1 Introduction

Your task is to write a code generator for the language LUCA. You will be given a front-end that produces a control flow graph of

a) expression trees,
b) quadruples, or
c) triples

from LUCA source files. You should write a code generator that produces either

a) SPARC-V9 assembly code that runs on 1ectura, or
b) MIPS assembly code that runs on the SPIM simulator

from one of these intermediate forms. Honors students must generate code for the SPARC.

There are five different versions of LUCA described in this document. You can decide to write your code generator for any one of them. If you decide to work on LUCA-1, for example, the maximum number of points you can receive is 50; if you work on LUCA-4 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.

This assignment will be implemented in Icon.

You may use whatever code generation algorithm you want. I will only grade on correctness, not the speed of the generated code. However, if you feel you’ve done something very clever I might be convinced to give extra credit. Be sure to indicate what you have implemented in your README file.

2 The LUCA Compiler

Version 6.10 of the gc compiler can be picked up from the usual places. After you have copied the source to your own directory, you can try the following:

> make
    # Lots of output here...
> gc -h
Usage: gc [options] [infile] [outfile]
   -p    print the AST after syntactic analysis.
   -P    print the AST after semantic analysis.
   -S    Exit after semantic analysis.
   -t    print the intermediate tree code.
   -k[level] print the stack code at detail level <level>.
   -q    print the intermediate quaduple code.
   -c[kind] print the control flow graph of the code.
       kind is one of 'tree', 'stack', 'quad', 'triple'.
   -s[level] prints symbol table at detail level <level>.
   -a[arch] Set architecture. Possible values are
gvm, alpha, sparc-v9, sparc, mips.
   -m    generate machine code.
   -x[level] execute program with trace level <level>.
   -M2   Generate Modula-2 code.
   -H|h   print this message.
   -v    print version.
if   infile is missing, input is from standard input.
if outfile is missing, output is to standard output.
> gc -amips -m 0.gus
   # LineNo=7  Code generated from treecode_Version
   # LineNo=3  Code generated from treecode_VarDecl
   # LineNo=5  Code generated from treecode_Store
   # LineNo=5  Code generated from treecode_AddressOf
   # LineNo=5  Code generated from treecode_LoadLit
   # LineNo=6  Code generated from treecode_Write
   # LineNo=6  Code generated from treecode_Load
   # LineNo=6  Code generated from treecode_AddressOf

As you see, at this point the only code generated is some comments. You should update the compiler to emit the appropriate assembly code. Note that some command line options have changed.

You can also ask the front-end to print out the intermediate code:

> gc -ctree 0.gus
Global Declarations:
====================
Version: Pos=4 Major=6 Minor=10

Control Flow Graphs:
=====================
PROCEDURE (1:$MAIN:Pos=0:PROC) EntryBlock=1 ExitBlock=3
BLOCK #1: Pre={}; Suc={2}
BLOCK #2: Pre={1}; Suc={3}
Write: Pos=3 Type=INTEGER
      Expr=LoadLit: Pos=3 Type=INTEGER Value='5'
BLOCK #3: Pre={2}; Suc={}

2
> gc -cquad 0.gus

Global Declarations:
=======================
Version Major=6 Minor=10

Control Flow Graphs:
=====================
PROCEDURE (1:$MAIN:Pos=0:PROC) EntryBlock=1 ExitBlock=3
BLOCK #1: Pre={}; Suc={2}

BLOCK #2: Pre={1}; Suc={3}
LoadLit/INTEGER (24:00:Pos=3:TEMP:Type=INTEGER) (25:%0:Pos=3:CONST:Type=INTEGER:Value=5)
Write/INTEGER (24:00:Pos=3:TEMP:Type=INTEGER)

BLOCK #3: Pre={2}; Suc={}

> gc -c triple 0.gus

Global Declarations:
=====================
(1) Version Major=6 Minor=10

Control Flow Graphs:
=====================
PROCEDURE (1:$MAIN:Pos=0:PROC) EntryBlock=1 ExitBlock=3
BLOCK #1: Pre={}; Suc={2}

BLOCK #2: Pre={1}; Suc={3}
(1) LoadLit/INTEGER 5
(2) Write/INTEGER (1)

BLOCK #3: Pre={2}; Suc={}

We have a few new modules in addition to the modules in the previous assignment:

- `cfg` Definition of control flow graph data structures.
- `buildcfg` Builds control flow graphs.
- `showcfg` Displays control flow graphs.
- `treetcode` Definition of expression tree nodes.
- `showtreetcode` Displays expression tree nodes.
- `treetcode` Definition of expression tree nodes.
- `showquadcode` Displays quadruple instructions.
- `quadcode` Definition of quadruple instructions.
- `showtriplecode` Displays triple instructions.
- `triplecode` Definition of triple instructions.
- `machcode` Generates assembly code.
- `mcode` Maintains a list of generated instructions.
Most of your work should take place in machcode.icn.

3 LUCA-1 [60 Points]

LUCA-1 has constant and variable declarations, integer arithmetic, assignment statements, READ, WRITE, and WRITELN statements. Only integers and characters can be read, strings can also be written. EXIT statements can only occur within LOOP statements. Identifiers have to be declared before they are used. Identifiers cannot be redeclared. There are three (incompatible) built-in types, INTEGER, 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’ ⟨stat_seq⟩ ‘END’
⟨decl_list⟩ ::= { ⟨declaration⟩ ‘;’ }
⟨declaration⟩ ::= ‘CONST’ ⟨ident⟩ ‘;’ ⟨ident⟩ ‘=’ ⟨expression⟩
‘VAR’ ⟨ident⟩ ‘;’ ⟨ident⟩
⟨expression⟩ ::= ⟨expression⟩ ⟨bin_operator⟩ ⟨expression⟩ |
⟨unary_operator⟩ ⟨expression⟩ |
‘(’ ⟨expression⟩ ‘)’ | ⟨integer_litera⟩ | ⟨char_litera⟩ | ⟨real_litera⟩ | ⟨designator⟩
⟨designator⟩ ::= ⟨ident⟩
⟨bin_operator⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’
⟨unary_operator⟩ ::= ‘-’
⟨stat_seq⟩ ::= { ⟨statement⟩ ‘;’ }
⟨statement⟩ ::= ⟨designator⟩ ‘:=’ ⟨expression⟩
‘WRITE’ ⟨expression⟩ | ‘WRITELN’ |
‘READ’ ⟨designator⟩

On the MIPS, integers are 32-bit quantities, character and booleans are 8-bits. On the SPARC, integers are 64-bit quantities, character and booleans are 8-bits.

On the SPARC, READ and WRITE should be implemented by calling functions in the standard C IO-library stdio. To see how to do this, inspect the output of the standard C compiler. For example, compile

```
#include <stdio.h>
main () {printf("%i\n",56);}
```

with ‘cc -S test.c’. The resulting assembly code contains the calling sequence for printf.

4 LUCA-2 [10 Points]

LUCA-2 adds IF, IF-ELSE, LOOP, EXIT, REPEAT, FOR, and WHILE statements. 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:
\(\langle \text{bin\_operator} \rangle \ ::= \ '\text{AND}' \mid '\text{OR}' \mid '<' \mid '<=\mid '=' \mid '>=' \mid '>'\)

\(\langle \text{unary\_operator} \rangle \ ::= \ 'NOT'\)

\(\langle \text{statement} \rangle \ ::= \ 'IF' \ (\text{expression}) \ 'THEN' \ (\text{stat\_seq}) \ 'ENDIF'\)

\(\ 'IF' \ (\text{expression}) \ 'THEN' \ (\text{stat\_seq}) \ 'ELSE' \ (\text{stat\_seq}) \ 'ENDIF'\)

\(\ 'WHILE' \ (\text{expression}) \ 'DO' \ (\text{stat\_seq}) \ 'ENDDO'\)

\(\ 'REPEAT' \ (\text{stat\_seq}) \ 'UNTIL' \ (\text{expression})\)

\(\ 'LOOP' \ (\text{stat\_seq}) \ 'ENDLOOP'\)

\(\ 'EXIT'\)

\(\ 'FOR' \ (\text{ident}) \ ':==' \ (\text{expression}) \ 'TO' \ (\text{expression}) \ ['BY' \ (\text{const\_expr})] \ 'DO' \ (\text{stat\_seq}) \ 'ENDFOR'\)

