Connections Beyond: Compilers/Interpreters

Comp 261/ Math 361

Many models of computation, despite having been developed for theoretical purposes, have practical applications. Also, there are many other models of computation that we don't have time to cover in class. These assignments will let your team explore one of these topics outside of what we cover in class. Form a team of **three** or **four** students, and complete the phases/milestones listed below before the due date for this option.

This option explores the connections between theoretical models and the front end of compilers or interpreters. Regular expressions and context-free grammars are central to defining most programming languages, and from them we create the part of the compiler that processes the text into individual tokens (words and meaningful symbols) and that parses the program tokens into its grammatical structure. You will learn to use compiler-generator tools, `flex` and `bison`, to do lexical and syntactic analysis, and you will write a grammar-checker for a small part of a programming language.

Project Timetable	
Milestone 1: Lexical Analysis	Start: when we cover regular expressions (about week 3)
Milestone 2: Parsing	Start: when we cover context-free grammars
Complete project and write-up	**Due: end of week following CFGs (about week 6)**

Milestone 1: Lexical Analysis with a Lexer-Builder

Compilers and interpreters start off with a phase called *lexical analysis* or *scanning*. This phase takes text characters and divides them into the meaningful words and symbols that make up the program. Similar techniques are used by other programs that read in formatted text input, including natural language processing systems.

The words, called *tokens*, are formed in regular patterns: they can be expressed by regular expressions and recognized by NFAs and DFAs. The set of all words we want to recognize could be thought of as one big regular expression, with different words or-ed together.

Lexical analyzers are not difficult to write, but they can be tedious. And many programs exist that can generate a lexical analyzer program for you. These programs take in a set of regular expressions for each token, and they convert that regular expression into an NFA, and from there to a DFA (using algorithms similar to those we described in class in proving that NFAs,

DFAs, and regular expressions are equivalent). The transition table for the DFA then forms the core of the lexical analyzer program, which simulates the DFA on the input strings. Each different final state of the DFA corresponds to a different identified token. The tokens can then be passed on to the next stage of the processing.

For this phase, you will learn to use an open-source lexer-generator: `flex`. This program is a free version of a very old lexer-generator called `lex` (free lex = flex). You can download `flex` and its companion parser-generator `bison`[1] or they will be installed on the department computers. On the Mac, the programs are typically installed as part of the command-line developer's tools, along with `gcc` and `g++`.

The `flex` program takes in a file that combines C code with regular expression patterns. It then generates a C program that can be compiled and run (you don't need a deep knowledge of C to do this project).

The `.lex` file format

Look at the Format section in the Flex reference manual for more details about the file format.

The specification of the lexical analyzer goes into a file with the extension `.lex`. There are three main sections to the Lex file: definitions, rules, and user code. These sections are separated from each other by the delimiter `%%` placed in the file. I will use the `example1.lex` file to illustrate each part, open that file in a text or program editor (I use emacs, but any editor that can keep the file as plain text should be fine).

The definitions section contains two things: user code and named patterns that don't correspond to tokens. C code that you want to appear at the top of the ultimate C file is placed in this section inside special marker: `%{` and `%}`. In particular, any libraries that are being included are put there. Below is that part of `example1.lex`. It includes two basic C libraries, and makes a "forward declaration" of a printer function that will be defined in a later section of the file.

```
%{
#include<stdio.h>
#include<stdlib.h>
void printer(char*);       // Forward declaration of printing function
%}
```

[1] `flex` and `bison` are free, open-source versions of the programs `lex` and `yacc` (Yet Another Compiler Compiler) that date back to the 1980s. The names are humorous/punny, like many old Unix programs. Do you get the pun?

