Out[1]: (Show / Hide code)

`flex` Notes

CSE 4305 / CSE 5317 Compilers

2020 Fall Semester, Version 1.2, 2020 October 02

`flex` generates a C (C++ compatible) lexical analyzer based on a description of the token patterns to match and the code to execute when matches happen. It's a FOSS version of `lex`, developed separately from `lex` starting in about 1987 to get around the proprietary nature of `lex` and also to offer better performance.

An analyzer generated by `flex` scans the incoming characters using a Deterministic Finite Automaton created from all of the Regular Expression patterns given in the original description. Thus, in general a `flex` generated lexical analyzer achieves $\mathcal{O}(n)$ time complexity on matching. However, if one uses the `REJECT` capability of `flex` (a way to force the DFA to “try again” on a match that has already been made), greater than linear time complexity can result.

A Note on Versions

`flex` is remarkably stable — unlike a lot of crappy code that gets released nowadays.

Consequently, `flex` doesn't *need* to be released all that often as it does what it does quite well. Having said that, though, you should be using the most recent version `2.6.4`, released `2017-May-06`. That's over three years ago so there's no excuse not to have it.

I have received some questions from students re: warnings / errors from `flex` and in each case it's been because an older version of `flex` was being used. In one of the cases, a version from `2016-Mar-01` was being used, in another a version from *before* `2001-May-01` (?!?).

Everything in these *Notes* has been tested using `flex 2.6.4` in two development environments, with different processor architectures and kernel versions,

```
$ cat /etc/os-release
VERSION="18.04.5 LTS (Bionic Beaver)"
$ uname -a
Linux svr-test-01 4.15.0-118-generic #119-Ubuntu SMP Tue Sep 8 12:30:0
1 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
$ flex --version
flex 2.6.4
$
```

and

```
$ cat /etc/os-release
PRETTY_NAME="Raspbian GNU/Linux 10 (buster)"
$ uname -a
Linux icywits-01 5.4.51-v7+ #1333 SMP Mon Aug 10 16:45:19 BST 2020 arm
v7l GNU/Linux
$ flex --version
flex 2.6.4
$
```

That's no guarantee that an error hasn't crept in, though, so report any difficulties you might encounter, but please, please, please try it with `flex 2.6.4` before reporting.

Process

Generally,

- Put all of your `flex` input in a file named `<whatever>.l`, as described below.
- Generate the lexical analyzer by executing `flex <whatever>.l`.
- Include the resulting `lex.yy.c` file (the generated lexical analyzer) in the build of your entire program.

Simple, eh?

Format

The input file for `flex` should have a `.l` suffix. `flex` processes it line-by-line, collects the information it needs and generates the `lex.yy.c` lexical analyzer.

`flex`'s parsing of its input file is quite primitive so it's easy to make a typographical error that destroys `flex`'s understanding of what you're trying to do.

Be careful.

Some items have to be on lines by themselves, some items have to start at the left margin (no whitespace allowed in front), some items *must* be indented, etc., etc. We try to highlight these requirements in the following explanations.

One significant issue is `flex`'s processing of comments. It's safest to stick with C's `/* ... */` style of comments. You might get away with `//` style comments in various places, but it could take fiddling to figure out how to get `flex` to accept it. From the `flex` documentation,

> If you want to follow a simple rule, then always begin a comment on a new line, with one or more whitespace characters before the initial `/*`. This rule will work anywhere in the input file.

The `minimal` examples all work properly, so you can check their structure, formatting, etc. to see what's allowed by `flex`.

Input File Structure

A `.l` file is split into three sections and has this general format,

```
/* Definitions */

%%

/* Rules */

%%

/* User Code */
```

The `%%` marks have to be the only characters on their lines and at the left margin (that is, have no whitespace in front of them).

Definitions Section

The *Definitions Section* includes the specification of “definitions” (names for reusable units of regular expressions) and `%option` lines (options that control the functioning of `flex`). Also included is any C code the user wants included at the front of the generated lexical analyzer file. This C code is normally items that will be used by the action routines associated with the patterns given in the *Rules Section*.

Definitions

`flex` 's regular expression (RE) syntax (see the *Appendix: `flex` Regular Expressions* for more details on this syntax) does not support constructs such as `\d` (decimal digit ≡ `[0-9]`), `\s` (whitespace character ≡ `[ \f\n\r\t\v]`), `\w` (“word” character ≡ `[_a-zA-Z0-9]`), etc.

