Laboratory Manual
for
Compiler Design

Robb T. Koether
### CONTENTS

3.3 The `Err` and `Warning` Classes .................................. 33
3.4 The `Token` Class .................................................. 34
3.5 Invoking the Lexer .................................................. 34
3.6 Building the Lexical Analyzer .................................... 34
3.7 The Makefile ...................................................... 35
3.8 Assignment ......................................................... 36

III Syntactic Analysis .................................................. 39

4 A Recursive-Descent Parser ........................................... 41
   4.1 Introduction .................................................... 41
   4.2 Modify the Lexical Analyzer .................................... 42
   4.3 The Recursive-Descent Parser .................................. 42
   4.4 Derivations and Parse Trees .................................... 44
   4.5 Improving the Parser ........................................... 45
   4.6 Assignment ...................................................... 45

5 Using JLex with a Predictive Parser ............................... 47
   5.1 Introduction .................................................... 47
   5.2 Lexer-Parser Interface .......................................... 48
   5.3 The Productions ................................................ 49
   5.4 The Parse Table ................................................ 50
   5.5 The Parsing Algorithm ......................................... 51
   5.6 Running the Program ........................................... 52
   5.7 Expand the Grammar ............................................ 53
   5.8 Assignment ...................................................... 53

6 Using JLex and CUP .................................................. 55
   6.1 Introduction .................................................... 56
   6.2 Modifying the JLex File ........................................ 56
   6.3 The `Symbol` Class ............................................. 57
   6.4 The CUP File ................................................... 58
   6.5 The Grammar ................................................... 58
   6.6 Shift/Reduce Conflicts ......................................... 60
   6.7 Semantic Actions ................................................ 61
CONTENTS

6.8 The sym and parser Classes ...................................... 61
6.9 The Action and Goto Tables ...................................... 62
6.10 The CUP Debugger ............................................... 63
6.11 Assignment ....................................................... 63

IV A Simple Compiler .................................................. 69

7 The Symbol Table ..................................................... 71
  7.1 Introduction ....................................................... 72
  7.2 The Symbol Table ................................................ 72
  7.3 The IdEntry Class ............................................... 73
  7.4 The Hashtable and LinkedList Classes ......................... 73
  7.5 The SymbolTable Class ........................................... 74
  7.6 Reserved Words and the Lexer .................................... 77
  7.7 The Parser ........................................................ 77
  7.8 The Ops.java File ............................................... 79
  7.9 Running the Program ............................................. 79
  7.10 Semantic Actions ................................................ 81
  7.11 Handling Syntax Errors ........................................ 83
  7.12 Assignment ....................................................... 84

8 The Abstract Syntax Tree ............................................. 85
  8.1 Introduction ....................................................... 86
  8.2 Pointers, l-values, and r-values ................................ 87
  8.3 Tree Nodes and the TreeNode Class .............................. 87
  8.4 The Ops Class .................................................... 89
  8.5 The TreeOps Class ............................................... 90
  8.6 The Utility Class ................................................ 90
  8.7 Printing the Abstract Syntax Tree ............................... 91
  8.8 Semantic Actions ................................................ 92
  8.9 The NUM Case .................................................... 93
  8.10 The id() Function ................................................. 94
  8.11 The dcl() Function ............................................... 95
  8.12 The deref() Function ............................................ 96
 CONTENTS

8.13 Parenthesized Expressions ................................. 97
8.14 The assign() Function .................................. 97
8.15 The arith() Function .................................... 98
8.16 Assignment .................................................. 99

9 Code Generation ................................................. 103
9.1 Introduction .................................................. 104
9.2 The compiler_v1 Class ...................................... 104
9.3 The Code Generator ......................................... 105
9.4 Allocation .................................................... 106
9.5 Identifier Nodes .............................................. 107
9.6 The Dereference Operation .................................. 107
9.7 Testing the Compiler ........................................ 108
9.8 Numbers ....................................................... 109
9.9 The Assignment Operator ................................... 110
9.10 The Addition Instruction ................................... 111
9.11 The Subtraction Instruction ............................... 112
9.12 The Negation Instruction .................................. 112
9.13 The Multiplication Instruction ............................ 113
9.14 The Division Instruction ................................... 113
9.15 The Mod Operator .......................................... 114
9.16 print and read Statements ................................. 114
9.17 Assignment .................................................. 116

V Additional Data Types ........................................... 117

10 Floating-Point Numbers and the AST ......................... 119
10.1 Introduction .................................................. 119
10.2 Version 2 of the Compiler ................................ 120
10.3 Introducing the double Keyword .......................... 120
10.4 Semantic Actions ............................................ 121
10.5 The dcl() Function ......................................... 121
10.6 The cast() Function ........................................ 122
10.7 The arith() Function ....................................... 123
## CONTENTS

10.8 The `assign()` Function ........................................ 123
10.9 The `print()` and `read()` Functions ...................... 123
10.10 The `mod()` Function ............................................. 125
10.11 `double` Literals .................................................. 125
10.12 Testing the Compiler ............................................. 125
10.13 Assignment ....................................................... 126

11 Floating-Point Numbers and the FPU .......................... 127
11.1 Introduction ....................................................... 127
11.2 The `ID` and `NUM` Cases ....................................... 128
11.3 The `DEREF` Case .................................................. 128
11.4 The `ASSIGN` Case ................................................. 129
11.5 The Arithmetic Operators ....................................... 129
11.6 The `CAST` Case .................................................... 130
11.7 The `PRINT` and `READ` Cases ................................. 131
11.8 Testing the Compiler ............................................. 131
11.9 Assignment ....................................................... 131

VI Functions .................................................................. 133

12 Function Definitions and the AST ......................... 135
12.1 Introduction ....................................................... 136
12.2 Miscellaneous Details .......................................... 139
12.3 The CUP file ....................................................... 140
12.4 The `fname()` Function .......................................... 141
12.5 The `arg()` Function ............................................ 142
12.6 The `dc1()` Function ............................................ 143
12.7 The `fbeg()` Function ............................................ 143
12.8 The `ret()` Function ............................................ 143
12.9 The `func()` Function ........................................... 144
12.10 Debugging and Testing the Compiler ...................... 144
12.11 Assignment ....................................................... 144
15.10 Backpatch Functions .................................. 168
15.11 The CUP File ........................................... 170
15.12 Sequences of Statements ............................. 173
15.13 Conditional Expressions ............................... 173
15.14 The exprToCExpr() Function ......................... 175
15.15 The One-Way if Statement ........................... 176
15.16 The Two-Way if Statement ............................ 177
15.17 Function Ends ........................................... 177
15.18 The READ Case .......................................... 178
15.19 Testing the Compiler ................................... 178
15.20 Assignment .............................................. 179

16 Control Flow Code Generation .......................... 181
16.1 Introduction ............................................. 182
16.2 The LABEL Case ......................................... 182
16.3 The EQU Case ............................................ 182
16.4 The JUMP Case ........................................... 183
16.5 The CMPNE Case ........................................ 184
16.6 The JUMPT Case ......................................... 185
16.7 Testing the Compiler .................................... 186
16.8 Assignment .............................................. 186

17 Boolean Expressions .................................... 187
17.1 Introduction .............................................. 187
17.2 Version 5 ................................................. 188
17.3 The Relational Operators .............................. 188
17.4 The Boolean Operators .................................. 189
17.5 Testing the Tree-Building ............................. 190
17.6 Code Generation ........................................ 191
17.7 Testing the Code-Generation .......................... 191
17.8 Assignment .............................................. 191
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>The symbol table</td>
<td>75</td>
</tr>
<tr>
<td>8.1</td>
<td>An example of an abstract syntax tree</td>
<td>86</td>
</tr>
<tr>
<td>8.2</td>
<td>A number tree</td>
<td>93</td>
</tr>
<tr>
<td>8.3</td>
<td>A declaration tree</td>
<td>95</td>
</tr>
<tr>
<td>8.4</td>
<td>A dereference tree</td>
<td>96</td>
</tr>
<tr>
<td>8.5</td>
<td>An assignment tree</td>
<td>98</td>
</tr>
<tr>
<td>8.6</td>
<td>An arithmetic tree</td>
<td>98</td>
</tr>
<tr>
<td>8.7</td>
<td>A mod tree</td>
<td>99</td>
</tr>
<tr>
<td>8.8</td>
<td>A negate tree</td>
<td>100</td>
</tr>
<tr>
<td>8.9</td>
<td>Print and read trees</td>
<td>100</td>
</tr>
<tr>
<td>10.1</td>
<td>An uncast tree node</td>
<td>122</td>
</tr>
<tr>
<td>10.2</td>
<td>A cast tree node</td>
<td>122</td>
</tr>
<tr>
<td>10.3</td>
<td>Casting an arithmetic operation</td>
<td>124</td>
</tr>
<tr>
<td>10.4</td>
<td>Casting an assignment</td>
<td>124</td>
</tr>
<tr>
<td>12.1</td>
<td>A tree for a function beginning</td>
<td>137</td>
</tr>
<tr>
<td>12.2</td>
<td>A function return tree</td>
<td>137</td>
</tr>
<tr>
<td>12.3</td>
<td>A tree for a function ending</td>
<td>138</td>
</tr>
<tr>
<td>13.1</td>
<td>The tree for the copy function</td>
<td>146</td>
</tr>
<tr>
<td>13.2</td>
<td>The tree for the sum function</td>
<td>148</td>
</tr>
<tr>
<td>13.3</td>
<td>The tree for a list of two parameters</td>
<td>149</td>
</tr>
<tr>
<td>13.4</td>
<td>The tree for a function call</td>
<td>150</td>
</tr>
<tr>
<td>14.1</td>
<td>The logical structure of a function tree</td>
<td>155</td>
</tr>
</tbody>
</table>
15.1 Equate trees ......................................................... 170
15.2 Backpatching statements in sequence .......................... 173
15.3 A comparison tree (not equal) ................................. 174
15.4 A comparison to zero .............................................. 174
15.5 Backpatching a one-way \textit{if} statement ....................... 176
15.6 Backpatching a two-way \textit{if} statement ....................... 177
15.7 Backpatching the end of a function ........................... 177

17.1 A tree comparing two expressions ......................... 188
17.2 Backpatching the AND operator ............................. 190
Part I

Preliminaries
Laboratory 1

Getting Started

Key Concepts

• The Cygwin window

• Environment variables

• The Java compiler

• The JLex lexical analyzer generator

• The CUP parser generator

• The WinZip program

Before the Lab

Read Chapter 1 of Compilers: Principles, Techniques, and Tools.

Preliminary

In your folder in //hams-acad-fs/Students, create a folder named Coms 480. Keep all of your work for this course in this folder.

Copy the folder Lab_01 from the Compiler Design CD to your folder.
1.1 Introduction

In this lab we will download and install a number of programs. The purpose of this is twofold. You will be more aware of the setup that we will be using and you will be able to set up the same software on your own computer.

1.2 The Cygwin Window

Throughout this course we will use the Cygwin command window. You may use the DOS command window if you want, but I think it would be better if you gained experience with a UNIX-type system. An exception to this will be editing source text. The UNIX editors are truly awful. Even though there is some benefit in learning how to use them, I recommend that you use CodeWarrior to edit your Java source files in this course.

Cygwin creates a UNIX-type environment for Windows. A large number of the standard UNIX commands are available.

Cygwin has already been installed on your computer. You will see the Cygwin icon on the desktop, so we will not download it now.

However, to download Cygwin, go to the web site

http://www.cygwin.com/

If you download Cygwin in your room, you should go to this web page and click on the install icon. The program setup.exe will be downloaded. When you run the setup program, one of the familiar installer programs will start up, asking you several questions. Generally, you should go with the defaults. However, when you choose which packages to install, be sure to select “install” for the development (Devel) package. You will get a minimal install plus the development tools. The program will go through three stages: downloading, installing, and executing. These stages should take roughly 25 minutes, 10 minutes, and 1 minute, respectively. If the installation fails, then try again.

After Cygwin is installed, double-click on the Cygwin icon on the desktop to start the Cygwin window. Then right-click on the title bar and select Properties. You may change the font, the size of the window, and the colors. My preference is to choose a small font (12 pt) and then make the window as wide and as tall as possible. Type the command pwd (print working directory) to see the pathname of the current directory.
Type the command `dir` (directory) to see the names of all the files in the current directory.

Choose the name of one of the subdirectories in the current directory and type the command

```
$ cd directory-name
```

where `directory-name` is the name of the subdirectory that you chose. Now repeat the commands `pwd` and `dir`. Type the command

```
$ cd ..
```

to return to the previous directory. A single period (`.`) refers to the current directory and two periods (`..`) refers to the parent directory. For example, to move up two levels and then down to a directory called `programs`, you could type

```
$ cd ../../programs
```

### 1.3 The HOME Environment Variable

The Cygwin window has opened to a default directory. Use `pwd` to see what it is. You will find it very convenient to set the HOME environment variable to your Coms 480 folder. Then Cygwin will always start there when you open a Cygwin window and you can always return there by typing

```
$ cd ~
```

To make this the home directory, bring up the System control panel. Click on the **Advanced** tab and then click on **Environment Variables**. In the top section, named **User Variables**, click **New**. Enter `HOME` as the **Variable name** and type the exact pathname of your Coms 480 directory as the **Variable value**. If you open a window to that directory, then you should be able to copy and paste the pathname. Be sure to use the backslash as a separator between directories. Then click **OK** (on all three windows) to save the settings.

Now close the Cygwin window and open a new one. (This is necessary in order to reinitialize the environment variables.) This window should have opened to your folder. You can confirm that by typing `pwd`. From now on, Cygwin will begin in this folder.
1.4 The Java Development Kit

The latest version of the Java Development Kit (JDK) is update 1.6, release 11. Go to the website

http://java.sun.com/javase/downloads/index.jsp

Next to the title “Java SE Development Kit (JDK) 6 Update 11,” click on Download. Follow the instructions through the next two web pages. (Do not download the Sun Download Manager.) When you are finished there should be a Java folder in the Program Files folder of the C drive. Inside one of the subfolders is the javac.exe compiler.

1.5 Running Java Programs

Change the current directory to Lab_01 if you are not already there.

I have written a Java program that prints "Hello, World!" to standard output. It is in the Lab_01 folder that you downloaded. Type the command

$ javac Hello.java

You probably will get an error message, because the computer could not find the Java compiler javac.exe.

Open the System program on the Control Panel and go the Environment Variables window again. This time we must add a PATH variable. The PATH variable tells the computer where to find executable files, including the Java compiler.

You may have to search for the Java compiler. Indeed, there may be more than one on the computer. We will use the one in the folder named C:\Program Files\Java\jdk1.6.0_11\bin. Once you know where it is, then create the PATH variable with this pathname as its value, as you did the HOME variable earlier.

Close the Cygwin window and open a new one. Now try again to compile the program. This time the program should compile. Type dir to see that the file Hello.class is in the directory. This is the compiled Java program. Now run the program by typing

$ java Hello

The program should print "Hello, world!" This confirms that you are set up to run Java programs in the Cygwin window.
1.6 The Java API

Our programs will be written Java. If you know C++, then you should have no
trouble picking up Java since it is quite similar. The two languages use mostly
the same keywords, same constructs, and the same syntax. One major difference,
however, is that Java is heavily object-oriented. Every function must be a member
function of some class. Another difference is that Java comes with an extensive library
of classes.

Open Internet Explorer and go to the website

http://java.sun.com/j2se/1.4.2/docs/api/

Save this as a bookmark. You will want to return here many times later in the course.

This web site contains the documentation for all Java classes. For example, in
the upper left frame, scroll down and click on java.lang. In the window below, the
names of all the classes in the java.lang package appear. Click on Integer. To the
right you see the information about the Integer class.

• In what ways can an Integer be constructed?

• How does one convert an Integer to a double?

Another difference between Java and C++ is that Java is weak on operators. Oper-
ators are defined only for the primitive objects: int, float, double, char, etc.

• How do you compare one Integer to another?

• How do you add two Integers and represent the result as an Integer?

Remember, there are no operators + or < for the Integer class!

1.7 The JLex Java-based Lexical Analyzer Generator

Be sure you are now in the Coms 480 folder (not Lab_01).

Open another instance of Internet Explorer and go to the web site

This is the web site for JLex, the Java lexical analyzer generator. Click on Installation Instructions. Read these instructions carefully. The directory “J” will be your Coms 480 folder. Create a subfolder named JLex. You can either do this in Windows or you can type the command

$ mkdir JLex

When you are ready, click on Source Code and save it in your JLex folder. The Java code for JLex (Main.java) should now be in the folder JLex. Move to the JLex directory and compile JLex by typing

$ javac Main.java

Now we will test our installation by using a test program available from the JLex web page. On the web page, click on Sample Input. Copy and paste the contents of the page into a new file in CodeWarrior and save it in Lab_01 as sample.lex.

This file must be edited slightly in order to work on our PCs. In DOS files, each line ends with a return character \r followed by a newline character \n. In UNIX files, each line ends with only a newline character. Go to line 116 in sample.lex:

<YYINITIAL,COMMENT> \n { }

Immediately below this line, add a similar line, replacing \n with \r.

Now type

$ java JLex.Main sample.lex

It probably did not work. The notation JLex.Main means the Main.class file in the JLex folder. The program javac could not find the file Main.java. To solve this problem, we must define the CLASSPATH variable. The CLASSPATH variable tells the Java compiler where to look for Java source code files. Set this variable as before, setting the value of CLASSPATH to

.;\hams-acad-fs\Students\your-name\Coms 480

and restart the Cygwin window. Be sure to replace your-name with the name of your workspace in the Students directory. Note the initial dot (.). This refers to the current directory. Thus, Cygwin will search the current directory first. The semicolon is a separator. Cygwin will next search
Now type

$ java JLex.Main sample.lex

again. It should work. This creates the file sample.lex.java, which contains Java source code for the classes Sample, Utility, Yytoken, and Yylex. Then type

$ javac sample.lex.java

to compile this, creating the files Sample.class, Utility.class, Yytoken.class, and Yylex.class. You will get an error message (7 of them) about the assert keyword. This program creates an assert() function, but the latest versions of Java use assert as a keyword. Go back to the file and change assert to myAssert. Then recreate the file sample.lex.java and compile it with javac.

Now we can test our program. Type

$ java Sample

Enter various lines of C code and see what the output is. Enter a line that contains an embedded /*-style comment. When you are satisfied, type CTRL-Z (as many times as necessary) to indicate end of file. The program should terminate.

1.8 The CUP Java-based Parser Generator

Open one more instance of Internet Explorer and go to the web site

http://www2.cs.tum.edu/projects/cup/

This is the web site for CUP, the Java parser generator. CUP stands for Constructor of Useful Parsers. It also is a play on the java theme. Get it? CUP? Java? Cup of java?

We will save the CUP files in a CUP directory. Create a CUP directory now as a subdirectory of your Coms 480 directory. On the web page, in the section “Archived versions,” click on CUP 10k sourcecode release. This is the most recent stable version. Then click on Save. Save it in your CUP folder. This will download a zip file to be unzipped. All the files needed for CUP will be extracted. Double-click on
the file `java_cup_v10k.tar.gz`. Follow the Zip instructions. You should save the extracted files in your CUP folder.

The Java classes in CUP are already compiled. We will now test CUP using a slightly modified version of the sample program that appears in the CUP User’s Manual. This example creates a program that evaluates integer expressions involving `+`, `-`, `*`, `/`, `%`, and parentheses.

In the Lab_01 folder, there are the files `scanner.java`, `grammar.cup`, and `Evaluator.java`. Type the command

```
$ java java_cup.Main < grammar.cup
```

Again, this probably did not work. The filename `java_cup.Main` refers to the file `Main.class` in the subdirectory `java_cup` of the CUP directory. Java does not know to look in the CUP directory for Java class files. Therefore, we must add the pathname of the CUP directory to the paths to be searched in the `CLASSPATH` environment variable. Make this change, close the Cygwin window, and open a new one.

Try again to execute the command

```
$ java java_cup.Main < grammar.cup
```

It should work. This will create the Java source files `parser.java` and `sym.java` from the grammar file `grammar.cup`. Compile these two files and then compile the files `scanner.java` and `Evaluator.java`.

Run the evaluator program by typing

```
$ java Evaluator
```

The program accepts keyboard input. Type in an integer expression such as

```
2 + 3 * 4;
```

Be sure to end the expression with a semicolon. The program will print the value of the expression. When you are finished, type CTRL-Z.
1.9 Zipping Files

Most of your assignment will involve a large number of source files. Rather than drag each one individually to the dropbox, you will place your work in a folder, zip the folder into a zip file, and drop the zip file in the dropbox. I will unzip it and test it.

Let’s test WinZip. We will zip the files used in the last two examples, namely, the files Sample.class, Utility.class, Yylex.class, Yytoken.class, scanner.java, parser.java, sym.java, and Evaluator.java, save the file as Lab_01.zip, and then unzip them into a test folder. To do this, start up WinZip and follow the instructions. Use the Wizard version of WinZip. Select Create a new Zip file. Give it the name Lab_01. Add the specified files by repeatedly clicking Add files... and selecting the files. Have the output directed to your Coms 480 folder. Then click Zip Now and exit WinZip. Next create a folder named Test in which to put the extracted files. Double-click on the zip file and follow the instructions to extract the files. Direct the output to folder Test. Open Test to verify that the original files are there.

Now test the results by recompiling the files (be sure to change the directory to Test in Cygwin) and running sample and Evaluator again.

1.10 Assignment

Turn in the file Lab_01.zip.
Part II

Lexical Analysis
Laboratory 2

Writing a Lexical Analyzer

Key Concepts

• Lexical Analyzers

• Redirected input and output

• Makefiles

Before the Lab


Preliminary

Copy the folder Lab_02 from the Compiler Design CD to your Coms 480 folder.

2.1 Introduction

In this lab we will create a lexical analyzer that will return the tokens that occur in statements of the form

\[ \text{id } = \text{expr}; \]

where \text{expr} is an infix expression consisting of integers, identifiers, parentheses, and the operators +, -, *, and /. Once we understand how this is done, we will be able to create a lexer for a larger set of C tokens.
2.2 Makefiles

Open the file Makefile. A makefile consists mainly of a list of dependencies and actions. A dependency is written in the form

\[ \text{target: sources} \]
\[ \text{action} \]

where target is a file name and sources is a list of file names. (The tab before the action part is mandatory.) This means that the target file depends on the source files. Whenever any of the source files is updated, then the target file will be updated by performing the action. This is all governed by the timestamps on the files.

The makefile for Lab 2 contains the dependencies among the files used by this program. In this case, it is very simple: there are only four files and two dependencies. The file MyLexer.class depends on the file MyLexer.java and the file Token.class depends (in the same way) on the file Token.java. That means that whenever MyLexer.java is updated, i.e., modified, then MyLexer.class should also be updated and whenever Token.java is updated, then Token.class should also be updated. In the makefile, the line

\[ \text{MyLexer.class: MyLexer.java} \]

expresses that dependency. The line below that,

\[ \text{javac MyLexer.java} \]

describes the action to be taken whenever the timestamp of MyLexer.java is more recent than the timestamp of MyLexer.class. Note that this line begins with a mandatory tab. A similar pair of lines appears for Token.class and Token.java.

As our programs become more and more complicated, you will come to appreciate the makefiles more and more. In Lab 3 the makefile will be more sophisticated.

To invoke the makefile, type the command

\[ $ \text{make} \]

Try this now. You should see that the Java compiler is invoked and MyLexer and Token are compiled. Type the command again and you will see that it says that MyLexer is up to date, so it does not recompile it.

Now we will delete the file MyLexer.class and rebuild it. Type the command
$ rm MyLexer.class

This removes the file. Now execute the make command.

Let’s do it one more time. This time we will not remove MyLexer.class, but merely change the timestamp of MyLexer.java. The touch command will change the timestamp of a file to the current time. Type

$ touch MyLexer.java

and then type the make command again.

2.3 Running the Lexer

Open the file MyLexer.java in CodeWarrior. This is a Java program that finds certain tokens in the input stream. Currently if finds only positive integers, plus signs (+), and times signs (*).

Look in the Lab_02 folder and see that there is now a file named MyLexer.class. This is the compiled bytecode version of MyLexer.java.

Now we will run MyLexer. Type

$ java MyLexer

The program expects input from the keyboard, so type

$ 123 + 456 * 789

The program should find the five tokens in this expression. Then type CTRL-Z and press return again. CTRL-Z is interpreted as end-of-file.

This program does not recognize any grammar rules; those will come later. Therefore, any string of legal tokens will be processed correctly. For example, try

$ 123***456++++

It also skips white space, so it will accept the input

$ 1 23 * * * 45 6 +++ +

We wish to expand the lexer so that it will recognize input such as

$ avg_grade = (3*test + 2*exam)/5;

Run MyLexer again, using this input to see what the output is.
2.4 Understanding the Lexer

Before improving the program, let’s take a look at how it works. First we will look at the tokens. Open the file Token.java. The purpose of the Token class is to provide a list of symbolic constants to be used by the lexer. The Token class also provides a set of strings so that we can print the name of the token in an readable form.

Now look in the file MyLexer.java. The program creates a BufferedReader object named source. Look at the function getNextChar(). It gets an integer iVal from source and then converts it to a character cVal.

Look at the function advance(). Once a character has been processed, we place it in a character buffer and read another character. The purpose of the character buffer is to store the “value” of the token as a character string for use in the cases when the token is an identifier or a number. For example, if the token is radius, then the type is ID and the value is "radius", and if the token is 123, then the type is NUM and the value is "123".

