A Sequence of Lab Exercises for an Introductory Compiler Construction Course

Lawrence A. Coon
Department of Computer Science
Rochester Institute of Technology
Rochester, New York 14623
(716) 475-2237
e-mail: lac@cs.rit.edu

1. Abstract

A sequence of laboratory exercises for an introductory compiler construction course is described. The labs are based on four increasingly complex versions of an imperative language designed so that each version builds on the previous. The third version supports integer and character data types and arrays of integers and characters. The fourth version adds procedures, but has only integer data. The procedures do not nest, but direct recursion is supported.

2. Introduction

There are several popular approaches to designing projects for an introductory compiler construction course. These include:

- the "tiny" language approach: using a subset of a popular language like Pascal or Ada. [Aho, Fis]
- the "mini" language approach: defining a language that concentrates on a specific feature with minimal support for other features. [Mar]
- the "filling in the blanks" approach: providing the skeleton of a compiler with implementation details for critical features to be provided by the student.

These can generate code for real machines, or for hypothetical machines for which a simulator is available.

In addition, there is the "reading" approach where students analyze and discuss the code for an existing compiler. This is probably most often used in conduction with one of the above.

Each has its advantages.

Basing the project language on a subset of a language known by the students minimizes the overhead needed to understand the language and enables the student to concentrate on the compiler design itself. Furthermore, simple variations of language features can generate interesting discussions about language design without adding undo confusion. These can also disabuse students of the notion that all that is good is in their favorite language, and introduce them to the idea that language and compiler design not only go hand-in-hand but are based on the art of compromise.

Using a mini-language that concentrates on a basic feature and reducing other language features to a bare core appears to maximize learning and minimize busy work. However, the core language must be designed carefully, and the satisfaction of ending up with a "real" language at the end of the course is missing.

Students can produce ugly code and few instructors have the time or the grading support needed to give the feedback they'd like, so reading (and modifying) well designed real code is also important.

In the following, a set of goals is developed based on a combination of the above approaches. Then a series of projects that meet those goals is discussed in detail.

3. Goals

The fundamental objective is to cram the basics of compiler design into a project that a diligent student can complete in one term. (At my school this means a 10 week quarter.) To that end the following goals were adopted:

(1) **Use tools where possible**
The mastery of some basic compiler construction tools is an important end in itself. Tool use also avoids students getting caught up in the details of writing lexical analyzers and parsers (which should be the subject of other courses) and allows them to quickly get on to more important issues. The tools I selected were the ubiquitous lex and yacc because of their wide availability and generality. [Lev]

(2) **Use both top-down and bottom-up parsing**
Introduce recursive-descent techniques and use them for the simplest project, then shift to top-down techniques using yacc for the rest of the projects.

(3) **Use ad hoc semantics**
Time considerations force ad hoc semantics (unless students enter with an understanding of one of the standard methodologies and appropriate support tools are available.)

(4) **Concentrate on the front end**
Again, time considerations force an emphasis on the well understood and a de-emphasis on the more "interesting" issues such as register allocation, optimization, etc. Also this is

an undergraduate course and front end issues are what will be the most useful for those that get programming jobs and might have to create interfaces and other simple language translators. For example, parsing and interpreting an argument stream to a cgi program used at a World Wide Web site.

(5) **Use quads as an intermediate code**
Quads are easy to understand, and they can be readily interpreted or used to generate code. Other approaches might be more interesting, but time is a hard master.

(6) **Use a quad interpreter for the more complex versions**
A simple template based code generator for an assembly language can be assigned for the simpler versions. However when implementing more advanced features code generation gets tedious and time consuming, so avoid it by just interpreting the quads.

(7) **Stick to the most basic control structures**
Use "while" as the only looping construct and "if/then/else" as the only decision construct. This gives full functionality without the busywork of implementing other "nonessential" constructs.

(8) **Only use two data types**
They should face the issues of type checking and two types is the minimum. I use integer and character, and include arrays of both. This way you have simple strings for free.

(9) **Stick to basic subroutine issues**
Use simple C-like procedures with no nesting to avoid the complex symbol table management and the complex runtime stack management needed to resolve scope.

(10) **Investigate some simple language design issues**
Here is where a few differences from the usual world of C, Pascal and Modula2 can be introduced. Some of the issues I like are:

- The use of end-of-line as a statement terminator rather than a semicolon.
- The use of end-of-line as part of the syntax of language elements to enforce style standards; e.g. the if syntax has an eol after the "then" to force the body to start on a separate line.
- Simple run-time and compile-time error detection; e.g. division by zero and array bounds checking.
- Global and local scoping for variables.