The **FOR**-loop **BY**-expression must be a compile-time constant expression. A **LUCA** **FOR**-loop

\[
\begin{align*}
\text{FOR } i & := e1 \ T0 \ e2 \ BY \ e3 \ D0 \\
& \ S \\
& \ \text{ENDFOR}
\end{align*}
\]

should be compiled into code that’s equivalent to

\[
\begin{align*}
i & := e1; \\
T1 & := e2; \\
T2 & := e3; \\
\text{IF } T2 & \geq 0 \ \text{THEN} \\
& \ \text{WHILE } i \ \leq T1 \ \text{DO} \\
& \ \ \ \ S; \\
& \ \ \ \ i := i + T2; \\
& \ \ \ \ \text{ENDDO}; \\
\text{ELSE} & \ \text{WHILE } i \ \geq T1 \ \text{DO} \\
& \ \ \ \ S; \\
& \ \ \ \ i := i + T2; \\
& \ \ \ \ \text{ENDDO}; \\
\text{ENDIF}
\end{align*}
\]

## 5 **LUCA-3 [10 Points]**

**LUCA-3** extends **LUCA-2** with real variable declarations and real arithmetic.

**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.

We add two operators **TRUNC** and **FLOAT**:

\[\langle \text{unary\_operator} \rangle \ ::= \ '\text{TRUNC}' \mid '\text{FLOAT}'\]

On the MIPS, reals are 32-bit quantities. On the SPARC, reals are 64-bit quantities.
6 LUCA-4 [10 Points]

LUCA-4 extends LUCA-3 with one-dimensional arrays and record types.

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).

The element count of an array declaration must be a constant integer expression. Arrays are indexed from 0; that is, an array declared as 'ARRAY 100 OF INTEGER' has the index range [0..99]. It is a checked run-time error to go outside these index bounds.

Here are the extensions to the concrete syntax:

\[
\langle declaration \rangle ::= \text{'TYPE'} \ (\text{ident}) \ ('=' \text{'ARRAY'} \ (\text{expression}) \ 'OF' \ (\text{ident})
\text{'TYPE'} \ (\text{ident}) \ ('=' \text{'RECORD'} \ [ \{ \text{field} \} \ ])
\langle field \rangle ::= \ (\text{ident}) \ ':' \ (\text{ident}) \ ';'
\langle designator \rangle ::= \ (\text{ident}) \ \{ \ (\text{designator}) \ \}\n\langle designator \rangle' ::= \ [ \ (\text{expression}) \ ]' \ (\text{designator}) \ | \ ' \ (\text{ident}) \ (\text{designator})
\]

7 LUCA-5 [10 Points]

LUCA-5 extends LUCA-4 with non-nested procedures.

There is no limit to the number of arguments a procedure may take. Value parameters (including structured types such as arrays and records!) should be passed by value, VAR parameters by reference.

Here are the extensions to the concrete syntax:

\[
\langle declaration \rangle ::= \text{'PROCEDURE'} \ (\text{ident}) \ ('\ (' [(\text{formal}\_\text{list})] \ ')') \ (\text{decl}\_\text{list}) \ (\text{block}) \ ';'
\langle formal\_\text{list} \rangle ::= \ (\text{formal}\_\text{param}) \ \{ \ ' \ (\text{formal}\_\text{param}) \ \}\n\langle formal\_\text{param} \rangle ::= \ [ \text{'VAR'} \ (\text{ident}) \ '':' \ (\text{ident})
\langle actual\_\text{list} \rangle ::= \ (\text{expression}) \ \{ \ ' \ (\text{expression}) \ \}\n\langle statement \rangle ::= \ (\text{id}\text{nt}) \ ('\ (' [\ (\text{actual}\_\text{list}) \ ] \ ')')
\langle field \rangle ::= \ (\text{id}\text{nt}) \ ':' \ (\text{id}\text{nt}) \ ';'
\]

8 Extension 1 [20 Points]

Implement at least the following local optimizations:

- algebraic simplification,
- jump-to-jump elimination,
• replace power-of-two multiplication and division by left and right shifts,
• constant folding.

9 Submission and Assessment

The deadline for this assignment is midnight, between the 2nd and 3rd of December. You should submit the assignment electronically using the Unix command `turnin cs453.5 <files> README ...`.