What `flex` *does* support is the creation of *Defined Names*. The definitions are put in the *Definitions Section* of the `.l` file. Here are some example definitions,

```
DIGIT          [0-9]
WHITE_SPACE    [ \f\n\r\t\v]
WORD_CHAR      [_a-zA-Z0-9]
HEX_DIGIT      [a-fA-F0-9]
```

The name part of the definition can have letters, digits, hyphens (`-`), and underscores (`_`), but it cannot start with a digit. It should be at the left margin (that is, no whitespace in front of the name).

In the *Rules Section* a defined name can be used in a pattern's RE by enclosing the name in braces `{ ... }` . So, for example, we could write the following,

```
/* Definitions Section */

DIGIT      [0-9]

%%

/ * Rules Section */

{DIGIT}+   {
    printf( "Saw integer %d.\n", atoi( yytext ) );
}
```

The `{ ... }` substitution can be used anywhere an RE can be used. The corresponding definition of the name (enclosed in parentheses `( ... )`) is substituted. The above example is equivalent to,

```
([0-9])+   {
    printf( "Saw integer %d.\n", atoi( yytext ) );
}
```

Note how the `{DIGIT}` name was replaced with `([0-9])` . The parentheses are automatically wrapped around the definition before doing the substitution to preserve the meaning of the definition no matter where it gets inserted.

Options

`%option` lines are used to inform `flex` as to how this `.l` file should be processed. Usually these options could have been specified on the `flex` command line, but putting them in the `.l` file is tidier.

There are zillions of `flex` options that you can read about in the `flex` documentation. Many are obscure and are relevant only in very special circumstances. We use a few in our `minimal` examples, so we'll explain them here.

- Header file

```
%option header-file="lex.yy.h"
```

`flex` can generate a header file that exposes the external interface through which one can interact with the generated lexical analyzer. This is not important in the `minimal` examples as we put everything in the `.l` file but it will be important later when we generate `bison` parsers. This option says to generate that `.h` file with the given name.

> You might wonder why `flex` 's generated lexical analyzer is named `lex.yy.c` by default (and why the header file is named `lex.yy.h`). The best answer is “for historical reasons”. Ha!

- Not an interactive lexer

```
%option never-interactive
```

Our `minimal` examples are not interactive—that is, input is not coming from an interactive session with a user. By setting this option, we let `flex` know this and `flex` can therefore generate a somewhat faster analyzer. (It has to do with how `flex` deals with lookahead.)

- A bunch of stuff we *don't* want

  ```
  %option nodefault
  %option noinput
  %option nounistd
  %option nounput
  %option noyywrap
  ```

By default, `flex` includes a bunch of stuff in the generated lexical analyzer. If we're not using it, we don't want it. Therefore we turn *off* a bunch of items.

`nodefault` means `flex` should not generate the *default* rule, which is a rule that would match any character that isn't otherwise consumed by the given patterns. We don't need this default rule in the `minimal` examples because we have written an explicit mechanism to catch *illegal characters*. Even going beyond simple examples, you shouldn't let `flex` catch unexpected characters for you; your rules should match *anything* that can occur in the character stream, even if it's illegal. That help in generating meaningful error messages.

`noinput` and `nounput` mean `flex` should not generate the `input()` and `yyunput()` functions. We don't use them and if they are generated we're going to get `defined but not used` messages from `gcc` . (There are obscure cases where these two functions are useful, but we're not going to encounter them in an introductory example.)

`nounistd` means don't try to include the `<unistd.h>` header file. We don't need it and don't want it.

`noyywrap` means when we come to the end of the given input, we stop. There's no more input to process. (Complex scanning cases might involve multiple input files, but that's not going to come up in an introductory example.)

- Reports and warnings

  ```
  %option perf-report perf-report
  %option verbose verbose
  %option warn
  ```

`flex` does a bunch of processing on your input. There are a number of statistics that can be displayed. It's useful to see them, especially the comments on anything that might be affecting the performance of generated lexical analyzer. The `perf-report` and `verbose` options are doubled because that causes `flex` to give more detailed reports.

