University of Arizona, Department of Computer Science

CSc 553 — Assignment 2 — Due noon, Tue March 8 — 15%

Christian Collberg
January 24, 2011

1 Introduction

This assignment consists of four tasks:

1. Extend the design of the virtual machine code from assignment 1 to handle the object-oriented features of Luca.
2. Extend the back-end to generate the new virtual machine code.
3. Extend the interpreter to execute the new virtual machine code.
4. Extend the interpreter with a garbage collector.

Note the following:

1. The object-oriented features of the Luca language are defined in Section A.
2. The new tree intermediate language operators are defined in Section B.2.
3. You should work in teams of two students.
4. Download the compiler front-end and test-cases from `http://www.cs.arizona.edu/~collberg/Teaching/553/2011/Assignments/lucadist2.zip`
5. The interpreter should be implemented using *indirect threaded code*.
6. You should write your interpreter in C or C++ using `gcc`'s *labels-as-values*.
7. The compiler should be named `lucac` and the interpreter should be named `lucax`. They should be called like this:

```
> lucac x.luc -o x.vm
> lucax x.vm
```

8. The interpreter *must* support a flag `-h size` which sets the size of the heap in kilobytes. When this flag is set the heap size is static, i.e. *you must not grow or shrink the heap*. This example runs `test.luc` with a one megabyte heap.

```
lucax -h 1024 test.vm
```

   If you use a copying collection algorithm then the size is the *total* size of the heap, i.e.

$$\texttt{size(to_space)} + \texttt{size(from_space)}.$$

9. You *may* but you don't *have to* grow the heap when you run out of heap memory. However, if you implement a static heap you must exit gracefully with an error message should you run out memory.

10. You may implement any garbage collection algorithm you want: reference-counting, copying, mark-and-sweep, generational, conservative, etc. Implementing `NEW` as a call to `malloc` is also OK, although you will get substantially reduced marks since you will never collect any garbage!

11. You should test the interpreter on `lectura`.

12. To facilitate debugging and evaluation, you should implement LUCA's **SPECIAL** functions, found in Section A.3.2.

2 Write your own test case! [10 bonus points]

For an additional 10 points, you should submit an object-oriented LUCA program that does something vaguely useful. A synthetic program (one that only exercises the OO features without performing any useful computations) is not acceptable.

3 Submission and Assessment

The deadline for this assignment is noon, Tue March 8. It is worth 15% of your final grade.

You should submit the assignment to `d2l.arizona.edu`.

You should submit *one* file, ass2.zip, containing all the files necessary to build the compiler and interpreter. Modify the `makefile` so that the grader can build the project by simply typing `make`, and nothing else.

Don't show your code to anyone, 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 instructor or the TA.

A The LUCA Language

A.1 LUCA Lexical Rules

- LUCA line comments start with a `--`-sign and extend to the end of the line.
- LUCA structured comments start with a `(*` and must end with `*)`. They are not allowed to be nested.
- LUCA is case-sensitive.
- Strings start and end with a `"`-character and cannot contain a `"`-character. They cannot extend past the end of a line.
- Character literals start and end with a `'`-character and must contain exactly one character (not a `'`).
- Identifiers consist of letters and digits, and must start with a letter.
- Integer literals consist of a sequence of digits.

- Real literals have the syntax

$$((\texttt{digit}*\texttt{.digit}+)|(\texttt{digit}+\texttt{.digit}*))(\texttt{E}(+|-)?\texttt{digit}+)?$$

  Examples of valid floating-point numbers:

```
0.5    .5    5.    5.0    5.E-6    100.587E99
```

- Control characters other than tabs and newlines are not allowed in LUCA source files.