To clear the buffer, we simply set charCnt to 0.

The main function initializes the lexer and then processes tokens by repeatedly calling next_token() until it returns the EOF token. Note the use of the expression

\[
\text{new String(buffer, 0, charCnt)}
\]

to convert the contents of the buffer to a String. Go to the Java web page, access the String page, and look at the String constructors. The URL is

http://java.sun.com/j2se/1.4.2/docs/api/

Find the constructor that is used here. You should develop the habit of referring to the Java API pages as often as necessary, i.e., often. You will find the answers to many of your Java questions there.

The heart of the MyLexer class is the next_token() function. By looking at the current character cVal, next_token() is able to decide which type of token is being read. It processes the token and returns the token type.

2.5 Assignment

For the “toy” lexer that we are building in this lab, the complete set of tokens is shown in Table 2.1.
Complete the `MyLexer` class by adding code that will recognize each of these types of token.

After you have finished the lexer, test it on the file `testfile`. The lexer uses standard input (keyboard) and standard output (monitor), but you may redirect them to files. To read input from the file `testfile`, type

$ java MyLexer < testfile

To redirect output to a file named, say, `outfile.txt`, type

$ java MyLexer >outfile.txt

Or you can do both at the same time.

$ java MyLexer < testfile > outfile.txt

Type this command and then open `outfile.txt` in CodeWarrior or Notepad and inspect it. When the output is complicated, this method allows you to inspect it at your leisure. Or you can print it and inspect it later.

Zip the files `MyLexer.java`, `Token.java`, and `Makefile` in a folder named `Lab_02` and drop it in the dropbox.
Laboratory 3

A Lexical Analyzer using JLex

Key Concepts
- Lexical analyzer generators
- JLex
- The JLex interface
- Regular expressions

Before the Lab

Preliminary
Copy the folder Lab_03 from the Compiler Design CD to your folder.

3.1 Introduction
In this lab we will create the same lexical analyzer as in Lab 2, but by using JLex and a file tokens.lex. The file tokens.lex contains the rules for recognizing tokens and the actions to take for each kind of token. JLex will use these rules to build a Java program that will be a lexical analyzer. The rules in the file tokens.lex are regular expressions and the lexical analyzer that JLex builds is a DFA. By inspecting
the Java code that JLex produces, one could in principle figure out how it works, although it is rather complicated.

3.2 JLex

JLex is a program that generates Java source code for a lexer. The original such program was lex, which works on UNIX systems and creates C source code. The gnu version of lex is called flex and it is included as one of the development tools in cygwin. It also produces C source code.

Open the file tokens.lex in CodeWarrior. A JLex file is divided into three parts, which are separated by `%%`. The first part contains code that is to be copied directly into the Java class Yylex, which is the name of the lexer. In this example, this section contains no code. The `yy` prefix is a carry-over from lex, which uses the `yy` prefix on all of its variables and file names in order to avoid conflicts with any programmer defined objects (provided the programmer avoids the `yy` prefix). See Section 2.1 of the JLex User’s Manual for more information.

The second part contains various JLex directives and definitions. Read the JLex User’s Manual for more information. This example contains one directive and three definitions. The first directive is

```
%integer
```

It tells JLex that the tokens that are returned will be of type `int`. It also causes JLex to define the constant `YYEOF = -1` in the `Yylex` class. This constant will be returned automatically by `Yylex` upon encountering end of file and then will be used in the main function to terminate the program. See Section 2.2 of the JLex User’s Manual for more information.

The three definitions

```
digit= [0-9]
num= {digit}+
ws= [\ \t]+
```

define a number and white space using regular expressions. A digit is a single character in the range `’0’` through `’9’`. A number is one or more digits. White space (`ws`) is a blank or a tab. See Section 2.3 of the JLex User’s Manual for more information.
The third section lists the regular expressions to be matched and the actions to be taken for each. In the example, the regular expressions are numbers `num`, a plus sign `"+"`, and a times sign `"*"`. In each case, the action is to print a message telling the type of token found and to return the corresponding constant from the `Token` class (to be discussed shortly). In the case of a number, the message will also include the string that constitutes the value of the token.

The double quotes around single characters are used to guard against the character being interpreted as a metacharacter.

Since white space should be ignored, there is no action associated with it.

Finally, note the dot (\.). This stands for any character except a newline. It should be used and it should appear last. If a string matches more than one pattern, JLex will choose the pattern that is listed earlier in the list. Thus, no pattern in the list after the dot, except a newline, will be matched. In this example, the dot is used to match any invalid character. Notice that the error message is handled by the `Err` class.

### 3.3 The `Err` and `Warning` Classes

Open the file `Err.java`. This class is designed to handle error messages. For each error, create a symbolic name and a message. For example, the “Illegal character” error has the symbolic name

```java
public static int ILLCHAR = 1;
```

and the message

"Illegal character: 

The illegal character itself is filled by passing the parameter `yytext()`. Notice also that the line number in which the error was detected is printed. For example, if a program contained the illegal character `#` in line 19, then the message

```
Error: (line 19) Illegal character: #
```

The file `Warning.java` has the same design as `Err.java`. The difference between errors and warnings is that errors are fatal, i.e., the program cannot be compiled. If there are only warnings, then the program can be compiled, but it may not run as expected. As we add to our compiler, whenever we want to report an error or a warning, use the `Err` and `Warning` classes, adding new messages as necessary.
3.4 The Token Class

Open the file Token.java. The Token class contains symbolic names for the int values returned by the lexer. These make the program more readable. In a later lab, this class will be replaced by the sym class, which is generated by the CUP parser generator. This code will be included in the output file from JLex and then it will be compiled into the file Token.class. This file defines four constants: ERROR, NUM, PLUS, and TIMES.

3.5 Invoking the Lexer

Open the file Lexer.java. The main function is defined in the Lexer class. This function simply processes tokens until reaching end of file. End of file is indicated by the token YYEOF.

Observe how a lexer is created:

```
lex = new Yylex(System.in);
```

Then observe how tokens are obtained from the lexer:

```
token = lex.yylex();
```

The variable token is of type int because we told our lexer to return ints. You should appreciate how simple the interface with the Lexer class is; all the work is being done in the Yylex class.

3.6 Building the Lexical Analyzer

To use JLex to create the lexical analyzer, we must invoke JLex’s Main function and specify the file tokens.lex as the input file.

```
$ java JLex.Main tokens.lex
```

This will create as output the file tokens.lex.java. Since we are creating the Yylex class, we will rename this file Yylex.java. Then we compile Token.java, Yylex.java and Lexer.java in the usual way.

Execute the above command now. Note the screen output from JLex. It describes the number of states in the NFA that it builds. Then it converts the NFA to a DFA. Finally, it minimizes the DFA and reports the number of states.
Open the file Yylex.java in CodeWarrior. Look at the Yylex class. It is complicated, but not impossible to figure out. Go to the function yytext(). It builds a string that contains the value of the current token. Note that it uses the same String constructor that we used in our program MyLexer. Look in the file tokens.lex and find the place where yytext() is used. Be aware that yytext() is not a public member function of the Yylex class. The reason we can use it in the JLex file is because this code is copied into the Yylex class. Therefore, when it is used, it is being used within the class.

Now look at the function yylength(). This function returns the length of the current token.

The function unpackFromString() is used to create the DFA transition tables that drive the lexer. It is invoked three times just before the yylex() function (not to be confused with the Yylex constructors).

In the yylex() function you will find the code that appeared in the tokens.lex file as the action part of each token. It appears now in a switch structure.

Now compile the files Token.java, Yylex.java, and Lexer.java. Then test the lexer by typing

```
$ java Lexer
```

The program will wait for input from the keyboard. Enter strings as you did in Lab 2. For example, try the strings

```
123 + 456 * 789
1 23 * * * 45 6 +++ +
avg_grade = (3*test + 2*exam)/5;
```

In the last example, did you get any error messages? Why?

### 3.7 The Makefile

Open the file Makefile. The makefile is now more complicated, and therefore more beneficial to us. This makefile begins with a rule. In the rule, the % is a wildcard, signifying any file name. Thus, the rule says that each .class file depends on the corresponding .java file. In the action part, the $< refers to whatever file name the % currently holds. In this example, the action part will invoke the java compiler.
By writing the rule, the makefile will automatically update any .class files by this rule, as necessary. Thus we do not need to tell it explicitly to make Lexer.class, Token.class, and Yylex.class.

In the next part of the makefile, there is the line

\texttt{all: Token.class Yylex.class Lexer.class}

This says that the “file” all depends on the other three files. In fact, there is no file all. (Notice that there is no action part telling how to update all.) This is a common makefile technique used to force the makefile to update a list of targets.

The final part contains the two instructions that are needed to update the Yylex class whenever a change is made to tokens.lex.

Read through the makefile and be sure that you understand it. For more information on makefiles, there are many good tutorials on the web. For example, see

\texttt{http://www.gnu.org/software/make/manual/make.html}

You should spend some time looking over one of them.

\section*{3.8 Assignment}

Add to the lexical analyzer the set of tokens shown in Table 3.1.

Your lexer must also detect the patterns described in Table 3.2, although they require no action and no return value. These are the patterns that the compiler will skip over.

Take actions similar to those already taken in the file tokens.lex. In the case of identifiers, be sure to print the value of the identifier as well as the token type. Test your work thoroughly. The /*-style comments extend only to the end of the line. The //*-style comments extend past line breaks to the next occurrence of */. Character strings do not extend past line breaks.

This lab will serve as Project 1. Zip the files tokens.lex, Token.java, Lexer.java, and Makefile in a folder named Project_1 and drop it in the dropbox.
<table>
<thead>
<tr>
<th>Token</th>
<th>Symbolic Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>ID</td>
<td>a letter followed by zero or more letters, digits, and underscores</td>
</tr>
<tr>
<td>number</td>
<td>NUM</td>
<td>one or more digits</td>
</tr>
<tr>
<td>string</td>
<td>STR</td>
<td>A double quote (&quot;), followed by non-double-quote</td>
</tr>
<tr>
<td>(</td>
<td>LPAREN</td>
<td>characters, followed by a double quote</td>
</tr>
<tr>
<td>)</td>
<td>RPAREN</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>LBRACE</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>RBRACE</td>
<td></td>
</tr>
<tr>
<td>[</td>
<td>LBRACK</td>
<td></td>
</tr>
<tr>
<td>]</td>
<td>RBRACK</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>PLUS</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>MINUS</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>TIMES</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>DIVIDE</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>MOD</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>INC</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>DEC</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>EQ</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>NE</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>LT</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>LE</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>GT</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>GE</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>COMP</td>
<td></td>
</tr>
<tr>
<td>Token</td>
<td>Symbolic Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>&amp;</td>
<td>BAND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>BXOR</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>SHL</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>SHR</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>ASSIGN</td>
<td></td>
</tr>
<tr>
<td>+=</td>
<td>APLUS</td>
<td></td>
</tr>
<tr>
<td>-=</td>
<td>AMINUS</td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td>ATIMES</td>
<td></td>
</tr>
<tr>
<td>/=</td>
<td>ADIVIDE</td>
<td></td>
</tr>
<tr>
<td>%=</td>
<td>AMOD</td>
<td></td>
</tr>
<tr>
<td>&amp;=</td>
<td>ABAND</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>ABOR</td>
</tr>
<tr>
<td>^=</td>
<td>ABXOR</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>ASHL</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>ASHR</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>COMMA</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>SEMI</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Tokens used in Project 3

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>//-comment</td>
<td>//, followed by zero or more characters, followed by end of line</td>
</tr>
<tr>
<td>/*-comment</td>
<td>/*, followed by zero or more characters, followed by */</td>
</tr>
<tr>
<td>white space</td>
<td>a blank or a tab</td>
</tr>
<tr>
<td>newline</td>
<td>\n</td>
</tr>
<tr>
<td>return</td>
<td>\r</td>
</tr>
</tbody>
</table>

Table 3.2: Patterns to be skipped over
Part III

Syntactic Analysis
Laboratory 4

A Recursive-Descent Parser

Key Concepts

- Recursive-descent parsers
- Leftmost derivations
- Parse trees

Before the Lab


Preliminary

Copy the folder Lab_04 from the Compiler Design CD to your folder. Also, copy the file MyLexer.java and Token.java from your Lab_02 folder to this folder. These two files should have been modified to accept identifiers, subtraction, division, assignments, parentheses, and semicolons.

4.1 Introduction

We will create a recursive descent parser that parses a series of statements of the form

\[
\text{id} = \text{expr};
\]
where \( \text{id} \) is an identifier and \( \text{expr} \) consists of numbers, variables (identifiers), +, -, *, /, and parentheses ( and ). The parser will print each token and each grammar rule as it is used. From this we may infer a derivation of the input string.

### 4.2 Modify the Lexical Analyzer

In this lab, the lexer will not be the main program. Instead, the parser will call on the lexer whenever it needs the next token. Thus, remove the \text{main()} function from \text{MyLexer}; the parser will call \text{MyLexer.next token()} directly. One purpose of \text{main()} was to print the tokens as they were identified. This has now been moved to a function \text{printToken()} that is in the parser class.

### 4.3 The Recursive-Descent Parser

Open the program \text{RDParse}.\text{java} in CodeWarrior. This program is a direct implementation of the grammar rules

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' | \varepsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' | \varepsilon \\
F & \rightarrow \text{num}
\end{align*}
\]

Note that when there is a choice of rules to apply, the decision can be made by looking at the current token only. If the current token does not fit any of the production rules, then an error message is printed. This is a necessary characteristic of the grammar in order for a recursive-descent parser to be feasible.

Notice that for each nonterminal in the grammar, there is a function in \text{RDParse}.\text{java} that decides which rule to apply to that nonterminal. The main function gets things started by initializing the lexer (\text{MyLexer}), getting the first token, and calling on \text{E().} When execution eventually returns, the class variable \text{error} tells whether an error was encountered, which determines whether the input is accepted.

The function \text{E()} applies the rule \( E \rightarrow TE' \). It prints an informative message and then calls on \text{T()} followed by \text{Eprime()}. The function \text{Eprime()} must make a choice since there are two possible grammar to apply: \( E' \rightarrow +TE' \) and \( E' \rightarrow \varepsilon \). This
choice is made by checking whether the current token is PLUS. The other functions are similar.

Finally, the match() function verifies that the token is what it is supposed to be and then it gets the next token.

Build the program by running the makefile. Then run the program by typing

$ java RDParse

Enter the input

$ 123 + 456 * 789

Press CTRL-Z for EOF. The input should be accepted. We are using the “verbose” versions of the lexer and the parser. Each program prints informative messages to apprise the user of its progress. These messages can be extremely helpful in debugging. You may turn them off by commenting them out. It is better not to delete them since you may need them later for further debugging.

You should have gotten the output

123 + 456 * 789
Token = NUM, value = ’123’
E -> T E’
T -> F T’
F-> num
Token = PLUS
T’ -> e
E’ -> + T E’
Token = NUM, value = ’456’
T -> F T’
F -> num
Token = TIMES
T’ -> * F T’
Token = NUM, value = ’789’
F -> num
^Z
Token = EOF
T’ -> e
E' -> e
Accepted

Notice that some productions were applied after you indicated EOF. Why is that?

4.4 Derivations and Parse Trees

Can you write a derivation for the input, based on the output? Write it out completely. Is it a rightmost derivation? Leftmost? Neither?

Draw the parse tree.

Now run the program again, entering the expression

$ 123 * 456 + 789$

Draw the parse tree. How does it compare to the parse tree of the previous expression?

Test the program, sending the output to an output file outfile.txt. The output will be rather long, but the input should be accepted. Print the output file.

Notice that RDParser.java prints the production used as soon as it decides which one to use, even before the production has been satisfied. One effect of this is that the productions are listed in the order of a leftmost derivation. As an experiment, in each case when a production is matched, move the output statement to the end of that block. For example, in the function E(), rewrite

```java
System.out.println("E -> T E'");
T();
Eprime();
```

as

```java
T();
Eprime();
System.out.println("E -> T E'");
```

Build and run the parser again, sending the output to the file outfile2.txt. Print the output file. Compare this output to the previous output. Read the output from bottom to top. In what order did the productions appear? What kind of derivation does this indicate? Can you explain this?
Notice how the product $3 \times 4 \times 5 \times 6$ was handled. What does this indicate about the depth of the recursion? In the same vein, notice the point at which the production $E' \rightarrow +TE'$ was matched. Again, this tells us something about the depth of recursion.

Notice how much output appeared after the EOF token was received. This output, when read from top to bottom, appears to indicate that the parser is a bottom-up parser following a rightmost derivation. That is misleading; it is a top-down parser that follows a leftmost derivation. However, this indicates that the difference is subtle.

Put the output statements back where they were. You may do this by pressing CTRL-Z repeatedly to undo the earlier changes.

4.5 Improving the Parser

Now let us test our program’s ability to detect syntax errors. Run the program and enter the line

```
$ 123 ++ 456
```

What error was detected? Can you see why? Now try

```
$ 123 456
```

Was an error detected? Why not? You might notice that you did not need to indicate EOF. Why not?

Trace through the program and find out exactly how execution terminated when the next token after 123 was a number. Did this number token cause `error` to be set to true?

The problem is that when the program quits, the last token received from the lexer should be EOF. Otherwise, the lexer was not at the end of the expression. However, the program does not check that this was the case. It simply quits after the production for $E$ has been satisfied. Correct this by requiring that after satisfying the production for $E$, the token be EOF in order for the parser to report that the input was accepted.

4.6 Assignment

Expand the grammar as follows
Acceptable input is a series of assignment statements of the form \( \text{id} = \text{expr}; \)

- Expressions may include parentheses.

- Expressions may include subtraction and division.

- Factors may be identifiers.

Thus, the grammar now is

\[
S \rightarrow SS | \varepsilon \\
S' \rightarrow \text{id} = E; \\
E \rightarrow TE' \\
E' \rightarrow +TE' | -TE' | \varepsilon \\
T \rightarrow FT' \\
T' \rightarrow *FT' | /FT' | \varepsilon \\
F \rightarrow (E) | \text{id} | \text{num}
\]

Be sure to test your program with incorrect input as well as correct input.

Place the files Token.java, MyLexer.java, RDParser.java, and Makefile in a folder named Lab_04, zip it, and drop it in the dropbox.
Laboratory 5

Using JLex with a Predictive Parser

Key Concepts

- The JLex-parser interface
- JLex EOF values
- Predictive parsing
- Java Stack objects

Before the Lab


Preliminary

Copy the folder Lab_05 from the Compiler Design CD to your folder. Also copy the files Err.java and Warning.java from your Lab_03 Folder.

5.1 Introduction

In this lab we will use a lexer generated by JLex together with a table-driven predictive parser. We will see how to set up the interface so that the parser will communicate properly with the lexer. That will be quite simple. The same interface would be used
with RDParse or any other hand-written parser. Then we will look at how the parser works. It is the model of elegance. Since the parser uses a stack, we will also have an opportunity to become familiar with the Java Stack class and some issues related to its use.

5.2 Lexer-Parser Interface

This parser uses a parse table. (See Table 5.1 below.) In the parse table, there is a column specifically for EOF. In past labs, JLex assigned EOF the value -1 and returned it automatically. In this lab we must be sure that EOF has the value that corresponds to its column in the table, which is 6. Therefore, in the file Token.java, we include EOF in the list of symbols. Open Token.java and see that that is done.

Now we must make JLex return 6 instead of -1 on end-of-file. To do this, we must make two changes in the JLex file. First, change the directive %integer to

```
%type int
```

This prevents JLex from returning -1. Second, since the return value is no longer automatic, we must tell JLex what it is. To do this, add the following in the directive section of the JLex file.

```
%eofval{
    return Token.EOF;
}%eofval}
```

This provides the action that JLex will take on end-of-file. See Sections 2.2.12 and 2.2.17 in the JLex manual.

Currently the parser in PredParser.java makes no mention of a lexer. We must add some lines to the parser so that it can accept tokens from a Yylex object. Open the file PredParser.java.

First, we must declare lex to be a Yylex object. Add the class variable

```
public static Yylex lex;
```

to the PredParser class. It should be declared before main() so that it will be a class variable, available to all functions in the class. Then in the main function, we must create the object. Add the line
lex = new Yylex(System.in);

at the beginning of main().

Now to get a token, we call the yylex() function of the Yylex class. Thus, where the recursive descent parser called the function next_token(), we will now call yylex(). Add the line

\[
\text{token = lex.yylex();}
\]

in main() just before the while loop, just as in Lab 3.

Add the same line at the end of the function match(), just before the return statement. That’s it! Our parser will now communicate with our lexer.

Now we will look at how the parser works.

### 5.3 The Productions

The nonterminals are $E, E', T, T', F$ and the grammar is

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' | \varepsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' | \varepsilon \\
F & \rightarrow (E) | \text{id} | \text{num}
\end{align*}
\]

Look at the beginning of the PredParser class. You will see the nonterminals defined as the negative integer constants $-1, -2, \ldots, -5$. The tokens, or terminals, will be assigned non-negative integer values. Thus, the sign of the grammar symbol will be our way of distinguishing between terminals and nonterminals. (Note that we avoid the use of $-0$ for a nonterminal since 0 is used for tokens and $+0 = -0$.)

Next, you see the productions themselves, as strings. This array is included so that we can print informative messages about which production is being matched.

Then there is a two-dimensional array containing the right-hand side of each production, as a list of grammar symbols. It is here that we must be able to distinguish between terminals and nonterminals, since, in general, they are mixed together in productions.
5.4 The Parse Table

The heart of this program is the parse table. See Table 5.1. It is this table that allows us to avoid using recursion. This table also makes it possible to perform more elegant error recovery and it sure would be fun to do that, but in the interest of time, we will have to skip that.

The tokens are defined in a separate Token class. At this point, it would be helpful to see what their values are. Open the file Token.java. You see that the tokens PLUS (+), TIMES (*), LPAREN (()), RPAREN (), ID, and NUM are assigned the values 0, 1, 2, \ldots, 5 and EOF ($) is assigned the value 6. These values will correspond to the columns in the parse table.

The rows of the parse table correspond to the nonterminals $E$, $E'$, $T$, $T'$, and $F$, in that order.

The entries in the table are integers representing the productions, as listed above in the array prodList. In cells representing an error, we have entered ERROR (−1), an integer which corresponds to no production. Now it is the job of the parsing
algorithm to read the table and take the appropriate action.

5.5 The Parsing Algorithm

The main function is now quite simple, thanks to the various arrays and tables already constructed. A few things need to be initialized. First, the stack must be created and then the symbols $ and $E must be pushed onto it, in that order. Recall that $ represents EOF. In Java it is easy to use a stack since there is already a Stack class. Go to the Java website

http://java.sun.com/j2se/1.4.2/docs/api/

and look up the Stack class in the package java.util. Read the Method Summary.

Now we will create and initialize a Stack. First, declare stack to be a Stack object:

```java
public static Stack stack;
```

Then, at the beginning of main(), create a new Stack object:

```java
stack = new Stack();
```

In Java, Stacks can hold any kind of object whatsoever. That is why the return type of peek() and pop() is Object. Every non-primitive class is a subclass of the Object class. However, the primitive classes, such as int, are not subclasses of the Object class. Therefore, we cannot push ints onto our stack. That’s too bad, because the grammar symbols in this program are all ints. We will have to convert our int primitives to the wrapper-class Integer objects. In general, to create an Integer from an int, say n, we write the expression

```java
new Integer(n)
```

Use this together with the push() function to initialize the stack with the grammar symbols $ and $E. Recall that $ is represented by Token.EOF.

Note that in the while statement we are using the Stack function empty() to see if the stack is empty. Once we enter the while loop, the first thing we need to do is to see what symbol is on top of the stack. We don’t want to pop it, but just look at it. So we should use the peek() function. The peek() function will return a
reference to the Object on top of the stack. To use the object, we must cast it as an Integer. Furthermore, to use it as an int, we must convert it to int. That is done with the Integer function intValue(). Put this all together to write a statement, or statements, that

- Retrieves the object from the top of the stack.
- Converts it to Integer.
- Gets its int value.
- Assigns the int value to the variable symbol.

Now we check to see if symbol is a terminal or a nonterminal. If it is non-negative, then it must be a terminal. If it is negative, then it must be a nonterminal. So the program divides into two cases accordingly.

If the symbol is a terminal, then it must match the current token. That is handled by the match() function, which also gets the next token. Look at match() to see how it works.

If the symbol is a nonterminal, then it is used, together with the current token, to look up a production in the parse table. Note that to use symbol as an index, it must be made positive and then decremented by 1. After retrieving the entry from the table, we first make sure that the entry is not ERROR. If not, then we announce the production we are using, pop the nonterminal from the stack, and push the grammar symbols from the production onto the stack.

That’s it! We let the while loop run until the stack is empty or until an error condition occurs. The program finishes by reporting whether the input was accepted or rejected.

5.6 Running the Program

Now use the makefile to build the program. Test your program with the input

\[ a + 3 \]

If it works, then test it with more complicated expressions:

\[ a*b + 3*c \]
\[ 123*(dog + 456*(cat + bear)) + 789 \]
Test your program with some invalid input, too, such as

```plaintext
a + 2b
b + + c
a & b
```

5.7 Expand the Grammar

Let us expand the grammar to include the productions

```plaintext
E' \rightarrow -TE'
T' \rightarrow /FT'
```

You will have to recompute FIRST and FOLLOW for the nonterminals and update the parse table. There are no new nonterminals, but there are two new terminals and two new productions, so you will have to

- Add two tokens, - and /, to the `Token` class.
- Add two strings to the `prodName` array.
- Add two productions to the `prodList` array.
- Add two integers to the `prodSize` array.
- Add two columns to the parse table.