4. Overview of the Languages

The project language is an imperative language with features from Pascal and C. It is developed in four stages: eenie, meenie, miney and moe. Each of the first three is a superset of the previous, and is totally written by the student from a BNF syntax specification and an informal statement of the semantics. The last version only supports integer data and no arrays — the emphasis is on procedures and scoping variables. For this version, the student adds code to support the new features to a working version from which that code has been stripped.

4.1. Eenie

Eenie is a simple expression language with no control structures. (See Appendix A for a complete specification). The language features supported are:

- input and output of integers.
- variable declaration
- assignment
- basic integer arithmetic

Here is an example eenie program:

```
--
-- Example
--
program example has
decls
        int sam, y
body
        read (sam)
        y <- sam + -3
        write (y)
end example
```

This leads to two projects. The first is to implement it using recursive descent (lex for lexical analysis but no yacc). The second is to implement an interpreter using yacc. Students have already seen how to handle arithmetic when studying lex and yacc, so they only need to add the io, declaration and syntactic sugar for the program structure.

4.2. Meeney

Meeney adds simple control structures to eenie. (See Appendix B for a complete specification). The language features supported are:

- relational operators
- boolean expressions in the C sense of zero meaning TRUE. (Other versions used the traditional Boolean operators.)
- if statement with optional else
- while statement

Here is an example meeney program:

```
program example has
locals
        int a
body
        a <- 9
        while a > 0 do
                a <- a-1
                if ( a - 2*(a/2)) then
                        write(a)
                endif
        endwhile
        writeln
end example
```

4.3. Miney

Miney adds a character data type and arrays to meeney. (See Appendix C for a complete specification). The language features supported are:

- character and integer data types and type checking
- character and integer arrays with run-time and compile-time bounds checking

- block assignment
- simple string support using character arrays

Here is an example miney program:

```
        program example has
        decls
              int array x with 5 elts
              char array y with 5 elts
        body
              x[1] <- 999
              write ( x[1] )
              y[1] <- 'a'
              y[2] <- 'b'
              y[3] <- 'c'
              y[4] <- '^'
              write ( y )
              writeln
        end example
```

4.4. Moe

Moe is a procedure language with only integers and no arrays. (See Appendix D for a complete specification). The language features supported are:

- procedures with recursion
- parameter passing by reference
- global and local scoping

Here is an example moe program:

```
        program factorial has
        ------------------------
        proc fact( int x, y ) has
        locals
              int a, b
        body
              a <- x
              if a = 1 then
                    y <- 1
              else
                    a <- a - 1
                    fact(a, b)
                    y <- x * b
              endif
        endproc fact
        ------------------------
        locals
              int a, ans
        body
              a <- 5
              fact(a, ans)
              write(ans)
              writeln
        end factorial
```

A global declaration section is added whose variables are available in all procedures and in the main. Also added is the ability to declare local variables for procedures and for the main.

5. Conclusions

For the last 7 years I have successfully utilized this approach in one quarter introductory undergraduate and graduate compiler courses where it is imperative that a meaningful amount of material be covered in a short period. Student feedback is quite positive and student success is very high.

References

[Aho] Aho, Sethi and Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1986.

[Fis] Fisher and LeBlanc, *Crafting a Compiler*, Benjamin Cummings, 1988.

[Mar] Marcotty and Ledgard, *Programming Language Landscape*, SRA, 1986.

[Lev] Levine, Mason and Brown, *lex and yacc*, O'Reilly & Associates, 1992.

Appendix A: Eenie

BNF for eenie

```
pgm        ::= head decpart bodypart tail
head       ::= program NAME has EOS
decpart    ::= decls EOS int varlst EOS | ε
bodypart   ::= body EOS stmtlst
tail       ::= end NAME
varlst     ::= varlst , ID | ID
stmlst     ::= stmtlst stmt | stmt
stmt       ::= io EOS | asgn EOS
io         ::= read ( ID ) | write ( exp )
               | writeln
asgn       ::= ID <- exp
exp        ::= exp + term | exp - term | term
term       ::= term * factor | term / factor
               | term % factor | factor
factor     ::= ID | NUM | ( exp ) | - factor
```

Semantics for eenie

The reserved words are: program, has, decls, int, body, end, read, write, and writeln. The ID and NUM stand for identifier and integer constant respectively. Identifiers will be any sequence of lower case letters. There is a separate declaration section for variables, as in Pascal, which are always of type integer.

When an identifier is defined, you should create a symbol table entry after checking for double definition. The symbol table should also be used for storage. When an identifier is referenced simply look it up in the table and use (or store) the value there.