Your submission must contain a README-file that states which parts of LUCA your code generator 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 code generator. 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!

This assignment is worth 100 points.

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 The tree intermediate code

The front-end generates control flow graphs where each node is a sequence of expression trees. The nodes (defined in treecode.icn) in these trees are defined below:

<table>
<thead>
<tr>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>record treecode_Version (Major,Minor,Pos) The version of the intermediate code language.</td>
</tr>
<tr>
<td>record treecode_VarDecl (Symbol,Pos) treecode_VarDecl declares a global or local variable. Symbol is the symbol table entry for the variable, from which we can retrieve information such as size (symbol.GetSize(S)), level of declaration (symbol.GetLevel(S)), type (symbol.GetType(S)), and address (symbol.GetOffset(S)).</td>
</tr>
<tr>
<td>record treecode_FormalDecl (Symbol,Pos) Declares the formal parameter of a procedure. Symbol is the symbol table entry, from which we can retrieve information such as mode (symbol.GetFormalMode(S)), size (symbol.GetSize(S)), level of declaration (symbol.GetLevel(S)), type (symbol.GetType(S)), and offset (symbol.GetOffset(S)). Note that the offset returned by symbol.GetOffset(S) is a suggestion only; you may find that a different activation record layout suits your back-end better.</td>
</tr>
<tr>
<td>record treecode_TypeDecl (Symbol,Pos) Declares a record or array type. Symbol is the symbol table entry.</td>
</tr>
</tbody>
</table>
**Loads and Stores**

<table>
<thead>
<tr>
<th>Record Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treecode.Store (Type, Left, Right, Pos)</td>
<td>Left is an expression tree computing an address. Right is an expression tree computing a value (it's type is given by Type) to be stored at that address.</td>
</tr>
<tr>
<td>treecode.Load (Type, Des, Pos)</td>
<td>Des is an expression tree computing an address. Load should load the value (whose type is given by Type) stored at that address.</td>
</tr>
</tbody>
</table>

**Expressions**

<table>
<thead>
<tr>
<th>Record Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treecode_BinExpr (Op, Type, Left, Right, Pos)</td>
<td>A node in an expression tree that computes Left Op Right. Op is a string.</td>
</tr>
<tr>
<td>treecodeUnaryExpr (Op, Type, Left, Pos)</td>
<td>A node in an expression tree that computes Op Left. Op is a string. Type is a symbol table reference.</td>
</tr>
<tr>
<td>treecode_LoadLit (Type, Value, Pos)</td>
<td>Load the literal value Value. In case of strings, the address should be loaded, not the value.</td>
</tr>
</tbody>
</table>

**Designators**

<table>
<thead>
<tr>
<th>Record Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treecode_AddressOf (Symbol, Type, Pos)</td>
<td>Load the address of Symbol (which could be a global variable, local variable, or formal parameter). Type is the type of the symbol.</td>
</tr>
<tr>
<td>treecode_IndexOf (Type, Base, Index, Pos)</td>
<td>Compute the address of an array element, i.e. Base + symbol.GetSize(symbol.GetArrayType(Type)) * Index. Base is an expression tree computing the base address of the array. Index is an expression tree computing the index value. Type is a symbol table reference to the array from which we can retrieve information such as symbol.GetArrayCount(S) and symbol.GetArrayType(S). It's a checked, fatal, run-time error for Index to be &lt;0 or &gt;symbol.GetArrayCount(S)-1.</td>
</tr>
<tr>
<td>treecode_FieldOf (Type, Field, Base, Pos)</td>
<td>Compute the address of a record field, i.e. Base + symbol.GetOffset(Field). Base is an expression tree computing the base address of the record. Field is a symbol table entry for the field from which we can retrieve information such as offset (symbol.GetOffset(S)). Type is a symbol table reference to the record type.</td>
</tr>
</tbody>
</table>

**Control**

<table>
<thead>
<tr>
<th>Record Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treecode_Branch (Op, Type, Left, Right, Label, Pos)</td>
<td>Equivalent to if Left Op Right then goto Label. Left and Right are expression trees, Op is a string, and Label is the number of the basic block to which we should jump.</td>
</tr>
<tr>
<td>treecode_Goto (Label, Pos)</td>
<td>Jump to basic block number Label.</td>
</tr>
</tbody>
</table>
### Input and Output

**record treecode_Write (Type,Expr,Pos)**  
Write the value of Expr (an expression tree) to the standard output. If Expr is a (constant) string, Expr will compute the string's *address*, not its value.

