The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 3 MODULE, AUTUMN SEMESTER 2017–2018

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

(a) Explain and give examples of the following kinds of compile-time error:

- lexical error
- syntax error (context-free)
- contextual error

(6)

Answer: *[C]*

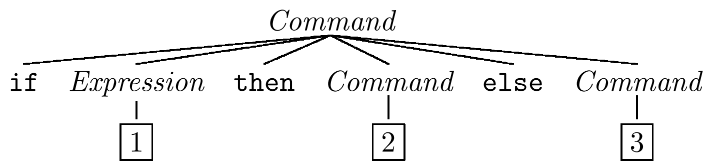
- *A lexical error occurs when the input does not conform to the lexical syntax of a language. Examples include encountering a character that cannot be part of any valid lexeme, or an ill-formed string or numerical literal.*
- *A syntax error occurs when the input does not conform to the context-free syntax of a language. A typical example would be unbalanced parentheses, or missing terminating keyword, such as a* `repeat` *without an* `until`*.*
- *A contextual error occurs when contextual constraints are violated. Examples include type errors, such as adding a Boolean to an integer in a language that requires the terms of addition to be of the same type, and a missing declaration of a variable in a language that requires declaration before use.*

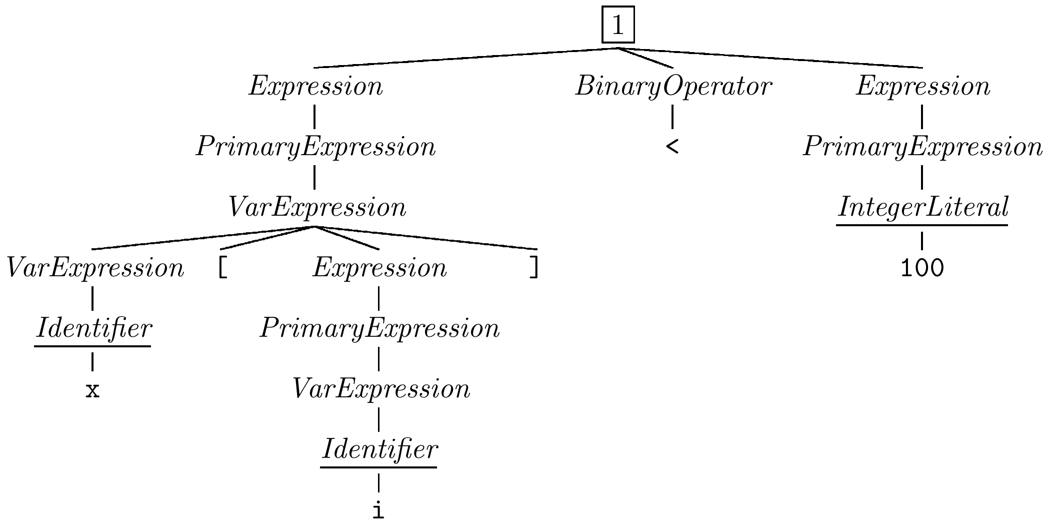
(b) Draw the *parse* (or *derivation*) tree for the following MiniTriangle fragment. The relevant grammar is given in Appendix A. Start from the production for "Command".

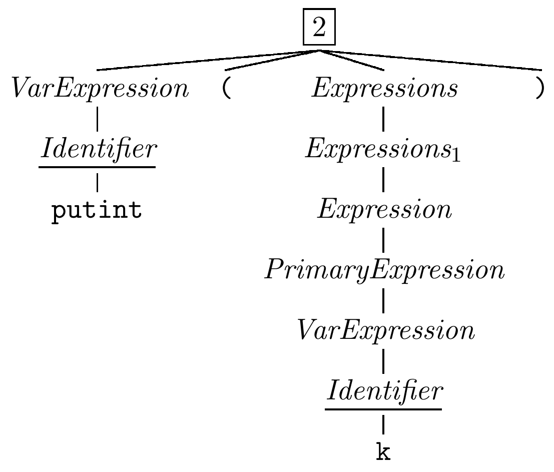
```
if x[i] < 100 then
    putint(k)
else
    i := (-i) - 1
```

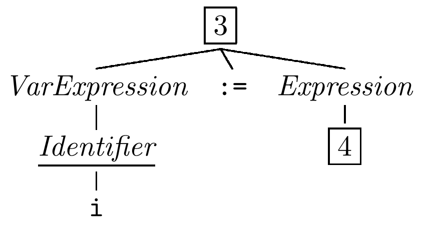
(9)