A.2 LUCA Syntax

program ::= '**PROGRAM**' ident ';' decl_list block '.'
block ::= '**BEGIN**' stat_seq '**END**'
decl_list ::= { declaration ';' }
declaration ::= '**CONST**' ident ':' ident '=' expression |
    '**VAR**' ident ':' ident |
    '**TYPE**' ident '=' '**ARRAY**' expression '**OF**' ident |
    '**TYPE**' ident '=' '**RECORD**' '[' [field_list] ']' |
    '**PROCEDURE**' ident '(' [formal_list] ')' ';' decl_list block
formal_list ::= formal_param { ';' formal_param }
field_list ::= field { ';' field }
formal_param ::= ['**VAR**'] ident ':' ident
field ::= ident ':' ident
stat_seq ::= { statement ';' }
statement ::= designator ':=' expression |
    designator '(' [actual_list] ')' |
    '**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
actual_list ::= expression { ',' expression }
expression ::= expression bin_operator expression |
    unary_operator expression |
    '(' expression ')' |
    integer_literal | char_literal | real_literal | string_literal | designator
designator ::= ident { designator' }
designator' ::= '[' expression ']' | '.' ident
bin_operator ::= '+' | '−' | '*' | '/' | '%' | '**AND**' | '**OR**' | '<' | '<=' | '=' |'#' | '>=' |'>'
unary_operator ::= '−' | '**NOT**' | '**TRUNC**' | '**FLOAT**'

This grammar is highly ambiguous. Here are the relevant operator precedence rules:

precedence	operator	arity	associativity
low	`+, -`	binary	left associative
	`*, /, %`	binary	left associative
	`AND,OR`	binary	left associative
	`<, <=, #, >, >=, =`	binary	left associative
high	`NOT, TRUNC, FLOAT, -`$_{\texttt{unary}}$	unary	right associative

A.2.1 Object-Oriented Features

The object-oriented version of LUCA adds the following syntax:

declaration ::= '**TYPE**' ident '=' '**CLASS**' ['**EXTENDS**' ident] '[' field_list ']' '[' method_list ']'
method_list ::= method_decl [';' method_list]
method_decl ::= '**METHOD**' ident '(' [formal_list] ')' ';' decl_list block
statement ::= designator '(' [actual_list] ')' |
'**SPECIAL**' string_literal
designator ::= designator '@' ident |
designator '‘' ident
unary_operator ::= '**NEW**'
bin_operator ::= '**ISA**'

A.3 Static Semantics

- The LUCA language is case sensitive.
- Luca has four (incompatible) built-in types: `INTEGER`, `CHAR`, `BOOLEAN` and `REAL`. All basic types are 32 bits wide.
- 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.
- 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.
- These 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

- The identifiers `TRUE` and `FALSE` are predeclared in the language.
- The **FOR**-loop **BY**-expression must be a compile-time constant.
- 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** is only defined for scalar values (integers, reals, and characters).
- **WRITE** is defined for scalar values (integers, reals, and characters). and literal strings.
- 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 esxpressions (such as '`A`' and '`A[5]`') can be passed to a **VAR** formal.
- Procedures cannot be nested.
- Identifiers have to be declared before they are used.

A.3.1 Object-Oriented Features

The object-oriented version of LUCA adds the semantics:

- `TYPE T=CLASS [][]` declares a new class `T` with no fields or methods. It extends the root class `OBJECT`.
- `TYPE U=CLASS EXTENDS T [][]` declares a new class `U` with all of `T`'s (its superclass) fields and methods. `U` may declare additional fields and methods that extend (in the case of fields) or overrides or extends (in the case of methods) the fields and methods of `T`.
- A method `P()` in class `T` will override another method `P` in a superclass of `T`. Both methods must have the same signature.
- `NEW T` allocates a new object of class type `T`.
- `d'T` narrows (*casts* in Java terminology) a designator `d` to class type `T` if this is possible (i.e. if `d` is of type `T` or of one of `T`'s subtypes) or aborts with an error-message otherwise.
- `e ISA T` evaluates to `TRUE` if `e` is of type `T` or of one of `T`'s subtypes.
- `x@f` is a designator referencing a field or method `f` in object `x`.
- `SELF` is a reference to the current object. It is only available in methods.
- Assume that `t` is an object of type `T` and `u` is an object of type `U`. The assignment `t := u` is legal if `U` is a subtype of `T`. If that is not the case, a compile-time or run-time error should be generated. Implicit assignments (actual parameters assigned to formal parameters, for example) are also checked in a similar way.
- `NIL` is a new constant which is compatible with all object types.
- Variables of `CLASS` type are essentially pointers. They are one-word quantities.
- Variables of `CLASS` type can be compared for equality (`=`) and inequality (`#`) only.
- Class types may only be declared at the global level. Methods may not be nested.

