



University of Arizona, Department of Computer Science

CSc 453 — Compilers and Systems Software — Assignment 2

Christian Collberg
September 21, 2001

1 Introduction

Your task is to write a semantic analyzer for the language LUCA. You will be given a lexer/parser that produces an abstract syntax tree from LUCA source files. You should write a tree-walk evaluator that traverses the AST, evaluates attributes, and produces appropriate error messages.

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 marks you can receive is 25; if you work on LUCA-4 you can receive 50 marks, 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, there must be no global symbol table. The tree-walk evaluator should make exclusive use of recursion; iteration is not allowed. If you do make use of iteration and global data, marks will be deducted.

Your program should be written in the ICON programming language. A skeletal evaluator will be provided for you.

2 The LUCA Compiler

The skeletal compiler `gc` can be picked up from `/users/studs/ugrad/stage3/G330B/1998/Ass4`. In the same directory you will also find some example LUCA programs. After you have copied the source to your own directory, you can try the following:

```
> make
  # Lots of output here...
> setenv IPATH /usr/local/lib/icon
> gc -S 0.gus
>   # No output here, since the program is correct...
> gc -S 1.gus
ERROR (Line 4), Type mismatch in assignment statement.
> make TestSyTab
> TestSyTab
  # Lots of output here...
```

The LUCA parser produces an abstract syntax tree, which can be printed out using the `-p` option. Here's a simple LUCA program and the corresponding AST, as produced by `gc -p P8.gus`:

```

PROGRAM:Pos=5: Ident=P8
  DeclsOut={NULL}
  Decls=
    DECL:Pos=3:
      Env=[NULL]
      DeclsIn={NULL}
      DeclsOut={NULL}
      Left=
        VARDECL:Pos=2: Ident=I:TypeIdent=INTEGER
          Env=[NULL]
      Right=
        DECLNULL:Pos=3:
          Stats=
            STAT:Pos=5:
              Env=[NULL]
              Left=
                ASSIGN:Pos=4:
                  Env=[NULL]
                  Left=
                    VARREF:Pos=4: Ident=I
                      Next=
                        DESNULL:Pos=4:
                          Right=
                            BINARY:Pos=4: Op=+
                              Left=
                                VARREF:Pos=4: Ident=I
                                  Next=
                                    DESNULL:Pos=4:
                                      Right=
                                        INTLIT:Pos=4: Value='1'
                                          Type=(0)
                              Right=
                                STATNULL:Pos=5:

```

```

PROGRAM P8;
  VAR I : INTEGER;
BEGIN
  I := I + 1;
END.

```

Words in all-capitals are AST node names. *Pos*, *Ident*, *Value*, etc, are *input* attributes. *Pos* refers to the source code line number corresponding to the AST node¹. *Left=*, *Right=*, etc, are children of AST nodes.

The LUCA compiler *gc* consists of several Icon modules:

<code>gc.icn</code>	The main program.
<code>ast.icn</code>	The declarations of abstract syntax tree nodes. Any new AST attributes that you need to use should be declared here.
<code>semantics.icn</code>	The tree-walk evaluator that performs semantic analysis. This is where you should be doing most of your work.
<code>showtree.icn</code>	A tree-walk procedure that displays the AST before or after semantic analysis. You should update this routine whenever you add a new attribute to <code>ast.icn</code> .
<code>symbols.u*</code>	Definitions of symbol table nodes and routines for manipulating and displaying these.
<code>sytab.u*</code>	Operations on symbol tables. A symbol table is a set of symbols, as defined in <code>symbol</code> .
<code>env.u*</code>	Operations on environments. An environment is a list of symbol tables, as defined in <code>sytab</code> .
<code>error.u*</code>	Operations for printing error messages.
<code>genM2.icn*</code>	A simple tree-walker that generates Modula-2 code from LUCA's AST. Only partially works, but may provide some useful insights into the structure of the AST.

¹Due to technical problems beyond our control, *Pos* will sometimes point to the wrong line number.

