Modern Compiler Design

Associated Supplemental Materials

C Implementation Details

David Galles

Contents

Chapter 1

Introduction & Explanation

1.1 What Is This Document?

This document is a companion to the textbook *Modern Compiler Design* by David Galles. The textbook covers compiler design theory, as well as implementation details for writing a compiler using JavaCC and Java. This document contains all of the implementation details for writing a compiler using C, Lex, and Yacc. Note that this document is not self contained, and is only meant to be used in conjunction with the textbook. Much of the material in this document will be difficult to understand if read in isolation.

1.2 How To Use This Document

This document is designed to be used in conjunction with the textbook `Compiler Design`. The chapters in this document correspond to the chapters in the textbook. For instance, Chapter 2 in the text covers lexical analysis, and Chapter 2 in this document covers writing a lexical analyzer in C. First, read the main textbook, starting with Chapter 1. When the textbook covers implementation details using Java, refer to the corresponding chapter of this document for an equivalent description in C.[1]

1.3 So Where Are Chapters 3 and 4?

Chapter 3 in the textbook covers Context-Free Grammars, and is a "theory only" chapter that contains no implementation details. Hence, there is no need for additional materials covering C implementation for Chapter 3. Chapter 4 in the textbook covers Top-Down parsing, while Chapter 5 in the textbook covers Bottom-Up parsing. Since JavaCC is a Top-Down parser generator, and Yacc is a Bottom-Up parser generator, Chapter 4 has no associated C implementation details (and hence does not appear in this document), which chapter 5 has no Java implementation details (though Bottom-Up parsing theory is indeed covered in the textbook.

1.4 Where Can I Find Necessary Files for Creating a Compiler in C?

The CD on which you found this document contains all necessary files for writing a compiler in either C or Java. You can also find updated versions of these files, as well as errata, at www.cs.usfca.edu/galles/compilertext

[1] With one exception – JavaCC is a Top-Down parser generator, so the Java implementation details for parser generation are in Chapter 4, which covers Top-Down parsing. Yacc is a Bottom-Up parser generator, so the C implementation details for parser generation are in Chapter 5 of this document, which covers Bottom-Up parsing.

Chapter 2

Lexical Analysis

Chapter Overview

2.1 Lex – A Lexical Analyzer Generator

Lex is a tool that takes as input a set of regular expressions that describe tokens, creates a DFA that recognizes that set of tokens, and then creates C code that implements that DFA. A lex file consists of regular expression / action pairs, where actions are represented by blocks of C code. Given a lex file, lex creates a definition of the C function

```
int yylex(void)
```

When the function yylex is called, the input file is examined to see which regular expression matches the next characters in the input file. The action associated with that regular expression is performed, and then lex continues looking for more regular expression matches. In pseudo-code:

```
int yylex() {
   while(TRUE) {
     find a regular expression  in the set of regular expression / action pairs
       that matches the prefix of the current input file
     execute the action associated with that regular expression
   }
```

2.1.1 Structure of a Lex file

Input files to lex have the extension ".lex", and have the following file format:

```
%{
/* C Declarations -- #includes, function definitions, etc */
%}

/* Lex Definitions */

%%

/* Lex rules -- Regular expressions / action pairs */

%%
```

The first section of the lex file (between %{ and %}) contains #includes and C definitions that can be used in the rest of the file. Note that both %{ and %} need to appear unindented on a line by themselves.

The second portion of the lex file (between%} and %%) contains simple name definitions and state declarations (name definitions are covered in section 2.1.2, while state declarations are covered in section 2.1.6)

The third portion of the lex file (between the %%'s) contains the real meat of the lex file – lex rules, in the form of regular expression / action pairs. This is where we define the tokens we want the lexer to recognize, and what to do when we recognize a token.

Let's take a look at the lex file simple.lex, in Figure 2.1. The C definitions file contains a single include statement:

```
#include "tokens.h"
```

The file tokens.h includes the definition of the symbolic constants ELSE, SEMICOLON, FOR, INTEGER_LITERAL, and IDENTIFIER, which can now be used in the body of the simple.lex file. There are no name definitions or state declarations in this .lex file. The lex rule segment of this .lex file has four lex rules, each of which consists of a regular expression / action pair. The first rule:

```
else        { return ELSE; }
```

defines a begin token. The regular expression `"BEGIN"` matches to a single string, "BEGIN". Begin is a reserved word in lex, which is why this regular expression needs to be in quotation marks. The regular expression for the second rule:

```
";"     { return SEMICOLON; }
```

also matches to a single string, ";". It is not required to include the quotation marks for a semicolon, so the rule:

```
;     { return SEMICOLON; }
```

would also be acceptable. Since many characters (such as *, +, (and)) have special meanings in lex, it is usually a good idea to always use quotation marks for non-alphanumeric characters. The last two lex rules:

```
[0-9]+                  { return INTEGER_LITERAL; }
[a-zA-Z][a-zA-Z0-9]*    { return IDENTIFIER; }
```

define integer literals and identifiers. The regular expression `[0-9]+` matches to any sequence of digits, such as "341", "5" and "985126". The regular expression `[a-zA-Z][a-zA-Z0-9]*` matches to any string of letters and digits that begins with a letter, such as "foo12", "rainforest" and "pixel".

Let's take a look at how the lexical analyzer generated form the file simple.lex works on the following input file:

```
%{
#include "tokens.h" /* includes definitions of the integer constants
                        ELSE SEMICOLON FOR INTEGER_LITERAL IDENTIFIER  */
%}

%%
else                     { return ELSE;
";"                      { return SEMICOLON; }
for                      { return FOR; }
[0-9]+                   { return INTEGER_LITERAL; }
[a-zA-Z][a-zA-Z0-9]*     { return IDENTIFIER; }
%%
```

Figure 2.1: A Very simple lex file, simple.lex

```
else;14for5;
```

The first time yylex() is called, it tries to match the input with a regular expression in one of the rules. What if there is more than one rule that matches the next segment of the input file? In this example, we could match "e" to the IDENTIFIER rule, "el" to the IDENTIFIER rule, "els" to the IDENTIFIER rule, "else" to the IDENTIFIER rule, or "else" to the ELSE rule. When there is more than one match, lex uses the following strategy:

1. Always match to the longest possible string.
2. If two different rules match the same longest string, use the regular expression that appears first in the input file.

The longest string that matches a one of our lex rules is "else". There are two rules that match "else", the ELSE rule and the IDENTIFIER rule. Since ELSE appears first in the lex file, yylex will execute the action

```
return ELSE;
```

This action contains a return statement, yylex will stop trying to match strings, and return the constant ELSE (defined in "tokens.h"). The second time yylex is called, the only regular expression that matches is ";", so yylex will execute the action

```
return SEMICOLON;
```

The third time yylex() is called, it will match 14 to the regular expression `[0-9]+` and return INTEGER_LITERAL; Unfortunately, while yylex returns the information that the matched token was in integer literal, it does not tell us *which* integer literal. Specifically, lex does not return the value 14 – it just returns the fact that an integer appeared in the input file. We'd like yylex to return 2 pieces of information in this instance – both the fact that the next token was an integer literal, and that the value of that literal is 14. We'll see how to do that in the next example (Figure 2.2). The fourth time yylex is called, it does not return FOR – since that is not the longest possible match. The longest possible match is an IDENTIFIER – "for5".

2.1.2 Named Regular Expressions

It is possible for the regular expressions used in lex rules can get quite complicated, making them hard to read. Lex allows us to break regular expressions into smaller pieces, and give those regular expression fragments symbolic names. We can then use the symbolic names in other regular expressions, to make our rules more readable. In the Lex Definitions segment of a .lex file, a name definition has the form:

name regular_expression

Once a name is defined, it can be used in a regular expression for a lex rule by enclosing it in braces { and }. For instance, if the line

```
DIGIT       [0-9]
```

appeared in the Lex Definitions of a .lex file, then {DIGIT} would be a symbolic name for the regular expression [0-9]. Anywhere [0-9] appears in a regular expression, we could use {DIGIT} instead. Thus the rule:

```
{DIGIT}+              { return INTEGER_LITERAL; }
```

would be equivalent to the rule:

```
[0-9]+              { return INTEGER_LITERAL; }
```

An example of using named regular expressions is in Figure 2.2.

2.1.3 Tokens With Values

For tokens that don't have values – ELSE, FOR, SEMICOLON, etc – we only care about which rule was matched. No other information is needed. Some tokens, such as INTEGER_LITERAL and IDENTIFIER, have values. 57, for instance, is an INTEGER_LITERAL token that has the value 57. foobar is an IDENTIFIER token that has the value "foobar". How can we have yylex return both which token was matched, and the value of the token? We can use a global variable to communicate the extra information. yylex() can set the value of the global variable before returning the token type. The function that calls yylex() can then examine the value of this global variable to determine the value of the token. Let's take a look at a slightly more complicated lex file, simple2.lex (Figure 2.2). In the C declarations portion of the file, we have defined a global variable yylval[1].

```
union {
    int integer_value;
    char *string_value;
} yylval;
```

When we wish to return the value of a token, we will set the value of this global variable. Consider the rule for INTEGER_LITERAL:

```
[0-9]+                  { yylval.integer_value = atoi(yytext);
                          return INTEGER_LITERAL; }
```

When we match an integer in the input file, we first set the integer_value field of the global variable yylval. To set the integer_value field to the correct value, we need to know the actual string that was matched to the regular expression `[0-9]+`. The variable yytext is automatically set to the value of the last string matched by lex. Thus, all we need to do is convert that string to an integer value through the ascii-to-integer function atoi defined in <string.h>, and set the integer_value field of the variable yylval. We can then return an INTEGER_LITERAL token.

The other token in this example that has a value is the IDENTIFIER token. The rule for an IDENTIFER is:

```
[a-zA-Z][a-zA-Z0-9]*    { yylval.string_value = malloc(sizeof(char) *
                                                (strlen(yytext)+1));
                          strcpy(yylval.string_value,yytext);
                          return IDENTIFIER; }
```

[1]Why do we call this variable yylval instead of a more meaningful name like tokenValue? For the final project, we will use lex in conjunction with another tool, yacc. Yacc has a rather odd naming convention, and we have to follow it for the two tools to work together.

```
%{
#include "tokens.h" /* includes definitions of the integer constants
                       ELSE SEMICOLON FOR INTEGER_LITERAL IDENTIFIER  */
#include <string.h>

union {
     int integer_value;
     char *string_value;
} yylval;
%}
DIGIT    [0-9]
%%
else                    { return ELSE; }
";"                     { return SEMICOLON; }
for                     { return FOR; }
{DIGIT}+                { yylval.integer_value = atoi(yytext);
                          return INTEGER_LITERAL; }
[a-zA-Z][a-zA-Z0-9]*    { yylval.string_value = malloc(sizeof(char) *
                                                (strlen(yytext)+1));
                          strcpy(yylval.string_value,yytext);
                          return IDENTIFIER; }
%%
```

Figure 2.2: Using tokens with values, simple2.lex.

Just as with the INTEGER_LITERAL token, we set the global variable yylval before returning IDENTIFIER. Since the value of an IDENTIFIER is a string, we need to set the string_value field of the yylval variable. The variable yytext is a buffer used by lex – the next time a string is matched, the contents of the buffer will be overwritten by the new string. Thus, we cannot just set `yylval.string_value = yyext` – instead we need to allocate space for yylval.string_value and copy in the value of yytext. Remember that C strings are null terminated – so we need to set aside an extra unit of space for the termination character '\0'.

2.1.4 Dealing with White Space

We would like our lexical analyzer to ignore whitespace – space characters, tabs, and end of lines. When given the input file:

```
else                ident

;
```

we would like the first call yylex() to return a BEGIN token, the second call to yylex() to return an IDENTIFIER token (while setting the string_value field of the yylval variable to the string "ident"), and the third call to yylex() to return a SEMICOLON token. Lex does not automatically skip over white space – in some languages white space is important, and lex needs to be able to process those languages as well as languages where whitespace is ignored. Thus, we need to tell lex what to ignore.

We can make lex ignore whitespace by adding rules that match whitespace that have no actions, as follows:

```
" "                 { }
\t                  { }
\n                  { }
```

Recall that the function yylex() operates as follows:

```
while(TRUE) {
  1. Find a regular expression rule that matches the next section of the input file
  2. Execute the action for that rule
}
```

Thus, if a rule has the action { }, then no action will be performed. The matched string will then be discarded, and yylex() will look for another string to match. Only when an action contains a return statement does yylex stop matching strings and return a value.

2.1.5 Keeping Track of Token Positions

Later on in the compilation process, when we are checking for syntax errors and semantic errors, we will want to know where each token appeared in the original input file. We need this information so that our compiler can give good error messages. "There is a type mismatch in the assignment statement in line 45" is a much better error message than "There is a type mismatch on an assignment statement somewhere in the program." Thus, we'd like tokens like ELSE and SEMICOLON to return a value – the position of the input file where the token occurred – and we'd like the token INTEGER_LITERAL to return two values – both the position of the token, and the value of the integer literal. We can easily modify the yylval variable so that it returns multiple values, as follows:

```
union {
     struct {
         int value;
         int line_number;
     } integer_value;
     struct {
         char *value
         int line_number;
     } string_value;
     int line_number;
} yylval;
```

Thus, in the action for the INTEGER_LITERAL rule, we will first set both `yylval.integer_value.value` to the value of the integer literal, and `yylval.integer_value.line_number` to the position of the integer literal in the input file before returning the constant INTEGER_LITERAL. For tokens like END and SEMICOLON that do not have values, we merely need to set `yylval.line_number` to the correct value.[2]

Of course, in order to set the line number field correctly, we will need some way of keeping track of the line number of every token in the input file. We will maintain a current_line_number variable, whose value will always be the current line number in the input file. Thus, every time an end-of-line character is matched, the current position will need to be updated. A function `newline()`, which updates the current line number, can be called each time a newline character is matched. The file simple3.lex (Figure 2.3) has examples of using lex to keep track of token positions.