**record treecode_Read (Type,Des,Pos)**  
Read a value into the address held by Des, an expression tree. The type of the data to be read is given by Type, a symbol table reference.

**record WriteLn (Pos)**  
Write a newline character to the standard output.

### Procedure call

**record treecode_ProcCall (Symbol,Actuals,Pos)**  
Call procedure Symbol. Symbol is a symbol table entry from which we can retrieve information such as formal parameters (symbol.GetProcForms(S)), local data size (symbol.GetLocalSize(S)), level of declaration (symbol.GetLevel(S)), and size of formal parameters (symbol.GetFormalSize(S)). Actual is an expression tree.

**record treecode_Actual (Type,Formal,Expr,Next,Pos)**  
Treecode_Actual nodes are linked together on Next to make a list of actual parameters. Expr (an expression tree) computes the value/address of the actual. Formal is a reference to the symbol table entry for the corresponding formal parameter, from which we can retrieve information such as size (treecode.GetSize(S)), offset within the activation record (treecode.GetOffset(S)), number (treecode.GetFormalNumber(S)), and mode (treecode.GetFormalMode(S)).

**record treecode_Null (Pos)**  
Treecode_Null terminates a sequence of treecode_Actual nodes.

---

### B The control flow graph

The intermediate code instructions are organized in a series of control flow graphs, one per procedure. The structure of the graph is defined in `cfg.icn`:

### Procedure call

**record cfg(Declarations, Procs)**  
Declarations is a list of global declaration instructions. Procs is a list of `cfg_PROCs`, one per procedure.

**record cfg_PROC(ProcSy, Declarations, EntryBlock, ExitBlock, Blocks)**  
ProcSy is a symbol table entry for the procedure. For the main program block, ProcSy=standard_mainProc. Declarations is a list of declaration instructions for the procedure. EntryBlock is the number of the entry block. ExitBlock is the number of the exit block. Blocks is a table of `cfg_BBs`, one entry per basic block.

**record cfg_BB(Number, Instrs, Successors, Predecessors)**  
Number is the number of the basic block. Instrs is a list of instructions (expression trees). Successors is a set of numbers of successor blocks. Predecessors is a set of numbers of predecessor blocks.
C The quadruple and triple code

The quadruple code is defined in `quadcode.icn`, the triple code in `triplecode.icn`. There is a more-or-less one-to-one correspondence between the treecode and the quadruple and the triple instruction-sets. For example, `quadcode.icn`, `treecode.icn`, and `triplecode.icn` all have a BinExpr instruction:

```plaintext
record treecode_BinExpr (
  Op, # STRING
  Type, # Symbol table reference
  Left, # Child
  Right, # Child
  Pos
)
record quadcode_BinExpr (
  Result, # Symbol table reference
  Op, # STRING
  Type, # Symbol table reference
  Left, # Symbol table reference
  Right, # Symbol table reference
  Pos
)
record triplecode_BinExpr (
  Op, # STRING
  Type, # Symbol table reference
  Left, # Triple number
  Right, # Triple number
  Pos
)
```

The difference is in how the arguments to the binary operation are encoded. In the treecode they are pointers to sub-trees. In the quadruple code they are references to temporary variables in the symbol table. In the triple code they are integers referencing other triples.

For example, the program

```plaintext
PROGRAM P;
BEGIN
  WRITE 44+55;
END.
```

will look like this in the three different intermediate forms:

```
> gc -t 1.gus
Version: Pos=4 Major=6 Minor=10
ProcBegin: Pos=4 Symbol=(1:$MAIN:Pos=0:PROC)
Write: Pos=3 Type=INTEGER
  Expr=BinExpr: Pos=3 Op='+' Type=INTEGER
    Left=LoadLit: Pos=3 Type=INTEGER Value='44'
    Right=LoadLit: Pos=3 Type=INTEGER Value='55'
```
ProcEnd: Pos=4 Symbol=(1:$MAIN:Pos=0:PROC)

