1 Introduction

Your task is to write a semantic analyzer for the language Luca. Your program should be named `luca_sem`. `luca_sem` reads an AST (in the S-expression format specified by assignment 2) from standard input, writes semantic error messages to `standard error`, and writes a symbol table and a decorated abstract syntax tree (both in an S-expression format) on `standard output`:

```plaintext
> cat 3.gus
PROGRAM P;
VAR X : INTEGER;
BEGIN
  X := X + 1;
  WRITE X;
END.

> luca_lex 3.gus | luca_parse | luca_sem

( (12 VariableSy X 2 0 1 4 0)
  )
(PROGRAM P 6
  (DECL 3
    (VARDECL X INTEGER 2
      12)
    (DECLNULL 3)
  )
  (STAT 6
    (ASSIGN 4
      (VARREF X 4
        12)
      (BINARY "+" 4
        (VARREF X 4
          12)
        (INTLIT 1 4 1)
      )
    )
    (STAT 6
      (WRITE 5
        (VARREF X 5
          12)
      )
    )
  )

```

Christian Collberg
October 21, 2002
The output consists of two parts, the symbol table and the decorated abstract syntax tree:

\[
(\text{Program P 6})
\]

The AST that \texttt{luca\_sem} generates is similar to the one that it reads except that new output attributes have been added. In the example above these attributes are boxed. Most output attributes are references to symbols in the symbol table. For example, consider the part of the AST representing \texttt{X+1}:

\[
(\text{Binary } "+") 4
(\text{VarRef X 4})
(\text{Desnull 4})
(\text{IntLit 1 4})
\]

Here, \texttt{1} is a reference to symbol number one in the symbol table, the \texttt{INTEGER} type symbol. \texttt{12} is a reference to the \texttt{X} variable.

Eleven symbols are predeclared by the compiler and not part of the \texttt{luca\_sem} output:

\[
(1 \texttt{TypeSy INTEGER 0 0 BasicType})
(2 \texttt{TypeSy REAL 0 0 BasicType})
(3 \texttt{TypeSy CHAR 0 0 BasicType})
(4 \texttt{TypeSy STRING 0 0 BasicType})
(5 \texttt{TypeSy BOOLEAN 0 0 EnumType})
(6 \texttt{EnumSy TRUE 0 0 5 0 1})
(7 \texttt{EnumSy FALSE 0 0 5 0 0})
\]
The format of each symbol is

\[
\text{(number kind name position level ...)}
\]

where \ldots\ represents information which is specific to each symbol kind. The exact format for each kind of symbol will be described in appendix B.

Semantically incorrect LUCA programs don’t have to produce any output other than error messages.

If the program is syntactically correct luca_parse will print (to standard output) an S-expression representing the symbol table and the AST.

There are four different versions of LUCA described in this document. You can decide to write your semantic analyzer for any one of them. If you decide to work on LUCA-1, for example, the maximum number of points you can receive is 60; if you work on LUCA-3 you can receive 90 points, etc. It’s a good idea to start with LUCA-1 and then incrementally add new features until you are compiling your chosen target language.

- Your tree-walk evaluator should be programmed in an applicative style. I.e., all information should be passed around the tree using synthesized, inherited, and threaded attributes; there should be no global variables. In particular, you should use symbol tables and environments to represent scope information, as shown in lectures. The tree-walk evaluator should make exclusive use of recursion; iteration is not allowed. If you do make use of iteration and global data, points will be deducted.

- This assignment can be coded in the language of your choice as long as it can be compiled and run on lectura.

- Make sure that your Makefile is working properly, and that luca_sem is called exactly as in the example above.

- You should not exit the semantic analyzer after the first error. Rather, you should attempt to generate as informative and as unconfusing messages as possible.

- The error messages should be of the form

\[
> \text{luca_lex file.gus | luca_parse | luca_sem > file.ast}
\]

\text{ERROR (Line 4), Integer type expected.}

and should be written to standard error.

