Project 2: Lexer

Due: 15 February 2012

In this project we'll actually get started on our C compiler; we'll use `flex` to process a C program into a stream of tokens (not just lexemes) which will later serve as input into a parser.

1 Getting started with `flex`

This section will give you a starter file and (together with §3.5 in the book) the bare minimum of information about `flex` that you'll need to work on this project. If you have it, now would be the time to begin consulting the *flex & bison* book.

1.1 Basic `flex`

Although traditionally written with C as the target language, modern `flex` code can be written to target C++; but much of the sample "getting started" code still has C in mind. So here's a really basic `flex` program just to illustrate the general form and its usage. Type in the following file as `sample.ll`:

```
%option c++
%option noyywrap

%%

\"[^\"\t\n]+\"     { printf ("STR"); }
[a-zA-Z0-9_]+      { printf ("IDENT"); }

%%

int main()
{
  FlexLexer *sample = new yyFlexLexer();
  sample->yylex();
  return 0;
}
```

If you run

```
flex -oflexlexer.cpp sample.ll
```

it will create a (complete!) C++ program which it places in the file `flexlexer.cpp`. You can compile that file in the perfectly usual way:

```
g++ flexlexer.cpp -o lexer
```

(You probably want to keep track of all that in a Makefile; there will be more later.) The resulting `lexer` program will read from standard input and write to standard output, and if you redirect a small file in there (or just type into it) you'll find that any quoted string is replaced by `STR`, and any other word or number is replaced by `IDENT`. The highlights:

- The `%option`s say that we are targeting C++ and that we don't want to wrap to subsequent files.
- Any character that isn't part of a match to any rule in the rules section is just echoed to the output.
- Once `yylex` is called, it keeps going, trying to match input against rules, and when it matches, it executes the corresponding code.
- The `yyFlexLexer` class, a subclass of the abstract `FlexLexer` class, is automatically defined for you.

As a first attempt at tweaking a `flex` program, try modifying this one to print `NUM` instead of `IDENT` when the "identifier" is just a string of digits.

1.2 Using `flex` to generate tokens

That program is really just a regex-based transformer[1]; one call to `yylex` sends it on its merry way, with no further intervention. What we want, though, is a program that will return a token stream. For historical reasons, the "stream" here involves successive calls to the same `yylex` method, each one producing the next token; and for other historical reasons, the return value is an int (the token id) *but* the other information (i.e. the string that was matched) is provided through a separate function. The upshot of this is that you have to do a little extra work to assemble the parts of the token.

Type in the following file as `sample2.ll`:

[1] Specifically, it is a form of finite-state transducer.

```
%option c++
%option noyywrap

%{
#define ID 401
#define STR 402
#define NUM 403
#define OTHER 404
#include <fstream>
%}

%%

\"[^\"\t\n]+\"       { return STR; }
[a-zA-Z0-9_]+        { return ID; }
[^ \t\n]+            { return OTHER; }

%%

int main(int argc, char *argv[])
{
  std::ifstream src_in (argv[1]);
  FlexLexer *tokstream = new yyFlexLexer(&src_in);
  int tok;
  while ((tok = tokstream->yylex()) != 0)
    printf ("%d:%s ", tok, tokstream->YYText());
  return 0;
}
```

As before, run `flex` to generate the lexer's C++ form, and then compile that and try it out. The burden of output has shifted into the `main` function; what is left in the rules section is only brief code to decide what token id to return. Some more highlights:

- There is no special significance to the numbers 401, 402,..., except that they're less likely to be confused with naturally-arising values in the source code.
- An alternate constructor for the lexer class accepts a pointer to an `istream`, which lets you read from a file or stringstream. (In practice, you might still want to use the other one for reasons I'll explain later, but I wanted to show you this.)
- The `yylex` method returns zero when the end of input is reached.
- After `yylex` has been run (and until the next time it is run), accessor methods of the lexer class let you get at other token attributes.

> Notably, the `YYText()` method gives the actual matched text; but BEWARE: it returns a `char *` that is only good until `yylex` is next called! If you need to hang on to this value, you need to copy it yourself (e.g. by making a `string`).

Tweak this one too, first by adding in your changes that recognise numbers (and returning the NUM token for them); and then to ensure that every character of the input gets matched against some rule (so that it doesn't echo to the output). Remember that not every rule has to return a token—this will be important if you want to skip whitespace without printing or making a corresponding token.

1.3 State within the lexer

The traditional way to pass additional information out of `yylex` and maintain state between calls was to just use a lot of global variables; easy in some ways, but it has the disadvantage of being hard to keep track of, and since it's not "re-entrant" you can't use multiple lexers in the same program or do anything in parallel. It's also not very C++ish.

