Flex Overview

Flex is a tool which allows us to build lexical analyzers quickly and easily. Given a list of regular expressions describing different tokens (and actions to take upon reading those tokens), Flex can generate a lexer in C or C++. Essentially, it does all the hard work for us, so we only need to worry about designing the syntax, rather than the implementation details.

Note that using Flex will require some background on regular expressions. Here is a good introduction to them.

A flex file takes the following form (in general):

```
%{

/* C Code to be placed at the top of the lexer */

%}

//additional Flex options

%%

<Regular expression>    {/*code action*/}
...

%%

/* C Code to be placed at the bottom of the lexer */
```

Each part is described as follows:

- The C code at the top is typically used for `#include` directives and global variable declarations.
- The additional Flex options are used for a wide variety of purposes, but are generally beyond the scope of the material being covered here.
- The regular expressions and code actions describe the lexer itself. Each regular expression describes a token, and the code action is C code which tells Flex what to do upon reading that token.
    - For the purposes of Programming Assignment #1, this will be your main concern for the Flex file.
- The C code at the bottom can be used for method definitions if needed, although it is often left blank.

Each regular expression has an associated code action, as shown above. The code action is run whenever the lexer reads text that matches that regular expression, and its job is to read and report all important information about the token that was read. This is done by returning an integer token and (optionally) by setting an additional value.

- The tokens are defined in an `enum` inside the parser (explained later). For example, say we've defined a token called `INT_LIT`. Whenever we read an integer literal, we want to return this token value so that the parser knows the type of token that was read.
- The optional additional value is part of a union called `yylval`, also defined in the parser. Depending on the type of token, we might want this additional value to take on one of many different types; for example, when reading an integer, we want to store an integer value, but when reading a float, we would want to store a float value. All of this behavior can be defined in the parser.

As an example, let's define the regular expression and code action for an integer literal. In our syntax (based on C), an integer literal is a non-empty sequence of digits 0-9 (so '123' and '5' are integer literals, but '3 42' and '3.14' are not). This can be described by the regular expression `[0-9]+`. When the lexer reads something of this form, it needs to report that what it read was an integer literal, and it needs to report its value, both are which are done in the code action. The resulting entry could look like this:

```
%%
...
[0-9]+  {yylval.intval = atoi(yytext); return
INT_LIT;}
...
%%
```

We set `yylval.intval` to the integer value which was read (note that `atoi` is C's built-in way to convert strings to integers), and afterwards we return the token value `INT_LIT`. Both the integer value and the token will be given to the parser, so it has all the information that it will need.

Most tokens are quite simple, and don't require the use of this extra value. For example, the semicolon character just has to return the token value `SEMICOLON`. There are a few, however, which may require some more thought. In particular, the string literals are likely the most complication token for this assignment, so they've been implemented for you (the implementation used is somewhat beyond the scope of this material).

Once the Flex file has been created, use the command `flex lexer.l` to generate the lexer automatically (this assumed that the file has been saved as lexer.l).

Bison Overview

Similar to Flex, Bison is a tool which allows us to build parsers quickly and easily. Given a context-free grammar and associated code actions (also known as syntax-directed definitions), Bison can generate a parser in C or C++. As with Flex, it does all the hard work for us, and follows a very similar syntax.

Note that this section requires familiarity with context-free grammars, which have been covered in lecture.

A Bison file takes the following form (in general):

```
%{

/* C Code to be placed at the top of the parser */

%}

/* definitions of tokens, states, etc, and additional
options */

%%

<Context-free grammar production>    {/*code action*/}
...

%%

/* C Code to be placed at the bottom of the parser */
```

As you can see, the general structure is very similar to that of a Flex file. Since parsers are generally more complicated than lexers, there are more additional options available for Bison than for Flex; these will not be explained here and have already been written for you in the starter code.

The code actions for Bison are less trivial than those in Flex. They'll be covered in detail in class and in the textbook whenever "Syntax-Directed Translation" is discussed. Generally speaking, their purpose is to transform the input text into an intermediate format that the compiler can work with – typically an Abstract Syntax Tree or AST. In code similar to the format you'll see in the Programming Assignment 1, we might do the following for an if statement:

```
%%
...
selStmt: IF LPAREN expr RPAREN stmt {
     $$ = make_node("selStmt");
     add_child($$, $3);
     add_child($$, $5);
} | IF LPAREN expr RPAREN stmt ELSE stmt {
     /* ... */
};
...
%%
```

Let's examine this piece-by-piece:

- `selStmt` = the name of the nonterminal for which we're defining a production.
- `IF LPAREN expr RPAREN stmt` = the production itself. Note that, unlike in your textbook, Bison uses a colon ':' instead of an arrow '->', but the production is otherwise the same.
    - `IF`, `LPAREN`, and `RPAREN` are names of tokens which we have defined.
    - `expr` and `stmt` are names of other nonterminals which we have defined.
- Inside the curly braces, we can C code for the action. Whenever a production is reduced, it is assigned a semantic value according to this code action - this value is referred to as `$$` inside the action. Here, you can see that we're setting the semantic value to a node by calling the `make_node` function, which has been implemented for you.
- Inside a production, you can access the semantic values for any of the children - here, we can access the semantic values of `expr` and `stmt` using `$3` and `$5` respectively. In general, the semantic value for the *n*th symbol in the production can be accessed by `$n`.
    - Note that some terminal symbols can also have semantic values, if desired. For example, integer literals might have a semantic value equal to the number that they represent. In general, the semantic value of a terminal symbol is whatever was stored in the `yylval` variable during lexing.
    - Here, we assume that each of `expr` and `stmt` have semantic values which are also nodes, so we add them as children of our new node.
- Just like in the textbook, we use a vertical bar to separate different productions for the same nonterminal.

There are a couple of other commands and details left to cover for Bison - these fall in the “definitions and additional options” section above.

- `%start <nonterm>` defines the starting nonterminal; i.e. the nonterminal for which the entire program should be a match. In our grammar, this is called `program`. (The angle brackets around the name should be excluded)
- `%union { int intval; double fltval; /* ... */ }` defines the union `yylval` discussed in the Flex section of this document. It follows the standard syntax of C union inside the curly brackets - each line contains a declaration.
    - Since this is a C union, it should only store “small” data, not anything large like arrays, structs, or objects. If you need one of these, instead store a pointer to it.
- `%token TOKEN1 TOKEN2 ...` defines `TOKEN1`, `TOKEN2`, etc to be tokens in the enum (again, this was discussed in the Flex section). All tokens needed for parsing and lexing must be defined here.
- `%type <typeval> symbol` defines the type of the semantic value of `symbol`. Every symbol with a semantic (terminal or nonterminal) must have its type declared in this fashion. Here, `<typeval>` is a member of the union defined above (for example, it could be `intval`).
- There are other options like these, but these ones are the mandatory ones that any Bison programmer must be aware of.

For more information on Flex and Bison, you can look at the O’Reilly book, which contains much more information about these very powerful programs.