# THE JASON PROGRAMMING LANGUAGE, AN AID IN TEACHING COMPILER CONSTRUCTION

*Robert M. Siegfried*
*Computer Science Department*
*Saint Peter's College*
*2641 John F. Kennedy Blvd.*
*Jersey City, NJ 07306*
*SIEGFRIED_R@spcvxa.spc.edu*

## ABSTRACT

Compiler construction is regarded as an important part of a computer science education, offered in most undergraduate programs and required in most graduate programs. The course covers the design and implementation of an important class of programs, as well as many concepts that are considered important to a computer science education. However, compiler construction is not always an easy course to teach to undergraduates, particularly in a small college environment. Examples can be frustrating with expression grammars being too small and subsets of languages such as Pascal being too complex.

The JASON (**J**ust **A**nother **S**imple **O**riginal **N**otion) programming language is an ALGOL-derived, limited-purpose, programming language specifically designed to illustrate the principles of compiler construction. It contains all the important concepts of procedural programming languages except for aggregate data types. As a result, a JASON compiler is small enough for undergraduates to understand all the aspects of its implementation, and comprehensive enough to allow students to extrapolate and understand how to design larger-scale compilers.

The specifications for the JASON language are given, and sample programs are shown. Sample programming projects are described for expanding JASON to include features not currently contained in the language such as functions and character and aggregate data types.

Presently, there are two front-ends written for a JASON compiler, one using a table-driven LL(1) parsing algorithm and the other using recursive descent, both producing a translation of the source program in quadruples. A summary of the design and organization for both front ends is given as well as initial design for the back end.

## INTRODUCTION

Compiler construction is generally regarded as an important course in a computer science education. Most graduate programs require a course in the subject*. Following the recommendation of the ACM, most undergraduate programs offer it as an advanced elective.1 On the graduate level, it is usually offered as a two-semester sequence, while most undergraduate courses in compiler construction concentrate on lexical and syntactic analysis.

Compiler construction is considered a fairly important course for several reasons. Compilers are considered important programs that are used daily by computer professionals. Their design is one of the oldest fields in computer science, intimately touching on several other areas of the discipline, such as programming language design, formal language theory, algorithmic analysis, computer architecture and software engineering. Because even a simple compiler can have between 2,000 and 10,000 lines of source code, it gives students an opportunity to learn how to develop larger programs. Lastly, compiler construction methods are used in the development of command languages, which is a feature included in a wide range of commercial software.

Compiler construction, however, is not a particularly easy course to teach to undergraduates. While many undergraduates study compiler construction, very few of the textbooks on the subject are written specifically for them. The classic text on the subject, Compilers: Principles, Techniques, and Tools by Alfred Aho, Ravi Sethi and Jeffrey Ullman (better known as the "Dragon book") is too difficult for most undergraduates. Thomas Parsons goes so far as to mention in the foreword of his book that it is intended to prepare students for the "Dragon book."2

Examples, too, can be frustrating, largely because they are too small to be useful in guiding a student through the process of working with a full programming language's grammar. Some textbooks use small symbolic grammars containing a handful of nonterminals and terminals that bear no resemblance to anything that one might see in a real programming language. Expression grammars are popular because they are a component of most programming languages and because of their highly recursive nature. But while they are useful in teaching certain principles, they cannot by themselves prepare a student for the larger-scale job of creating a parser or implementing semantic actions for a useful programming language, no matter how small it may be.

Additionally, expression grammars present an additional problem when teaching bottom-up parsing. Because of their extremely recursive nature, the initial state for such grammars will have an item for every production in the language. While this is true for most expression grammars, it is not usually the case for commercial programming languages. The author has used fairly simple grammars as in-class examples, homework problems and exam questions where there are only a small number of items in the initial

* An informal survey by the author of several graduate programs, primarily but not exclusively in the New York Metropolitan area, found that compiler was a core or distributive requirement or an admission requirement for a Master's degree for most graduate programs in computer science.

states; nevertheless, students frequently place an item for every production in the initial state even when it is not necessary to do so. Several students have stated that they believed that this was necessary because they used a state machine for an expression grammar as a guide.