This third version of the sample lexer will pull a couple tricks to let us use our more sophisticated C++ techniques (encapsulation, instance variables) with the code generated by `flex`. First, edit a new file named `lexer.h` and type in the following:

```
#ifndef _LEXER_H_
#define _LEXER_H_

#ifndef __FLEX_LEXER_H
#include <FlexLexer.h>
#endif

class SampleLexer : public yyFlexLexer
{
  public:
    virtual int yylex ();
    int get_lineno() const;

  private:
    int lineno;
};
#endif
```

In most respects this is a typical header file declaring a typical C++ class. The header it includes comes with flex and is already installed for you in `/usr/include/`. The class you define is a derived subclass of the lexer class that is autogenerated by flex.

Next, create a file `lexer.cpp` as follows:

```
#include "lexer.h"
#include <cstdio>
using namespace std;

int SampleLexer::get_lineno() const
{
  return lineno;
}

int main ()
{
  SampleLexer *tokstream = new SampleLexer();
  int tok;
  while ((tok = tokstream->yylex()) != 0)
  {
    printf ("%d:%s:%d ", tok, tokstream->YYText(), tokstream->get_lineno());
  }
  return 0;
}
```

The `main` function you define here is nearly the same as the one from `sample2.ll`, and you can pull that version into the file if you like—the primary changes are in printing the line number, and in how the lexer is constructed.

Finally, copy `sample2.ll` into a new file `sample3.ll` and modify it by adding the following two lines to the top group of definitions:

```
#define YY_DECL  int SampleLexer::yylex()
#include "lexer.h"
```

Then add a rule that matches `\n` and executes the code `{ ++lineno; }` and finally, remove `main` and replace it with

```
int yyFlexLexer::yylex()
{
  return -1;  // never called; should call derived version
}
```

The trick of the thing comes in the `YY_DECL` macro; the main function that `flex` autogenerates will have as its function header whatever you provide in this macro. What that means in this case is that although everything else in the autogenerated autolexer.cpp file has to do with the autogenerated class yyFlexLexer, you have fooled `flex` into defining its workhorse `yylex` as a method of *your derived class*, where it—and therefore any code executed by a rule—has access to read and/or modify any of the derived class's instance variables and methods.

Since the autogenerated code will no longer include a definition for the promised `yylex` method of `yyFlexLexer`, we have to provide a dummy body for that method (which will, however, never be called, since anyone making a `yyFlexLexer` will actually be making a `SampleLexer`).

The trick in this section can be used to arbitrarily expand what your rule code can do and store; in particular, this will be an important mechanism that will let you keep track of your symbol table. Just create the table when the lexer is constructed, and update it when the appropriate rules fire.

As an aside, once you have autogenerated lexer code (`autolexer.cpp`) and hand-written lexer code (`lexer.cpp`) side-by-side in your directory, you need to be somewhat careful not to overwrite your code with a careless -o option! Makefiles are a great idea on a project like this.

2 Lexing C

At this point, your job is to convert the input (a character stream) into an output (token stream) that is tailored specifically to being useful input to the parser. That means distinguishing anything that the syntax relies on, although you can group together elements that behave identically with respect to syntax trees. (The assignment operators are a great candidate for this sort of thing.) Note that by common convention, terminals that correspond to single-character operators are usually assigned the ASCII value of that character as their id. (So the terminal label returned when the match is `"+"` would be `'+'`.) If you do this, make sure that none of your *other* terminal labels overlap this ASCII range.

Here is a complete list of punctuation used in operators in C (from p19 of the C book):

`(`	`--`	`<`	`&&`	`%=`
`)`	`+`	`<=`	`||`	`&=`
`[`	`-`	`>`	`?`	`^=`
`]`	`*`	`>=`	`:`	`|=`
`->`	`&`	`==`	`=`	`<<=`
`.`	`/`	`!=`	`+=`	`>>=`
`!`	`%`	`&`	`-=`	`,`
`~`	`<<`	`^`	`*=`	
`++`	`>>`	`|`	`/=`	

plus three lexemes that are not operators but figure into the grammar: `{` (left curly), `}` (right curly), and `...` (ellipsis).

