EECS665

Compiler Construction

Drew Davidson
Ruturaj Vaidya

Lecture: LEEP2 G415
MWF 3:00-3:50

Lab: Eaton 1005B

ANOUNCEMENTS | LAB | SCHEDULE | MATERIALS | ASSIGNMENTS

Project 2

Due on 9/24 @ 11:59 PM

Accepted for 90% credit or 1 late day on 9/25 @ 11:59 PM

Accepted for 80% credit or 2 late days on 9/26 @ 11:59 PM

Accepted for 70% credit or 3 late days on 9/27 @ 11:59 PM

Updates

1. A hanging reference to symbols.hh has been removed and replaced with a reference to the correct file, grammar.hh
2. The link to eof.txt has been fixed

Overview

For this assignment you will use Flex to write a scanner for our language, called , a small subset of the C++ language. Features of that are relevant to this assignment are described below. You

will also write a main program (P2.cpp) to test your scanner. You will be graded both on the correctness of your scanner and on how thoroughly your main program tests the scanner.

Specifications

- Getting started
- Flex
- The Array Language
- What the Scanner Should Do
- Errors and Warnings
- The Main Program
- Testing
- Working in Pairs

Getting Started

Skeleton files on which you should build are in: p2.tgz

The files are:

- grammar.hh: Token definitions (this file will eventually be generated by the parser generator). *Do not change this file.*
- P2.cpp: Contains the entrypoint "driver" that tests the scanner. *You do not need to change this file, but you are free to do so.*
- lilc_compiler.cpp: Contains the main coordinating class for the compiler, and the body of the method to test the compiler. *You do not need to change this file, but you are free to do so.*
- lilc_compiler.hpp: Header declaration for the lilc_compiler class. *You do not need to change this file, but you are free to do so.*
- lilc_scanner.hpp: Header declaration for the lilc compiler class. *You do not need to change this file, but you are free to do so.*
- lilc_lexer.l: Flex definition for the scanner class. ***You must change this file***
- Makefile: A Makefile that uses Flex to create a scanner, and also makes P2.cpp. ***You must change this file.***

Flex

Use the on-line Flex reference manual, and/or the on-line Flex notes for information about writing a Flex specification.

If you work on an EECS Linux machine, you should have no problem running Flex. You will not be able to work on the Department's Windows machines.

The Language

This section defines the lexical level of the CFlat language. At this level, we have the following language issues:

Tokens

The tokens of the Array language are defined as follows:

- Any of the following reserved words (remember that you will need to give the Flex patterns for reserved words *before* the pattern for identifier):

```
bool   int    void   true   false   struct
cin    cout   if     else   while   return
```

- Any identifier (a sequence of one or more letters and/or digits, and/or underscores, starting with a letter or underscore, excluding reserved words).
- Any integer literal (a sequence of one or more digits).
- Any string literal (a sequence of zero or more *string* characters surrounded by double quotes). A *string* character is either
  - an escaped character: a backslash followed by any one of the following six characters:
    1. n
    2. t
    3. a single quote
    4. a double quote
    5. a question mark
    6. another backslash

    or
  - a single character other than newline or double quote or backslash.

  Examples of legal string literals:

```
""
"&!88"
"use \n to denote a newline character"
"include a quote like this \" and a backslash like this \\"
```

  Examples of things that are ***not*** legal string literals:

```
"unterminated
"also unterminated \"
```

```
"backslash followed by space: \ is not allowed"
"bad escaped character: \a AND not terminated
```

- Any of the following one- or two-character symbols:

```
{       }       (       )       ;
,       .       <<      >>      ++
--      +       -       *       /
!       &&      ||      ==      !=
<       >       <=      >=      =
```

Token "names" (i.e., values to be returned by the scanner) are defined in the file grammar.hh, and you can see them in action in lilc_compiler.cpp, For example, the name for the token to be returned when an integer literal is recognized is INTLITERAL and the token to be returned when the reserved word int is recognized is INT.

Note that code telling Flex to return the special EOF token on end-of-file has already been included in the file lilc_lexer.l -- you don't have to include a specification for that token. Note also that the READ token is for the 2-character symbol >> and the WRITE token is for the 2-character symbol <<

If you are not sure which token name matches which token, ask!

Comments

Text starting with a double slash (//) or a sharp sign (#) up to the end of the line is a comment (except of course if those characters are inside a string literal). For example:

```
// this is a comment
# and so is this
```

The scanner should recognize and ignore comments (but there is no COMMENT token).

Whitespace

Spaces, tabs, and newline characters are whitespace. Whitespace separates tokens and changes the character counter, but should otherwise be ignored (except inside a string literal).

Illegal Characters

Any character that is not whitespace and is not part of a token or comment is illegal.

Length Limits

You may not assume any limits on the lengths of identifiers, string literals, integer literals, comments, etc.

What the Scanner Should Do

The main job of the scanner is to identify and return the next token. The value to be returned includes:

- The token "name" (e.g., INTLITERAL). Token names are defined in the file grammar.hh.
- The line number in the input file on which the token starts.
- The number of the character on that line at which the token starts.
- For identifiers, integer literals, and string literals: the actual value (a String, an int, or a String, respectively). For a string literal, the value should include the double quotes that surround the string, as well as any backslashes used inside the string as part of an "escaped" character.