> gc -q 1.gus
Version      Major=6 Minor=10
ProcBegin    (1:$MAIN:Pos=0:PROC)
LoadLit/INTEGER (25:01:Pos=3:TEMP:Type=INTEGER) (26:%0:Pos=3:CONST:Type=INTEGER:Value=44)
LoadLit/INTEGER (27:02:Pos=3:TEMP:Type=INTEGER) (28:%1:Pos=3:CONST:Type=INTEGER:Value=55)
BinExpr/+/INTEGER (24:00:Pos=3:TEMP:Type=INTEGER) (25:01:Pos=3:TEMP:Type=INTEGER) (26:02:Pos=3:TEMP:Type=INTEGER)
Write/INTEGER (24:00:Pos=3:TEMP:Type=INTEGER)
ProcEnd       (1:$MAIN:Pos=0:PROC)

> gc -e 1.gus
(1) Version      Major=6 Minor=10
(2) ProcBegin    $MAIN
(3) LoadLit/INTEGER 44
(4) LoadLit/INTEGER 55
(5) BinExpr/+/INTEGER (3) (4)
(6) Write/INTEGER  (5)
(7) ProcEnd       $MAIN

When you have chosen your intermediate form you should make the appropriate changes to gc.icn:

```
#########################################################
# Uncomment one of the segments below depending on what type of #
# intermediate code you want to generate machine code from. #
#########################################################

if PrintMachineCode >= 0 then {
  # CFGtree := buildcfg_Build(TreeCode,treecode_classification())
  # MachineCode := machcode_gen(CFGtree)

  # TripleCode := gentriple_generate(TreeCode)
  # CFGtriple := buildcfg_Build(TripleCode,tiplecode_classification())
  # MachineCode := machcode_gen(CFGtriple)

  # QuadCode := genquad_generate(TreeCode)
  # CFGquad := buildcfg_Build(QuadCode,quadcode_classification())
  # MachineCode := machcode_gen(CFGquad)

  mcode_Write(MachineCode, OutFile)
```