All this points to a great need for a good source language to be used as an example when teaching compiler construction. Obviously, such a language should be as simple as possible, but must include the most important features in a programming language. Because even simple programs need input and output, statements supporting these must be included. Similarly, a source language must have an assignment statement. Because selection and repetition are essential for any real algorithm, a programming language needs a WHILE statement and an IF-THEN-ELSE construction.

Although procedures are not essential for the implementation of algorithms, sound software engineering principles demand them as well as a method for parameter passing. Passing parameters by reference has the advantage of allowing values to be returned to the main program or calling procedure; this allows us to omit passing parameters by value as well as including functions in this sample language. While functions are useful in serious programming, they introduce unnecessary complications in a language whose sole purpose is to illustrate compiler construction.

Lastly, the source language must have two data types. While algorithms could be implemented with only one data type (real numbers would be practical for numeric algorithms), a second data type is needed if students are going to learn anything about type checking; more than two data types starts to make the language too complex. It seems most practical to use integers and real numbers because integer values are compatible with real variables but the reverse is not true.

## THE JASON LANGUAGE

The JASON language began as a parsing problem for a compiler construction class. Recognizing its potential as a teaching tool, the author wrote a front end for the original syntax, which was extremely limited, and expanded it to the current language, expanding the compiler as necessary. Since the original syntax specified a "baby" language, the author named it for his then-four-month-old son. The name is also an acronym for "Just Another Simple Original Notion."

The basic syntax for a JASON program is:

```
PROGRAM ProgramName;
  Procedures, if any
  DECLARE
    DataType IdList;
      ....
  BEGIN
    Statement(s)
  END.
```

The reserved word **DECLARE** is not used if there are no variables or procedures being declared. JASON has two valid data types, REAL and INTEGER. An IdList is one or more identifiers separated by commas. Procedures have the following syntax:

```
PROCEDURE ProcName;
  PARAMETERS
    DataType Identifier;
      ....
  Procedures, if any
  DECLARE
    DataType  IdList;
      ....
  BEGIN
    Statement(s)
  END;
```

The reserved word PARAMETERS is not used if there are no parameters being declared. The language is case-insensitive and all key words are reserved; they are shown in upper case for emphasis.

JASON has the seven valid types of statements:

```
READ Identifier
WRITE Identifier
SET Identifier = Expression
IF Condition THEN Statement(s) ELSE Statement(s) ENDIF
WHILE Condition DO Statement(s) ENDWHILE
UNTIL Condition DO Statement(s) ENDUNTIL
CALL Identifier(ArgList)
```

The parentheses are not used in a **CALL** statement if no parameters are being passed.

JASON includes four arithmetic operators (+, -, *, and /) and four relational operators (=, >, <, and !). The exclamation point (!) is used to connote "is not equal to" so that the entire operator set consists of one-character operators. Comments are enclosed in braces, e.g.,

```
{ This is a comment in JASON }
```

Below is an example of a program written in JASON:

```
PROGRAM  EvalFormula;
  { Calculates the result of a basic formula
    Illustrates the use of procedures in JASON }
  DECLARE
    INTEGER a;
    REAL b;
  { The procedure which evaluates the formula repeatedly }
  PROCEDURE FindFormula;
    PARAMETERS
      INTEGER x;
      REAL y;
```

```
    BEGIN
      WHILE x ! 0 DO
        IF x < 0 THEN SET y = 10 - 4.5*x
                 ELSE SET y = 4.5*x + 10
        ENDIF;
        WRITE y;
        READ x
    END;
  { Main program }
  BEGIN
    READ a;
    CALL FindFormula(a, b);
    WRITE b
  END.
```

There are several advantages in using JASON as a source language when in teaching compiler construction. Firstly, the language's syntax is completely predictive; every statement begins with a reserved word and variable declarations begin with the data type. This simplifies the process of creating the table for top-down parsing and for coding the recursive descent parser, which makes it easy for students to follow the compiler development process.

JASON is a relatively small language, with only 27 nonterminals and 39 tokens. The entire language is specified in BNF in only 54 productions, making it much smaller than Pascal or C. Both front ends are under 3000 lines of source code written in C, making it fairly easy for undergraduates to understand the code.

