The Flex Scanner Generator

- As we've seen, there's a lot that can go into a scanner, and implementing a working scanner for a real language can be a huge undertaking.

- While some modern compiler writers still choose to make this undertaking and hand code a custom scanner for their language, many others take an easier route: using a tool called a ***scanner generator*** to automatically generate a working scanner for their language.

- A scanner generator typically takes in a specification of the source language's microsyntax in the form of a collection of regular expressions and outputs code implementing a scanner for that microsyntax specification.

- One of the most widely-used scanner generators is Flex (Fast LEXical analyzer generator).

- Flex is an open-source successor of Lex, a scanner generator written in the 1970's by Mike Lesk and Eric Schmidt, the latter of whom would eventually go on to become CEO of Google among other things.

- At its core, Flex allows a compiler writer to specify a source language's microsyntax as a collection of regular expressions and to attach actions, in the form of C/C++ source code, to each regular expression.

- To generate a scanner, Flex reads the microsyntax specification (i.e. the regular expressions and their attached actions) written by the compiler writer and outputs C/C++ code containing a corresponding scanning function. This code can then be compiled into an application from which the scanning function is called to scan input source code.

- A Flex microsyntax specification is typically written in a file with a `.l` extension. This input file consists of 3 main sections separated by double percent sign (`%%`) delimiters:

```
definitions
%%
rules
%%
user code
```

- Let's look at each of these sections separately.

The definitions section of the Flex input file

- The definitions section of the Flex input file can contain three main things:
  - Initial code declarations
  - Name definitions
  - Flex option directives

Initial code declarations

- Initial code declarations must go within a block enclosed within the delimiters `%{` and `%}` and consist of C/C++ code to be copied verbatim to the top of the generated scanner file.

- The code placed in the initial declarations is typically used to perform some kind of initialization.

- For example, the following initial code declarations might be used to set up a scanner that counted characters, words, and lines in the source code:

```
%{
int num_chars = 0;
int num_words = 0;
int num_lines = 0;
%}
```

Name definitions

- Name definitions are named regular expression patterns used to help make the rules section of the Flex input file more readable. Each name definition consists of a name and a regular expression pattern:

```
name pattern
```

- Here, `name` may be any standard C-style identifier, and `pattern` is a regular expression specification that begins with the first non-whitespace character after name and runs to the end of the line.

- Once a name definition is made, `{name}` can be used in the rules section to refer to `(pattern)`.

- For example, the following name definition could be specified:

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

- Then, in the rules section, `DIGIT` could be used to build larger regular expressions, e.g.:

```
{DIGIT}+"."{DIGIT}*
```

- This would be equivalent to the following regular expression matching unsigned floating point numbers:

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

- We'll talk more later about how regular expression patterns can be defined in Flex.

Flex option directives

- Flex makes available many options for controlling its behavior. Settings for these options may be specified in the definitions section of the Flex input file using `%option` directives.

- For example, the compiler writer can direct Flex to make the number of the current source code line being scanned available in a global variable `yylineno` by specifying the following option directive:

```
%option yylineno
```

- Option directives can also be used to turn off certain behaviors. For example, the following option directive tells the scanner *not* to try to call the function `yywrap()` when the end of the source code is reached, instead assuming that there are no more files to scan:

  ```
  %option noyywrap
  ```

- More information on the available options can be found [in the Flex documentation](#).

The rules section of the Flex input file

- The rules section of the Flex input file contains a set of rules defining the microsyntax recognized by the generated scanner. Each rule consists of a regular expression pattern and an associated action:

  ```
  pattern action
  ```

- Here, `pattern` is an un-indented regular expression specification, and `action` is a set of C/C++ statements to be executed whenever a string in the source code matches the specified regular expression.

- The action associated with a particular pattern begins with the first unescaped whitespace character after the pattern and continues until the end of the line. An action can be made to span multiple lines by enclosing it within curly braces (`{}`).

- For example, both of the following are valid Flex rules that increment values declared and initialized in the definitions section of the Flex input file:

  ```
  \n    {
              num_lines++;
              num_chars++;
        }

  .     num_chars++;
  ```

- In addition to such rules, the rules section may also contain, before the first rule definition, an optional block of C/C++ code enclosed within the delimiters `%{` and `%}` that is copied verbatim to the top of the generated scanning function.