After this subsection, you can define named patterns: regular expressions that you will find useful but that aren't tokens in their own right. First you give a name for this particular regular expression, and then the pattern. See the Patterns section in the Flex reference manual for more details about specifying regular expression patterns. It is considerably more elaborate than the regular expression forms from class, but still captures the same general set of languages. In `example1.lex`, there are two named patterns. The first defines a digit as any of the individual characters from 0 to 9 (0 or 1 or 2…). And the second defines an alphabetic symbol as any individual alphabetic character, upper or lower case.

```
digit     [0-9]
alpha     [a-z,A-Z]
```

The second section in the Lex file is the most important one: the rules section. In this section we list a rule, which consists of a regular expression in Flex form. Each pattern is followed by a snippet C code that will be performed every time that pattern is seen. A lexical analyzer working as part of a larger piece of software would return some value to identify the token, but in this phase you will just print the value.

```
%%

{alpha}({alpha}|{digit})*         { printer("Identifier"); }
{digit}+                          { printer("Integer"); }
"="                               { printer("Equals"); }
[ \t\n]+                          ; /*do nothing*/

%%
```

The rules in `example1.lex` describe a simple form of identifier (variable name) that starts with an alphabetic character and is followed by zero or more alphabetic or numeric characters. It also describes a simple integer (one or more digits). It matches the equals sign, and it also describes what to do when reading whitespace (space, tab, or newline). For the first three patterns, the program will call a special `printer` function I have provided. For whitespace, the program will do nothing.

Anything that does not match these three patterns will be printed to standard output and otherwise ignored. Including the rule for whitespace ensures that the whitespace is not printed to the output.

The final section contains any other code that is needed for your program. You can specify your own `main` function, if you want, but you don't need to. If none is specified, `flex` will generate a basic `main` function that loops until its input ends and repeatedly calls the lexer.

```
 int printer(char* str)
{
 printf("     Recognized %s: %s\n", str, yytext);
}
```

In this case, I have written a simple printing function that indents the output for things that matched the patterns, and also prints the specific text that matched the pattern, which is held in the global variable `yytext`.

Step one: running flex on my example

The first step is to make sure you can run the example correctly. The instructions below should be right for Mac or Linux machines, for Windows we'll have to do a bit of work together.

1. Open a terminal, and `cd` to the folder where `example1.lex` is stored.
2. Type: `flex example1.lex`. You should see nothing, but if you check the folder there should now be a file called `lex.yy.c`.
3. Compile the C program with: `gcc -o lexer lex.yy.c -ll` (the last part is dash-ell-ell). You might get warnings, but should get no errors, and it should create an executable called `lexer`.
4. Run the executable program by typing: `./lexer`. The lexer program reads input you type, and it displays the type, if it can determine it. If you type something it can't recognize, it prints it and goes on. Type some identifiers, numbers, equals, and some things that it won't recognize (you have to hit return before it will process a line). See what the output is. Type `Control-d` (hold down the `control/ctrl` key and hit the `d` key) to end the program.

Once you have `flex` working, the next step is to add to this example.

Step two: recognizing new patterns

1. Change the name of the Flex file to something else, like `myLang.lex`.
2. Modify the regular expression for identifiers so that it allows the underscore character, including as the first character in the variable name.
3. Modify the integer regular expression so that it has an optional sign (`+` or `-`).
4. Add regular expressions for the four basic arithmetic operators (`+`, `-`, `*`, `/`) and left and right parentheses. Make them respond in a way similar to identifiers and integers; print a

different name for each different operator and parenthesis (`Plus`, `Minus`, `Times`, `Divide`, `LParen`, and `RParen`).
5. Add a regular expression for floating-point numbers. Make it print `Float` along with the number.

Step three: support for statements

The tokens your lexer can identify so far allow you to specify basic expressions: numbers, variables, and arithmetic expressions. The last step is to add support for basic programming language statements: assignment statements, if-else statements, while and for loops. Rather than giving you the syntax to use here, you may choose a syntax for yourself. In your final write-up, describe the syntax you chose, explain the reasoning behind your choice, and give concrete examples of how to write code in that form.

Think about how you will indicate blocks of statements, how you will mark the end of an if or loop, and what form assignment statements should be in. Then, list the new tokens that your lexer needs to recognize for these statements. Add to your lexer file new patterns and the appropriate response.

Note: the lexer stops with the first pattern to match a given string of input. You can use this to recognize specific keywords while still allowing for a general regular expression for identifiers.

