The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 3 MODULE, AUTUMN SEMESTER 2018–2019

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

Knowledge classification: Following School recommendation, the (sub)questions have been classified as follows, using a subset of Bloom's Taxonomy:

K: Knowledge

C: Comprehension

A: Application

Note that some 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

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.

```
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`.

```
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] }
```

(6)

Answer: *[C]*

1	=	`[$1]`
2	=	`$1 : $3`
3	=	`CmdAssign $1 $3`
4	=	`CmdIf $2 $4 $6`
5	=	`CmdWhile $2 $4`
6	=	`CmdSeq $2`

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 * i`
- `for j from 2 * m to n step -2 do sum := 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 * 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. (4)

Answer: *[A] The following productions need to be added to the lexical grammar:*

$$\mathit{Keyword} \quad \rightarrow \quad \texttt{for} \mid \texttt{from} \mid \texttt{step} \mid \texttt{to}$$

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

$$\begin{array}{lcl} \mathit{Command} & \rightarrow & \texttt{for}\ \mathit{VarExpression}\ \texttt{from}\ \mathit{Expression} \\ & & \texttt{to}\ \mathit{Expression}\ \mathit{OptStep}\ \texttt{do}\ \mathit{Command} \\ \mathit{OptStep} & \rightarrow & \epsilon \quad \mid \quad \texttt{step}\ \mathit{Expression} \end{array}$$

(ii) 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. (5)

Answer: *[A] Abstract syntax extension:*

```
data Command = ...
             | CmdFor Expression Expression Expression
                      (Maybe Expression) Command
```

Extension of the parser specification:

```
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

$$execute\ n\ [\![\ \texttt{for}\ E_{\mathrm{x}}\ \texttt{from}\ E_{\mathrm{f}}\ \texttt{to}\ E_{\mathrm{t}}\ \texttt{do}\ C\ ]\!] = \ldots$$

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 `LOADI` and `STOREI`). Call $execute$ 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: *[A] The following cases generate code for the `for`-loop:*

$execute\ n\ [\![\, \texttt{for}\ E_{\mathrm{x}}\ \texttt{from}\ E_{\mathrm{f}}\ \texttt{to}\ E_{\mathrm{t}}\ \texttt{do}\ C\,]\!] =$

$\quad execute\ n\ [\![\, \texttt{for}\ E_{\mathrm{x}}\ \texttt{from}\ E_1\ \texttt{to}\ E_2\ \texttt{step 1 do}\ C\,]\!]$

$execute\ n\ [\![\, \texttt{for}\ E_{\mathrm{x}}\ \texttt{from}\ E_{\mathrm{f}}\ \texttt{to}\ E_{\mathrm{t}}\ \texttt{step}\ E_{\mathrm{s}}\ \texttt{do}\ C\,]\!] =$

```
    evaluate [[ Ex ]]
    evaluate [[ Ef ]]
    LOAD [ST - 2]
    STOREI 0
    evaluate [[ Et ]]
    evaluate [[ Es ]]
loop:
    LOAD [ST - 3]
    LOADI 0
    LOAD [ST - 3]
    LOAD [ST - 3]
    LOADL 0
    LSS
    JUMPIFNZ negstep
    GTR
    JUMPIFNZ out
    JUMP body
negstep:
    LSS
    JUMPIFNZ out
body:
    execute (n + 3) [[ C ]]
    LOAD [ST - 3]
    LOADI 0
    LOAD [ST - 2]
    ADD
    LOAD [ST - 4]
    STOREI 0
    JUMP loop
out:
    POP 0 3
```

Question 2

(a) Consider the following expression language:

$$
\begin{array}{lll}
e & \rightarrow & & \textit{expressions:} \\
 & | & n & \textit{natural numbers, } n \in \mathbb{N} \\
 & | & x & \textit{variables, } x \in \mathrm{Name} \\
 & | & e \texttt{ + } e & \textit{addition} \\
 & | & e \texttt{ - } e & \textit{subtraction} \\
 & | & e \texttt{ * } e & \textit{multiplication} \\
 & | & e \texttt{ = } e & \textit{equality test} \\
 & | & \texttt{if } e \texttt{ then } e \texttt{ else } e & \textit{conditional} \\
 & | & \texttt{let var } x = e \texttt{ in } e & \textit{variable definition} \\
 & | & \texttt{let fun } f\,(x\!:\!t)\!:\!t = e \texttt{ in } e & \textit{function definition} \\
 & | & e\,(e) & \textit{function application}
\end{array}
$$

