Compiler Contruction for the 21st Century: Report on the Compiler Technology Project funded by Microsoft

Neil Johnson and Alan Mycroft
University of Cambridge
Computer Laboratory

December 4, 2004

Introduction

Cambridge University has a long-established record of excellence in the field of Computer Science. The Computer Laboratory was originally founded in May 1937, and began teaching the world's first taught course in Computer Science in 1953. Today, the lab enjoys a world-class record of both undergraduate and postgraduate teaching, and leading research in the fields of programming languages, security, theorem proving, natural language processing, systems research, theory and semantics, and processor hardware design.

The second-year *Compiler Construction* course at the Computer Laboratory currently uses the Java JVM intermediate byte-code language for illustrating code generation techniques for a stack machine. A newer virtual machine—Microsoft's .Net—is growing in popularity, being used for modern Windows applications, from web servers to word processors.

This report describes the three main examples presented in the *Compiler Construction* course—a recursive-descent four function calculator, a machine-generated four function calculator for comparison, and a compiler for a simple C-like language that generates .Net assembly code. This report also includes a list of resources that students may find useful in their studies.

The final year *Optimising Compilers* course focuses on more advanced topics, namely optimising for faster execution, and touching on related issues such as decompilation. The compiler presented in this report is an ideal base from which student projects can explore the concepts presented in the advanced course.

Chapter 1

What is .Net?

The .Net platform, announced by Microsoft in July 2000, is a cross-platform operating environment for hosting a wide variety of services and applications, and supporting communication between not only different applications, but also applications written in seemingly different languages (*interoperability*).

Of itself, .Net is neither an operating system nor a programming language. It is more a layer that sits between the lower levels of the platform-specific operating system, and the higher levels of applications, services, libraries, development environments, *etc.* .Net is currently hosted on Windows, FreeBSD, and Apple's OS X (see later for details of MONO and running .Net applications on Linux). Applications can be written in C, C++, C#, Java, J#, Visual Basic, COBOL, ML, Haskell, Oberon, Eiffel, Fortran, Perl, Python, and Smalltalk to name a few. And the list is steadily growing.

While the full .Net includes many interesting components, for the purposes of this report perhaps the more interesting components are:

- **Development environment** of .Net is enriched with many freely available tools, both from Microsoft and other sources. This allows a great deal of experimentation and exploration of compiler technology for both host and target for a wide range of programming models (*e.g.,* object-oriented or functional).

- **Object model** of .Net automatically supports many of the object-oriented features required to implement object-oriented programming languages. This simplifies compiler design, and allows easier experimentation with new programming models.

- **Common language runtime** simplifies compiler development, minimizing the need to develop entire runtime support libraries for any new compiler. As the runtime suppors even the basic data types, there is

greater interoperability between modules written in different languages; so, for example, one could write a new compiler for language A, and as long as the compiler matches A's type system to that of .Net, then applications written in A have access to libraries and test harnesses written in another language, say C#.

The .Net Common Language Infrastructure has been standardised by the international standards body ECMA. Specifically, ECMA-335: Common Language Infrastructure (CLI) specifies a file format, a common type system, an extensible metadata system, an intermediate language, access to the underlying platform, and a factored base class library. As well as the CLI, Microsoft have also standardised C#, as ECMA-334: C# Language Specification.

These standards have opened the way forward for a variety of implementations of .Net, the CLR, and C#. The ROTOR project from Microsoft is a shared-source .Net project, providing compilers, a runtime, and the majority of the CLR to enable many applications to be compiled and interpreted on both Windows and Mac OS X. Another project, this time open-source, is the Mono project, released under various public licenses—the C# compiler under the terms of the GNU GPL, the CLR under the GNU LGPL, and the class libraries are released under the MIT X11 license.

As a side note, all of the applications described in this report have been successfully executed on a Linux box under Mono and a Windows box under the Microsoft .Net runtime, both as C# files compiled with Mono's C# compiler, and as compiled `.exe` files, illustrating the cross-platform nature of .Net.