2.1.6 States in Lex

In addition to skipping over whitespace in the input file, we would also like our lexical analyzer to skip over comments. We could skip over comments in the same way that we skip over white space. For instance, a C++ single line comment headed by "//" could be skipped using the lex rule:

```
"//"[^\n]*\n    { }
```

[2]Why have structs inside unions, instead of three separate global variables `int integer_value`, `char *string_value` and `int line_number`? We will be using lex in conjunction with yacc, which expects all extra token information to be stored in a single union variable.

```
%{
#include "tokens.h"
#include <string.h>

union {struct {
         int value;
         int line_number;
       } integer_value;
       struct {
         char *value
         int line_number;
       } string_value;
       int line_number;
} yylval;
int current_line_number = 1;

void newline() {
  current_line_number++;
}
%}
%%
else                    { yylval.line_number = current_line_number;
                          return ELSE; }
" "                     {     }
\n                      { newline(); }
";"                     { yylval.line_number = current_line_number;
                          return SEMICOLON; }
for                     { yylval.line_number = current__line_number;
                          return FOR; }
[0-9]+                  { yylval.integer_value.line_number = current_line_number;
                          yylval.integer_value.value = atoi(yytext);
                          return INTEGER_LITERAL; }
[a-zA-Z][a-zA-Z0-9]*    { yylval.string_value.line_number = current_line_number;
                          yylval.string_value.value = malloc(sizeof(char) *
                                                      (strlen(yytext)+1));
                          strcpy(yylval.string_value.value,yytext);
                          return IDENTIFIER; }
%%
```

Figure 2.3: Keeping track of token positions, simple3.lex.

It is more difficult (though possible) to write a regular expression that matches multi-line comments separated by /* and */, but it is impossible to write a regular expression that matches to nested comments correctly. Fortunately, lex provides a mechanism that allows for easy handling of comments – lexical states.

Each regular expression in the lex file can be labeled with a lexical state. Rules that are unlabeled are considered to be in the default INITIAL lexical state. The lex starts out in the INITIAL state, and will only match regular expressions for the INITIAL state. We can switch states in the action for a lex rule with the BEGIN(NEWSTATE) command. For instance, the action for the rule:

```
"/*"                    { BEGIN(COMMENT); }
```

switches the current lexical state to the COMMENT state. Once in the COMMENT state, only regular expressions labeled with the COMMENT state will be matched. Rules are labeled by placing <STATENAME> in front of the regular expression. So the rule:

```
<COMMENT>"*/"           { BEGIN(INITIAL); }
```

is a COMMENT rule, and will only be matched when lex is in the COMMENT state. Once this rule is matched, lex switches back to the default INITIAL state. We can use lexical states to skip over comments. All rules for tokens are in the INITIAL state. When an opening comment delimited "(*" is seen, the lexical state changes to COMMENT. In the COMMENT state, all input is discarded until a closing comment delimiter "*)" is seen, when the lexical state switches back to INITIAL for processing more tokens.

All lexical states (other than INITIAL) need to be declared before they are used. Lexical states are declared in the Lex Definitions segment of the .lex file. Each lexical state needs to be defined in the lex definition segment of the .lex file as follows:

```
%x STATENAME
```

The "%x" in the state definition stands for eXclusion – while in the state STATENAME, only rules labeled STATENAME are active. For instance, consider the lex file simple4.lex, in Figure 2.4. The lex definition segment of the file contains the definition of a single lex state COMMENT:

```
%x STATENAME
```

There are three rules for the comment state. Note that < and > are required in the syntax for labeling rules with states. Two regular expressions are used to skip over the body of a comment, both . and \n, since . matches to any single character except newline.

2.1.7 Using Lex with Other C Programs

When we run lex on a .lex file, the C file lex.yy.c is created. This file describes the function yylex(), which scans the input file for the next token, based on the regular expression rules. What input file does yylex use? Lex defines a global file descriptor variable yyin, which points to the file that lex will use for input. By default, this file descriptor points to standard input, but we can change it if we like. Consider the driver program in Figure 2.5. This program uses the command line argument to set the filename that lex uses, then repeatedly calls yylex to get the next token in that file, printing the token number of that token to standard out.

2.1.8 Advanced Lex Features

Lex is a very powerful file processing tool, which can be used for many of the same tasks as the programming language perl. The focus of this chapter is on using lex to create a lexical analyzer for a compiler, but we will take just a little time to outline some of the advanced features and uses of lex.

```
%{
#include "tokens.h" /* includes definitions of the integer constants
                       ELSE and FOR   */
%}

%x COMMENT

%%
else                    { return ELSE;
for                     { return FOR; }
"(*"                    { BEGIN(COMMENT); }
<COMMENT>"*)"           { BEGIN(INITIAL); }
<COMMENT>\n             { }
<COMMENT>.              { }
%%
```

Figure 2.4: Using lex states, file simple4.lex.

```
#include <stdio.h>
#include "tokens.h"

extern FILE *yyin;            /* File descriptor lex uses for the input       */
                              /*   file, defined in yylex                     */

extern union { int integer_value;    /* yylval is the global variable set by  */
               char *string_val;     /*  lex for  tokens with values          */
             } yylval;

int yylex(void);              /* Prototype for main lex function              */


int main(int argc, char **argv) {
  char *filename;
  int token;

  filename=argv[1];              /* Get the filename from the command line     */
  yyin = fopen(filename,"r");    /* Set file descriptor that lex uses          */

  token = yylex();            /* Call yylex to get the next token from lex     */
  while(token) {              /*   yylex returns 0 on end-of-file              */
    printf("Read token # %d \n", token);
                              /*  If a token has a value, we could examine the */
                              /*    appropriate field of yylval here          */
    token = yylex();
  }
  return 0;
}
```

Figure 2.5: A main program that utilizes the yylex function defined by lex.

Default Actions

What happens when no regular expression matches the next characters in the input? Lex will execute the default action, which merely prints the unmatched characters to standard out. The easiest way to understand how default actions work is to assume that the lines:

```
.                { printf("%s",yytext); }
\n               { printf("%s", yytext); }
```

appear at the end of our lex file, to be used if no other rules match the input. These default rules allow us to easily write lex code to make slight modifications to an input file. We only need to specify what changes we want to make, and the rest of the file is left alone. Examples of using lex to transform files can be found in Figures 2.6 and 2.7.

What if we do not want to use the default rules? All we need to do is make sure that our regular expressions match all possible strings. If we have our own rules for the regular expressions "," and "\n", then the default rules will never be used.

yywrap

When lex reaches the end of an input file, it calls the function `int yywrap(void)`. If yywrap returns 1, then yylex exits, returning a 0. If yywrap returns 0, lex will continue to try to match regular expressions to the input file. We can use yywrap to allow lex to analyze several files one after the other, by changing the file pointer yyin and returning 0. An example of using yywrap in this way can be found in the changedates.lex file, in Figure 2.7. If we no not wish to use multiple input files, and we don't need lex to do any cleanup work when the end of the file is reached, then we can use the default version of yywrap:

```
int yyrwap(void) {
   return 1;
}
```

If we do not include a definition of yywrap, lex is *supposed* to use this default definition. However, on some implementations of lex, if we do not include the above definition in the C code segment of lex, the lexical analyzer will not compile correctly.

Including a Main Program in Lex

Any C functions that we place after the final %% in the lex file will be copied directly into the C code that lex generates. We can use this feature to create a stand alone program in lex, by including a main function after the final %%. Consider the lex program in Figure 2.6. When we run lex on this file, and then compile lex.yy.c, we get a complete, stand-alone program that removes the comments from a C source file. How does this program work? After setting yyin (so that lex reads from a file instead of from standard input), the main program calls yylex. Since none of the actions include a `return` statement, yylex will continue matching expressions and executing their actions until the end of file is reached. Then the function yywrap will be called, and yylex will end. The main program will then print out some statistics. Note that we could have put the line

```
   printf("/* Comments removed = %d */", commentsRemoved);
```

in the body of `yywrap` instead of in the main program, without changing the output of the program. Also, since the regular expressions:

```
.                { printf("%s",yytext); }
\n               { printf("%s", yytext); }
```

just implement the default action, we could remove them without affecting the output of our program.

```
%{
int commentsRemoved = 0;

int yywrap(void) {
  return 1;
}
%}

%x COMMENT
%%
"/*"              { commentsRemoved++; BEGIN(COMMENT);}
<COMMENT>"*/"     { BEGIN(INITIAL); }
<COMMENT>\n       { }
<COMMENT>.        {}
.                 { printf("%s",yytext);}
\n                { printf("%s", yytext);}
%%
int main() {
     yylex();
     printf("/* Comments removed = %d */", commentsRemoved);
}
```

Figure 2.6: A Lex file for removing comments from C code.

Example Use of Advanced Lex Features

We will now examine a lex file that uses all of the above advanced features of lex. The problem that we would like to solve is updating the dates in a series of text files. We have a set of text files, which contain (among other information), dates of the form mm/dd/yy or mm/dd/yyyy. We would like to convert all of the dates in all of the input files to the form mm/dd/yyyy. Dates of the form mm/dd/yy that are in the years 00-29 will be assumed to be in the twenty-first century, while dates of the form mm/dd/yy that are in the years 30-99 will be assumed to be in the twentieth century. Our program will input file names from the command line and create new files with the ".new" extension which have corrected dates.

The lex file `changedates.lex`, in Figure 2.7, implements our date updater. Note that the rule

```
[0-1][0-9]"/"[0-3][0-9]"/"[0-9][0-9][0-9][0-9] {
                          fprintf(outfile,"%s",yytext); }
```

is necessary, otherwise the date "10/03/1969" would be converted into the incorrect date "10/03/201969". Also, note how the function `yywrap` checks to see if there are more input files, and either opens a new file and returns 0 (so that the new file will be processed), or returns 1 (to end file processing).

2.2 Creating a Lexical Analyzer for simpleJava in Lex

2.2.1 Project Definition

The first segment of the simpleJava compiler is the lexical analyzer. You will write a lex file sjava.lex which creates a lexical analyzer for simpleJava tokens. A skeleton of sjava.lex can be found in Figure 2.8 to get you started. Once you have successfully built your lexical analyzer, it can be tested using the main function in the file 2.9.

Your lexical analyzer needs to be able to recognize the following keywords and symbols:

```
and class do else false for if true while
+ - * / [ ] \{ \} ( ) . , ; == != < > <= >= = && || !
```

```
%{
  #include <string.h>
  FILE *outfile;
  char **fileNames;
  int currentFile, numFiles;

  void OpenFiles() {
    char outFileName[20];
    printf("Correcting file %s ... \n",fileNames[currentFile]);
    yyin = fopen(fileNames[currentFile],"r");
    strcpy(outFileName,fileNames[currentFile]);
    strcat(outFileName,".new");
    outfile = fopen(outFileName,"w");
  }

int yywrap(void) {
  if (++currentFile == numFiles) {
    return 1;
  } else {
    OpenFiles();
    return 0;
  }
}
%}
DIGIT [0-9]
%%
[0-1][0-9]"/"[0-3][0-9]"/"[0-9][0-9][0-9][0-9] {
                            fprintf(outfile,"%s",yytext); }

[0-1][0-9]"/"[0-3][0-9]"/"[0-2][0-9] {
                             fprintf(outfile,"%c%c/%c%c/20%c%c",
                                     yytext[0],yytext[1],yytext[3],yytext[4],
                                     yytext[6],yytext[7]); }

[0-1][0-9]"/"[0-3][0-9]"/"[3-9][0-9]  {
                            fprintf(outfile,"%c%c/%c%c/19%c%c",
                                    yytext[0],yytext[1],yytext[3],yytext[4],
                                    yytext[6],yytext[7]); }
.                         { fprintf(outfile,"%s",yytext);}
\n                        { fprintf(outfile,"%s", yytext);}
%%
int main(int argc, char **argv) {
  fileNames = &(argv[1]);
  currentFile = 0;
  numFiles = argc - 1;
  OpenFiles();
  yylex();
}
```

Figure 2.7: The lex file changedates.lex, which changes all the dates in a set of files from the form mm/dd/yy to mm/dd/yyyy. Years in the range 00-29 are assumed to be in the twenty-first century, while years in the range 30-99 are assumed to be in the twentieth century.

```
%{
#include <string.h>
#include "errors.h"
#include "tokens.h"

  /* The following code deals with keeping track of */
  /*   where the tokens are in the file.            */

int current_line_number = 1;

void newline() {
  current_line_number++;
}

int yywrap(void) {
  return 1;
}

%}


%%
" "             {  }
\n              { newline();}
else            { yylval.line_number = current_line_number;
                  return ELSE;}
for             { yylval.line_number = current_line_number;
                  return FOR;}
"+"             { yylval.line_number = current_line_number;
                  return PLUS;}
.               { error(current_line_number,"BAD TOKEN %s",yytext);}
```

Figure 2.8: Skeleton pascal.lex file, to get you started on your lexical analyzer.

```
#include <stdio.h>
#include "errors.h"
#include "tokens.h"
extern FILE *yyin;
YYSTYPE yylval;
int yylex(void);
char *tokenNames[] = {"IDENTIFIER","INTEGER_LITERAL","CLASS","DO","ELSE",
                      "TRUE","FALSE","FOR","IF","WHILE","PLUS","MINUS",
                      "MULTIPLY","DIVIDE","LBRACK","RBRACK","LBRACE","RBRACE",
                      "LPAREN","RPAREN","DOT","COMMA","SEMICOLON","EQ","NEQ",
                      "LT","GT","LEQ","GEQ","GETS","AND","OR","NOT","PLUSPLUS",
                      "MINUSMINUS","RETURN",};
char *tokenName(int token) {
  if (token < 257 || token > 292)
    return "BAD TOKEN";
  else
    return tokenNames[token-257];
}
int main(int argc, char **argv) {
 char *filename;
 int token;
 if (argc!=2) {
   fprintf(stderr,"usage: %s filename\n",argv[0]);
   return 0;
 }
 filename=argv[1];
 yyin = fopen(filename,"r");
 if (yyin == NULL) {
   fprintf(stderr,"Cannot open file:%s\n",filename);
 } else {
   token = yylex();
   while(token) {
     switch(token) {
     case INTEGER_LITERAL:
       printf("%17s line number:%2d   %d\n",tokenName(token),
              yylval.integer_value.line_number,
              column,yylval.integer_value.value);
       break;
     case IDENTIFIER:
       printf("%17s line number:%2d    %s\n",tokenName(token),
              yylval.string_value.line_number,
              string_value.value);
       break;
     default:
       printf("%17s line number:%2d   \n",tokenName(token),
              yylval.line_number);
     }
     token = yylex();
   }
 }
 return 0;
}
```

Figure 2.9: File tokentest.c – a main program to test your lexical analyzer.

```
typedef union {
     struct {
         int value;
         int line_number;
     } integer_value;
     struct {
         char *value
         int line_number;
     } string_value;
     int line_number;
} YYSTYPE;

YYSTYPE yylval;

#define IDENTIFIER 257       /* Identifier Token      */
#define INTEGER_LITERAL 258  /* Integer Literal Token */
#define CLASS 259            /* Keyword class         */
#define DO 260               /* Keyword do            */
#define ELSE 261             /* Keyword else          */
#define TRUE 262             /* Keyword true          */
#define FALSE 263            /* Keyword false         */
#define FOR 264              /* Keyword for           */
#define IF 265               /* Keyword if            */
#define WHILE 266            /* Keyword while         */
#define PLUS 267             /* Symbol +              */
#define MINUS 268            /* Symbol -              */
#define MULTIPLY 269         /* Symbol *              */
#define DIVIDE 270           /* Symbol /              */
#define LBRACK 271           /* Symbol [              */
#define RBRACK 272           /* Symbol ]              */
#define LBRACE 273           /* Symbol {              */
```

Figure 2.10: First half of file "tokens.h", which contains all the tokens your lexer needs to recognize.