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

$$
\begin{array}{llll}
t & \rightarrow & & \textit{types:} \\
 & | & \texttt{Nat} & \textit{natural numbers} \\
 & | & \texttt{Bool} & \textit{Booleans} \\
 & | & t \rightarrow t & \textit{function (arrow) type}
\end{array}
$$

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

$$\Gamma \vdash n : \texttt{Nat} \qquad \text{(T-NAT)}$$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Nat} \quad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{Nat}} \qquad \text{(T-ADD)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Nat} \quad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{Nat}} \qquad \text{(T-SUB)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Nat} \quad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 \texttt{ * } e_2 : \texttt{Nat}} \qquad \text{(T-MUL)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Nat} \quad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 \texttt{ = } e_2 : \texttt{Bool}} \qquad \text{(T-EQ)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t} \qquad \text{(T-COND)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma,\ x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{let var}\ x\ =\ e_1\ \texttt{in}\ e_2 : t_2} \qquad \text{(T-LETVAR)}$$

$$\frac{\Gamma,\ f : t_{11} \to t_{12},\ x : t_{11} \vdash e_1 : t_{12} \quad \Gamma,\ f : t_{11} \to t_{12} \vdash e_2 : t_2}{\Gamma \vdash \texttt{let fun}\ f\,(x{:}t_{11})\,{:}t_{12}\ =\ e_1\ \texttt{in}\ e_2 : t_2} \qquad \text{(T-LETFUN)}$$

$$\frac{\Gamma \vdash e_1 : t_2 \to t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1(e_2) : t_1} \qquad \text{(T-APP)}$$

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

$$\texttt{x} : \texttt{Nat},\ \texttt{y} : \texttt{Bool},\ \texttt{z} : \texttt{Nat}$$

or empty, denoted $\emptyset$. Typing contexts are extended on the right, e.g. $\Gamma,\ \texttt{z} : \texttt{Nat}$, the membership predicate is denoted by $\in$, and lookup is from right to left, ensuring recent bindings hide earlier ones.

Use the typing rules given above to formally derive the type of the following (well-typed) expressions in the empty environment ($\emptyset$). Your proof should be in the form of a *proof tree*.

(i)
```
let var x = 1 + 7 in x * x
```
(4)

Answer: *[C]*

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \texttt{1} : \texttt{Nat}}\,\text{T-NAT} \quad \dfrac{}{\emptyset \vdash \texttt{7} : \texttt{Nat}}\,\text{T-NAT}}{\emptyset \vdash \texttt{1 + 7} : \texttt{Nat}}\,\text{T-ADD} \qquad \dfrac{}{\emptyset,\ \texttt{x} : \texttt{Nat} \vdash \texttt{x * x} : \texttt{Nat}}\,\text{below}}{\emptyset \vdash \texttt{let var x} = \texttt{1 + 7 in x * x} : \texttt{Nat}}\,\text{T-LETVAR}$$

$$\dfrac{\dfrac{\texttt{x} : \texttt{Nat} \in \emptyset,\ \texttt{x} : \texttt{Nat}}{\emptyset,\ \texttt{x} : \texttt{Nat} \vdash \texttt{x} : \texttt{Nat}}\,\text{T-VAR} \quad \dfrac{\texttt{x} : \texttt{Nat} \in \emptyset,\ \texttt{x} : \texttt{Nat}}{\emptyset,\ \texttt{x} : \texttt{Nat} \vdash \texttt{x} : \texttt{Nat}}\,\text{T-VAR}}{\emptyset,\ \texttt{x} : \texttt{Nat} \vdash \texttt{x * x} : \texttt{Nat}}\,\text{T-MUL}$$

(ii)
```
let fun fac(n : Nat) : Nat =
    if n = 0 then 1 else n * fac(n - 1)
in
    fac(7)
```
(9)