A.3.2 Special Functions

Luca supports the `SPECIAL` statement which takes a string argument and is used to call special built-in functions. You may add whatever special functions you want, but at least the following should be implemented:

`SPECIAL "GC"`	Force a garbage collection. Only effective in systems that implement a stop-and-copy garbage collector.
`SPECIAL "DUMPHEAP"`	Print the objects allocated on the heap to standard error. It is up to the implementation if it wants to print only reachable objects or both live and dead objects.
`SPECIAL "PRINTMEM";`	Print the amount of free heap memory, the amount of allocated heap memory (`HEAPSIZE = FREE + USED`), and the number of garbage collections that the program has performed since program startup. Print to standard error.

`SPECIAL "STARTTIME";`	The implementation should keep two global timing variables initialized at startup:
	`double cpu = 0.0;`
	`double wall = 0.0;`
	To start timing, execute the equivalent of:
	`double currentCPUTime = GetCPUTime();`
	`double currentWallTime = GetWallTime();`
	The timing functions are defined below.

`SPECIAL "STOPTIME";`	Execute:
	`cpu += GetCPUTime() - currentCPUTime;`
	`time += GetWallTime() - currentWallTime;`

`SPECIAL "PRINTTIME";`	Print the timing variables to standard error.

To get the current cpu and wall time use these functions:

```
#include<sys/resource.h>
#include<sys/time.h>
double GetCPUTime () {
   struct timeval Time;
   double cpu;
   struct rusage Resources;

   getrusage(RUSAGE_SELF, &Resources);
   Time = Resources.ru_utime;
   cpu = (double)Time.tv_sec +
         (double)Time.tv_usec/1000000.0;

   return cpu;
}
```

```
double GetWallTime () {
   struct timeval Time;
   double wall;

   gettimeofday(&Time, NULL);
   wall = (double)Time.tv_sec +
         (double)Time.tv_usec/1000000.0;

   return wall;
}
```

A.4 Context Conditions

Below are the error conditions the compiler needs to check, organized by AST node.

DECL:

An identifier can only be declared once in each scope. A procedure's formal parameters and local declarations form one scope. If `ID` is declared more than once, the compiler should issue this error message:

```
<SEMANTIC_ERROR pos="..." message="Multiple declaration" argument="ID"/>
```

VARDECL, FIELDDECL, FORMALDECL:

1. The type name must be declared:

   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared" argument="TypeName"/>
   ```

2. And, if the type name is declared, it has to be declared to be a type:

   ```
   <SEMANTIC_ERROR pos="..." message="Type identifier expected" argument="TypeName"/>
   ```

CONSTDECL:

1. The type name must be declared:

   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared" argument="TypeName"/>
   ```

2. And, if the type name is declared, it has to be declared to be a type:

   ```
   <SEMANTIC_ERROR pos="..." message="Type identifier expected" argument="TypeName"/>
   ```

3. And, if it's declared a type, it has to be declared a *scalar* type (integer, character, real, boolean):

   ```
   <SEMANTIC_ERROR pos="..." message="Scalar type expected"/>
   ```

4. If the declared type is OK, you need to check that the expression is of the same type:

   ```
   <SEMANTIC_ERROR pos="..." message="Wrong expression type"/>
   ```

5. Regardless of the type checks above, the expression has to be constant-valued:

   ```
   <SEMANTIC_ERROR pos="..." message="Constant expression expected"/>
   ```

ARRAYDECL:

1. The type name must be declared:

   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared" argument="TypeName"/>
   ```

2. And, if the type name is declared, it has to be declared to be a type:

   ```
   <SEMANTIC_ERROR pos="..." message="Type identifier expected" argument="TypeName"/>
   ```

3. The array size must be of type integer:

   ```
   <SEMANTIC_ERROR pos="..." message="Integer expression expected"/>
   ```

4. Regardless, of its type, the array size must be a constant expression:

   ```
   <SEMANTIC_ERROR pos="..." message="Constant expression expected"/>
   ```

ASSIGN:

1. The left hand and the right hand side must be of scalar (integer, real, char, boolean) type.

   ```
   <SEMANTIC_ERROR pos="..." message="Scalar type expected"/>
   ```

2. The left hand and the right hand side must be the same type.

   ```
   <SEMANTIC_ERROR pos="..." message="Type missmatch in assignment statement"/>
   ```

3. The left hand side must be a L-value, i.e. something you can assign to.

   ```
   <SEMANTIC_ERROR pos="..." message="Can't assign to a constant"/>
   ```

PROCCALL:

1. The designator must be a single declared identifier:

   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared"/>
   ```

2. If the identifier is declared, it must be declared to be a procedure:

   ```
   <SEMANTIC_ERROR pos="..." message="Procedure identifier expected"/>
   ```

WRITE:

The expression must evaluate to an integer, real, character, or string.

```
<SEMANTIC_ERROR pos="..." message="INTEGER, REAL, CHAR, STRING type expected"/>
```

READ:

1. The designator must evaluate to an integer, real, or character:

   ```
   <SEMANTIC_ERROR pos="5" message="INTEGER, REAL, CHAR type expected"/>
   ```

2. The designator has to be an L-value (i.e. something you can assign to):

   ```
   <SEMANTIC_ERROR pos="..." message="Can't read to a constant"/>
   ```

WHILE, REPEAT, IF1, IF2:

The expression must be a boolean:

```
<SEMANTIC_ERROR pos="..." message="Boolean type expected"/>
```

EXIT:

`EXIT` must not occur outside of a loop:

```
<SEMANTIC_ERROR pos="..." message="EXIT only within LOOP"/>
```

ACTUAL:

1. There have to be the same number of actual and formal parameters:

   ```
   <SEMANTIC_ERROR pos="..." message="Too many actual parameters"/>
   <SEMANTIC_ERROR pos="..." message="Too few actual parameters"/>
   ```

2. Regardless, the actual parameter has to be assignable to the corresponding formal parameter.

```
<SEMANTIC_ERROR pos="..." message="Actual/formal parameter type missmatch"/>
```

3. Regardless, if a formal parameter is declared to be a **VAR** parameter, then the corresponding actual has to be an L-value (cannot be a constant):

   ```
   <SEMANTIC_ERROR pos="..." message="VAR formal parameter requires variable actual"/>
   ```

VARREF:

1. The identifier has to be declared:

   ```
   <SEMANTIC_ERROR pos="..." message="Identifier not declared" argument="ID"/>
   ```

2. The identifier must be a formal parameter, a variable (global or local), a constant identifier, or a procedure:

   ```
   <SEMANTIC_ERROR pos="..." message="Variable expected"/>
   ```

INDEX:

1. The index expression must evaluate to an integer type:

   ```
   <SEMANTIC_ERROR pos="..." message="Integer type expected"/>
   ```

2. The designator must be of array type:

   ```
   <SEMANTIC_ERROR pos="..." message="Array variable expected"/>
   ```

FIELDREF:

1. The designator must be of record type:

   ```
   <SEMANTIC_ERROR pos="..." message="Record variable expected"/>
   ```

2. If the designator is or record type, then the field must be declared in the record:

   ```
   <SEMANTIC_ERROR pos="..." message="Field identifier not declared" argument="ID"/>
   ```

BINARY, UNARY:

The table in the previous section gives the semantic rules for expressions. For constant expressions, division by zero isn't allowed.