`flex` can also detect certain issues with your input that are not really *errors* per se but would probably lead to unintended consequences. Specifying the `warn` option tells `flex` to warn you if these issues arise. (Why this option isn't *on* by default is beyond my comprehension.)

- Line numbering

  ```
  %option yylineno
  ```

`flex` will count line numbers for you, but the capability is not turned on by default. This option enables line counting and makes the current line number available through the global variable `yylineno`.

> While `flex` will *count* lines when this option is enabled, it doesn't *initialize* `yylineno`, so don't forget to set `yylineno` to `1` in your `main` routine.

Included C Code

Sometimes you want to access certain C items inside the action routines associated with the patterns in the *Rules Section*. `flex` defines a bunch of useful items for you automatically. Others you will have to define yourself. You can put your C definitions in the *Definitions Section* inside `%{ ... %}` markers.

The format is,

```
%{
    /* Your C items */
%}
```

The `%{` and `%}` have to each be on their own lines and must be at the left margin (that is, no whitespace before them).

In the `minimal1` example, we include `<stdio.h>` (so we can do an `fprintf` in an action routine), define a token ID enumeration, define a spot to keep token values (`yylval`), and define a spot to keep the current column number (`yycolno`). Our included C code therefore looks like this,

```
%{
#include <stdio.h>

enum {
  tok_ID = 256,
  tok_INT_LIT,
};

union {
  int intval;
} yylval;

int yycolno;
%}
```

All of these items are threafter accessible in the *Rules Section* action routines.

Rules Section

The *Rules Section* includes the Regular Expressions (REs) that define the patterns of the token categories and the action routines that are associated with those patterns.

Every rule must be of the form,

```
regular-expression {
    action-routine
}
```

The *regular expression* must start at the left margin. No leading whitespace is allowed. The `{` starting the *action routine* must be on the same line as the regular expression.

For example, the following entry,

```
[0-9]+   {
    printf( "Saw digit string \"%s\", which has %d digit%s.",
      yytext, yyleng, yyleng == 1 ? "" : "s" );
}
```

has `[0-9]+` as its pattern. This regular expression matchs one-or-more (`+`) decimal digits (`[0-9]`). The action routine begins with the left brace `{` , *which must be on the same line as the regular expression*. This action routine merely prints a message showing the digit sequence that was matched as well as how many characters long it is.

`yytext` is a `char *` that points to a `NUL` -terminated string of the characters that were matched by the regular expression. `yyleng` is an `int` giving the length of the `yytext` string. `yyleng` is the number of characters matched; it does *not* include the `'\0'` (`NUL`) character that terminates the string.

Both `yytext` and `yyleng` are automatically set by the lexical analyzer before the action routine code is executed.

Matching Patterns

When the lexical analyzer runs, it tries to match the input characters against the patterns represented by the rules' REs. If more than one pattern would match the input characters, the lexical analyzer prefers the pattern that matches the *most* characters.

It's OK to have two REs that match as long as they match different numbers of characters. After all, that's how one distinguishes a pattern that's a prefix of another pattern. An example is `INTEGER_LITERAL` and `DOUBLE_LITERAL` . The string `123.456` begins with what looks like an `INTEGER_LITERAL` , `123` . However, the succeeding `.456` will also be matched by the `DOUBLE_LITERAL` pattern resulting in a longer match. Therefore `DOUBLE_LITERAL` 's pattern will be preferred. (This is know as the *maximal munch* principle—consume as many characters as possible.)

On the other hand, if multiple patterns match the *same* number of characters, the lexical analyzer prefers the pattern that occurs earliest in the *Rules Section*.

Do *not* write REs that would match the same number of characters. If you do, the action routine that gets executed will be (somewhat) arbitrarily selected, as it's based on the order of the rules. Rethink your patterns so this case does not occur.

Writing Action Routines

OK, so that's the format of a rule: a pattern RE and an action routine. It's also pretty obvious what it means to write the RE of a pattern (hey, everyone's used REs before, right?). But what about the *action routine*? What gets done in that? What *should* get done?

An action routine gets executed when its corresponding pattern RE matches the input. When the action routine is entered, the string of characters matched by the RE is available using the `char * yytext`. The length of the match (not including the trailing `NUL` terminator) is available using the `int yyleng`.

> Do *not* change `yytext` or the characters it points to or the value of `yyleng` in the action routine. There are obscure reasons one might want to do so, but they do not come up in these introductory examples.