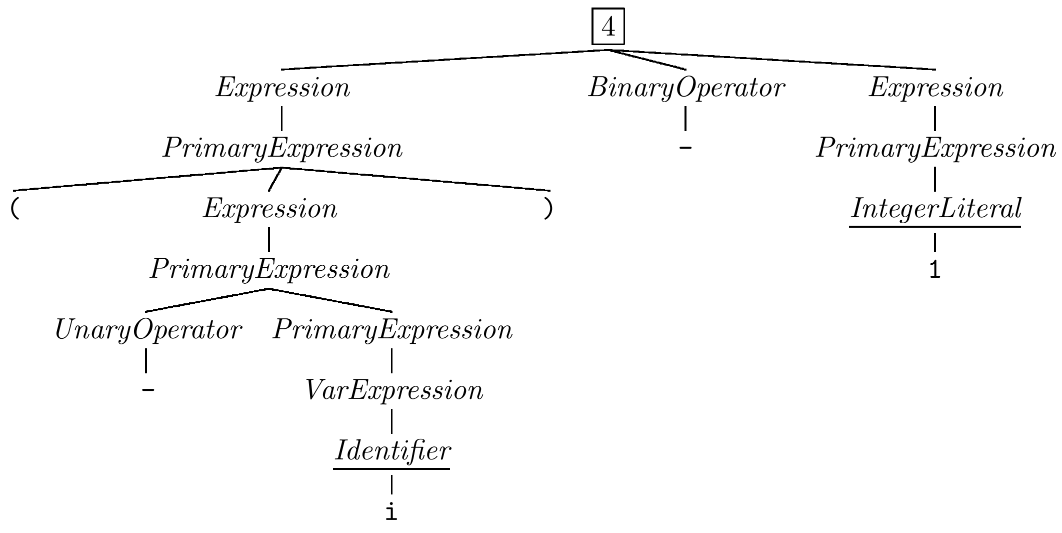
Answer: *[A]*






2
VarExpression
(
Expressions
)
Identifier
putint
Expressions1
Expression
PrimaryExpression
VarExpression
Identifier
k


3
VarExpression
:=
Expression
Identifier
i
4

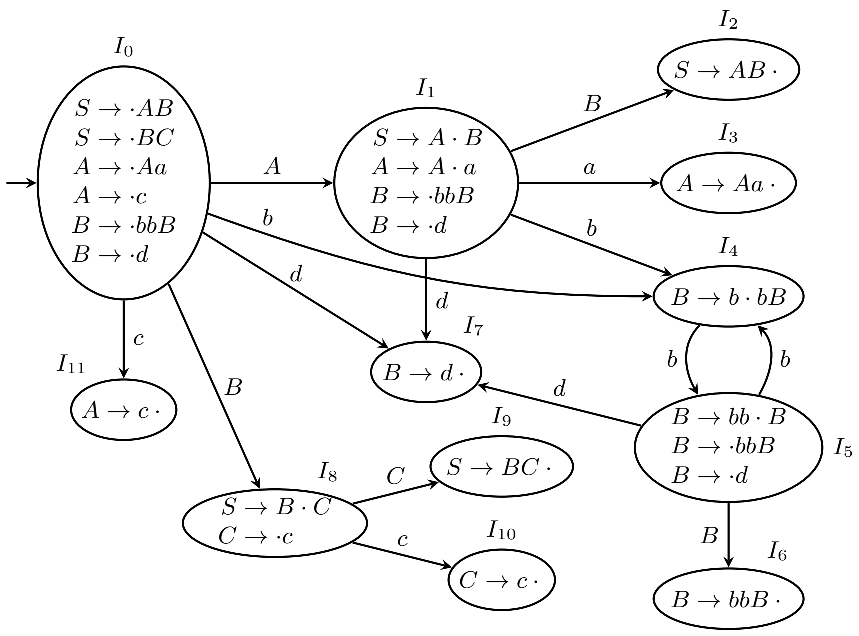

4
Expression
BinaryOperator
Expression
PrimaryExpression
-
PrimaryExpression
(
Expression
)
IntegerLiteral
PrimaryExpression
1
UnaryOperator
PrimaryExpression
-
VarExpression
Identifier
i

(c) Consider the following context-free grammar (CFG):

$$\begin{aligned} S &\rightarrow AB \mid BC \\ A &\rightarrow Aa \mid c \\ B &\rightarrow bbB \mid d \\ C &\rightarrow c \end{aligned}$$

S, A, B, and C are nonterminal symbols, S is the start symbol, and a, b, c, and d are terminal symbols.