Your scanner will return this information by creating a new Symbol object in the action associated with each regular expression that defines a token (the Symbol type is defined externally; you don't need to look at that definition). A Symbol includes a field of type int for the token name, and a field of type Object (named value), which will be used for the line and character numbers and for the token value (for identifiers and literals). See lilc_lexer.l for examples of how to call the SynSymbol (subclass) constructors. See lilc_compiler.cpp for code that accesses the fields of a SynSymbol.

In your compiler, the SynSymbol will actually be subclassed, with a value() function to get the actual value; that type is defined in lilc_lexer.l. Every TokenVal includes a linenum field, and a charnum field (line and character numbers start counting from 1, not 0). Subtypes of TokenVal with more fields will be used for the values associated with identifier, integer literal, and string literal tokens. These subtypes, IntLitTokenVal, IdLitTokenVal, and StrLitTokenVal are also defined in lilc.l.

Line counting is done by the scanner generated by Flex (the variable yylineno holds the current line number, counting from 0), but you will have to include code to keep track of the current character number on that line. The code in lilc.l does this for the patterns that it defines, and you should be able to figure out how to do the same thing for the new patterns that you add.

The Flex scanner also provides a method yytext that returns the actual text that matches a regular expression. You will find it useful to use this method in the actions you write in your Flex specification.

Note that, for the integer literal token, you will need to convert a string (the value scanned) to an int (the value to be returned). You should use code like the following:

```
double d = std::stod(yytext); // convert String to double
// INSERT CODE HERE TO CHECK FOR BAD VALUE -- SEE ERRORS AND WARNINC
int k =  atoi(yytext)    // convert to int
```

Errors and Warnings

The scanner should handle the following errors as indicated:

Illegal characters

Issue the error message: illegal character ignored: ch (where ch is the illegal character) and ignore the character.

Unterminated string literals

A string literal is considered to be unterminated if there is a newline or end-of-file before the closing quote. Issue the error message: unterminated string literal ignored and ignore the unterminated string literal (start looking for the next token after the newline).

Bad string literals

A string literal is "bad" if it includes a bad "escaped" character; i.e., a backslash followed by something other than an n, a t, a single quote, a double quote, or another backslash. Issue the error message: string literal with bad escaped character ignored and ignore the string literal (start looking for the next token after the closing quote). If the string literal has a bad escaped character *and* is unterminated, issue the error message unterminated string literal with bad escaped character ignored, and ignore the bad string literal (start looking for the next token after the newline). Note that a string literal that has a newline immediately after a backslash should be treated as having a bad escaped character and being unterminated. For example, given:

```
"very bad string \
abc
```

the scanner should report an unterminated string literal with a bad escaped character on line 1, and an identifier on line 2.

Bad integer literals **(integer literals larger than MAX_INT)**

Issue the warning message: integer literal too large; using max value and return MAX_INT as the value for that token.

For unterminated string literals, bad string literals, and bad integer literals, the line and column numbers used in the error message should correspond to the position of the *first* character in the string/integer literal.

Use the fatal and warn methods of the ErrMsg class to print error and warning messages. Be sure to use *exactly* the wording given above for each message so that the output of your scanner will match the output that we expect when we test your code.

The Main Program

In addition to specifying a scanner, you should extend the main program in P2.cpp. The program opens a file called allTokens.in for reading; then the program loops, calling the scanner's next_token method until the special end-of-file token is returned. For each token, it writes the corresponding lexeme to a file called allTokens.out. You can use diff to compare the input and output files (diff allTokens.in allTokens.out). If they differ, you've found an error in the scanner. Note that you will need to write the allTokens.in file.

Testing

Part of your task will be to figure out a strategy for testing your implementation. Part of your

Part of your task will be to figure out a strategy for testing your implementation. Part of your grade will be determined by how thoroughly you test your scanner. At minimum, you should test that every class of token is recognized and that each error message is exercised.

You will test your program by providing input files for distinct test cases. Note that the input files do *not* have to be legal Array or C++ programs, just sequences of characters that correspond to Array tokens. **Don't forget to include tests for the correct character number (as well as line number) is returned for every token!**

Your input files should exercise all of the code in your scanner, including the code that reports errors. Add to the provided Makefile (as necessary) so that running make test runs your P2 and does any needed file comparisons (e.g., using diff) and running make cleantest removes any files that got created by your program when P2 was run. It should be clear from what is printed to the console when make test is run what errors have been found.

To test that your scanner correctly handles an unterminated string literal with end-of-file before the closing quote, you may use the file eof.txt. On a Linux machine, you can tell that there is no final newline by typing: cat eof.txt or od -a eof.txt You should see your command-line prompt at the *end* of the last line of the output instead of at the beginning of the following line for cat, and the file will not display a newline.

Working in Pairs

Computer Sciences and Computer Engineering graduate students must work alone on this assignment. Undergraduates, special students, and graduate students from other departments may work alone or in pairs.