2 LUCA-1 [60 points]

LUCA-1 has constant declarations, integer and real arithmetic, and character, real and integer literals. Identifiers have to be declared before they are used. Identifiers cannot be redeclared. There are four (incompatible) built-in types, INTEGER, REAL, BOOLEAN and CHAR. The identifiers TRUE and FALSE are predeclared in the language. Here is the concrete syntax of LUCA-1:
\( \langle \text{program} \rangle ::= \text{'PROGRAM'} \ (\text{ident}) \ \text{'};\ \langle \text{decl\_list} \rangle \ \text{'.'} \)

\( \langle \text{block} \rangle ::= \text{'BEGIN'} \ \langle \text{block} \rangle \ \text{'END'} \)

\( \langle \text{decl\_list} \rangle ::= \{ \langle \text{declaration} \rangle \ \text{';'} \} \)

\( \langle \text{declaration} \rangle ::= \text{'CONST'} \ (\text{ident}) \ \text{'}; \ (\text{ident}) \ \text{'}=\ \langle \text{expression} \rangle \)

\( \langle \text{expression} \rangle ::= \langle \text{expression} \rangle \ \langle \text{bin\_operator} \rangle \ \langle \text{expression} \rangle \ |
\langle \text{unary\_operator} \rangle \ (\langle \text{expression} \rangle) \ |
\text{'}(\langle \text{expression} \rangle)\text{'} \ |
\langle \text{integer\_literal} \rangle \ | \langle \text{char\_literal} \rangle \ | \langle \text{real\_literal} \rangle \ | \langle \text{designator} \rangle \)

\( \langle \text{designator} \rangle ::= \langle \text{ident} \rangle \)

\( \langle \text{bin\_operator} \rangle ::= \text{'+'} \ | \text{'-'} \ | \text{'*'} \ | \text{'/'} \ | \text{'\%'} \ | \text{'AND'} \ | \text{'OR'} \ | \text{'<'} \ | \text{'<='} \ | \text{'#'} \ | \text{'>='} \ | \text{'>'} \)

\( \langle \text{unary\_operator} \rangle ::= \text{'-'} \ | \text{'NOT'} \ | \text{'TRUNC'} \ | \text{'FLOAT'} \)

The \text{,'#'} symbol means "not equal to". \text{AND} and \text{OR} have lower precedence than the comparison operators, which in turn have lower precedence than the arithmetic operators. All \text{LUCA} languages are case sensitive.

\text{LUCA} does not allow \textit{mixed arithmetic}, i.e. there is no \textit{implicit conversion} of integers to reals in an expression. For example, if \( I \) is an integer and \( R \) is real, then \text{'}R:=I*R\text{'} is illegal. \text{LUCA} instead supports two explicit conversion operators, \text{TRUNC} and \text{FLOAT}. \text{TRUNC} \( R \) returns the integer part of \( R \), and \text{FLOAT} \( I \) returns a real number representation of \( I \). Note also that \text{'}\%\text{'} (remainder) is not defined on real numbers. Here are the type rules for \text{Luca}:

<table>
<thead>
<tr>
<th>Left</th>
<th>Operators</th>
<th>Right</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>'+', '-', '*', '/', '%'</td>
<td>Int</td>
<td>Int</td>
</tr>
<tr>
<td>Real</td>
<td>'+', '-', '*', '/'</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Int</td>
<td>'&lt;', '&lt;=', '&gt;=', '&gt;', '</td>
<td>'</td>
<td>Int</td>
</tr>
<tr>
<td>Real</td>
<td>'&lt;', '&lt;=', '&gt;=', '&gt;', '</td>
<td>'</td>
<td>Real</td>
</tr>
<tr>
<td>Char</td>
<td>'&lt;', '&lt;=', '&gt;=', '&gt;', '</td>
<td>'</td>
<td>Char</td>
</tr>
<tr>
<td>Bool</td>
<td>'AND', 'OR', '!\text{NOT}'</td>
<td>Bool</td>
<td>Bool</td>
</tr>
<tr>
<td></td>
<td>'TRUNC', 'FLOAT'</td>
<td>Real</td>
<td>Int</td>
</tr>
</tbody>
</table>

Additionally, these error conditions should be checked in a constant declaration:

1. Division by 0 is not allowed.
2. Arithmetic operations on characters are not allowed.
3. Variables are not allowed in constant expressions.\(^1\)
4. The type of the constant expression should be the same as the declared type.
5. Identifiers must be declared before they are used.

Here are some examples:

\(^1\)You don’t need to check this for \text{LUCA-1} since it doesn’t have variable declarations, but you do need to check it for \text{LUCA-2}, etc.