You can do whatever processing you want in the action routine, but in general there are two cases: ① A token is recognized and must be returned and ② The characters are to be ignored.

① *Token Recognition*. In the case when a token is recognized, you must `return` a value from the action routine indicating which kind of token was seen. The `tok_INT_LIT` and `tok_ID` action routines in the `minimal1` example are of this kind.

> See *Appendix: Character Tokens* for more information on single-character tokens.

If the token kind being returned is simple, the `return` value itself identfies it. For example, recognizing the token kind *left parenthesis* is simple since all left parentheses are the same. On the other hand, recognizing the token kind *integer* is *not* simple as integers can have different values from one another.

In the case of a complex token kind such as *integer*, not only must the token kind be returned, but an indication of the *value* of this specific integer token must be made available. This is done through the `yylval` variable. In the action routine for the token kind `tok_INT_LIT`, the characters that were recognized as an integer literal (in `yytext`) are converted to an integer value (using `atoi( yytext )`) and that value is copied to the integer field `intval` of the `yylval` union.

Similar processing should be taken in the action routines of other complex token kinds: convert the characters in `yytext` to a suitable value and make that value available through the corresponding field of the `yylval` union. The definition of the `yylval` union should be updated to include whatever alternate fields are required to hold the values corresponding to those token kinds.

> The integer value of the token kind returned from the action routine must not be equal to `0`. Token kind `0` is used to indicate that the end of the token stream has been reached and there are no more tokens to retrieve.

② *Ignoring Characters*. Sometimes characters are recognized but do not form part of any token. A typical example of this would be whitespace. Most programming languages are *free form* in that the specific indentation of the code does not matter. In such languages, the only meaningful whitespace is that inside string or character literals or that used to separate otherwise ambiguous tokens.

Even though the whitespace is meaningless, the characters that comprise it must be consumed from the input character stream. After matching, though, there is no token to return, so no `return` statement should exist in the action routine. Just drop off the end of the action routine. At that point the lexical analyzer will begin trying to match the *next* set of characters.

Aside from whitespace, another reason to ignore a character is that it is *illegal*, that is, it doesn't occur in the construction of a legal token. In this case, the character should be consumed but there is no token to return.

The `minimal` examples show how to consume whitespace as well as how to report illegal characters. Study their processing of these two cases.

Another item that should be carefully considered in every action routine is the tracking of the current column number. While the generated lexical analyzer automatically counts the *line* number (in `yylineno`), the *column* position must be tracked manually.

This is done in a simple way in `minimal1` (using `yycolno`) and in a more sophisticated way in `minimal2` (using `yycolnoBegin` and `yycolnoEnd`). Study those two examples carefully so that you can make more useful error messages.

User Code Section

The *User Code Section* includes any C code that you want to be put in the lexical analyzer file `lex.yy.c`. It's useful for the body of routines that get used by action routines. The `minimal` examples are so simple we even put the `main` routine here.

Anything in the *User Code Section* just gets copied verbatim from the `.l` file to the `lex.yy.c` file.

> There is an important difference between ① code in the *User Code Section* and ② “Included C Code” in the *Definitions Section*.
>
> In the ① case, the C code is inserted in the `lex.yy.x` file *after* the code of the action routines. In the ② case, the C code is inserted in the `lex.yy.x` file *before* the code of the action routines. We therefore put declarations in the “Included C Code” in the *Definitions Section*. Actual routines are put in the *User Code Section*.

Usage Model

Given you have a valid set of patterns and action routines, how do you actually perform the scanning? The usage model is to ① set up the character stream, ② scan the tokens, and ③ tear down the character stream.

Set up the Character Stream

The lexical analyzer needs an input stream of characters. In the earliest `minimal` example `minimal1`, this stream comes from a character literal. This is particularly easy to set up. We use the `yy_scan_string()` routine to set the string that is to be scanned. Trivial!

In `minimal2` we want the lexical analyzer to take its input character stream from a file. In this case, we have to first open the file for input (using `fopen( fileName, "r" )`) and then use the `yyrestart()` routine on the obtained `FILE *`. Again, trivial!

Scan the Tokens

The lexical analyzer generated by `flex` is used by calling `yylex()`. This routine takes no arguments and returns the number of the token category that is next recognized. When there are no more tokens to return, `yylex()` returns `0`.