Milestone 2: Parser Generators

The second phase of a compiler or interpreter, after lexical analysis, is parsing, sometimes called syntactic analysis. In this phase the parser takes the tokens generated by the lexical analyzer, and it checks the sequence of tokens to see if the text is grammatical. A parser may also be responsible for building some kind of internal, structured representation of what it finds, often a parse tree.

It turns out that the most interesting feature of a parser is the grammar upon which it is based. There are several typical algorithms for parsing, but by and large implementing them can be left to a *parser generator* program instead of being hand-coded. Bison is a nice, free parser generator. It is often bundled with *gcc* compilers; it is part of the command-line tools on the Mac. As I noted earlier, the name Bison is a play on words: it was an open-source version of a previous program called `yacc`, which stood for “Yet Another Compiler-Compiler.” Both `yacc` and `bison` were originally created for Unix systems, where whimsy is a valued part of the operating system culture.

For this problem, your team will learn to use `bison` for its most simple purpose: creating a parser program that will check its input to see if it is a grammatical program. Much like `flex`, `bison` takes in a file that combines C code with specifications of tokens and grammar rules.

The `.y` file format

The specification of the grammar rules for Bison goes into a file with a `.y` extension. The sections of a Bison file are similar to the sections of a Lex file: definitions, rules, and user code. These sections are separated from each other by the delimiter `%%` placed in the file. I will use the `example1.y` file to illustrate each part. See the Bison Grammar File part of the Bison reference manual for further details.

The definitions section contains two things: user code and declarations of tokens to be used by the lexical analyzer and the parser. C code that you want to appear at the top of the ultimate C file is placed in this section inside special marker: `%{` and `%}`. Any libraries that are being included are put there, and declarations of functions that will be defined elsewhere (like `yyerror`, `yylex`, and `parseprint`):

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

void yyerror(const char* s) {
   fprintf(stderr, "%s\n", s);
};

int yylex();            // forward declaration of yylex, provided by flex
void parseprint(char*); // forward declaration of printing function
%}
```

Also in the first section, you put the definitions of the tokens on which your grammar will be built. These are typically shared with the lexical analyzer, as you will see later. In this case we have three tokens, for identifiers, integers, and the equals sign. For binary operators like addition and multiplication, you can also specify that grouping should happen from the left or from the right (so that `a + b + c` is interpreted uniquely as `(a + b) + c` and not as `a + (b + c)`). This way Bison can deal with grammar ambiguity. To do this, you use `%left` and then the name of the token (see Bison manual's precedence page for details).

```
%token INT
%token IDENT
```

```
%token EQUALS
```

The second section in the Bison file has the grammar rules. Each rule can be accompanied by C code to the right (much like for Flex) that can print values, build parse trees, or other similar things. In this case, you will just put in the grammar rules.

```
%%
assignments: assign
| assign assignments
;
assign:      IDENT EQUALS INT
 ;
%%
```

The rules in `example1.y` would look like the syntax below in the form we've been using in class. The rules specify that a grammatical text has a sequence of assignment statements. Each assignment statement has an identifier, the equals sign, and then an integer.

$$S \rightarrow A \mid AS$$

$$A \rightarrow \textit{ident equals int}$$

The final section contains any other code that is needed for your program. In this case I included a print function, and then a `main` function that calls `yyparse`, which is the parser function created by Bison. The process of parsing may print some information, but the `main` function then prints a message to indicate whether the parse succeeded or not.

```
void parseprint(char* str)
{
 printf("             PARSED: %s\n", str);
}

int main() {
 fprintf(stderr, "Enter statements/expressions to parse:\n"
 int res = yyparse();
 if (res == 0)
 fprintf(stderr, "Successful parsing.\n");
 else if (res == 1)
 fprintf(stderr, "Parsing failed due to incorrect input.\n");
 else if (res == 2)
```