1. In arithmetic expressions, when a real or integer is expected, issue:

   ```
   <SEMANTIC_ERROR pos="..." message="Numeric type expected"/>
   ```

2. For $a\%b$, if a and b aren't integer types, issue

   ```
   <SEMANTIC_ERROR pos="..." message="Integer type expected"/>
   ```

3. For `AND`, `OR`, `NOT`, if the arguments aren't boolean types, issue

   ```
   <SEMANTIC_ERROR pos="..." message="Boolean type expected"/>
   ```

4. When the arguments to comparison operators (`#, <, >, <=, >=, =`) aren't integers, reals, booleans, or chars, issue

   ```
   <SEMANTIC_ERROR pos="..." message="Scalar or reference type expected"/>
   ```

   (Reference type refers to another version of Luca that also has pointer types.)

5. For `TRUNC` and `FLOAT`, respectively, when the arguments are of the wrong type, issue

```
<SEMANTIC_ERROR pos="..." message="Real type expected"/>
<SEMANTIC_ERROR pos="..." message="Integer type expected"/>
```

6. Otherwise, if the left and right hand sides are of different types, issue:

   ```
   <SEMANTIC_ERROR pos="..." message="Type missmatch"/>
   ```

A.5 Checked Runtime Errors

- 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. You should generate the following error message:

  ```
  <RUNTIME_ERROR pos="3" message="Array index out of range"/>
  ```

  Note that the source code line number is part of the error message.

- Division by zero should generate this error message:

  ```
  <RUNTIME_ERROR pos="3" message="Division by zero"/>
  ```

B The Luca Translator

B.1 The Luca Virtual Machine

- The Luca virtual machine is a word-addressed machine. Words are 32 bits wide. The size of all basic types (integers, reals, booleans, and chars) is one word.
- The Luca virtual machine is a stack machine. Conceptually, there is just one stack and it is used both for parameter passing and for expression evaluation. An implementation may – for efficiency or convenience – use several stacks. For example, in a three stack implementation one stack can be used to store activation records, one can be used for integer arithmetic and one can be used for real arithmetic.
- Execution begins at the (parameterless) procedure named `$MAIN`.
- Large value parameters are passed by reference. It is the responsibility of the called procedure to make a local copy of the parameter. For example, if procedure `P` passes an array `A` by value to procedure `Q`, `P` actually pushes the *address* of `A` on the stack. Before execution continues at the body of `Q`, a local copy of `A` is stored in `Q`'s activation record. The body of `Q` accesses this copy. A special instruction `Copy` is inserted by the front end to deal with this case.
- When a `ProcCall` instruction is encountered the arguments to the procedure are on the stack, with the first argument on the top. In other words, arguments are pushed in the *reverse* order.
- Variables whose names start with "$" are temporaries inserted by the front end. They are currently only used in the implementation of `FOR`-loops.

B.2 The tree intermediate code

The front-end generates an intermediate representation that is a sequence of expression trees. Below are listed the tree-code node types the frontend generates.

Declarations

`Version(Major,Minor,Pos)` The version of the intermediate code language.

`VarDecl(Symbol,Pos)` `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(`S.GetSize()`), level of declaration (`S.GetLevel()`), type(`S.GetType()`), and address(`S.GetOffset()`).

`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(`S.GetFormalMode()`), size(`S.GetSize()`), level of declaration (`S.GetLevel()`), type(`S.GetType()`), and offset(`S.GetOffset()`). Note that the offset returned by `S.GetOffset()` is a suggestion only; you may find that a different activation record layout suits your back-end better.