Robert Sebesta points out that the use of a reserved word for selection closure makes if-then-else statements more regular, especially when there are compound if-then-else constructs.3 He also observes that ENDIF provides more information than END does by itself. For this reason, JASON closes IF-THEN-ELSE statements with the reserved word ENDIF and similarly provides the reserved words ENDWHILE and ENDUNTIL to close these respective loops, making programs in JASON easier to read. While the use of these ENDIF, ENDWHILE and ENDUNTIL added a few extra tokens to the language, it proves quite valuable in debugging JASON programs and in debugging projects that expand the capabilities of the JASON compiler.

As noted above, operators in JASON all have one character only. This simplifies the scanner, in keeping the design philosophy that "simple is beautiful" and allows an instructor to assign the simple scanner project of enhancing JASON by adding several two-character operators, such as >=, <=, and !=.

Lastly, JASON programs are fairly easy to read as well as to write, simplifying the task of writing programs for the compiler to use as sample data. This becomes particularly important if students are given the assignment of extending the compiler. It is particularly frustrating if one cannot tell if the error is in the compiler's source code or in the sample program that the compiler is trying to read. The simpler the language is, the less likely it is that this problem will occur.

## THE JASON COMPILER

The JASON compiler consists of 4 source files:

| | |
|---|---|
| **jasonX.c** | version X of the parser (complete with semantic actions) |
| **scan.c** | the scanner |
| **symbol.c** | the symbol table manager |
| **quad.c** | the intermediate code generator |

with declarations in the header files **scan.h**, **symbol.h** and **quad.h**. There are actually two parsers, the table-driven LL(1) parser (**jason0.c**) and the recursive descent parser (**jason4.c**) which can be used interchangeably and will generate identical intermediate code for the same JASON programs.

The scanner uses three different functions to scan words, numbers and operators as well as a fourth function to skip past comments and white space. This modular structure makes it easy it for students to relate the finite automaton that recognizes the lexical elements of the language to the source code of the scanner.

The symbol table consists of four structures:

- a string table, containing the lexemes as one long array of characters.
- a name table, containing the starting position of the lexemes within the string as their lengths.
- an attribute table, containing the various attributes of the lexemes, including the token associated with it.
- a hash table, which points to the name table.

Figure 1 shows the basic symbol table structure.

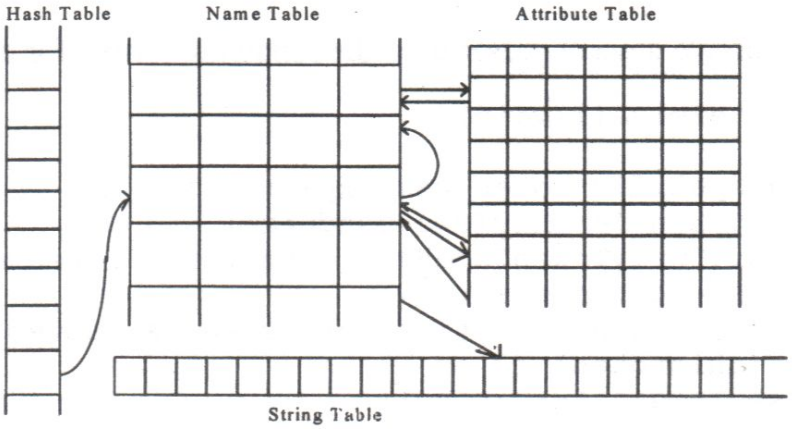


Figure 1 - The basic organization of the symbol table.

Any assignment involving the addition of arrays and records to JASON requires the addition of an extra table, containing a list of pointers to other attribute table entries. This also necessitates the addition of two extra fields to the attribute table, holding the starting position within the extra table and the number of entries.

Because ALGOL-style name scoping rules apply to JASON, the symbol table must have the capacity to open and close scopes as required by the source program. This is supported by a series of pointers, which connect an attribute table entry to all other entries in its scope as well as to other entries with outer scopes that share its lexeme. Figure 2 illustrates how name scoping is implemented within the symbol table.

