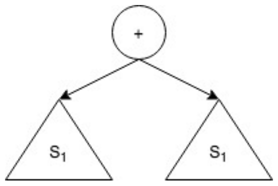
An Abstract Syntax Tree for Nolife

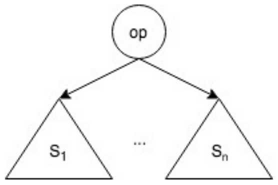
1 Introduction

An *abstract syntax tree* (AST) is a useful intermediate data structure for a compiler. It directly reflects the syntactic structure of the input program. It is more compact than a syntax tree because it doesn't contain artifacts of the grammar like nonterminal symbols.

An AST consists of a *root node* and some number of *subtrees* that are the ASTs for the components of the construct that the root represents. The root node contains a value that indicates the construct being represented along with links to its subtrees. For example, an expression which sums the result of two subexpressions might be represented as:



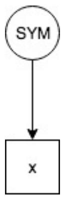
The number of subtrees of a node varies for nodes of different types. In addition, some node types may permit a variable number of subtrees; in other words, different instances of the node type may have different numbers of subtrees. We will depict this situation like this:



The remainder of this document depicts the subtrees that should be built for various Nolife constructs. The pictures are intended to convey structure; they do not dictate an implementation. Be sure that you read the section titled "Implementation Suggestions" and think through the various options before you begin implementing the tree.

2 References

A reference to a symbol is represented by a `SYM` node that indicates the type and the symbol name. The name can be kept as a pointer to a character string, as an index into a string table, or as an index into a symbol table. In this document, we will assume that the name is kept in a character string, and represent it pictorially with a box (rather than a circle). The choice between these strategies is left to the implementor. For example, the following tree represents a reference to the variable x.



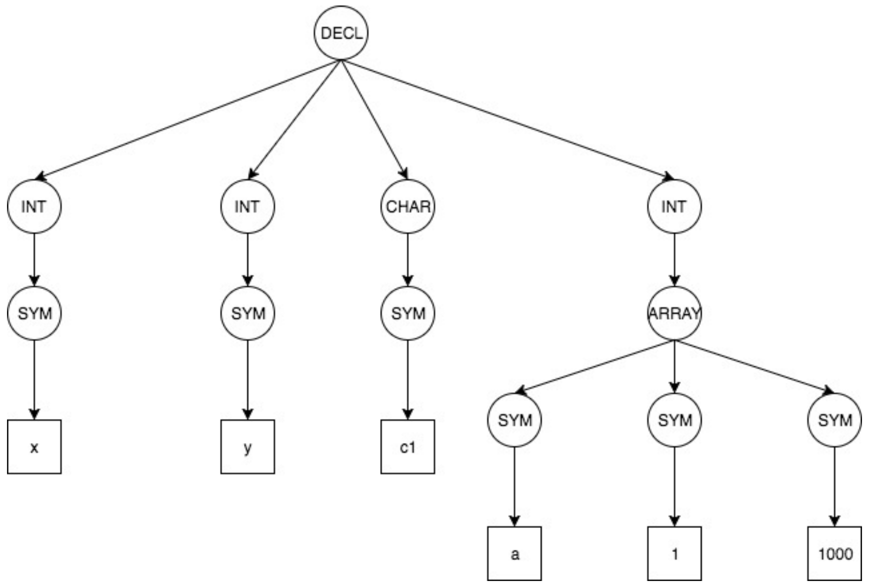
3 Declarations

The entire set of declarations for a procedure are represented by a tree with a `DECL` node as its root. In other words, each declared variable in the procedure gives rise to a single subtree of the `DECL` node. The root of each of these subtrees is a node indicating whether the item declared has type `INTEGER`, `FLOAT` or

CHARACTER. This node will have a single subtree, consisting of either a SYM node to a scalar variable or an ARRAY node. The ARRAY node has subtrees indicating the name, lower bound, and upper bound of the array. For example, the Nolife declaration list