4
PROGRAM P1;

VAR X : INTEGER;
CONST C0 : CHAR = "c";       ⇒ OK!
CONST C1 : INTEGER = 5/3;    ⇒ OK!
CONST C2 : INTEGER = 5/(0/3); ⇒ ERROR: "Division by 0!"
CONST C3 : INTEGER = 5/"x";  ⇒ ERROR: "/ requires numeric arguments!"
CONST C4 : INTEGER = 5/C0;   ⇒ ERROR: "/ requires numeric arguments!"
CONST C5 : INTEGER = 5/X;    ⇒ ERROR: "Constant identifier expected!"
CONST C6 : INTEGER = 5/C1+3*C1; ⇒ OK!
CONST C7 : CHAR = 5/C2;      ⇒ ERROR: "Type mismatch!"
CONST C8 : CHAR = 5/C9;      ⇒ ERROR: "Identifier not declared!"
CONST C9 : X = 5;           ⇒ ERROR: "Type identifier expected!"
CONST CA : BOOLEAN = TRUE;  ⇒ OK!
CONST CB : BOOLEAN = TRUE AND CA; ⇒ OK!
END.

3 LUCA-2 [20 Points]

LUCA-2 extends LUCA-1 with variable declarations, assignment statements, READ, WRITE, and WRITELN statements, IF, IF-ELSE, LOOP, EXIT, REPEAT and WHILE statements. Only reals, integers, and characters can be read or written. EXIT statements can only occur within LOOP statements. The expression in an IF, IF-ELSE, REPEAT or WHILE statement must be of boolean type. Here are the extensions to the concrete syntax:

\[\text{block} ::= \text{`BEGIN'} \text{stat_seq}\text{`END'}\]
\[\text{declaration} ::= \text{`VAR'} \text{ident} \text{`:='} \text{ident}\]
\[\text{stat_seq} ::= \{ \text{statement} \text{`;'} \}\]
\[\text{statement} ::= \text{designator} \text{`:='} \text{expression}\]
\[\text{IF} \text{expression} \text{`THEN'} \text{stat_seq}\text{`ENDIF'}\]
\[\text{IF} \text{expression} \text{`THEN'} \text{stat_seq}\text{`ELSE'} \text{stat_seq}\text{`ENDIF'}\]
\[\text{WHILE} \text{expression} \text{`DO'} \text{stat_seq}\text{`ENDDO'}\]
\[\text{REPEAT} \text{stat_seq} \text{UNTIL} \text{expression}\]
\[\text{LOOP} \text{stat_seq}\text{`ENDDO'}\]
\[\text{EXIT}\]
\[\text{WRITE} \text{expression} \mid \text{WRITELN}\]
\[\text{READ} \text{designator}\]

Here is an example of a statically correct and a statically incorrect LUCA-2 program:
4 LUCA-3 [10 Points]

LUCA-3 extends LUCA-2 with declarations of non-nested procedures with parameters and procedure calls with parameters.

A procedure’s formal parameters and local declarations form one scope, which means that it is illegal for a procedure to have a formal parameter and a local variable of the same name. Parameters are passed by value unless the formal parameter has been declared VAR. Only L-valued expressions (such as ‘A’ and ‘A[5]’) can be passed to a VAR formal.

\[\text{declaration} ::= \text{PROCEDURE} \ (\text{ident}) \ (\text{formal_list}) \ (\text{decl_list}) \ (\text{block}) \ (;')\]
\[\text{formal_list} ::= \text{formal_param} \ (';') \ (\text{formal_param}) \]
\[\text{actual_list} ::= \text{expression} \ (';') \ (\text{expression}) \]
\[\text{statement} ::= \text{ident} \ (’[') \ (\text{actual_list}) \ (']')\]
Here is an example:

```pascal
PROGRAM P2;
PROCEDURE Q (
    F : INTEGER; \leftarrow Undeclared identifier 'INTEGER'!
    VAR V : INTEGER;
    F : INTEGER \leftarrow Multiple declaration: 'F'!
);
VAR V : INTEGER; \leftarrow Multiple declaration: 'V'!
BEGIN
    Q(1,V,2);
    Q(1,V,3.5); \leftarrow Type mismatch!
END;
BEGIN
    Q(1,2,3); \leftarrow VAR formal requires variable!
    Q(1,3); \leftarrow Too few parameters!
    Q(1,2,3,4); \leftarrow Too many parameters!
END.
```

5 LUCA-4 [10 Points]

LUCA-4 extends LUCA-3 with

- One-dimensional array type declarations,
- Record type declarations, and
- Array and record element references.