1.1 Supporting Materials

All of the source code described in this report is freely available for download from the website accompanying the *Compiler Construction* course. It is currently located at

`http://www.cl.cam.ac.uk/Teaching/current/CompConstr/`

Within each directory there is a Makefile and associated source files. Once the files are unpacked the Makefile explains the steps necessary to build the projects.

For projects using the ANTLR package you will need to have this installed on your system prior to building or running the examples. Full details, with downloads, are available from the ANTLR website:

`http://www.antlr.org/`

Chapter 2

Hand-written Recursive Descent Parser

The design of the recursive descent four-function calculator follows that of the example given in section 4.1 of the lecture notes. The implementation described here is written in C# (the sources for this, and all the other examples in this report, are in the included ZIP archive).

The grammar has four operators, and parentheses to modify evaluation order:

```
T -> T + F | T - F | F
F -> F * P | F / P | P
P -> ( T ) | n
```

As mentioned in the lecture notes, the actual implementation uses right recursion to avoid loops in the parser. The top-level function simply repeatedly calls for `T` items until there is no more input:

```
public void Parse ()
{
    while( true )
    {
        lexer.lex();
        switch( lexer.getToken() )
        {
        case Token.EOF: return;

        default: Console.WriteLine( "= " + RdT() );
                 break;
        }
    }
}
```

The first rule, for "`T -> T + F | T - F | F`" produces the parsing function `RdT()`:

```
Int32 RdT()
{
    Int32 result = RdF();

    while( true )
        switch( lexer.getToken() )
        {
        case Token.PLUS:  lexer.lex();
                          result += RdF();
                          continue;

        case Token.MINUS: lexer.lex();
                          result -= RdF();
                          continue;

        default:          return result;
        }
}
```

The second rule, for "`F -> F * P | F / P | P`" produces the parsing function `RdF()`:

```
Int32 RdF()
{
    Int32 result = RdP();

    while( true )
        switch( lexer.getToken() )
        {
        case Token.TIMES:  lexer.lex();
                           result *= RdP();
                           continue;

        case Token.DIVIDE: lexer.lex();
                           result /= RdP();
                           continue;

        default:           return result;
        }
}
```

The final rule, "`P -> ( T ) | n`" produces the parsing function `RdP()`:

```
Int32 RdP()
{
    Int32 result;

    switch( lexer.getToken() )
    {
    case Token.NUMBER: result = (Int32)lexer.getNumber();
                       break;

    case Token.LPAREN: lexer.lex();
                       result = RdT();
            if ( lexer.getToken() != Token.RPAREN )
            {
                Console.WriteLine( "Error: missing closing ')'" );
                throw new System.Exception();
            }
            break;

    default:   Console.WriteLine( "Error in expression." );
               throw new System.Exception();
    }

    lexer.lex();
    return result;
}
```

For brevity the above code has been simplified and the code for the lexer has been omitted. Fully working source code may be found in the accompanying source archive.

Exercises

1. Extend the language with unary minus, remainder, and exponential operators.

2. Add user variables to this calculator. As a first instance fix the number of user variables to, say, twenty-six variables (hint: use single letters of the alphabet as symbolic names). You will also need to extend the language to add the assignment operator.

3. Extend the user variable feature to allow arbitrary variable names.

4. Change the numeric types of this calculator to support arbitrary precision decimal arithmetic. For example, consider that a signed 32-bit number is too small to record the gross domestic product of the USA.

Chapter 3

Machine-generated Parser

The previous chapter presented a hand-written recursive-descent calculator. In this chapter we present a machine-generated version of the same program as a comparison of the two techniques. The tool we use is called ANTLR, and it generates LL(k) (*i.e.,* recursive-descent) parsers which output an Abstract Syntax tree (AST).

One goal of the design of this tool was that the output code should be almost human-readable; the more curious reader will find it interesting to examine the output of the ANTLR parser generator of the grammar described below.