11
## D symbol.icn

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

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol_MAKEArrayType(Name)</td>
<td>Create a new array type symbol.</td>
</tr>
<tr>
<td>symbol_MAKERecordType(Name)</td>
<td>Create a new record type symbol.</td>
</tr>
<tr>
<td>symbol_MAKEVar(Name)</td>
<td>Create a new variable symbol.</td>
</tr>
<tr>
<td>symbol_MAKEConst(Name)</td>
<td>Create a new constant symbol.</td>
</tr>
<tr>
<td>symbol_MAKEFormal(Name)</td>
<td>Create a new formal parameter symbol.</td>
</tr>
<tr>
<td>symbol_MAKEField(Name)</td>
<td>Create a new record field symbol.</td>
</tr>
<tr>
<td>symbol_MAKEProc(Name)</td>
<td>Create a new procedure symbol.</td>
</tr>
<tr>
<td>symbol_SETName(Id)</td>
<td>Get the name of symbol Id.</td>
</tr>
<tr>
<td>symbol_GETNumber(Id)</td>
<td>Get the number of symbol Id.</td>
</tr>
<tr>
<td>symbol_GETLevel(Id)</td>
<td>Get the declaration level of symbol Id.</td>
</tr>
<tr>
<td>symbol_SETLevel(Id, Level)</td>
<td>Set the declaration level of symbol Id.</td>
</tr>
<tr>
<td>symbol_SETKind(Id)</td>
<td>Return the strings &quot;VariableSy&quot;, &quot;FieldSy&quot;, &quot;FormalSy&quot;, &quot;ProcedureSy&quot;, &quot;TypeSy&quot;, &quot;ConstSy&quot;, depending on what kind of symbol we're dealing with.</td>
</tr>
<tr>
<td>symbol_SETTypeKind(Id)</td>
<td>Return the strings &quot;BasicType&quot;, &quot;RecordType&quot;, &quot;ArrayType&quot;, depending on what kind of symbol we're dealing with.</td>
</tr>
<tr>
<td>symbol_SETSize(Id, Size)</td>
<td>Set size of symbol Id, a type, formal, field, variable, or constant.</td>
</tr>
<tr>
<td>symbol_GETSize(Id)</td>
<td>Set size of symbol Id, a type, formal, field, variable, or constant symbol.</td>
</tr>
<tr>
<td>symbol_GETType(Id)</td>
<td>Get the type of a variable, field, constant, or formal.</td>
</tr>
<tr>
<td>symbol_SETType(Id, Type)</td>
<td>Set the type of a variable, field, constant, or formal.</td>
</tr>
<tr>
<td>symbol_GETOffset(Id)</td>
<td>Get the offset/address of a variable, field, or formal.</td>
</tr>
<tr>
<td>symbol_SETOffset(Id, Offset)</td>
<td>Set the offset/address of a variable, field, or formal.</td>
</tr>
<tr>
<td>symbol_GETArrayCount(Id)</td>
<td>Return the number of elements in the array Id.</td>
</tr>
<tr>
<td>symbol_SETArrayCount(Id, Count)</td>
<td>Set the number of elements in the array Id.</td>
</tr>
<tr>
<td>symbol_GETArrayElementType(Id)</td>
<td>Return the element type(a symbol) of array Id.</td>
</tr>
<tr>
<td>symbol_SETArrayElementType(Id, ET)</td>
<td>Set the element type ET(a symbol) of array Id.</td>
</tr>
<tr>
<td>symbol_SETFields(Id)</td>
<td>Get the fields(a sytab) of record type Id.</td>
</tr>
<tr>
<td>symbol_SETFields(Id, Fields)</td>
<td>Set the fields(a sytab) of record type Id.</td>
</tr>
<tr>
<td>Procedure</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>symbol_GetConstantValue(Id)</td>
<td>Get the value of a constant.</td>
</tr>
<tr>
<td>symbol_SetConstantValue(Id, Value)</td>
<td>Set the value of a constant.</td>
</tr>
<tr>
<td>symbol_GetProcLocals(Id)</td>
<td>Get the local variables(a sytab) of procedure Id.</td>
</tr>
<tr>
<td>symbol_GetProcLocals(Id, Locals)</td>
<td>Set the local variables(a sytab) of procedure Id.</td>
</tr>
<tr>
<td>symbol_SetProcFormals(Id)</td>
<td>Get the formal parameters(a sytab) of procedure Id.</td>
</tr>
<tr>
<td>symbol_SetProcFormals(Id, Formals)</td>
<td>Set the formal parameters(a sytab) of procedure Id.</td>
</tr>
<tr>
<td>symbol_GetFormalParam(Formals, N)</td>
<td>Get formal parameter number N of procedure Id.</td>
</tr>
<tr>
<td>symbol_GetLocalSize(Id)</td>
<td>Get the size of local variables of procedure Id.</td>
</tr>
<tr>
<td>symbol_SetLocalSize(Id, Size)</td>
<td>Set the size of local variables of procedure Id.</td>
</tr>
<tr>
<td>symbol_GetFormalSize(Id)</td>
<td>Get the size of formal parameters of procedure Id.</td>
</tr>
<tr>
<td>symbol_SetFormalSize(Id, Size)</td>
<td>Set the size of formal parameters of procedure Id.</td>
</tr>
<tr>
<td>symbol_GetFormalNumber(Id)</td>
<td>Get formal number of formal parameter Id.</td>
</tr>
<tr>
<td>symbol_SetFormalNumber(Id, Number)</td>
<td>Set formal number of formal parameter Id.</td>
</tr>
<tr>
<td>symbol_GetFormalMode(Id)</td>
<td>Get formal mode(string &quot;VAR&quot; or &quot;VAL&quot;) of formal parameter Id.</td>
</tr>
<tr>
<td>symbol_SetFormalMode(Id, Mode)</td>
<td>Set formal mode(string &quot;VAR&quot; or &quot;VAL&quot;) of formal parameter Id.</td>
</tr>
<tr>
<td>symbol_2String(S, Kind)</td>
<td>Convert symbol S to a string. Kind=0 ⇒ short output. Kind=3 ⇒ medium output.</td>
</tr>
<tr>
<td>symbol_Show(S)</td>
<td>Print symbol S.</td>
</tr>
<tr>
<td>symbol_ShowAll(Kind)</td>
<td>Print all symbols. Kind=0 ⇒ short output. Kind=3 ⇒ medium output. Kind=1 ⇒ long output.</td>
</tr>
</tbody>
</table>