Appendix B: Triangle Abstract Machine (TAM) Instructions

<table>
<thead>
<tr>
<th>Meta variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>Address: one of the forms specified by table below when part of an instruction, specific stack address when on the stack</td>
</tr>
<tr>
<td>$b$</td>
<td>Boolean value (false = 0 or true = 1)</td>
</tr>
<tr>
<td>$ca$</td>
<td>Code address; address to routine in the code segment</td>
</tr>
<tr>
<td>$d$</td>
<td>Displacement; i.e., offset w.r.t. address in register or on the stack</td>
</tr>
<tr>
<td>$l$</td>
<td>Label name</td>
</tr>
<tr>
<td>$m$, $n$</td>
<td>Integer</td>
</tr>
<tr>
<td>$x$, $y$</td>
<td>Any kind of stack data</td>
</tr>
<tr>
<td>$x^n$</td>
<td>Vector of $n$ items, in this case any kind</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[SB + d]$</td>
<td>Address given by contents of register SB</td>
</tr>
<tr>
<td>$[SB - d]$</td>
<td>(Stack Base) $+/-$ displacement $d$</td>
</tr>
<tr>
<td>$[LB + d]$</td>
<td>Address given by contents of register LB</td>
</tr>
<tr>
<td>$[LB - d]$</td>
<td>(Local Base) $+/-$ displacement $d$</td>
</tr>
<tr>
<td>$[ST + d]$</td>
<td>Address given by contents of register ST</td>
</tr>
<tr>
<td>$[ST - d]$</td>
<td>(Stack Top) $+/-$ displacement $d$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL $l$</td>
<td>—</td>
<td>Pseudo instruction: symbolic location</td>
</tr>
</tbody>
</table>

Load and store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOADL $n$</td>
<td>$\ldots \Rightarrow n, \ldots$</td>
<td>Push literal integer $n$ onto stack</td>
</tr>
<tr>
<td>LOADCA $l$</td>
<td>$\ldots \Rightarrow \text{addr}(l), \ldots$</td>
<td>Push address of label $l$ (code segment) onto stack</td>
</tr>
<tr>
<td>LOAD $a$</td>
<td>$\ldots \Rightarrow [a], \ldots$</td>
<td>Push contents at address $a$ onto stack</td>
</tr>
<tr>
<td>LOADA $a$</td>
<td>$\ldots \Rightarrow a, \ldots$</td>
<td>Push address $a$ onto stack</td>
</tr>
<tr>
<td>LOADI $d$</td>
<td>$a, \ldots \Rightarrow [a + d], \ldots$</td>
<td>Load indirectly; push contents at address $a + d$ onto stack</td>
</tr>
<tr>
<td>STORE $a$</td>
<td>$n, \ldots \Rightarrow \ldots$</td>
<td>Pop value $n$ from stack and store at address $a$</td>
</tr>
<tr>
<td>STOREI $d$</td>
<td>$a, n, \ldots \Rightarrow \ldots$</td>
<td>Store indirectly; store $n$ at address $a + d$</td>
</tr>
<tr>
<td>Instruction</td>
<td>Stack effect</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Block operations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LOADLB</strong> m n</td>
<td>( \ldots \Rightarrow m^n, \ldots )</td>
<td>Push block of ( n ) literal integers ( m ) onto stack</td>
</tr>
<tr>
<td><strong>LOADIB</strong> n</td>
<td>( a, \ldots \Rightarrow [a + (n - 1)], \ldots, [a + 0], \ldots )</td>
<td>Load block of size ( n ) indirectly</td>
</tr>
<tr>
<td><strong>STOREIB</strong> n</td>
<td>( a, x^n, \ldots \Rightarrow \ldots )</td>
<td>Store block of size ( n ) indirectly</td>
</tr>
<tr>
<td><strong>POP</strong> m n</td>
<td>( x^m, y^n, \ldots \Rightarrow x^m, \ldots )</td>
<td>Pop ( n ) values below top ( m ) values</td>
</tr>
<tr>
<td><strong>Arithmetic operations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ADD</strong></td>
<td>( n_2, n_1, \ldots \Rightarrow n_1 + n_2, \ldots )</td>
<td>Add ( n_1 ) and ( n_2 ), replacing ( n_1 ) and ( n_2 ) with the sum</td>
</tr>
<tr>
<td><strong>SUB</strong></td>
<td>( n_2, n_1, \ldots \Rightarrow n_1 - n_2, \ldots )</td>
<td>Subtract ( n_2 ) from ( n_1 ), replacing ( n_1 ) and ( n_2 ) with the difference</td>
</tr>
<tr>
<td><strong>MUL</strong></td>
<td>( n_2, n_1, \ldots \Rightarrow n_1 \cdot n_2, \ldots )</td>
<td>Multiply ( n_1 ) by ( n_2 ), replacing ( n_1 ) and ( n_2 ) with the product</td>
</tr>
<tr>