Everything else should work as before. I suggest that you add these things to the ends of the arrays so that you do not disturb what is already there. The one exception might be the list of tokens and the parse table. It is customary, but certainly not necessary, that $ be the last token. That will be your call. However, it is necessary that the `ERROR` token be at the end of the list of tokens since it is not a terminal.

Test your program with input such as

```plaintext
a + b*(c - (d + e)/3) - 2/3*f
```

5.8 Assignment

Zip the files `tokens.lex`, `Token.java`, `PredParser.java`, `Err.java`, `Warning.java`, and `Makefile` in a folder named `Lab_05` and drop it in the dropbox.
Laboratory 6

Using JLex and CUP

Key Concepts

• The CUP files

• Context-free grammars

• Semantic actions

• LR and LALR parsing

• Precedence

• Associativity

• Shift/reduce conflicts

Before the Lab

Read Sections 4.5 and 4.7 of Compilers: Principles, Techniques, and Tools. Also read the CUP User’s Manual.

Preliminary

Copy the folder Lab_06 from the Compiler Design CD to your folder. Also copy the file tokens.lex from your Lab_03 folder and the files Err.java and Warning.java from your Lab_05 folder.
6.1 Introduction

The program CUP is a parser generator. The programmer specifies a context-free grammar in a .cup file. For each production in the grammar, he specifies an action to be taken. Whenever a rule is matched, the specified action is taken.

CUP creates an LALR parser, which is a variation of an LR(1) parser. It creates the action and goto tables and uses the LALR algorithm to parse the input. In this lab we will learn how to write a CUP file and how to make CUP and JLex work together to create a complete program.

6.2 Modifying the JLex File

We will not need the file Token.java since the tokens will be defined by CUP.

The JLex file must be modified so that its output will interface with the parser generated by CUP. In the user-code section, we need the line

```java
import java_cup.runtime.Symbol;
```

This import statement will be copied into the file Yylex.java where it will make the lexer aware of the Symbol class. The lexer will use the Symbol class, which is found in the directory CUP/java_cup/runtime, to return tokens to the parser. Later in this lab, we will take a closer look at the Symbol class.

To tell JLex that the tokens will be sent to a program generated by CUP, we must add the %cup directive

```
%cup
```

in the directive section. Also, remove the directive

```%integer```

The return value will be of type Symbol, not int. We must use the %eofval to specify the return value. In the directive section, add the lines

```java
return new Symbol(sym.EOF);
```

Now we must instruct the lexer to return Symbol objects consisting of a token and possibly a value.
6.3 The Symbol Class

A Symbol object has several data members. We will be interested in two of them: `sym` and `value`. Open the file `Symbol.java` in the directory `CUP/java_cup/runtime`. Scroll to the bottom of the file to see the data members. What are the types of `sym` and `value`? The value of `sym` will be taken from the `sym` class, which will be created by CUP. It will be similar to the `Token` class that we have been using. The value of `value` will be set by us as necessary in the JLex file. Note that `value` is of type `Object`, so we may assign to it an object of any type. We will assign it a `String` value.

Now look at the `Symbol` constructors. One `Symbol` constructor has prototype

```
public Symbol(int sym_num);
```

Another one has prototype

```
public Symbol(int id, Object o);
```

These are the two constructors that we will use. If the token does not have an associated value, then we use the constructor that takes only the token number. For example, the PLUS token has no associated value, so we would return

```
new Symbol(sym.PLUS)
```

If the token has an associated value, then we will use the second constructor, which takes the token number and its value. For example, the ID token has an associated value, the name of the identifier. In this case, we would return

```
new Symbol(sym.ID, new String(yytext()))
```

In the JLex file, replace each `Token` return object with a `Symbol` object. For the ID and NUM tokens, we must also return the associated value. This will come from the `yytext()` function. Create a new `String` from this and use it as the second parameter in the `Symbol` constructor for these two token types, as shown in the example above. Also, be sure that the JLex file returns the token `sym.error` when an invalid token is found. Make these changes throughout the JLex file. Now the JLex file is ready to be used by CUP.
6.4 The CUP File

Open the file `grammar.cup`. CUP will use the CUP file to create the `parser` and `sym` classes. The CUP file begins with user code that will be copied directly into the parser. We have included the `import` statement

```
import java_cup.runtime.*;
```

Next, we have some “parser code.” We could write a separate program to contain a main function, as we did in Lab 3. Instead, for now we will use the `parser code` directive to create a main function in the `parser` class. Read the parser code in the CUP file. Later we will look at the `parser` class to see what this code does.

Next we list the terminals. Generally, these are the same as the tokens returned by the lexer, except for `EOF`, which we do not list. CUP will copy these names to the `sym` class and assign them numerical values. This file currently defines all of the tokens that were used in Lab 3, even though a number of them are not used in the grammar, yet. If any are missing, you will get an error message. In that case, just go back and add them in. In any case, you will get some warnings about the unused terminals. You can ignore those warnings for now.

Next we list the nonterminals. Each of these should appear on the left side of a production later in the file. If any nonterminals are listed, but not used, then CUP will give us a warning. Indeed, if one is listed, but is not accessible from the start symbol, CUP will give us a warning.

To resolve shift/reduce conflicts arising from the operators, we may specify precedence levels. The two lines

```
precedence left PLUS;
precedence left TIMES;
```

say that `PLUS` has a lower precedence than `TIMES` and that both operators are left associative. (Terminals are listed in order of *increasing* precedence.)

6.5 The Grammar

The grammar currently in the CUP file is

```
expr → expr + expr | expr * expr | num
```
where \( expr \) is the only nonterminal. This grammar by itself is ambiguous because it is not clear whether addition or multiplication comes first, but the ambiguity has been removed by the precedence rules. The expression \( E+E \) will not be reduced to \( E \) if the next token is \( * \), but the expression \( E*E \) will be reduced to \( E \) if the next token is \( + \). Also, in an expression such as \( b + c + d \), left associativity will cause the rule \( E \rightarrow E+E \) to be applied first to \( b + c \) rather than to \( c + d \).

Build the parser from the file `grammar.cup`, by typing the command

```
$ java java_cup.Main < grammar.cup
```

This creates the files `parser.java` and `sym.java`. Next, build the lexer by typing the command

```
$ java JLex.Main tokens.lex
```

Now compile the files `tokens.lex.java`, `parser.java`, and `sym.java`. To run the parser, type

```
$ java parser
```

First test the parser on some simple input such as

```
$ 123 + 456
```

Use CTRL-Z (4 times!) to indicate EOF. Now run the program on the input

```
$ 123 + 456 * 789
```

and note that the multiplication is done first. Run it again on the input

```
$ 123 * 456 + 789
```

and note that again the multiplication is done first.

To see the associativity, use the input

```
$ 123 + 456 + 789
```

As an experiment, change “left” to “right” in the precedence statement for `PLUS` in the CUP file, rebuild the parser, and run it again with the previous input. You should see that \( 456 + 789 \) was processed first. How can you tell that \( 456 + 789 \) was processed first? You might try running the parser with the input
$ 123 \times 456 \times 789$

which is still left associative, and compare the output.

Change “right” back to “left” before continuing. (Do not bother rebuilding yet since we are about to make more changes.)

6.6 Shift/Reduce Conflicts

As an experiment, comment out the precedence rules by using //-comments. Build the program again. This time we get a number of messages telling us of shift/reduce conflicts in the LR parse table. In each case, we are told how the conflict was resolved. Generally, shift/reduce conflicts are resolved in favor of shifting.

However, CUP will not produce Java source code unless the number of conflicts found matches the expected number. (Note the message “No code produced.”) In the makefile, in the command

```
java java_cup.Main < grammar.cup
```

we may add a command-line argument that gives the expected number of conflicts. In the output from the build attempt, count the number of conflicts (or read the message that tells you the number). Then write

```
-expect n
```

in the above command before the <, where n is the number of conflicts that you counted (the expected number). (Make this change in the makefile, if you are using the makefile.) Build the program again and test it. It should work, although you should test to see exactly what the precedence rules are. Use input such as

$ 123 + 456 \times 789$

and

$ 123 \times 456 + 789$

to find out. What are the precedence rules, apparently?

Now restore things by removing the -expect argument (from the makefile, if necessary) and uncommenting the precedence rules in the CUP file. Whenever possible,
we should resolve conflicts through precedence rules. Later we will encounter an ambiguity in the statement

\[
\text{if (expr) if (expr) else stmt;}
\]

that cannot be resolved through precedence rules.

### 6.7 Semantic Actions

With each grammar rule there is an associated semantic action. This is what gives meaning (semantics) to the grammar rules. In a later lab we will fill in more meaningful semantic actions. Right now, we just want to print the grammar rules that are applied so that we can track the progress of the parser as the input is processed. Later, in our compiler, the semantic actions will be used to generate assembly code.

When you read the output, note that the rules are applied after the right-hand sides are matched. That is because this is an LR parser.

### 6.8 The sym and parser Classes

Open the file `sym.java`. It was created by CUP from the CUP file. Note that it defines an integer value for each of the terminals listed in the CUP file. It also defines two more constants, `EOF` and `error`.

Now open the file `parser.java`. It was also created by CUP from the CUP file. Scroll down a bit and see that there are three tables: a production table, an action table, and a reduce table. They are in a compressed, unreadable (by us) form. Scroll down a bit further and you will see the parser code that we wrote in the CUP file. (It is now the main function.)

The main function creates a `parser` object, which is simply an instance of this class. At the top of this file, there were two `parser` constructors (above the tables). The second one takes as its parameter a `Scanner` object. (Recall that “scanner” is another name for a lexer.) Open the file `Yylex.java` and read the first line of the class definition:

```
class Yylex implements java_cup.runtime.Scanner
```

This tells us that a `Yylex` object is a `Scanner` object. Thus, we pass to this `parser` constructor a new instance of a `Yylex` object. Altogether, this means that the parser will use `Yylex` as its lexer. It will call on `Yylex` as necessary for tokens.
Next, note that after the parser object is created, we call on the function parse(). You can search throughout the file parser.java and you will not find the member function parse(). But look at the first line of the parser class definition:

```java
public class parser extends java_cup.runtime.lr_parser
```

This tells us that the parser class is a subclass of the lr_parser class. Open the file lr_parser.java, which is found in the directory CUP/java_cup/runtime. This is an abstract class. Down around line 500 you will find the parse() function. Take a few minutes to look it over. It is not hard to see that it is implementing an LR algorithm. First, it initializes the stack to the start state (0). Then it goes into a while loop where it gets the value of act from the action table. If act is greater than 0, it represents a shift operation, so it pushes a symbol onto the stack. If act is less than 0, it represents a reduce operation, so it pops several symbols from the stack and pushes a new symbol. If act equals 0, it indicates an error. We used ideas very similar to that in the predictive parser in Lab 5.

Go back to the main function in the parser class. Notice that we use a try/catch construct. If an error occurs in the try clause, execution jumps to the catch clause. Otherwise, when the try clause is finished, execution skips over the catch clause. Notice how we have used this to report whether the input was accepted or rejected.

### 6.9 The Action and Goto Tables

Let’s use some command-line arguments to see what is going on inside CUP. Our example is so small that it shouldn’t be very complicated. Use CUP to build the parser once more, this time using the -dump_grammar switch. Type

```
$ java java_cup.Main -dump_grammar < grammar.cup
```

You should see listed all the terminals, nonterminals, and productions, each with a number. Note that the grammar has been augmented with a new start symbol and the production

\[ [1] \texttt{$START ::= expr \; EOF} \]

Now build the parser again using the switch -dump_tables. This will output the action and goto tables. Using the numbered terminals, nonterminals, and productions seen above, we could easily use this output to fill in all the entries in the action and goto tables.
6.10 The CUP Debugger

In the file grammar.cup, in the parser code section, change the parser call parse() to debug_parse() and recompile. Run the program again with some simple input. You will see that the CUP debugger informs you of each action in the LR algorithm. This might be useful if we had the tables written out.

Change debug_parse() back to parse() and rebuild.

6.11 Assignment

Expand the grammar to include all of the following. This will be the full grammar for our C compiler. In subsequent labs, we will implement more and more parts of this grammar.

In the JLex file, add the keywords int, double, if, else, return, read, and print as separate tokens. In Lab 7 we will learn a better way to handle keywords, but this will suffice for now.

The terminals are given in Table 6.1. The nonterminals are listed in Table 6.2.

<table>
<thead>
<tr>
<th>ID</th>
<th>MOD</th>
<th>COMP</th>
<th>ABOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM</td>
<td>UNARY</td>
<td>BAND</td>
<td>ABXOR</td>
</tr>
<tr>
<td>STR</td>
<td>INC</td>
<td>BOR</td>
<td>ASHL</td>
</tr>
<tr>
<td>LPAREN</td>
<td>DEC</td>
<td>BXOR</td>
<td>ASHR</td>
</tr>
<tr>
<td>RPAREN</td>
<td>EQ</td>
<td>SHL</td>
<td>COMMA</td>
</tr>
<tr>
<td>LBRACE</td>
<td>NE</td>
<td>SHR</td>
<td>SEMI</td>
</tr>
<tr>
<td>RBRACE</td>
<td>LT</td>
<td>ASSIGN</td>
<td>INT</td>
</tr>
<tr>
<td>LBRACK</td>
<td>LE</td>
<td>APLUS</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>RBRACK</td>
<td>GT</td>
<td>AMINUS</td>
<td>IF</td>
</tr>
<tr>
<td>PLUS</td>
<td>GE</td>
<td>ATIMES</td>
<td>ELSE</td>
</tr>
<tr>
<td>MINUS</td>
<td>NOT</td>
<td>ADIVIDE</td>
<td>RETURN</td>
</tr>
<tr>
<td>TIMES</td>
<td>AND</td>
<td>AMOD</td>
<td>READ</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>OR</td>
<td>ABAND</td>
<td>PRINT</td>
</tr>
</tbody>
</table>

Table 6.1: The terminals for the CUP file

The precedence and associativity of the operators is shown in Table 6.3. The table is arranged from highest to lowest precedence. Finally, the productions of the grammar
arg  expr  func  n
args exprs glbl  prog
expr fargs glbls stmt
dcl fbeg lval stmts
dcls fname m type

Table 6.2: The nonterminals for the CUP file

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNARY, NOT</td>
<td>right</td>
</tr>
<tr>
<td>TIMES, DIVIDE, MOD</td>
<td>left</td>
</tr>
<tr>
<td>PLUS, MINUS</td>
<td>left</td>
</tr>
<tr>
<td>LT, LE, GT, GE</td>
<td>left</td>
</tr>
<tr>
<td>EQ, NE</td>
<td>left</td>
</tr>
<tr>
<td>AND</td>
<td>left</td>
</tr>
<tr>
<td>OR</td>
<td>left</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>right</td>
</tr>
</tbody>
</table>

Table 6.3: The precedence and associativity of operators

are the following.
The Program

\[ prog \rightarrow glbls \]

Globals

\[ glbls \rightarrow glbls \, glbl \]
\[ \quad | \quad \varepsilon \]
\[ glbl \rightarrow dcl \]
\[ \quad | \quad func \]

Markers

\[ m \rightarrow \varepsilon \]
\[ n \rightarrow \varepsilon \]

Declarations

\[ dcls \rightarrow dcls \, dcl \]
\[ \quad | \quad \varepsilon \]
\[ dcl \rightarrow type \, ID \, SEMI \]
\[ type \rightarrow INT \]
\[ \quad | \quad DOUBLE \]

Function Definitions

\[ func \rightarrow fbeg \, stmts \, m \, RBRACE \]
\[ fbeg \rightarrow fname \, fargs \, LBRACE \, dcls \]
\[ fname \rightarrow type \, ID \]
\[ \quad | \quad ID \]

Function Arguments

\[ fargs \rightarrow LPAREN \, args \, RPAREN \]
\[ \quad | \quad LPAREN \, RPAREN \]
\[ args \rightarrow args \, COMMA \, arg \]
\[ \quad | \quad arg \]
\[ arg \rightarrow type \, ID \]
Statements

\[ \text{stmts} \rightarrow \text{stmts} \; m \; \text{stmt} \]
\[ \rightarrow \epsilon \]
\[ \text{stmt} \rightarrow \text{expr} \; \text{SEMI} \]
\[ \rightarrow \text{RETURN} \; \text{SEMI} \]
\[ \rightarrow \text{RETURN} \; \text{expr} \; \text{SEMI} \]
\[ \rightarrow \text{LBRACE} \; \text{stmts} \; \text{RBRACE} \]
\[ \rightarrow \text{IF} \; \text{LPAREN} \; \text{cexpr} \; \text{RPAREN} \; m \; \text{stmt} \]
\[ \rightarrow \text{IF} \; \text{LPAREN} \; \text{cexpr} \; \text{RPAREN} \; m \; \text{stmt} \; n \; \text{ELSE} \; m \; \text{stmt} \]
\[ \rightarrow \text{SEMI} \]
\[ \rightarrow \text{READ} \; \text{lval} \; \text{SEMI} \]
\[ \rightarrow \text{PRINT} \; \text{lval} \; \text{SEMI} \]

Conditional Expr.

\[ \text{cexpr} \rightarrow \text{expr} \; \text{EQ} \; \text{expr} \]
\[ \rightarrow \text{expr} \; \text{NE} \; \text{expr} \]
\[ \rightarrow \text{expr} \; \text{LT} \; \text{expr} \]
\[ \rightarrow \text{expr} \; \text{LE} \; \text{expr} \]
\[ \rightarrow \text{expr} \; \text{GT} \; \text{expr} \]
\[ \rightarrow \text{expr} \; \text{GE} \; \text{expr} \]
\[ \rightarrow \text{NOT} \; \text{cexpr} \]
\[ \rightarrow \text{cexpr} \; \text{AND} \; m \; \text{cexpr} \]
\[ \rightarrow \text{cexpr} \; \text{OR} \; m \; \text{cexpr} \]
\[ \rightarrow \text{LPAREN} \; \text{cexpr} \; \text{RPAREN} \]
\[ \rightarrow \text{expr} \]
Numerical Expr.

\[
\begin{align*}
\text{exprs} & \rightarrow \text{exprs COMMA expr} \\
\& \rightarrow \text{expr} \\
\text{expr} & \rightarrow \text{lval ASSIGN expr} \\
\& \rightarrow \text{expr PLUS expr} \\
\& \rightarrow \text{expr MINUS expr} \\
\& \rightarrow \text{expr TIMES expr} \\
\& \rightarrow \text{expr DIVIDE expr} \\
\& \rightarrow \text{expr MOD expr} \\
\& \rightarrow \text{MINUS expr} \\
\& \rightarrow \text{LPAREN expr RPAREN} \\
\& \rightarrow \text{lval} \\
\& \rightarrow \text{ID LPAREN RPAREN} \\
\& \rightarrow \text{ID LPAREN exprs RPAREN} \\
\& \rightarrow \text{NUM} \\
\& \rightarrow \text{STR} \\
\text{lval} & \rightarrow \text{ID}
\end{align*}
\]

For each production, provide an action that prints the production. For example, the action for the production

\[
\text{expr} \rightarrow \text{ID LPAREN exprs RPAREN}
\]

should be

\[
{\text{: System.err.println("expr -> ID ( exprs )"); ;}}
\]

Also, as a very special case, in the production

\[
\text{expr ::= MINUS expr}
\]

add the phrase `\%prec UNARY` after the action part. This will give the negation operator the same precedence as `UNARY`.

This lab will serve as Project 2. Zip the files `tokens.lex`, `grammar.cup`, and `Makefile` in a folder named `Project_2` and drop it in the dropbox.
Part IV

A Simple Compiler
Laboratory 7

The Symbol Table

Key Concepts

- Symbol tables
- Reserved words
- Hash tables
- Linked lists
- Block level
- Semantic actions
- try/catch blocks
- Syntax errors

Before the Lab

Read Sections 2.7, 5.1, 7.4, and 7.6 of Compilers: Principles, Techniques, and Tools. Section 2.7 gives an easy introduction to the idea of a symbol table. Section 5.1 discusses inherited and synthesized attributes of grammar symbols. Some of Section 7.4 discusses function calls, which we are not concerned with yet. Try to understand it now, and then we will re-read it later when we introduce function calls into our compiler.

For information on hash tables, read the chapter or section on hash tables in the textbook you used in Coms 262.
Preliminaries

Copy the folder Lab_07 from the Compiler Design CD to your folder. Also, copy the files tokens.lex, grammar.cup, Err.java, and Warning.java from your Lab_06 folder to your Lab_07 folder.

7.1 Introduction

The main purpose of this lab is to introduce the symbol table, the table in which all the identifiers are stored along with information about them. Since the symbol table is rather complicated, I have written nearly all the code. Your main job in this lab will be to understand how the code works.

The compiler will interact with the symbol table in two fundamental ways. When a variable is declared, the compiler will enter it as a new entry in the symbol table. When a variable is referenced in an expression, the compiler will look it up in the symbol table to retrieve necessary information about it, such as its data type.

This lab will also introduce (non-trivial) semantic actions. Up to this point, the only actions taken by the parser in response to matching grammatical patterns has been to announce that the pattern was matched. Now it will have to respond in a more substantial way to declarations of variables and uses of variables in expressions.

7.2 The Symbol Table

Each identifier in a program has various attributes associated with it. For example, a variable has a name, a type, and a memory location. (Of course, it also has a value, but that is not determined until run time.) All information associated with the identifiers is organized in a symbol table. The symbol table consists of a collection of IdEntry objects. Each IdEntry object stores all information relevant to a single identifier.

When an identifier is first found, an IdEntry object is created and placed in the symbol table. The symbol table is built in levels, according to the levels at which the identifiers are declared. Level 1 contains reserved words (keywords). This will be explained in more detail later. (Reserved words are not really identifiers.) Level 2 contains global variables, including function names. In C, all functions are global. Level 3 contains local variables. Levels beyond 3 contain variables that are declared
within blocks within functions. At the present time, all of our identifiers are global. We will not have local variables until we introduce functions in Lab 11.

When the program enters a new block (by encountering a left brace), a new level is created in the symbol table. Any variables declared at that level are placed in this level of the symbol table. When the program exits a block (by encountering a right brace), the corresponding level of the symbol table is disposed of; those variables no longer exist.

7.3 The IdEntry Class

Open the file IdEntry.java to see what the data members are. The member name is the identifier value, i.e., the string that was found in the source file. The member type is the data type. For the time being, this will be int. In Lab 10, we will introduce the double type. The member blockLevel is the level at which the identifier was defined: level 1 for reserved words, level 2 for globals, level 3 for locals, and so on. The member scope is either GLOBAL, LOCAL, or PARAM. The member offset will be used later when we introduce function calls. It represents the position of a local variable or parameter in the activation record on the run-time stack.

The IdEntry class includes a toString() function. In Java, whenever an object is sent to the println() function, that object’s toString() member function is automatically invoked. We include the function here for debugging purposes. To help follow what is happening, we may want to print an IdEntry object. If you want less IdEntry information displayed, you may modify this function.

7.4 The Hashtable and LinkedList Classes

The various levels of the symbol table could be implemented individually as linked lists. Instead, we will implement them as hash tables in order to speed up access to the elements and to give ourselves come experience with using hash tables. Java provides a complete Hashtable class. Go to the Java web page and read about the Hashtable class. The class is found in the java.util package. Note the functions put() and get(). These functions will allow us to put identifiers in the table and later retrieve them. Hold on to that page; you will need it again in a few minutes. You may want to read later the full description of hash tables in the introductory part of the Hashtable web page.
Since levels 3 and above of the symbol will be created and destroyed as we enter and exit functions, we need to be able to create and destroy the various hash tables. To do this, we will implement the symbol table as a linked list of hash tables. The highest level, which will always be the current level, will be at the head of the list. When we move back down to the next lower level, we will destroy this table. The previous table will then move to the head of the list and become the current table.

This is as it should be. For suppose a variable count is declared as a global variable and then later declared as a local variable to a function. If we encounter a use of count within the function, it should refer to the local variable, not the global variable. Our compiler will work this way if it searches the hash table that is at the head of the list first. If it fails to find the identifier there, it should search the next hash table, and so on.

To implement this linked list of hash tables, we use Java’s LinkedList class. Go back to the Java web page and read about the LinkedList class. Note the functions addLast(), removeLast(), and get(). These functions will allow to add and remove hash tables from the list and to get a particular hash table from the list.

7.5 The SymbolTable Class

Now we get to the heart of the matter: the SymbolTable class. Open the file SymbolTable.java. As expected, class variables include a LinkedList called idTable and a variable level that keeps track of the current block level. There are three more variables (dclSize, fArgSize, and retType) which will be used later when we implement function calls.

The structure of the symbol table is shown in Figure 7.1 (supposing we had 3 levels defined):

The SymbolTable class includes eight functions:

- init()
- initResWords()
- installResWord()
- enterBlock()
- leaveBlock()
• install()

• idLookup()

• idDump()

The function init() initializes the symbol table by creating the linked list and putting a null entry at level 0. The function initResWords() installs the reserved words by calling on installResWord() once for each reserved word. Right now we have only one keyword: int. Later we will install other reserved words as they are needed.

Take a minute to look at the functions initResWords() and installResWord(). They are very simple.

Why install reserved words in the symbol table, as though they were identifiers? The sole purpose for doing this is to simplify the lexer, which speeds up the process of lexical analysis. If we had the lexer detect each reserved word on its own, the DFA transition table would become quite large. Instead, we let the lexer first find the reserved word as an identifier. Then it is looked up in the symbol table. If it is found, then the block level is checked. If the block level is 1 (reserved word), then the identifier must be a reserved word. Clever! This also prevents the use of a keyword as an identifier.

The enterBlock() and leaveBlock() Functions

When the compiler enters a new block, the block level should be increased by 1. When it leaves a block, the block level should be decreased by 1. In addition to adjusting the block level, the function enterBlock() must create a new hash table in the linked
list of hash tables. The function `leaveBlock()` must destroy the most recent hash
table in the list.

Read the code for `enterBlock()`. See that it increments `level`, creates a new
hash table, and adds it to the end of the linked list. Now look at the function
`leaveBlock()`. This is just as simple as `enterBlock()` since all we have to do is
to destroy the top-level hash table and decrement `level`. Write the body of this
function now.

The functions `install()` and `idLookup()` are a little more complicated.

The `install()` Function

This function has two parameters, which are the identifier as a string and the block
level at which it should be installed. If the block level parameter is 0, then the
identifier is installed at the current level `level`, whatever that may be. Otherwise, it
is installed at the specified level.

The second part of the function gets the hash table at the specified level from the
linked list.

The third part creates a new `IdEntry` object, assigns to it the currently known
information, and puts it in the hash table. As more information about the object
becomes available, we will look it up in the table and add the information. This
function uses the `Hashtable` functions `get` and `put()`. Be sure to read about these
functions on the Java website. What does `get()` return if the object is not found?

The `idLookup()` Function

The purpose of the `idLookup()` function is to look up the name of an identifier in
the symbol table. If the identifier is found, then the function will return a reference
to the `IdEntry` object in the table. Otherwise, it returns `null`.

The parameters of `idLookup()` are

(String s, int blkLev)

Obviously, `s` is the name of the identifier. The parameter `blkLev` is the block level at
which the identifier is defined. If a positive value is passed, then `idLookup()` will look
for the identifier at that block level only. If the value 0 is passed, then `idLookup()`
will search through all block levels, beginning with the current (highest) level. In
either case, if `idLookup()` cannot find the identifier, then it returns `null`. 
7.6 Reserved Words and the Lexer

The reserved words are the keywords in the language. We could have designed the lexer to find the keywords and return the appropriate token. However, that would have made the lexer far more complex. True, it would be simple enough to type the keywords in the JLex file and let JLex handle it, but the lexer itself would be unnecessarily complicated, thereby slowing down the compiler.

The standard approach to handling keywords is to enter them into the symbol table at level 1, just below the global variables and functions, along with their token values. This is done before the lexer is started. Then later when the lexer locates an “identifier,” it will first check level 1 of the symbol table. If it finds the “identifier” there, it will return the associated token from the table instead of the identifier token. This is more efficient than designing a complicated lexer. Furthermore, it is very easy to maintain the program in the event that we want to add a new keyword.

Open the files token.lex and YylexFunction.lex. Copy and paste the contents of YylexFunction.lex into the directive section of token.lex. Now you will have two additions to token.lex: a Yylex class variable lineNum and a Yylex class function lookup() defined. The variable lineNum keeps track of the current line number. In the file tokens.lex, add the action

    lineNum++;

as the action to take when the newline character is matched. Now read the code for the function lookup() and see that it looks up an identifier and takes the appropriate action if it is found in the reserved-word part of the symbol table. This function is called in the lexer action associated with identifiers later in the file. Change the action for the identifier pattern so that it simply calls lookup() and returns to the parser the value returned by lookup().

At this time, remove the patterns int, double, if, else, return, read, and print from the lexer and have them installed in the symbol table by the initResWords() function of the SymbolTable class.

7.7 The Parser

In the file grammar.cup, we will implement actions for the following productions:

    glbl  ::= dcl
In what follows, modify the actions for only these productions. The nonterminal `dcl` represents a declaration. The nonterminal `type` is a data type (so far, only `int`). And the nonterminal `lval` represents an `l`-value, i.e., an expression that is permitted to appear on the left side of an assignment. In our compiler, that will probably be only an identifier, but in C `l`-values include array members (e.g., `a[i]`), pre-incremented objects (e.g., `++n`), dereferenced pointers (e.g., `*p`), as well as a number of other expressions. In C++ it also includes function calls if the function returns a value by reference.

As we add to our compiler later, we will expand these productions. For example, we will add

