Flex and lexical analysis

October 25, 2016

Flex and lexical analysis

From the area of compilers, we get a host of tools to convert text files into programs. The first part of that process is often called lexical analysis, particularly for such languages as C.

A good tool for creating lexical analyzers is flex, based on the older lex program. Both take a specification file and create an analyzer, usually called lex.yy.c.

Flex is reasonably compatible with the UTF-8 encoding for Unicode.

Terminology

- A token is a minimal group of characters having collective meaning.
- A lexeme is an actual character sequence forming a specific instance of a token, such “book”.
- A pattern is a rule expressed as a regular expression (or even an extended regex) describing how a particular token can be formed. For example, a common convention for variable name tokens is [A-Za-z][A-Za-z_0-9]*.
- Characters between tokens are generally called whitespace; these include spaces, tabs, newlines, and formfeeds.

Attributes for tokens

One common characteristic of standard lexical analysis (but certainly not universal) is the assignment of types; another is passing attributes to the caller.

Upon recognizing a numerical constant, for instance, the scanner might pass back the the type (e.g., integer), the lexeme string, and the value as an C int.

Upon recognizing an identifier, the lexer might pass back the type (e.g., identifier), the lexeme string, and a pointer to information about the identifier.

General approaches to lexical analysis

Use a tool, like flex or re2c. (Example: code)

Write a one-off analyzer in your favorite programming language. (Most common strategy these days.) For example, you can use libc's strtok() function. (Example: code)

Write a one-analyzer in assembly. (Usually done for bootstrapping purposes, though lexical analysis in assembly against a mmap(2) can be an exceedingly fast technique.)

Flex

While it might be found in some libc's, you might also have to link explicitly with -lfl.

The lexer function is called yylex(), and it is quite easy to interface with bison/yacc.

```
*.l file --> flex --> lex.yy.c

lex.yy.c --> C compiler --> lexical analyzer

input stream --> lexical analyzer --> actions taken when ru
```

Using Flex

Flex source structure:

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

Definitions

- Declaration of ordinary C variables and whatnot.
- flex definitions

Rules

The form of rules are

```
regularexpression      action
```

The actions are C code.

Flex's regular expressions

`s`	literal string s
`\c`	character c literally
`[s]`	character class
`^`	beginning of line
`[^s]`	characters not in character class
`s?`	s occurs zero or one time

Flex's regular expressions

```
.        any character except newline

s*       zero or occurrences of s

s+       one or more occurrences of s

r|s      r or s

{s}      grouping

$        end of line

s{m,n}   m through n occurrences of s
```

Examples

```
a*        zero or more a's

.*         zero or more of any char except newline

.+         one or more characters

[a-z]      a lower-case letter

[a-zA-Z]   any letter

[^a-zA-Z] not a letter
```

Examples

`a.b`	a followed by any char then followed by b
`rs|tu`	rs or tu
`a(b|c)d`	abd or acd
`^start`	"start" at the beginning of line
`END$`	the characters END followed by end-of-line

Flex actions

Actions are just C code. If it is compound, or requires more than a single line, enclose with curly braces.

Examples:

```
[a-z]+      printf("found word\n");
[A-Z[a-z]*  { printf("found capitalized word:\n");
            printf("   '%s'\n",yytext);
            }
```

Flex definitions

The form is simply

```
name definition
```

The name is just a word beginning with a letter (or underscore, but I don't recommend those) followed by zero or more letters, underscores, or dashes. The definition actually from the first non-whitespace character to the end of line. You can refer to it via {name}, which will expand to your definition.

Flex definitions

For example:

```
DIGIT [0-9]
{DIGIT}*\.{DIGIT}+
```

is equivalent to

```
([0-9])*\.([0-9])+
```

Flex example

```
%{   int num_lines = 0;
     int num_chars = 0;
%}
%%
\n {++num_lines; ++num_chars;}
.  {++num_chars;}
%%
int main(int argc, char **argv)
{
  yylex();
  printf("# of lines = %d, # of chars = %d\n",
         num_lines, num_chars);
}
```

code

Another example

```
digits    [0-9]
ltr       [a-zA-Z]
alphanum  [a-zA-Z0-9]
%%
(-|\+)*{digits}+       printf("found number: '%s'\n",yytext)
{ltr}(_|{alphanum})*   printf("found identifier: '%s'\n",yyt
\.                     printf("found character: {%s}\n",yyte
.                      { /* ignore others */ }
%%
int main(int argc, char **argv)
{
  yylex();
}
```

code