The recursive descent parser was written in five discrete stages, adding additional features to the language with each new stage and then having the necessary semantic actions added. This helped ensure that programs were parsed correctly and that the correct intermediate code was generated. After the front end was completed, the table-driven parser was written and tested for the entire language and then the semantic actions were added. When this was completed, further testing was done to ensure that the two programs generated the same intermediate code for the same JASON programs.

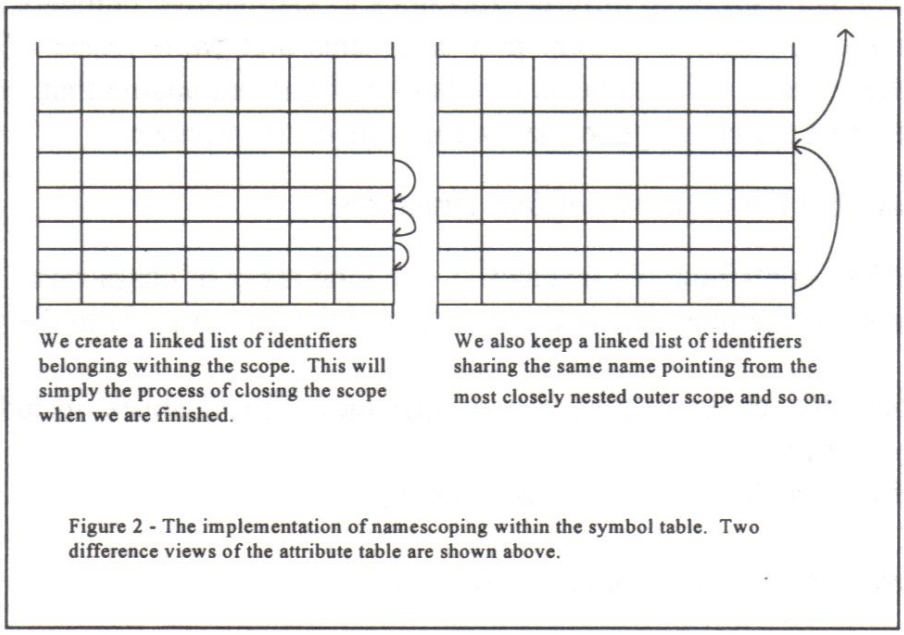


Figure 2 - The implementation of namescoping within the symbol table. Two difference views of the attribute table are shown above.

The recursive descent parser used the following format for all procedures:

```
void ProcNonterminal1(void)
{
    /* If a specific token was expected */
    if (thistoken != OnlyTokenUsedHere)
        error("OnlyTokenUsedHere expected", linenum);
    thistoken = gettoken();
    /* If one of n alternative tokens was expected */
    switch (thistoken) {
        case TokenAlternative1:
        case TokenAlternative2:
            ...
       case TokenAlternativeN: thistoken = gettoken(); break;
       default: error("Expected one of these tokens", linenum);
    }
    /* If a nonterminal was expected */
    ProcNonterminal2();
}
```

The semantic actions were added to these functions in the appropriate places.

The table-driven parser was based on an algorithm in the "Dragon book."4

The grammar of the language was stored in three structures. The parse table is a 27 by 39 array containing the number of the production where the right sentential form derived from a given nonterminal begins with a given token and zeros for all other possible tokens. There is also an array containing the right sentential forms of all the productions, excluding epsilon- productions and an array containing the starting position of each production within the array of right sentential forms as well as the number of variables in that right sentential form. After testing to ensure that it parsed programs correctly, the semantic actions were added as well as the function that calls the proper semantic routine.

The intermediate code chosen was quadruples, which are stored as an array of records, each of which containing an opcode followed by three addresses. These addresses are themselves a record, containing the type entry (variable, constant, or label) followed by a pointer to its attribute table entry. There is also a function which prints the intermediate code in easy-to-read format.