3 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:

```

<program> ::= 'PROGRAM' <ident> ';' <decl_list> <block> '.'
<block> ::= 'BEGIN' 'END'
<decl_list> ::= { <declaration> ';' }
<declaration> ::= 'CONST' <ident> ':' <ident> '=' <expression>
<expression> ::= <expression> <bin_operator> <expression> |
    <unary_operator> <expression> |
    '(' <expression> ')' |
    <integer_literal> | <char_literal> | <real_literal> | <designator>
<designator> ::= <ident>
<bin_operator> ::= '+' | '-' | '*' | '/' | '%' | 'AND' | 'OR' | '<' | '<=' | '=' | '#' | '>=' | '>'
<unary_operator> ::= '-' | 'NOT' | 'TRUNC' | 'FLOAT'

```

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

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

Left	Operators	Right	Result
Int	'+', '-', '*', '/', '%'	Int	⇒ Int
Real	'+', '-', '*', '/'	Real	⇒ Real
Int	'<', '<=', '=', '#', '>=', '>'	Int	⇒ Bool
Real	'<', '<=', '=', '#', '>=', '>'	Real	⇒ Bool
Char	'<', '<=', '=', '#', '>=', '>'	Char	⇒ Bool
Bool	'AND', 'OR'	Bool	⇒ Bool
	'NOT'	Bool	⇒ Bool
	'_'	Int	⇒ Int
	'_'	Real	⇒ Real
	'TRUNC'	Real	⇒ Int
	'FLOAT'	Int	⇒ Real

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.²
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:

```
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.
```

4 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:

```
<block> ::= 'BEGIN' <stat_seq> 'END'
<declaration> ::= 'VAR' <ident> ':' <ident>
<stat_seq> ::= { <statement> ';' }
<statement> ::= <designator> ':=' <expression>
               'IF' <expression> 'THEN' <stat_seq> 'ENDIF' |
               'IF' <expression> 'THEN' <stat_seq> 'ELSE' <stat_seq> 'ENDIF' |
               'WHILE' <expression> 'DO' <stat_seq> 'ENDDO' |
               'REPEAT' <stat_seq> 'UNTIL' <expression> |
               'LOOP' <stat_seq> 'ENDLOOP' |
               'EXIT' |
               'WRITE' <expression> | 'WRITELN' |
               'READ' <designator>
```

²You don't need to check this for LUCA-1 since it doesn't have variable declarations, but you do need to check it for LUCA-2, etc.

Here is an example of a statically correct and a statically incorrect LUCA-2 program:

<pre> PROGRAM P1; VAR I : INTEGER; VAR J : INTEGER; VAR C : CHAR; BEGIN I := 1; J := 5 * I + 6; C := "x"; LOOP IF X < 3 THEN WRITE X+3; ELSE READ X; EXIT; ENDIF; ENDLOOP; END. </pre>	<pre> PROGRAM P2; VAR B : BOOLEAN; VAR I : INTEGER; VAR J : INTGER; ← Undeclared identifier 'INTGER'! VAR I : CHAR; ← Multiple declaration: 'I'! BEGIN I := "x"; ← Type incompatibility! J := 5 * I + "v"; ← Type incompatibility! C := "x"; ← Undeclared identifier: 'C'! INTEGER := 5; ← Variable expected: 'INTEGER'! IF X + 3 THEN ← Boolean type expected! WRITE Q; ← Variable expected! ELSE READ B; ← Cannot read Booleans! EXIT; ← EXIT only within LOOP-statements! ENDIF; END. </pre>
---	---

5 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.

```

<declaration> ::= 'PROCEDURE' <ident> '(' [ <formal_list> ] ')' <decl_list> <block> ';'
<formal_list> ::= <formal_param> { ',' <formal_param> }
<formal_param> ::= ['VAR'] <ident> ':' <ident>
<actual_list> ::= <expression> { ',' <expression> }
<statement> ::= <ident> '(' [ <actual_list> ] ')'

```

Here is an example:

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

6 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 $\lceil A := B \rceil$ 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.
```

7 LUCA-5 [10 Extra Points]

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

$\langle \textit{statement} \rangle ::= \textit{'BEGIN' } \langle \textit{decl_seq} \rangle \textit{'IN' } \langle \textit{stat_seq} \rangle \textit{'END' }$

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.
```

8 Submission and Assessment

The deadline for this assignment is midnight, between the 10th and 11th of October. You should submit the assignment electronically using the Unix command `turnin cs453.2 <files> README ..` (see `man turnin`).

Your submission *must* contain a file called `README` that states which parts of LUCA your interpreter 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 interpreter. 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!

<p>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.</p>

A The Abstract Syntax Tree

The file `ast.icn` contains definitions of the node types of the abstract syntax tree. Some of the node-types (which are Icon records) are listed on the next page. Input attributes (attributes generated by the parser) are marked `# Input`, children are marked `# Child`. You may freely add new (synthesized, inherited, or threaded) attributes to these declarations. Note, however, that you must add new fields *after* the predeclared ones, and that the order of the predeclared fields must not be altered.