```
type ::= DOUBLE
```

and we could add

```
lval ::= ID LBRACK expr RBRACK
```

for references to array elements.

In productions involving the symbol table, we will soon add semantic actions. Semantic actions are actions that give semantic meaning to the grammar rules. To keep things orderly, we will relegate most of our semantic actions to a Java file `SemanticAction.java`. The `SemanticAction` class will contain as its member functions the actions to be taken for each of the grammar rules (whenever action is required).

Now let’s look in `SemanticAction.java` to see what the actions are. There are two functions: `id()` and `dcl()`. It will be convenient whenever possible to name the functions after the corresponding grammar symbols. Let’s look at the `dcl()` function first, because a variable must first appear in a declaration before it is used in an expression.

This function looks up the identifier in the symbol table. If it is there, a (non-null) reference to the `IdEntry` is returned. That should be an error since an identifier should not be declared more than once. If it is not there, then a null value is returned.
In that case, the identifier is installed in the symbol table at the current level and some of its attributes are recorded.

Now look at the \texttt{id()} function. This function will be called when an identifier is encountered outside of a declaration. In this case, the variable should already have been declared and entered in the symbol table. If it hasn’t, then an error message is printed and the identifier is installed at the current level, but with no attributes other than its name and current block level. (Nothing else is known about it.)

7.8 The \texttt{Ops.java} File

The purpose of the file \texttt{Ops.java} is just to define a number of symbolic constants that will be used elsewhere. In C this would have been done in a header file. This file will define the different possible scopes, which so far is only global, and the different possible object types, which so far is only \texttt{int}. Later we will add quite a bit more to this file.

7.9 Running the Program

We are just about ready to test the program. There is only one file left to look at. Open the file \texttt{TableBuilder.java}. This contains the main function that gets the whole process started. It is very simple. It calls on \texttt{init()} to initialize the symbol table. Then it calls on \texttt{enterBlock()} to bump the level from 0 to 1 and create the level-1 hash table. Then it calls \texttt{initResWords()} to install all of the keywords. Then it calls \texttt{enterBlock()} again to bump the level from 1 to 2 for the globals. Only then does it start up the parser, which starts up the lexer. It creates an instance of the parser and then tells it to begin parsing. If anything goes wrong during parsing, then the \texttt{try} block fails and turns control over to the \texttt{catch} block, which prints an error message.

After the \texttt{catch} block, note the call to the \texttt{SymbolTable} class function \texttt{idDump()}. We will get back to that shortly.

Now we will build the program. First, use JLex to process \texttt{tokens.lex}, producing \texttt{tokens.lex.java} and copy it to the file \texttt{Yylex.java}.

Next, we must use CUP to process the file \texttt{grammar.cup}. Type the command

\texttt{$ java java\_cup.Main -expect 4 < grammar\_cup}$
This produces the two files `parser.java` and `sym.java`.

It is important that the files `Yylex.java`, `parser.java`, and `sym.java` are all created before any of them are compiled since they depend on each other.

Now we should compile all of our Java files. Compile `Yylex.java`, `parser.java`, `sym.java`, `IdEntry.java`, `SymbolTable.java`, `Ops.java`, `SemanticAction.java`, and `TableBuilder.java`. Do you see the benefit of using makefiles?

Run the program by typing

```
$ java TableBuilder
```

The first thing that appears is some messages informing us what the block level is and that keywords have been added to the symbol table.

Now, one by one, type the following lines and note the output.

```java
int a;
int b;
int main()
{
    a = b + 2;
    b = a*(a - 1);
}
```

These lines are all correct, so there should be no error messages. Now enter some lines containing undeclared variables and redeclared variables.

```java
int a;
int b;
int a;
int main()
{
    c = a + b;
}
```

Were error messages printed? Why not?
7.10 Semantic Actions

Now we will add semantic actions associated with the symbol table. At the present time, the action associated with each production is to print the production being applied, for debugging purposes. In the cases of identifiers, let's enhance the output by printing the value of the identifier. First, we must declare the ID terminal to be of String type. Do that in the terminal statements. Change the line to read

\[ \text{terminal String ID;} \]

While you are at it, in the same manner declare the nonterminal type to be of Integer type. We will need that soon.

In the productions listed above in which ID or type appear on the right side, introduce a variable associated with them. For IDs, this variable will automatically take on the value that was placed in the Symbol object by the lexer. For example, for the ID terminal, you should replace ID with ID:i. Now i is a String variable whose value is the value of the ID token. Do a similar thing with type. We will see shortly that nonterminals, such as type, get their values from their productions. These values may now be used in the semantic actions.

With the production

\[ \text{type ::= INT} \]

add the action

\[ \text{RESULT = new Integer(Ops.INT);} \]

The value assigned to RESULT is automatically passed to the nonterminal on the left side of the production. Therefore, type will take on the value Ops.INT, which is an Integer. That is why we declared type to be an Integer in the nonterminal statement above.

With the production

\[ \text{dcl ::= type:t ID:i SEMI} \]

add the action

\[ \text{SemanticAction.dcl(i, t);} \]
Do not remove the output statement that is already there. Note that i and t, the values of ID and type, are passed as parameters.

The actions of these two productions differ in two important ways. First, one calls a semantic action function and the other does not. Second, one returns a value through RESULT and the other does not. We will see numerous examples of each type. Generally, we will write a special semantic action function if the action is at all complicated. Otherwise, our CUP file would be too hard to read.

We see that the RESULT mechanism allows the attribute Ops.INT to be inherited by the terminal ID from the nonterminal type. (The actual transfer takes place in the dcl() function.) This is how CUP is able to handle inherited and synthesized attributes.

Finally, with the production

\[
\text{lval ::= ID:i}
\]

add the action

\[
\text{SemanticAction.id(i);}
\]

Also, in the output statement in that action part, have it print the name of the identifier.

Rebuild the program and run TableBuilder again, using the same input as above, including the errors:

```c
int a;
int b;
int a;
int main()
{
    a = b + 2;
    b = a*(a - 1);
    c = a + b;
}
```

Read the output and be sure that you understand it all.
7.11 Handling Syntax Errors

What happens if a grammar rule is violated? Let’s try a couple of examples to find out. First, let’s use an invalid token. Run the program and enter

```c
int a;
int b;
int main()
{
    a = b # 2;
}
```

Second, let’s enter a line with correct tokens, but with invalid syntax. Run the program and enter

```c
int a;
int b;
int main()
{
    (a + b) = 2;
}
```

Notice three things about the output. First, a message

```
Syntax error
      Couldn't repair and continue parse
```

appears. This appears nowhere in our code, so it must have been generated by CUP. (It was.) The second error message

**Error: (line 5) Syntax error**

which we should recognize as the one we wrote in the `catch` block in the main function of the `TableBuilder` class. Finally, note that the function `idDump()` was called, even though there was a syntax error. This indicates that the `try/catch` blocks work as expected, preventing the program from crashing.
7.12 Assignment

The SymbolTable class function idDump() currently does nothing except print a line. Its purpose is to display all the contents of the symbol table. There is a Hashtable member function elements() that can be used to display the elements of the hash table. It returns an Enumeration object. Look up the Hashtable class and the Enumeration interface on the Java website, read about them, and figure out how to use them to display the contents of all levels of the symbol table. In the section on the Enumeration interface, there is an excellent example that shows how to print the values contained in a vector.

Zip the files tokens.lex, grammar.cup, IdEntry.java, SymbolTable.java, Ops.java, SemanticAction.java, TableBuilder.java, and Makefile in a folder named Lab_07 and drop it in the dropbox.
Laboratory 8

The Abstract Syntax Tree

Key Concepts

• Abstract syntax trees (AST)

• l-values and r-values

• Dereferencing l-values

• Trees and tree nodes

• The mode of an expression

• Inherited and synthesized attributes

Before the Lab

Read Sections 5.2 and 5.6 of Compilers: Principles, Techniques, and Tools. Section 5.2 introduces the basics of abstract syntax trees. Section 5.6 may be tough going, but that material will be explained further in later labs. Try to get what you can from it now.

Preliminaries

Copy the folder Lab_08 from the Compiler Design CD to your folder. Copy the files IdEntry.java, Ops.java, SymbolTable.java, tokens.lex, grammar.cup, Err.java, and Warning.java from your Lab_07 folder to this folder.
8.1 Introduction

The purpose of this lab is to create and print an abstract syntax tree for a C program. The C program will use only a small subset of the grammar we introduced in Project 2.

As an example of a syntax tree, consider the statement

\[ \text{tri\_area} = (\text{base} \ast \text{height})/2; \]

The root node is an assignment operation. Its left subtree is a pointer to \text{tri\_area}. Its right subtree represents the expression \((\text{base} \ast \text{height})/2\). The tree looks like the tree in Figure 8.1.

The program TreeBuilder.java in this lab will display it in the form

\begin{verbatim}
ASSIGN INT
  ID PTR\|INT value = "tri\_area"
\end{verbatim}
8.2 Pointers, $l$-values, and $r$-values

While we will not formally introduce pointers into our C programs, we should be aware that ordinary variables are pointers in the sense that in a machine instruction they hold the address of the value rather than the value itself. In a machine instruction, it is possible to use the value instead of its address only if the value is a constant, in which case it is built into the instruction as an immediate operand.

That is why the variables must be dereferenced if their values are to be used. It is also the difference between an $l$-value and an $r$-value. An $l$-value is an address to which a value may be assigned. An $r$-value must be dereferenced to produce its value. When the value is assigned to an $l$-value, it is stored at the address of the $l$-value. The previous value of the $l$-value is irrelevant, so it makes no appearance in the code. That is, there is no need to dereference an $l$-value.

That is why, in the above example, `tri_area` is not dereferenced, but `base` and `height` are.

8.3 Tree Nodes and the TreeNode Class

A tree node will be implemented by the `TreeNode` class. If a tree node is an interior node, then it will contain an operator that acts on the left and right subtrees. The operator will have a mode, which will be the data type involved in the operation. For example, if the mode of an assignment operator is `INT`, then the operator will assign an `int` to an `int`. If a tree node is an exterior (leaf) node, then it will contain an
object, which will be an identifier or a number (and later a string). The mode of
an exterior node will be the kind of object stored in that node. For example, if the
object is an integer variable (l-value), then the mode will be a pointer to an INT. If
the object is an integer constant, then the mode will be INT.

Open the file TreeNode.java. This file defines the TreeNode class whose objects
have the following attributes: the operation (oper) represented by the node, the
mode (mode) of the operation, a reference to the left subtree (left), a reference to
the right subtree (right), the identifier (id) represented by the node, the number
(num) represented by the node, and the string (str) represented by the node.

If the node is a binary interior node, then left and right will be non-null, and
id, num, and str will be undefined. On the other hand, if the node is an exterior
node, then left and right will be null, while exactly one of id, num, and str will
be defined, depending on the kind of exterior node. From time to time, we will have
unary interior nodes. They will always use the left subtree rather than the right
subtree.

Note the types of the data members oper, mode, left, right, id, num, and str.

Also, one constructor

public TreeNode(IdEntry i)

and the toString() function have been defined. You will define three additional
constructors. First, define the default constructor:

public TreeNode()

It should set oper, mode, and num to 0 and left, right, id, and str to null.

Next, define the following constructor:

public TreeNode(int op, int m, TreeNode l, TreeNode r)

The purpose of this constructor is to join together two existing trees, with root nodes
l and r, as the left and right subtrees of a new tree with this node as its root node.
In the root node, the value of oper should be op and the value of mode should be m.

Finally, define the constructor

public TreeNode(int n)

It will create a node that represents a number. The member oper should be Ops.NUM,
mode should be Ops.INT, and num should be the value of n.

Write these constructors. We will use these constructors later in this lab.
8.4 The Ops Class

Open the file Ops.java. The Ops class does nothing but define a number of constants. It is similar to the file sym.java, except that the sym class defines the tokens from the parser. The Ops class defines the operations that may appear at tree nodes. The order in which the operators are listed is arbitrary. As examples, the ERROR and ALLOC values have been defined as

    public static final int ERROR = 0;
    public static final int ALLOC = 1;

The entire list is

    ERROR
    ALLOC
    ID
    NUM
    PLUS
    MINUS
    TIMES
    DIVIDE
    MOD
    NEGATE
    ASSIGN
    DEREF
    READ
    PRINT

PRINT and READ represent our pseudo-keywords read and print. Type in the remaining operators, assigning them numerical values in sequence. Some of these are terminals and some are tokens, but this list is not the same as the list of terminals or the list of tokens. This is a list of operations that may appear in the syntax tree.

Two other groups of constants are defined. In the first group, each constant represents the scope of a variable: global, local, or parameter. The constants GLOBAL, LOCAL, and PARAM are defined with the values 0, 1, and 2, respectively.

The final group of constants represents the types of objects. Right now the only type is INT. The constant PTR is also defined so that an identifier can be a pointer to an int.
8.5 The TreeOps Class

Open the file TreeOps.java. This file will be used primarily in the Utility class, to be discussed soon.

The TreeOps class has three class constants LEAF_NODE, UNARY_NODE, and BINARY_NODE, representing various types of tree nodes: leaf nodes (no children), unary nodes (one child), and binary nodes (two children).

A TreeOps object has three data members:

```java
int opcode;
String opstring;
int kind;
```

The integer `opcode` is a value from the list defined in Ops.java. The string `opstring` is a readable string version of the opcode. It will be used when the opcode needs to be printed in a readable form. The integer `kind` will store the kind of node: leaf, unary, or binary. This class creates the data type, but does not instantiate any objects. That will be done next in the Utility class in the array `opInfo[]`.

The TreeOps class has a constructor and two functions that return information about an identifier. The function `baseType()` returns the fundamental data type of an identifier, e.g., INT. Later it will also return the type DOUBLE.

The function `baseSize()` returns the size, in bytes, of the fundamental data type. The size of an INT is 4. Later, it will also return 8 for the type DOUBLE.

Take a minute to look at these two functions to see how they work. Note especially how the binary “and” operator & is used to mask out the pointer bit, leaving only the bit for the base type.

8.6 The Utility Class

Now open the file Utility.java. At the beginning of the class, just before the first `printTree()` function, define an array of TreeOps objects named `opInfo`. Each element should be a TreeOps object whose values are the opcode (as defined in Ops.java), the string equivalent of the operator’s name (matching the constant names in Ops.java), and an integer representing the kind of node (LEAF_NODE, UNARY_NODE, or BINARY_NODE).

For example, the array members for ERROR and ALLOC are
new TreeOps(Ops.ERROR, "ERROR", 0),
new TreeOps(OpsALLOC, "ALLOC", TreeOps.BINARY_NODE)

The constant ERROR is a special case, since it does not represent an actual tree node; the constant ALLOC is more typical.

Add entries for the remaining node types that were listed in the Ops class.

8.7 Printing the Abstract Syntax Tree

Once a statement has been completely parsed, the syntax tree will be printed. This is also handled by the Utility class.

Each tree will be printed recursively. After all, a binary tree is a naturally recursive structure. First, we need a function printTree() that gets the process started. It takes one parameter: a TreeNode object. A second printTree() function makes recursive calls to itself. It takes two parameters: a TreeNode object and the indentation level. Thus, the nonrecursive printTree() function should make the call

    printTree(t, 0);

to the recursive printTree() function. When the recursive printTree() function makes a recursive call, the indentation level will be increased by 1.

The recursive printTree() function is fairly simple. Its basic structure is exactly what you would expect of a recursive function that traverses a binary tree. It first checks whether the node is null, to be safe. Then it prints the current node, using the function printNode(), to be discussed shortly. It increases the indentation level and then makes the recursive calls. If the node is a binary node, it makes calls for the left and right subtrees, in that order. If the node is a unary node, it makes a call only for the left subtree. If the node is a leaf node, it makes no recursive call.

The function printNode() receives a TreeNode object and prints one line of output for that node. It prints the opcode (string version, found in the opInfo[] array) and the mode. For example, if the node represents an integer variable, the function should print

    ID PTR|INT

There are two special cases concerning leaf nodes. Depending on whether the node is an identifier node or a number node, we should also print the value of the identifier
or the number. For example, nodes for the integer variable count and the number 123 would be displayed as

\begin{verbatim}
ID PTR|INT value = "count"
NUM INT value = 123
\end{verbatim}

respectively.

I have included a function printType() that will print the data mode of a node, given the type data member of the TreeNode. Notice how it uses bitwise operators to test the bits of the integer type.

### 8.8 Semantic Actions

In the previous lab, we provided semantic actions for those productions in grammar.cup that involved declarations and the symbol table. Now we will add semantic actions for a few more productions.

Open the file grammar.cup. Every production should have an output statement that displays the production. These are enormously helpful in debugging the compiler. Once we are satisfied that things are working properly, we should comment out the output statements. Do not remove them. We may need to uncomment them later as we add more productions to the grammar and new errors occur.

Now open the file SemanticAction.java. We will add seven new functions:

- arith()
- assign()
- deref()
- mod()
- negate()
- print()
- read()

The function arith() covers addition, subtraction, multiplication, and division. Since the tree has exactly the same form in each of these operations, only one function is needed for them. The function mod() handles the mod operator %. It is different from the other arithmetic operators since it applies only to ints. The function assign() handles assignments. The function deref() will dereference a variable. The function negate() handles the operation of negating an expression.
We will provide two other functions, \texttt{read()} and \texttt{print()}, that will read and print, respectively, an integer variable. When we generate code, these two functions will allow us to input and output values in order to test our programs better.

The purpose of each function is to build its part of the tree, printing the tree as necessary. We will discuss how to write each of the eight functions listed above. We will begin with a case so simple that it doesn’t require a function.

\section*{8.9 The NUM Case}

The token \texttt{num()} should cause a \texttt{NUM} tree to be constructed. The form of a \texttt{NUM} tree is shown in Figure 8.2.

The mode must be \texttt{Ops.INT}.

We will modify the grammar accordingly. In the file \texttt{grammar.cup}, declare the nonterminal \texttt{expr} to be of the \texttt{TreeNode} type and the terminal \texttt{NUM} to be of type \texttt{String}. Then modify the action part of the production

\texttt{expr ::= NUM:n}

\begin{verbatim}
to be
RESULT = new TreeNode(Integer.parseInt(n));
\end{verbatim}

Now the \texttt{TreeNode} that is returned by \texttt{RESULT} will be assigned to the \texttt{expr} nonterminal.

To test this, let’s make a modification. We can restore it to the above form once we are sure that it is working. Replace the above statement with the following sequence of statements:

\begin{verbatim}
TreeNode t = new TreeNode(Integer.parseInt(n));
Utility.printTree(t);
RESULT = t;
\end{verbatim}
Clearly, we are just inserting a call to printTree() in order to see the result.

Now test this by building TreeBuilder and running it with the test file testfile.c. This file contains a simple (and pointless) C program that uses each of the syntactic structures covered in this lab. At this point, TreeBuilder should print a NUM tree for each integer constant that it encounters in the program.

As you add the other functions to SemanticAction.java, pause after each one, build TreeBuilder, and test it on testfile.c. When this lab is done, be sure to remove all the extraneous calls to Utility.printTree().

8.10 The id() Function

We must revisit the id() function because now it must return a tree containing a single ID node. Currently, the id() function looks up the identifier in the symbol table. If it is not there, it prints an error message and then installs the identifier as an integer at the current level.

Now we need id() to create an ID TreeNode and return a reference to it. Make the following changes:

- Change the return type from void to TreeNode.
- Just before returning, add the statement.

        TreeNode t = new TreeNode(p);

This will create a TreeNode from the symbol table entry p (an IdEntry object) that was returned by idLookup(). Look in the file TreeNode.java to see what this TreeNode constructor does. In particular, note that the mode is set to

        i.type | Ops.PTR

For the time being, i.type is Ops.INT. Look in the file Ops.java to see the values of Ops.INT and Ops.PTR. What value do we get when we “or” them together? This is our method of recording the fact that the mode of the node is “pointer to int.”

Now continue by doing the following.

- Add a call to printTree() to print the ID tree.
- Modify the return statement so that it returns t.
• In grammar.cup, make the nonterminal lval of the TreeNode type.

• In the production

\[
\text{lval} ::=: \text{ID}:i
\]

change the action to

\[
\text{RESULT} = \text{SemanticAction.id}(i);
\]

Test the id() function by running TreeBuilder with the testfile.

### 8.11 The dcl() Function

We also need to revisit the dcl() function. The change here is a little more substantial. The purpose of a declaration is to allocate memory for an object, so this function must build an allocation tree as shown in Figure 8.3.

This is our first real “action” tree, i.e., a tree whose root node represents an operation. The number in the right subtree is the number of bytes of memory required by the identifier in the left subtree. The mode of the ID node is the type of the identifier, as seen in the declaration statement, “or”-ed with Ops.PTR to make it a pointer to that type. On the other hand, the mode of the ALLOC node is simply the type of the identifier. The statements needed to build this tree are