The DFA below recognizes the viable prefixes for this CFG:



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

State	Stack	Input	Move
I_0	ϵ	*caabbbbd*	Shift
I_{11}	c	*aabbbbd*	Reduce by $A \rightarrow c$
⋮	⋮	⋮	⋮
	S	ϵ	Done

(10)

Answer: *[A]*

State	*Stack*	*Input*	*Move*
I_0	ϵ	$caabbbbd$	*Shift*
I_{11}	c	$aabbbbd$	*Reduce by* $A \to c$
I_1	A	$aabbbbd$	*Shift*
I_3	Aa	$abbbbd$	*Reduce by* $A \to Aa$
I_1	A	$abbbbd$	*Shift*
I_3	Aa	$bbbbd$	*Reduce by* $A \to Aa$
I_1	A	$bbbbd$	*Shift*
I_4	Ab	$bbbd$	*Shift*
I_5	Abb	bbd	*Shift*
I_4	$Abbb$	bd	*Shift*
I_5	$Abbbb$	d	*Shift*
I_7	$Abbbbd$	ϵ	*Reduce by* $B \to d$
I_6	$AbbbbB$	ϵ	*Reduce by* $B \to bbB$
I_6	$AbbB$	ϵ	*Reduce by* $B \to bbB$
I_2	AB	ϵ	*Reduce by* $S \to AB$
	S	ϵ	*Done*

Question 2
Consider the language given by the following abstract syntax:

C	$\rightarrow$		*Commands:*
		`skip`	*Do nothing*
	$\mid$	C `;` C	*Sequencing*
	$\mid$	x `:=` E	*Assignment*
	$\mid$	`if` E `then` C `else` C	*Conditional command*
	$\mid$	`while` E `do` C	`while`-*loop*
E	$\rightarrow$		*Expressions:*
		n	*Literal integer*
	$\mid$	x	*Variable*
	$\mid$	E `+` E	*Addition*
	$\mid$	E `=` E	*Comparison*

For this question, you will develop code generation functions for the above language, targeting the Triangle Abstract Machine (TAM). See appendix B for a specification of the TAM instructions. Assume conventional (imperative) semantics for the above language constructs, along with the following:

- x is the syntactic category of variable identifiers, ranging over the 26 names `a`, `b`, ..., `z`. They refer to 26 global variables stored at `SB` $+ 0$ (`a`) to `SB` $+ 25$ (`z`).
- The `while`-loop has the following semantics: the loop expression E is evaluated; if the result is true, the loop body C is executed next and then the process is repeated from the evaluation of the loop expression; otherwise execution continues after the loop.

The code generation functions should be specified through *code templates* in the style used in the lectures. Assume a function $addr(x)$ that returns the address (of the form `[SB + `d`]`) for a variable x. Further, you will have to consider generation of fresh labels. Assume a monadic-style operation $l \leftarrow fresh$ to bind a variable l to a distinct label that then can be used in jumps and as jump targets. For example:

$$
\begin{array}{lcrl}
execute\ [\![\, \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2\,]\!] & = & & l_1 \leftarrow fresh \\
 & & & \ldots \\
 & & & \texttt{JUMP}\ l_1 \\
 & & & \ldots \\
 & & l_1\texttt{:} & \ldots
\end{array}
$$

(a) Write a code generation function *evaluate* that generates TAM code for evaluating an expression. The first case should start like:

$$\textit{evaluate}\ [\![\, n \,]\!] \quad = \quad \ldots$$

(4)

Answer: *[A] The following function generates code for the specified expressions:*

$$\begin{array}{lcl}
\textit{evaluate}\ [\![\, n \,]\!] & = & \texttt{LOADL}\ n \\
\textit{evaluate}\ [\![\, x \,]\!] & = & \texttt{LOAD}\ \textit{addr}(x) \\
\textit{evaluate}\ [\![\, E_1\ \texttt{+}\ E_2 \,]\!] & = & \textit{evaluate}\ E_1 \\
 & & \textit{evaluate}\ E_2 \\
 & & \texttt{ADD} \\
\textit{evaluate}\ [\![\, E_1\ \texttt{=}\ E_2 \,]\!] & = & \textit{evaluate}\ E_1 \\
 & & \textit{evaluate}\ E_2 \\
 & & \texttt{EQL}
\end{array}$$

Marking: 1 mark for each case.