In addition, your lexical analyzer should recognize integer literals and identifiers. An integer literal is a sequence of digits. An identifier consists of a sequence of letters (both upper and lower case), the underscore character "_", and digits, that begins with a letter or the underscore character.

Your lexical analyzer should skip over comments. Comments in simpleJava are bracketed by /* and */, and can be nested.[3] So /* This is a a comment */ is a legal comment. Since comments can be nested, /* this /* is a */ comment */ is also a legal comment.

2.2.2 Project Difficulty

The lexical analyzer is by far the easiest section of the compiler project. The three main difficulties for this project are learning the syntax of lex and getting nested comments to work correctly. The estimated completion time for the lexical analyzer project is 1-2 hours.

2.2.3 Project "Gotcha"s

- To get nested comments to work correctly, you will need to use some extra C code in the C declaration block, as well as lexical states.

[3]Note that comments do *not* nest in standard Java.

```
#define RBRACE 274          /* Symbol }             */
#define LPAREN 275          /* Symbol (             */
#define RPAREN 276          /* Symbol )             */
#define DOT 277             /* Symbol .             */
#define COMMA 278           /* Symbol ,             */
#define SEMICOLON 279       /* Symbol ;             */
#define EQ 280              /* Symbol ==            */
#define NEQ 281             /* Symbol !=            */
#define LT 282              /* Symbol <             */
#define GT 283              /* Symbol >             */
#define LEQ 284             /* Symbol <=            */
#define GEQ 285             /* Symbol >=            */
#define GETS 286            /* Symbol =             */
#define AND 287             /* Symbol &&            */
#define OR 288              /* Symbol ||            */
#define NOT 289             /* Symbol !             */
#define PLUSPLUS 290        /* Symbol ++            */
#define MINUSMINUS 291      /* Symbol --            */
#define RETURN 292          /* Keyword return       */
```

Figure 2.11: Second half of file "tokens.h", which contains all the tokens your lexer needs to recognize.

- The beginning open brace of a lex rule needs to appear on the same line as the rule itself. Thus the rule

  ```
  [0-9] +
              { return INTEGER_LITERAL; }
  ```

  will not compile correctly. However, the rule

  ```
  [0-9] +     {
                return INTEGER_LITERAL; }
  ```

  will work just fine.

2.2.4 Provided Files

The following files are available on the web site for the text, under the lexer subdirectory:

- **tokens.h** The code in Figures 2.10 and 2.11.
- **sjava.lex** The code in Figure 2.8.
- **tokentest.c** The code in Figure 2.9.
- **errors.h, errors.c** C code to do some error handling. The function error(int, char *) is defined in this file.
- **makefile** A makefile, which creates the lexical analyzer, and a test program named a.out. "make clean" removes all object and executable files, as well as the files created automatically by lex.
- **test1.sjava .. test4.sjava** Some sample test programs to test your lexical analyzer.

Once you have modified pascal.lex to implement a complete lexical analyzer, you can build the lexical analyzer, either with the included makefile, or with the command:

```
lex sjava.lex
gcc lex.yy.c tokentest.c errors.c
```

You can test your lexical analyzer with the command:

```
a.out <filename>
```

Chapter 5

Bottom-Up Parsing & Yacc

Chapter Overview

This segment will cover the use of Yacc – a tool that creates C code that implements an LALR parser given a Context Free Grammar.

5.1 Yacc – Yet Another Compiler Compiler

Just as lex is a tool that takes as input a group of regular expressions and creates C code that implements a lexical analyzer, yacc is a tool that takes as input a context-free grammar and produces as output C code that implements a LALR(1) parser. Yacc and lex were meant to work together, so when yacc needs to see the next token, it calls on lex's next token function, yylex. Likewise, yacc generates the token definitions ("tokens.h" from the last assignment) that lex uses.

5.1.1 Structure of a Yacc File

All yacc files have the filename suffix .grm (for "grammar") and the following structure:

```
%{
 /* C definitions -- #includes, functions, etc  */
 /*                  just like in .lex files  */


%}
/* yacc definitions   */

%token Names of all the tokens -- terminals -- in the CFG
```

```
%{

/* No extra C code required  */

%}

%token PLUS MINUS TIMES DIVIDE IDENTIFIER

%%

exp : exp PLUS term
exp : exp MINUS term
exp : term
term : term TIMES IDENTIFIER
     | term DIVIDE IDENTIFIER
     | IDENTIFIER
```

Figure 5.1: A yacc .grm file for simple expressions.

```
%%

CFG rules, of form:
lhs : rhs
```

Of course, just to make your life difficult, in yacc files terminals are in UPPERCASE while non-terminals are lowercase – exactly the opposite of the convention for CFGs. Why the difference? Probably because C constants are often all uppercase by convention, and tokens are really nothing more than #define'd constants.

For instance, a yacc .grm file for the simple expressions is in Figure 5.1. Note that the first non-terminal that appears after the %% is assumed to be the starting symbol. You can also use the CFG "shorthand" in a .grm file – as we have with the term rules.

If we run yacc on the file of Figure 5.1, then the following files are created:

y.tab.h	The tokens file, like "tokens.h", for lex to use
y.tab.c	C code that implements the parsing function `int yyparse(void)`. yyparse returns 0 on a successful parse, and non-zero on a unsuccessful parse.
y.output	Human readable form of the parse table

Now, all we have to do is write a lex file that can recognize the tokens in "y.tab.h", and we have a parser that returns 0 or 1 for a successful or unsuccessful parse of expressions.

5.1.2 Dealing With Parsing Errors

Of course, we'd like a little more feedback than just a yes/no on whether or not the parsing was successful. When yacc finds an error – that is, a missing entry in the parse table, it calls the function `void yyerror(char *s)`, passing in the string "syntax error". All we have to do is write the body of that function. We place this definition in the C code segment, as in Figure 5.2

The variable current_line_number is the global variable set by the lexer, which stores the line number of the last token returned by the lexer. If a program is not syntactically correct, our parser will print out the line number of the last token returned from the lexer. Of course this is not always the exact location of the error, but it is usually close. If you've done much programming, you have probably had a compiler give you an error message with a less than helpful location.

```
%{
#include <stdio.h>
#include "errors.h"    /* Definition of global variable current_line_number, */
                       /*  whose value is set by the lexer                   */
void yyerror(char *s) {
   fprintf(stderr,"line %d, %s \n",current_line_number, s);
}
%}

%token PLUS MINUS TIMES DIVIDE IDENTIFIER

%%

exp : exp PLUS term
    | exp MINUS term
    | term
term : term TIMES IDENTIFIER
     | term DIVIDE IDENTIFIER
     | IDENTIFIER
```

Figure 5.2: A yacc .grm file that deals with errors in a more elegant fashion

5.1.3 Tokens With Values

So far, we have ignored the values of tokens. Many tokens – such as PLUS, MINUS, and IF, do not need values, while other tokens – such as INTEGER_LITERAL and IDENTIFIER, do need values. Our lexical analyzer actually gave values to *all* tokens – each token was tagged with the line number on which that token appeared. While we won't be using the values of the tokens just yet, we do need to tell yacc which tokens have values, and what those values are. First, we have to let yacc know what the possible token values are. We do this with a %union declaration in the yacc definitions segment of the file. For instance, a

```
%union {
   int integer_value;
   char *string_value;
}
```

declaration in the yacc definitions segment of the file tells yacc that some of the tokens will have `int` values, and some will have `char *` values. We now need to tell yacc *which* of the tokens can have *what* values. A simple change to the token declarations does this for us. Consider the .grm file in Figure 5.3. The token ID has the value char *, the token INTEGER_LITERAL has the value int, and all other tokens have no values.

Figure 5.4 shows a yacc file with that handles tokens with more complicated values, like those in the programming project.

5.1.4 When Yacc Has Conflicts

Not all grammars are LR(1).Occasionally, when building the LR parse table, yacc will find duplicate entries. There are two kinds of duplicate entries – shift-reduce conflicts and reduce-reduce conflicts. We will look at each in turn

Shift-Reduce Conflicts

Shift-reduce conflicts arise when yacc can't determine if a token should be shifted, or if a reduce should be done instead. The canonical example of this problem is the dangling else. Consider the following two rules

```
%{

/* No extra C code required  */

%}
%union{
  int integer_value;
  char *string_value;
}

%token <string_value> IDENTIFIER
%token <integer_value> INTEGER_LITERAL
%token PLUS MINUS TIMES DIVIDE

%%

exp : exp PLUS term
exp : exp MINUS term
exp : term
term : term TIMES IDENTIFIER
term : term DIVIDE IDENTIFIER
term : IDENTIFIER
```

Figure 5.3: A yacc .grm file that handles tokens with values

for an if statement:
(1) $S \rightarrow$ if (E) S else S
(2) $S \rightarrow$ if (E) S
Now consider the following sequence of tokens:

```
if (id) if (id) print(id) else print(id)
```

What happens when we hit that else? Should we assume that the interior if statement has no else, and reduce by rule (2)? Or should we assume that the interior if statement does have an else, and shift, with the intent of reducing the interior if statement using rule (1)? If it turns out that both ifs have elses, we'll be in trouble if we reduce now. What if only one of the ifs has an else? We'll be OK either way. Thus the correct answer is to shift in this case. If we have an if statement like `if (<test>) if (<test>) <statement> else <statement>` , shifting instead of reducing on the else will cause the else to be bound to the innermost if. That's another good reason for resolving the dangling if by having the else bind to the inner statement – writing a parser is easier!

Thus, whenever yacc has the option to either shift or reduce, it will always shift. A warning will be produced (since other than the known dangling-else problem, shift-reduce conflicts are almost always signal errors in your grammar), but the parse table will be created.

Reduce-Reduce Conflicts

Consider the following simple grammar:
(1) $S \rightarrow A$
(2) $S \rightarrow B$
(3) $A \rightarrow$ ab
(4) $B \rightarrow$ ab
What should an LR parser of this grammar do with the file:

```
ab
```

```
%{

/* No extra C code required  */

%}
%union{
  struct {
      int value;
      int line_number;
  } integer_value;
  struct {
      char *value
      int line_number;
  } string_value;
  int line_number;
}

%token <string_value> IDENTIFIER
%token <integer_val> INTEGER_LITERAL
%token <line_number> PLUS MINUS TIMES DIVIDE

%%

exp : exp PLUS term
exp : exp MINUS term
exp : term
term : term TIMES IDENTIFIER
term : term DIVIDE IDENTIFIER
term : IDENTIFIER
```

Figure 5.4: A yacc .grm file that handles tokens with more complicated values.

```
%{
  /* C Definitions */
%}

%token PLUS MINUS TIMES DIVIDE LPAREN RPAREN IDENTIFIER INTEGER_LITERAL

%%

exp : exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | exp DIVIDE exp
    | MINUS exp
    | LPAREN exp RPAREN
    | IDENTIFIER
    | INTEGER_LITERAL
```

Figure 5.5: A yacc file for an ambiguous grammar. All shift reduce conflicts will be resolved by shifting, which will lead to incorrect operator precedence

First, the `a` and `b` are shifted. Then what? Should the ab on the stack be reduced to an A by rule (3), or should the ab be reduced to a B by rule (4)? Should an r(3) or an r(4) entry appear in the table? Reduce-reduce conflicts are typically more serious than shift-reduce conflicts. A reduce-reduce conflict is almost always a sign of an error in the grammar. Yacc resolves reduce-reduce conflict by using the first rule that appears in the .grm file. So, in the above example, the table would contain an r(3) entry. However, when using yacc, never count on this behavior for reduce-reduce conflicts! Instead, rewrite the grammar to avoid the conflict. Just because yacc produces code on an reduce-reduce conflict does not mean that the code does what we want!

5.1.5 Operator Precedence

In Chapter 3, we covered one method for dealing with operator precedence – rewriting the grammar itself to encode precedence information. In this section, we will examine a different method for dealing with expressions – leaving the grammar ambiguous, but giving yacc precedence directives to determine when to shift, and when to reduce. Using yacc precedence directives to disambiguate grammars can be dangerous – especially if the subtleties of how the precedence operators work is not clearly understood. Yacc precedence directives can also easily be over used and abused – they should only be considered for simple binary relations.

While using yacc precedence operators is problematic, it is important to understand how they operate – since they commonly occur in yacc code. Also, exploring precedence operators is an excellent way to solidify understanding of LALR parser generators.

Yacc allows us to specify ambiguous grammars for expressions, and then give precedence rules to disambiguate the grammar. Let's take a look at an example. Consider the file in Figure 5.5.