Note that in the following grammar, `!` denotes non-AST tokens (*i.e.,* those which do not produce nodes in the resulting AST), and `^` denotes tokens which produce AST root nodes. Terminal symbols, defined elsewhere in the provided source file, are written in uppercase.

```
class CalcParser extends Parser;

expr
    :   subexpr SEMI!
    ;

subexpr
    :   addexpr (MINUS^ addexpr)*
    ;

addexpr
    :   mulexpr (PLUS^ mulexpr)*
    ;

mulexpr
    :   divexpr (STAR^ divexpr)*
    ;

divexpr
    :   atom (DIVIDE^ atom)*
    ;

atom:   INT
    |   LPAREN! subexpr RPAREN!
    ;
```

The above grammar will be turned into a machine which constructs an abstract syntax tree (AST) of the expression. To actually compute the result, we define a tree walker that does the actual computation:

```
class CalcTreeWalker extends TreeParser;

expr returns [float r]
{
    float a,b;
    r=0;
}
    :   #(PLUS   a=expr b=expr) {r = a+b;}
    |   #(MINUS  a=expr b=expr) {r = a-b;}
    |   #(STAR   a=expr b=expr) {r = a*b;}
    |   #(DIVIDE a=expr b=expr) {r = a/b;}
    |   #(LPAREN a=expr RPAREN) {r = a;}
    |   i:INT         {r = Convert.ToSingle(i.getText());}
    ;
```

The `#` construct defines an AST node pattern, with the first argument being the root node, followed by the child nodes. For example, the first rule matches an AST node with a PLUS terminal symbol root node, and two sub-trees '*a*' and '*b*'.

Note that the code in this report has been laid out for ease of readability; the grammar could be written in six lines, with another six for the tree walker.

This chapter has shown how automated toosl (*i.e.,* ANTLR in this case) can greatly simplify the task of writing compilers and interpreters. The ANTLR grammar uses several meta-characters to simplify the layout of grammar rules, which are summarised below:

- `!` identifies tokens which, although necessary for the syntactic structure, do not appear in the AST;
- `^` identifies tokens which form the roots of new ASTs (or subtrees thereof);
- `#` is short-hand notation for describing AST nodes, taking a list as an argument consisting of the root node and and child nodes of the AST node;
- `*` denotes zero-or-more copies of the preceding item;
- `?` denotes zero-or-one (*i.e.,* optional) item.

Exercises

1. Repeat the same exercises from the previous chapter. Compare how much effort is required (lines of codes, hours spent typing, *etc*) in using automated tools compared to manual methods.

2. Extend the calculator language with user-definable functions, in the style:

   ```
   f(a,b,c) = ( a + b ) * c;
   ```

Chapter 4

Elisa .Net— Experimental Language Compiler

Elisa is a simplified C-like language designed for this project to aid the teaching of modern compiler techniques. The compiler presented here generates intermediate code for a stack machine. The current implementation directly targets the .Net virtual machine, but could be retargeted to other stack machines, *e.g.,* the JVM.

It differs from the example compiler in the ROTOR package by (a) using an automated tool (ANTLR), and (b) not relying on .Net CLR support for generating target code.

4.1 The Elisa Language

The Elisa language looks very similar to C, and indeed the standard C preprocessor can be used to provide, for example, source file inclusion and conditional compilation features. The following discussion refers to parts of the Elisa grammar (simplified for exposition), included for reference.

A program is a list of declarations, with each declaration being either a global variable or a function:

```
program
    : ( declaration )+
    ;

declaration
    : declarator SEMI!
    | declarator LPAREN! ( parameter_list )? RPAREN! stmt_block
    ;
```

A declarator specifes the type, a name, and an optional array suffix:

```
declarator
    : KW_INT IDENT ( arraydecl )?
    ;

arraydecl
    : LBRACKET! INTEGER RBRACKET!
    | LBRACKET! RBRACKET!
    ;
```

An array declaration with no dimension is treated as if the dimension was zero[1].