Assignment is defined for scalars only, not for variables of structured type. In other words, the assignment `A:=B` is illegal if A or B are records or arrays. **READ** and **WRITE** are only defined for scalar values (integers, reals, and characters). We make the following extensions to the syntax of LUCA-3:

```
declaration ::= "TYPE" (ident) '=' "ARRAY" (expression) 'OF' (ident)
              "TYPE" (ident) '=' "RECORD" '{' {<field>} '}'
<field> ::= (ident) ':' (ident) ':'
<designator> ::= (ident) {<designator>}
<designator> ::= '[' (expression) ']' <designator> | '.' (ident) <designator>
```

The element count of an array declaration must be a constant integer expression.
Here is an example of a statically incorrect LUCA-4 program:

```
PROGRAM P10E;
CONST V : INTEGER = 5;
TYPE A = ARRAY 100*V OF INTEGER;
TYPE T = ARRAY "c" OF INTEGER; ← Type incompatibility!
TYPE B = ARRAY 20 OF A;
TYPE C = RECORD [a:CHAR; c:INTEGER ];
VAR y : A; VAR x : B; VAR z : C;
BEGIN
  y["c"] := 45; ← Type incompatibility!
  y[5] := 45.5; ← Type incompatibility!
  READ x; ← Scalar expected!
  x := x; ← Illegal assignment!
  z.b := "c"; ← No such field!
  z.c := 3; ← OK!
END.
```

6 LUCA-5 [20 Extra Points]

LUCA-5 extends LUCA-4 with nested blocks. We make the following extensions to the syntax of LUCA-4:

```
<statement> ::= ‘BEGIN’ <decl>seq ‘IN’ <stat>seq ‘END’
```

Procedures cannot occur in the declaration sequence. Variable references follow the most-closely-nested-rule. Here is an example of a statically correct LUCA-5 program:

```
PROGRAM N;
VAR x : INTEGER;
BEGIN
  BEGIN
    VAR x : REAL;
    IN
      x := 4.5;
    BEGIN
      VAR x : CHAR;
      IN
        x := "c";
      END;
      x := 5.5;
    END;
    x := 5;
  END.
```

Note that this extension will require changes to the lexer and the parser.
7 Submission and Assessment

The deadline for this assignment is 23:59, November 4. You should submit the assignment electronically using the Unix command 

```
upload cs453.3 <files> README ...
```

(see man upload).

Your submission must contain a file called README that states which parts of LUCA your semantic analyzer can handle. Also, list the name of your team, the team members, and how much each team member contributed to the assignment.

Your electronic submission must contain a working Makefile, and all the files necessary to build the semantic analyzer. If your program does not compile “out of the box” you will receive zero (0) points. The grader will not try to debug your program or your makefile for you!

Don’t show your code to anyone outside your team, don’t read anyone else’s code, don’t discuss the details of your code with anyone. If you need help with the assignment see the TA or the instructor.
A Abstract Syntax

Below are the nodes in the Luca decorated abstract syntax tree, in the format they are generated by luca-sem. Node-names are in bold, input attributes are in a typewriter font, node references in italics, and output attributes in small caps.

A.1 Declarations

(PROGRAM Ident Pos Decls Stats SYMBOL)
This is the topmost node of any AST. Ident is the name of the program, Pos the line number where Ident occurs. Decls and Stats are the sub-trees for declarations and statements, respectively. SYMBOL is the number of the main program, symbol number 11.

(DECL Pos Left Right)
A list of declarations are linked together using DECL nodes. Left points to the actual declaration, Right is the remaining subtree of declarations.

(DECLNULL Pos)
This node ends a sequence of declarations.

(PROCDECL Ident Pos Formals Decls Stats SYMBOL)
This is the topmost node of a procedure declaration. Formals is a list of FORMALDECL nodes. Decls and Stats are the sub-trees for declarations and statements, respectively. SYMBOL is the symbol number of the procedure.

(FORMALDECL Ident TypeName Mode Pos SYMBOL)
Ident is the name of a formal parameter, TypeName its type. Mode is VAL or VAR. FORMALDECLs are linked together using DECL nodes to form the lists of formal declarations used in PROCDECLs. SYMBOL is the symbol number of the formal.

(VARDECL Ident TypeName Pos SYMBOL)
VARDECLs are linked together using DECL nodes to form lists of variable declarations. Ident is the name of the variable, TypeName its type. SYMBOL is the symbol number of the variable.