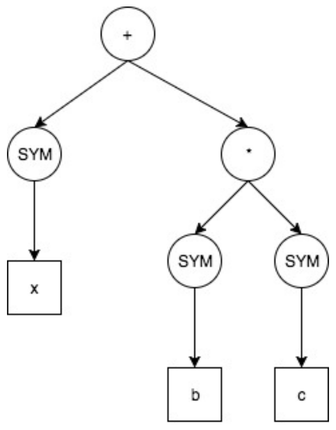
```
VAR x,y :  INTEGER;
   c1 :  CHARACTER;
   a :  ARRAY [ 1 ... 1000 ] OF INTEGER;
```

is represented by the following subtree:

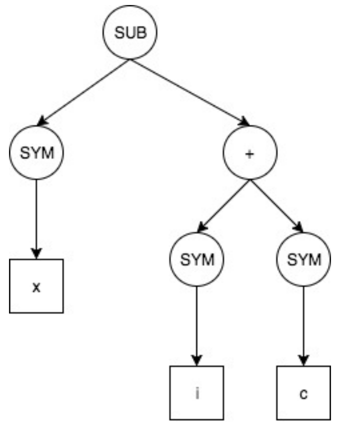


4 Expressions

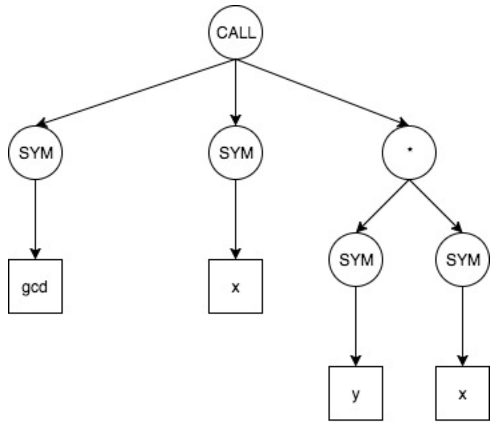
Operators in expressions are represented by nodes with two subtrees. The subtrees, of course, represent the operands. Thus, a + b * c is represented by



Similar nodes are constructed for all of the operators which are valid in Nolife expressions. Other elements of expressions are subscripted variables and function calls. These are represented by `SUBSCRIPT` and `CALL` nodes, respectively. The expression `x[i+1]` is represented by the following tree:

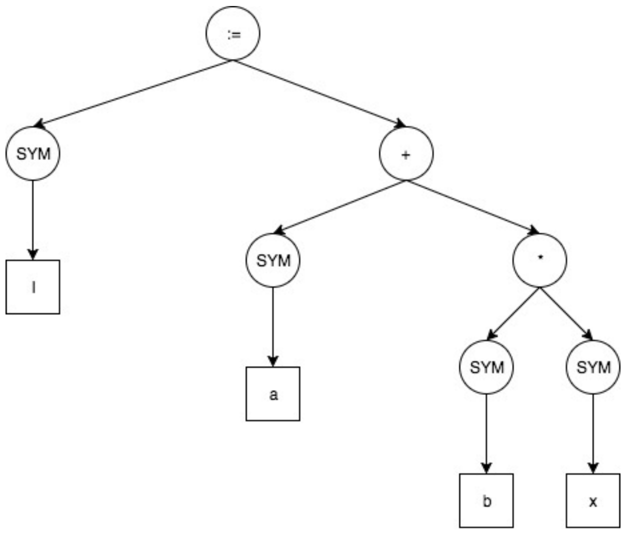


Similarly, the function call `gcd (x, y+x)` is represented by a the tree:

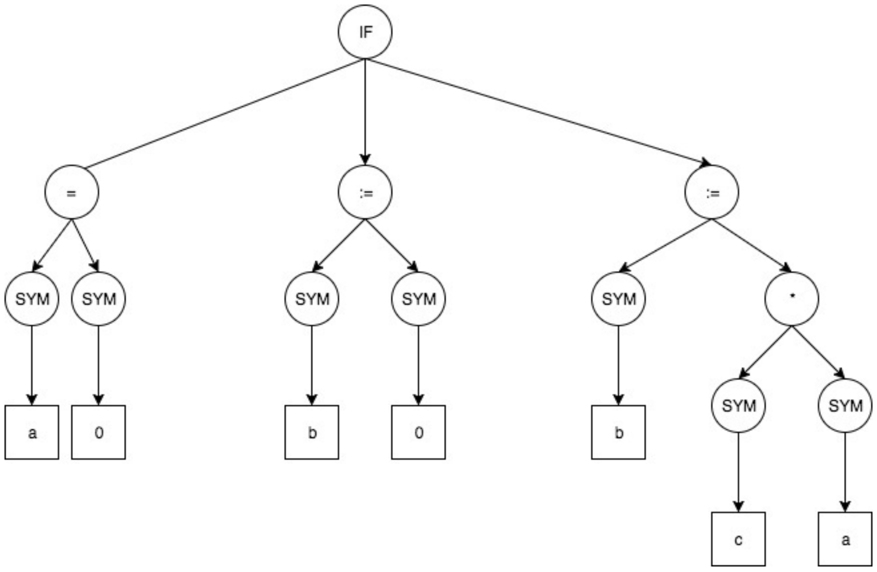


5 Statements

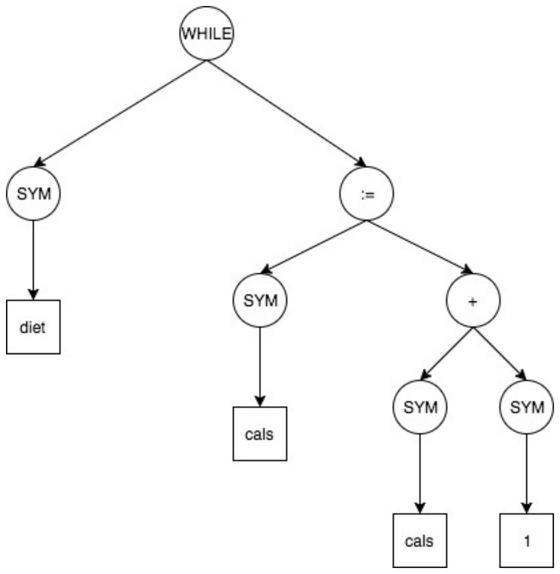
Nolife has the six statement types: *assignment, if, while, case, procedure call, input-output*, and *compound.* Each of these statements is represented in the AST by a different node. These are described below. The assignment statement is represented as a binary tree with the left and right hand sides as subtrees. For example, the statement `l := a + b * c;` would be represented:



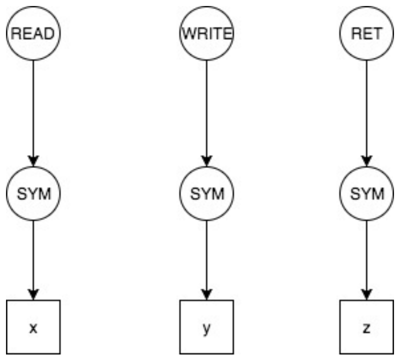
The if statement is represented by a tree with up to three subtrees. One subtree represents the test expression, one represents the `THEN` statement, and one represents the (optional) `ELSE` statement. Hence, the if statement `IF a = 0 THEN b := 0 ELSE b := c * a` gives rise to the following tree:



The while loop is represented by a binary tree, with one subtree for the expression governing loop execution, and another for the controlled statement. Hence `WHILE diet DO cals := cals + 1` would be represented by:



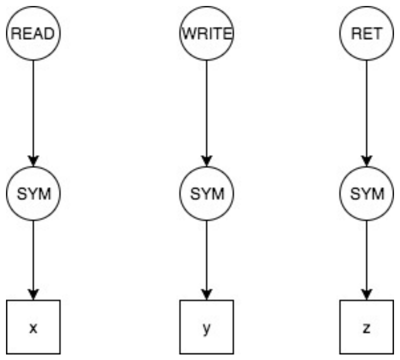
The input, output, and return statements are each represented by simple unary nodes.