Functions take an optional comma-separated list of parameters:

```
parameter_list
    : declarator ( COMMA! declarator )*
    ;
```

The body of a function is a statement block: a list of zero or more declarators followed by zero or more statements.

```
stmt_block
    : LCURLY! ( declarator SEMI! )* ( statement )* RCURLY!
    ;
```

Declarators have already been described. A statement can be an expression (including assignment), `if`, `while`, `return`, another statement block, or the empty (null) statement:

[1]This supports pointer-like behaviour.

```
statement
    : expression SEMI!
    | if_stmt
    | while_stmt
    | return_stmt
    | stmt_block
    | SEMI!
    ;
```

`if` statements provide Elisa with a selection mechanism. The current implementation of the Elisa compiler uses a feature of ANTLR to handle the *"dangling-else"* issue:

```
if_stmt
    : KW_IF! LPAREN! expression RPAREN! statement
        ( options { warnWhenFollowAmbig = false; } :
        KW_ELSE! statement )?
    ;
```

The `while` statement takes an expression and a statement (which may be a statement block). The statement is repeatedly executed while the expression evaluates to true.

```
while_stmt
    : KW_WHILE^ LPAREN! expression RPAREN! statement
    ;
```

The last statement, `return`, takes a single expression, and exits the enclosing function.

```
return_stmt
    : KW_RETURN^ expression SEMI!
    ;
```

Expressions in Elisa are handled through a stack of grammar rules to enforce the precedence directly in the grammar, rather than though parser-tool directives.

```
expression
    : or_expr ( ASSIGN^ expression )?
    ;

or_expr
    : and_expr ( OR^ and_expr )*
    ;

 ... etc ...

mult_expr
    : unary_expr ( ( TIMES^ | DIVIDE^ | MOD^ ) unary_expr )*
    ;

unary_expr
    : ( NOT^ | COMP^ | MINUS^ )? primary_expr
    ;
```

Note that we do not use left-recursion here. In general, $LL(k)$ grammars do not like left-recursion grammars, since it cannot be determined statically how large k (the number of lookahead tokens) needs to be to parse all possible inputs. In general most left-recursion rules can be rewritten using ANTLR's EBNF-style operators, as shown above.

Primary expressions include numbers, characters, variables, function calls, and parenthesised expressions:

```
primary_expr
    : INTEGER
    | CHAR
    | IDENT
    | IDENT LBRACKET! expression RBRACKET!
    | IDENT LPAREN! ( expression ( COMMA! expression )*  )? RPAREN!
    | LPAREN! expression RPAREN!
    ;
```

The lexical analyser is not described here; the reader is directed to the source of the Elisa compiler, where the patterns for the lexical elements are to be found.

4.2 Building the Elisa Compiler

The Elisa compiler requires the use of ANTLR to auto-generate the lexer and parser code, and a C# compiler to produce the executable compiler. The `ElisaParser.g` file is processed by ANTLR, producing two files: `ElisaParser.cs` and `ElisaCodeGenerator.cs`. All the remaining files (including those generated by ANTLR) are passed to the C# compiler for compilation and linking with the CLR runtime libraries.

The remaining files describe various components of the Elisa compiler:

- `Elisa.cs` is the top-level module, handling command line options, selecting the target code generator, and calling the lexer and parser.
- `ElisaParser.cs` is produced by ANTLR and contains the parser code.
- `ElisaCodeGenerator.cs` is also produced by ANTLR and contains the code for the tree walking code generator.
- `SymbolTable.cs` provides symbol table and basic type management.
- `CodeGen.cs` is an abstract class which specifies the code generator API, and
- `DotNetCodeGen.cs` is the .Net code generator class which provides a .Net code generator.

4.3 Intermediate Code Generation

The Elisa compiler generates stack-based intermediate code in two phases. The first phase parses the input and produces abstract syntax trees (ASTs). This is handled by the first part of the `ElisaParser.g` file. The second phase walks the ASTs and emits stack code.

For stack machine targets the intermediate code output of the compiler is the target code itself, no further translationa necessary to execute it. For non-stack machines (*e.g.,* `x86`) a further step of target code generation will be required.