(b) Write a code generation function *execute* that generates TAM code for executing commands. It should handle the five forms of commands specified by the abstract syntax above. (12)

Answer: *[A]*

$$
\begin{array}{llll}
\mathit{execute}\,[\![\,\texttt{skip}\,]\!] & = & & \epsilon \\
\mathit{execute}\,[\![\,C_1\ \texttt{;}\ C_2\,]\!] & = & & \mathit{execute}\ C_1 \\
 & & & \mathit{execute}\ C_2 \\
\mathit{execute}\,[\![\,x\ \texttt{:=}\ E\,]\!] & = & & \mathit{evaluate}\ E \\
 & & & \texttt{STORE}\ \mathit{addr}(x) \\
\mathit{execute}\,[\![\,\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2\,]\!] & = & & \mathit{else} \leftarrow \mathit{fresh} \\
 & & & \mathit{endif} \leftarrow \mathit{fresh} \\
 & & & \mathit{evaluate}\ E \\
 & & & \texttt{JUMPIFZ}\ \mathit{else} \\
 & & & \mathit{execute}\ C_1 \\
 & & & \texttt{JUMP}\ \mathit{endif} \\
 & & \mathit{else}: & \mathit{execute}\ C_2 \\
 & & \mathit{endif}: & \\
\mathit{execute}\,[\![\,\texttt{while}\ E\ \texttt{do}\ C\,]\!] & = & & \mathit{loop} \leftarrow \mathit{fresh} \\
 & & & \mathit{out} \leftarrow \mathit{fresh} \\
 & & \mathit{loop}: & \mathit{evaluate}\ E \\
 & & & \texttt{JUMPIFZ}\ \mathit{out} \\
 & & & \mathit{execute}\ C \\
 & & & \texttt{JUMP}\ \mathit{loop} \\
 & & \mathit{out}: &
\end{array}
$$

Marking: 1 mark each for skip, sequence; 2 for assignment; 4 marks for conditional; 4 marks for `while`*-loop.*

(c) Now assume we wish to extend the language with the commands `break` and `continue` :

$$
\begin{array}{lcll}
C & \rightarrow & & \textit{Commands:} \\
 & | & \ldots & \\
 & | & \texttt{break} & \textit{Terminate innermost loop} \\
 & | & \texttt{continue} & \textit{Continue with next loop iteration}
\end{array}
$$

The semantics is that `break` will terminate the innermost loop, with execution continuing immediately after the loop, while `continue` will skip whatever remains of the loop body, and continue execution directly with the next loop iteration.

Modify and extend *execute* to generate code for the extended language. Note that *execute* will need (an) extra argument(s) for contextual information to keep track of the current innermost loop. You may assume that using `break` or `continue` outside any loop is a *static* error. Thus your code generator does not need to handle that case. Your answer should include the modified *execute* cases for `if` and `while`, as well as the cases for the two new commands. (9)

Answer: *[A]*

```
$execute\ bl\ cl\ [\![$ if $E$ then $C_1$ else $C_2\ ]\!]$  =          $else \leftarrow fresh$
                                                                      $endif \leftarrow fresh$
                                                                      $evaluate\ E$
                                                                      JUMPIFZ $else$
                                                                      $execute\ bl\ cl\ C_1$
                                                                      JUMP $endif$
                                                             $else$:  $execute\ bl\ cl\ C_2$
                                                             $endif$:
$execute\ bl\ cl\ [\![$ while $E$ do $C\ ]\!]$          =             $loop \leftarrow fresh$
                                                                      $out \leftarrow fresh$
                                                             $loop$:  $evaluate\ E$
                                                                      JUMPIFZ $out$
                                                                      $execute\ out\ loop\ C$
                                                                      JUMP $loop$
                                                             $out$:
$execute\ bl\ cl\ [\![$ break $]\!]$                    =             JUMP $bl$
$execute\ bl\ cl\ [\![$ continue $]\!]$                 =             JUMP $cl$
```

Marking: 3 marks for getting the general idea; 2 marks for correct calls to execute in case for `if`*; 2 marks for correct call to execute in case for* `while`*; 1 mark each for correct code for* `break` *and* `continue`*.*

Question 3
This question concerns code improvement (optimisation) and internal representations that facilitate analysis and code improvement.