## POTENTIAL PROJECTS

Although the JASON compiler's front end and the source language that it translates can be considered complete enough to use as a teaching example, there are several additional features that can be added to JASON as a student assignment. They include:

### Compound Conditions

The JASON language currently allows for simple conditions only. Adding the conjunctive logical operators & (AND) and \ (OR) requires a minor change to the symbol table manager to include the additional tokens and replacing the production

Condition ::= Expression RelOp Expression

with the productions

Condition ::= Condition \ CompoundCondition
Condition ::= CompoundCondition
CompoundCondition ::= CompoundCondition & SimpleCondition
CompoundCondition ::= SimpleCondition
SimpleCondition ::= Expression RelOp Expression

The necessary semantic actions must also be added. The rest of the grammar is unaffected by this change.

### Character Strings

All character strings are assumed to hold up to 255, using an array of 256 bytes to hold the current length of the string and the string itself. String literals are enclosed in quotation marks, which are not allowed to appear in strings themselves. This requires students to extend the scanner to allow it to read quoted strings, to add the reserved word **STRING** to the symbol table manager and to add the necessary semantic action to handle type checking and the assignment of values to string variables. A more ambitious project would also include implementing concatenation using the + sign and the functions **POSITION**(s, t), which returns the position of a substring t within the string s and **SUBSTRING**(s, i, j) which returns a substring beginning at position i that includes j characters. A program in extended JASON involving string is shown below:

```
PROGRAM StringSample;
  VARIABLES
    STRING Name, First;
    INTEGER      FirstLength
   BEGIN
      SET Name  = "John Smith";
      SET FirstLength = POSITION(Name, " ") - 1;
      SET First = SUBSTRING(Name, 1, FirstLength);
      WRITE First
    END.
```

The keyword DECLARE is changed to VARIABLES for reasons discussed below.

### Arrays and records

Arrays and records are more involved and are best implemented using a user-defined type. The syntax for programs using type declarations becomes:

```
TYPES
   TypeDeclarations
    ....
```

where array type declarations and record type declarations can be appear in whatever order or combination as the user wishes. Declaring an array type has the general form

```
DataTypeName     ARRAY DataType[Size];
```

where DataType can be either INTEGER, REAL, STRING (if added to the language) or any user-defined type. Similarly, declaring a record has the general form:

```
DataType RECORD
            Variable declarations
         END;
```

where field declarations can be grouped or declared one at a time depending on the user's needs. In adding data types to JASON, the keyword DECLARE is replaced with VARIABLES to make the language a but more readable. A program using records and arrays might look like this:

```
PROGRAM ExtendedSample;
  { Illustrates the use of records and arrays }
  TYPES
    ARRAY Arraytype INTEGER[5];
    RECORD RecordType
      STRING  Name;
               ArrayType Grades;
           END;
  VARIABLES
    RecordType StudentRecord;
   BEGIN
    READ StudentRecord.Name
    READ StudentRecord.Grades[1];
    WRITE StudentRecordGrades[1];
  END.
```

The implementation of type declarations requires the addition of an auxiliary table to hold the size of an array and pointers to the attribute table entries for the fields of a record, as discussed above. The current attribute table uses an enumerated type to store the data type of a variable or constant; the inclusion of user-defined types requires that variable types be stored as a pointer to the appropriate attribute table entry. Implementation of type declarations also requires modifying the grammar to allow for the nonterminal variable, which can include an index, a record field or both. The changes in the grammar appear in Appendix 2.

## Functions

In general, the syntax for a function is:

```
FUNCTION FunctionName RETURNS DataType;
  PARAMETERS
    DataType Identifier;
      ....
  DECLARE (or VARIABLES)
    DataType  IdList;
      ....
  BEGIN
    Statement(s);
    RETURN(Variable)
  END;
```

A sample function appears below:

```
FUNCTION Average3 RETURNS REAL;
    PARAMETERS
      INTEGER a;
      INTEGER b;
      INTEGER c;
    VARIABLES
      REAL Sum, Average;
    BEGIN
      SET Sum = a + b + c;
      SET Average = Sum/3;
      RETURN(Average)
    END;
```

## PLANS FOR THE BACK END

At the present time, there is no back end for the JASON compiler. When written, it will consist of two parts: a code generator and an optimizer.