```java
TreeNode t1 = new TreeNode(id);
TreeNode t2 = new TreeNode(TreeOps.baseSize(id.type));
TreeNode t = new TreeNode(Ops.ALLOC, id.type, t1, t2);
```
The first tree $t_1$ represents the identifier. The second tree $t_2$ represents the number of bytes of memory required by the identifier. Note that its `num` data member is set to the size of the base type of the identifier. Then an allocation tree is created with $t_1$ and $t_2$ as its subtrees. The `IdEntry` object should be returned by `dcl()`.

The return type of `dcl()` should now be `IdEntry`. Make these changes. In the CUP file, make the nonterminal `dcl` of the `IdEntry` type. Also, change the action associated with the production

$$dcl ::= type:t ~ ID:i ~ SEMI$$

so that it assigns the value returned by `dcl()` to the nonterminal `dcl`.

There is one more thing `dcl()` should do before returning. It should print the tree. The statement to do this is

```
Utility.printTree(t);
```

Add this statement to `dcl()` and then test it using the testfile.

### 8.12 The deref() Function

Recall that an $l$-value must be dereferenced before it can be used as an $r$-value. The `deref()` function receives a tree that represents an expression and creates a tree with the dereference operator at the root node and the expression as its left subtree. See Figure 8.4.

The `DEREF` operator should be applied only to an identifier. (In C it could also be applied to an array element or an expression such as $++n$, but those are not in
our grammar.) The mode of the DEREF node is the base type of the identifier. If the identifier’s mode is PTR|INT, then the mode of the DEREF node is INT.

Write the deref() function. The deref() function should return a TreeNode. Make the appropriate changes in the grammar file. This is accomplished by introducing the variable v to represent lval in the production

\[
\text{expr ::= lval}
\]

and then adding the action

\[
\text{RESULT = deref(v);
}
\]
to the production. Add a printTree() statement and test the function with the testfile.

8.13 Parenthesized Expressions

The action for the production

\[
\text{expr ::= LPAREN expr RPAREN}
\]

is trivial. The tree representing the expression tree on the right is simply passed on to the expression on the left. Write the action for this production that will do that. No special semantic action function is necessary.

8.14 The assign() Function

Next, we will write the assign() function. It receives two trees as parameters. The first represents the destination variable. The second represents the value to be assigned. This function should create one tree whose root node is an assignment operator whose left subtree is the destination variable and whose right subtree is the value. See Figure 8.5.

Again, using a TreeNode constructor, this is very easy. We must construct a TreeNode with Ops.ASSIGN at the root. Its mode is the base type of the variable on the left, not the expression on the right. Use the function baseType() to extract the base type that is being assigned to v. Note that it is a synthesized attribute of the assignment node. The function should return the new tree.
Right now, all objects are ints, so e is an int and v is a pointer to an int. Later, when we introduce doubles, it will be possible that e will be a double and v will be a pointer to an int, or e will be an int and v will be a pointer to a double.

Make the corresponding changes in the CUP file to the production

\[ \text{expr} ::= \text{lval ASSIGN expr} \]

### 8.15 The `arith()` Function

Now we will write the `arith()` function. It has three parameters. The first is the integer constant representing the operation; the other two are trees representing expressions. We need to create a single tree with these two trees as its left and right subtrees and the arithmetic operator at the root node, as shown in Figure 8.6.

Using a `TreeNode` constructor, this is very easy. Write the function and modify the grammar so that the tree node returned by `arith()` is assigned to the nonterminal `expr`. 
Later we will have to allow that the mode may be `Ops.DOUBLE`.

### 8.16 Assignment

Complete the set of functions by writing `mod()`, `negate()`, `print()`, and `read()`. Remove the calls to `printTree()` from all functions except `dcl()`, `print()`, and `read()`. In the action associated with the production

\[
\text{stmt ::= expr\:e SEMI}
\]

add the action

```
Utility.printTree(e);
```

because this marks the final stage in the development of that tree.

The functions `negate()` and `mod()` are similar to `arith()` and `deref()`. They are arithmetic, but `negate()` has only a left subtree. You should be able to write them without much trouble. The function `mod()` should create the tree shown in Figure 8.7 and the function `negate()` should create the tree in Figure 8.8.

The functions `print()` and `read()` each have an argument that represents an identifier. They should build the trees appearing in Figure 8.9.

In both cases, the mode of the root node is the base type of the identifier node. Each of these two functions should print the tree.

Throughout all of these functions, use the base type wherever appropriate. Also, be sure to dereference `l`-values whenever necessary.

This lab will serve as Project 3. Zip the files `tokens.lex`, `grammar.cup`, `Err.java`, `Warning.java`, `Id.java`, `Ops.java`, `SemanticAction.java`, `SymbolTable.java`, `TreeBuilder.java`,
Figure 8.8: A negate tree

Figure 8.9: Print and read trees
TreeNode.java, TreeOps.java, and Utility.java in a folder named Project_3 and drop it in the drop box.
Laboratory 9

Code Generation

Key Concepts

- Command-line arguments
- Post-order traversals
- Stack operations
- Addressing modes
- The gnu assembler
- `printf()` and `scanf()`

Before the Lab


Preliminaries

Copy all of the files in your Lab_08 folder into a folder named Lab_09. Copy the files in the folder Lab_09 from the Compiler Design CD to your Lab_09 folder. This will replace the makefile and will add the files `CodeGen.java`, `compiler_v1.java`, and the `NewPrintTreeFunc.java`.
9.1 Introduction

In this lab we will finally create a working compiler. Our program, `compiler_v1.java`, will compile simple C programs into assembly code. If we invoke the gnu assembler, we will obtain an executable program.

The subset of C that is handled by this compiler is very limited. All variables must be global and they should be declared before `main()`. The program has exactly one function, `main()`. We will implement our special `read` and `print` statements so that our program can read and print integers. Therefore, a program that will print the average of two integers should be written like this:

```c
int a;
int b;
int average;

int main()
{
    read a;
    read b;
    average = (a + b)/2;
    print average;
}
```

The input and output of the program would appear as

```
Enter a: 8
Enter b: 16
average = 12
```

9.2 The `compiler_v1` Class

We will invoke our compiler by running the program `compiler_v1`. This program expects command-line arguments. The argument `-t` means to output the abstract syntax tree. The argument `-c` means to output the assembly code. If we use both arguments, we will get both outputs. That can be extremely useful when debugging.

For example, if we wanted to compile the program `average.c`, we would type
or if we wanted to compile the program and print the tree, we would type

$ java compiler_v1 -t -c < average.c

The order of -t and -c does not matter, provided they appear after the name of the compiler and before the name of the source file.

How does compiler_v1 know if there are command-line arguments? Open the file compiler_v1.java. At the beginning of the main() function there is a for loop that uses args.length. The object arg is an array of Strings and it appears as the parameter of the main() function. When a command is typed, the operating system passes all command-line arguments as members of the String array arg. This idea was taken from C, where the prototype of main() is

```java
int main(int argc, char** argv);
```

In Java, arrays automatically come with a public member length. Thus, args.length is the number of Strings in the array.

The for loop compares each argument to "-t" and to "-c". If there is a match, then the corresponding boolean variable tree or code is set to true, for future reference. Later, when printTree() is called, the Utility class checks the values of tree and code to decide what to output. Open Utility.java, look at the printTree() function, and note that it uses compiler_v1.code to decide whether to call CodeGenerator.generateTreeCode() and that it uses compiler_v1.tree to decide whether to print the abstract syntax tree.

### 9.3 The Code Generator

The grammar and the semantic actions in this lab are the same as in Lab 8. What we have added is a file named CodeGenerator.java. The CodeGenerator class contains the functions that write the assembly code. Open the file CodeGenerator.java. The primary function is generateCode(). It is called from other classes and it in turn calls traverseTree(). The function traverseTree() will generate assembly code for an entire abstract syntax tree. Since a tree is a recursive structure, this function is recursive. It first generates code for the left subtree, then it generates code for the
right subtree, then it generates code for the node. In other words, it performs a post-order traversal of the tree, writing the code for each node. That is very important to keep in mind as we consider how to write the assembly code.

Look at the function `generateCode()` and see that it distinguishes one special case: `ALLOC`. This special case is not recursive, so we do not call `traverseTree()`. In the future there will be a few more special cases. When a case is handled recursively, at each level of recursion, the result of that operation is left on the stack to be used at the next level up. That means that at the root node, the value will also be left on the stack. This could be a problem in a for loop such as

```
for (i = 0; i < 1000000; i++)
    a = a + 1;
```

since the 1000000 values assigned to `a` will be left on the stack, causing stack overflow. Therefore, the last action in `traverseTree()` is to pop the final value of the tree to avoid this problem.

Now look at the function `generateNodeCode()`. This is the function that writes the assembly code for a single node. At this point, we have thirteen different types of node (plus the error node). Let us look at each of these thirteen. I have arranged them in alphabetical order in the program order to facilitate locating them. I suggest that you maintain them in alphabetical order as we add more cases later.

### 9.4 Allocation

To allocate memory for a global, we write the assembly directive `.comm`, for common, followed by the name of the global, followed by the number of bytes to be allocated. The necessary information has all been stored in the tree, which makes it very simple to write the assembly code. Pay careful attention to the way in which the name was retrieved from the `ID` node in the syntax tree and the way in which the number was retrieved from the `NUM` node. We will often have occasion to do that sort of thing in the future.

For example, declaring the global `a` to be an `int` would be written in assembly code as

```
.comm a,4
```
This assembler directive will allocate 4 bytes of memory and associate the name `a` with the address of this memory.

Notice that we print the line

```
# Code for ALLOC
```

This will be interpreted as a comment by the assembler, since it begins with `#`. It is very helpful to have these comments written into the assembly code when we are trying to debug it.

### 9.5 Identifier Nodes

The purpose of an identifier node is simply to load an address and push it onto the stack. The following code does this.

```
# Code for ID, global = name
lea name,%eax  # Load address of global
push %eax     # Push address
```

where `name` is the name of the global identifier. Notice that we use the mnemonic `lea` (load effective address) instead of `mov` (move). Had we written

```
mov name,%eax  # Move value to eax
```

the effect would have been to move the value stored at `name` into register `eax`. That is not what we want. However, if the value of `name` is needed, then the `lea` operation will be followed by a dereference operation.

This example and the previous one make it clear that we must always be aware of the exact effect of the assembly instruction. Are we operating on the value in the register or on the value pointed to by the address in the register? Are we operating on the address itself or on the value stored at the address? These distinctions are very important.

### 9.6 The Dereference Operation

In a dereference tree, the source address is stored in the left subtree; there is no right subtree. The address should have already been pushed onto the stack. Therefore, the assembly code for `DEREF` must first pop the source address, then push the value stored at that address. The following two instructions will do this.
# Code for DEREF
    pop %eax  # Pop address
    push (%eax)  # Push value at address

Notice that the parentheses are used to indicate indirect addressing. That is, (%eax) is interpreted as “the value stored at the address in eax,” whereas %eax means “the value stored in eax.”

Write the code for the DEREF case.

9.7 Testing the Compiler

In this lab, we will implement the operators in a logical order that allows us to test our work as we go along. The print and read statements have already been implemented to facilitate the testing of the other statements. Therefore, we can begin testing right now in order to learn the test procedure and to see if our operations work so far.

Write the following program and save it as test1.c.

```c
int a;

int main()
{
    read a;
    print a;
}
```

Now build the compiler using the makefile. Then type the command

```bash
$ java compiler_v1 -c < test1.c > test1.s
```

This will read the C program from test1.c and write the assembly program to test1.s. Open the file test1.s to see what is there. (The extension .s is used for assembly-language files.) If a lot of trace information was output to the file, then go back into those files and comment those statements out. Do not remove them! They will be very useful later on when we have to debug.

Now we will assemble the file test1.s. Type

```bash
$ gcc -o test1.exe test1.s
```
The command gcc invokes the gnu C compiler. (The UNIX command is cc.) The -o option means “output file,” and it designates test1.exe as the name of the output file. If there were no error messages, then we now have an executable program test1.exe. Let’s test it. The command to run test1.exe is

$ test1

However, the system does not know to look in this folder for executables. Therefore, we must modify the PATH environment variable. Open the System control panel and add the path “.” to the PATH variable. Recall that the dot means “the current directory.” Also, it might be good to add the “.” at the beginning of the list of paths rather than at the end so that if there is another executable out there with the same name, it will find the one in this folder first. Close the Cygwin window and open a new one so that this change will take effect.

Now run the program. The program should prompt you to enter a value for a. Then it prints the value that you entered. We will follow this procedure at various points in the lab, whenever we wish to test our work so far. But we must be sure that our test program contains only operators that we have implemented.

Let us pause for a moment to fully take in what just happened. You just executed a program whose code was produced by your compiler. That calls for a moment of silent reflection. Reflect on the major milestones in your life: birth, marriage, death, and the day you wrote a compiler that produced executable code. When you feel ready, continue with the next section.

9.8 Numbers

We are now empowered. Let us push on. A number is an immediate value that should be pushed onto the stack. The code to do this is

```c
    # Code for NUM
    push $number  # Push number onto stack
```

where number is a specific value. Note the use of $ to indicate an immediate value. The value of number is stored in the node and must be written into this statement. You will have to use the num field of the appropriate TreeNode object to get the value of number. Write the code in the NUM case.

Before we can test this, we will need to encode assignment statements.
9.9 The Assignment Operator

As the value of each subtree is computed, it is pushed onto the runtime stack. It is the responsibility of the next tree node to pop it off the stack in order to use it. At an assignment node, we have the destination variable in the left subtree and the expression whose value is to be assigned to it in the right subtree. Therefore, we may assume that the left subtree, an ID node, has produced code that will push the address of the variable onto the stack. Similarly, the right subtree has pushed the value of the expression onto the stack. Since the right subtree is processed after the left subtree, the value will be on top of the stack, with the address of the variable just under it.

Thus, the assignment node should

- Pop the value from the stack into a register.
- Pop the address of the variable from the stack into a register.
- Assign the value to the address of the variable.
- Push the value onto the stack

The assembly code should look like this:

```
# Code for ASSIGN
pop    %eax   # Pop value to be assigned
pop    %edx   # Pop destination address
mov    %eax,(%edx) # Move value to destination address
push   %eax   # Push value onto stack
```

Enter this code into the ASSIGN case of the `generateNodeCode()` function. This example should set the pattern for many of the following operations.

Notice that I have included in-line comments for each assembly instruction. It is worth your time to type these comments since they will be an enormous help when you are trying to understand or debug the compiler-generated assembly code.

Now we can test a program with simple assignment statements that assign numbers and variables to variables. Be sure to rebuild your compiler before trying a test program.

Create a file test2.c that contains the following C program.
int a;
int b;

int main()
{
    a = 123;
    b = a;
    print a;
    print b;
}

Compile this program (using compiler_v1, not gcc!) and then assemble test2.s (using gcc) and run test2.exe.

9.10 The Addition Instruction

An addition node needs to add the values of the left and right subtrees and push the sum onto the stack. Since the left and right subtrees have already been evaluated, their values should be the top two values on the stack. Thus, we should pop them, add them, and push the sum.

Look up the add operator in the Intel Developer’s Manual, Vol. 2A. Write the assembly code to pop the right operand into register eax, pop the left operand into register edx, add edx to eax, and push the value in eax onto the stack. Be sure to include the comment

    # Code for PLUS

and write appropriate in-line comments for each line. For example, you might comment “Pop right operand,” “Pop left operand,” “Add values,” and “Push result.” Be brief, but informative.

Be sure to rebuild your compiler.

Create a file test3.c containing the C program:

    int a;
    int b;
    int sum;
int main()
{
    read a;
    read b;
    sum = a + b;
    print sum;
}

Test this program. If it works, then test other addition statements, such as

    sum = a + b + 2;

and

    sum = a + (b + 2);

9.11 The Subtraction Instruction

Subtraction is very similar to addition. Check the Intel Developer’s Manual, Vol. 2B, for details on the sub instruction and write the assembly code for MINUS. You must be careful with MINUS since subtraction is not commutative. That is, if the C statement says a - b, then you must be careful to subtract b from a, not a from b. Write the code for a subtraction node.

Create a test program test4.c that includes a subtraction operator. If it works, then try statements such as

    diff = a - b - 2;

and

    diff = a - (b - 2);

to see if subtraction is left associative.

9.12 The Negation Instruction

A negation node needs to reverse the sign of the value on the stack and return it to the stack. The neg opcode will negate an integer, so this is very simple. Look up neg in the Intel Developer’s Manual, Vol. 2B, and write the code for the negation
operator. Write a test program test5.c to test this operator. Test the precedence of negation by combining it with other operators. For example, you might try

\[ a = -b + 2; \]

to verify that the operator is applied only to \( b \), not \( b + 2 \).

### 9.13 The Multiplication Instruction

Multiplication is tricky because the product, in general, occupies about twice as many bytes as the two factors. For example, the product of two 2-byte integers is, in general, a 4-byte integer and the product of two 4-byte integers is, in general, an 8-byte integer. The `imul` opcode is for multiplication of signed integers. (Be careful not to use `mul`, which is for multiplication of unsigned integers.) It has various forms, but the simplest one,

\[ \text{imul register} \]

is designed to multiply the accumulator `eax` by the value in the specified register. The product is stored in the 64-bit register `pair edx:eax`, with the high-order 4 bytes of the product stored in `edx` and the low-order 4 bytes stored in `eax`. In our compiler, we will assume that the product of any two 4-byte integers is another 4-byte integer, or else the result will not be mathematically correct. (The value will “wrap around” from \( 2^{32} - 1 \) to \( -2^{32} \).) Therefore, the assembly code is

```
# Code for TIMES
pop %eax   # Pop right factor
pop %ecx   # Pop left factor
imul %ecx  # Calculate product
push %eax  # Push product
```

Add this code to the TIMES case. Then create, compile, and run a program test6.c that will test multiplication. Be sure to test the associativity and precedence of multiplication over addition and subtraction.

### 9.14 The Division Instruction

Division is similar to multiplication. Read about the `idiv` operator in the Intel Developer’s Manual, Vol. 2A. If we use the simple form
idiv register
	hen the divisor is assumed to be in the specified register and the dividend is in the 64-bit register pair edx:eax. The division produces both a quotient and a remainder. Read the manual to see exactly where they are stored.

You must be a little careful with the register pair edx:eax. Our compiler assumes that the dividend is only 32 bits, even though edx:eax is 64 bits. Therefore, you should load the dividend into register eax. However, it is not enough simply to clear register edx. If the value in eax is negative, then the sign must be extended through edx. This will require an instruction that converts a doubleword to a quadword. Find the convert-doubleword-to-quadword instruction in the Intel Developer’s Manual. When you write the assembly code, after loading the divisor and the dividend, convert the doubleword eax to the quadword edx:eax, then divide and push the quotient.

Write a test program test7.c. Be sure to check that the division was done in the right order. That is, the expression a/b should divide a by b, not b by a. Also, test the associativity and precedence of division.

9.15 The Mod Operator

The mod operator is similar to division, except that we want to keep the remainder, not the quotient. Write the code for the mod operator. Then write a test program test8.c that tests the mod operator. Be sure to include statements that test the associativity and precedence of mod.

9.16 print and read Statements

In C programs, input is performed by calling the scanf() function and output is performed by calling the printf() function. We have not incorporated function calls into our compiler yet, so we cannot handle calls to scanf() or printf() in our C source. Yet we would like to read and print integers. Therefore, we have introduced the print and read statements into our compiler just so that we do that. The statements

read a;

and
print a;

will generate the assembly code necessary to make function calls to `scanf()` and `printf()`.

The form of the `scanf()` function call is

```
scanf("format", &var1, ..., &varn);
```

where `var1, ..., varn` is a list of variable names. The string `format` contains `%d` to read an `int`, `%c` to read a `char`, `%s` to read a string, and `%f` to read a `float`. For example, if you wanted to read two `ints` and a `float` (in that order), then `format` would be "%d%d%f". We will use only `%d`. Furthermore, since our read statements read only one variable, the format string parameter will be followed by just one variable parameter.

A read tree has READ at the root and an ID node as the left subtree. When the left subtree is evaluated, the address of the variable is pushed onto the stack.

The format string and the address of the variable must be passed to the function as parameters. Parameter passing on the x86 is done by pushing the parameters onto the stack, in order from right to left, so that they will be popped by the function in the correct order, from left to right. Therefore, the address of the variable must be pushed first. But this was already done when the left subtree was evaluated. So we need only push the format string onto the stack. This is done by creating the string in memory and pushing its address.

The assembly code for the call

```
scanf("%d", &a);
```

would look like this:

```
# Code for READ
.data
L01: .asciz "%d"       # Format string
.text
lea L01,%eax           # Load address of format string
push %eax              # Push address of format string
call _scanf            # Call scanf
add $8,%esp            # Pop parameters
```
The assembler directive .asciz means a null-terminated ASCII string. (z = “zero” for the null character at the end of the string.) It is stored in this very location, so its address is given by the label L01.

The assembly code created for read also includes a call to printf() which will print a prompt.

The printf() function call is handled similarly, except the variables are passed by value and the format string may contain additional characters.

The assembly code for the call

\begin{verbatim}
printf("a = %d\n", a);
\end{verbatim}

would look like this:

\begin{verbatim}
# Code for PRINT
.data
L02: .asciz "a = %d\n" # Format string
.text
lea L02,%eax # Load address of format string
push %eax # Push address of format string
call _printf # Call printf
add $8,%esp # Pop parameters
\end{verbatim}

Again, the variable has already been dereferenced and its value pushed onto the stack when the left subtree was evaluated, so that does not need to be done here.

Soon we will incorporate function calls into our compiler (version 3), so you should try to understand what is going on here with scanf() and printf().

### 9.17 Assignment

Finish implementing all of the operations discussed above. The finished product will be turned in as Project 4. Put all of your source files in a folder named Project_4, zip it, and drop it in the dropbox. Congratulations! You have built a compiler!
Part V

Additional Data Types
Laboratory 10

Floating-Point Numbers and the Abstract Syntax Tree

Key Concept

• Casting types

Before the Lab


Preliminaries

Copy the all files from your Lab_09 folder into a new folder named Lab_10.

10.1 Introduction

We have been using only one data type in our compiler so far in order to keep it simple. Now we would like to introduce a second data type, double. We could introduce other types, such as char, float, and short, but that would only take more time, with little additional benefit. We will learn how different types are handled by working with just the two types int and double. This will create the potential for mixed expressions, including mixed assignment statements. In any situation where a particular type is expected, we will have to check the actual type. If it is not the expected type, then a type conversion will have to be performed or an error message will have to be printed.
In this lab, we will build only the syntax tree. Before we can write the assembly code, we must learn how the floating-point unit (FPU) works. All of that will be done in the next lab.

10.2 Version 2 of the Compiler

Change the name compiler_v1 to compiler_v2 throughout your files. This is now version 2.0 of our compiler. To find out in which files compiler_v1 occurs, use the grep command. It is designed to search files for text that matches a regular expression. The form of grep is

$grep pattern files

Use compiler_v1 for pattern and the wildcard * for files. Then make the change in the files listed.

10.3 Introducing the double Keyword

Make sure that the keyword double has been entered into level 1 of the symbol table so that the lexer will recognize it as a keyword, not an identifier.

In the grammar, the terminal DOUBLE must be included as a possible value of the type nonterminal. Make sure that the production

\[ type \rightarrow \text{DOUBLE} \]

is in the file grammar.cup. Include a semantic action similar to the semantic action for the production

\[ type \rightarrow \text{INT}. \]

In the file Ops.java, add the constant DOUBLE as

\[
\text{public static final int DOUBLE = 1 << 2;}
\]

This defines Ops.DOUBLE to be the integer with bit 1 set (binary 00000010). The constant Ops.INT is already defined to be the integer with bit 0 set (binary 00000001) and Ops.PTR is the integer with bit 5 set (binary 00100000). In SemanticAction.java, in the dcl() function, you will see that when a node for a variable is created, its type is set to
id.type | Ops.PTR

where id.type is Ops.INT or Ops.DOUBLE. Thus, for int variables this value is binary 00100001 and for double variables it is binary 00100010. Later, in the baseType() function, we wish to recover the value Ops.INT or Ops.DOUBLE from this value. We will do this by “and”-ing the type with the value Ops.INT | Ops.DOUBLE (binary 00000011), which produces 00000001 for pointers to ints and 00000010 for pointers to doubles, which are the values of Ops.INT and Ops.DOUBLE.

Therefore, the baseType() function in TreeOps.java should return the value

\[
\text{type} \& (\text{Ops.INT} | \text{Ops.DOUBLE})
\]

Make that change.

We must also change the functions baseSize() in TreeOps.java and printType() in Utility.java. The baseSize() function currently returns only 4 for ints. Modify it so that it will also return 8 for doubles. The printType() function must also be extended to handle doubles.

### 10.4 Semantic Actions

Most of the changes necessitated by the introduction of the double type will be in the file SemanticAction.java. These changes will be of two types. One type of change will be in mixed-mode expressions and assignments, where one type will be cast to the other type to make the types compatible. The other change will be to use the baseType() function when dereferencing a variable. Up to this point, we knew the base type was int, so we could just say Ops.INT. Now it could be int or it could be double, so we will have to call on baseType() to return the correct type.

Let us consider this file, function by function. First, let’s consider a function that needs no change.

### 10.5 The dcl() Function

No changes are necessary, but notice that when this function creates an allocation tree for a double, the number in the right subtree will be 8 instead of 4. That is because the baseSize() function will return the size of a double. (If your dcl() function does not use baseSize(), then make that change.)
10.6 The cast() Function

Recall that we are not yet actually changing types; we are only building a tree that represents the operation of changing types. Later, when we write the assembly code, we will see how types are actually changed. Thus, our job now is to create a CAST node that represents the change.

First, in the file Ops.java, create a new constant CAST. Then in Utility.java, add a new element