Answer: *[C]* Let

$$\begin{aligned} b &= \texttt{if n = 0 then 1 else n * fac(n - 1)} \\ \Gamma_1 &= \emptyset,\ \texttt{fac} : \texttt{Nat} \to \texttt{Nat},\ \texttt{n} : \texttt{Nat} \\ \Gamma_2 &= \emptyset,\ \texttt{fac} : \texttt{Nat} \to \texttt{Nat} \end{aligned}$$

$$\dfrac{\dfrac{}{\Gamma_1 \vdash b : \texttt{Nat}}\ \text{below} \qquad \dfrac{\dfrac{\texttt{fac} : \texttt{Nat} \to \texttt{Nat} \in \Gamma_2}{\Gamma_2 \vdash \texttt{fac} : \texttt{Nat} \to \texttt{Nat}}\ \text{T-VAR} \qquad \dfrac{}{\Gamma_2 \vdash \texttt{7} : \texttt{Nat}}\ \text{T-NAT}}{\Gamma_2 \vdash \texttt{fac(7)} : \texttt{Nat}}\ \text{T-APP}}{\emptyset \vdash \texttt{let fun fac(n:Nat):Nat} = b \ \texttt{in fac(7)} : \texttt{Nat}}\ \text{T-LETFUN}$$

$$\dfrac{\dfrac{}{\Gamma_1 \vdash \texttt{n = 0} : \texttt{Bool}}\ \text{below} \qquad \dfrac{}{\Gamma_1 \vdash \texttt{1} : \texttt{Nat}}\ \text{T-NAT} \qquad \dfrac{}{\Gamma_1 \vdash \texttt{n * fac(n - 1)} : \texttt{Nat}}\ \text{below}}{\Gamma_1 \vdash b : \texttt{Nat}}\ \text{T-COND}$$

$$\dfrac{\dfrac{\texttt{n} : \texttt{Nat} \in \Gamma_1}{\Gamma_1 \vdash \texttt{n} : \texttt{Nat}}\ \text{T-VAR} \qquad \dfrac{}{\Gamma_1 \vdash \texttt{0} : \texttt{Nat}}\ \text{T-NAT}}{\Gamma_1 \vdash \texttt{n = 0} : \texttt{Bool}}\ \text{T-EQ}$$

$$\dfrac{\dfrac{\texttt{n} : \texttt{Nat} \in \Gamma_1}{\Gamma_1 \vdash \texttt{n} : \texttt{Nat}}\ \text{T-VAR} \qquad \dfrac{\dfrac{\texttt{fac} : \texttt{Nat} \to \texttt{Nat} \in \Gamma_1}{\Gamma_1 \vdash \texttt{fac} : \texttt{Nat} \to \texttt{Nat}}\ \text{T-VAR} \qquad \dfrac{}{\Gamma_1 \vdash \texttt{n - 1} : \texttt{Nat}}\ \text{below}}{\Gamma_1 \vdash \texttt{fac(n - 1)} : \texttt{Nat}}\ \text{T-APP}}{\Gamma_1 \vdash \texttt{n * fac(n - 1)} : \texttt{Nat}}\ \text{T-MUL}$$

$$\dfrac{\dfrac{\texttt{n} : \texttt{Nat} \in \Gamma_1}{\Gamma_1 \vdash \texttt{n} : \texttt{Nat}}\ \text{T-VAR} \qquad \dfrac{}{\Gamma_1 \vdash \texttt{1} : \texttt{Nat}}\ \text{T-NAT}}{\Gamma_1 \vdash \texttt{n - 1} : \texttt{Nat}}\ \text{T-SUB}$$

(b) Suppose we wish to extend MiniTriangle with a command `break`:

$$\begin{array}{lll} \mathit{Command} & \to & \ldots \qquad\qquad\qquad\quad \ldots \\ & | & \texttt{break}\ \underline{\mathit{IntegerLiteral}} \quad \text{CmdBreak} \end{array}$$

See Appendix A for the abstract syntax for the remaining MiniTriangle commands. The intended semantics of `break` n, where $n \geq 1$, is to terminate the innermost n loops, with the execution continuing immediately after the nth loop. It should be a static error if there are fewer than n loops enclosing a command `break` n or if $n < 1$. Define, using inference rules, a binary relation *Well Enclosed* on numbers and commands characterising the static correctness of commands in this sense. *Hint:* Think of the number as a form of context keeping track of the number of enclosing loops. (12)