The code generator will create a version of the program in PC assembler. Despite the popularity of the Intel 80x86 family of processors and MS-DOS/Windows as a platform, the choice is not so obvious given the fact that the early processors in the 80x86 family did not include floating point arithmetic. However, given the fact that the 486 and Pentium processors all include a floating point unit, this should not be a problem. Assembly language was chosen as the target language because of its greater readability; machine code for the Intel processors is notoriously difficult to read. Assembly language allows students the opportunity to see the result of compiling a JASON program more clearly manner than they would if machine language was used as the target language.

Only local block optimization is planned for both the intermediate and final code. Although few undergraduate courses in compiler construction cover optimization in any detail, it is too important to be omitted from any compilers that computer science students are going to study.

## SOURCE CODE FOR JASON FRONT ENDS

Anyone wishing a copy of source code for the JASON compiler front ends should contact the author at the above address.

## ACKNOWLEDGMENTS

Niraj C. Shah of the IBM Corporation wrote part of the code for the first two stages of the recursive descent parser; the author wants to thank him for his assistance.

This project was completed while on a sabbatical. The author would like to acknowledge this support from Saint Peter's College.

## REFERENCES

[1] Allen B. Tucker, ed., **Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force**, Association for Computing Machinery, New York, 1990.

[2] Thomas W. Parsons, **Introduction to Compiler Construction**, Computer Science Press, New York, 1992, p. xiii.

[3] Robert W. Sebesta, **Concepts of Programming Languages**, 2nd edition, Benjamin/Cummings Publishing Company, Reading, MA, 1993, pp. 264-265.

[4] Alfred Aho, Ravi Sethi and Jeffrey Ullman, **Compilers: Principles, Techniques, and Tools**, Addison-Wesley Publishing Company, Reading MA, 1986, p. 187.

## APPENDIX 1
## THE LEXICAL ELEMENTS AND BNF GRAMMAR OF JASON

The classes of token in JASON are:

Keyword (each keyword is a distinct token)
identifier
constant (numeric literal)
operator (a single character)

The keywords of JASON are:

| | | | |
|---|---|---|---|
| BEGIN | ENDIF | PARAMETERS | SET |
| CALL | ENDUNTIL | PROCEDURE | THEN |
| DECLARE | ENDWHILE | PROGRAM | UNTIL |
| DO | IF | READ | WHILE |
| ELSE | INTEGER | REAL | WRITE |
| END | | | |

Identifer ::= Letter (Letter | Digit)*

Letter is one of

A B C D E F G H I J K L M N O P Q R S T U V W Y Z

where all letters are converted to upper while scanning.

Digit is one of 0 1 2 3 4 5 6 7 8 9

Constant ::= Digit* {.} Digit*

Operator is one of . ; , ( ) = < > ! + - * /

Program ::= Header DeclSec Block .
Header ::= program identifier ;
DeclSec ::= declare VarDecls ProcDecls | <nil>
VarDecls ::= VarDecls VarDecl | VarDecl
VarDecl ::= DataType IdList ;
DataType ::= real | integer
IdList ::= IdList , identifier | identifier
ProcDecls ::= ProcDecls ProcDecl | ProcDecl | <nil>
ProcDecl ::= ProcHeader ProcDeclSec Block ;
ProcHeader ::= procedure identifier ;
ProcDeclSec ::= ParamDeclSec DeclSec
ParamDeclSec ::= parameters ParamDecls | <nil>
ParamDecls ::= ParamDecls ParamDecl | ParamDecl

```
ParamDecl  ::=  DataType identifier ;
Block  ::= begin Statements end
Statements  ::= Statements ; Statement  |  Statement
Statement  ::= read identifier
             |  set identifier = Expression
             |  write identifier
             |  if Condition then Statements ElseClause
             |  while Condition do Statements endwhile
             |  until Condition do Statements enduntil
             |  call identifier Arglist
             |  <nil>

ElseClause  ::=  else Statements endif  |  endif

ArgList  ::=  ( Arguments )  |  <nil>
Arguments  ::= Arguments , Argument  |  Argument
Condition  ::=  Expression RelOp Expression
Expression  ::=  Expression  AddOp Term  |  Term
Term  ::= Term MultOp Factor  |  Factor
Factor  ::= identifier  |  constant
RelOp  ::=  =  |  !  |  >  |  <
AddOp  ::= +  |  -
MultOp  ::=  *  |  /
```