```
new TreeOps(Ops.CAST, "CAST", TreeOps.UNARY_NODE)
```

to the opInfo[] array. Note that it is a unary node.

Now we can write the cast() function. Given a TreeNode e of type e.mode and a type t to which it should be converted, we need to convert the tree in Figure 10.1 to the tree in Figure 10.2.

To do this, we create a new TreeNode with operator Ops.CAST, type t, and left subtree e. (The right subtree is null). The statement
new TreeNode(Ops.CAST, t, e, null)

will do that. However, what if the type of e is already the same as t? In that case, there is no need to create a new tree node. In other words, if e.mode equals t, then cast() should return e unchanged. Otherwise, it should return the new tree node created by the above statement.

The cast() function will be used many times by other functions. Whenever we need to cast a tree e to the type t, we call

cast(e, t)

and it will return the modified tree.

10.7 The arith() Function

The parameter list of this function is

(int op, TreeNode e1, TreeNode e2)

The type of the returned tree is Ops.INT unless either e1 or e2 is of type Ops.DOUBLE, in which case the returned tree is of type Ops.DOUBLE. Thus, set mode to Ops.INT or Ops.DOUBLE accordingly. Then the two subtrees must be cast as type of the returned tree. (Recall that cast() does nothing if the subtree is already of the appropriate type.) Thus, we should return the tree in Figure 10.3. The CAST nodes will appear only if they are necessary. Add this to the arith() function.

10.8 The assign() Function

With an assignment operator, the r-value being assigned must be cast to the base type of the l-value on the left. Thus, the function assign() should return the tree shown in Figure 10.4.

Make this change in assign().

10.9 The print() and read() Functions

In the newly created tree node, the mode is no longer necessarily Ops.INT, but it is the base type of the variable v. Make that change by replacing Ops.INT with

TreeOps.baseType(v.mode)

Make a similar change in the read() function.
Figure 10.3: Casting an arithmetic operation

Figure 10.4: Casting an assignment
10.10 The mod() Function

The mod operator is different from the other arithmetic operators in that it must have integer operands. The operator \% is not defined for floating-point numbers. Therefore, the action to be taken in the mod() function is to verify that both operands are integers. If either is a double, then an error message should be displayed.

10.11 double Literals

Our program will not be able to handle double literals, that is, doubles written as literal numbers such as 123.456. The reason for that is that the assembly language accepts only integer literals as operands. If we introduce double literals as operands, we would have to first store them as assembler constants in their hexadecimal form and then refer to them by their memory address in the assembly instructions. That is not too difficult since Java provides us with functions that we can use to convert a double into a String representing the double’s hexadecimal form. But, like a number of other things, it would just use up valuable time.

Therefore, in order to assign a numerical constant to a double, we will have to assign an integer literal. The cast operation will convert it to a double and then it will be stored. Thenceforth, operations on that number will be performed as doubles. For example, to create the value 0.5, we could write

```java
double x;
x = 1; // Converts 1 to 1.0 and stores it
x = x/2; // Performs floating-point division, creating 0.5
x = 1/2; // Performs integer division, creating 0
```

10.12 Testing the Compiler

Write simple test programs that include mixed-mode expressions where the left operand is int and the right operand is double, and vice versa. Also test using statements that assign a double to an int as well as an int to a double. Try reading and printing doubles. Be sure that the CAST node is created whenever necessary, but only when necessary.
10.13 Assignment

Write test programs that thoroughly test your program. Make sure that the syntax tree contains \texttt{CAST} nodes only when necessary. Then put all of your source files and the makefile in a folder named \texttt{Lab\_10}, zip it, and drop it in the dropbox.
Laboratory 11

Floating-Point Numbers and the FPU

Key Concepts

- The x87 FPU
- The FPU stack
- Floating-point arithmetic
- Converting integers and doubles

Before the Lab


Preliminaries

Copy all the files from your Lab_10 folder to a new folder named Lab_11. Copy the file PrintRead.java from the Lab_11 folder on the Compiler Design CD to your Lab_11 folder.

11.1 Introduction

We begin with the most basic operations of load and store. Then we will consider arithmetic instructions. The term load refers to moving data from somewhere else
(usually memory) into a register. The term store refers to moving data from a register to somewhere else. Therefore, to dereference a floating-point variable is to load its value into the FPU, and to assign a floating-point value to a variable is to store the value in memory. We need to consider every case in CodeGenerator.java that is affected by the existence of floating-point numbers.

11.2 The ID and NUM Cases

The code associated with an identifier node simply loads the address of the identifier and pushes it onto the stack. That has not changed.

In our compiler, numerical literals can be only integers. Therefore, a NUM node cannot hold a double and there is nothing new to do.

11.3 The DEREF Case

The DEREF case loads a floating-point value into the FPU. The address is on the stack, so we must pop the address from the runtime stack and then load the value into register st(0) of the FPU. The instructions to do this are

```
pop  %eax    # Pop address
fldl (%eax) # Load value into FPU
```

The code currently written in the DEREF case is

```
  pop  %eax    # Pop address
  push (%eax) # Push value onto stack
```

which will push the dereferenced integer onto the stack. Notice that the two blocks of code have the first instruction in common. Therefore, we should modify the DEREF case so that it looks like

```
  System.out.println(" pop  %eax    # Pop address");
  if (t.mode == Ops.INT)
    System.out.println(" push (%eax) # Push value onto stack");
  else
    System.out.println(" fldl (%eax) # Load value into FPU");
```

Modify the DEREF case in this way. Other cases will be modified in a similar way.
11.4 The ASSIGN Case

The ASSIGN case stores a floating-point number from the FPU to memory. As with DEREF, we must pop the address from the runtime stack. Then we store the value that is in st(0) of the FPU at that address. The instruction that stores is

\[ \text{fstpl} \; (%eax) \]

As with the DEREF case, we must use an if statement that distinguishes whether we are storing an integer or a floating-point value. Using the DEREF case as a model, modify the ASSIGN case so that it stores floating-point numbers.

11.5 The Arithmetic Operators

All of the cases PLUS, MINUS, TIMES, DIVIDE, and NEGATE are very simple since each is performed by a single floating-point operation. Also, in operations involving only floating-point numbers, the operands have already been placed on the FPU stack. If there are two operands, then the left operand is in st(1) and the right operand is in st(0). These are the registers that the floating-point operations are designed to act on. For example, addition is performed by the instruction

\[ \text{faddp} \]

This instruction will add st(1) and st(0), store the sum in st(1), and then pop st(0) off the FPU stack, causing st(1) to move up to st(0). The other four operations are similarly performed by the instructions

\[ \text{fsubrp} \]
\[ \text{fmulp} \]
\[ \text{fdivrp} \]
\[ \text{fchs} \]

Note the “r” in fsupbrp and fdivrp. It stands for “reverse.” Those instructions reverse the order of the operands so that they compute a - b and a / b instead of b - a and b / a.

The gnu assembler’s notation is different from many other assemblers. It adopted the AT&T notation, which is different from the Intel notation. See the web page at
http://www.delorie.com/djgpp/doc/brennan/
brennan_att_inline_djgpp.html

for some useful information on the AT&T notation.

The gnu assembler reverses the order of the operands from the usual order. For example, to add ebx to eax, one would normally write

```
add %eax,%ebx  # eax <- eax + ebx
```

That is, the first operand is normally the destination and the second operand is the source. The gnu assembler reverses this. So we write

```
add %ebx,%eax  # eax + ebx -> eax
```

Apparently for the same reason, it reverses the meanings of fsubp and fsubrp. The manual states that fsubp subtracts st(0) from st(1), storing the result in st(1). The machine code for fsubp is DEE9. It also state that fsubrp subtracts st(1) from st(0), storing the result in st(1). The machine code for fsubrp is DEE1. However, the gnu assembler assembles fsubp as DEE1 and fsubrp as DEE9, just the reverse of what the Intel manual says.

Modify the cases PLUS, MINUS, TIMES, DIVIDE, and NEGATE to perform floating-point operations.

Now it remains only to handle mixed-mode expressions.

11.6 The CAST Case

This case must convert types. If the mode is Ops.INT, then a floating-point number must be converted to an integer. If the mode is Ops.DOUBLE, then an integer must be converted to a floating-point number.

The instruction fild will load an integer from memory onto the FPU stack in st(0), storing it as a floating-point number. This is almost exactly what we want. The only problem is that it leaves the integer on the runtime stack. We should remove it. The trouble with popping it is that we have nowhere to pop it to. So, instead, we will “remove” it from the stack by adjusting the stack pointer. We will have more occasions to do this in the future, so it is good to see this technique now.

The stack grows downward, so to “pop” a value, we need to increase the stack pointer. Since an int occupies 4 bytes, we must add 4 to esp.
add $4,%esp

To convert a double to an int, the process is similar, but in reverse. The double is initially on the FPU stack. The command fistpl converts the contents of st(0) to an integer and stores it at the specified location, then it pops the value from the FPU stack. (The letter l on the end means “long.” That is the gnu way of indicating a doubleword, which in this instance is necessary. In general, the choices are b for byte, w for word, and l for long. It is probably a good idea to use l on all instructions, but I dropped it to keep things looking simple.) We would like to store the integer on the stack, so the location should be (%esp). However, since this is not a push operation, we must provide space on the stack. That is, we must subtract 4 from esp before moving the value.

Write the code for the CAST case, using the ideas discussed above.

11.7 The PRINT and READ Cases

These two cases involve function calls, which we have not discussed yet. Therefore, I have provided the updated code in the file PrintRead.java in the Lab_11 folder. Open that file and copy and paste its contents into the PRINT and READ cases.

11.8 Testing the Compiler

Be sure to test integer expressions, floating-point expressions, and all kinds of mixed-mode expressions, including mixed assignments.

11.9 Assignment

This lab will also serve as Project 5. Place all of the source files and the makefile in a folder named Lab_11, zip it, and drop it in the dropbox.
Part VI

Functions
Laboratory 12

Function Definitions and the Abstract Syntax Tree

Key Concepts

• Function definitions

• Formal arguments

• Local variables

• Return values

• Stack frames

Before the Lab


Preliminaries

Copy all of your source files from your Lab_11 folder to a new folder named Lab_12. Copy the file SemanticActionFunctions.java from the Lab_12 folder on the Compiler Design CD. It contains a number of functions and skeletons of functions that you will need to add to SemanticAction.java.
12.1 Introduction

Compiling function calls is fairly complicated. Therefore, we will break it up into three labs. The first lab will build the syntax tree for the function definitions. The next lab will build the syntax tree for the function calls. The third lab will generate the code for both the function definitions and the function calls.

The main tasks we face in handling function definitions are

- Allocating memory (the activation record) on the runtime stack for local variables.
- Returning the appropriate function value, if any.
- Popping the activation record off the runtime stack.

A function definition involves the productions

\[
\begin{align*}
\text{func} & \rightarrow \text{fbeg stmts RBRACE} \\
\text{fbeg} & \rightarrow \text{fname fargs LBRACE dcls} \\
\text{fname} & \rightarrow \text{type ID | ID} \\
\text{fargs} & \rightarrow \text{LPAREN args RPAREN | LPAREN RPAREN} \\
\text{args} & \rightarrow \text{args COMMA arg | arg} \\
\text{arg} & \rightarrow \text{type ID} \\
\text{stmt} & \rightarrow \text{RETURN expr SEMI | RETURN SEMI}
\end{align*}
\]

When you put them all together, the form of a function definition is

\[
\text{type ID}(\text{arg, ..., arg}) \\
\{}
\begin{align*}
\text{declarations} \\
\text{statements and RETURN statements}
\end{align*}
\}
\]

The syntax tree for the beginning of a function definition (\text{fbeg}) has the form shown in Figure 12.1.

The \text{ID} node contains the name of the function and other information stored in an \text{IdEntry} object. The \text{NUM} node is an integer representing the number of bytes required by the local variables.
As the formal parameters are processed, we must keep a running total of the number of bytes used. This running total is used to assign an offset from the base pointer ebp for each parameter. These will all be positive offsets as the parameters are pushed onto the stack before the function call. This will be handled by the \texttt{arg()} function.

As the local variable declarations are processed, we must keep another running total of bytes used. This running total is also used to assign an offset from the base pointer ebp for each local variable. These offsets will be negative since space for the local variables is allocated on the stack after the function call has been made. This will be handled by the \texttt{dcl()} function.

The syntax tree for a \texttt{return} statement has the form shown in Figure 12.2.

If \texttt{e.mode} is different from \texttt{retType} (the return type of the function), then the expression \texttt{e} must be cast to the type \texttt{retType}.

Recall that a \texttt{return} statement does not necessarily occur at the end of a function and that there may be more than one \texttt{return} statement in a function. Therefore, we
need a separate tree to handle the details at the physical end of the function. The syntax tree for the end of a function definition is almost trivial. See Figure 12.3.

The action to be taken is to reset the base pointer ebp and the stack pointer esp to their previous values, the values that they had before the function was called.

As an example, the trees for the function definition

```c
int sum(int a, int b)
{
    int c;
    c = a + b;
    return c;
}
```

should be output as

```plaintext
FUNC PTR|PROC|INT
    ID PTR|PROC|INT value = "sum"
    NUM INT value = 4

ASSIGN INT
    ID PTR|INT value = "c"
    PLUS INT
    DEREF INT
        ID PTR|INT value = "a"
    DEREF INT
        ID PTR|INT value = "b"
```
12.2 Miscellaneous Details

In the file SymbolTable.java, install the keyword return, if you haven’t done that already.

Change the name compiler_v2 to compiler_v3 throughout your files, as described in Lab 10.

When we store a function name in the symbol table, we must store its type as “pointer to a procedure that returns type,” where type is either int or double. We have the values Ops.PTR and Ops.INT already defined, but we must add a new symbolic constant Ops.PROC. Let its value be 1 << 3. Now we will be able to write expressions like

\[ \text{Ops.PTR | Ops.PROC | Ops.INT} \]

to represent the type of a function that returns an int.

We must also define the constants Ops.FUNC, Ops.FEND, and Ops.RET as types of tree node and add corresponding entries in the opInfo[] array in the Utility.java file.

Next to the constant Ops.GLOBAL, we should define two more constants: Ops.PARAM and Ops.LOCAL if that has not already been done.

Finally, in the file Utility.java, in the function printType(), we need to add one more case. The type may include Ops.PROC. Add a case that will print "PROC|".
12.3 The CUP file

Let us begin by adding semantic actions to the productions listed above. This is the top-down approach. We will see the function names and their parameters and we will describe briefly what each function will do. In the next section we will add the details to the functions.

For the production

\[
\text{fname ::= type:t ID:f}
\]

make a call to \text{SemanticAction.fname(f, t)} and for the production

\[
\text{fname ::= ID:f}
\]

make a call to

\[
\text{SemanticAction.fname(f, new Integer(Ops.INT))}
\]

Be sure to pass the function value on to the nonterminal by using \text{RESULT}. The \text{fname()} function registers the function name in the symbol table and returns an \text{IdEntry} for the function. Therefore, the nonterminal \text{fname} must be declared to be of \text{IdEntry} type. The second of those two productions indicates that if the return type of a function is not declared, then it will be assumed to be \text{int}.

Next, consider the production

\[
\text{arg ::= type:t ID:i}
\]

When this is matched, we should make a call to \text{SemanticAction.arg(i, t)}. The \text{arg()} function installs a function parameter in the symbol table, but there is no need to return an \text{IdEntry}. Therefore, this is a \text{void} function, so the \text{arg} nonterminal is not given a type.

The production

\[
\text{fbeg ::= fname:f fargs LBRACE dcls}
\]

calls the function \text{SemanticAction.fbeg(f)}. This function returns the same \text{IdEntry} object that is passed to it, so the type of the nonterminal \text{fbeg} must be \text{IdEntry}.

The production

\[
\text{func ::= fbeg:f stmts RBRACE}
\]
calls on `SemanticAction.func(f)`, which returns `void`.

Finally, there are the productions

\[
\text{stmt ::= RETURN SEMI} \\
\text{stmt ::= RETURN expr:e SEMI}
\]

Both of these call the function `SemanticAction.ret()`, but the first one passes a `NUM` tree containing the integer 0 while the second one passes the parameter `e`.

Let’s consider the semantic actions in the order in which they will be applied by the compiler. Open the file `SemanticActionFunctions.java`. It contains a new `dcl()` function and several skeletons functions. As each function is discussed in the next section, copy and paste the function skeleton from this file to the file `SemanticAction.java`, maintaining the functions in alphabetical order.

### 12.4 The `fname()` Function

The `fname()` function is called in response to the productions

\[
\text{fname} \rightarrow \text{type ID | ID}
\]

The second form is matched if the function is declared without specifying a return type. The old C rule was that the default return type is `int`. In the second case, the parameter

\[
\text{new Integer(Ops.INT)}
\]

is passed in place of the parameter `t`.

The purpose of the `fname()` function is to install the function name in the symbol table and store all relevant information about the function.

Initially, `fname()` should verify that the current level is global. If it is not global, then display an appropriate error message, including the function name. Then it should look up the function name in the symbol table to see if it has already been installed. If it has been, then we should avoid installing it again. Next, (if the name is not already in the symbol table) we install the function name in the symbol table at the global level, set its scope data member to `Ops.GLOBAL`, and set its type equal to

\[
\text{Ops.PTR | Ops.PROC | type.intValue()}
\]
Then we initialize some symbol table variables concerning the argument list and the local variables. Add the statements

```java
SymbolTable.fArgSize = 8;
SymbolTable.dclSize = 0;
SymbolTable.retType = type.intValue();
SymbolTable.enterBlock();
```

The integer `SymbolTable.fArgSize` is initialized to 8, representing the space used by the instruction pointer `eip` and the base pointer `ebp`. As arguments are encountered, we will increase `fArgSize`. The integer `SymbolTable.dclSize` is initialized to 0. This will be increased as local variable declarations are encountered. The integer `SymbolTable.retType` is set to the return type, as defined by the parameter `type`. Finally, since we are entering a block, we should call the `enterBlock()` function of the `SymbolTable` class to increase the block level and create a new hash table.

Write the function `fname()`.

### 12.5 The arg() Function

The function `arg()` is invoked by matching the production

\[
\text{arg} \rightarrow \text{type} \ \text{ID}
\]

which is used in the larger productions

\[
\text{args} \rightarrow \text{args} \ \text{COMMA} \ \text{arg} \mid \text{arg}
\]

Before installing the argument in the symbol table, we should look up the name to see if it is already in the table at the local level. If it is, then we should print an error message and not install it again. If it isn’t there, then we should go ahead and install it.

Each argument must be installed in the symbol table at the local level along with its data type, scope, and offset. The data type is given by the parameter `type`, the scope is `Ops.PARAM`, and the offset is the current value of `fArgSize`.

After assigning `fArgSize` to `id.offset`, we should increment `fArgSize` by the base size of this parameter in preparation for the next parameter.

Write the function `farg()`.
12.6 The dcl() Function

Up to this point, the dcl() function simply builds an allocation tree for a global variable. Recall that global variables are stored in a part of memory designated for globals, while local variables are stored on the runtime stack. Thus, dcl() must divide into two cases now.

Look at the dcl() function in SemanticActionFunctions.java and see that the division is based on the current level. (Therefore, it is important that we increased the block level in the fname() function.) The first part of the if-else block is the same as what was there before. The second part

```java
id.scope = Ops.LOCAL;
SymbolTable.dclSize += TreeOps.baseSize(id.type);
id.offset = -SymbolTable.dclSize;
```

sets the scope to local, increments the size of the local variable block, and assigns the negative of the current size as the offset for this variable. The reason we assign the offset after incrementing the size rather than before, as we did in arg(), is because we are building down from the base pointer now, whereas we were building up before.

Copy dcl() and paste it in SemanticAction.java in place of the old dcl() function.

12.7 The fbeg() Function

This function is invoked by the production

```
fbeg → fname fargs LBRACE dcls
```

which means that the parameters (fargs) and the local variables (dcls) have been processed. Thus, we are ready to print the FUNC syntax tree shown in Figure 12.2. This is a very straightforward exercise using TreeNode constructors. The tree should be constructed and printed and the parameter id should be returned.

Write the fbeg() function.

12.8 The ret() Function

When the RET tree is built, we must be sure to cast the return value to the correct type, if necessary. The variable SymbolTable.retType holds the return type. The
parameter e is a reference to the returned object. Call on the cast() function, which will create a CAST node, if necessary.

Now build the RET tree, as shown in the diagram earlier in the lab, and print it.

12.9 The func() Function

This function is called when the function definition is complete. That is indicated by the right brace } at the end. When the right brace is received, that completes the pattern in the production

\[ func \rightarrow fbeg stmts RBRACE \]

Since the function fbeg() returned the IdEntry for the function name, we have that parameter available now.

This function should build the FEND tree, as in Figure 12.3, display the tree, then set the return type retType to 0 and call the leaveBlock() function.

Write the func() function.

12.10 Debugging and Testing the Compiler

Debugging the compiler may be more of a challenge than before. I suggest that you uncomment all of the debugging print statements in the relevant productions in the CUP file and in the relevant semantic action functions. This way you will be able to trace the compiler's progress up to the point where an error occurred. This technique should be standard practice for all of your debugging work from here on. Once your compiler is debugged, you may comment out those print statements again.

Use a test program that contains function definitions, but no function calls, since we have not implemented function calls yet. Be sure to test all possibilities concerning data types. For example, test both int and double arguments, local variables, and return types.

12.11 Assignment

Put the source files and the makefile in a folder named Lab_12, zip it, and drop it in the dropbox.
Laboratory 13

Function Calls and the Abstract Syntax Tree

Key Concepts

- Function calls
- Actual parameters
- Strings

Before the Lab

The reading assignment is the same as in Lab 12.

Preliminaries

Copy all of your source files from your Lab_12 folder to a new folder named Lab_13.

13.1 Introduction

In this lab, we will continue to build the syntax tree for functions, but this time we will build the tree for the function call. Together with Lab 12, this will complete the tree-building part. In the next lab, we will do the code generation for function calls, which will result in Version 3 of our compiler.
13.2 Function Calls

When a function is called, we pass a list of expressions as the actual parameters. Often each expression is just a variable, but they may be any legal expression of the appropriate type.

In this lab we will create abstract syntax trees for function calls. The root node will be the `CALL` operation. Its left subtree will be the name of the function. Its right subtree will be an expression or a list of expressions or null if there is no parameter.

A list of expressions is a tree whose root node is a `LIST` operation. Its left subtree is an expression or a list of expressions, recursively defined, and its right subtree is a single expression. The final `LIST` node will contain expressions on both its left and right subtrees.

Consider first a function with only one parameter.