Answer: *[A] We need to define a relation on numbers and commands*

$$n \vdash \mathit{Command}$$

such that a number n is related to a $\mathit{Command}$ c, $n \vdash c$, iff enclosing c in n loops ensures that all contained commands `break` m *are enclosed by at least m loops and for all arguments m of contained commands* `break` m*, $m \geq 1$.*

$$n \vdash e_1 \texttt{ := } e_2 \qquad \text{(WE-ASSIGN)}$$

$$n \vdash e_1\texttt{(}e_2\texttt{)} \qquad \text{(WE-CALL)}$$

$$\frac{n \vdash \overline{c}}{n \vdash \texttt{begin}\ \overline{c}\ \texttt{end}} \qquad \text{(WE-SEQ)}$$

$$\frac{n \vdash c_1 \quad n \vdash c_2}{n \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2} \qquad \text{(WE-IF)}$$

$$\frac{n + 1 \vdash c}{n \vdash \texttt{while}\ e\ \texttt{do}\ c} \qquad \text{(WE-WHILE)}$$

$$\frac{n \vdash c}{n \vdash \texttt{let}\ \overline{d}\ \texttt{in}\ c} \qquad \text{(WE-LET)}$$

$$\frac{1 \leq m \leq n}{n \vdash \texttt{break}\ m} \qquad \text{(WE-BREAK)}$$

Question 3

(a) Transform the following code fragment into *static single assignment* (SSA) form:

```
a := 1;
b := 17;
i := 0;
while i < n do begin
    c := a + i;
    i := i + 1;
    a := c
end;
b := b + a
```

(10)

Answer: *[A]*

a_1 := 1;
b_1 := 17;
i_1 := 0;
while (a_2 = ϕ(a_1,a_3), i_2 = ϕ(i_1,i_3), i_2 < n) do begin
    c := a_2 + i_2;
    i_3 := i_2 + 1
    a_3 := c;
end;
b_2 := b_1 + a_2

(b) This question concerns *register allocation by graph colouring*. Consider the following assembly code fragment for a typical register machine:

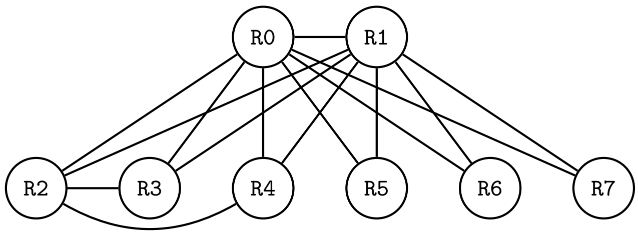
```
        load    R0, 1
        load    R1, 0
loop:   mul     R2, R0, R0
        mul     R3, R0, R0
        mul     R4, R3, R0
        add     R5, R2, R4
        add     R1, R1, R5
        load    R6, 1
        add     R0, R0, R6
        load    R7, 10
        cmp     R0, R7
        ble     loop
```

The `load` instruction stores a numeric constant into the designated register. Arithmetic instructions with three register arguments perform the arithmetic operation on the two last registers and store the result into the first. The instruction `ble` is a conditional branch (jump) instruction.

(i) Draw the *interference graph* for the above code fragment. It should have one node for each of the eight registers being used. (6)

Answer: *[A]*

`R0` and `R1` are loop variables (registers), live at the start of the loop and used before being updated in each iteration. Their live ranges thus overlap with those of all other variables, including each other. `R2` is used in the definition (computation) of `R5`. It's live range thus overlaps with those of `R3` and `R4`. All other variables are short lived: there are thus no further overlapping live ranges.



(ii) "Colour" the interference graph using as few colours as possible such that no two adjacent nodes have the same colour. Use this result to carry out register allocation for the above code fragment by associating each colour with one register. Your answer should include the coloured graph and the final version of the code using a minimal number of registers. (9)

Answer: *[A]*

Node	*Colour*	*Register*
R0	*red*	*R0*
R1	*green*	*R1*
R2	*blue*	*R2*
R3	*black*	*R3*
R4	*black*	*R3*
R5	*black*	*R3*
R6	*black*	*R3*
R7	*black*	*R3*

(This is not the only possible (minimal) colouring, and of course it does not matter whether actual colour names or some other naming scheme is used. Indeed, in practice, "colouring" would typically be done directly in terms of physical registers.)

```
load    R0, 1
load    R1, 0
```

```
loop:   mul     R2, R0, R0
        mul     R3, R0, R0
        mul     R3, R3, R0
        add     R3, R2, R3
        add     R1, R1, R3
        load    R3, 1
        add     R0, R0, R3
        load    R3, 10
        cmp     R0, R3
        ble     loop
```