Assuming that yacc resolves all shift-reduce conflicts by shifting, what happens when we parse the string IDENTIFIER * IDENTIFIER + IDENTIFIER? We shift the IDENTIFIER, reduce the IDENTIFIER to an exp, shift the *, shift the IDENTIFIER, reduce the IDENTIFIER to an exp, and our stack looks like the following:

Stack	Input
exp * exp	+ IDENTIFIER

Now what do we do? We have two choices:

1. Reduce the *exp * exp* to *exp*. We can then shift the + IDENTIFIER, and reduce again. This leads to a (IDENTIFIER * IDENTIFIER) + IDENTIFIER parse of the string, which is what we want.

2. Shift the +. We can then shift and reduce the IDENTIFIER, and we will have *exp * exp + exp* on the stack. Reduce the *exp + exp* to *exp*, leaving *exp * exp* on the top of the stack, then reduce again. This leads to a IDENTIFIER * (IDENTIFIER + IDENTIFIER) parse of the string.

Now consider what happens when we parse the string IDENTIFIER + IDENTIFIER * IDENTIFIER. We shift the IDENTIFIER, reduce the IDENTIFIER to exp, shift the +, shift the IDENTIFIER, reduce the IDENTIFIER to an exp, and we are in the following position:

Stack	Input
exp + exp	* IDENTIFIER

Again, we have two choices

1. Reduce the *exp + exp* to *exp*. We can then shift the * IDENTIFIER, and reduce again. This leads to a (IDENTIFIER + IDENTIFIER) * IDENTIFIER parse of the string.

2. Shift the *. We can then shift and reduce the IDENTIFIER, and we will have *exp + exp * exp* on the stack. Reduce the *exp * exp* to *exp*, leaving *exp + exp* on the top of the stack, then reduce again. This leads to a IDENTIFIER + (IDENTIFIER * IDENTIFIER) parse of the string, which is what we want.

What will yacc do? Yacc always shifts on a shift-reduce conflict, which sometimes gives us what we want, and sometimes does not. What do we want to do in general? If we are trying to decide between shifting and reducing, and the next token is an operator with lower precedence than what's on the stack, we want to reduce. If the next token is an operator with higher precedence than what's on the stack, we want to shift. What if the operators have the same precedence, like 3-4-5? Shifting will cause a right-to-left association, i.e., 3 - (4 - 5), while reducing will cause a left-to-right association, i.e., (3 - 4) - 5. All we need to do is tell yacc what the precedence of the operators is. We can do this with yacc definitions, as in Figure 5.6

Figure 5.6 is almost correct – it works for everything except unary minus. So, IDENTIFIER * IDENTIFIER - IDENTIFIER, IDENTIFIER + IDENTIFIER * IDENTIFIER, and IDENTIFIER - IDENTIFIER - IDENTIFIER are all handled correctly. However, consider what happens with - IDENTIFIER * IDENTIFIER. How will the precedence rules handle this case? Shift a -, shift and IDENTIFIER, reduce to an exp, and then we have:

Stack	Input
- exp	* ID

Do we reduce, leading to (-IDENTIFIER) * IDENTIFIER, or do we shift, leading to -(IDENTIFIER * IDENTIFIER). We'd like to reduce, since unary minus has higher precedence than multiplication. However, yacc looks at the precedence of -, and the precedence of *, and since * has higher precedence than -, yacc will shift. What can we do? Instead of letting yacc decide the precedence of a rule based on the terminals that it contains, we can set the precedence of a rule explicitly. The directive %prec <token> after a rule gives that rule the same precedence as <token>. So, all we need is a token with higher precedence than TIMES – but there is no such token. What to do? Make one up! We create a "phantom token" UMINUS that will never be returned by lex, but will help us with precedence, as in Figure 5.7

There are 3 different precedence directives – %left for left associativity, %right for right associativity, and %nonassoc for no associativity. So, if you had the directives:

```
%left PLUS MINUS
%left TIMES DIVIDE
```

```
%{
  /* C Definitions */
%}

%token PLUS MINUS TIMES DIVIDE LPAREN RPAREN IDENTIFIER INTEGER_LITERAL

%left PLUS MINUS
%left TIMES DIVIDE

%%

exp : exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | exp DIVIDE exp
    | MINUS exp
    | LPAREN exp RPAREN
    | IDENTIFIER
    | INTEGER_LITERAL
```

Figure 5.6: A yacc file for an ambiguous grammar. The precedence operators, %left, tell yacc how to resolve this ambiguity using correct operator precedence – except for unary minus, which is still incorrect

```
%{
  /* C Definitions */
%}

%token PLUS MINUS TIMES DIVIDE LPAREN RPAREN IDENTIFIER INTEGER_LITERAL UMINUS

%left PLUS MINUS
%left TIMES DIVIDE
%left UMINUS

%%

exp : exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | exp DIVIDE exp
    | MINUS exp      %prec UMINUS
    | LPAREN exp RPAREN
    | IDENTIFIER
    | INTEGER_LITERAL
```

Figure 5.7: A yacc file for an ambiguous grammar. The precedence operators, %left, tell yacc how to resolve this ambiguity using correct operator precedence – even for unary minus.

Then IDENTIFIER + IDENTIFIER + IDENTIFIER would associate as (IDENTIFIER + IDENTIFIER) + IDENTIFIER, IDENTIFIER * IDENTIFIER + IDENTIFIER would associate as (IDENTIFIER * IDENTIFIER) + IDENTIFIER, and IDENTIFIER + IDENTIFIER * IDENTIFIER would associate as IDENTIFIER + (IDENTIFIER * IDENTIFIER). If instead, you had the precedence directives:

```
%left TIMES DIVIDE
%right PLUS MINUS
```

Then IDENTIFIER + IDENTIFIER + IDENTIFIER would associate as IDENTIFIER + (IDENTIFIER + IDENTIFIER), IDENTIFIER * IDENTIFIER + IDENTIFIER would associate as IDENTIFIER * (IDENTIFIER + IDENTIFIER), and IDENTIFIER + IDENTIFIER * IDENTIFIER would associate as (IDENTIFIER + IDENTIFIER) * IDENTIFIER. Finally, if you had the precedence directives:

```
%left PLUS MINUS
%nonassoc TIMES DIVIDE
```

Then IDENTIFIER + IDENTIFIER + IDENTIFIER would associate as (IDENTIFIER + IDENTIFIER) + IDENTIFIER, IDENTIFIER * IDENTIFIER + IDENTIFIER would associate as (IDENTIFIER * IDENTIFIER) + IDENTIFIER, IDENTIFIER + IDENTIFIER * IDENTIFIER would associate as IDENTIFIER + (IDENTIFIER * IDENTIFIER), and IDENTIFIER * IDENTIFIER * IDENTIFIER would be a syntax error (since TIMES is nonassoc)

5.2 Writing a Parser For simpleJava Using Yacc

5.2.1 Project Definition

For your next project, you will write a yacc .grm file for simpleJava. Appendix A contains a description of the syntax of simpleJava.

Note that for this phase of the project, your parser will not check for type errors, only syntax errors. Thus programs like:

```
PROGRAM foo;

BEGIN

  x := true + 5

END.
```

should parse just fine, even though the variable `x` is defined before it is used, and a boolean value is added to an integer value. Why aren't we doing any type checking at this stage? It turns out that there is some type checking that cannot be done in the grammar construction phase. Context-Free Grammars are not powerful enough, for instance, to define a language where variables are always declared before they are used. Since we will have to do some other type checking later anyway, it is easier to put off all type checking until that phase of compilation.

5.2.2 Project Difficulty

This project is slightly more difficult that the lexical analyzer. In addition to learning the syntax of yacc rules, you will need to create a CFG for simpleJava. Much of this grammar will be straightforward, though expressions and complex variables (class variable access and record accesses) are a little tricky. Expect to spend some time tracking down shift-reduce and reduce-reduce errors. Estimated completion time is 3-4 hours.

5.2.3 Project "Gotcha"s

- Be sure to get the precedence for expressions correct. You can either write your grammar so that it is unambiguous, or use precedence operators for expressions. Do not overuse precedence operators – they should *only* be used for expressions, and not for other statements. If you find yourself using the precedence operators for variable accesses or statements, stop and try rewriting the grammar instead. When in doubt, rewrite the grammar to be unambiguous.

- Structured variable accesses (class instance variables and arrays) are also a little tricky. Recall that structured variable accesses are much like operator expressions and can be handled in a similar fashion (see Chapter 3 for examples). *Do not* use yacc precedence operators for structured variable accesses!

- Avoid having a specific case in the grammar that is also handled by a more general case. For instance, consider the following grammar fragment, which displays this problem. In this fragment, the non-terminal E stands for an expression, while the non-terminal V stands for a variable.

  $E \rightarrow \text{id}$
  $E \rightarrow V$
  $E \rightarrow \ldots$
  $V \rightarrow \text{id}$
  $V \rightarrow \ldots$

  An expression E can turn into a simple variable identifier in two ways: either using the specific case $E \rightarrow \text{id}$, or through the general rule for a variable in an expression, $E \rightarrow V$.

5.2.4 Provided Files

You can find the following files on the same CD as this document:

- **sjava.grm** A skeleton yacc .grm for simpleJava, to get you started.

- **parsetest.c** This file contains a main program that tests you parser. It is reproduced in Figure 5.8.

- **test1.sjava – test9.sjava** Various simpleJava programs to test your parser. Some are correct, and some have parsing errors.

```
#include <stdio.h>

extern int yyparse(void);
extern FILE *yyin;

void parse(char *filename) {
  yyin = fopen(filename,"r");
  if (yyin == NULL) {
    fprintf(stderr,"Could not open file:%s\n",filename);
  } else {

    if (yyparse() == 0) /* parsing worked */
      fprintf(stderr,"Parsing successful!\n");
    else
      fprintf(stderr,"Parsing failed\n");
  }
}


int main(int argc, char **argv) {
 if (argc!=2) {
   fprintf(stderr,"usage: %s filename\n",argv[0]);
   exit(1);
 }
 parse(argv[1]);
 return 0;
}
```

Figure 5.8: C file parsetest.c, for testing your parser.

5.3 Exercises

1. Give a set of yacc rules for parsing variable accesses in C. Your rules should be able to handle simple variables, record accesses, and pointer dereferences, in any combination. Assume that lex returns the following tokens: IDENTIFIER, DOT, LPAREN, RPAREN, STAR. Thus, some legal token sequences would be:

C code	Tokens
x	IDENTIFIER
x.y	IDENTIFIER DOT IDENTIFIER
(*x).y	LPAREN STAR IDENTIFIER RPAREN DOT IDENTIFIER
x.y.z	STAR IDENTIFIER DOT IDENTIFIER DOT IDENTIFIER

   Note that (*x).(*y) and x.(y.z) are *not* legal

2. Consider the following expression grammar:

   Terminals = {num, +, *, $}
   Non-Terminals = $\{E', E\}$
   Rules = (0) $E' \rightarrow E\$$
   (1) $E \rightarrow$ num
   (2) $E \rightarrow E + E$
   (3) $E \rightarrow E * E$
   Start Symbol = E'

   (a) Create the LR(0) states and transitions for this grammar

   (b) Since this grammar is ambiguous, it is not LR(k), for any k. We can, however, create a parse table for the grammar, using precedence rules to determine when to shift, and when to reduce. Assuming that * binds more strongly than +, and both are left-associative, create a parse table using the LR(0) states and transitions. Test your parse table with the following strings:
   num * num + num num + num * num num + num + num

   (c) Now create an LR parse table for this grammar, this time assuming that + binds more strongly than *, and both are right-associative. Test your parse table with the following strings:
   num * num + num num + num * num num * num * num

Chapter 6

Abstract Syntax Trees in C

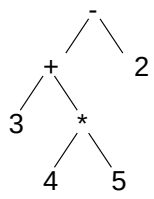
Chapter Overview

6.1 Implementing Trees in C

We need a way of representing these abstract syntax trees in C. Consider the following expression:

$$3 + 4 * 5 - 2$$

Assuming the standard order of operations, this expression has the following abstract syntax tree:



How should we represent this tree in C?

6.1.1 Representing trees – structs and unions

We need some way of representing expression trees in C. An obvious solution is to use pointers and structs, as follows:

```
typedef struct exp *expPtr;
typedef enum {PlusOp, MinusOp, TimesOp, DivideOp} binOp;

struct exp { enum {Number,Operator} kind;
             binOp operator;
             int value;
             expPtr left;
             expPtr right;
           };
```

There are a few problems with this representation. The first is merely one of convenience. Whenever you deal with trees, you tend to use pointers much more frequently than the nodes themselves. Why use the longer name (expPtr) for what we will use the most, and the shorter name (exp) for what we will use the least? C programmers hate to type – let's use exp to represent the pointer, and pick a different name for the struct. By convention, we will end all struct names will end with an _. Pointers to structs will have the same name, without the _. Looking at the new version, we have:

```
typedef struct exp_ *exp;
typedef enum {PlusOp, MinusOp, TimesOp, DivideOp} binOp;

struct exp_ { enum {Number,Operator} kind;
              binOp operator;
              int value;
              exp left;
              exp right;
};
```

Another problem with this representation involves wasted space – we don't need to store an operator, value, left, and right field in every node – we need either a value or an operator, left, and right. Unions to the rescue:

```
typedef struct exp_ *exp;
typedef enum {PlusOp, MinusOp, TimesOp, DivideOp} binOp;

struct exp_ { enum {Constant,Operator} kind;
              union {
                 int value;
                 struct { binOp operator;
                          exp left;
                          exp right;
                        } op;
              } u;
};
```

If we had the following variable in our program:

```
 exp myExp;
```

Assuming we had already allocated space for myExp and set the values of the fields, we can get at the kind field as follows:

```
myExp->kind
```

the value field (if it exists) as follows:

```
myExp->u.value
```

And the operator field (if it exists) as follows:

```
myExp->u.op.operator
```

6.1.2 Using Constructors in C

Even though we will be programming in C (and not an object-oriented language like Java or C++), we will still be using some handy object oriented concepts, like constructors. A constructor is just a function that creates and initializes a data structure. There are two different kinds of expressions (Number expressions and Operator expressions), so we will write two different constructors:

```
exp ConstantExpression(int value);
exp OperatorExpression(exp left, exp right, binOp operator);
```

Constructors are pretty straightforward – all they do is allocate memory and set up initial values, as follows:

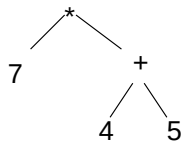
```
exp ConstantExpression(int value) {
   exp retval;

   retval = malloc(sizeof(*retval));
   retval->kind = Number;
   retval->u.value = value;
   return retval;
}
exp OperatorExpression(exp left, exp right, binOp operator) {
   exp retval;

   retval = malloc(sizeof(*retval));
   retval->kind = Operator;
   retval->u.op.left = left;
   retval->u.op.right = right;
   retval->u.op.operator = operator;
   return retval;
}
```

By convention, the constructor for a type will have the same name, but with an initial capital letter. When possible, the names of parameters in the constructor will match the field names.