Below we show the structure of a tree-walk procedure, as used in the semantic analyzer (in `semantics.icn`) and the routine that displays the abstract syntax tree (in `showtree.icn`). The tree-walker could, for example, be called on the empty LUCA program: `S(PROGRAM("P",DECLNULL(1),STATNULL(1)),1)`.

```

procedure S(E)
case type(E) of {
  "PROGRAM"   : {S(E.Decls);
                 S(E.Stats)}
  "DECL"      : {S(E.Left);
                 S(E.Right)}
  "DECLNULL"  : {}
  "VARDECL"   : {}
  "ARRAYDECL" : {}
  "RECORDDECL": {S(E.Fields)}
  "FIELDDECL" : {}
  "PROCDECL"  : {S(E.Formals);
                 S(E.Decls);
                 S(E.Stats)}
  "FORMALDECL": {}
  "STAT"      : {S(E.Left)}
  "STATNULL"  : {}
  "ASSIGN"    : {S(E.Left);
                 S(E.Right)}
  "PROCCALL"  : {S(E.Actuals)}
  "ACTUAL"    : {S(E.Expr);
                 S(E.NextActual)}
  "ACTUALNULL": {}
  "WRITE"     : {S(E.Expr)}
  "WRITELN"   : {}
  "READ"      : {S(E.Des)}
  "WHILEDO"   : {S(E.Expr);
                 S(E.Stats)}
  "IF1"       : {S(E.Expr);
                 S(E.Then)}
  "IF2"       : {S(E.Expr);
                 S(E.Then);
                 S(E.Else)}
  "FOR"       : {S(E.Idx);
                 S(E.From);
                 S(E.To); S(E.By);
                 S(E.Stats)}
  "VARREF"    : {S(E.Next)}
  "DESNUL"    : {}
  "INDEX"     : {S(E.Index);
                 S(E.Next)}
  "FIELDREF"  : {S(E.Next)}
  "CHARLIT"   : {}
  "STRINGLIT" : {}
  "INTLIT"    : {}
  "REALLIT"   : {}
  "BINARY"    : {
    S(E.Left); S(E.Right)
    case E.Op of {
      "+"|"-"|"*"|"/" : {}
      "%" : {}
      "<"|">"|"="|"#"|"<="|">=" : {}
      "AND"|"OR" : {}
    }
  }
  "UNARY"     : {
    S(E.Left)
    case E.Op of {
      "NOT" : {}
      "TRUNC" : {}
      "FLOAT" : {}
      "-" : {}
    }
  }
}
end

```

DECLARATIONS	STATEMENTS	EXPRESSIONS
<pre> record PROGRAM(Ident, # Input Decls, # Child Stats, # Child Pos # Input) record DECL(Left, # Child Right, # Child Pos # Input) record DECLNULL(Pos # Input) record PROCDECL(Ident, # Input Formals, # Child Decls, # Child Stats, # Child Pos # Input) record FORMALDECL(Ident, # Input TypeName, # Input Pos # Input) record VARDECL(Ident, # Input TypeName, # Input Pos # Input) record ARRAYDECL(Ident, # Input Count, # Input ElementTypeName, # Input Pos # Input) record RECORDDECL(Ident, # Input Fields, # Input Pos # Input) record FIELDDECL(Ident, # Input TypeName, # Input Pos # Input) </pre>	<pre> record STAT(Left,Right, # Child Pos # Input) record STATNULL(Pos # Input) record PROCCALL(Ident, # Input Actuals, # Input Pos # Input) record ACTUAL(Expr, # Child NextActual, # Child Pos # Input) record ACTUALNULL(Pos # Input) record WRITE(Expr, # Child Pos # Input) record READ(Des, # Child Pos # Input) record WRITELN(Pos # Input) record IF1(Expr, # Child Then, # Child Pos # Input) record FOR(Idx, # Input From,To,By,Stats# Child Pos # Input) record IF2(Expr,Then,Else # Child Pos # Input) record WHILEDO(Expr,Stats # Child Pos # Input) record ASSIGN(Left,Right # Child Pos # Input) </pre>	<pre> record BINARY(Op, # Input Left, # Child Right, # Child Pos # Input) record UNARY(Op, # Input Left, # Child Pos # Input) record INTLIT(Value, # Input Pos # Input) record CHARLIT(Value, # Input Pos # Input) record STRINGLIT(Value, # Input Pos # Input) record REALLIT(Value, # Input Pos # Input) record VARREF (Ident, # Input Next, # Child Pos # Input) record DESNULL (Pos # Input) record INDEX(Index, # Child Next, # Child Pos # Input) record FIELDREF(Ident, # Input Next, # Child Pos # Input) </pre>

B Symbol Tables and Environments

Three Icon modules are used to create and manipulate symbols, symbol tables, and environments. `symbol` creates “boxes” to store information about variables, procedures, etc. `sytab` manipulates symbol tables, which are sets of symbols. Finally, `env` manipulates environments, which are lists of symbol tables.

