The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 3 MODULE, AUTUMN SEMESTER 2015–2016

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) This question concerns lexical syntax and analysis.

- What is the part of the compiler that carries out the lexical analysis called, and what are its usual tasks? (5)

  Answer: *[K] The lexical analyser is usually referred to as the* scanner*. It groups individual character from the input text into lexical symbols, or* tokens*, according to the lexical grammar, thus changing the representation of the source program from a sequence of characters into a sequence of tokens. Additionally it reports lexical errors (parts of the input not confirming to the lexical syntax) and (typically) discards white space and comments.*

- Consider the lexical syntax for MiniTriangle given in Appendix A. Suggest a suitable Haskell representation for the lexical symbols or *tokens*. (4)

  Answer: *[A]*

```
data Token = T_comma | T_semicolon | T_colon | T_coleq
           | T_equal | T_lpar | T_rpar | T_lbrk | T_rbrk
           | T_begin | T_const | T_do | T_else | T_end
           | T_fun | T_if | T_in | T_let | T_out
           | T_proc | T_then | T_var | T_while
           | T_intlit Int | T_ident String | T_operator String
```

  There are a range of reasonable possibilities. Anything reasonable yields full marks. Representing tokes by a short string of characters, each token thus being represented by its spelling, is not *reasonable in this setting as this would not facilitate the subsequent parsing, particularly not when it comes to tokens with variable spelling like identifiers and integer literals. While white space and comments typically are discarded, one can imagine applications where they need to be preserved so it is not an error to include tokens also for those.*

- Show how a function carrying out lexical analysis might be implemented in Haskell. Give its type and the implementation for the three lexical symbols `:`, `:=`, and `=` (from Appendix A). No other lexical symbols or tasks need be considered. (6)

  Answer: *[A]*

```
scanner :: [Char] -> [Token]
scanner ...
scanner ':' : '=' : cs = T_coleq : scanner cs
scanner ':' : cs       = T_colon : scanner cs
scanner '=' : cs       = T_equal : scanner cs
scanner ...
```

(b) This question concerns syntactic analysis and construction of an abstract sytnax tree.

- Consider the abstract syntax for MiniTriangle given in Appendix A. Suggest a suitable definition of a Haskell type `Command` to represent commands. You only need to consider the commands assignment, if, while, and sequence. The type `Expression` is used to represent expressions. (4)

  Answer: *[A]*

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

- The following is a Happy parser specification for the above MiniTriangle command selection. The semantic actions for constructing an abstract syntax tree (AST) have been left out (indicated by boxed numbers, 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: *[A]*

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. ($6 \times 1 = 6$*)*

Question 2

(a) Consider the following expression language:

e	$\rightarrow$		*expressions:*
	$\mid$	n	*natural numbers,* $n \in \mathbb{N}$
	$\mid$	x	*variables,* $x \in$ Name
	$\mid$	e `+` e	*addition*
	$\mid$	e `-` e	*subtraction*
	$\mid$	e `*` e	*multiplication*
	$\mid$	e `=` e	*equality test*
	$\mid$	`if` e `then` e `else` e	*conditional*
	$\mid$	`let var` $x = e$ `in` e	*variable definition*
	$\mid$	`let fun` f`(`x`:`t`)`$:t = e$ `in` e	*function definition*
	$\mid$	e`(`e`)`	*function application*

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

t	$\rightarrow$		*types:*
	$\mid$	`Nat`	*natural numbers*
	$\mid$	`Bool`	*Booleans*
	$\mid$	$t \rightarrow t$	*function (arrow) type*

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:

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

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

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

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

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

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

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

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

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

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

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: *[A]*

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

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

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

Answer: *[A]* Let

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

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

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

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

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

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

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

Command	→	...	...
	|	`break` *IntegerLiteral*	CmdBreak

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 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) Explain the following optimization techniques:

- Common Subexpression Elimination
- Loop Unrolling

For each one, outline the idea, illustrate with a small example, and discuss limits on its applicability and any caveats. (10)

Answer: *[C]*

- *Common subexpression elimination avoids evaluating the "same" (sub)expression more than once. For example, a code fragment like*

```
y := (x * x) + (x * x);
```

could be optimised to:

```
t := x * x;
y := t + t;
```

Care has to be taken to ensure that the expressions truly will evaluate to the same value, rather than just being syntactically the same. For example, if an expression has a side effect, like `i++` *in C, it would be wrong to change the number of times such an expression is evaluated.*

- *As loops carry certain overheads (evaluation of loop condition, jumps), it can be beneficial to unroll loops that are known to be short. Also, this could open up for further optimisations. For example:*

```
for (i := 0; i < 5; i++) do
    a[i] := b[4 - i] * 2^i;
```

can be unrolled into:

```
a[0] := b[4 - 0] * 2^0;
a[1] := b[4 - 1] * 2^1;
a[2] := b[4 - 2] * 2^2;
a[3] := b[4 - 3] * 2^3;
a[4] := b[4 - 4] * 2^4;
```

Care has to be taken to ensure the generated code does not grow too large. Besides potentially wasting space, too much code could actually slow down execution compared with a tight loop due to cache effects.