The compound statement is represented by a tree which has a subtree for every statement which appears inside it. Thus

```
BEGIN
   a := 0;
   b := 1
END
```

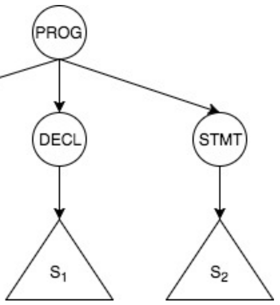
is represented by:



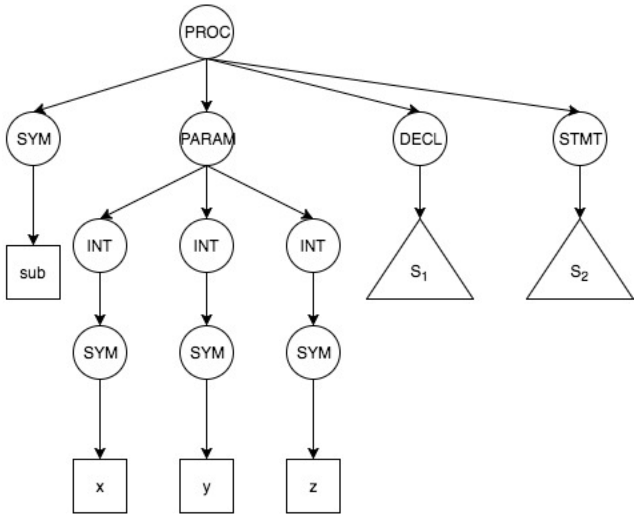
A procedure call statement is represented by a subtree that is identical to the one used for a function call in an expression.

6 Procedure Declarations

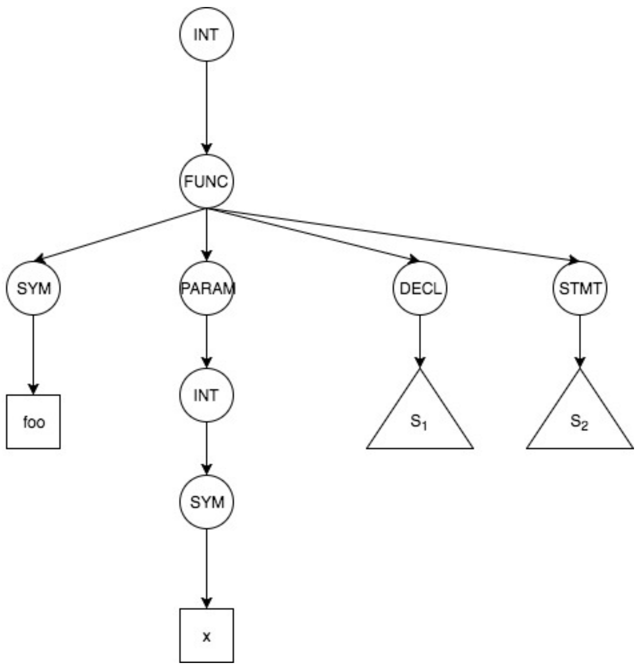
An Nolife program is just a header statement, a set of declarations, a set (possibly empty) of subprogram declarations, and a compound statement. To represent a program, then, we need a node which has sons for the information in the header statement, for the declarations, and for the compound statement. For a program headed by `PROGRAM main;` the parser should build this tree:



The subtree labelled S_1 contains anything declared in the main program — both variable declarations and any subprograms. A subprogram has the same structure as a program except that the parameters must be specified as subtrees of a PARAM tree, So that the declaration `PROCEDURE sub (x, y, z: INTEGER);` would have the representation:



A function declaration is the same as a procedure declaration except that it is represented as a typed tree (INT, FLOAT, or CHAR node at the root) with a single subtree with a procedure node at the root. For example,
`FUNCTION foo (x: INTEGER) :INTEGER;` would be represented as:



7 Example Program

The following program is a simple example program written in Nolife. The tree built from this program is shown on the following page.

```
PROGRAM ex;
   VAR x, y :  INTEGER;
   FUNCTION gcd (a,b:  INTEGER):INTEGER;
      BEGIN
         IF b=0
            THEN gcd := a
            ELSE gcd := gcd(b, a MOD b)
      END;
   BEGIN
      READ (x);
      READ (y);
      WHILE (x <> 0) OR (y <> 0) DO
         BEGIN
            WRITE (gcd (x,y));
            READ (x);
            READ (y)
         END
   END
```

The tree for this program is shown in Figure 1 below.