(CONSTDECL Ident TypeName Pos Expr SYMBOL)
Ident is the name of the constant, TypeName its type, and Expr the root of an expression tree. SYMBOL is the symbol number of the constant.

(ARRAYDECL Ident ElementType Pos Count SYMBOL)
ARRAYDECL nodes represent an array type declaration. Ident is the name of the type, Count is the root of an expression tree computing the number of elements in the array, and ElementType the type of the elements of the array. SYMBOL is the symbol number of the array type.

(RECORDDECL Ident Pos Fields SYMBOL)
RECORDDECL nodes represent a record type declaration. Ident is the name of the type, and Fields is a list of FIELDDECL nodes, linked together on DECLs. SYMBOL is the symbol number of the record type.

(FIELDDECL Ident TypeName Pos SYMBOL)
FIELDDECL nodes represent a field in a record. They are linked together in a list of DECL nodes, terminated by a DECLNULL. SYMBOL is the symbol number of the field.
A.2 Statements

(STAT Pos Left Right)
A list of statements are linked together using STAT nodes. Left points to the actual statement, Right is the remaining subtree of statements.

(STATNULL Pos)
This node ends a sequence of statements.

(PROCCALL Pos Des Actuals Symbol)
PROCCALL nodes represent a procedure call statement. Des is a designator representing the procedure to be called (always just one VARREF for this version of Luca), and Actuals is a tree of ACTUAL nodes, the actual arguments to the call. Symbol is the symbol number of the procedure.

(ACTUAL Pos Expr Next Symbol)
A tree of ACTUAL nodes are used to represent the argument list to a procedure call. Expr is the root of an expression tree, Next refers to another ACTUAL node or ACTUALNULL. Symbol is the symbol number of the formal parameter corresponding to this actual.

(ACTUALNULL Pos)
This node ends a sequence of expressions.

(WRITE Pos Expr)
Expr is the root of an expression tree whose value is to be written.

(WRITELN Pos)
Ident is the name of the constant, TypeName its type, and Expr the root of an expression tree.

(READ Pos Des)
Read is the root of a designator list representing the variable into which the value is supposed to be read.

(IF1 Pos Expr Then)
Expr is the root of an expression tree. Then is the body of the “then” part of the statement.

(IF2 Pos Expr Then Else)
Expr is the root of an expression tree. Then and Else are the bodies of the “then” and “else” parts of the statement.

(WHILE Pos Expr Stats)
Expr is the root of the expression tree for the loop condition. Stats is the body of the loop.

(REPEAT Pos Expr Stats)
Expr is the root of the expression tree for the loop condition. Stats is the body of the loop.

(LOOP Pos Stats)
Stats is the body of the loop.

(EXIT Pos)
This node represents the EXIT statement which can only occur within the body of a LOOP statement.

(ASSIGN Pos Des Expr)
Expr is the expression tree for the right hand side of the statement, Des represents the designator for the left and side.
A.3 Expressions and Designators

(INTLIT Value Pos SYMBOL)
Value is a decimal integer literal. SYMBOL is the symbol number of the integer type, symbol number 1.

(CHARLIT "Value" Pos SYMBOL)
Value is a one-character string. SYMBOL is the symbol number of the character type, symbol number 3.

(STRINGLIT "Value" Pos)
Value is a string. SYMBOL is the symbol number of the string type, symbol number 4.

(REALLIT Value Pos SYMBOL)
Value is a floating-point number. SYMBOL is the symbol number of the real type, symbol number 2.

(VARREF Ident Pos Next SYMBOL)
VARREFs are the root of a designator list. This list begins with a VARREF, and ends with a DESNULL. Inbetween are INDEX and FIELDREF nodes representing record field and array references. SYMBOL is the symbol number of the variable.

(DESNULL Pos)
The last node of a designator list.

(INDEX Index Next Pos SYMBOL)
INDEX nodes represent an array index reference. Index is the root of an expression tree. Next is the next element of the designator list. SYMBOL is the symbol number of the array type.

(FIELDREF Ident Next Pos SYMBOL)
FIELDREF nodes represent a record field reference. Next is the next element of the designator list. SYMBOL is the symbol number of the field.

(BINARY "Op" Pos Left Right SYMBOL)
Left and Right are the expression trees for the left and right hand side of the operator. "Op" is the operator. SYMBOL is the symbol number of the resulting arithmetic type, i.e. one of integer or real.