So, if we had the following variable declaration in our code:

```
exp myExp;
```

We could set myExp to store the tree:



with the following command:

```
myExp = OperatorExpression(ConstantExpression(7),
                           OperatorExpression(ConstantExpression(4),
                                              ConstantExpression(5),
                                              PlusOp),
                           TimesOp);
```

```
%{
#include <stdio.h>
%}

%union{
  int integer_value;
  char *string_value;
}

%token <string_value> IDENTIFIER
%token <integer_value> INTEGER_LITERAL
%token PLUS MINUS TIMES DIVIDE

%left PLUS MINUS
%left TIMES DIVIDE
%%

exp : exp PLUS exp          { printf("+"); }
    | exp MINUS exp         { printf("-"); }
    | exp TIMES exp         { printf("*"); }
    | exp DIVIDE exp        { printf("/"); }
    | IDENTIFIER            { printf("%s ",$1); }
    | INTEGER_LITERAL       { printf("%d ",$1); }
```

Figure 6.1: A yacc file that converts a simple infix expression into the postfix equivalent.

6.1.3 Traversing Trees in C

Once we have built an Abstract Syntax Tree for a program, we will need to traverse it – either to print out the tree for debugging purposes, or to do semantic analysis or code generation. When programming in C, recursive data structures like trees are traversed by creating a mutually recursive suite of functions, each of which handles a particular type of node.

6.2 Yacc Actions

Recall that in the lex file, we could attach arbitrary C code to each regular expression. That code would be executed when the regular expression was matched. In the same way, yacc allows us to attach C code to each rule in the grammar. The C code for a yacc rule is executed when the rule is used in a reduction.

6.2.1 Simple Yacc Actions

Yacc actions appear to the right of each rule, much like lex actions. We can associate pretty much any C code that we like with each rule. Consider the yacc file in Figure 6.1. Tokens are allowed to have values, which we can refer to in our yacc actions. The $1 in `printf("%s",$1)` (the 5th rule for exp) refers to the value of the IDENTIFIER (notice that IDENTIFIER tokens are specified to have string values). The $1 in `printf("%d",$1)` (the 6th rule for exp) refers to the value of the INTEGER_LITERAL. We use $1 in each of these cases because the tokens are the first items in the right-hand side of the rule. We will discuss the $ notation more completely in section 6.2.2

Let's look at some example input files for this yacc file. Starting with some very simple examples, consider the input:

4

What happens when we try to parse this file? First, we shift INTEGER_LITERAL(4) onto the stack. We then reduce by the rule exp: INTEGER_LITERAL, and execute the printf statement, printing out a 4. So the output would be:

```
4
```

OK, so that's not terribly interesting. Let's look at a slightly more complicated example:

```
4 + 5
```

First, we shift INTEGER_LITERAL(4) onto the stack. We then reduce by the rule exp: INTEGER_LITERAL, and execute the printf statement, printing out a 4. We then shift a PLUS, then shift an INTEGER_LITERAL(5). Next, we reduce by the rule exp: INTEGER_LITERAL and print out a 5. Finally, we reduce by the rule exp: exp + exp and print out a +. So the output would be:

```
4 5 +
```

What about:

```
4 + 5 * 6
```

First, we shift INTEGER_LITERAL(4) onto the stack. We then reduce by the rule exp: INTEGER_LITERAL, and execute the printf statement, printing out a 4. We then shift a PLUS, then shift an INTEGER_LITERAL(5). Next, we reduce by the rule exp: INTEGER_LITERAL and print out a 5. Next, we shift a TIMES (remember those precedence directives!), then shift an INTEGER_LITERAL(6). We reduce by exp: INTEGER_LITERAL (printing out a 6), then we reduce by the rule exp: exp * exp (printing out a *), and finally we reduce by the rule exp: exp + exp and print out a +. So the output would be:

```
4 5 6 * +
```

Finally, what about:

```
4 * 5 +  6
```

First, we shift INTEGER_LITERAL(4) onto the stack. We then reduce by the rule exp: INTEGER_LITERAL, and execute the printf statement, printing out a 4. We then shift a TIMES, then shift an INTEGER_LITERAL(5). Next, we reduce by the rule exp: INTEGER_LITERAL and print out a 5. Next, we reduce by the rule exp: exp TIMES exp (again, remember those precedence directives!) and print out a *. Next, we shift a PLUS and an INTEGER_LITERAL(6). We reduce by exp: INTEGER_LITERAL (printing out a 6), then we reduce by the rule exp: exp + exp (printing out a +), giving an output of:

```
4 5 * 6 +
```

So what does this parser do? It converts infix expressions into postfix expressions.

6.2.2 Using the Yacc Stack

In the C code that is executed during a reduction, we can take a look at the values of the tokens that are on the stack (for tokens that have values, of course). We can also give values to the non-terminals that we place on the stack. $1 refers to the value of the first symbol on the right hand side of the rule, $2 refers to the value of the second symbol on the right-hand side of the rule, $3 refers to the value of the third variable on the right-hand side of a rule, and so on. $$ refers to the value of the non-terminal that we place on the stack when we do the reduction. Just as we had to specify the types for the tokens that can take values, we will have to specify types for the non-terminals that we would like to take values. Let's consider an example – a simple yacc calculator

```
%{
  /* C Definitions */
%}

%union{
  int integer_value;
}

%token <integer_value> INTEGER_LITERAL
%token PLUS MINUS TIMES DIVIDE
%type <integer_value> exp

%left PLUS MINUS
%left TIMES DIVIDE
%%

prog : exp                  { printf("%d \n",$1); }

exp : exp PLUS exp          { $$ = $1 + $3; }
    | exp MINUS exp         { $$ = $1 - $3; }
    | exp TIMES exp         { $$ = $1 * $3; }
    | exp DIVIDE exp        { $$ = $1 / $3; }
    | INTEGER_LITERAL       { $$ = $1$; }
```

Figure 6.2: A simple yacc calculator.

6.2.3 A Simple Yacc Calculator

Consider the yacc code in Figure 6.2. For each non-terminal that we want to have a value, we need to tell yacc the type that non-terminal will have. The %type command is identical to the %token command, except that it is used to specify the types of non-terminals instead of tokens. Thus in this example, the token INTEGER_LITERAL takes in a integer value, the non-terminal exp takes on a integer value, and the non-terminal prog has no value.

Let's go over an example of this parser in action. Consider the input 3 + 4. Initially, the stack is empty and the input left to consider is 3 + 4:

Stack	Input
<empty>	INTEGER_LITERAL(3) PLUS INTEGER_LITERAL(4)

We shift an INTEGER_LITERAL:

Stack	Input
INTEGER_LITERAL(3)	PLUS INTEGER_LITERAL(4)

Now we reduce by the rule exp: INTEGER_LITERAL. We execute the C code `$$ = $1`. The value of the INTEGER_LITERAL ($1) is 3, so when we push the exp on the top of the stack, we will also push the value 3.

Stack	Input
exp(3)	PLUS INTEGER_LITERAL(4)

Next, we shift a PLUS, then we shift an INTEGER_LITERAL, to get:

Stack	Input
exp(3) PLUS INTEGER_LITERAL(4)	<empty>

```
/* File treeDefinitions.h   */

typedef struct expression *expressionTree;

typedef enum {PlusOp, MinusOp, TimesOp, DivideOp} optype;

struct expression {
  enum {operatorExp, constantExp, variableExp} kind;
  union {
    struct {
      optype op;
      expressionTree left;
      expressionTree right;
    } oper;
    int constantval;
    char *variable;
  } u;
};

expressionTree operatorExpression(optype op, expressionTree left,
                                  expressionTree right);
expressionTree IdentifierExpression(char *variable);
expressionTree ConstantExpression(int constantval);
```

Figure 6.3: Header file treeDefinitions.h, used by the yacc grammar in Figure 6.5.

We again reduce by the rule exp : INTEGER_LITERAL to get:

Stack	Input
exp(3) PLUS exp(4)	<empty>

Now we reduce by the rule exp : exp + exp. The C code for this rule is $$ = $1 + $3. So, we find add the value of the first exp ($1), which is 3, to the value of the second exp ($3), which is 7, and push the result on the top of the stack when we do the reduce, to get:

Stack	Input
exp(7)	<empty>

Finally, we reduce by the rule prog : exp, and execute the C code `printf("%d",$1)`, and print out a 7.

6.2.4 Creating an Abstract Syntax Tree for a Simple Language

The calculator in the above example uses yacc to create an interpreter for a very simple expression language. While it is possible to write an interpreter for a standard programming language within yacc, and it is even possible to write a compiler completely in yacc, we are going to use yacc just for creating abstract syntax trees. Let's start with a simple example – expression trees. The definitions for the expression trees that we will be creating are in Figure 6.3. The .c file to implement constructors for these trees is in Figure 6.4.

The yacc grammar to build expression trees is in Figure 6.5. Let's look at each section of this file:

- **C Definitions** Between %{ and %} in the yacc file are the C definitions. The #include is necessary for the yacc actions to have access to the tree types and constructors, defined in treeDefinitions.h. The global variable root is where yacc will store the finished abstract syntax tree.
- **%union** The %union{ } command defines every value that could occur on the stack – not only token values, but non-terminal values as well. This %union tells yacc that the types of values that we could

```
/* File treeDefinitions.c   */
#include "treeDefinitions.h"
#include <stdio.h>

expressionTree operatorExpression(optype op, expressionTree left,
                                  expressionTree right) {
  expressionTree retval = (expressionTree) malloc(sizeof(struct expression));
  retval->kind = operatorExp;
  retval->u.oper.op = op;
  retval->u.oper.left = left;
  retval->u.oper.right = right;
  return retval;
}

expressionTree  IdentifierExpression(char *variable) {
  expressionTree retval = (expressionTree) malloc(sizeof(struct expression));
  retval->kind = variableExp;
  retval->u.variable = variable;
  return retval;
}

expressionTree ConstantExpression(int constantval) {
  expressionTree retval = (expressionTree) malloc(sizeof(struct expression));
   retval->kind = constantExp;
   retval->u.constantval = constantval;
   return retval;
}
```

Figure 6.4: File treeDefinitions.c, containing the constructors for building simple expression trees.

```
%{
#include "treeDefinitions.h"
expressionTree root;
%}

%union{
  int integer_value;
  char *string_value;
  expressionTree expression_tree;
}

%token <integer_value> INTEGER_LITERAL
%token <string_value> IDENTIFIER
%token PLUS MINUS TIMES DIVIDE

%type <expresstion_tree> exp

%left PLUS MINUS
%left TIMES DIVIDE
%%

prog : exp                  { root = $$; }

exp : exp PLUS exp          { $$ = OperatorExpression(PlusOp,$1,$3); }
    | exp MINUS exp         { $$ = OperatorExpression(MinusOp,$1,$3); }
    | exp TIMES exp         { $$ = OperatorExpression(TimesOp,$1,$3); }
    | exp DIVIDE exp        { $$ = OperatorExpression(DivideOp,$1,$3); }
    | IDENTIFIER            { $$ = IdentifierExpression($1); }
    | INTEGER_LITERAL       { $$ = ConstantExpression($1); }
```

Figure 6.5: Creating an abstract syntax tree for simple expressions.

push on the stack are integers, strings (char *), and expressionTrees. Each element on the yacc stack contains two items – a state, and a union (which is defined by %union). by %union{ }.

- **%token** For tokens with values, `%token <field>` tells yacc the type of the token. For instance, in the rule `exp : INTEGER_LITERAL { $$ = ConstantExpression($1)}`, yacc looks at the union variable on the stack to get the value of $1. The command `%token <integer_value> INTEGER_LITERAL` tells yacc to use the integer_value field of the union on the stack when looking for the value of an INTEGER_LITERAL.
- **%type** Just like for tokens, these commands tell yacc what values are legal for non-terminal commands
- **%left** Precedence operators. See section 5.1.5 for more details.
- Grammar rules The rest of the yacc file, after the %%, are the grammar rules, of the form

  ```
  <non-terminal> : <rhs>     { /* C code  */ }
  ```

  where <non-terminal> is a non-terminal and <rhs> is a sequence of tokens and non-terminals.

Let's look at an example. For clarity, we will represent the token INTEGER_LITERAL(3) as just 3. Pointers will be represented graphically with arrows. The stack will grow down in our diagrams. Consider the input string 3+4*5. First, the stack is empty, and the input is 3+4*x.

Stack	Input
<empty>	3 + 4 * 5

We shift the 3, then reduce by the rule exp : INTEGER_LITERAL to get:

Stack	Input
exp → 3	+ 4 * 5

We then shift the + and the 4, and reduce by the rule exp : INTEGER_LITERAL to get:

Stack	Input
exp → 3 + exp → 4	* 5

Next, we shift the *, then shift the x, and reduce by the rule exp : INTEGER_LITERAL to get:

Stack	Input
exp → 3 + exp → 4 * exp → 5	<empty>

Next, we reduce by the rule exp : exp * exp to get

Stack	Input
exp → 3 + exp → * (4, 5)	<empty>

We reduce by the rule exp : exp + exp to get:

Stack	Input
exp → + (3, * (4, 5))	<empty>

Finally, we reduce by the rule prog : exp, and the tree is assigned to the global variable root.