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 <u>*this*</u>

MiniTriangle Lexical Syntax:

Program → (*Token* | *Separator*)*

Token → *Keyword* | *Identifier* | *IntegerLiteral* | *Operator*
| `,` | `;` | `:` | `:=` | `=` | `(` | `)` | `[` | `]` | <u>*eot*</u>

Keyword → `begin` | `const` | `do` | `else` | `end` | `fun` | `if` | `in`
| `let` | `out` | `proc` | `then` | `var` | `while`

Identifier → *Letter* | *Identifier Letter* | *Identifier Digit*
except *Keyword*

IntegerLiteral → *Digit* | *IntegerLiteral Digit*

Operator → `^` | `*` | `/` | `+` | `-` | `<` | `<=` | `==` | `!=` | `>=` | `>` | `&&` | `||` | `!`

Letter → `A` | `B` | ... | `Z` | `a` | `b` | ... | `z`

Digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Separator → *Comment* | <u>*space*</u> | <u>*eol*</u>

Comment → `//` (any character except <u>*eol*</u>)* <u>*eol*</u>

MiniTriangle Concrete Syntax:

Program → *Command*

Commands → *Command*
  | *Command* `;` *Commands*

Command → *VarExpression* `:=` *Expression*
  | *VarExpression* `(` *Expressions* `)`
  | `if` *Expression* `then` *Command*
    `else` *Command*
  | `while` *Expression* `do` *Command*
  | `let` *Declarations* `in` *Command*
  | `begin` *Commands* `end`

Expressions → ϵ
  | $\mathit{Expressions}_1$

$\mathit{Expressions}_1$ → *Expression*
  | *Expression* `,` $\mathit{Expressions}_1$

Expression → *PrimaryExpression*
  | *Expression* *BinaryOperator* *Expression*

PrimaryExpression → <u>*IntegerLiteral*</u>
  | *VarExpression*
  | *UnaryOperator* *PrimaryExpression*
  | *VarExpression* `(` *Expressions* `)`
  | `[` *Expressions* `]`
  | `(` *Expression* `)`

VarExpression → <u>*Identifier*</u>
  | *VarExpression* `[` *Expression* `]`

BinaryOperator → `^` | `*` | `/` | `+` | `-` | `<` | `<=` | `==` | `!=` | `>=` | `>` | `&&` | `||`

UnaryOperator → `-` | `!`

$$
\begin{array}{lcl}
\mathit{Declarations} & \rightarrow & \mathit{Declaration} \\
 & | & \mathit{Declaration}\ \texttt{;}\ \mathit{Declarations} \\
 & & \\
\mathit{Declaration} & \rightarrow & \texttt{const}\ \underline{\mathit{Identifier}}\ \texttt{:}\ \mathit{TypeDenoter}\ \texttt{=}\ \mathit{Expression} \\
 & | & \texttt{var}\ \underline{\mathit{Identifier}}\ \texttt{:}\ \mathit{TypeDenoter} \\
 & | & \texttt{var}\ \underline{\mathit{Identifier}}\ \texttt{:}\ \mathit{TypeDenoter}\ \texttt{:=}\ \mathit{Expression} \\
 & | & \texttt{fun}\ \underline{\mathit{Identifier}}\ \texttt{(}\ \mathit{ArgDecls}\ \texttt{)}\ \texttt{:}\ \mathit{TypeDenoter}\ \texttt{=}\ \mathit{Expression} \\
 & | & \texttt{proc}\ \underline{\mathit{Identifier}}\ \texttt{(}\ \mathit{ArgDecls}\ \texttt{)}\ \mathit{Command} \\
 & & \\
\mathit{ArgDecls} & \rightarrow & \epsilon \\
 & | & \mathit{ArgDecls_1} \\
 & & \\
\mathit{ArgDecls_1} & \rightarrow & \mathit{ArgDecl} \\
 & | & \mathit{ArgDecl}\ \texttt{,}\ \mathit{ArgDecls_1} \\
 & & \\
\mathit{ArgDecl} & \rightarrow & \underline{\mathit{Identifier}}\ \texttt{:}\ \mathit{TypeDenoter} \\
 & | & \texttt{in}\ \underline{\mathit{Identifier}}\ \texttt{:}\ \mathit{TypeDenoter} \\
 & | & \texttt{out}\ \underline{\mathit{Identifier}}\ \texttt{:}\ \mathit{TypeDenoter} \\
 & | & \texttt{var}\ \underline{\mathit{Identifier}}\ \texttt{:}\ \mathit{TypeDenoter} \\
 & & \\
\mathit{TypeDenoter} & \rightarrow & \underline{\mathit{Identifier}} \\
 & | & \mathit{TypeDenoter}\ \texttt{[}\ \underline{\mathit{IntegerLiteral}}\ \texttt{]}
\end{array}
$$