`TypeDecl(Symbol,Pos)` Declares a record or array type. `Symbol` is the symbol table entry.

Loads and Stores

`Store(Type,Left,Right,Pos)` `Left` is an expresion 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.

`Load(Type,Des,Pos)` `Des` is an expression tree computing an address. `Load` should load the value (whose type is given by `Type`) stored at that address.

Expressions

`BinExpr(Op,Type,Left,Right,Pos)` A node in an expression tree that computes `Left Op Right`. `Op` is defined in `lexer.Token`.

`UnaryExpr(Op,Type,Left,Pos)` A node in an expression tree that computes `Op Left`. `Op` is defined in `lexer.Token`. `Type` is a symbol table reference.

`LoadLit(Type,Value,Pos)` Load the literal value `Value`. In case of *strings*, the *address* should be loaded, not the value.

Designators

`AddressOf(Symbol,Type,Pos)` Load the address of `Symbol`(which could be a global variable, local variable, or formal parameter). `Type` is the type of the symbol.

`IndexOf(Type,Base,Index,Pos)` Compute the *address* of an array element, i.e. `Base` + `S.GetSize(S.GetArrayElementType(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 `S.GetArrayCount()` and `S.GetArrayElementType()`. It's a *checked, fatal, run-time error* for `Index` to be `<0` or `>S.GetArrayCount()-1`.

`FieldOf(Type,Field,Base,Pos)` Compute the *address* of a record field, i.e. `Base` + `S.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(`S.GetOffset()`). `Type` is a symbol table reference to the record type.

Control

`Branch(Op,Type,Left,Right,Label,Pos)` Equivalent to `if Left Op Right then goto Label`. `Left` and `Right` are expression trees, `Op` is defined in `lexer.Token`, and `Label` is the number of the label to which we should jump.

`Goto(Label,Pos)` Jump to `Label`.

`Label(Label,Pos)` Declare a `Label`.

Input and Output

`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.

`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.

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

`Special(Value,Pos)` Call the special funtion `Value`.

Procedure call

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

`Actual(Type,Formal,Expr,Next,Pos)` `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(`GetSize()`, offset within the activation record (`GetOffset()`), number(`GetFormalNumber()`), and mode(`GetFormalMode()`).

`Null(Pos)` `Null` terminates a sequence of `Actual` nodes.

Object-Oriented Features

`MethodCall(Symbol, Self, Base, Actuals, Pos)` Call a method `Symbol`. `SELF` corresponds to the first formal argument to the method. Base is the expression that evaluates to `SELF`. `Actuals` is the list of actual expression arguments.

`NarrowVal(Expr, Template, Pos)` `Template` evaluates to a pointer to the class template for a class T. If `Expr` evaluates to a pointer to an object whose runtime type is of type T or a subtype of T then `NarrowVal` evaluates to `Expr`. Otherwise, it generates a fatal type cast error.

`NarrowVar(Des, Template, Pos)` Same as `NarrowVal` except `Des` is a the *address* of the object pointer, rather than the pointer itself.

`LoadNil(Pos)` Evaluates to `Nil` ("0" on most machines).

`ObjectFieldOf(Class,Field,Base,Pos)` Compute the *address* of a class field. `Base` evaluates to a pointer to the object.

`Branch(ISA,Type,Left,Right,Label,Pos)` `Right` is a template pointer of class type `Type`. `Left` is an object pointer. If `Left`'s runtime type is `Right` or a subtype of `Right` then goto `label`, otherwise fall through to the next instruction.

`UnaryExpr(NEW,Type,Left,Pos)` A node in an expression tree allocates a new object of class-type `Type`. `Left` is a pointer to the class' template.