8 Implementation Suggestions

Your task is to implement an AST that represents the somewhat abstract view presented in this document. Experience in tree implementations suggests that there are several key issues for you to consider before

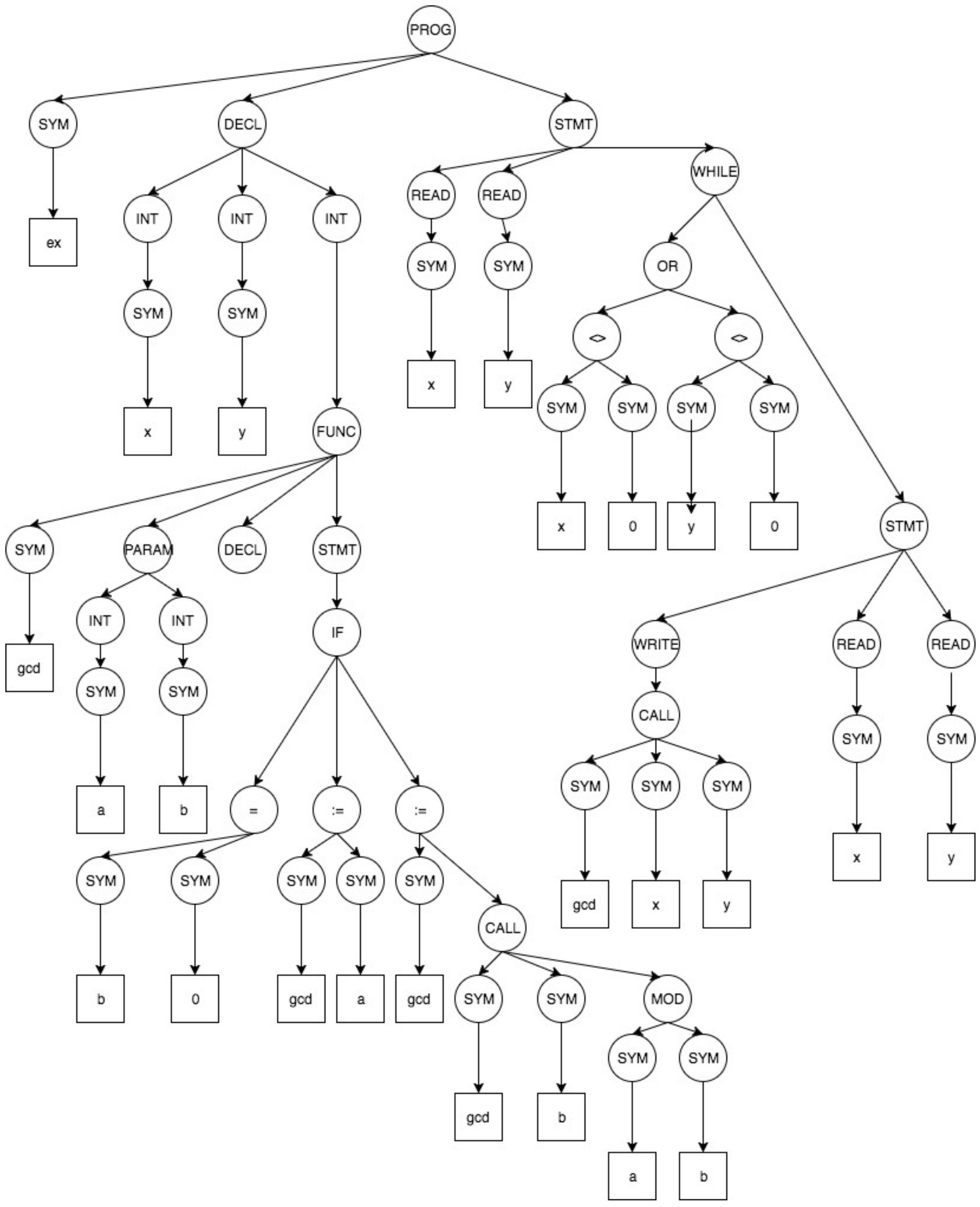


Figure 1: example program

committing a design to code.

8.1 Arity

As drawn, the AST has a number of nodes that carry a variable number of children. As examples, consider the `DECL` and `STMT` nodes. A `DECL` node has one child for each local variable and for each procedure declared in that scope. A `STMT` node has one child for each statement in the compound statement.

One alternative is to delay allocation of the node until the number of sons is known and to allocate a node of appropriate size. (Of course, such a node will probably include a counter for the number of children.)