4.3.1 Declarations

Global declarations (both functions and global variables) are added to the global symbol table as they are processed. Global variables are added to the

symbol table, and we choose to decorate their target-specific name xName with the prefix "EL_" within the tree walker:

```
declaration
{
    Symbol s;
    ArrayList plist = null;
}
    : #(DECLARATION s=declarator {
        globals.Add( s );
        s.xName = "EL_" + s.Name;
    } )
```

4.3.2 Functions

Functions require a little more treatment. First, we mark the function identifier's type as a function type. All functions other than `Main` are decorated with the `EL_` prefix, as before. Then a new symbol table scope is created for the function parameters:

```
    | #(FUNCTION_DECL s=declarator {
        s.Type = new Type( Type.T.FUNC, 0, s.Type );

        if ( s.Name == "Main" )
            s.xName = s.Name;
        else
            s.xName = "EL_" + s.Name;

        symtab = new Symboltable( symtab );
```

Once the parameters have been processed, the function body is then created, the list of local variables is initialised, and we define the exit label. Local variables are announced to the backend during parsing. Array declarations defined at any scope within a function generate code to allocate sufficient memory for the array. For the .Net target, this code calls the runtime function `[mscorlib]System.Int32`, passing it the number of words to allocate.

```
        ( plist=parameter_list )? {
            target.openFunction( s, plist );
            locals      = new ArrayList();
            exitlabel   = label++;
            needsReturn = true;
    }
```

Finally, the body of the function itself, which is treated as any other statement block (see below) is processed. A check is then made on `needsReturn`, emitting a warning and a default value if the last statement was not a `return` statement. We then emit the exit label, the instruction to return control to the function caller, and we close the function itself. The last action is to remove the function parameter symbol table.

```
        stmt_block {
            if ( needsReturn )
            {
                Console.Error.WriteLine( "Warning: missing return" );
                target.emit_iconst( 0 );
            }

            target.emit_label( exitlabel );
            target.emit_ret();
            target.closeFunction( locals );

            // remove parameter table
            symtab = symtab.Parent;
        }
```

For the .Net target all global array variables are allocated by code generated for the constructor method. All declarations, both variables and functions, are declared as static members and methods respectively.

4.3.3 Expressions

Stack machines are especially good at computing expressions[2]. Generating expression code for stack machines is done by a recursive-descent tree walker that builds the expression on the stack, with the result of the expression always on the top of the stack.

[2]Hewlett Packard produced (and still do to this day) many engineering and scientific calculators that were based on stack-oriented Reverse Polish Notation (RPN).

Constants are loaded directly onto the evaluation stack. Depending on their scope, variables are read from the local variable stack (including function parameters) or from global memory. Because Elisa does not support explicit pointers as found in, say, C, there is no dereferencing operator. However, pointer-like behaviour can be simulated with arrays, whereupon the index expression is computed, and then an array access from the array base is performed.

Assignments require careful handling. In Elisa, the address of the lvalue expression is computed first, including any side-effects. Then the rvalue expression is computed. Finally, the store of the rvalue into the location addressed by the lvalue is generated.

One careful issue with assignments on stack machines is to ensure that the result of the assignment, even after a store instruction, is left on the top of the stack. The assignment operator achieves ensures this with the "`dup`" instruction, which duplicates the value on the top of the stack. Assignment statements pop the remaining unused value from the stack. Thus we correctly compile such statements as:

```
x = y = 1 + z;
```

where the first assignment (to `y`) duplicates the result of the right-hand expression on the stack, before popping the topmost copy off the top of the stack and storing it in `y`. The second assignment, to `x`, similarly duplicates the value on the top of the stack, takes the topmost item off the stack and stores it in `x`. Finally, the end of the expression statement pops the unused value off the top of the stack.

All expressions take two labels: `tlab` and `flab`. Expressions which are computed solely for their value have both these labels set to zero. Where an expression is required to determine control flow (`if...then...else`) one of `tlab` or `flab` will be given a value other than zero. Then, depending on which label is non-zero, code wil be emitted to jump to the non-zero label if the result of the expression is zero or non-zero.

The logical operators must implement short-circuit evaluation (see lecture notes, section 6.6). They do not in themselves compute a value. For example, the logical AND operator `&&` evaluates its left-hand first. If the result of that is true, it then evaluates the right-hand expression, otherwise it branches to the false label. If the result of the right-hand expression is similarly false, the code branches to the false label. Otherwise execution continues at the true label.