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

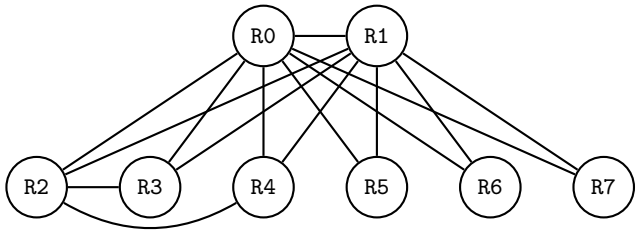
```
        load    R0, 1
        load    R1, 0
loop:   mul     R2, R0, R0
        mul     R3, R0, R0
        mul     R4, R3, R0
        add     R5, R2, R4
        add     R1, R1, R5
        load    R6, 1
        add     R0, R0, R6
        load    R7, 10
        cmp     R0, R7
        ble     loop
```

The `load` instruction in the form used here stores a numeric constant into the designated register. Arithmetic instructions with three register arguments, here `add` and `mul`, perform the arithmetic operation on the two last registers and store the result into the first register. The instruction `ble` is a conditional branch (jump) instruction. The jump is taken if the result of a preceding comparison instruction (here `cmp`) is that the first argument is less than or equal to the second.

(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 make clear what the colour is for each node in the graph and

include the final version of the code, using a minimal number of registers. (6)

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
```

(iii) Briefly explain what needs to be done if the number of registers needed after register allocation exceeds the number of physical registers of the target machine. (3)

Answer: *[K] If the number of registers in use, the* register pressure*, exceeds the number of available registers, the number of registers in use needs to be reduced by storing some of the data in memory. This is known as* register spilling*. Additional instructions have to be inserted to move data between memory and registers as needed.*

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	→	(*Token* | *Separator*)*
Token	→	*Keyword* | *Identifier* | *IntegerLiteral* | *Operator* | `,` | `;` | `:` | `:=` | `=` | `(` | `)` | `[` | `]` | *eot*
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* | *space* | *eol*
Comment	→	`//` (any character except *eol*)* *eol*

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
	|	$Expressions_1$
$Expressions_1$	→	*Expression*
	|	*Expression* `,` $Expressions_1$
Expression	→	*PrimaryExpression*
	|	*Expression BinaryOperator Expression*
PrimaryExpression	→	*IntegerLiteral*
	|	*VarExpression*
	|	*UnaryOperator PrimaryExpression*
	|	*VarExpression* `(` *Expressions* `)`
	|	`[` *Expressions* `]`
	|	`(` *Expression* `)`
VarExpression	→	*Identifier*
	|	*VarExpression* `[` *Expression* `]`
BinaryOperator	→	`^` | `*` | `/` | `+` | `-` | `<` | `<=` | `==` | `!=` | `>=` | `>` | `&&` | `||`
UnaryOperator	→	`-` | `!`

Declarations → *Declaration*
| *Declaration* `;` *Declarations*

Declaration → `const` *Identifier* `:` *TypeDenoter* `=` *Expression*
| `var` *Identifier* `:` *TypeDenoter*
| `var` *Identifier* `:` *TypeDenoter* `:=` *Expression*
| `fun` *Identifier* `(` *ArgDecls* `)` `:` *TypeDenoter* `=` *Expression*
| `proc` *Identifier* `(` *ArgDecls* `)` *Command*

ArgDecls → ϵ
| *ArgDecls*$_1$

ArgDecls$_1$ → *ArgDecl*
| *ArgDecl* `,` *ArgDecls*$_1$

ArgDecl → *Identifier* `:` *TypeDenoter*
| `in` *Identifier* `:` *TypeDenoter*
| `out` *Identifier* `:` *TypeDenoter*
| `var` *Identifier* `:` *TypeDenoter*

TypeDenoter → *Identifier*
| *TypeDenoter* `[` *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: *Name* = *Identifier* ∪ *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	→	*IntegerLiteral*	ExpLitInt
	|	*Name*	ExpVar
	|	*Expression* `(` *Expression** `)`	ExpApp
	|	`[` *Expression** `]`	ExpAry
	|	*Expression* `[` *Expression* `]`	ExpIx
Declaration	→	`const` *Name* `:` *TypeDenoter* `=` *Expression*	DeclConst
	|	`var` *Name* `:` *TypeDenoter* (`:=` *Expression* | ϵ)	DeclVar
	|	`fun` *Name* `(` *ArgDecl** `)` `:` *TypeDenoter* `=` *Expression*	DeclFun
	|	`proc` *Name* `(` *ArgDecl** `)` *Command*	DeclProc
ArgDecl	→	*ArgMode* *Name* `:` *TypeDenoter*	ArgDecl
ArgMode	→	ϵ	ByValue
	|	`in`	ByRefIn
	|	`out`	ByRefOut
	|	`var`	ByRefVar
TypeDenoter	→	*Name*	TDBaseType
	→	*TypeDenoter* `[` *IntegerLiteral* `]`	TDArray