B.3 Symbols

These are the procedures available in `sym/*.java`, to extract data on symbols:

Procedure	Description
`S.GetName()`	Get the name of symbol `S`.
`S.GetNumber()`	Get the unique identifying number of symbol `S`.
`S.GetLevel()`	Get the declaration level of symbol `S`.
`S.SetLevel(Level)`	Set the declaration level of symbol `S`.
`S.SetSize(Size)`	Set size of symbol `S`, a type, formal, field, variable, or constant.
`S.GetSize()`	Set size of symbol `S`, a type, formal, field, variable, or constant symbol.
`S.GetType()`	Get the type of a variable, field, constant, or formal.
`S.SetType(Type)`	Set the type of a variable, field, constant, or formal.
`S.GetOffset()`	Get the offset/address of a variable, field, or formal.
`S.SetOffset(Offset)`	Set the offset/address of a variable, field, or formal.
`S.GetArrayCount()`	Return the number of elements in the array `S`.
`S.SetArrayCount(Count)`	Set the number of elements in the array `S`.
`S.GetArrayElementType()`	Return the element type(a symbol) of array `S`.
`S.SetArrayElementType(ET)`	Set the element type `ET` (a symbol) of array `S`.
`S.GetFields()`	Get the fields (a `sym.SyTab`) of record type `S`.
`S.SetFields(Fields)`	Set the fields (a `sym.SyTab`) of record type `S`.
`S.GetConstantValue()`	Get the value of a constant.
`S.SetConstantValue(Value)`	Set the value of a constant.
`S.GetProcLocals()`	Get the local variables(a `sym.SyTab`) of procedure `S`.
`S.SetProcLocals(Locals)`	Set the local variables(a `sym.SyTab`) of procedure `S`.
`S.GetProcFormals()`	Get the formal parameters(a `sym.SyTab`) of procedure `S`.
`S.SetProcFormals(Formals)`	Set the formal parameters(a `sym.SyTab`) of procedure `S`.
`S.GetFormalParam(Formals, N)`	Get formal parameter number `N` of procedure `S`.
`S.GetLocalSize()`	Get the size of local variables of procedure `S`.
`S.SetLocalSize(Size)`	Set the size of local variables of procedure `S`.
`S.GetFormalSize()`	Get the size of formal parameters of procedure `S`.
`S.SetFormalSize(Size)`	Set the size of formal parameters of procedure `S`.
`S.GetFormalNumber()`	Get formal number of formal parameter `S`.
`S.SetFormalNumber(Number)`	Set formal number of formal parameter `S`.
`S.GetFormalMode()`	Get formal mode(string `"VAR"` or `"VAL"`) of formal parameter `S`.
`S.SetFormalMode(Mode)`	Set formal mode(string `"VAR"` or `"VAL"`) of formal parameter `S`.
`S.GetEnumValue()`	Get the value (an int) of enumeration identifier `S`.
`S.SetEnumValue(Value)`	Set the value (an int) of enumeration identifier `S`.

B.3.1 Object-Oriented Features

`S.GetParent()`	Get the class to which the method `S` belongs.
`S.SetParent(Parent)`	Set the parent class of `ClassType` object.
`S.GetMethodProc()`	Get the `ProcedureSy` symbol corresponding to the method `S`.
`S.SetMethodProc(MethodProc)`	Set the `ProcedureSy` symbol corresponding to the method `S`.
`S.GetInstanceVariables()`	Get the instance variables of the class `S`.
`S.GetClassMethods()`	Get the methods of the class `S`.
`S.GetSuperType()`	Get the parent class of this `ClassType` object.
`S.SetSuperType(SuperType)`	Set the parent class of this `ClassType` object.
`S.SetMethodSize(Size)`	Get the size of the methods in the class `S`, i.e. the number of methods (including inherited methods) times the size of a pointer.
`S.GetMethodSize()`	Set the size of the methods in the class `S`, i.e. the number of methods (including inherited methods) times the size of a pointer.
`S.SetFieldSize(Size)`	Set the size of the fields in the class `S`, including inherited fields.
`S.GetFieldSize(Type)`	Get the size of the fields in the class `S`, including inherited fields.