6.2.5 Tokens With Complicated Values

All of our examples so far have dealt with tokens that have simple values – either integers, strings, or no values at all. However, the values returned by our lexical analyzer are more complicated, and include location information. We would like to use this extra information to include location information in our abstract syntax tree. Figure 6.6 contains a yacc file that uses tokens that have complex values. We have added an extra final parameter to each of the constructors, to store the line number from the source file.

6.3 Creating an Abstract Syntax Tree for simpleJava Using C and Yacc

6.3.1 Project Definition

For your next programming project, you will add yacc actions to your parser to create an abstract syntax tree for simpleJava. A .h file that describes simpleJava abstract syntax trees is in Figures 6.7 and 6.8.

The constructors for all of the AST nodes of simpleJava can be found in Figure 6.8. For most tree nodes, the form of the struct can be readily derived from the constructor – each parameter in the constructor maps to an element in the struct with the same name. For instance, the constructor for an instance variable declaration is:

```
ASTinstanceVarDec ASTInstanceVarDec(int line, char *type, char *name, int arraydimension);
```

and the struct that defines an instance variable declaration is:

```
struct ASTinstanceVarDec_ {
  int line;
  char *type;
  char *name;
  int arraydimension;
};
```

A constructor merely allocates enough space for the tree node and sets the fields with the appropriate values. There are three exceptions to this rule – statements, expressions, and function declarations. Each of these AST node types has variations, much like subclasses. For instance, statements can be if statements, for statements, while statements, and so on. Function Declarations can be prototypes or function definitions. These structs contain a kind field to denote the type of tree node, and a union to hold type-specific information. For instance, the struct for a function declaration is:

```
struct ASTfunctionDec_ {
  int line;
  enum {Prototype, FunctionDef} kind;
  union {
    struct {
      char *returntype;
      char *name;
      ASTformalList formals;
    } prototype;
    struct {
      char *returntype;
      char *name;
      ASTformalList formals;
      ASTstatementList body;
    } functionDef;
  } u;
};
```

```
%{
#include "treeDefinitions.h"
expressionTree root;
%}


%union{
  struct {
      int value;
      int line_number;
  } integer_value;
  struct {
      char *value
      int line_number;
  } string_value;
  int line_number;
  expressionTree expression_tree;
}

%token <integer_value> INTEGER_LITERAL
%token <string_value> IDENTIFIER
%token <line_number> PLUS MINUS TIMES DIVIDE

%type <expresstion_tree> exp

%left PLUS MINUS
%left TIMES DIVIDE
%%

prog : exp              { root = $1; }

exp : exp PLUS exp      { $$ = OperatorExpression(PlusOp,$1,$3,$2); }
    | exp MINUS exp     { $$ = OperatorExpression(MinusOp,$1,$3,$2); }
    | exp TIMES exp     { $$ = OperatorExpression(TimesOp,$1,$3,$2); }
    | exp DIVIDE exp    { $$ = OperatorExpression(DivideOp,$1,$3,$2); }
    | IDENTIFIER        { $$ = IdentifierExpression($1.value, $1.line_number); }
    | INTEGER_LITERAL   { $$ = ConstantExpression($1.value, $1.line_number); }
```

Figure 6.6: Creating an abstract syntax tree, using tokens with complex values.

```
typedef struct ASTprogram_ *ASTprogram;
typedef struct ASTclass_ *ASTclass;
typedef struct ASTclassList_ *ASTclassList;
typedef struct ASTinstanceVarDec_ *ASTinstanceVarDec;
typedef struct ASTinstanceVarDecList_ *ASTinstanceVarDecList;
typedef struct ASTfunctionDec_ *ASTfunctionDec;
typedef struct ASTfunctionDecList_ *ASTfunctionDecList;
typedef struct ASTformal_ *ASTformal;
typedef struct ASTformalList_ *ASTformalList;

typedef struct ASTstatement_ *ASTstatement;
typedef struct ASTstatementList_ *ASTstatementList;
typedef struct ASTexpression_ *ASTexpression;
typedef struct ASTvariable_ *ASTvariable;
typedef struct ASTexpressionList_ *ASTexpressionList;

typedef enum {AST_EQ, AST_NEQ, AST_LT, AST_GT, AST_LEQ, AST_GEQ,
              AST_AND, AST_OR, AST_NOT, AST_PLUS, AST_MINUS,
              AST_MULTIPLY, AST_DIVIDE} ASToperator;
```

Figure 6.7: Type definitions for simpleJava abstract syntax trees, from file AST.h

There are two types of function declarations – prototypes and function definitions. Prototypes contain the type and name of the function, and a list of formal parameters. Function definitions contain the type and name of the function, a list of formal parameters, and the body of the function itself.

Now let's take a look at the skeleton .grm file for creating abstract syntax trees for simpleJava in Figure 6.9. You should probably not modify this file, but instead modify your own code from the first parser assignment, using this as a guide.

The C definitions segment between % and % contains #include files for building the tree. The "AST.h" file contains definitions of the abstract syntax tree, while the "errors.h" contains a definition for the variable `Current_Line`, used by our yyerror function. The definition of `yyerror` is also found in the C segment.

The yacc definition segments of the file contain the %union definition (which you will need to expand by adding more AST types), the %token directives (which you can leave as they are) and the %type directives (which you will also need to expand.) The %start directive tells yacc that program is the initial non-terminal

Finally, the there are some grammar rules to get you started. Note that these rules *will* need to be heavily modified, I've only provided some examples of how to call the tree constructors within yacc.

6.3.2 Project Difficulty

This project is potentially slightly easier than the parser. If all goes well, you should be able to add constructors to your simplejava.grm file from the previous assignment. You may, however, need to modify your grammar to make creating the abstract syntax tree easier.

6.3.3 Project "Gotcha's"

- The abstract syntax tree structure is a little complicated. Take some time to understand the structure of the sjava ASTs.
- `<var>++` is equivalent to `<var> = <var> + 1`. `-<exp>` is equivalent to `0 - <exp>`
- It is easy to make mistake in stack elements – inadvertently substituting a $3 for a $4, for instance. Most (but not all) of these errors will be caught by the C type checker.

```
ASTprogram ASTProgram(int line, ASTclassList classes,
                      ASTfunctionDecList functiondecs);
ASTclass ASTClass(int line, char *name, ASTinstanceVarDecList instancevars);
ASTclassList ASTClassList(int line, ASTclass first, ASTclassList rest);
ASTinstanceVarDec ASTInstanceVarDec(int line, char *type, char *name,
                                    int arraydimension);
ASTinstanceVarDecList ASTInstanceVarDecList(int line, ASTinstanceVarDec first,
                                            ASTinstanceVarDecList rest);
ASTfunctionDec ASTFunctionDef(int line, char *returntype, char *name,
                              ASTformalList formals,
                              ASTstatementList body);
ASTfunctionDec ASTPrototype(int line, char *returntype, char *name,
                            ASTformalList formals);

ASTfunctionDecList ASTFunctionDecList(int line, ASTfunctionDec first,
                                      ASTfunctionDecList rest);
ASTformal ASTFormal(int line, char *type, char *name, int arraydimension);
ASTformalList ASTFormalList(int line, ASTformal first, ASTformalList rest);

ASTstatement ASTAssignStm(int line, ASTvariable lhs, ASTexpression rhs);
ASTstatement ASTIfStm(int line, ASTexpression test, ASTstatement thenstm,
                      ASTstatement elsestm);
ASTstatement ASTForStm(int line, ASTstatement initialize, ASTexpression test,
                       ASTstatement increment, ASTstatement body);
ASTstatement ASTWhileStm(int line, ASTexpression test, ASTstatement body);
ASTstatement ASTDoWhileStm(int line, ASTexpression test, ASTstatement body);
ASTstatement ASTVarDecStm(int line, char *type, char *name, int arraydimension,
                          ASTexpression init);
ASTstatement ASTCallStm(int line, char *name, ASTexpressionList actuals);
ASTstatement ASTBlockStm(int line, ASTstatementList statements);
ASTstatement ASTReturnStm(int line, ASTexpression returnval);
ASTstatement ASTEmptyStm(int line);
ASTstatementList ASTStatementList(int line, ASTstatement first,
                                  ASTstatementList rest);
ASTexpression ASTIntLiteralExp(int line, int value);
ASTexpression ASTBoolLiteralExp(int line, int value);
ASTexpression ASTOpExp(int line, ASToperator operator, ASTexpression left,
                       ASTexpression right);
ASTexpression ASTVarExp(int line, ASTvariable var);
ASTexpression ASTCallExp(int line, char *name, ASTexpressionList actuals);
ASTexpression ASTNewExp(int line, char *name);
ASTexpression ASTNewArrayExp(int line, char *name, ASTexpression size,
                             int arraydimension);
ASTexpressionList ASTExpressionList(int line, ASTexpression first,
                                    ASTexpressionList rest);
ASTvariable ASTBaseVar(int line, char *name);
ASTvariable ASTClassVar(int line, ASTvariable base, char *instance);
ASTvariable ASTArrayVar(int line, ASTvariable base, ASTexpression index);
```

Figure 6.8: Constructor prototypes for simpleJava abstract syntax trees, from file AST.h

```
%{
#include <stdio.h>
#include "errors.h"
#include "AST.h"
ASTprogram absyn_root;
void yyerror(char *s) {
    error(current_line_number,"%s\n",s);
}
%}

%union{
  struct {
      int value;
      int line_number;
  } integer_value;
  struct {
      char *value
      int line_number;
  } string_value;
  int line_number;
  ASTProgram program;
  ASTStatement statement_tree;
  ASTExpression expression_tree;
  ASTVariable variable_tree;
}

%token <string_value> IDENTIFIER
%token <integer_value> INTEGER_LITERAL

%token <line_number>  CLASS DO ELSE TRUE FALSE FOR IF
       WHILE PLUS MINUS MULTIPLY DIVIDE LBRACK RBRACK LBRACE
       RBRACE LPAREN RPAREN DOT COMMA SEMICOLON
       EQ NEQ LT GT LEQ GEQ GETS AND OR NOT PLUSPLUS
       MINUSMINUS RETURN

%type <statement_tree> statement
%type <expression_tree> exp
%type <variable_tree> var

%start program
%%

program: classdefs functiondefs { ASTroot = ASTProgram(1, $1, $2); }


statement: var GETS exp        {$$ = ASTAssignmentStatement($2,$1,$3);}

var:  IDENTIFIER               {$$ = ASTBaseVariable($1.line_number,$1.value);}

exp:  INTEGER_LITERAL          {$$ = ASTIntegerLiteral($1.line_number,$1.value);}
```

Figure 6.9: Skeleton .grm file for simpleJava.

6.3.4 Provided Files

You can find the following files in the same CD as this document:

- **AST.h, AST.c** Implementation of simpleJava Abstract Syntax Trees
- **ASTPrint.h, ASTPrint.c** Code to print out Abtract Syntax Trees, to debug your tree generator
- **parsetest2.c** Main program to test your tree generator
- **sjava.grm** Skeleton yacc .grm file. You should modify your own .grm file, using ideas from this file
- **test1.sjava .. testn.sjava** Some sample simpleJava programs
- **makefile** Standard makefile

Chapter 7

Semantic Analysis in C

Chapter Overview

7.1 Types in simpleJava

First, we will consider the implementation of types for simpleJava in C. A header file that defines the internal representation of types is in Figure 7.1.

7.1.1 Built-in Types

For the build-in types integer and boolean, we do not need to keep track of any extra information. The constructors for the base types are thus extremely simple – just IntegerType(), BooleanType(), and VoidType(). To make our life a little easier, multiple calls to each of these constructors will always return the same pointer, so we can use pointer comparisons to check for type equivalence. That is, in the following code:

```
type t1,t2;
t1 = IntegerType();
t2 = IntegerType();
if (t1 == t2) {
   ...
}
```

```
typedef struct type_ *type;
typedef struct typeList_ *typeList;

struct type_ {
  enum {integer_type, boolean_type, void_type,
        class_type, array_type} kind;
  union {
    type array;
    struct {
      environment instancevars;
    } class;
  } u;
};

struct typeList_ {
  type first;
  typeList rest;
};

type IntegerType();
type BooleanType();
type VoidType();
type ClassType(environment instancevars);
type ArrayType(type basetype);
```

Figure 7.1: File types.h

```
type integerType_ = NULL;

type IntegerType() {
   if (integerType_ == NULL) {
       integerType_ = (type) malloc(sizeof(type_));
       integerType_-> kind = integer_type;
   }
   return integerType_;
}
```

Figure 7.2: Implementation of the constructor IntegerType().

The condition `t1 == t2` will be true. Figure 7.2 shows how this constructor works.

7.1.2 Array Types

Array types require some additional information – the base type of the array. The internal representation of the type of the two-dimensional variable `A`:

```
int A[][];
```

can be created with the code:

```
type t3;
t3 = ArrayType(ArrayType(IntegerType()));
```

7.1.3 Class Types

For class tyes, we need to keep track of the names and types of all the instance variables of the class. We can do this by mantaining a variable environment for each class type, which stores all of the variable definitions for that class.

We can build up an internal representation of the class:

```
class foo {
   int x;
   boolean y;
}
```

using the C code:

```
type t4;
environment instanceVars = Environment();

enter(instanceVars, "x", VariableEntry(IntegerType()));
enter(instanceVars, "y", VariableEntry(BooleanType()));
t4 = ClassType(instanceVars);
```

7.2 Implementing Environments in C

There is a slight complication implementing environments in C, since there is some mutual recursion in the type descriptions – types require environments, and environments require types. We will manage this data recursion by splitting the .h file for environments into two halves – one half that defines pointer types and constructor prototypes, and another that describes the structure of environments. These two header files are in Figures 7.3 and 7.4. Any file that uses environments or types needs to include the environment and type header files in the following order:

```
typedef struct environment_ *environment;
typedef struct envEntry_ *envEntry;

environment Environment();

void AddBuiltinTypes(environment env);
void AddBuiltinFunctions(environment env);

void beginScope(environment env);
void endScope(environment env);

void enter(environment env, char * key, envEntry entry);
envEntry find(environment env, char *key);
```

Figure 7.3: The first of two environment header files, environment1.h.

```
#include "environment1.h"
#include "type.h"
#include "environment2.h"
```

7.3 Writing a Suite of Functions for Semantic Analysis

To analyze the abstract syntax tree, we will write a suite of recursive functions that traverse the tree, maintain the various environments, and check for semantic errors in the program. Each function will traverse a subsection of the tree, possibly add to the environments, and check for semantic errors in that subtree.