Note that the productions for *Expression* make the grammar as stated above ambiguous. Operator precedence and associativity for the *binary* operators as defined in the following table are used to disambiguate:

Operator	Precedence	Associativity
`^`	1	right
`* /`	2	left
`+ -`	3	left
`< <= == != >= >`	4	non
`&&`	5	left
`||`	6	left

A precedence level of 1 means the highest precedence, 2 means second highest, and so on.

MiniTriangle Abstract Syntax: $\underline{Name} = \underline{Identifier} \cup \underline{Operator}$.

Program	→	*Command*	Program
Command	→	*Expression* `:=` *Expression*	CmdAssign
	|	*Expression* `(` *Expression** `)`	CmdCall
	|	`begin` *Command** `end`	CmdSeq
	|	`if` *Expression* `then` *Command* `else` *Command*	CmdIf
	|	`while` *Expression* `do` *Command*	CmdWhile
	|	`let` *Declaration** `in` *Command*	CmdLet
Expression	→	$\underline{IntegerLiteral}$	ExpLitInt
	|	$\underline{Name}$	ExpVar
	|	*Expression* `(` *Expression** `)`	ExpApp
	|	`[` *Expression** `]`	ExpAry
	|	*Expression* `[` *Expression* `]`	ExpIx
Declaration	→	`const` $\underline{Name}$ `:` *TypeDenoter* `=` *Expression*	DeclConst
	|	`var` $\underline{Name}$ `:` *TypeDenoter* `(` `:=` *Expression* $\mid \epsilon$ `)`	DeclVar
	|	`fun` $\underline{Name}$ `(` *ArgDecl** `)` `:` *TypeDenoter* `=` *Expression*	DeclFun
	|	`proc` $\underline{Name}$ `(` *ArgDecl** `)` *Command*	DeclProc
ArgDecl	→	*ArgMode* $\underline{Name}$ `:` *TypeDenoter*	ArgDecl
ArgMode	→	ϵ	ByValue
	|	`in`	ByRefIn
	|	`out`	ByRefOut
	|	`var`	ByRefVar
TypeDenoter	→	$\underline{Name}$	TDBaseType
	→	*TypeDenoter* `[` $\underline{IntegerLiteral}$ `]`	TDArray

Appendix B: Triangle Abstract Machine (TAM) Instructions

Meta variable	Meaning
a	Address: one of the forms specified by table below when part of an instruction, specific stack address when on the stack
b	Boolean value ($\mathrm{false} = 0$ or $\mathrm{true} = 1$)
ca	Code address; address to routine in the code segment
d	Displacement; i.e., offset w.r.t. address in register or on the stack
l	Label name
m, n, p	Integer
x, y, z	Any kind of stack data
x^n	Vector of n items, $n \geq 0$, here any kind

Address form	Description
[SB + d] [SB - d]	Address given by contents of register SB (Stack Base) $+/-$ displacement d
[LB + d] [LB - d]	Address given by contents of register LB (Local Base) $+/-$ displacement d
[ST + d] [ST - d]	Address given by contents of register ST (Stack Top) $+/-$ displacement d

Instruction	Stack effect	Description
	Label	
LABEL l	—	Pseudo instruction: symbolic location
	Load and store	
LOADL n	$\ldots \Rightarrow n, \ldots$	Push literal integer n onto stack
LOADCA l	$\ldots \Rightarrow \mathrm{addr}(l), \ldots$	Push address of label l (code segment) onto stack
LOAD a	$\ldots \Rightarrow [a], \ldots$	Push contents at address a onto stack
LOADA a	$\ldots \Rightarrow a, \ldots$	Push address a onto stack
LOADI d	$a, \ldots \Rightarrow [a+d], \ldots$	Load indirectly; push contents at address $a+d$ onto stack
STORE a	$n, \ldots \Rightarrow \ldots$	Pop value n from stack and store at address a
STOREI d	$a, n, \ldots \Rightarrow \ldots$	Store indirectly; store n at address $a+d$