If the true label is zero and flab is L, then we generate code of the form

```
    if lexpr == 0 goto L
    if rexpr == 0 goto L
```

Conversely, if tlab is L and flab is zero, then we generate code:

```
    if lexpr == 0 goto L'
    if rexpr != 0 goto L
L':
```

And similarly for logical OR and logical NOT.

In addition, if neither label is supplied then the logical operator is required to produce a numeric value: 0 for false, 1 for true. This is used for code such as

```
put('0' + ( x == y ) );
```

which prints a '0' or a '1' depending on the values of `x` and `y`.

4.3.4 Statements

The Elisa language has a few basic statements. In the following, note that labels are represented in the compiler as positive integers.

A statement block is defined as a block beginning with zero or more declarations, then zero or more statements. A new symbol table is opened, and any locals are both added to the symbol table, and announced to the backend:

```
stmt_block
{
    Symbol s;
}
    : #(STMT_BLOCK {
        symtab = new Symboltable( symtab );
    } ( s=declarator {
        locals.Add(s);
        target.announceLocal( s );
    } )* ( statement )* { symtab = symtab.Parent; } )
    ;
```

On exit of the statement block, the block's symbol table is removed, preventing any variables defined in the block from being used outside the block.

Assignment Statements

Expressions are compiled for their effects (usually assignment). As noted above, all expressions leave their result on the top of the stack, so an expression statement pops the redundant result off the stack:

```
: ty=expression[0,0]  {
    target.emit_pop();
    needsReturn = true;
}
```

The setting of `needsReturn` tells the code generator that the current last statement is *not* a `return` statement, causing the function epilogue code to print a warning and to emit default " `return 0;`" statement. Conversely, if the last statement *is* a `return` statement, this flag is cleared and function compiles without a warning.

If Statements

The `if` e `then` S_T `else` S_F statement (where the `else` clause is optional) first evaluations expression e, and if true (*i.e.,* non-zero) executes statement S_T. If there is an `else` clause, then if e evaluates to false, the S_F is executed, otherwise control flow follows to the next statement. Note that both S_T and S_F can be single statements or statement blocks.

```
if_stmt
{
    Type ty;
    uint flab   = label++;
    uint endlab = label++;
}
    : #(IF_STMT ty=expression[0,endlab] statement {
        target.emit_label( endlab );
    } )

    | #(IF_ELSE_STMT ty=expression[0,flab] statement {
        target.emit_branch( endlab );
        target.emit_label( flab );
    } statement {
        target.emit_label( endlab );
    } )
    ;
```

The code generated for the `if` statement needs one or two labels, depending

on the presence, or absence, of the `else` clause. When there is an `else` clause, we generate code of the form

```
        e[0,LF]
        S_T
        bra LX
LF:
        S_F
LX:
```

The evaluation of the expression e takes two labels, as described above. The true label is 0, to indicate that control flow should fall through the expression to the true statement, S_T. If the expression is false, then it should jump to label LF. After S_T we branch to the exit label, LX. This is followed by the false label, LF, and the false statement, S_F.

The code structure for an `if` statement without an `else` clause is similar, but without the branch to LX, without S_F, and without LX.

While Statements

The looping `while` statement is of the form `while` e S. For as long as expression e evaluates to true (non-zero), statement S is executed. The code structure for the `while` statement is of the form

```
L:
        e[0,LX]
        S
        bra L
LX:
```

The code that actually generated `while` loops is equally simple:

```
while_stmt
{
    Type ty;
    uint toplab = label++;
    uint btmlab = label++;
}
    : #(KW_WHILE { target.emit_label( toplab ); }
        ty=expression[0,btmlab] statement )
    {
        target.emit_branch( toplab );
        target.emit_label( btmlab );
    }
    ;
```