Another alternative is to hand a variable-dimension array of pointers or indices from the node, adding a level of indirection in the implementation that is not shown in the drawings. This simplifies a number of issues, like naming the sons and allocating the nodes.

A final option is to use an old trick described in Knuth Volume 1 (around page 332). This technique would give each node a single pointer (or index) for the first child, as well as a pointer (or index) for the next sibling. In this scheme, to traverse a node's children, the treewalker goes down to the first child and then moves laterally among the siblings.

8.2 Missing Children

Occasions will arise where a specified child of some node is unneeded. (For example, look at the `DECL` subtree in the declaration of routine `gcd` in the example program.) You must decide whether to represent these nodes explicitly or implicitly. You should be consistent across node types.

8.3 Number of Supported Nodes

This document describes a fair number of distinct node types. The sheer number of types requires some management. Two obvious choices present themselves:

1. You can declare distinct structures for each node type and fill them in, using some sort of `union` in `C` to allow you to distinguish between them. This minimizes wasted space in nodes that have few fields.

2. You can declare a single node type and interpret its contents based on finding a known value in a `node type` field. This requires that the standard node have all the fields needed in *any* node, but simplifies allocation and coding.

Either choice is acceptable. You should decide which coding style fits you best.

8.4 Printing the Tree

Your parser is required to produce a printed listing of the tree. The tree listing should be printed to `stdout`, `cout` or `System.out` as appropriate for your implementation language. Each node should be displayed on its own line. For each node, the listing should show the node's type (`SYM`, `DECL`, etc.) and any associated string value (a name, a constant).

Relationships between nodes should be represented by indentation. Thus, the root node for a tree would be printed in column k. Its children would be printed in column $k+c$, where c is a small integer (like 3 or 4). The nodes should be listed in *preorder*, that is, a node is printed before any of its children.

Left to right ordering, as shown pictorially in the examples, should be observed, with nodes printed before nodes to their right. This dictates that, for example, the subtrees for successive statements will appear in the listing in the same order that occurred in the input program.

9 Tree Grammar for the AST

This section gives a tree grammar that approximates the graphical representations given above.

Program	$\rightarrow$	`PROGRAM` *Identifier*; *Decls* *SubprogramDecls* *CompoundStatement*	P	$\rightarrow$	$\#(\$prog\ S\ V\ F\ C)$
Decls	$\rightarrow$	`VAR` *DeclList*	V	$\rightarrow$	$\#(\$decl\ (T\ S)^*)$
	$\mid$	ϵ			
DeclList	$\rightarrow$	*IdentifierList* : *Type* ;			
	$\mid$	*DeclList* *IdentifierList* : *Type* ;			
IdentifierList	$\rightarrow$	*Identifier*	S	$\rightarrow$	$\#(\$sym\ @str)$
	$\mid$	*IdentifierList* , *Identifier*			
Type	$\rightarrow$	*StandardType*			
	$\mid$	*ArrayType*			
StandardType	$\rightarrow$	`INTEGER`	T	$\rightarrow$	$\#(\$int)$
	$\mid$	`FLOAT`		$\mid$	$\#(\$float)$
	$\mid$	`CHARACTER`		$\mid$	$\#(\$char)$
ArrayType	$\rightarrow$	`ARRAY [` *Dim* `] OF` *StandardType*	T	$\rightarrow$	$\#(\$array\ S\ S\ T)$
Dim	$\rightarrow$	*Intnum* .. *Intnum*			
	$\mid$	*CharConstant* .. *CharConstant*			
SubprogramDecls	$\rightarrow$	*SubprogramDecls* *SubprogramDecl* ;	F	$\rightarrow$	$\#(\$decl\ P_r^*)$
	$\mid$	ϵ			
SubprogramDecl	$\rightarrow$	*SubprogramHead* *Decls* *CompoundStatement*			
SubprogramHead	$\rightarrow$	`FUNCTION` *Identifier* *Arguments*: *StandardType* ;	P_r	$\rightarrow$	$\#(\$func\ T\ P_f\ C)$
	$\mid$	`PROCEDURE` *Identifier* *Arguments* ;		$\mid$	$\#(\$proc\ P_f\ C)$
Arguments	$\rightarrow$	(*ParameterList*)			
	$\mid$	ϵ			
ParameterList	$\rightarrow$	*IdentifierList* : *Type*	V	$\rightarrow$	$\#(\$param\ (T\ S)^*)$
	$\mid$	*ParameterList* ; *IdentifierList* : *Type*			
Statement	$\rightarrow$	*Assignment*			
	$\mid$	*IfStatement*			
	$\mid$	*WhileStatement*			
	$\mid$	*CaseStatement*			
	$\mid$	*ProcedureInvocation*			
	$\mid$	*IOStatement*			
	$\mid$	*CompoundStatement*			
	$\mid$	*ReturnStatement*			
Assignment	$\rightarrow$	*Variable* `:=` *Expr*	S_t	$\rightarrow$	$\#(\$:=\ S\ E)$
IfStatement	$\rightarrow$	`IF` *Expr* `THEN` *Statement* `ELSE` *Statement*	S_t	$\rightarrow$	$\#(\$if\ E\ C\ C)$
	$\mid$	`IF` *Expr* `THEN` *Statement*		$\mid$	$\#(\$if\ E\ C)$
WhileStatement	$\rightarrow$	`WHILE` *Expr* `DO` *Statement*	S_t	$\rightarrow$	$\#(\$while\ E\ C)$
CaseStatement	$\rightarrow$	`CASE` *Expr* `OF` *Cases* `END`	S_t	$\rightarrow$	$\#(\$case\ E\ C_l^+)$
Cases	$\rightarrow$	*CaseElement*			
	$\mid$	*Cases* ; *CaseElement*			