(a) Explain the code improvement technique *common subexpression elimination*, illustrating with an example. Also discuss when the technique cannot be applied, again illustrating with an example. (6)

Answer: *[K] The idea is to avoid evaluating the "same expression" more than once. For example, the statements:*

```
x1 := y1 + 7 * z + 42;
x2 := y2 + 7 * z + 42;
```

can be transformed to:

```
t := 7 * z + 42;
x1 := y1 + t;
x2 := y2 + t;
```

thus avoiding evaluating `7 * z + 42` *twice.*

However, it has to be ensured that the expressions actually have the same meaning and not just are syntactically the same, and that they do not have any side effects as eliminating an effect generally will change the meaning of a program. For example, in

```
let x = y + 1 in let y = 10 in let z = y + 1
```

the two expressions `y + 1` *are not the same as they refer to two different variables that both happen to have the name* `y`*. For an example involving effects, consider a C-like increment operation applied to a variable:* `i++`*. Computing this expression only once and replacing multiple occurrences of it by the result will clearly change the meaning of the program.*

(b) Show how the following program fragment involving a C-like `for`-loop might be transformed by means of *loop unrolling* in a situation where the loop bound `n` is *not* statically known:

```
b[0] := a[0];
for (i := 1; i < n; i++) do
    b[i] := b[i-1] + a[i];
```

Also discuss the potential advantages and disadvantages of this transformation. (9)

Answer: *[K,A]*

```
b[0] := a[0];
for (i := 1; i < ((n-1)/2)*2; i := i + 2) do begin
    b[i] := b[i-1] + a[i];
    b[i + 1] := b[i] + a[i + 1]
end;
if (i < n) then begin
    b[i] := b[i-1] + a[i];
    i++ end;
```

Benefits include that the number of iterations are reduced (here halved), meaning fewer jumps (which can be expensive) and a larger loop body that may open up for other optimisation's and improved register allocation. Drawbacks include that the code becomes larger, which could have an impact on cache performance.

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

```
a := 0;
b := 1;
i := 2;
while i < n do begin
    c := a + b;
    a := b;
    b := c;
    i := i + 1
end
```

(10)

Answer: *[A]*

```
$a_1$ := 0;
$b_1$ := 1;
$i_1$ := 2;
while ($i_2$ = $\phi$($i_1$,$i_3$), $a_2$ = $\phi$($a_1$,$a_3$), $b_2$ = $\phi$($b_1$,$b_3$), $i_2$ < n) do begin
    c := $a_2$ + $b_2$;
    $a_3$ := $b_2$;
    $b_3$ := c;
    $i_3$ := $i_2$ + 1
end
```

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:

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

MiniTriangle Concrete Syntax:

Program	$\rightarrow$	*Command*
Commands	$\rightarrow$	*Command*
	$\mid$	*Command* `;` *Commands*
Command	$\rightarrow$	*VarExpression* `:=` *Expression*
	$\mid$	*VarExpression* `(` *Expressions* `)`
	$\mid$	`if` *Expression* `then` *Command*
		`else` *Command*
	$\mid$	`while` *Expression* `do` *Command*
	$\mid$	`let` *Declarations* `in` *Command*
	$\mid$	`begin` *Commands* `end`
Expressions	$\rightarrow$	ϵ
	$\mid$	$\textit{Expressions}_1$
$\textit{Expressions}_1$	$\rightarrow$	*Expression*
	$\mid$	*Expression* `,` $\textit{Expressions}_1$
Expression	$\rightarrow$	*PrimaryExpression*
	$\mid$	*Expression* *BinaryOperator* *Expression*
PrimaryExpression	$\rightarrow$	$\underline{\textit{IntegerLiteral}}$
	$\mid$	*VarExpression*
	$\mid$	*UnaryOperator* *PrimaryExpression*
	$\mid$	*VarExpression* `(` *Expressions* `)`
	$\mid$	`[` *Expressions* `]`
	$\mid$	`(` *Expression* `)`
VarExpression	$\rightarrow$	$\underline{\textit{Identifier}}$
	$\mid$	*VarExpression* `[` *Expression* `]`
BinaryOperator	$\rightarrow$	`^` $\mid$ `*` $\mid$ `/` $\mid$ `+` $\mid$ `-` $\mid$ `<` $\mid$ `<=` $\mid$ `==` $\mid$ `!=` $\mid$ `>=` $\mid$ `>` $\mid$ `&&` $\mid$ `||`
UnaryOperator	$\rightarrow$	`-` $\mid$ `!`