The easiest thing to do (at least in the introductory examples) is to use a `while` statement to process each token. Like this,

```
// Get the tokens one-by-one
int tok;
while ( ( tok = yylex() ) ) {
  switch ( tok ) {
    case Some-Token-ID:
      // Process a token of category Some-Token-ID
      break;

    case Some-Other-Token-ID:
      // Process a token of category Some-Other-Token-ID
      break;

    ...
    // Put the rest of the token category cases here
    ...

    default:
      // Here's the "unrecognized token category" case.
      break;
  }
}

// All done!  No tokens left.
```

The identifiers *`Some-Token-ID`* and *`Some-Other-Token-ID`* should be the token category names from the `enum` you defined, as described in *Definitions Section* > *Included C Code* above. Your `switch` statement here should have a `case` for each possible token kind as well as a `default` section just in case an error occurs. (Don't forget the `break` statement at the end of each block of processing!)

Tear Down the Character Stream

Once there are no more tokens, you have to tear down the character stream you set up. In the `maximal1` case, there's nothing to do; the characters were coming from a string literal. In the `maximal2` case, the characters were coming from a file. That file should be closed using `fclose()` .

Appendix: `flex` Regular Expressions

In the *Rules Section* `flex` uses Regular Expressions (REs) to describe the patterns of each rule. As is usual with a tool that uses REs, `flex` has its own peculiarities of what is and isn't an RE and its own syntax for expressing them. Fortunately, `flex` 's notions are fairly standard. We present an introductory summary of `flex` 's syntax for REs here, with some description. More complex RE structures (if you need them) can be found in `flex` 's documentation.

In the following summary, `w` , `x` , `y` , and `z` stand for *any* characters. `R` and `S` stand for *any* RE. `n` and `m` stand for any non-negative integers.

> Remember that in the *Rules Section* the a patterns's RE must start in first column (no whitespace before it) and the `{` of its action routine must be one the same line as this RE and separated from the RE by whitespace.

Characters

`.`

Matches *any* character *except* newline.

`w`

Matches the character `w` . (Read *Special Cases* below.)

`\w`

If `w` is `a` , `b` , `f` , `n` , `r` , `t` , or `v` , `\w` matches the C interpretation of the escape sequence `\w` . For example, `\n` matches newline, `\t` matches the tab character. If `w` is *not* one of those seven special characters, `\w` matches the explicit character `w` . For example, `\\` matches a `\` character.

Unlike some RE implementations, `flex` gives no special meaning to sequences such as `\d` , `\s` , `\w` , etc. `flex` uses *Defined Names* for this purpose.

`\0`

The `NUL` character.

`\123`

The character with *octal* value `123` , where `123` can be any octal value from `000` to `377` .

`\x12`

The character with *hexadecimal* value `12` , where `12` can be any hexadecimal value from `00` to `FF` .

`"..."`

The characters within the quotation marks `"` are matched *literally*. No character inside the marks has any special meaning. For example, `"[a-z]+"` means to literally match the six characters `[` , `a` , `-` , `z` , `]` , and `+` . No special interpretation is given to any character inside the `"` marks.

Character Classes

In general, characters classes are used to match any one of a given set of characters. The character class syntax `[ ... ]` is used to make it easier to specify the sets. Within the brackets, no character has any special meaning unless explicitly mentioned below. If an explicit `]` character is desired inside a class, it must be the *first* character listed.

`[wxyz]`

Match either of the `w` , `x` , `y` , or `z` characters. You can have as many individual characters as you want inside the `[ ]` marks and any *one* of them will be matched. Inside a character class, the `\w` construct may be used and will be properly interpreted. For example, `[ \t]` is a character class that matches the space or tab character.

`[^wxyz]`

Using a `^` as the *first* character inside a character class *negates* its meaning. In this example, this means that any character *except* `w` , `x` , `y` , or `z` will be matched. This interpretation means a newline `\n` will be matched. If you don't want a newline matched, be certain to explicitly exclude it.

If an explicit `^` character is desired inside a class, it must *not* be the first character listed.

`[0-9]`

Using a `-` inside a character class indicates a *range* of characters is to be matched. In this example any one of the characters `0` , `1` , `2` , `3` , `4` , `5` , `6` , `7` , `8` , or `9` will be matched. `flex` will let you use *any* characters in *any* order when expressing a range. Ensure that the characters you pick make sense in the order they are listed or some very unexpected results are possible.