CaseElement	$\rightarrow$	*CaseLabels* : *Statement*	C_l	$\rightarrow$	$\#(\$clause\ S^+\ C)$
	$\mid$	ϵ			
CaseLabels	$\rightarrow$	*Constant*			
	$\mid$	*CaseLabels* , *Constant*			
ProcedureInvocation	$\rightarrow$	*Identifier*`()`	S_t	$\rightarrow$	$\#(\$call\ S\ E^*)$
	$\mid$	*Identifier* `(` *ExprList* `)`			
IOStatement	$\rightarrow$	`READ` `(` *Variable* `)`	S_t	$\rightarrow$	$\#(\$read\ E)$
	$\mid$	`WRITE` `(` *Expr* `)`	S_t	$\rightarrow$	$\#(\$writeE)$
	$\mid$	`WRITE` `(` *StringConstant* `)`			
CompoundStatement	$\rightarrow$	`BEGIN` *StatementList* `END`	C	$\rightarrow$	$\#(\$stmt\ S_t^+)$
StatementList	$\rightarrow$	*Statement*			
	$\mid$	*StatementList* `;` *Statement*			
ReturnStatement	$\rightarrow$	`RETURN` *Expr*	S_t	$\rightarrow$	$\#(\$retE)$
ExprList	$\rightarrow$	*Expr*			
	$\mid$	*ExprList* `,` *Expr*			
Expr	$\rightarrow$	*Expr Op Expr*	E	$\rightarrow$	$\#(O\ E\ E)$
	$\mid$	`NOT` *Factor*		$\mid$	$\#(\$not\ E)$
	$\mid$	*Factor*			
Op	$\rightarrow$	*LogOp*	O	$\rightarrow$	$\$and \mid \$or$
	$\mid$	*Relop*		$\mid$	$\$< \mid \$<= \mid \$>= \mid \$> \mid \$= \mid \$<>$
	$\mid$	*AddOp*		$\mid$	$\$+ \mid \$-$
	$\mid$	*MulOp*		$\mid$	$\$* \mid \$mod$
Factor	$\rightarrow$	*Variable*			
	$\mid$	*Constant*			
	$\mid$	`(` *Expr* `)`			
	$\mid$	*ProcedureInvocation*			
Variable	$\rightarrow$	*Identifier*			
	$\mid$	*Identifier* `[` *Expr* `]`			
Identifer			S	$\rightarrow$	$\#(\$sym\ @str)$
Constant			S	$\rightarrow$	$\#(\$sym\ @int)$
				$\mid$	$\#(\$sym\ @float)$
StringConstant			S	$\rightarrow$	$\#(\$sym\ @literal)$