The semantics are as expected. Comments start with two dashes and continue to the end of the line (as in Ada). For this language, the end of statement (EOS) is the end of the line. This has the advantage of forcing some formatting conventions on a program. For example, the declaration section is introduced by a line containing only the reserved word decls. This also has the disadvantage of restricting the size of arithmetic expressions. To get around this we will use an @ character as a line continuation character. The sequence "@\n" indicates that the next line is a continuation of the current line. When printing an integer, leave one space after it so that when more than one is printed on a line the values are readable.

Appendix B: meeny

BNF for meeney

```
pgm        ::= head decpart bodypart tail
head       ::= program NAME has EOS
decpart    ::= decls EOS int varlst EOS | ε
bodypart   ::= body EOS stmtlst
tail       ::= end NAME
varlst     ::= varlst , ID | ID
stmlst     ::= stmtlst stmt | stmt
stmt       ::= io EOS | asgn EOS | loop EOS
               | condl EOS
```

```
condl     ::= if bexp then EOS stmtlst endif
              | if bexp then EOS stmtlst else EOS
              stmtlst endif
loop      ::= while bexp do EOS stmtlst endwhile
bexp      ::= exp | exp relop exp
relop     ::= < | > | <= | >= | = | #
io        ::= read ( ID ) | write ( exp ) | writeln
asgn      ::= ID <- exp
exp       ::= exp + term | exp - term | term
term      ::= term * factor | term / factor
              | term % factor | factor
factor    ::= ID | NUM | ( exp ) | - factor
```

Interpreting meeney

You are to implement a translator for the language that produces quads and interprets them. You are to submit all necessary files including a Makefile, yacc and lex source files. Running make should produce an executable module called *meeney* that compiles the source file into quads as described below and interprets them. If *meeney* is given a flag -q, it should write the quads to a text file called name.q (where name is the name of the source file) before doing the interpreting.

The quad operators you will use in this lab are:

+	add
-	subtract
*	multiply
/	divide
%	modulo
<	less than, if opd1 is less than opd2, set result to 1 else to 0
>	greater than, if opd1 is less than opd2, set result to 1 else to 0
l	less than or equal, if opd1 is less than or equal to opd2, set result to 1 else to 0
g	greater than or equal, if opd1 is greater than or equal to opd2, set result to 1 else to 0
e	equal, if opd1 is equal to opd2, set result to 1 else to 0
#	not equal, if opd1 is not equal to opd2, set result to 1 else to 0
r	read an integer into result
w	write an integer from result
n	write a newline
=	assign the value of the first operand to result
t	test first operand, if it is zero jump to the quad in result
j	jump to the quad in result
?	handle a runtime error, result will hold an error message number
h	halt

Appendix C: Miney

BNF for miney

```
pgm       ::= head decpart bodypart tail
head      ::= program NAME has EOS
decpart   ::= decls EOS declst | ε
declst    ::= decl declst | decl
decl      ::= int varlst EOS
              | char varlst EOS
varlst    ::= varlst , vardecl | vardecl
vardecl   ::= ID | array ID with NUM elts
bodypart  ::= body EOS stmtlst
tail      ::= end NAME
stmlst    ::= stmtlst stmt | stmt
stmt      ::= io EOS | asgn EOS | loop EOS
              | condl EOS
condl     ::= if bexp then EOS stmtlst endif
              | if bexp then EOS stmtlst else EOS
              stmtlst endif
loop      ::= while bexp do EOS stmtlst endwhile
bexp      ::= exp | exp relop exp
relop     ::= < | > | <= | >= | = | #
io        ::= read ( var ) | write ( exp ) | writeln
asgn      ::= var <- exp
exp       ::= exp + term | exp - term | term
term      ::= term * factor | term / factor
              | term % factor | factor
factor    ::= var | NUM | CHAR | ( exp ) | - factor
var       ::= ID | ID [ exp ]
```

Semantics for miney

Initialize integer arrays to all zeros and character arrays to all ?'s Array bounds start at 0.

Since there are now two basic data types and two compound data types, type checking is necessary. Compile time type checking should include checking for assignment of variables of one type to variables of another (e.g. character constants to integer variables) and checking for arithmetic operations involving characters constants or variables. These are errors. Type checking should include array bounds checking as with zero division, this can occur at compile time or run time.

Reading in a character variable should be done using getchar(). Reading in an integer variable should be done using scanf("%d"). Two other semantic niceties should be implemented: block assignments and string I/O. These involve using the bare array name, other uses of the bare array name are errors.