To see the actual use of symbols, symbol tables, and environments, have a look at the file `TestSyTab.icn`. You can try it out like this:

```
> make TestSyTab
> TestSyTab
  # Lots of output here...
```

B.1 `symbol.icn`

These are the procedures available in `symbol.icn`, to create symbols, and store/extract data on symbols:

Procedure	Description
<code>symbol_MakeArrayType(Name)</code>	Create a new array type symbol.
<code>symbol_MakeRecordType(Name)</code>	Create a new record type symbol.
<code>symbol_MakeVar(Name)</code>	Create a new variable symbol.
<code>symbol_MakeConst(Name)</code>	Create a new constant symbol.
<code>symbol_MakeFormal(Name)</code>	Create a new formal parameter symbol.
<code>symbol_MakeField(Name)</code>	Create a new record field symbol.
<code>symbol_MakeProc(Name)</code>	Create a new procedure symbol.
<code>symbol_GetName(Id)</code>	Get the name of symbol <code>Id</code> .
<code>symbol_GetNumber(Id)</code>	Get the number of symbol <code>Id</code> .
<code>symbol_GetLevel(Id)</code>	Get the declaration level of symbol <code>Id</code> .
<code>symbol_SetLevel(Id, Level)</code>	Set the declaration level of symbol <code>Id</code> .
<code>symbol_GetKind(Id)</code>	Return the strings "VariableSy", "FieldSy", "FormalSy", "ProcedureSy", "TypeSy", "ConstSy", depending on what kind of symbol we're dealing with.
<code>symbol_GetTypeKind(Id)</code>	Return the strings "BasicType", "RecordType", "ArrayType", depending on what kind of symbol we're dealing with.
<code>symbol_GetType(Id)</code>	Get the type of a variable, field, constant, or formal.
<code>symbol_SetType(Id, Type)</code>	Set the type of a variable, field, constant, or formal.

Procedure	Description
<code>symbol_GetArrayCount(Id)</code>	Return the number of elements in the array Id.
<code>symbol_SetArrayCount(Id, Count)</code>	Set the number of elements in the array Id.
<code>symbol_GetArrayElementType(Id)</code>	Return the element type(a symbol) of array Id.
<code>symbol_SetArrayElementType(Id, ET)</code>	Set the element type ET(a symbol) of array Id.
<code>symbol_GetFields(Id)</code>	Get the fields (a sytab) of record type Id.
<code>symbol_SetFields(Id, Fields)</code>	Set the fields (a sytab) of record type Id.
<code>symbol_GetConstantValue(Id)</code>	Get the value of a constant.
<code>symbol_SetConstantValue(Id, Value)</code>	Set the value of a constant.
<code>symbol_GetProcLocals(Id)</code>	Get the local variables(a sytab) of procedure Id.
<code>symbol_SetProcLocals(Id, Locals)</code>	Set the local variables(a sytab) of procedure Id.
<code>symbol_GetProcFormals(Id)</code>	Get the formal parameters(a sytab) of procedure Id.
<code>symbol_SetProcFormals(Id, Formals)</code>	Set the formal parameters(a sytab) of procedure Id.
<code>symbol_GetFormalParam(Formals, N)</code>	Get formal parameter number N of procedure Id.
<code>symbol_GetFormalNumber(Id)</code>	Get formal number of formal parameter Id.
<code>symbol_SetFormalNumber(Id, Number)</code>	Set formal number of formal parameter Id.
<code>symbol_GetFormalMode(Id)</code>	Get formal mode(string "VAR" or "VAL") of formal parameter Id.
<code>symbol_SetFormalMode(Id, Mode)</code>	Set formal mode (string "VAR" or "VAL") of formal parameter Id.
<code>symbol_2String(S, Kind)</code>	Return a string representation of symbol S. Kind=0 ⇒ short output. Kind=3 ⇒ medium output. Kind=1 ⇒ long output.
<code>symbol_Show(S)</code>	Print symbol S.
<code>symbol_ShowAll(Kind)</code>	Print all symbols. Kind=0 ⇒ short output. Kind=3 ⇒ medium output. Kind=1 ⇒ long output.