    - This code is local to the scanning function and is invoked each time the scanning function is called. It can include, among other things, initializations that could not be written in the global code included in the initial declarations in the definitions section of the flex input file (e.g. function calls, etc.).

- There are a number of useful methods and global values that can be used within the code for an action. We'll take a look at these after looking at how to express regular expression patterns in the rules.

Regular expression patterns in Flex rules

- Flex provides an extended regular expression syntax that provides many conveniences beyond the basic and shorthand regular expression operators we looked at in class.

- Here is a subset of the regular expression syntax available in Flex:

`c`	Single character	Matches the single character `c`.
`rs`	Concatenation	For any two regular expressions `r` and `s`, matches the concatenation of `r` and `s`.
`r|s`	Union	For any two regular expressions `r` and `s`, matches the union of `r` and `s`.
`r*`	Closure	For any regular expression `r`, matches the closure of `r`, i.e. zero or more `r`'s.
`r+`	Positive closure	For any regular expression `r`, matches one or more `r`'s.
`r?`	Optional	For any regular expression `r`, matches. zero or one `r`'s (i.e. an optional `r`).
`.`	Any character	Matches any single character except a newline. Any operation may be applied, e.g. `.+` matches at least one of any non-newline character, or `a.*` matches any word that starts with `a`.

`\`*`c`*	Escape	If *`c`* is one of `a`, `b`, `f`, `n`, `r`, `t`, or `v`, then this matches the ANSI-C interpretation of `\`*`c`*. Otherwise, matches a literal *`c`*. Can be used to escape other operators in the regular expression syntax. For example `\t` matches a tab character and `\*` matches the character `*`.
`[abc]`	Character class	Matches the union of the characters specified, e.g. `[abc]` matches `(a|b|c)`, i.e. either a single `a`, a single `b`, or a single `c`.
`[0-9]`	Character range class	Matches the union of the characters in the specified range, e.g. `[0-9]` matches any digit between 0 and 9. Multiple ranges may be specified, e.g. `[a-zA-z0-9]`.
`[^]`	Complement	Indicates the complement of a character class, i.e. the union of all characters *except* the ones specified in the class. For example, `[^ \t\n]` matches any single character except space, tab, or newline, and `[^0-9]` matches any single non-digit character.
`^`	Beginning of line	Matches the beginning of a new line of text, e.g. `^a+` matches one or more `a`'s at the beginning of a line, and `^[0-9]` matches any single digit at the beginning of a line.
`$`	End of line	Matches the end of line of text, e.g. `;$` matches a semicolon at the end of a line, and `[ \t]*$` matches any number of spaces or tabs at the end of a line.
`""`	Literal string	Quoted strings are matched literally. For example `"x.*"` matches an `x` followed by a dot followed by an asterisk.
`(`*`r`*`)`	Parentheses	Parentheses group multiple regular expressions in order to override precedence, e.g. `(cat)*` matches any number of the string `cat`.

`<<EOF>>`	End of file	Matches the end of the input source code file.

- A full list of the regular expression patterns available in Flex is provided in the Flex documentation.

- A Flex-generated scanner will read characters one-by-one from the input source code and gather them into a sequence until it forms the longest possible match of any of the specified regular expression patterns, at which point it will execute the action corresponding to that pattern.
    - If multiple patterns match an equally long sequence of input characters, the one listed first in the Flex specification is used.

- If a token exists in the source code that is not matched by any pattern, it is copied to the scanner's output, which is by default `stdout`. For this reason, a catch-all rule is usually specified to recognize unrecognized individual characters and report them as invalid to the user, e.g.:

```
.     {
          std::cerr << "Invalid character: " << yytext
               << std::endl;
          yyterminate();
      }
```

Methods and global values available for use in actions

- There are several methods, macros, and global values every action can make use of. Here are some of the most notable ones:

Global values

`char* yytext`	This is a null-terminated character string holding the text of the lexeme just matched. Read the Flex documentation before modifying `yytext`.
`int yyleng`	This holds the length of `yytext`, null-character excluded.
`int yylineno`	This holds the number of the source code line in which `yytext` was recognized. This must be enabled by enabling `%option yylineno`.