## APPENDIX 2
## THE LEXICAL ELEMENTS AND BNF GRAMMAR OF EXTENDED JASON

The classes of token in JASON are:

Keyword (each keyword is a distinct token)
identifier
numeric literals
string literals
operator (a single character)

The keywords of JASON are:

| | | | |
|---|---|---|---|
| ARRAY | ENDWHILE | PROGRAM | SUBSTRING |
| BEGIN | FUNCTION | READ | THEN |
| CALL | IF | REAL | TYPES |
| DO | INTEGER | RECORD | UNTIL |
| ELSE | PARAMETERS | RETURN | VARIABLES |
| END | POSITION | SET | WHILE |
| ENDIF | PROCEDURE | STRING | WRITE |
| ENDUNTIL | | | |

Identifer ::= Letter (Letter | Digit)*

Letter is one of

A B C D E F G H I J K L M N O P Q R S T U V W Y Z

where all letters are converted to upper while scanning.

Digit is one of 0 1 2 3 4 5 6 7 8 9

NumericLiterals ::= Digit* {.} Digit*

String Literals ::= " AnyCharacter* "

AnyCharacter is any keyboard character EXCEPT the quotation mark (").

Operator is one of . ; , ( ) = < > ! + - * / & \

```
Program ::= Header DeclSec Block .
Header ::= program identifier ;
DeclSec ::= TypeDeclSec VarDeclSec SubProgramDecls | <nil>
TypeDeclSec ::= types TypeDecls
TypeDecls ::= TypeDecls TypeDecl | TypeDecl
TypeDecl ::= identifier TypeSpecification
TypeSpecification ::= ArraySpecification | RecordSpecification
ArraySpecification ::= array DataType [ numericliteral ] ;
RecordSpecifcation ::= record VarDecl end;

VarDeclSec ::= variables VarDecls
VarDecls ::= VarDecls VarDecl | VarDecl
VarDecl ::=          DataType IdList ;
DataType ::=         real | integer | identifier
IdList ::= IdList , identifier | identifier
SubProgramDecls ::= SubProgramDecls SubProgramDecl
                  | SubProgramDecl
                  | <nil>
SubProgramDecl ::= ProcDecl | FunctionDecl
ProcDecl ::= ProcHeader SubProgramDeclSec Block ;
ProcHeader ::= procedure identifier ;
SubProgramDeclSec ::= ParamDeclSec DeclSec
ParamDeclSec ::= parameters ParamDecls | <nil>
ParamDecls ::= ParamDecls ParamDecl | ParamDecl
ParamDecl ::= DataType identifier ;
Block ::= begin Statements end
```

```
FunctionDecl ::= FunctionHeader  SubProgramDecl FunctionBlock ;
FunctionBlock ::= begin Statements ReturnStatement end
Statements  ::= Statements  ;  Statement    |    Statement
Statement   ::= read  Variable
                |   set  Variable  =  Expression
                |   write  Variable
                |   if  Condition  then  Statements ElseClause
                |   while  Condition  do  Statements endwhile
                |   until  Condition  do  Statements  enduntil
                |   call  identifier  Arglist
                |   <nil>

ElseClause  ::=  else  Statements  endif    |    endif
ReturnStatement ::= ;  return ( identifier )
ArgList  ::=   ( Arguments )    |    <nil>
Arguments   ::= Arguments  ,  Argument    |    Argument
Condition ::= Condition \ CompoundCondition  | CompoundCondition
CompoundCondition ::= CompoundCondition  &  SimpleCondition
                |   SimpleCondition

SimpleCondition   ::=  Expression  RelOp  Expression
Expression  ::=  Expression    AddOp  Term    |    Term
Term   ::= Term MultOp Factor    |    Factor
Factor   ::= Variable    |    numericliteral  | stringliteral
Variable ::= Variable.Variable        | identifier [ Expression ] | identifier
RelOp  ::=  =    |   !   |   >    |   <
AddOp  ::= +    |   -
MultOp  ::=
```