If an assignment statement takes the form 'fred <- sam' where fred and sam are arrays of the same size and element type, do a block transfer (i.e. copy all elements in sam into fred). This avoids one of the principle uses of the for-statement and makes its omission less onerous.

If the name of a char array occurs in an I/O statement; read into the array until a newline is encountered and add a '^' at the end, or write from the array until a '^' character is encountered. This will allow us to simulate strings. I want the terminating character *explicitly* placed to reduce the chance of error and I want a 'visible' fill character. This removes the up-arrow character from the usable printable character set for this language.

For example, executing the statement read(x) where x is declared to be a 7 element character array and the input from standard input is "hello" will fill the first 5 elements array, put a '^' in the 6th and leave a '?' in the 7th. Then executing the statement write(x) will cause "hello" to be sent to standard output. Executing **write(x)** when x contains 'h', 'e', 'l', 'l', 'o', ' ', '?' is an error and should be detected when the upper bound is encountered since blanks and question marks are perfectly good printable characters. Also, executing the statement read(x) where x is declared to be a 7 element character array and the input from standard input is "hello fred" is an error since there is no room. In fact, x can only hold strings with at most 6 characters since there must be room for the '^' terminating character. But executing read(y) where y is a char variable or read(x[5]) with h in the input is fine. Note that the quotes are not really in the input or output but are just there for emphasis in this document.

Appendix D: Moe

BNF for moe

```
pgm        ::= head gbldecl rtndecl locdecl
               bodypart tail
head       ::= program NAME has EOS
gbldecl    ::= globals EOS declst | ε
rtndecl    ::= rtnlst | ε
rtnlst     ::= rtnlst rtndef | rtndef
rtndef     ::= rtnhead locdecl bodypart rtntail
rtnhead    ::= proc NAME ( pardeclst ) has EOS
               | proc NAME has EOS
rtntail    ::= endproc NAME EOS
pardeclst  ::= pardeclst ; pardecl | pardecl
pardecl    ::= int varlst
locdecl    ::= locals EOS declst | ε
declst     ::= declst decln | decln
decln      ::= int varlst EOS
varlst     ::= varlst , ID | ID
bodypart   ::= body EOS stmtlst
stmt       ::= io EOS | asgn EOS | condl EOS
               | loop EOS | procall EOS
io         ::= read ( ID ) | write ( exp )
               | writeln
asgn       ::= ID <- exp
condl      ::= if bexp then EOS stmtlst endif
               | if bexp then EOS stmtlst else EOS
               stmtlst endif
loop       ::= while bexp do EOS stmtlst endwhile
procall    ::= ID ( parlst ) | ID
parlst     ::= parlst , ID | ID
bexp       ::= exp  | exp rop exp
rop        ::= < | > | <= | >= | = | #
exp        ::= exp + term | exp - term | term
term       ::= term * factor | term / factor
               | term % factor | factor
factor     ::= ID | NUM | ( exp ) | - factor
```

Some Lexical Details

Parameters are to be passed by reference and the procedures can be called recursively. Since there is no 'forward' statement and I wouldn't require you to make two passes over the procedure declaration section, indirect recursion need not be implemented -- only direct recursion. Therefore, procedures can be called only after they are declared.

The global declaration section defines variables that are accessible to all routines including the main program. The local declaration sections define variables accessible to the body of the associated procedure or to the main program. Note that procedures are declared in the global section so local variables can be declared with the same name as a procedure.

References

[Aho] Aho, Sethi and Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1986.

[Fis] Fisher and LeBlanc, *Crafting a Compiler*, Benjamin Cummings, 1988.

[Mar] Marcotty and Ledgard, *Programming Language Landscape*, SRA, 1986.

[Lev] Levine, Mason and Brown, *lex and yacc*, O'Reilly & Associates, 1992.

******************************Undergraduate From Page 50*********************************

Werth and J. Werth (Eds.), 13th Intern. Conference on Software Engineering, Austin (TX), May 12-16, 1991

[40] Zalewski J., Cohesive Use of Commercial Tools in a Classroom. Proc. 7th SEI Conf. on Software Engineering Education, pp. 65-75, San Antonio (TX), January 5-7, 1994, J.L. Diaz-Herrera (Ed.), Springer-Verlag, Berlin, 1994

[41] Zalewski J., Boiler Water Content Controller Based on EWICS Safety Model. Proc. Intern. Invitational Workshop on the Design and Review of Software Controlled Safety-Related Systems, Ottawa, Canada, June 28-29, 1993. University of Waterloo, Institute of Risk Research, 1994

[42] Zalewski J. (Ed.), Advanced Multimicroprocessor Bus Architectures. IEEE Computer Society Press, Los Alamitos (CA), 1995