Function Calls

Function calls require very little processing in the front end, other then evaluating the function arguments and the function address. The back end code generator then emits the code to call the function, since the argument values will already be on the evaluation stack in the correct order.

```
#(FUNCTION fid:IDENT {
        ArrayList arglist = new ArrayList();
    } ( rty=expression[0,0] {
        arglist.Add(rty);
    } )* ) {
    Symbol fs = symtab.find( fid.getText() );

    // Catch special cases
    if ( fid.getText() == "put" )
    {
        target.PUT();
    }
    else if ( fid.getText() == "get" )
    {
        target.GET();
    }
    else
    {
        target.emit_call( "int32 " + target.BaseName + "::"
                                   + fs.xName, arglist );
    }

    mkPredicate( tlab, flab );
    exprty = fs.Type.Of;
}
```

There are two special function names in Elisa—`put` and `get`. The above code catches these special cases and passes them on to the code generator to emit the target-specific code. All other functions have their names decorated, and the argument list supplied to the back end.

On return from the function, the return value will be on the top of the evaluation stack. This is optionally converted into a predicate value if required.

Return Statements

The `return` statement takes a single expression argument, whose value becomes the result of the function. The code generated for the `return` statement evaluates the expression, and then jumps to the function's exit label.

```
return_stmt
{
    Type ty;
}
    : #(KW_RETURN ty=expression[0,0] )
    {
        target.emit_branch( exitlabel );
    }
    ;
```

Accessing Variables

Variables can appear on both the left hand side of an assignment (an *lvalue*) and on the right hand side (an *rvalue*).

Rvalues are the simpler case. Variables represent the address of some storage location in the processor's memory. To read from the address represented by that variable we use an indirection node to implement reading memory.

```
(INDIR lty = iid:expression[0,0] {
    if ( iid.Type == ARRAY )
    {
        // Load from array and strip off ARRAY from result type
        target.emit_ldelem();
        lty = lty.Of;
    }
    else if ( iid.Type == IDENT )
    {
        s = symtab.find( iid.getText() );

        if ( s.Scope == 0 )
            target.emit_ldglobal( s );
        else if ( s.Scope == 1 )
            target.emit_ldarg( s.Offset );
        else
            target.emit_ldloc( s.Offset );
    }
    else
    {
        Console.Error.WriteLine( "Error: bad rvalue" );
        throw new System.Exception();
    }

    mkPredicate( tlab, flab );
    exprty = lty;
    } )
```

At this stage in compilation the types of variables are either identifiers or arrays. If the variable is an array name then we use the special `ldelem` instruction to index into the array. Otherwise we simply load from memory, which can be global memory, function parameters (which require special instructions), or local stack. Finally, we optionally make this a predicate value (0 or 1) if either tlab or flab is non-zero.

Lvalues require special care. Their address must only be computed once, *and* before the right hand side of an assignment is evaluated.

```
#(ASSIGN lty = lid:expression[0,0] rty = rid:expression[0,0]
{
    target.emit_dup();

    if ( lid.Type == ARRAY )
    {
        target.emit_stelem();
    }
    else if ( lid.Type == IDENT )
    {
        s = symtab.find( lid.getText() );

        if ( s.Scope == 0 )
            target.emit_stglobal( s );
        else if ( s.Scope == 1 )
            target.emit_starg( s.Offset );
        else
            target.emit_stloc( s.Offset );
    }
    else
    {
        Console.Error.WriteLine( "Error: bad lvalue" );
        throw new System.Exception();
    }

    exprty = rty;
} )
```

The front end constructs the assignment node with the lvalue as the first child and the rvalue as the second child. Once both expressions have been computed and their values placed on the evaluaion stack we first duplicate the rvalue, then emit the store instruction (dependent on the type and scope of the lvalue).

Exercises

1. The current version of the language only supports single-dimension arrays. Extend the language to support multi-dimensional arrays. The grammar will need modifying to support both multi-dimensional declarations and multi-dimensional accesses.

2. Add support for strings to Elisa. The change to the grammar is trivial, while supporting strings in the compiler requires attention to such issues as where the strings are stored in the output. Would you treat strings and arrays as distinct types, or related (*i.e.,* a string as a single-dimension array). How would you implement simple operations on strings, such as '`+`' (*i.e.,* concatenation)?