<td><strong>DIV</strong></td>
<td>( n_2, n_1, \ldots \Rightarrow n_1/n_2, \ldots )</td>
<td>Divide ( n_1 ) by ( n_2 ), replacing ( n_1 ) and ( n_2 ) with the (integer) quotient</td>
</tr>
<tr>
<td><strong>NEG</strong></td>
<td>( n, \ldots \Rightarrow -n, \ldots )</td>
<td>Negate ( n ), replacing ( n ) with the result</td>
</tr>
<tr>
<td><strong>Comparison &amp; logical operations</strong></td>
<td>(false = 0, true = 1)</td>
<td></td>
</tr>
<tr>
<td><strong>LSS</strong></td>
<td>( n_2, n_1, \ldots \Rightarrow n_1 &lt; n_2, \ldots )</td>
<td>Check if ( n_1 ) is smaller than ( n_2 ), replacing ( n_1 ) and ( n_2 ) with the Boolean result</td>
</tr>
<tr>
<td><strong>EQL</strong></td>
<td>( n_2, n_1, \ldots \Rightarrow n_1 = n_2, \ldots )</td>
<td>Check if ( n_1 ) is equal to ( n_2 ), replacing ( n_1 ) and ( n_2 ) with the Boolean result</td>
</tr>
<tr>
<td><strong>GTR</strong></td>
<td>( n_2, n_1, \ldots \Rightarrow n_1 &gt; n_2, \ldots )</td>
<td>Check if ( n_1 ) is greater than ( n_2 ), replacing ( n_1 ) and ( n_2 ) with the Boolean result</td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>( b_2, b_1, \ldots \Rightarrow b_1 \land b_2, \ldots )</td>
<td>Logical conjunction of ( b_1 ) and ( b_2 ), replacing ( b_1 ) and ( b_2 ) with the Boolean result</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>( b_2, b_1, \ldots \Rightarrow b_1 \lor b_2, \ldots )</td>
<td>Logical disjunction of ( b_1 ) and ( b_2 ), replacing ( b_1 ) and ( b_2 ) with the Boolean result</td>
</tr>
<tr>
<td><strong>NOT</strong></td>
<td>( b, \ldots \Rightarrow \neg b, \ldots )</td>
<td>Logical negation of ( b ), replacing ( b ) with the result</td>
</tr>
<tr>
<td>Instruction</td>
<td>Stack effect</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Control transfer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP $l$</td>
<td>—</td>
<td>Jump unconditionally to location identified by label $l$</td>
</tr>
<tr>
<td>JUMPIFZ $l$</td>
<td>$n, \ldots \Rightarrow \ldots$</td>
<td>Jump to location identified by label $l$ if $n = 0$ (i.e., $n$ is false)</td>
</tr>
<tr>
<td>JUMPIFNZ $l$</td>
<td>$n, \ldots \Rightarrow \ldots$</td>
<td>Jump to location identified by label $l$ if $n \neq 0$ (i.e., $n$ is true)</td>
</tr>
<tr>
<td>CALL $l$</td>
<td>$\ldots \Rightarrow pc + 1, lb, 0, \ldots$</td>
<td>Call global subroutine at location identified by location $l$, setting up activation record by pushing static link (0 for global level), dynamic link (value of LB), and return address ($PC+1$, address of instruction after the call instruction) onto the stack</td>
</tr>
<tr>
<td>CALLI</td>
<td>$ca, sl, \ldots \Rightarrow \ldots$</td>
<td>Call subroutine indirectly; address of routine ($ca$) and static link to use ($sl$) on top of the stack; activation record as for CALL</td>
</tr>
<tr>
<td>RETURN $m$ $n$</td>
<td>$x^m, pc, lb, 0, y^n \ldots \Rightarrow x^m, \ldots$</td>
<td>Return from subroutine, replacing activation record by result and restoring LB</td>
</tr>
<tr>
<td><strong>Input/Output</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUTINT</td>
<td>$n, \ldots \Rightarrow \ldots$</td>
<td>Print $n$ to the terminal as a decimal integer</td>
</tr>
<tr>
<td>PUTCHR</td>
<td>$n, \ldots \Rightarrow \ldots$</td>
<td>Print the character with character code $n$ to the terminal</td>
</tr>
<tr>
<td>GETINT</td>
<td>$\ldots \Rightarrow n, \ldots$</td>
<td>Read decimal integer $n$ from the terminal and push onto the stack</td>
</tr>
<tr>
<td>GETCHR</td>
<td>$\ldots \Rightarrow n, \ldots$</td>
<td>Read character from the terminal and push its character code $n$ onto the stack</td>
</tr>
<tr>
<td><strong>TAM Control</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HALT</td>
<td>—</td>
<td>Stop execution and halt the machine</td>
</tr>
</tbody>
</table>