Macros and methods

`ECHO`	This is a macro that copies `yytext` to the scanner's output (`stdout` by default).
`REJECT`	This is a macro that indicates to the scanner to execute the action of the "second best" rule that matched `yytext` (or a prefix of `yytext`, in which case `yytext` and `yyleng` are reset appropriately). Any amount of processing may be done in the rule before invoking `REJECT`. This can be useful for running special processing on certain tokens in the source code while still allowing them to be handled by a different rule. We will see an example of this later.
`unput(c)`	Puts the character `c` back into the input stream to be the next character scanned. If using `unput()` to put a string into the input stream, the last character of the string must be unput first, and the first character must be unput last, since `unput()` always places its argument at the beginning of the input stream. Note that calling `unput()` destroys the contents of `yytext`.
`int input()`	This function reads and returns the next character in the input stream (thus removing it from the input stream). Note that if the scanner is compiled using C++, then `input()` is instead called `yyinput()`.
`yyterminate()`	This can be called to terminate the current call to the scanner function and return 0 to its caller (indicating "all done"). By default, this is called when the end of file is encountered.

- More methods and values available within actions are outlined in the Flex documentation under "actions" and "values available to the user."

The user code section of the Flex input file

- The final section of the flex input file is the user code section. In this section, the user may write any C/C++ code to be copied verbatim to the bottom of the generated scanner file. This section is optional.

- The user code section can be used to define subroutines that will be called in conjunction with the generated scanning function.

- Frequently, the user code section is used to include a `main()` function, so that the generated code file can be compiled directly into an executable. For example, here is a simple user code section that does this (we will talk more about `yylex()`, the generated scanning function, below):

```
int main() {
     return yylex();
}
```

A simple example of a Flex file

- Let's look at a simple example of a Flex scanner definition. Below is a Flex file that defines a scanner that simply counts characters, words, and lines in the source file:

```
%{
#include <iostream>

int num_chars = 0;
int num_words = 0;
int num_lines = 0;
%}

%option noyywrap

%%

\n        {
            num_lines++;
            num_chars++;
          }

[^ \t\n]+ {
            num_words++;
            num_chars += yyleng;
          }
```

```
.           { num_chars++; }

%%

int main() {
  yylex();
  std::cout << num_chars << " characters" << std::endl;
  std::cout << num_words << " words" << std::endl;
  std::cout << num_lines << " lines" << std::endl;
  return 0;
}
```

- The specification here should seem pretty straightforward. The call to `yylex()` is the key.

- Without specifying any additional Flex options, the `yylex()` function reads characters from `stdin` until it encounters an `EOF`. Every character it reads, it executes a transition in the DFA corresponding to the regular expressions specified in the Flex file.

- When no transition is possible after the next character, `yylex()` determines which rule was matched and executes that rule's action.

- Under the default set of Flex options, `yylex()` will continue to read characters until a selected action contains a `return` statement, which will result in `yylex()` returning. `yylex()` could then be called again, and it would resume scanning from the same point in the input stream where it left off the previous time.

- The prototype for `yylex()` is as follows:

```
int yylex();
```

- Any action may return an integer value.
    - Once we start to use the Bison parser generator, we will see that actions can return values corresponding to the syntactic categories of recognized lexemes.

    - Actions may also return values indicating error conditions.

- When an `EOF` is encountered, `yylex()` returns 0, under the default Flex options.

- Note also that in our scanner specification above we use `%option noyywrap` to simplify the implementation, assuming that we will scan only a single input file.
    - If we wanted to scan multiple input files, we would need to provide our own implementation of `yywrap()`, which would be called to move to the next input file once the end of the previous one was reached.

Compiling and running a Flex scanner

- Now that we have a basic scanner specification, let's see how we can compile and run it.

