Features in Flex

- Start conditions
- Values available to the user
- C++ lexical analyzer
- Interfacing with yacc

Start conditions

- flex provides a mechanism for conditionally activating rules
- Any rule whose pattern is prefixed with "<sc>" will only be active when the scanner is in the start condition named "sc“

Start condition example

```
<STRING>[^"]* { /* eat up the string body */
... }
```

will be active only when the scanner is in the "STRING" start condition

```
<INITIAL,STRING,QUOTE>\. { /* handle an escape */
... }
```

will be active only when the current start condition is either "INITIAL", "STRING", or "QUOTE".

Start condition declaration

- Start conditions are declared in the definitions (first) section of the input using unindented lines beginning with either `%s' or `%x' followed by a list of names
  - The former (%s) declares *inclusive* start conditions
  - the latter (%x) declares *exclusive* start conditions

Activation

- A start condition is activated using the BEGIN action
- Until the next BEGIN action is executed, rules with the given start condition will be active and rules with other start conditions will be inactive
- If the start condition is *inclusive*, then rules with no start conditions at all will also be active

Exclusive

- If it is *exclusive*, then *only* rules qualified with the start condition will be active
- A set of rules contingent on the same exclusive start condition describe a scanner which is independent of any of the other rules in the flex input
  - Because of this, exclusive start conditions make it easy to specify "mini-scanners" which scan portions of the input that are syntactically different from the rest (e.g., comments).

Little Example

- Part of a flex file with start conditions:

```
%x example                                  declares example
%%
<example>foo    do_something();             only rule used
bar             something_else();
"=="            BEGIN(example);             activates example
%%
```

Controlling Start Conditions

- Once activated, a start condition is in effect until another start condition is activated
  - So how do we get back to our initial state?
- BEGIN(INITIAL) or BEGIN(0) reset the start condition back to the original state
- Also <*> can be used as a start condition that will match any start condition

C comment example

- Here is a scanner which recognizes (and discards) C comments while maintaining a count of the current input line

```
%x comment
%%
    int line_num = 1;
"/*"                   BEGIN(comment);
<comment>[^*\n]*       /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*  /* eat up '*'s not followed by '/'s */
<comment>\n            ++line_num;
<comment>"*"+"/"       BEGIN(INITIAL);
```

Values available to user

- 'char *yytext' holds the text of the current token.
- 'int yyleng' holds the length of the current token.
- 'FILE *yyin' is the file which by default flex reads from.
- 'void yyrestart(FILE *new_file)' may be called to point yyin at the new input file.
- 'FILE *yyout' is the file to which `ECHO' actions are done.
- YY_START returns an integer value corresponding to the current start condition.

C++ lexical analyzer

- **`flex` provides two different ways to generate scanners for use with C++**
- **The first way is to simply compile a scanner generated by `flex` using a C++ compiler instead of a C compiler**
- **You can then use C++ code in your rule actions instead of C code**
  - **Note that the default input source for your scanner remains `yyin`, and default echoing is still done to `yyout`. Both of these remain `FILE *' variables and not C++ `streams`.**

C++ lexical anlyzer

- You can also use `flex` to generate a C++ scanner class, using the `-+' option
  - This option is automatically specified if the name of the flex executable ends in a `+', such as flex++
- When using this option, flex defaults to generating the scanner to the file `lex.yy.cc' instead of `lex.yy.c'
- The generated scanner includes the header file `FlexLexer.h', which defines the interface to two C++ classes.

C++ classes

- The first class, FlexLexer, provides an abstract base class defining the general scanner class interface. It provides the following member functions:
  - 'const char* YYText()' – returns the text of the most recently matched token, the equivalent of yytext.
  - 'int YYLeng()' – returns the length of the most recently matched token, the equivalent of yyleng.
  - 'int lineno() const' – returns the current input line number
  - 'void set_debug(int flag)' – sets the debugging flag for the scanner, equivalent to assigning to yy_flex_debug
  - 'int debug() const' – returns the current setting of the debugging flag.

C++ classes

- The first class, FlexLexer, provides an abstract base class defining the general scanner class interface.
- The second class defined in \`FlexLexer.h' is yyFlexLexer, which is derived from FlexLexer.
- For details on these classes you should visit http://dinosaur.compilertools.net/flex/

Interfacing with yacc/bison

- One of the main uses of flex is as a companion to the yacc parser-generator
- yacc parsers expect to call a routine named \`yylex()' to find the next input token
- The simplest way is to use an include statement in the yacc/bison file (I'll show an example)

Introducing Bison

- Bison is like yacc (I'd suggest using bison, but if you only have access to yacc, then use yacc)
- Input to Bison specifies the grammar rules and the semantic actions
- Often flex is used for the lexical analysis

General Bison file format

- The general form of a Bison grammar file is as follows:

  %{
  C declarations
  %}
  Bison declarations
  %%
  Grammar rules
  %%
  Additional C code

  - The \`%%', \`%{' and \`%}' are punctuation that appear in every Bison grammar file to separate the sections.

Example

- Consider the grammar

  S ? LET OP A
  A ? A DIG | DIG

Bison file

- In **bison.y** there would be:

```
%{
%}
%token NUM, LET, OP
%%
S:          LET  OP  A      {printf("Found S\n");}
;
A:          A NUM           {printf("Found A NUM\n");}
            | NUM           {printf("Found NUM\n");}
;
%%
#include "lex.yy.c"
```