You will also need to recognise identifiers and reserved words. The following is the complete list of reserved words in C99 (per the list on p7 of the C book):

`auto`	`double`	`inline`	`sizeof`	`volatile`
`break`	`else`	`int`	`static`	`while`
`case`	`enum`	`long`	`struct`	`_Bool`
`char`	`extern`	`register`	`switch`	`_Complex`
`const`	`float`	`restrict`	`typedef`	`_Imaginary`
`continue`	`for`	`return`	`union`	
`default`	`goto`	`short`	`unsigned`	
`do`	`if`	`signed`	`void`	

In addition to those, you'll need to recognise four kinds of literals:

Integer. You should recognise integer literals in base 8, 10, and 16; the value to be stored should be the appropriate integer (e.g. for `0x23` the value stored should be the same as for `35` and for `043`). You should recognise the suffixes L, LL, and U, and their lowercase counterparts.

Floating point. You should recognise floating point literals whether they have a decimal point or an exponent, or both. Again, the value to be stored should be the appropriate double value. Note that the exponent marker can be E or P (or their lowercase versions).

Character. The value that will be stored for a character literal will be the integer value of the character. Note that a newline will probably be entered as `'\n'` but should be stored as 10 (i.e. the ASCII for linefeed); and likewise for the other special characters.

String. The value to be stored for a string literal will be an integer index into a string table (which is similar to, but not the same as, the symbol table). If the same string appears more than once in a file, it should get the same index each time, just like for identifiers.

The final thing you need to recognise is a special: lines that begin with a hash mark (`#`) are not C code per se but notes to the compiler. Pull these off the input stream and create a special token with the full contents of the line.

Many tokens will thus have two parts, just as we've discussed in class: a terminal label, and a semantic value indicating which instance of that terminal the token represents. Rather than returning it as a pair, `flex` is set up to only return the terminal label; you are responsible for updating instance variables so that a subsequent call to a `semantic_int()` method (which you will have to define) will return the associated number. The interpretation of that semantic value will vary, of course; for an integer literal the value will be the actual value, while for an identifier it will be an index into a symbol table.

3 Expected output

The main task of this project is producing the lexer itself, which will eventually drop in to the pipeline and produce output for the parser; but for now, for purposes of testing and evaluation, you should have it output a table. Each line corresponds to a single token, and contains three columns.

The first column contains the terminal label for the token. The second column contains the string that matched; in many cases this will be redundant, but in many cases it's open-ended (e.g. with identifiers). The third column contains the semantic value associated with the string that was matched, whether that is a table index, or just a number, or whatever.

For instance, output on a simple C file might begin as follows:

```
IDENT       "int"     1
IDENT       "main"    2
'('         "("
')'         ")"
IDENT       "int"     1
```

```
IDENT       "n"         3
'='         "="
INTLIT      "3"         3
';'         ";"
```

It does *not* have to look *exactly* like this, but the format should be clear and documented.

3.1 Context and subsetting and extras

The actual input to the lexer in real life wouldn't be a C file itself, but the output of the C preprocessor. That's why you don't have to worry about comments. You can run it by hand; if you type

```
cpp testfile.c
```

the preprocessor does its thing and sends the result to standard output. If your lexer reads from standard input, you can just pipe it right through:

```
cpp testfile.c | lexer
```

The output from the lexer will, eventually, become the input to the parser. There will be a little tweaking, but for the most part you'll be able to use your `yylex` without change when we build a parser. (In fact, one reason we're *not* modifying `yylex` itself to return the entire token pair is because that would break the connection with the autogenerated parser; parsers produced by yacc and bison expect to call `llval`, get a terminal label, and only then ask for the semantic value associated with it.

All the above is specified with respect to the full C language, whole and entire. Some of it we may never get around to implementing, but lexing an additional type of terminal label adds little marginal work to this project. Still, there are a few things that are a bit of a pain to get just right, and you don't have to worry about them.

- Keeping track of the suffixes for the number literals.
- Computing the actual value of all forms of floating point literals (including exponentiated ones).

- Handling numeric escapes in character and string literals.

Getting the octal and hex values is not as difficult as you might think—see `strtod`.

The "other" line type gives rise to an important extra, if you have time. It's not part of the C language proper, but part of the output of the preprocessor, and they correspond to where there used to be `#include` lines, to tell which line of which source file we've just switched to. This is crucial information if you plan to write helpful error messages! So while you don't have to process these (just get them and then throw them out) if your goal is just compiling, it's a useful extra to process them meaningfully and have them affect `lineno` and so on.

4 Handing in, etc

On torvalds, type

```
handin cmsc445 proj2  lexerdir/
```

to hand in your entire `lexerdir` directory. Be sure to include a readme with any notes I'll need, and your Makefile if you have one.

I should probably mention that the general structure of the project, and the first example lex file, were inspired by a project I did when I took a compilers class at Brown. I'd like to say the rest of it was inspired by all the various websites I've looked at with descriptions on how `flex` can work with C++, but quite frankly, half of them contradict each other and/or document out-of-date versions, so that's rather more my own.