```c
int copy(int a) {
    return a;
}
```

The tree for the function call `copy(b)` would be as shown in Figure 13.1 and the compiler would display it as

```
CALL INT
```
On the other hand, if the function has three parameters, such as

```java
int sum(int x, int y, int z)
{
    return x + y + z;
}
```

then the tree for the function call `sum(10, 20, 30)` would be the tree in Figure 13.2.

and the compiler would print

```
CALL INT
ID PTR|PROC|INT value = "sum"
LIST
    LIST
        NUM INT value = 10
        NUM INT value = 20
        NUM INT value = 30
```

Note that the arguments appear in order from bottom to top. That will be important when the code generator pushes them onto the runtime stack, since they must be pushed in order from right to left.

### 13.3 The Argument List

Let us build the subtree of arguments first. Then we will build the CALL tree.

We must add `Ops.CALL` and `Ops.LIST` to the files `Ops.java` and `Utility.java` in the `opInfo[]` array. Do that now, following the pattern that was set with other tree operators.

The argument list is a sequence of expressions, separated by commas. The productions matched are

\[
\text{exprs} \rightarrow \text{exprs} \ COMMA \ \text{expr} \ | \ \text{expr}
\]

In our compiler we will pass arguments by value only. Thus, each expression in the actual argument list must be evaluated before its value can be passed. Also,
our compiler will not check that the type of the argument matches the type of the parameter. If they do not match, then the compiled program will not run correctly and that will not be the fault of the compiler.

The function \texttt{exprs()} receives two parameters \texttt{e1} and \texttt{e2}. The parameter \texttt{e1} represents the list of expressions so far and \texttt{e2} represents the latest expression to be added to the list. A new \texttt{LIST} tree is to be created with the form shown in Figure 13.3.

Note that \texttt{e1} is on the left. Thus, the rightmost argument \texttt{e2} will appear highest in the tree.

Write the code for the \texttt{exprs()} function.

The action for the production

\[
\texttt{exprs} \rightarrow \texttt{expr}
\]

is simply to return the expression tree itself.
13.4 The call() Function

The call() function will be invoked by the productions

\[ expr \rightarrow \text{ID LPAREN} \text{exprs} \text{RPAREN} \]
\[ | \text{ID LPAREN} \text{RPAREN} \]

The call() function receives two parameters. The first parameter s is the name of the function. The second parameter e is a LIST TreeNode at the root of a tree of expressions or it is a single expression (when there is only one parameter), or it is null (when there is no parameter).

The call() function should first look up the name of the function in the symbol table at the global level. If the returned IdEntry id is null, then call() should create an entry with type

\[ \text{Ops.PTR} \mid \text{Ops.PROC} \mid \text{Ops.INT} \]

and scope \text{Ops.GLOBAL}. That is, the default return type is int.

Then it should create and return a tree of the form in Figure 13.4.

Now that we can call functions, we can call functions in the C library, such as sqrt(), cos(), and printf(). This creates a new need: the need for strings, since the first parameter of the printf() function is a format string. We have postponed strings for as long as possible. Now we will deal with them.

13.5 The String Type

Our lexer already returns string tokens as a Symbol object containing the symbolic integer sym.STR and the value of the string. (Check the lexer to see that when it
returns a string, the quotations marks are part of the string. We should keep that in mind.) Thus, we must instruct the parser to take the appropriate action in the production

\[ \text{expr} \rightarrow \text{STR} \]

It will call on the \text{str()} function in the \text{SemanticAction} class. The \text{str()} function should create and return a string \text{TreeNode}. To keep our compiler well organized, we should create a \text{TreeNode} constructor for strings that is analogous to the \text{TreeNode} constructors for identifiers and numbers that we created in earlier labs.

The body of this \text{TreeNode} constructor should be

\begin{verbatim}
  oper = Ops.STR;
  mode = Ops.PTR | Ops.CHAR;
  str = s;
\end{verbatim}

Note the appearance of two new constants, \text{Ops.CHAR} and \text{Ops.STR}. Add \text{Ops.CHAR} to the \text{Ops} class as a data type, in the group with \text{Ops.INT}. You might give it the value \(1 << 2\). Add \text{Ops.STR} as a new type of tree node, in the group with \text{Ops.CALL}. Also, add a corresponding line in the \text{opInfo[]} array in \text{Utility.java}. We will not actually have \text{char} objects in our programs, but the string type in C is officially a pointer to \text{char}, so we need them for that purpose.

While you are in \text{Utility.java}, add a case to the function \text{printType()} that handles \text{Ops.CHAR}. Note also that \text{Ops.STR} is a type of tree node, while the data type is a pointer to a character. We must be careful to distinguish between the data type and the tree type. Also, in the function \text{printNode()}, we have cases that print identifier and number leaf nodes. Add a case that will print a string leaf node.
Create the TreeNode constructor and then have str() call on it, passing it the string.

13.6 Testing the Compiler

At the present time, we are only building the syntax tree. Therefore, be sure to use the -t option and not the -c option.

To test our compiler, we should now use test programs that contain function calls. Try all kinds of calls. Test arguments that are ints, doubles, numbers, and expressions. Test functions with no parameter, one parameter, and several parameters. Also use arguments that are themselves function calls. For example, if you have a sqr() function that squares a number, you might try statements like

\[ y = \text{sqr}(\text{sqr}(x)); \]

that will compute the fourth power of x.

13.7 Assignment

Put the source files and the makefile in a folder named Lab_13, zip it, and drop it in the dropbox.
LABORATORY 13. FUNCTION CALLS AND THE AST
Laboratory 14

Functions and Code Generation

Key Concepts

• Pushing parameters
• Calling functions
• Clearing parameters
• String objects

Before the Lab

The reading assignment is the same as in Lab 12.

Preliminaries

Copy all of your source files from your Lab_13 folder to a new folder named Lab_14.

14.1 Introduction

To implement function calls, we will need to create six new cases in CodeGenerator.java to deal with new types of nodes in the syntax tree. The new cases are

  Ops.CALL
  Ops.FEND
  Ops.FUNC
  Ops.LIST
We will start with the FUNC case, which is also the simplest case.

### 14.2 The FUNC Case

The task in this case is to write the assembly code that is needed to start a function definition. This code follows a standard pattern:

```assembly
# Code for FUNC
.text
.globl _fname
.def _fname; .scl 2; .type 32; .endef
_fname:
push %ebp  # Save base ptr
mov %esp,%ebp  # Make stack ptr new base ptr
sub $num,%esp  # Adjust stack ptr for local block
```

where `fname` is the function name and `num` is the number of bytes required by the local variables (not the parameters). The gnu assembler expects function names to begin with an underscore. For example, if we named a function `copy`, then gnu expects `name` in the assembly code to be `_copy`.

By the time execution reaches this point, the call statement has already been executed. Therefore, the parameters and the instruction pointer are already on the stack. Notice that the base pointer is pushed onto the runtime stack next. Then the current value of the stack pointer becomes the new base pointer. Finally, by subtracting `num` from the stack pointer, we provide stack space for the local variables. Notice also that each parameter will have a *positive* offset from the (new) base pointer and each local variable will have a *negative* offset from the base pointer.

The FUNC tree has the logical structure shown in Figure 14.1. The name of the function can be obtained from the left subtree, which should be an identifier node. The number of bytes for local variables can be obtained from the right subtree.

If you look in the `CodeGenerator` member function `init()`, you will see that this code was generated for the `main()` function. We should now remove this function and the call to it in `compiler.v3.java` since our compiler will now generate this code when it sees the main function.
Figure 14.1: The logical structure of a function tree

Write the Java code that will output the assembly code for a FUNC tree.

14.3 The FEND Case

At the end of a function, if there is no return statement, then we must execute a return with the default return value 0. Before making the jump back to the calling function, we must restore the stack pointer and the base pointer. Essentially, this undoes what was done above when the function was called.

Write statements in the FEND case that will output assembly code that will

- Move the base pointer to the stack pointer, thereby restoring the old stack pointer. (Recall that we saved the old stack pointer as the new base pointer.)

- Pop the value that is on the stack to the base pointer, thereby restoring the old base pointer. (Recall that we pushed the old base pointer onto the stack.)

- Return to the calling function.

In that last step, we really should return 0 since the function has a return type and is expecting a value. However, for simplicity we will not do that (unless you want to). That means that it is the programmer’s responsibility to return a value whenever the program expects a value, or else the program may crash. A value will not be returned automatically.

That is all there is in the FEND case. This code appeared in the CodeGenerator function finish(), which you should now remove. It was there to finish the main() function. Also, remove the call to finish() found in compiler_v3.java.
14.4 The RET Case

This case is very similar to the FEND case, except that we have to return the value specified in the left subtree.

Our compiler does not recognize the void function type, so there will be a return value. Recall that the semantic action for

\[ expr \rightarrow \text{RETURN SEMI} \]

fills in the value 0. We must cast the return value to the return type. Then we must check whether it is an int value or a double value to see how to return it. If it represents a double value, then there is nothing to do. The double value is already in register st(0) of the FPU, which is where it is expected to be upon return from the function. On the other hand, if it represents an int value, then we need to pop that value from the stack and place it in register eax, which is where it is expected to be.

Write the code that will do this.

Why not return an integer value by pushing it onto the stack and letting the calling function pop it off the stack? There is a good reason why we should not do that. What is it?

14.5 The CALL Case

The CALL case is a bit more complicated since it must first push the parameters onto the stack. Furthermore, the parameters may themselves be expressions that must first be evaluated. As our assembly-language program processes each parameter, it will create code to evaluate it and push it onto the stack. Since that is what happens anyway when an expression is evaluated, it ought not be a complication.

One issue is that floating-point expressions ordinarily leave their values on the floating-point stack, in st(0). Thus, we will need to move them over to the runtime stack. (All parameters of a function call, both integer and floating-point, must appear on the runtime stack when the function is called.)

Therefore, our procedure for processing parameters will traverse the tree, evaluating expressions in the usual way, except that if the expression is floating-point, then we move its final value to the runtime stack.
As we process each parameter, we need to keep track of the size of the parameter block. That is so that we can generate the instruction that will clear the parameter block upon returning from the called function. The complication is that the parameter list may itself contain calls to other functions (which may contain calls to functions, etc.). Thus, we may have to interrupt our count of the parameter block size for this function while we initiate a similar count for another function.

This calls for a stack. As a static object in the CodeGenerator class, create a Stack named paramStack. Use the Java Stack class. You may look up the details at

http://java.sun.com/j2se/1.4.2/docs/api/

The Stack class is in the package java.util. Therefore, you must include the statement

```java
import java.util.*;
```

in CodeGenerator.java. Also, create an object paramBlockSize that is initialized to 0.

In the CALL case, first push the current value of paramBlockSize onto the stack and initialize paramBlockSize to 0. As parameters are encountered, we will increase paramBlockSize by the size of those parameters.

Now let us begin processing parameters. First, if the right subtree is null, then there is no parameter and there is nothing to do.

If it is not null, but is not a LIST node, then it represents a single parameter. Its tree should be built by calling traverseTree() and, if its value is floating-point, the value should be moved from st(0) to the stack. Note: you cannot just push it. You must make room on the stack and then use fstpl to move it. After processing it, add its size to paramBlockSize. You can use the baseSize() function for this, but you must modify it to allow for strings, i.e., pointers to chars.

Now, if the right subtree is a LIST node, then it should be processed recursively by calling generateNodeCode(), which will handle it under the LIST case.

### 14.6 The LIST Case

To process the LIST tree, begin at the root node and process the right subtree, which is an expression tree, by calling on traverseTree(). After that is done, check to
see if the result is floating-point. If it is, then move the value from \texttt{st(0)} onto the runtime stack. Then add the parameter size to \texttt{paramBlockSize}.

After doing the right subtree, consider the left subtree. If it is itself a \texttt{LIST} tree, then it can be handled recursively; just call on \texttt{generateNodeCode()} and consider it done. On the other hand, if you are at the bottom of the tree, then the left subtree is also an expression tree, so it must be evaluated and possibly moved from \texttt{st(0)} to the runtime stack. The pattern here should be exactly the same as the pattern for the right subtree. Be sure to add the parameter size to \texttt{paramBlockSize}.

14.7 The CALL Case, Continued

That takes care of the parameters. Now back to the \texttt{CALL} case. We are ready to make the call itself. The form of the call statement is

\begin{verbatim}
call _fname
\end{verbatim}

where \texttt{fname} is the name of the function, as stored in the symbol table. It can be retrieved from the left subtree, which consists of a single identifier node.

Once the call is made and execution returns, there are two more things to be done. We must clear the parameters off the stack and then, if the return value is an \texttt{int}, we must push it onto the stack, where the next instruction expects to find it.

To clear the parameters, we need a statement of the form

\begin{verbatim}
add %n,%esp
\end{verbatim}

where \texttt{n} is the size of the parameter block. Output this statement, using the value of \texttt{paramBlockSize}. Then you must restore \texttt{paramBlockSize} by popping the previous value off \texttt{paramStack} and assigning it to \texttt{paramBlockSize}.

Finally, check the type of the returned value. If it is \texttt{int}, then the value is currently in \texttt{eax}. If so, then we need to push it onto the runtime stack. On the other hand, if it is \texttt{double}, then it is already in \texttt{st(0)}, which is where it should be.

Once you write all of that, the \texttt{CALL} case should be done.

There are two more details to deal with concerning the \texttt{FUNC}, \texttt{FEND}, \texttt{LIST}, and \texttt{CALL} cases. The \texttt{FUNC}, \texttt{FEND}, and \texttt{CALL} trees all contain \texttt{ID} nodes on the left side. However, in none of these cases should the \texttt{ID} tree be processed in the usual way. Since the syntax tree is normally traversed post-order, the \texttt{ID} node would be processed before we knew that it was part of the \texttt{FUNC}, \texttt{FEND}, or \texttt{CALL} tree. We faced a similar problem
with `ALLOC` trees earlier. We will solve this problem in a similar way. At the beginning of the `generateCode()` function, make `FUNC` and `FEND` special cases along with `ALLOC`. The action should be the same. In the `traverseTree()` function, make `CALL` a special case. If the tree is a `CALL` tree, then we should skip processing the left and right subtrees; they will be handled correctly in the `CALL` case of `generateNodeCode()`. The reason that we handle `CALL` differently from `FUNC` and `FEND` is that a call statement with its attendant parameter list may appear within a statement, while `FUNC` and `FEND` trees cannot occur within statements.

The `LIST` case must be handled differently because we traverse the `LIST` subtrees from right to left. (Otherwise, the parameters will be pushed in the reverse order.) The `traverseTree()` function processes them from left to right. Thus, the `LIST` case must be handled similarly to the `CALL` case in `traverseTree()`. That is, skip over the recursive calls. In the `LIST` case in the `generateNodeCode()`, we have it traverse the right subtree first, then the left subtree.

14.8 The ID Case

There remains the problem of accessing parameters and local variables of a function when they are used in expressions within the function. Each is accessed by applying an offset to the address in the `ebp` register. That offset is stored in the offset data member of the `IdEntry` object for the identifier stored in the symbol table.

When we process an identifier, we must test to see if it is a global or a parameter or local variable. If it is global, then we perform the action that is already in the ID case. If it is not global, then it must be a local or a parameter. In either case, we follow the same general pattern of the global case, but replace the identifier name with

\[ n(%ebp) \]

where \( n \) is the offset for that identifier. Write the code for the ID case.

14.9 The STR Case

All we need to do in the case of a string is to create the string in memory and push its address onto the runtime stack. The assembly code to do this is
.data
L0n: .asciz "string"
.text
    lea L0n,%eax       # Load addr of string
    push %eax         # Push addr of string

where \( n \) is the current value of jmpLabel and "string" is the string. The variable jmpLabel is a static member of the CodeGenerator class that is used to create unique labels. Whenever it is used, it should be incremented before use to give the next label number. The leading 0 is used to distinguish it from other labels that we will use later. If you look at the code in the READ and PRINT cases, you will see how jmpLabel was used there. The assembler treats L0n as a symbolic name whose value is the address at which it occurs in the program. The string "string" is stored by the assembler in the “data” area of the assembled program. That is what the directive .data does. The .text directive sends us back to the code area.

Write the code for the STR case.

14.10 Testing the Compiler

Now you are ready to test your compiler. If you wrote good test programs in Lab 13, then you can use them here. In any case, you want to be sure to try a variety of return types; a variety of parameter lists, including zero, one, or more than one parameter; parameters that are themselves function calls; and a variety of return statements, including return statements that fail to specify a value.

Since we do not have any means of making decisions, you should not try recursive calls since there would be no way to end the recursion. Once we implement if statements, we will be able to handle recursive calls.

14.11 Assignment

This lab will serve as Project 6. Put all of your source files and the makefile in a folder named Project_6, zip it, and drop it in the dropbox.
Part VII

Control Flow Structures
Laboratory 15

Control Flow Structures and the Abstract Syntax Tree

Key Concepts

- Labels and jumps
- if statements
- Backpatching
- Conditional expressions
- Linked lists

Before the Lab

Read Sections 8.4 and 8.6 in Compilers: Principles, Techniques, and Tools.

Preliminaries

Copy all the source files from your Lab_14 folder to a new folder named Lab_15.

15.1 Introduction

At long last, we will incorporate decision structures into our compiler. The two basic decision structures that we will implement are one-way if statements
if (condition)
    stmt

and two-way if statements

    if (condition)
        stmt1
    else
        stmt2

It turns out that the technique, called backpatching, that we use to do this will also allow us to implement while loops and for loops very easily.

15.2 Version 4

This will be version 4 of our compiler, so change the name to compiler.v4 throughout your files. As before, use grep to find out where the name compiler.v3 occurs and then change it to compiler.v4. Also, be sure that the keywords if and else have been installed in the symbol table.

15.3 Labels and Jumps

In order to implement if statements, we must be able to jump over a block of code. This will be a forward jump. (To implement loops, we create a backward jump.) There are two kinds of jump statement: conditional and unconditional. We will use both kinds.

All jump statements must specify a destination. In machine code, this is either an absolute address or an offset from the current instruction pointer. In assembly language, it can be a label. A label is an assembly-code identifier that is written beginning in the leftmost column and followed by a colon. A jump statement will name a label as its destination. For example, we might write

    LoopBegin:
    :
        jmp LoopBegin
    LoopEnd:
to create a loop.

With most statements, there is a single destination, which we will call the “next” destination, to which execution goes once that statement has been executed. However, in the case of conditional expressions, there are two destinations: a “true” destination and a “false” destination.

15.4 Markers in the Grammar

We will now introduce two markers into the grammar as nonterminals. The nonterminal \( m \) generates a label. The nonterminal \( n \) generates an unconditional jump. They will be used in the following productions:

\[
\begin{align*}
stmts & \rightarrow \ stmts_1 \ m \ stmt \\
stmt & \rightarrow \ \text{IF LPAREN } cexpr \ \text{RPAREN } m \ stmt \\
stmt & \rightarrow \ \text{IF LPAREN } cexpr \ \text{RPAREN } m_1 \ stmt_1 \ n \ ELSE \ m_2 \ stmt_2 \\
func & \rightarrow \ fbeg \ stmts \ m \ RBRACE
\end{align*}
\]

The nonterminal \( cexpr \) is a conditional expression. For the time being, it will be a numerical expression with the rule that zero is interpreted as false and nonzero is interpreted as true. That is the standard rule in C.

In the first production,

\[
stmts \rightarrow stmts_1 \ m \ stmt,
\]

the label produced by \( m \) serves as a destination for the preceding statement. In the second production,

\[
stmt \rightarrow \ \text{IF LPAREN } cexpr \ \text{RPAREN } m \ stmt,
\]

the label produced by \( m \) serves as the “true” destination for \( cexpr \), i.e., the destination when \( cexpr \) evaluates to true. In the third production,

\[
stmt \rightarrow \ \text{IF LPAREN } cexpr \ \text{RPAREN } m_1 \ stmt_1 \ n \ ELSE \ m_2 \ stmt_2,
\]

the label produced by \( m_1 \) serves as the “true” destination for \( cexpr \) and the label produced by \( m_2 \) serves as the “false” destination for \( cexpr \). The jump produced by \( n \) will jump over \( stmt_2 \).
In the fourth production,

\[ \text{func} \rightarrow \text{fbeg stmts m RBRACE}, \]

the label generated by \( m \) serves as the "next" destination of \( \text{stmts} \), immediately before the \text{return} statement that is automatically generated at the physical end of the function.

The most obvious problem here is in the second production. Where is the "false" destination of \( \text{cexpr} \)? Almost as obvious is the question, exactly where should \( n \) jump to in the third production? That is where backpatching comes in.

### 15.5 Backpatching

Backpatching is a technique of creating a temporary label (a backpatch label) as the label of a destination that has yet to be determined. Once the destination is determined, the backpatch label is resolved with an actual label. (Backpatch labels are not used as actual labels.) Backpatch labels are named \( B_1, B_2, B_3, \ldots \) and actual labels are named \( L_1, L_2, L_3, \ldots \), except that, to avoid confusion, we will not use the same number for both a backpatch label and an actual label.

In the case of statements matching the production

\[ \text{stmts} \rightarrow \text{stmts} _1 m \text{stmt} \]

the "next" destination of \( \text{stmts}_1 \) will be the label produced by \( m \). But in the case of \text{if} statements, we will need to use a backpatch node to store a pair of destinations for the conditional expression.

### 15.6 Backpatch Nodes

Create a file named \text{BackpatchNode.java}. An object of this class has two data members: \text{trueList} and \text{falseList}. Each is a \text{LinkedList} object. Each linked list is a list of \text{Integers} representing backpatch labels.

Provide two constructors: the default constructor and a constructor that has two linked lists as its parameters, to be assigned to \text{trueList} and \text{falseList}.

Why store a list of labels instead of a single label? That is because often there are several jumps that must all be resolved to the same destination. For example,
consider again the production

\[ stmt \rightarrow IF \ LPAREN \ cexpr \ RPAREN \ m_1 \ stmt_1 \ n \ ELSE \ m_2 \ stmt_2 \]

The destination of the jump statement produced by \( n \) must be the same as the “next” destination of \( stmt_1 \), which is also the “next” destination of \( stmt_2 \). Thus, all three backpatch labels will be collected into a list and later resolved to the same destination.

Create the two `BackpatchNode` constructors now. The first should be the default constructor. It should set `trueList` and `falseList` to null. The second should have two parameters, `tList` and `fList`, that are `LinkedList`. The parameter `tList` should be assigned to `trueList` and the parameter `fList` should be assigned to `falseList`. We will use these constructors shortly.

### 15.7 The `printNode()` Function

In the file `Utility.java`, the `printNode()` function prints leaf nodes in a special way. Now that we have `LABEL` and `BLABEL` nodes, we have two new types of leaf node. In the function `printNode()`, add two new cases, along with the existing cases of `Ops.ID`, `Ops.NUM`, and `Ops.STR`, that will print `LABEL` and `BLABEL` nodes.

The output of these nodes should be of the form

\[ \text{LABEL label=n} \]

and

\[ \text{BLABEL blabel=n} \]

where \( n \) is the numerical value of the label.

### 15.8 Two Label Functions

Two convenient functions will be

- `TreeNode(int op, int labl)`
- `int newLabel()`

We should write these two first since other functions will use them. In the `TreeNode` constructor, the parameter `op` is either `Ops.LABEL` or `Ops.BLABEL`. Add the constants
Ops.LABEL and Ops.BLABEL to the files Ops.java and Utility.java in the usual way. The sole purpose of this constructor is to create and return a LABEL or BLABEL node. The first parameter should be assigned to oper and the second should be assigned to num.

The function newLabel() returns a new integer to be used for the next label. It should increment a SemanticAction class variable labelNum and return its new value. Write these two functions.

15.9 The m() and n() Functions

The function m() needs to create and print a LABEL tree and return the Integer used in the label. First, get an integer representing a new label number. Then you can use a TreeNode constructor to create the LABEL tree with that number for the label. Then print the LABEL tree. Finally, have m() return the label as an Integer. (It should be an Integer because it will be put into a LinkedList of Integers. Java will not allow us to put ints into a LinkedList.) Therefore, the prototype of m() is

\[
\text{public static Integer m()}\
\]

The function n() must first create a BLABEL tree. That is done in the same way as you created a LABEL tree. Then create a JUMP tree, attach the BLABEL tree as its left subtree, and print the tree. Finally, create a LinkedList containing the integer that was used in the backpatch label and return that LinkedList. Therefore, the prototype of n() is

\[
\text{public static LinkedList n()}\
\]

15.10 Backpatch Functions

Three functions will allow us easily to manage the backpatch labels.

\[
\text{LinkedList makeList(int labl)}\
\text{LinkedList merge(LinkedList b1, LinkedList b2)}\
\text{void backpatch(LinkedList b, Integer labl)}\
\]

See Section 8.6 of Compilers for an excellent discussion of these functions.

The function makeList() will create a LinkedList object with a single Integer in it, an integer with the value labl.
The function \texttt{merge()} will take two linked lists \texttt{b1} and \texttt{b2} and merge them into one list. The merged list will replace the old \texttt{b1} and it will be returned.

The function \texttt{backpatch()} will resolve all the backpatch labels in the list \texttt{b} with the destination label \texttt{labl}. It will construct and print an \texttt{EQU} tree (equate tree) for each backpatch label in the list. (An \texttt{EQU} tree equates a backpatch label with an actual label.)

Now let us write each of the three backpatching functions, beginning with \texttt{makeList()}. You should go to the Java web site for the \texttt{LinkedList} class to see what member functions are available.

The function \texttt{makeList()} should begin by creating a new \texttt{LinkedList} object. Then it should add the label \texttt{labl} to the list and return the list. See the Java \texttt{LinkedList} web page for details on the member functions.

The function \texttt{merge()} should merge the lists \texttt{b1} and \texttt{b2} and return the merged list. Look at the \texttt{LinkedList} web page and figure out which \texttt{LinkedList} function(s) will do this.

The function \texttt{backpatch()} is more substantial than the others, but still pretty simple, thanks to the \texttt{LinkedList} class. First, it must create a \texttt{LABEL TreeNode} for the label \texttt{labl}. Call it \texttt{labTree}. Then, for each \texttt{Integer} stored in the list \texttt{b}, it must create a \texttt{BLABEL TreeNode} and then attach it and the \texttt{LABEL TreeNode} already created as the left and right subtrees of a new \texttt{EQU} tree. (You will need to define the symbol \texttt{Ops.EQU} in \texttt{Ops.java} and update the \texttt{opInfo[]} array in \texttt{Utility.java}.) For example, if \texttt{b} is \{3, 4, 6\} and \texttt{labl} is 8, then the \texttt{EQU} trees in Figure 15.1 will be created.

To create each \texttt{BLABEL TreeNode}, you will have to get the next backpatch label out of the linked list and use \texttt{intValue()} to get its \texttt{int} value. Call it \texttt{blabl}. Then the statement

\[
\text{TreeNode blabTree = new TreeNode(Ops.BLABEL, blabl);}
\]

will construct the \texttt{BLABEL TreeNode}. The statement

\[
\text{TreeNode equTree = new TreeNode(Ops.EQU, 0, blabTree, labTree);}
\]

will join them together in an \texttt{EQU} tree. The \texttt{backpatch()} function should print each \texttt{EQU} tree as it is produced. Then it is finished.
The type of the nonterminals `stmt`, `stmts`, and `n` is now `LinkedList`. Therefore, in the CUP file, we need to declare them to be `LinkedLists`. In order for the `LinkedList` class to be recognized in `parser.java`, be sure to include the statement