```
 fprintf(stderr, "Parsing failed due to lack of memory.\n");
 else
 fprintf(stderr, "Weird value: %d\n", res);
}
```

Step one: changes to the Flex file

In the Flex file, there is a commented-out include statement, to include a local file called `example1.tab.h`. This file will be created by Bison and will contain the definitions of the tokens to be used by the lexical analyzer. Now that we have a Bison file, we're ready to uncomment this include statement, and to make other changes to the Flex file to connect it to the parser. We will change the code snippets associated with the lexer patterns so that they return the appropriate tokens.

1. First of all, uncomment the include statement described above.
2. Next, add a `return` statement to the code snippet after the call to `printer`, where you return the token associated with that pattern (e.g., `return INT;` for the integer).
3. Make sure you can generate a working lexer and parser with these example files.
    a. Open a terminal, and `cd` to the folder where `example1.lex` and `example1.y` are stored.
    b. Type: `bison -d example1.y` at the prompt. The `d` option causes Bison to create the `example1.tab.h` file. This will also create the C code for the parser, putting it in `example1.tab.c`.
    c. Type: `flex -o example1.lex.c example1.lex`. This will create the lexical analyzer and save its C code in `example1.lex.c`.
    d. Compile the C program with:
        `gcc -o example1 example1.lex.c example1.tab.c -ll -lm`
    (the second-to-last word is dash-ell-ell). This creates an executable called `example1`. You might get warnings, but if you get errors then there are issues to be resolved.
    e. Run the executable program by typing: `./example1`. It should wait for you to type input. Type in one or more valid assignment statements. To end the input properly, type `control-d`. Try some things that are invalid, as well.

Here is the series of calls to make **every time you make a change to the flex or bison files:**

```
$ bison -d example1.y
$ flex -o example1.lex.c example1.lex
$ bison -d example1.y
$ gcc -o example1 example1.lex.c example1.tab.c -ll -lm
$ ./example1
```

Once you have the example working, the next step is to modify it.

Step two: handling arithmetic expressions

1. Find your Flex file from Milestone One.
2. Rename the `example1.y` Bison file to match your Flex file, for example `myLang.lex` and `myLang.y`.
3. Modify the parser to add the following tokens: `FLOAT`, `LPAREN`, `RPAREN`, `PLUS`, `MINUS`, `TIMES`, `DIVIDE`, and `END`. Declare that the four arithmetic operators all group to the left.
4. Modify your Flex file like you did the `example1.lex` file in the previous step:
    a. Uncomment the include for the new `tab.h` file (change the local include to look for `myLang.tab.h`, or whatever makes sense for your file name).
    b. Modify the Flex file so that each of the patterns that corresponds to a token in the parser returns that token, as we did in Step one for `example1.lex`.
    c. Add a new rule to recognize a sequence of newline characters and return `END` for every such sequence.
5. Modify the parser so that it parses the grammar below instead. Translate these rules into Bison's form and incorporate them into your Bison file.

S → L | LS

L → *END* | E1 *END*

E1 → E1 *PLUS* E1 | E1 *MINUS* E1 | E2

E2 → E2 *TIMES* E2 | E2 *DIVIDE* E2 | E3

E3 → *INT* | *FLOAT* | *IDENT* | *LPAREN* E1 *RPAREN*

Step three: Extending to statements

The last step for this milestone is creating grammar rules for the statements you were asked to consider in Milestone One (assignment statements, if statements, for and while loops), and to integrate them into your parser and lexer. You will need tokens for each new keyword or piece of syntax, modifying the lexer to return those tokens. You will need to translate the rules into a form that Bison understands. And you need to make sure the new forms are unambiguous enough for Bison to work with them. Modify the grammar above so that the L rule (which specifies what may appear on a line) has an extra option to allow a statement before *END*, as well as an expression.

Final Write-up

In the end, you will hand in your Flex and Bison files, along with a write-up. Take this write-up seriously as a piece of writing. It is a report, but it should have introduction, conclusion, and flow. In your report, describe what happened with each milestone, how you tested your lexer and parser, any remaining bugs, what worked, and what didn't. Describe the set of tokens your lexer recognizes (in English, not just by giving the regular expressions for them), and provide the final grammar your parser recognizes, written in the form shown above, not Bison format. Find an organization for all of this that works and flows.