Declarations	$\rightarrow$	*Declaration*
	$\mid$	*Declaration* `;` *Declarations*
Declaration	$\rightarrow$	`const` $\underline{\textit{Identifier}}$ `:` *TypeDenoter* `=` *Expression*
	$\mid$	`var` $\underline{\textit{Identifier}}$ `:` *TypeDenoter*
	$\mid$	`var` $\underline{\textit{Identifier}}$ `:` *TypeDenoter* `:=` *Expression*
	$\mid$	`fun` $\underline{\textit{Identifier}}$ `(` *ArgDecls* `)` `:` *TypeDenoter* `=` *Expression*
	$\mid$	`proc` $\underline{\textit{Identifier}}$ `(` *ArgDecls* `)` *Command*
ArgDecls	$\rightarrow$	ϵ
	$\mid$	$\textit{ArgDecls}_1$
$\textit{ArgDecls}_1$	$\rightarrow$	*ArgDecl*
	$\mid$	*ArgDecl* `,` $\textit{ArgDecls}_1$
ArgDecl	$\rightarrow$	$\underline{\textit{Identifier}}$ `:` *TypeDenoter*
	$\mid$	`in` $\underline{\textit{Identifier}}$ `:` *TypeDenoter*
	$\mid$	`out` $\underline{\textit{Identifier}}$ `:` *TypeDenoter*
	$\mid$	`var` $\underline{\textit{Identifier}}$ `:` *TypeDenoter*
TypeDenoter	$\rightarrow$	$\underline{\textit{Identifier}}$
	$\mid$	*TypeDenoter* `[` $\underline{\textit{IntegerLiteral}}$ `]`

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: $\underline{\textit{Name}} = \underline{\textit{Identifier}} \cup \underline{\textit{Operator}}$.

Program	$\rightarrow$	*Command*	Program
Command	$\rightarrow$	*Expression* `:=` *Expression*	CmdAssign
	$\mid$	*Expression* `(` *Expression** `)`	CmdCall
	$\mid$	`begin` *Command** `end`	CmdSeq
	$\mid$	`if` *Expression* `then` *Command* `else` *Command*	CmdIf
	$\mid$	`while` *Expression* `do` *Command*	CmdWhile
	$\mid$	`let` *Declaration** `in` *Command*	CmdLet
Expression	$\rightarrow$	$\underline{\textit{IntegerLiteral}}$	ExpLitInt
	$\mid$	$\underline{\textit{Name}}$	ExpVar
	$\mid$	*Expression* `(` *Expression** `)`	ExpApp
	$\mid$	`[` *Expression** `]`	ExpAry
	$\mid$	*Expression* `[` *Expression* `]`	ExpIx
Declaration	$\rightarrow$	`const` $\underline{\textit{Name}}$ `:` *TypeDenoter* `=` *Expression*	DeclConst
	$\mid$	`var` $\underline{\textit{Name}}$ `:` *TypeDenoter* (`:=` *Expression* $\mid$ ϵ)	DeclVar
	$\mid$	`fun` $\underline{\textit{Name}}$ `(` *ArgDecl** `)` `:` *TypeDenoter* `=` *Expression*	DeclFun
	$\mid$	`proc` $\underline{\textit{Name}}$ `(` *ArgDecl** `)` *Command*	DeclProc
ArgDecl	$\rightarrow$	*ArgMode* $\underline{\textit{Name}}$ `:` *TypeDenoter*	ArgDecl
ArgMode	$\rightarrow$	ϵ	ByValue
	$\mid$	`in`	ByRefIn
	$\mid$	`out`	ByRefOut
	$\mid$	`var`	ByRefVar
TypeDenoter	$\rightarrow$	$\underline{\textit{Name}}$	TDBaseType
	$\rightarrow$	*TypeDenoter* `[` $\underline{\textit{IntegerLiteral}}$ `]`	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 (false = 0 or 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 (false = 0, 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	$\ldots \Rightarrow \texttt{PC} + 1, \texttt{LB}, 0, \ldots$	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`	$ca, sl, \ldots \Rightarrow \texttt{PC} + 1, \texttt{LB}, sl, \ldots$	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