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`. It takes a specification file and creates an analyzer, usually called `lex.yy.c`.

Lexical analysis terms

- A token is a group of characters having collective meaning.
- A lexeme is an actual character sequence forming a specific instance of a token, such as `num`.
- A pattern is a rule expressed as a regular expression and describing how a particular token can be formed. For example, `[A-Za-z][A-Za-z_0-9]*` is a rule.
- Characters between tokens are called whitespace; these include spaces, tabs, newlines, and formfeeds. Many people also count comments as whitespace, though since some tools such as `lint/splint` look at comments, this conflation is not perfect.

Attributes for tokens

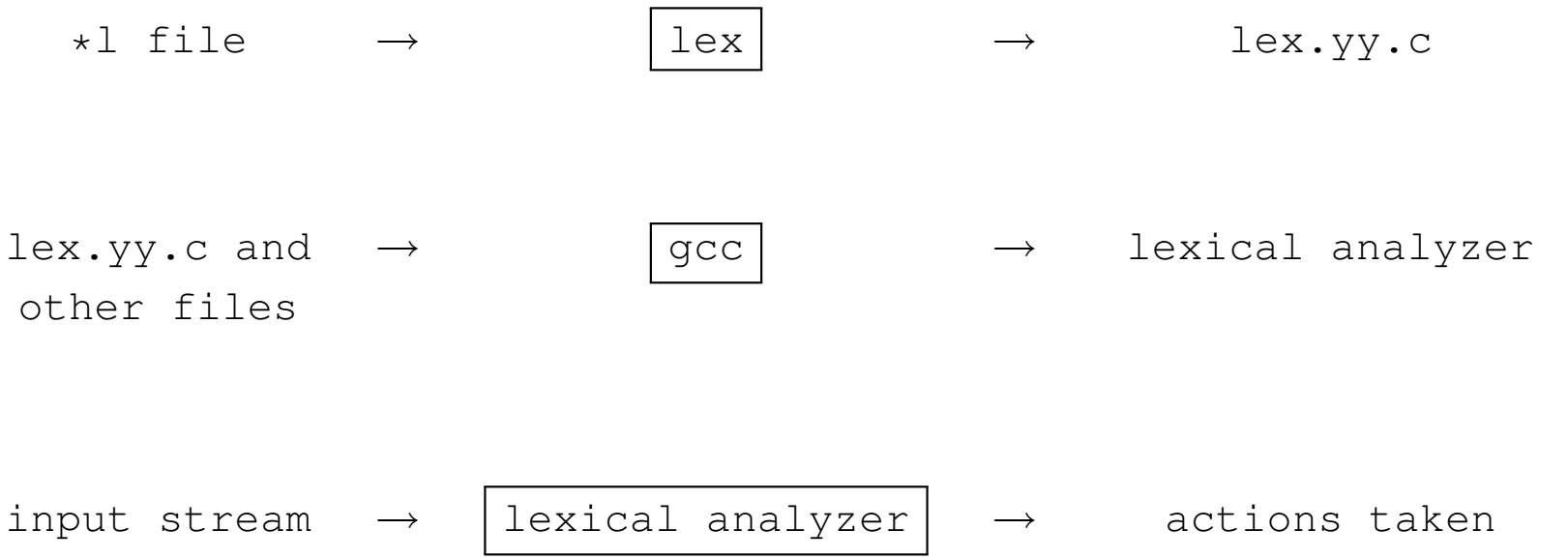
Tokens can have attributes that can be passed back to the calling function.
Constants could have the value of the constant, for instance.
Identifiers might have a pointer to a location where information is kept about the identifier.

Some general approaches to lexical analysis

Use a lexical analyzer generator tool, such as `flex`.
Write a one-off lexical analyzer in a traditional programming language.
Write a one-off lexical analyzer in assembly language.

Flex - our lexical analyzer generator

Is linked with its library (`libfl.a`) using `-lfl` as a compile-time option (or is now sometimes/often found in libc).
Can be called as `yylex()`.
It is easy to interface with `bison/yacc`.



Flex specifications

Lex source:

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

Definitions

- Declarations of ordinary C variables and constants.
- `flex` definitions

Rules

The form of rules are:

```
regularexpression     action
```

The actions are C/C++ code.

Flex regular expressions

`s`	string s literally
`\c`	character c literally, where c would normally be a lex operat
`[s]`	character class
`^`	indicates beginning of line
`[^s]`	characters not in character class
`[s-t]`	range of characters
`s?`	s occurs zero or one time

Flex regular expressions, continued

```
.         any character except newline
s*        zero or more occurrences of s
s+        one or more occurrences of s
r|s       r or s
(s)       grouping
$         end of line
s/r       s iff followed by r (not recommended) (r is *NOT* consumed)
s{m,n}    m through n occurences of s
```

Examples of regular expressions in `flex`

`a*`	zero or more a's
`.*`	zero or more of any character except newline
`.+`	one or more characters
`[a-z]`	a lowercase letter
`[a-zA-Z]`	any alphabetic letter
`[^a-zA-Z]`	any non-alphabetic character
`a.b`	a followed by any character followed by b
`rs|tu`	rs or tu
`a(b|c)d`	abd or acd
`^start`	beginning of line with then the literal characters start
`END$`	the characters END followed by an end-of-line.

Flex actions

Actions are C source fragments. If it is compound, or takes more than one line, enclose with braces ('{' '}').
Example rules:

```
[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 an underscore, but I don't recommend those for general use) followed by zero or more letters, underscore, or dash. The definition actually goes from the first non-whitespace character to the end of line. You can refer to it via `{name}`, which will expand to `(definition)`. (cite: this is largely from "man flex".)

For example:

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

Now if you have a rule that looks like

```
{DIGIT}*\.{DIGIT}+
```

that is the same as writing

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

An example Flex program

```
    /* either indent or use %{ %} */
%{
    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 );
}
```

Another example program

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