Functions that analyze expression trees will return a value – the type of the expression represented by the tree. Most other functions will not return a value, but will modify the environments and check for semantic errors.

We will use an error handling module to keep track of any errors that occur in the semantic analysis phase. The .h file that describes the interface to this module is in Figure 7.5. The function `Error` works much like printf, with an additional input parameter which gives the line number of the error. Some legal calls to `Error` would be:

```
char *name;
int intval;

Error(3, "Variable %s not defined", name);
Error(15, "Function %s requires %d input parameters",name, intval);
```

We will now look at some examples of the recursive functions used for semantic analysis.

7.3.1 Analyzing a simpleJava Program

The top-level recursive function needs to set up the initial environments, examine the class definitions (making appropriate additions to the type environment), and then examine the function prototypes and function definitions. A skeleton of the function for analyzing the entire program can be found in Figure 7.6.

7.3.2 Analyzing simpleJava Expressions in C

The functions which analyze simpleJava expressions will first ensure that the expression has no semantic errors, and then return the internal representation of the type of the expression. First we must determine what kind of expression is being evaluated, and then call the appropriate function to analyze the expression. A function to analyze expressions is in Figure 7.7.

```
envEntry VarEntry(type typ);
envEntry FunctionEntry(type returntyp, typeList formals);
envEntry TypeEntry(type typ);

struct envEntry_ {
  enum {Var_Entry, Function_Entry,Type_Entry} kind;
  union {
    struct {
      type typ;
    } varEntry;
    struct {
      type returntyp;
      typeList formals;
    } functionEntry;
    struct {
      type typ;
    } typeEntry;
  } u;
};
```

Figure 7.4: The second of two environment header files, environment2.h.

```
void Error(int linenum, char *message, ...);
int anyerrors();
int numerrors();
```

Figure 7.5: The file errors.h, which describes the interface to the C error handler.

```
void analyzeProgram(ASTprogram program) {
  environment typeEnv;
  environment functionEnv;
  environment varEnv;

  typeEnv = Environment();
  functionEnv = Environment();
  varEnv = Environment();

  AddBuiltinTypes(typeEnv);
  AddBuiltinFunctions(functionEnv);

  analyzeClassList(typeEnv, functionEnv, varEnv, program->classes);
  analyzeFunctionDecList(typeEnv,functionEnv,varEnv, program->functiondecs);
}
```

Figure 7.6: Part of the code for the function to handle type checking of a simpleJava program

```
void analyzeExpression(typeEnvironment typeEnv,
                       functionEnvironment functionEnv,
                       variableEnvironment varEnv, ASTexpression expression) {
  switch(expression->kind) {
  case ASTvariableExp:
    return analyzeVariable(varEnv,exp->u.var);
  case ASTintegerLiteralExp:
    return IntegerType();
  case ASTbooleanLiteral:
    return BooleanType();
  case ASToperatorExp:
    return analyzeOperatorExpression(typeEnv,functionEnv,varEnv, expression);
  case ASTcallExp:
    return analyzeCallExpression(typeEnv,functionEnv,varEnv, expression);
  }
}
```

Figure 7.7: C code to analyze simpleJava expressions.

```
type analyzeBaseVariable(variableEnvironment varEnv, ASTVariable var) {
    envEntry base;
    base = find(varEnv, var->u.baseVar.name);
    if (base == NULL) {
      Error(var->line,"Variable %s not defined",var->u.baseVar.name);
      return IntegerType();
    }
    return base->u.typeEntry.typ;
}
```

Figure 7.8: C code for analyzing base variables.

Analyzing Variables Expressions in C

There are three different kinds of variables that we will need to analyze – simple, or base variables, class variables, and array variables. We will now consider analyzing base variables in more detail.[1] To analyze a base variable, we only need to look up that variable in the variable environment to find its type. If the variable has been defined, we can return the type we found in the variable environment. If the variable has not been defined, then we output an error. Since the rest of the semantic analyzer expects that this analyze function will return a valid type, we need to return some type for undefined variables. For lack of a better choice, we will assume that all undefined variables are integers. The C code to analyze a base variable is in Figure 7.8.

7.3.3 Analyzing simpleJava Statements in C

To analyze a statement, we need to first determine what kind of a statement we are analyzing, and then call the appropriate function, as in Figure 7.9. Note that functions to analyze statements do not (yet) need to return a value, since statements have no type.

We will look at if statements in more detail. To analyze an if statement, we need to analyze the test of the if statement, ensure that this test is of type boolean, and then analyze the body of the if statement, as in Figure 7.10.

[1] Structured variables are left as an exercise to the reader.

```
void analyzeStatement(environment typeEnv, environment functionEnv,
                      environment varEnv, ASTstatement statement) {
  switch(statement->kind) {
  case AssignStm:
    analyzeAssignStm(typeEnv, functionEnv, varEnv, statement);
    break;
  case IfStm:
    analyzeIfStm(typeEnv, functionEnv, varEnv, statement);
    break;
  case ForStm:
    analyzeForStm(typeEnv, functionEnv, varEnv, statement);
    break;
  case WhileStm:
    analyzeWhileStm(typeEnv, functionEnv, varEnv, statement);
    break;
  case DoWhileStm:
    analyzeDoWhileStm(typeEnv, functionEnv, varEnv, statement);
    break;
  case CallStm:
    analyzeCallStm(typeEnv, functionEnv, varEnv, statement);
    break;
  case BlockStm:
    analyzeBlockStm(typeEnv, functionEnv, varEnv, statement);
    break;
  case ReturnStm:
    analyzeReturnStm(typeEnv, functionEnv, varEnv, statement);
    break;
  case EmptyStm:
    break;
  default:
    Error(statement->line,"Bad Statement");
  }
}
```

Figure 7.9: C code to analyze statements.

```
void analyzeIfStm(environment typeEnv, environment varEnv,
                  environment functionEnv, ASTstatement statement) {
   type testType;
   type test = analyzeExpression(typeEnv, functionEnv, varEnv,
                                 statement->u.ifStm.test);
   if (test != BooleanType()) {
      Error(statement->line,"If test must be a boolean");
   }
   analyzeStatement(typeEnv, functionEnv, varEnv, statement->u.ifStm.thenstm);
   analyzeStatement(typeEnv, functionEnv, varEnv, statement->u.ifStm.elsestm);
}
```

Figure 7.10: Partial functions to type-check if statements.

Other types of statements are analyzed in a similar fashion.

7.4 Semantic Analyzer Project in C

7.4.1 Project Definition

For the next project, you will create a semantic analyzer for simpleJava by writing a suite of mutually recursive functions that analyze a simpleJava Abstract Syntax Tree. The type rules for simpleJava are as follows:

Type Rules For Statements

- **if (<exp>) <statement>** The expression <exp> needs to be of type boolean
- **if (<exp>) <statement1> else <statement2>** The expression <exp> needs to be of type boolean.
- **while (<exp>) <statement>** The expression <exp> needs to be of type boolean
- **do <statement> while (<exp>)** The expression <exp> needs to be of type boolean
- **for(<init>; <exp>; <increment>) <statement>** The expression <exp> must be of type boolean. <init> and <increment> must both be assignment statements (note that x++ is a shorthand for x = x + 1, and is considered to be an assignment statement). The parser should enforce the requirement that <init> and <increment> are assignment statements.
- **<var> = <exp>** <var> needs to be a declared variable, with the same type as <exp> item **Function Calls**. Must be declared before they are called The number and type and of parameters must match the declaration. Unlike C functions, only void functions can be statements in simpleJava.

Type Rules For Expressions

- **Integer Literals** Such as 3, 14, 27, etc. have the type integer.
- **Boolean Literals** Such as TRUE and FALSE have the type boolean.
- **Variables** Must be declared before they are used. Their type is determined by the declaration.
- **Function Calls** Functions must be declared before they are called. The number and types of parameters must match the declaration. The type of a function call is also determined by its declaration.
- **Binary Integer Operators** <exp1> + <exp2>,<exp1> - <exp2>,<exp1> * <exp2>,<exp1> / <exp2>. The expressions <exp1> and <exp2> must both be ints. The type of the expression is int.
- **Unary Integer Operators** - <exp>. The type of <exp> muse must be int. The type of the expression is int.
- **Relational Operators** <exp1> > <exp2>, <exp1> >= <exp2>, <exp1> < <exp2>, <exp1> <= <exp2>, <exp1> <> <exp2>. The types of <exp1> and <exp2> must be int. The type of the expression is boolean.
- **Binary Boolean Operators** <exp1> || <exp2>, <exp1> && <exp2>. The types of <exp1> and <exp2> must be boolean. The type of the expression is boolean.
- **Unary Boolean Operator** !<exp>, The type of the expression <exp> must be boolean. The type of the expression is boolean.

7.4.2 Project Difficulty

The semantic analyzer is quite a bit harder than any of the previous projects. This project can take at least 10-20 hours, depending upon your familiarity with C programming and your experience programming with mutual recursion.

7.4.3 Project "Gotcha"s

- Be sure to allow enough time for this project. It will take longer to complete than previous projects.
- This project is very pointer-intensive. Be sure you understand exactly how environments and type representations work before starting to code.
- Variables (especially class variables and array variable accesses) can be can be tricky. Be sure you go over the material on variables form chapter 6 in the main text before you start coding.

7.4.4 Provided Files

You can find the following files in the semantic analysis section of the web site for this text:

- **AST.h, AST.c** These files contain definitions of Abstract Syntax Trees.
- **SemanticAnalyzerTest.c** This file contains a main program that tests your semantic analyzer. It is reproduced in Figure 7.11.
- **ASTPrint.h, ASTPrint.c** These files contain functions that print out the abstract syntax tree, to assist in debugging.
- **test1.sjava – testn.sjava** Various simpleJava programs to test your abstract syntax tree generator. Be sure to create your own test cases as well!

```
#include <stdio.h>
#include "errors.h"
#include "AST.h"
#include "ASTPrintTree.h"
#include "semantic.h"

extern int yyparse(void);
extern FILE *yyin;

ASTprogram parse(char *filename) {
  yyin = fopen(filename,"r");
  if (yyin == NULL) {
    fprintf(stderr,"Cannot open file:%s\n",filename);
  }
  if  (yyin != NULL && yyparse() == 0)  { /* parsing worked */
    return ASTroot;
  } else {
    fprintf(stderr,"Parsing failed\n");
    return NULL;
  }
}

int main(int argc, char **argv) {
  ASTprogram program;
 if (argc!=2) {
   fprintf(stderr,"usage: %s filename\n",argv[0]);
   exit(1);
 }
 program = parse(argv[1]);
 if (program != NULL) {
   printTree(program);
   analyzeProgram(program);
 }
 return 0;
}
```

Figure 7.11: The file SemanticAnalyzerTest.c.

Chapter 8

Generating Abstract Assembly in C

Chapter Overview

8.1 Creating Abstract Assembly Trees in C

To build abstract assembly trees in C, we will first implement an interface that builds assembly trees for simpleJava statements and expressions, using abstract assembly subtrees. We will then add calls to the functions defined in this interface to the semantic analyzer, so that the semantic analyzer will build abstract assembly trees as well as check for semantic errors.

8.1.1 Assembly Language Labels

We will need to generate assembly language labels when building abstract assembly – both for function calls and for implementing if statements and loops. The file label.h in Figure 8.1 contains prototypes for functions which allow us to generate both unique labels (such as ifEnd001, ifEnd002, etc), and specific labels (for calls to library functions, when we know the exact name of the label). The first call to `newLabel("foo")` returns the label foo001, the second call returns the label foo002, and so on.

8.1.2 Interface for Building Abstract Assembly Trees in C

The interface for building abstract assembly trees is in Figure 8.2. We will now see some examples of using these functions to build abstract assembly trees for various statements.

Consider the following simpleJava program:

```
void foo(int a) {
   int b;
   int c;
```

```
typedef char *label;

label newLabel(char *name);
label absLabel(char *name);

char *getLabelName(label l);
```

Figure 8.1: Prototypes for functions that handle assembly language labels.

```
AATstatement functionDefinition(AATstatement body, int framesize,
                                label start, label end);
AATstatement IfStatement(AATexpression test, AATstatement ifbody,
                         AATstatement elsebody);
AATexpression Allocate(AATexpression size);
AATstatement WhileStatement(AATexpression test,
                            AATstatement whilebody);
AATstatement DoWhileStatement(AATexpression test,
                              AATstatement dowhilebody);
AATstatement ForStatement(AATstatement init, AATexpression test,
                          AATstatement increment, AATstatement body);
AATstatement EmptyStatement();
AATstatement CallStatement(expressionRecList actuals, label name);
AATstatement AssignmentStatement(AATexpression lhs,
                                 AATexpression rhs);
AATstatement SequentialStatement(AATstatement first,
                                 AATstatement second);
AATexpression BaseVariable(int offset);
AATexpression ArrayVariable(AATexpression base,
                            AATexpression index,
                            int elementSize);
AATexpression ClassVariable(AATexpression base, int offset);
AATexpression ConstantExpression(int value);
AATexpression OperatorExpression(AATexpression left,
                                 AATExpression right,
                                 int operator);
AATexpression CallExpression(expressionRecList actuals, label name);
AATstatement ReturnStatement(AATexpression value, label functionend);
```

Figure 8.2: Interface for building abstract assembly trees from simpleJava statements.

```
   if (a > 2)
      b = 2;
   else
      c = a + 1;
}
```

Assume that the size of a machine word is 4, and that all integers and pointers reside in a single word.