If an explicit `-` character is desired inside a class, it must be the *first* or *last* character listed.

`[A-Z]{-}[AEIOU]`

`{-}` between two character classes performs *set difference*. In this case, the first character class matches all uppercase letters. The second character class matches all uppercase vowels. By performing this set difference, the resulting character set is all uppercase consonants.

RE Modifiers

`R*`

Match zero or more of the RE `R` .

`R+`

Match one or more of the RE `R` .

`R?`

Match zero or one of the RE `R` .

`R{n}`

Match `n` occurences of the RE `R` .

`R{n,m}`

Match `n` through `m` occurences of the RE `R` .

`R|S`

Match either the RE `R` or the RE `S` .

`( ... )`

Group the contents into a single RE. The grouping construct may be used affect how much the modifiers affect.

Match Modifiers

`^`

Match the beginning of a line. This must be the *first* character of an RE.

`$`

Match the end of a line. This must be the *last* character of an RE.

`(?i:R)`

Match the RE `R` in a case-insensitive way.

Defined Names

As described in *Definitions Section* > *Definitions* above, `flex` supports the creation of *Defined Names*. The definition of such a name goes in the *Definitions Section*. Such a name is used in a pattern's RE by enclosing the name in braces `{ ... }` . For example, the definition,

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

may be used in a rule thusly,

```
{DIGIT}*({DIGIT}\.|\.{DIGIT}){DIGIT}*    {
    printf( "Saw float value %f, from characters \"%s\".\n",
        atof( yytext ), yytext );
}
```

Special Cases

The quotation character `"` should be written in a character class, that is, as `["]`, unless you are trying to write an explict string of characters to match `" ... "`. This always comes up when students try to write the RE for a string literal. It's natural to want to write `"[^"\n]*"` as the RE for the pattern. Even though in the abstract that seems as if it *ought* to work, it fails in a fairly obscure way, matching only the explicit characters `[`, `^`, etc. Instead, write something like `["][^"\n]*["]`. The `"` characters are inside the character class notation and will not have their special meaning.

> You can also write `\"`, which also matches a literal `"` character just as `["]` does.

Another special case is whitespace itself. Sometimes a pattern may require whitespace characters to be recognized either at the beginning of the RE or at its end. Either way, trying to match whitespace there causes problems as `flex` forbids whitespace at the beginning of the pattern and uses whitespace at the end of the pattern to separate the pattern from the action routine. Instead, escape it (i.e., write `\ ` instead of just ` `) or use a character class (i.e., `[ ]`) to hold the whitespace.

Appendix: Character Tokens

Most programming languages have a variety of, e.g., punctuation marks and operators that are single characters. For example, C has `+`, `-`, `*`, and `/` for the four fundamental arithmetic operations. Normally, we would define a separate token ID for each of these, perhaps `tok_PLUS`, `tok_MINUS`, `tok_MULTIPLY`, and `tok_DIVIDE`. While possible, this is pretty tedious. (And, as you'll see when we begin with `bison`, it obfuscates the grammar rules.)

`flex` provides a mechanism for avoiding this complexity. The `yylex()` routine returns an integer value indicating the ID of the scanned token (or `0` at the end of the token stream). The integer values `1` through `255` are *reserved* so that they can be used to indicate a single `ASCII` character. That is, the token ID value `40` is used to represent the *left parenthesis* token because `40` is the `ASCII` value of the `(` character. For example, the following rule is quite convenient,

```
[(]    { return '('; }
```

This rule recognizes a left parenthesis and returns `40` as the token ID. We get away with this in C because a character literal, `'('` here, is an `int` with the value of the `ASCII` code of the character. Pretty neat, eh?

We can extend this concept to recognize *all* of the single character tokens of our language by writing something like this,

```
[-+*/()[\]{}<>]    { return yytext[0]; }
```

This pattern uses a character class to match one instance of any of the characters given in the class. Here we've included not only the four fundamental arithmetic operators but also the fours sets of bracketing characters.

> The `-` character has to be first in the character class so that it's recognized as a literal `-` character and not interpreted as indicating a range of characters. Similarly, we have to write `\]` to ensure that it does not close the character class.