```java
import java.util.*;
```

at the beginning of the CUP file. This statement must also be included in the `SemanticAction.java` file.

Similarly, the nonterminal `m` will be an `Integer` and the nonterminal `cexpr` will be a `BackpatchNode`.

First, add semantic actions to the productions

\[
\begin{align*}
m & \rightarrow \varepsilon \\
n & \rightarrow \varepsilon
\end{align*}
\]

The actions are simply calls to the `SemanticAction` functions `m()` and `n()`.

Also, in several productions we need to add variable names to some of the symbols. These productions should now be
func ::= fbeg:f stmts:s m:m1 RBRACE
stmts ::= stmts:s1 m:m1 stmt:s2
stmt ::= LBRACE stmts:s RBRACE
  | IF LPAREN cexpr:c RPAREN m:m1 stmt:s
  | IF LPAREN cexpr:c RPAREN m:m1 stmt:s1 n:n1 ELSE m:m2 stmt:s2
cexpr ::= expr:e

The variables s, s1, s2, and n1 will be LinkedLists and the variables m1 and m2 will be Integers.
Some of these productions do not currently have semantic actions associated with them. The details of these actions will be relegated to functions in the SemanticAction class, so right now all we need to do is to add the semantic action function calls.
In the production
func ::= fbeg:f stmts:s m:m1 RBRACE
we have already been using the action func(f), but now we must add two more parameters. The new action is
SemanticAction.func(f, s, m1);
The production
stmts ::= stmts:s1 m:m1 stmt:s2
requires the action
RESULT = SemanticAction.stmts(s1, m1, s2);
The production
stmts ::= 
requires the action
RESULT = new LinkedList();
That is because an empty statement should have an empty list of backpatch labels. While we are on that subject of empty linked lists, certain other stmt productions that previously returned null now must return an empty LinkedList, as in the above
example. Make that change *wherever necessary*. In some cases that change will show up in the `SemanticAction` function rather than in the CUP file. In general it will show up at any point where we previously returned null, or nothing, for one of the nonterminals `stmts`, `stmt`, or `n`.

For the productions

```
stmt ::= IF LPAREN cexpr:c RPAREN m:m1 stmt:s
```

and

```
stmt ::= LPAREN cexpr:c RPAREN m:m1 stmt:s1 n:n1 ELSE m:m2 stmt:s2
```

the actions are

```
RESULT = SemanticAction.ifStmt(c, m1, s);
```

and

```
RESULT = SemanticAction.ifElseStmt(c, m1, s1, n1, m2, s2);
```

respectively.

For the production

```
stmt ::= LBRACE stmts:s RBRACE
```

add the action

```
RESULT = s;
```

Finally, the production

```
cexpr ::= expr:e
```

requires the action

```
RESULT = SemanticAction.exprToCExpr(e);
```

We will write the semantic action functions one by one as we consider the different types of statements.
Now we will begin to do the backpatching, beginning with sequences of statements. The production is

\[
stmts \rightarrow stmts_1 \ m \ stmt
\]

Draw a diagram that shows how the “branching” goes. See Figure 15.2.

This diagram indicates that the linked list from \(stmts_1\) should be backpatched to the label \(m\), and that the production should return the linked list from \(stmt\) (as \(RESULT\)), to be resolved at some higher level in the parse tree. Thus, the \(stmts()\) function should be

```java
public static LinkedList stmts(LinkedList s1, Integer m, LinkedList s2) {
    backpatch(s1, m);
    return s2;
}
```

For all the hoopla, that was awfully simple. Once we finish with conditional expressions, it will be just about as simple to deal with \(if\) statements. That is the power of organization.

### 15.13 Conditional Expressions

When we use a numerical expression as a conditional expression in an \(if\) statement, it is interpreted as true if its value is zero and false if its value is nonzero. Thus, we must compare the numerical value to 0. That means that the form

\[
IF (cexpr) \ m \ stmt
\]

is logically equivalent to
The “true” destination is \texttt{m} and the false destination is whatever label follows the \texttt{if} statement.

Altogether we will have six different kinds of comparison nodes: \texttt{CMPEQ}, \texttt{CMPNE}, \texttt{CMPLT}, \texttt{CMPGT}, \texttt{CMPLE}, and \texttt{CMPGE}. They correspond to the C operators \texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, and \texttt{>=}. Right now we need only the \texttt{CMPNE}, because the expression counts as true if it does not equal 0. The general form of a comparison tree is shown in Figure 15.3.

Our grammar contains the production

\[
\texttt{cexpr} \rightarrow \texttt{expr}
\]

The action to be taken by the \texttt{exprToCExpr()} function is to construct a special \texttt{CMPNE} tree that looks like the tree in Figure 15.4.

If \texttt{e.mode} is \texttt{DOUBLE}, then there will have to be a \texttt{CAST} node on the left side, casting the 0 as a \texttt{double}. You will have to add \texttt{Ops.CMPEQ} to \texttt{Ops.java} and \texttt{Utility.java}.

The \texttt{CMPNE} tree must now be attached to a “jump-true” tree (\texttt{JUMPT}). A \texttt{JUMPT} tree has a boolean expression in its right subtree and a \texttt{BLABEL} tree or a \texttt{LABEL} tree.
in its left subtree. If the boolean value is true, it takes the jump. Otherwise, it continues on to the next instruction. The next instruction should be a JUMP tree that unconditionally jumps to the “false” destination. (Ops.JUMPT and Ops.JUMP must also be added to Ops.java and Utility.java.)

The phrase

\[
\text{if (a)}
\]

would create the following combined JUMPT and CMPNE tree and the JUMP tree below it.

\[
\begin{align*}
\text{JUMPT INT} \\
\quad \text{BLABEL blabel=3} \\
\quad \text{CMPNE INT} \\
\qquad \text{NUM INT value=0} \\
\qquad \text{DEREF INT} \\
\qquad \quad \text{ID PTR|INT value="a"} \\
\text{JUMP INT} \\
\quad \text{BLABEL blabel=4}
\end{align*}
\]

This indicates that execution jumps to B3 if \(a\) is nonzero and it jumps to B4 if \(a\) is zero.

15.14 The exprToCExpr() Function

Now we can write the exprToCExpr() function. The function has a single parameter, which is a TreeNode representing an expression tree. The nonterminal cexpr is of BackpatchNode type, so this function should return a BackpatchNode object.

The first thing the function should do is to create the CMPNE tree. Then it must generate a new label lab1 (by calling newLabel()) and build a BLABEL node, which will serve as the “true” destination.

Then construct the JUMPT tree, attaching the comparison tree on the right and the BLABEL tree on the left. (A JUMPT tree has no particular mode.) This tree is now complete and should be printed.
Next, generate a new label $labl_2$, use it to create a BLABEL node, and then construct a JUMP tree with the BLABEL node attached on its left. This is the “false” destination. Print this tree.

The final step is to create and return a BackpatchNode containing the “true” and “false” destinations of $cexpr$. Use a BackpatchNode constructor, where $labl_1$ is the “true” destination and $labl_2$ is the “false” destination.

The hard work is over. Now we can write the functions that handle the one-way and two-way if statements.

### 15.15 The One-Way if Statement

The production for an if statement is

\[
stmt \rightarrow \text{IF} (cexpr) \ m \ stmt
\]

The branching is as shown in Figure 15.5.

This tells us that the “true” destination ($trueList$) of the $cexpr$ BackpatchNode should be backpatched to $m$. The “false” destination ($falseList$) should be merged with the LinkedList from $stmt$ and returned as the LinkedList for the if statement as a whole.

The heading of the $ifStmt()$ function is

```java
public static LinkedList ifStmt(BackpatchNode c, Integer m, LinkedList s)
```

Write the $ifStmt()$ function.
15.16 The Two-Way if Statement

The production for the two-way if statement is

\[
stmt \rightarrow \text{IF ( cexpr ) } m_1 \text{ stmt}_1 n \text{ ELSE } m_2 \text{ stmt}_2
\]

The branching is as shown in Figure 15.6.

From this diagram, determine what needs to be backpatched, what needs to be merged, and what needs to be returned by the ifElseStmt() function. Then write the function.

15.17 Function Ends

As mentioned earlier, the function func() now has two more parameters: the linked list s and the integer m. However, its return type is still void, because the end of a function marks the termination of backpatching at that level; no backpatch label can be resolved to an actual label outside of that function. The diagram for the end of a function is very simple. See Figure 15.7.

From this diagram, figure out how the backpatching should be done and modify the func() function accordingly.
15.18 The READ Case

Now that we can use the `printf()` function, there is no need for the code generated by the READ case of `generateNodeCode()` to generate code for a prompt. Remove that part of the code generated in the READ case. Henceforth, if we want to prompt the user for input, we can output the prompt using the `printf()` function.

15.19 Testing the Compiler

Be sure to test your compiler thoroughly. Write test programs with one-way and two-way if statements. Write test programs with nested if statements. Test your compiler with multi-way if statements such as

```c
if (a)
    d = 100;
else if (b)
    d = 200;
else if (c)
    d = 300;
else
    d = 400;
```

Now that we have if statements, we can write test programs with recursive function calls. You might try a recursive version of the `gcd()` function.

```c
int gcd(int a, int b)
{
    if (a % b)
        return gcd(b, a % b); // a % b != 0
    else
        return b; // a % b == 0
}
```

This implementation of `gcd()` assumes that `a` is nonnegative and `b` is positive.

Another function you might test is the `Hanoi()` function. Its prototype is

```c
int Hanoi(int num, int src, int dst, int extra);
```
The function ought to return `void`, but we have no `void` type, so we will make the return type `int` and return a 0. The parameter `num` is the number of disks to be moved. As long as it is more than 1, then we will make a recursive call. If it is 1, then we will just move the disk. The parameters `src` and `dst` are the source and destination posts. (The posts are numbered 1, 2, and 3, with the disks originally on post 1 with the goal of moving them to post 3.) The parameter `extra` is the remaining post. You will find this function in the file `Hanoi.c`.

Have the main function read the number of disks from the user, using our special `read` statement. Call that number `n`. Then the initial function call to `Hanoi()` from `main()` should be

```
Hanoi(n, 1, 3, 2);
```

### 15.20 Assignment

Write the production for a `while` loop. The form is similar to the form of the one-way `if` statement, except that at the bottom of the loop, there is an unconditional branch back to the conditional expression. Test your program with `if` statements and `while` loops nested in various ways.

Place all of the source files, including a makefile, in a folder named `Lab_15`, zip it, and drop it in the dropbox.
Laboratory 16

Control Flow Structures and Code Generation

Key Concepts

- Labels
- Equate statements
- Unconditional jumps
- Conditional jumps
- Condition code testing

Before the Lab

Read Sections 3.4.3 and 8.1.2 in Intel’s Developer’s Manual, Vol. 1. These sections discuss flags and condition codes. In Vol. 2, read about the FUCOMPP instruction and the Jcc and JMP instructions. These instructions are used in conditional and unconditional branches.

Preliminaries

Copy your source files from your Lab_15 folder to a new folder called Lab_16.
16.1 Introduction

Most of the code generated in this lab is straightforward. The one feature that is new to us is testing the condition codes for equality or inequality. You should read the Intel manuals to become familiar with how conditional jumps are handled. The single most important instruction is \( J_{cc} \), where \( cc \) stands for a condition code.

We will generate code for the following types of tree node:

\[
\begin{align*}
\text{Ops.LABEL} \\
\text{Ops.EQU} \\
\text{Ops.JUMP} \\
\text{Ops.JUMPT} \\
\text{Ops.CMPNE}
\end{align*}
\]

16.2 The LABEL Case

The purpose of a LABEL node is simply to print a label. A label has the form

\[ \text{Ln:} \]

where \( n \) is the number of the label. For example, it might be

\[ \text{L4:} \]

A LABEL tree has the form

\[ \text{LABEL label=n} \]

where \( n \) is the number of the label.

Write the code for the LABEL case.

16.3 The EQU Case

An equate statement is an assembler directive. It tells the assembler to assign to one symbolic name the value held by another symbolic name. In our case, the equate statement will assign the value (address) of an actual label to a backpatch label. Thus, its form will be

\[ \text{Bn1=Ln2} \]
where \( n_1 \) is the number of the backpatch label and \( n_2 \) is the number of the actual label. For example, if the tree is

```
EQU
  BLABEL blabel=6
  LABEL label=8
```

then the generated code will be

```
B6=L8
```

An EQU tree has a LABEL tree as a subtree. As a subtree of an EQU tree, the LABEL tree should not be processed as described above. Therefore, we must make a special case of an EQU tree. Write the code for the EQU case as a special case in the `generateCode()` function. Recall that ALLOC, FEND, and FUNC were also special cases.

### 16.4 The JUMP Case

A JUMP tree has the following form.

```
JUMP INT
  BLABEL blabel=n
```

where \( n \) is the number of the backpatch label. In some cases, it is possible that the destination will be an actual label (LABEL) rather than a backpatch label (BLABEL). This will happen with backward jumps since the destination is known. All forward jumps will be jumps to backpatch labels.

An unconditional jump statement will have the form

```
jmp   Bn
```

or

```
jmp   Ln
```

where \( n \) is the number of the label. The JUMP case must also be handled as a special case for the same reason that EQU was special: the LABEL or BLABEL subtrees should not be handled as an ordinary LABEL or BLABEL case.
16.5 The CMPNE Case

Now things get a little more complicated. The effect of the CMPNE case is to leave an integer 0 or 1 on the stack, where 0 signifies false and 1 signifies true. This value will be used by the JUMPT node to decide whether to jump.

This method is somewhat inefficient since it requires that we perform two tests. First, we do the original test and store a true or false value on the stack. Then, later, we must test the value on the stack to see if it is true or false. Why not just make the jump after performing the first test? The reason is so that we can disentangle the compiling of the CMPNE and the JUMPT cases from one another. It is simpler if we deal with them independently.

The following is an example of a CMPNE tree.

```
CMPNE INT
  NUM INT value=0
  DEREF INT
    ID PTR|INT value="a"
```

Note that the left subtree is a NUM node containing the number 0. Later when we consider general boolean expressions, this may not be 0. The right subtree is the numerical expression that appeared as the conditional expression in the if statement.

The form of the generated code for an integer comparison is

```
mov $1,%ecx # Set ecx to true (1)
pop %eax # Load right operand
pop %edx # Load left operand
cmp %eax,%edx # Compare operands
jne L02 # Jump to L02 if left != right
dec %ecx # Set ecx to false (0)
```

```
L02:
push %ecx # Push T/F result
```

Notice that we first put 1 (true) in register ecx. Then if eax does not equal edx, execution jumps to L02, leaving 1 in ecx. The mnemonic jne means “jump on not equal.” However, if eax equals edx, then execution drops through to the statement that decrements ecx, making it 0 (false). In either case, the value of ecx is pushed onto the stack.
The code is a little different if the comparison is between floating point quantities.

```
mov $1,%ecx  # Set ecx to true (1)
fucompp     # Compare operands
fnstsw %ax   # Move status word to ax
sahf        # Store ah in eflags
jne L02     # Jump to L02 if left != right
dec %ecx     # Set ecx to false (0)
L02:
push %ecx    # Push T/F result
```

Look up the mnemonics `fucompp`, `fnstsw`, and `sahf` in Intel’s Developer’s Manual, Vol. 2A. The instruction `fucompp` will compare the two operands on top of the floating-point stack and pop them both. It also sets certain bits (C0, C2, and C3) in the floating-point status word, depending on how the comparison turns out. The next instruction `fnstsw` will store the 16-bit FPU status word in register `ax`. Then the instruction `sahf` stores `ah` in the `eflags` register. See Intel’s Developer’s Manual, Vol. 1, Section 8.1.2, x87 FPU Status Register, and Section 8.1.3, Branching and Conditional Moves on Condition Codes. You will see that `sahf` moves C0 to CF (carry flag), C2 to PF (parity flag), and C3 to ZF (zero flag). These flags are automatically tested when a conditional jump such as `jne` is executed.

The number of the label L02 was gotten from the `jmpLabel` class variable in the `CodeGenerator` class. Be sure to include the leading 0 to distinguish L02 from L2, which occurs elsewhere.

When you write the code for the `CMPNE` case, note that the integer and floating-point cases begin and end with the same code. Only the middle parts are different.

### 16.6 The JUMPT Case

The JUMPT case should pop the boolean value left on the stack by the `CMPNE` case, test it, and branch if it is true (1). Using the ideas discussed, write the code for the JUMPT case. Allow that the destination of a JUMPT node may be either a backpatch label or an actual label. If it is a backpatch label, then its name should begin with the letter B. If it is an actual, then its name should begin with the letter L.

The following is an example of a JUMPT tree.
JUMPT INT
BLABEL  blabel=3
CMPNE INT
   NUM INT value=0
DEREF INT
   ID PTR|INT value="a"

The JUMPT tree should also be treated as a special case, since the “true” destination label may be an actual label and we don’t want the label printed here as a label. However, we should pass the right subtree to generateTreeCode() for processing. The code generated by the CMPNE tree will leave the boolean value on the stack, which the JUMPT tree will test.

This should complete the code generation for if statements and while statements.

16.7 Testing the Compiler

Test your compiler with very simple programs that contain one-way if statements, two-way if statements, and very simple while loops. Then test it with nested structures: if statements and while loops nested inside of if statements and while loops.

You may use the test programs gcd.c, gcd2.c, and Hanoi.c. The programs gcd2.c and Hanoi.c use recursive function calls.

16.8 Assignment

This lab will serve as Project 7. Place all of your source files in a folder named Project 7, then zip the folder and drop it in the dropbox.
Laboratory 17

Boolean Expressions

Key Concepts

- Relational operators
- Boolean operators

Before the Lab

Read Sections 8.4 and 8.6 in Compilers: Principles, Techniques, and Tools.

Preliminaries

Copy your source files from the Lab_16 folder to a new folder named Lab_17.

17.1 Introduction

The purpose of this lab is to implement the boolean operators `&&`, `||`, and `!`, and the relational operators `==`, `!=`, `<`, `<=`, and `>=`. All of these operators appear in the productions for conditional expressions.

\[
\begin{align*}
cexpr & \rightarrow \ expr \ EQ \ expr \\
cexpr & \rightarrow \ expr \ NE \ expr \\
cexpr & \rightarrow \ expr \ LE \ expr \\
cexpr & \rightarrow \ expr \ GE \ expr \\
cexpr & \rightarrow \ expr \ LT \ expr
\end{align*}
\]
17.2 Version 5

This is now Version 5 of our compiler. Make the necessary change in Utility.java, compiler_v4.java, and the makefile.

17.3 The Relational Operators

We will deal with the six relational operators as a group, similar to the way we dealt with the four basic arithmetic operators. For each type of relation there will be a type of tree node. We already have the constant Ops.CMPNE. Therefore, we need to introduce the constants Ops.CMPEQ, Ops.CMPLT, Ops.CMPGT, Ops.CMPEQ, and Ops.CMPGE. The general form of a comparison tree is seen in Figure 17.1, where \( cc \) stands for “condition code,” which may be EQ, NE, LT, GT, LE, or GE. The semantic action for each of the six productions is to build this tree, placing the appropriate comparison at the root node.
Name the semantic action function \texttt{relOp()}. The header of the function should be

\begin{verbatim}
public static BackpatchNode relOp(int op, TreeNode e1, TreeNode e2)
\end{verbatim}

where \texttt{op} is the relational operator (e.g., \texttt{Ops.CMPNE}) and \texttt{e1} and \texttt{e2} are the expressions that are being compared. The only issue other than building the tree is to determine the mode of the operations. That will follow the same rule used in the function \texttt{arith()}. If both operands are \texttt{int}, then the operation is \texttt{int}. If either operand is \texttt{double}, then the operation is \texttt{double}, with possible casting.

Once the comparison tree is built, it must be attached to a \texttt{JUMPT} tree. To do this, we will have \texttt{relOp()} call a function \texttt{relOpToCExpr()} which will be nearly identical to the \texttt{exprToCExpr()} function that we wrote earlier. In fact, it is so similar that you might want to copy, paste, and edit \texttt{exprToCExpr()} to create \texttt{relOpToCExpr()}. The only difference is that \texttt{relOp()} passes to \texttt{relOpToCExpr()} the comparison tree already built. In \texttt{exprToCExpr()}, it was necessary to build the \texttt{CMPNE} tree first. Therefore, the code in \texttt{relOpToCExpr()} should be identical to the subsequent code in \texttt{exprToCExpr()}. Just as in \texttt{exprToCExpr()}, the function \texttt{relOpToCExpr()} should return to \texttt{relOp()} a \texttt{BackpatchNode} containing the “true” and “false” destinations of the \texttt{JUMPT} tree and the \texttt{JUMP} tree that were printed.

Write the \texttt{relOp()} and \texttt{relOpToCExpr()} functions in \texttt{SemanticAction.java} and add the function calls in the CUP file.

### 17.4 The Boolean Operators

The boolean operators \texttt{&&}, \texttt{||}, and \texttt{!} are more interesting because they involve backpatching. As such, they will build various \texttt{JUMP} and \texttt{JUMPT} trees.

Consider first the \texttt{!} operator. The semantic action function will be the \texttt{notOp()} function. The “not” operator is applied only to Boolean expressions, which are represented in the grammar as \texttt{BackpatchNodes}, with a “true” list and a “false” list of destinations. The only thing to be done is to swap the true and false lists. That is, create and return a new \texttt{BackpatchNode} with the lists reversed.

The \texttt{andOp()} function will perform the action for the \texttt{&&} operator. The conjunction of two boolean expression is again a boolean expression. That means that the \texttt{andOp()} function should return a \texttt{BackpatchNode}. The function will require backpatching because of the “short-circuit” evaluation of “and.” Recall that for the expression \texttt{p
\&\& q to be true, both p and q must be true. Therefore, if we find p to be false, then there is no need to evaluate q. In fact, this method of evaluation is required by the ANSI C standard. To see how to implement this, we need a diagram. See Figure 17.2.

This indicates that the “true” list from cexpr_1 should be backpatched to m and the “false” list from cexpr_1 should be merged with the “false” list from cexpr_2. (Note that m must be one of the parameters of andOp().) Then a BackpatchNode should be constructed and returned that has the “true” and “false” lists indicated in the diagram. Write the andOp() function.

The “or” operator || is very similar to the “and” operator. In this case, short-circuit evaluation of p || q means that if p is true, then there is no need to evaluate q. Write the orOp() function.

Finally, there is the production

\[ cexpr \rightarrow LPAREN cexpr RPAREN \]

This is similar to the production

\[ expr \rightarrow LPAREN expr RPAREN \]

Using that production as a guide, write the appropriate action.

### 17.5 Testing the Tree-Building

Before generating assembly code, it would be a good idea to pause and test the tree building, especially the boolean operators. Try test programs with various combinations of \&\&, ||, and !. Also, be sure that the precedence, associativity, and short-circuit rules are working.
17.6 Code Generation

Let’s generate code first for the relational operators. The pattern has already been established in the CMPNE case that we wrote in the previous lab.

In an integer comparison, we used the jne mnemonic which left true (1) in register ecx if the operands were not equal. The alternative was to drop through and change the value in ecx to false (0). In a similar way, CMPEQ will use je, and so on. Look in Intel’s Developer’s Manual for the other conditional jump mnemonics that are available.

Then write the cases for CMPEQ, CMPLT,CMPGT, CMPLE, and CMPGE. You can copy and paste from CMPNE into the other cases, modifying jne to the appropriate mnemonic.

For floating-point comparisons, you must use the conditional jumps ja, jae, jb, and jbe. That is because the code for floating-point comparisons sets different flags. Recall that fucompp reverses the expected order of the operands, so you should use the reverse condition from the expected one. In other words, to test for less than, you should use ja (jump above), to test for greater than, you should use jb (jump below), and so on.

You might review again Sections 8.1.2 and 8.1.3 in Intel’s Developer’s Manual, Vol. 1, and read about the fucompp and jcc instructions in Vol. 2.

17.7 Testing the Code-Generation

Test a variety of Boolean expressions. Test each relational operator at least once and each Boolean operator at least once. Test combinations of Boolean operators to see if everything works correctly. You may use the test program prime.c.

17.8 Assignment

Implement do loops and for loops. The productions for these statements are

\[
\begin{align*}
stmt & \rightarrow \text{DO } stmt_1 \text{ WHILE LPAREN } cexpr \text{ RPAREN SEMI} \\
stmt & \rightarrow \text{FOR LPAREN } expr_1 \text{ SEMI } cexpr \text{ SEMI } expr_2 \text{ RPAREN } stmt
\end{align*}
\]

I recommend that you begin with the do loops, since they are easier. In a do loop, \(stmt_1\) is executed unconditionally on the first pass. Then \(cexpr\) is evaluated. If it is
false, then execution exits the loop. If it is true, then $stmt_1$ is executed again and then $cexpr$ is evaluated again, and so on.

When the `for` loop is executed, $expr_1$ is evaluated first and unconditionally. Next, $cexpr$ is evaluated. If it evaluates to false, then execution exits the `for` loop. If it is true, then $stmt$ is executed. After $stmt$ is executed, $expr_2$ is evaluated and then $cexpr$ is evaluated again, with the same consequences as previously described.

You will have to figure out where to use the markers $m$ and $n$. Then draw the backpatching diagram. Be sure that all the parts are connected and that the final exit from each kind of loop is the false destination of $cexpr$. You should find that the abstract syntax tree and the assembly code are created automatically by existing functions. You may use the program `pi.c` as a test program.

This lab will serve as Project 8. Copy your source files and the makefile to a folder name `Project_8`, zip it, and drop it in the dropbox.