3. Add declaration initializers (*e.g.,* "`int a=12;`") to Elisa. For scalar variables this is easy, but what about arrays? How would your solution handle multi-dimensional arrays? What about partial initializers (*e.g.,* "`int a[10]={3,2,1,0};`".

4. Extend the set of operators in Elisa with, for example, compound assignment operators ("`x+=5;`"), more relational operators, *etc.*

5. Add the conditional operator "`c ? e1 : e2`" to Elisa.

6. Rewrite the grammar rule for the `if` statement to avoid using tool-specific hacks for handling the *"dangling-else"* ambiguity.

7. The code structure for the `while` loop is not as efficient as it could be, requiring $2n + 1$ branches to iterate n times. Design a code structure which is faster.

8. Add the "`do S while(e);`" statement to Elisa. How does this relate to your answer to the previous exercise?

9. Extend Elisa with the `for(e1; e2; e3)` statement. As for other languages, all three expressions are optional, with varying default behaviours. Include support for *all* versions of this statement.

10. Most languages have some form of `switch` or `select` statement. Design one for Elisa. How do you propose to deal with breaks and fall-throughs? For example, C allows fall-through from one case block to the nest one, while C# does not.

11. Implement a lambda evaluator, so that one could write, for example:

```
int computefac(int p) {
    let int fac(int z) = z == 1 ? 1 : z*fac(z-1); in {
        return fac(p);
    }
}
```

Note that the above is a recursive lambda (refer to lecture nodes, section 9.8). Also consider the scope in which `fac()` executes in, and what variables it has access to. Investigate the many ways that have been developed to solve this issue. Choose one and implement it.

12. The grammar has very little error checking and reporting. A production compiler should provide far more information to the user about errors and potential errors (warnings) found in the source program during compilation. Augment the grammar (a) with more informative error messages, and (b) with additional rules to catch common errors so that more meaningful error messages can be generated. How would you change the grammar to be able to recover from minor errors?

13. Port the Elisa compiler to the JVM. Ideally, make the choice between targets a command line flag for the compiler (*e.g.,* "`--dotnet`" or "`--jvm`").

14. Section 7 of the lecture notes describes a method for compiling stack-based code into non-stack (*e.g.,* `x86`) code. Implement a simple `x86` code generator that will produce output that can be assembled into an executable program (*e.g.,* using `as` on Unix or the free `MASM` for Windows).

15. Taking guidance from section 7 of the notes, improve the quality of the code output by your `x86` code generator.

16. Design a decompiler that matches your Elisa compiler. Ideally, if C represents the compiler, and D the decompiler, then $D(C(S)) = S$ for source S, subject to differences in local variable names.

Chapter 5

Resources

Further information can be found in the following forms.

5.1 Books

- Programming C# (3rd edition), *Jesse Liberty, O'Reilly*
- Inside Microsoft .Net IL Assembler, *Serge Lidin, Microsoft Press*

5.2 Web Resources

5.2.1 Course Support Materials

- *Compiler Construction* homepage
  `http://www.cl.cam.ac.uk/Teaching/current/CompConstr/Elisa`

5.2.2 Language Development Tools

- ANTLR
  `http://www.antlr.org`
- Grammatica
  `http://www.nongnu.org/grammatica/`

5.2.3 Other .Net Compiler Projects

- LCC.Net
  `http://www.cs.princeton.edu/software/lcc/`

- Cflat
  http://iti.spbu.ru/eng/grants/Cflat.asp

5.2.4 .Net Online

Microsoft

- Shared-Source CLI (and Rotor)
  http://msdn.microsoft.com/netframework/using/understanding/cli/default.aspx
- .Net
  http://msdn.microsoft.com/netframework/downloads/updates/default.aspx

ECMA

- CLI
  http://www.ecma-international.org/publications/standards/Ecma-335.htm
- C#
  http://www.ecma-international.org/publications/standards/Ecma-334.htm

Other Sources

- MONO
  http://www.go-mono.com