(UNARY "Op" Right SYMBOL)
Right is the expression tree for the right hand side of the operator. "Op" is the operator. SYMBOL is the symbol number of the resulting arithmetic type, i.e. one of integer or real.

B Symbol Tables

The format of each symbol is

( number kind name position level ... )

where ... represents information which is specific to each symbol kind. number is a unique number used to identify each symbol. kind describes what type of symbol we’re dealing with, one of VariableSy, ConstSy, EnumSy, FormalSy, FieldSy, ProcedureSy and TypeSy. name is the name of the symbol. level is 0 for global symbols and 1 for symbols declared within procedures.
The information specific to each symbol is given below. Attributes in italic font are standard for all symbols. Attributes in bold font are atoms describing the symbol kind. Attributes in typewriter font are specific to a particular symbol.

\textbf{(number VariableSy name pos level type size offset)}
This entry represents a declared variable. \texttt{type} is the symbol number of the type of the variable. \texttt{size} and \texttt{offset} are the size (in bytes) and the address of the variable. For the purposes of this assignment these can be set to 0.

\textbf{(number ConstSy name pos level type value )}
This entry represents the value of a constant declaration. For integers, floats, and characters the value is simply the obvious textual representation. For booleans it is the atom \texttt{TRUE} or \texttt{FALSE}.

\textbf{(number EnumSy name pos level type size value)}
This is only used for BOOLEAN types since this version of \textsc{Luca} does not allow the declaration of enumeration types.

\textbf{(number FormalSy name pos level type size copy offset formalNo mode)}
This represents a formal parameter of a procedure. \texttt{formalNo} is the number of the formal, where the first parameter has the number 1. \texttt{size} and \texttt{offset} can be set to 0. \texttt{copy} should be set to 9 ($\texttt{NOSYMBOL}$). \texttt{mode} is one of \texttt{VAL} and \texttt{VAR}.

\textbf{(number FieldSy name pos level type size offset parent)}
This represents a field in a record. \texttt{type} is the symbol number of the type of the symbol. \texttt{size} and \texttt{offset} can be set to 0. \texttt{parent} is the symbol number of the record type itself.

\textbf{(number ProcedureSy name pos level formals locals localSize formalSize)}
This represents a procedure declaration. \texttt{formals} is a list (for example, "(12 13 14)") of the symbol numbers of the formal parameters. \texttt{locals} is a list of the symbol numbers of the local variables. \texttt{localSize} and \texttt{formalSize} can be set to 0.

\textbf{(number TypeSy name pos level BasicType size)}
This represents a basic type such as integer or real. \texttt{size} can be set to 0.

\textbf{(number TypeSy name pos level ArrayType count type size)}
This represents an array type. \texttt{count} is the number of elements of the array. \texttt{type} is the symbol number of the element type. \texttt{size} can be set to 0.

\textbf{(number TypeSy name pos level RecordType fields size)}
This represents a record type. \texttt{fields} is the list of symbol numbers of the fields of the record. \texttt{size} can be set to 0.

\textbf{(number TypeSy name pos level EnumType size)}
This represents an enumeration type type. This version of \textsc{Luca} doesn’t have declarations of enumeration types so the only place where this symbol occurs is in the declaration of the standard boolean type. \texttt{size} can be set to 0.

The following symbols are predeclared by the compiler and not written to the output:

(1 TypeSy INTEGER 0 0 BasicType)
(2 TypeSy REAL 0 0 BasicType)
(3 TypeSy CHAR 0 0 BasicType)
(4 TypeSy STRING 0 0 BasicType)
This has the following consequences:

1. The first symbol declared by the program will get symbol number 12.
2. Integer types are always symbol number 1, reals are symbol number 2, etc.
3. The symbols `$NOTYPE` and `$NOSYMBOL` can be used internally to signal error types/symbols.
4. The main program is represented by symbol number 11.

Here is an example of the output from `luca_sem` for a simple program:

```plaintext
> cat t.gus
PROGRAM P;
VAR X : BOOLEAN;
TYPE A = ARRAY 10 OF CHAR;
TYPE R = RECORD [x:INTEGER];
CONST C : INTEGER = 10;
PROCEDURE P (VAR x : REAL; y: R);
BEGIN END;
BEGIN
END.
> luca_lex t.gus | luca_parse | luca_sem

(PROGRAM P 10 ... )

Note that this representation of the symbol table allows forward references. For example, symbol 14 (the record type R) is given before the declaration of the field 15 which it references. This is acceptable, but not required, behavior.