In the action routine, the `ASCII` value of the matched character is returned as the token ID. We retrieve it as the first character in `yytext`. Really, really neat, eh?

Appendix: Distinguishing Keywords from IDs

Most programming languages have the concept of an *identifier*, a name created by the user representing some object in the program. These languages usually also have a number of *keywords* (also known as *reserved words*), names that have the same form as identifiers but that are reserved by the language itself.

For example, C has (since C11) these keywords,

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

The form of these keywords is the same as identifiers so there will be a clash between a rule to recognize one of the keywords and the general rule to recognize an identifier. For example, in the *Definitions Section*, we could write,

```
   //---- DEFINITIONS --------
%{
enum {
  ...
  tok_AUTO,
  tok_BREAK,
  tok_CASE,
  ...
  tok_INLINE,
  ...
  tok__STATIC_ASSERT,
  tok__THREAD_LOCAL,
  ...
};

%}
```

We would then write a *Rules Section* such as this,

```
%% //---- RULES -------------

auto                       { return tok_AUTO; }
break                      { return tok_BREAK; }
case                       { return tok_CASE; }
...
inline                     { return tok_INLINE; }
...
_Static_assert             { return tok__STATIC_ASSERT; }
_Thread_local              { return tok__THREAD_LOCAL; }

[a-zA-Z_][a-zA-Z_0-9]*     { return tok_ID; }
```

(The `...` spots indicate where lines have been omitted.)

If the character stream has for example as its next letters `inline`, two patterns will match: the one for `tok_INLINE` and the one for `tok_ID`. Because the `tok_INLINE` rule occurs earlier than the rule for `tok_ID`, its action routine will be executed and the characters will be scanned as the reserved word `tok_INLINE` instead of the general `tok_ID`.

As mentioned above, having two patterns that match the same characters is bad style. We therefore would rather have a way to process keywords and identifiers in a unified way. In particular, in the *Definitions Section*, where we created `enum` constants for the token numbers for the keywords, we put a forward declaration for the `isKeyword()` function, which we define in the *User Code Section*.

```
   //---- DEFINITIONS --------
%{
enum {
  ...
  tok_AUTO,
  tok_BREAK,
  tok_CASE,
  ...
  tok_INLINE,
  ...
  tok__STATIC_ASSERT,
  tok__THREAD_LOCAL,
  ...
};

extern int isKeyword( const char *str );
%}
```

In the *Rules Section*, we have only a single rule that is used to handle both keywords and IDs. Notice that in the action routine for this rule, we check whether the matched text (in `yytext`) is a keyword using the `isKeyword()` function. If it is, we return the indicated token number. If the matched text is not a keyword, we return `tok_ID` instead.

```
%% //---- RULES --------------

[a-zA-Z_][a-zA-Z_0-9]*    {
  int kw = isKeyword( yytext );

  if ( kw != 0 ) {
    // It's in the keyword list so return its token number.
    return kw;
  } else {
    // It's not in the keyword list so return tok_ID.
    return tok_ID;
  }
}
```

In the *User Code Section*, we define a structure type `Keyword` and an array of known keywords `keywords[]`. The `isKeyword()` function takes a pointer to the matched characters as input and looks through the `keywords[]` array to see if they line up with a known keyword. If so, the corresponding token number is returned, otherwise `0`, indicating that the matched characters are *not* a keyword.

```
%% //---- USER CODE ----------

typedef struct {
  char *kw;
  int   tok;
} Keyword;

Keyword keywords[] = {
  { "auto",             tok_AUTO; },
  { "break",            tok_BREAK; },
  { "case",             tok_CASE; },
  ...
  { "inline",           tok_INLINE },
  ...
  { "_Static_assert",   tok__STATIC_ASSERT; },
  { "_Thread_local",    tok__THREAD_LOCAL; },
};

#define NUM_KEYWORDS ( sizeof(keywords) / sizeof(Keyword) )

int isKeyword( const char *str )
{
  for ( size_t i=0; i < NUM_KEYWORDS; i++ ) {
    if ( strcmp( str, keywords[i].kw ) == 0 ) {
      return keywords[i].tok;
    }
  }

  return 0;
}
```

By using this technique to separate keywords from identifiers, we simplify the *Rules Section* and avoid ambiguous matching. It's tidier.