- Let's say our scanner specification lives in the file `counter.l`. We can generate C/C++ code implementing our scanner using the `flex` command in the terminal. For example, to generate a file `counter.cpp` from our specification we would run a command like this (the `-o` option specifies the output file):

  ```
  flex -o counter.cpp counter.l
  ```

- Since our scanner specification's user code section contains a `main()` function, `counter.cpp` can be compiled directly into an executable scanner:

  ```
  g++ counter.cpp -o counter
  ```

- If our scanner specification did not contain a `main()` function, we could write a simple `.cpp` file containing a `main()` function and then compile counter.cpp along with that file to generate an executable scanner, e.g.:

  ```
  g++ main.cpp counter.cpp -o counter
  ```

- By default, `yylex()` reads input from `stdin`, so to run our scanner on a source file, we can use input redirection. For example, we could run our counting scanner on its own specification like this:

  ```
  ./counter < counter.l
  ```

- Note that if the code in our scanner specification was in C instead of C++, we could output a `.c` file from the flex command and use `gcc` for compilation.

Using the REJECT macro

- To demonstrate the use of `REJECT`, let's augment our scanner specification above slightly. Let's say that, in addition to counting the number of characters, words, and lines, we want to count the number of times the word `int` appears in the source code.

- We could start by adding an additional declaration in the definitions section:

```
int num_ints = 0;
```

- We could then add a rule like the following:

```
int         {
              num_ints++;
              REJECT;
            }
```

- This rule increments the number of `int`s each time the word `int` is encountered. However, instead of also incrementing the number of words and the number of characters, it instead calls `REJECT`.

- The effect is that the action for the pattern `[^ \t\n]+` also executes, since this pattern also matches the word `int`. This means that each `int` encountered will still contribute to the number of characters and the number of words.

A scanner for a simple language

- Let's finish up by writing a specification for a scanner that recognizes a simple language. Our language will have the following syntax features:
    - The language allows only semicolon-terminated assignments, e.g.:

      ```
      a1 = b + 2;
      ```

    - The operations available (in addition to assignment) are: `+`, `-`, `*`, `/`.

    - Expressions may contain any combination of identifiers and numbers.
    - An identifier in the language consists of a lower-case letter and an optional single digit, e.g. `x2`.
    - A number can be an integer or a floating point number. All numbers are unsigned.
    - Expressions may be grouped in parentheses `()`.
    - All whitespace is ignored.

- Our goals for the scanner's behavior are as follows:
    - If all words in the source file are valid, output each word along with its syntactic category.
    - If all words in the source file are valid, also output a list of all of the unique identifiers encountered in the source file.
    - If any symbols are invalid, output each invalid symbol and the line number where it occurred.

- Given those goals, we can make a few design decisions up front:
    - We can create a new structure to hold a recognized word/syntactic category pair and store the list of all recognized words in a C++ STL `vector` of these structures.
    - We can store the collection of unique identifiers in a C++ STL `set`.

Definitions section

- Let's start writing our scanner specification, starting with the initial code declarations in the definitions section. First off, we can add some basic `#include` statements to make available the things we'll need to write the rest of the scanner:

```
%{
#include <iostream>
#include <vector>
#include <set>
%}
```

- Next, let's add declarations for some data structures we'll use throughout the scanner to keep track of the recognized words and their syntactic categories as well as the collection of unique identifiers. We'll start by defining a simple struct to hold a lexeme/syntactic category pair:

```
struct _word {
     std::string category;
     std::string lexeme;
};
```

- Given this struct, we can create instantiate an STL vector to hold all of the recognized words and an STL set to hold the collection of unique identifiers:

```
std::vector<struct _word> _words;
std::set<std::string> _ids;
```

- We'll also use a boolean value to keep track of whether we've seen an error so far. We'll use this once we reach the end of the source code to determine whether to return an error code or a success code:

```
bool _error = false;
```

- The last thing we'll add to the initial code declarations is a function to generate a `_word` struct from a lexeme and syntactic category and save that struct into the vector of recognized words:

```
void _save_lexeme(std::string lexeme, std::string category)
{
  struct _word word = { .lexeme = lexeme,
    .category = category };
  _words.push_back(word);
}
```

- To round out the definitions section, we can set some Flex options (now outside the `%{ %}` delimiters), setting `noyywrap`, so we don't have to implement `yywrap()` and setting `yylineno`, so we have the current line number available:

```
%option noyywrap
%option yylineno
```

Rules section

- Now, let's move on to the rules section (not forgetting the `%%` delimiter). We can start with a rule that ignores whitespace:

```
[ \t\n]*           /* Ignore whitespace. */
```

- Since this rule has no action, nothing will be done whenever a whitespace character is encountered.