The abstract assembly tree for the expression `(a > 2)` in the above function `foo` could be created with the C code:

```
AATexpression e1;

e1 = OperatorExpression(BaseVariable(4),
                        ConstantExpression(2),
                        GREATER_THAN);
```

Likewise, the abstract assembly for the statements `b = 2;` and `c = a + 1;` could be created with the C code:

```
AATstatement s1, s2;

s1 = AssignmentStatement(BaseVariable(0),
                         ConstantExpression(2));
s2 = AssignmentStatement(BaseVariable(-4),
                         OperatorExpression(
                         BaseVariable(4),
                         ConstantExpression(1),
                         AAT_PLUS));
```

Finally, given the above definitions, the abstract assembly for the if statement `if (a > 2) b = 2; else c = a + 1;` could be created with the C code:

```
AATstatement s3;

s3 = IfStatement(e1,s1,s2);
```

Some of the functions for building abstract assembly trees could use more explanation. We now explain some of them in depth:

- `AATstatement FunctionDefinition(AATstatement body, int framesize, Label start, Label end);` The assembly tree for a function definition starts with the assembly language label for the function, followed by a series of instructions to save the old values of the Return, FP, and SP pointers, followed by code to set the new values of the FP and SP, followed by the body of the function (passed in), followed by a label for the end of the function, followed by code to restore the old values of the return address, FP, and SP registers, followed by an abstract assembly Return statement, to return the flow of control to the calling function. The `end` label is required to simplify the implementation of a simpleJava return statement.
- `AATstatement ReturnStatement(AATexpression value, Label functionend);` The assembly tree for a return statement copies the value of the return statement into the Result register, and then jumps to the label at the end of the function.
- `public AATexpression Allocate(AATexpression size);` This function is called on a new, to allocate space from the heap. It is implemented by a function call to the built-in allocate function – just as the input/output functions are implemented by a call to built in functions. The built-in allocate function takes a single input parameter – the size (in bytes) to allocate, and returns a pointer to the beginning of the allocated block.

```
typedef struct expressionRec_ expressionRec;

expressionRec ExpressionRec(type typ, AATexpression tree);

struct expressionRec_ {
   type typ;
   AATexpression tree;
};
```

Figure 8.3: Struct used by expression analyzer to return both a type and an abstract syntax tree.

8.1.3 Adding Code to the Semantic Analyzer to Build Abstract Assembly Trees

Once we have implemented the BuildTree interface, we can use its functions to build the abstract assembly tree. Instead of creating a separate module that traverses the tree again, we will modify the code for our semantic analyzer so that it builds an abstract assembly tree, as well as checking for semantic errors. Each of the functions in our semantic analyzer will return two pieces of information instead of one – the type of the subtree, as before, plus an abstract assembly tree. For example, to analyze an abstract syntax tree:
the semantic analyzer currently returns `IntegerType()`. To analyze the abstract syntax tree:
the semantic analyzer calls the accept method for each of the subtrees – `Constant(3)` and `Constant(4)`, ensures that they are both of type IntegerType(), and returns IntegerType(). After we modify the semantic analyzer to produce abstract assembly trees, when we analyze the abstract syntax tree:
the semantic analyzer will return two values: `IntegerType()` and `ConstantExpression(3)`. When we analyze the abstract syntax tree:
we will first call the Accept method on the subtrees, ensure that their types are integers, and then build an operator expression based on the values of the subtrees.

8.1.4 Functions That Return Abstract Assembly Trees

The functions in our semantic analyzer that analyze statements will now return a value – an abstract assembly tree that implements the statement. The functions that analyze expressions will return two values – the type of the expression, and an abstract assembly tree that implements the expression. Figure 8.3 contains the necessary type definitions for return values for expressions. Figure 8.4 contains a partial semantic analyzer function for analyzing operator expressions, while Figure 8.5 contains the semantic analyzer function for analyzing if statements. Note that very little extra code needs to be added to the semantic analyzer to build abstract assembly trees – most of the work is done through buildtree functions.

8.1.5 Variables

In our semantic analyzer, when we analyzed a variable declaration, that variable was added to the variable environment along with the type of the variable. In the semantic analysis phase of compilation, that is all the information we needed about the variable to check for semantic errors. To create the abstract assembly for a variable, we need to know the offset of the variable from the Frame Pointer, in addition to the type of the variable. We can modify the variable environment so that two pieces of information are stored for a variable – the type of the variable and the offset of the variable from the frame pointer, as in Figure 8.6.

When we add variables into the variable environment, we will add their offset as well. We will then need to keep track of the offset for each variable in our program. This can be accomplished by maintaining a global variable in our semantic analyzer, which keeps track of the current offset.[1] When a new function is started, this offset is set to zero. Each time a new variable declaration is encountered, that variable is

[1] Alternately, this value could be passed to all functions as a parameter, much like the type, function, and variable environments are passed to each function.

```
expreressionRec analyzeOpExpression(environment typeEnv, environment functionEnv,
                                    environment varEnv, ASTexpression opexp) {
  switch(exp->u.opExp.operator) {
    case AST_EQ:
      { expressionRec LHS;
        expressionRec RHS;
        LHS = analyzeExpression(typeEnv,functionEnv,varEnv,exp->u.opExp.left);
        RHS = analyzeExpression(typeEnv,functionEnv,varEnv,exp->u.opExp.right);
        if (LHS->typ != RHS->typ)
          Error(exp->line,"Type mismatch on =");
        return ExpressionRec(BooleanType(),
                             OpExp(LHS->tree, RHS->tree, AAT_EQUAL);
      }
  case AST_NEQ:
      ...
```

Figure 8.4: Part of the semantic analyzer function for operator expressions, expanded with code to create assembly trees.

```
AATstatement analyzeIfStatement(environment typeEnv, environment functionEnv,
                                environment varEnv, ASTstatement ifstm) {
  expressionRec test = analyzeExpression(typeEnv, functionEnv, varEnv,
                                         ifstm->u.ifStm.test);
  AATstatement thenstm;
  AATstatement elsestm;

  if (test->typ != BooleanType()) {
    Error(statement->line,"If test must be a boolean");
  }
  thenstm = analyzeStatement(typeEnv, functionEnv, varEnv,
                             statement->u.ifStm.thenstm);
  elsestm = analyzeStatement(typeEnv, functionEnv, varEnv,
                             statement->u.ifStm.elsestm);
  return IfStatement(test->tree, thenstm, elsestm);
}
```

Figure 8.5: Semantic analyzer function for if statements, expanded with code to create assembly trees.

```
envEntry VarEntry(type typ, int offset);
envEntry FunctionEntry(type returntyp, typeList formals, label startlabel);
envEntry TypeEntry(type typ);

struct envEntry_ {
  enum {Var_Entry, Function_Entry,Type_Entry} kind;
  union {
    struct {
      type typ;
      int offset;
    } varEntry;
    struct {
      type returntyp;
      typeList formals;
      label startlabel
    } functionEntry;
    struct {
      type typ;
    } typeEntry;
  } u;
};
```

Figure 8.6: Modified environment entries, which store offset information for variables, and label information for functions.

entered into the variable environment, using the current offset, and the current offset is updated. When we have finished analyzing the function, the offset will contain the size of the local variable segment of the activation record for the function. A similar strategy can be used to keep track of the offsets for instance variables in classes.

8.1.6 Function Prototypes and Function Definitions

Just as we needed to store offset information in the variable environment, we will need to store some extra information in the function environment – the assembly language labels for the beginning and end of the function. We need to store the label for the beginning of the function so that control can be transferred to the beginning of the function on a function call, and we need to store the label for the end of the function so that control can be transferred to the end of the function on a return statement. The header file for environment entries in Figure 8.6 shows the additional information necessary for function entries. When a prototype is visited, two labels need to be created, and stored in the function environment along with the return type of the function and the formal parameter types. When a function definition is visited, the end-of-function label needs to be stored in an instance variable of the visitor, so that it can be retrieved when a return statement is visited.[2]

8.2 Building Assembly Trees for simpleJava in C

8.2.1 Project Definition

The assembly tree project has two parts – implementing the BuildTree interface to create abstract assembly trees, and adding calls to BuildTree methods to your semantic analyzer. It is strongly recommend that you

[2]Note if we visit a function definition for a function without a prototype, we will need to add the prototype information to the function environment.

test your BuildTree functions thoroughly before making changes to the semantic analyzer, so that don't try to chase down bugs in the wrong source file.

8.2.2 Project Difficulty

This project should be easier than the semantic analyzer, but more time consuming than the parsing and lexical analysis projects. Estimated completion time is 5-10 hours.

8.2.3 Project "Gotcha's"

- Be careful that structured variable accesses (instance variables and array dereferences) are correct. It is easy to add too many (or too few) Memory nodes in your abstract assembly tree.
- The size of a single variable should not be hard-coded into your tree generator or abstract assembly tree. Instead, use a constant (as defined in the file MachineDependent.h)
- Be sure to distinguish between the number of variables in an activation record, and the size of the activation record.

8.2.4 Provided Files

The following files are provided on the same CD as this document:

- **AST.h, AST.c** Abstract syntax tree files.
- **AAT.h, AAT.c** Abstract Assembly files.
- **type.h, type.c** Internal representations of types.
- **environment.h, environment.c** C files to implement environments. These are modified from the previous assignment to contain offset an label information.
- **AssemblyTreeTest.c** A main program that tests your assembly tree generator.
- **BuildTreeTest.c** A main program that tests you BuildTree interface.
- **ASTPrint.java** Methods that print out the abstract syntax tree, to assist in debugging.
- **AATprint.java** Methods that print out the abstract assembly tree, to assist in debugging.
- **MachineDependent.h** Machine dependent constants
- **register.h, register.c** Files for manipulating registers.
- **test1.sjava – testn.sjava** Various simpleJava programs to test your abstract syntax tree generator. Be sure to create your own test cases as well!

Chapter 9

Code Generation

Chapter Overview

9.1 Project Definition

For your final programming assignment, you will write a code generator that produces MIPS assembly code. For this project you will need to:

- Create a set of tiles to cover the abstract assembly tree. Be sure that your tile set includes small tiles that can cover each single node in the assembly tree (with the exception of move nodes).
- Create the code to go with each tile.
- Write C code that tiles the abstract assembly tree, and emits the proper assembly code.

Creating the tiles and associated code should be fairly straightforward, given the above notes. How can you create write a C program to do the actual tiling and emit the correct code? You will need to write a suite of mutually recursive functions to tile the tree. The two main functions are generateExpression, which generates code for an expression tree, and generateStatement, which generates code for a statement tree. Each of these functions will check to see which is the largest tile that will cover the root of the tree, make recursive calls to tile the subtrees, and emit code for the root tile. A skeleton of codegen.c is available in the assignment directory, pieces of which are reproduced in Figures 9.1 – 9.2

The function emit works exactly like printf (except the statements are printed to the file specified by the command line argument, instead of to standard out)

9.1.1 Project Difficulty

The largest difficulty for this assignment will not be creating the code generator, but debugging it. Writing and debugging a code generator that creates unoptimized assembly should take 5-10 hours. Optimizing the code generator can take an additional 10 hours, depending upon which optimizations you attempt.

9.1.2 Project "Gotcha's"

- Debugging assembly – especially machine generated assembly – can be quite difficult. Start small – build a code generator that can create unoptimized assembly for a small subset of simpleJava. Test and debug this simple code generator before expanding it to cover all of simpleJava

```
void generateExpression(AATexpression tree) {
  switch (tree->kind) {
  case AST_MEM:
    generateMemoryExp(tree);
    break;
  case AST_CONSTANT:
    emit("addi %s,%s,%d",ACC(),Zero(),tree->u.constant);
    break;
    ...  /* More cases for generateExpression */
  }
}

void generateMemoryExpression(AATexp tree) {
  /* This code only generates small, simple tiles  */
  generateExpression(tree->u.mem);
  emit("lw %s,0(%s) ",ACC(),ACC());
}
```

Figure 9.1: The beginnings of the function genExp, from the file codegen.c.

- Creating a code generator that just uses small tiles is easier that creating a code generator that uses complex tiles – but debugging assembly created from large tiles is much easier than debugging assembly created from small tiles.

9.1.3 Provided Files

The following files are provided in the Assembly Tree Generation segment of the web site for the text:

- **AST.h, AST.c** Abstract syntax tree files.
- **AAT.h, AAT.c** Abstract Assembly files.
- **type.h, type.c** Internal representations of types.
- **environment.h, environment.c** C files to implement environments.
- **sjc.c** Main program (simple java compiler)
- **ASTPrint.java** Methods that print out the abstract syntax tree, to assist in debugging.
- **AATprint.java** Methods that print out the abstract assembly tree, to assist in debugging.
- **MachineDependent.h** Machine dependent constants
- **register.h, register.c** Files for manipulating registers.
- **codegen.c** A sample skeleton code generator file
- **library.s** Assembly code for library functions (print, println, read), along with some setup code to make SPIM happy.
- **test1.sjava – testn.sjava** Various simpleJava programs to test your abstract syntax tree generator. Be sure to create your own test cases as well!

```
void generateStatement(AATstatement tree) {
  switch (tree->kind) {
  case AAT_JUMP:
    emit("j %s",tree->u.jump);
    break;
  case T_seq:
    genStm(tree->u.seq.left);
    genStm(tree->u.seq.right);
    break;
  case T_move:
    generateMove(tree);
   ...  /* more cases for statements  */
  }
}

void generateMove(AATstatement tree) {

  /* This code only generates small, simple tiles  */

  if (tree->u.move.lhs->kind == AAT_REGISTER) {
    genExp(tree->u.move.rhs);
    emit("addi %s,%s,0 ", tree->u.move.lhs->u.reg, Acc());
  } else {
    genExp(tree->u.move.rhs);
    emit("sw %s, 0(%s)", Acc(), AccSP());
    emit("addi %s,%s,%d",AccSP(), AccSP(), 0-WORDSIZE);
    genExp(tree->u.move.lhs->u.mem);
    emit("addi %s,%s,%d", AccSP(), AccSP(), WORDSIZE);
    emit("lw %s,0(%s)", Temp1(), AccSP());
    emit("sw %s,0(%s) ", Temp1(), Acc();
  }
}
```

Figure 9.2: The beginnings of the function generateStatement, from the file codegen.c.