Instruction	Stack effect	Description
	Block operations	
LOADLB m n	$\ldots \Rightarrow m^n, \ldots$	Push block of n literal integers m onto stack
LOADIB n	$a, \ldots \Rightarrow$ $[a+(n-1)], \ldots, [a+0], \ldots$	Load block of size n indirectly
STOREIB n	$a, x^n, \ldots \Rightarrow \ldots$	Store block of size n indirectly
POP m n	$x^m, y^n, \ldots \Rightarrow x^m, \ldots$	Pop n values below top m values
	Arithmetic operations	
ADD	$n_2, n_1, \ldots \Rightarrow n_1 + n_2, \ldots$	Add n_1 and n_2, replacing n_1 and n_2 with the sum
SUB	$n_2, n_1, \ldots \Rightarrow n_1 - n_2, \ldots$	Subtract n_2 from n_1, replacing n_1 and n_2 with the difference
MUL	$n_2, n_1, \ldots \Rightarrow n_1 \cdot n_2, \ldots$	Multiply n_1 by n_2, replacing n_1 and n_2 with the product
DIV	$n_2, n_1, \ldots \Rightarrow n_1/n_2, \ldots$	Divide n_1 by n_2, replacing n_1 and n_2 with the (integer) quotient
NEG	$n, \ldots \Rightarrow -n, \ldots$	Negate n, replacing n with the result
	Comparison & logical operations ($\text{false} = 0$, $\text{true} = 1$)	
LSS	$n_2, n_1, \ldots \Rightarrow n_1 < n_2, \ldots$	Check if n_1 is smaller than n_2, replacing n_1 and n_2 with the Boolean result
EQL	$n_2, n_1, \ldots \Rightarrow n_1 = n_2, \ldots$	Check if n_1 is equal to n_2, replacing n_1 and n_2 with the Boolean result
GTR	$n_2, n_1, \ldots \Rightarrow n_1 > n_2, \ldots$	Check if n_1 is greater than n_2, replacing n_1 and n_2 with the Boolean result
AND	$b_2, b_1, \ldots \Rightarrow b_1 \wedge b_2, \ldots$	Logical conjunction of b_1 and b_2, replacing b_1 and b_2 with the Boolean result
OR	$b_2, b_1, \ldots \Rightarrow b_1 \vee b_2, \ldots$	Logical disjunction of b_1 and b_2, replacing b_1 and b_2 with the Boolean result
NOT	$b, \ldots \Rightarrow \neg b, \ldots$	Logical negation of b, replacing b with the result

Instruction	Stack effect	Description
	Control transfer	
JUMP l	—	Jump unconditionally to location identified by label l
JUMPIFZ l	$n, \ldots \Rightarrow \ldots$	Jump to location identified by label l if $n = 0$ (i.e., n is false)
JUMPIFNZ l	$n, \ldots \Rightarrow \ldots$	Jump to location identified by label l if $n \neq 0$ (i.e., n is true)
CALL l	$\ldots \Rightarrow$ PC $+ 1$, LB, 0, $\ldots$	Call global subroutine at location l: Activation record set up 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; PC $= l$; LB = start of activation record (address of static link)
CALLI	ca, sl, $\ldots \Rightarrow$ PC $+ 1$, LB, sl, $\ldots$	Call subroutine indirectly: address of routine (ca) and static link to use (sl) on top of the stack; activation record and new PC and LB as for CALL
RETURN m n	x^m, y^p, ra, olb, sl, y^n, $\ldots \Rightarrow x^m, \ldots$	Return from subroutine, replacing activation record by result, jumping to return address (PC $= ra$), and restoring the old local base (LB $= olb$)
	Input/Output	
PUTINT	$n, \ldots \Rightarrow \ldots$	Print n to the terminal as a decimal integer
PUTCHR	$n, \ldots \Rightarrow \ldots$	Print the character with character code n to the terminal
GETINT	$\ldots \Rightarrow n, \ldots$	Read decimal integer n from the terminal and push onto the stack
GETCHR	$\ldots \Rightarrow n, \ldots$	Read character from the terminal and push its character code n onto the stack
	TAM Control	
HALT	—	Stop execution and halt the machine