- Next, let's write a rule to recognize identifiers. Remember, identifiers in our language have one lowercase letter followed by one optional digit. When we encounter an identifier, we'll save it to the list of recognized words then make sure it's contained in our set of unique identifiers:

```
[a-z][0-9]?   {
                _save_lexeme(yytext, "IDENTIFIER");
                _ids.insert(yytext);
              }
```

- Note that in this rule, we're using `yytext` to refer to the text of the identifier we just recognized.

- We can write a similar rule to recognize numbers, which, again, are unsigned and can be integers or floating-point numbers. This rule captures those constraints and saves the recognized number to our list of recognized words:

```
[0-9]+("."[0-9]+)?  { _save_lexeme(yytext, "NUMBER"); }
```

- We can include a number of literal-string regular expressions to recognize operators and punctuation:

```
"="     { _save_lexeme(yytext, "EQUALS"); }
"+"     { _save_lexeme(yytext, "PLUS"); }
"-"     { _save_lexeme(yytext, "MINUS"); }
"*"     { _save_lexeme(yytext, "TIMES"); }
"/"     { _save_lexeme(yytext, "DIVIDEDBY"); }

";"     { _save_lexeme(yytext, "SEMICOLON"); }
"("     { _save_lexeme(yytext, "LPAREN"); }
")"     { _save_lexeme(yytext, "RPAREN"); }
```

- Any other character other than the ones comprising the patterns above we is not valid in our simple language, so we can include a rule to match any other single character and report an error, remembering the fact that we saw an error:

```
.       {
          std::cerr << "Invalid symbol on line "
            << yylineno << ":" << yytext << std::endl;
          _error = true;
        }
```

- Finally, we can include a rule that returns 1 if an error has been encountered when the end of the file is reached or returns 0 if one has not:

```
<<EOF>> {
          if (_error) {
            return 1;
          }
          return 0;
        }
```

- This will allow us to determine from `yylex()`'s calling function whether or not a scan error was encountered.

- One thing you might notice when looking over these rules is that they may allow authors of programs in our simple language to write invalid identifier names.

- For example, the string `r2d2` may likely be an attempt to define an identifier, but it is invalid under the constraints our language places on identifiers. However, our scanner will not flag it as an invalid word but will scan it as two (concatenated) identifiers: `r2` and `d2`.

- While it would be possible to perform this kind of checking in the scanner, this type of error can eventually be easily detected by the parser, which will recognize that no valid statement in the language can have consecutive identifiers. Thus, we choose to keep the scanner's specification relatively simple and to defer these kinds of checks until the parser.

User code section

- We can wrap our scanner specification up with a straightforward user code section (not forgetting the `%%` delimiter between sections). Here, we'll simply define a `main()` function to make our generated scanner file executable:

```
int main() {
  ...
}
```

- Within `main()`, we'll call `yylex()`. Remember that `yylex()` will return 0 if the scan encounters no errors, in which case we'll want to print all of the recognized words and the set of unique identifiers. If `yylex()` returns 1, we won't do these things. This requirement sets up an `if` statement like this:

```
if (!yylex()) {
  ...
}
```

- Within this `if` statement, we can simply iterate through both the list of recognized words and the set of unique identifiers, printing them all out:

```
std::vector<struct _word>::iterator wit;
for (wit = _words.begin(); wit != _words.end(); ++wit) {
  std::cout << wit->lexeme << "\t\t" << wit->category
    << std::endl;
}

std::cout << std::endl ;
std::cout << "Unique identifiers:" << std::endl;
std::set<std::string>::iterator iit;
for (iit = _ids.begin(); iit != _ids.end(); ++iit) {
  std::cout << *iit << std::endl;
}
```

- That's it! Now we can compile our scanner like this (assuming it is in a file named `scanner.l`):

```
flex -o scanner.cpp scanner.l
g++ scanner.cpp -o scanner
```

- And if we run it on a program like the following, we'll see it successfully print all the lexemes in the program and their syntactic categories, plus the set of unique identifiers in the program:

```
a1 = 7;
b = a1 + 2;
c2 = (b - 2.75) * 3.1415;
```