Note that symbols are represented as Icon *records*. This means that they have to be compared using the `===` operator:

```
if E.Symbol === standard_NoSy then
    error_SemId(E.Pos, "Variable identifier not declared", E.Ident)
```

B.2 sytab.icn

These are the procedures available in `sytab.icn`, to create and lookup symbols in symbol tables:

Procedure	Description
<code>sytab_Create()</code>	Create an empty symbol table.
<code>sytab_CreateSingleton(S)</code>	Create a symbol table with one element, <code>S</code> .
<code>sytab_Copy(S)</code>	Return a fresh copy of symbol table <code>S</code> .
<code>sytab_Merge(S1, S2)</code>	Return a new symbol table with the symbols from symbol tables <code>S1</code> and <code>S2</code> .
<code>sytab_Insert(SyTab, S)</code>	Return a new symbol table, consisting of the symbols from <code>SyTab</code> and symbol <code>S</code> .
<code>sytab_DeleteByName(SyTab, Name)</code>	Return a new symbol table with the elements from <code>SyTab</code> , but with any symbol with name <code>Name</code> deleted.
<code>sytab_LocateByName(SyTab, Name)</code>	Return the symbol in symbol table <code>SyTab</code> whose name is <code>Name</code> , or <code>standard_NoSy</code> if no such symbol exists.
<code>sytab_Ids(SyTab)</code>	Generate the elements of <code>SyTab</code> .
<code>sytab_Count(SyTab)</code>	Count the number of symbols in <code>SyTab</code> .
<code>sytab_2String(SyTab)</code>	Return the symbol numbers of the symbols in <code>SyTab</code> as a string.
<code>sytab_2StringOfNames(SyTab)</code>	Return a string of the names of the symbols in <code>SyTab</code> .
<code>sytab_2StringOfSymbols(SyTab)</code>	Return <code>SyTab</code> as a string.
<code>sytab_Show(SyTab)</code>	Print <code>SyTab</code> .

B.3 env.icn

These are the procedures available in `env.icn`, to create and lookup symbols in environments:

Procedure	Description
<code>env_Create()</code>	Create an empty environment.
<code>env_Cons(SyTab, E)</code>	Return a new environment, consisting of symbol table <code>SyTab</code> , followed by the symbol tables of environment <code>E</code> .
<code>env_Snoc(E, S)</code>	Return a new environment, consisting of the symbol tables of environment <code>E</code> , followed by symbol table <code>SyTab</code> .
<code>env_Append(E1, E2)</code>	Return a new environment, consisting of the symbol tables of environment <code>E1</code> , followed by the symbol tables of environment <code>E2</code> .
<code>env_LocateByName(E, N)</code>	Return the <i>first</i> symbol in environment <code>E</code> whose name is <code>N</code> , or <code>standard_NoSy</code> if no such symbol exists.
<code>env_2StringOfNames(E)</code>	Return a string of the names of the symbols in environment <code>E</code> .
<code>env_2StringOfSymbols(E)</code>	Return environment <code>E</code> as a string.
<code>env_Show(E)</code>	Print environment <code>E</code> .

B.4 standard.icn

These are the procedures and global variables available in `standard.icn`:

Procedure	Description
<code>standard_env()</code>	Return a standard environment, consisting of the built-in types (<code>INTEGER</code> , <code>REAL</code> , <code>CHAR</code> , <code>STRING</code> , <code>BOOLEAN</code>), and the predeclared constants <code>TRUE</code> and <code>FALSE</code> .
<code>standard_IntType</code>	The symbol representing <code>INTEGER</code> .
<code>standard_CharType</code>	The symbol representing <code>CHAR</code> .
<code>standard_RealType</code>	The symbol representing <code>REAL</code> .
<code>standard_StringType</code>	The symbol representing <code>STRING</code> .
<code>standard_BoolType</code>	The symbol representing <code>BOOLEAN</code> .
<code>standard_NoType</code>	The symbol representing an illegal or erroneous type.
<code>standard_NoSy</code>	The symbol representing a missing or erroneous symbol.

C Utility routines

These are the procedures available in `error.icn`:

Procedure	Description
<code>error_Internal(Proc, Msg)</code>	Indicates that an error has occurred in procedure <code>Proc</code> of the compiler.
<code>error_SemanticCount()</code>	The number of semantic errors that have been generated so far.
<code>error_Sem(Pos, Msg)</code>	Generate a semantic error message on line <code>Pos</code> .
<code>error_SemId(Pos, Msg, Id)</code>	Generate a semantic error message (involving symbol <code>Id</code>) on line <code>Pos</code> .

Use these routines to generate error messages in the semantic analyser. For example:

```
if E.Symbol === standard_NoSy then
    error_SemId(E.Pos, "Variable identifier not declared", E.Ident)

error_Internal("semantics_VarDecl", "Unexpected problem.")

if E.Left.Type ~=== E.Right.Type then
    error_Sem(E.Pos, "Type mismatch in assignment statement.")
```