Program 2

Overview

Due by Sep 30 at 11pm.

For this assignment you will use JLex to write a scanner for our language, called `harambe`, a small subset of the C++ language. Features of `harambe` that are relevant to this assignment are described below. You will also write a main program (`P2.java`) 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

Dowload the skeleton code for this project here. Once you unzip it, you will see two folders *deps* and *files*. To compile your project, run the makefile inside the *files* folder.
Note that you need to have the *deps* folder to make this project without errors.

The *files* folder contains the following.

- `cats.jlex`: An example JLex specification. *You will need to add to this file*.
- `sym.java`: Token definitions (this file will eventually be generated by the parser generator). Do not change this file.
- `ErrMsg.java`: The `ErrMsg` class will be used to print error and warning messages. Do not change this file.
- `P2.java`: Contains the main program that tests the scanner. *You will need to add to this file*.
- `Makefile`: A Makefile that uses JLex to create a scanner, and also makes `P2.class`. *You may want to change this file*.

JLex

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

If you work on a CS Dept. Linux machine, you should have no problem running JLex. You will not be able to work on the CS Dept. Windows machines.

The Language

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

Tokens

The tokens of the `cats` language are defined as follows:

- Any of the following reserved words (remember that you will need to give the JLex 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 new line 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 `sym.java`. 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 JLex to return the special `EOF` token on end-of-file has already been included in the file `cats.jlex` -- 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 `sym.java`.
- 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 in `java_cup.runtime`; 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 `cats.jlex` for examples of how to call the `Symbol` constructor. See `P2.java` for code that accesses the fields of a `Symbol`.

In your compiler, the `value` field of a `Symbol` will actually be of type `TokenVal`; that type is defined in `cats.jlex`. 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 `cats.jlex`.

Line counting is done by the scanner generated by JLex (the variable `yyline` 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 `cats.jlex` 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 JLex 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 JLex 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 = (new Double(yytext())).doubleValue(); // convert String to double
// INSERT CODE HERE TO CHECK FOR BAD VALUE -- SEE ERRORS AND WARNINGS BELOW
int k =  (new Integer(yytext())).intValue();    // 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, another backslash, or a question mark. 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 `Integer.MAX_VALUE`)**

Issue the warning message: `integer literal too large; using max value` and return `Integer.MAX_VALUE` 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.java`. 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. As mentioned in the Overview, part of your grade will be determined by how thoroughly your main program tests your scanner.

You will probably want to change `P2.java` to read multiple input files so that you can test other features of the scanner. You will need to create a new scanner each time and you will need to set `CharNum.num` back to one each time (to get correct character numbers for the first line of input). Note that the input files do *not* have to be legal `cats` or C++ programs, just sequences of characters that correspond to `cats` tokens. **Don't forget to include code that tests whether the correct character number (as well as line number) is returned for every token!**

Your `P2.java` 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 `files/eof.txt`. On a Linux machine, you can tell that there is no final newline by typing: `cat 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.

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.

If you plan to work with a partner, you must let us know ***no later than September 22nd***. To let us know, each partner should hand in a README.txt file, through learn@UW into the p2 submission, filled in with the name and CS login of *both* partners.

If you want to work with a partner, but don't have one, check out the "Search for Teammates!" note in Piazza.

If you are working with a partner and you decide to split up, you must let the TAs know by email so that we can arrange how to divide up any code that has already been written.

Below is some advice on how to work in pairs.

This assignment involves two main tasks:

1. Writing the scanner specification (`cats.jlex`).
2. Writing the main program (`P2.java`).

An excellent way to work together is to do *pair programming*: Meet frequently and work closely together on both tasks. Sit down together in front of a computer. Take turns "driving" (controlling the keyboard and mouse) and "verifying" (watching what the driver does and spotting mistakes). Work together on all aspects of the project: design, coding, debugging, and testing. Often the main advantage of having a partner is not having somebody to write half the code, but having somebody to bounce ideas off of and to help spot your mistakes.

If you decide to divide up the work, you are *strongly* encouraged to work together on task (1) since both partners are responsible for learning how to use JLex. You should also work together on testing; in particular, you should each test the other's work.

Here is one reasonable way to divide up the project:

- Divide up the tokens into two parts, one part for each person.
- Each person extends their own copy of `cats.jlex` by adding rules for their half of the tokens, and extends their own copy of the main program to handle those same tokens.
- Decide together how your `P2.java` should work, and write that code.
- Write test input files for your own tokens, and for the other person's tokens, too.
- After each person makes sure that their scanner and main program work on their own tokens, combine the two (it should be pretty easy to cut and paste one person's JLex rules into the other person's `cats.jlex`, and similarly for the main program).
- Do *not* try to implement all of your half of the tokens at once. Instead, implement just a few to start with to make sure that you both know what you're doing and that you're able to combine your work easily.

The most challenging JLex rules are for the `STRINGLITERAL` token (for which you will need several rules: for a correct string literal, for an unterminated string literal, for a string literal that contains a bad escaped character, and for a string literal that contains a bad escaped character *and* is unterminated). Be sure to divide these up so that each person gets to work on some of them.

It is **very** important to set deadlines and to stick to them. I suggest that you choose one person to be the "project leader" (plan to switch off on future assignments). The project leader should propose a division of tokens, as well as deadlines for completing phases of the program, and should be responsible for keeping the most recent version of the combined code (be sure to keep back-up versions, too, perhaps in another directory or using a version-control system like Mercurial or Git).

To share your code, you can either use e-mail or the project leader can create a directory for the combined code (*not* the directory in which that person develops the code). I suggest that you create a new top-level directory (i.e., at the same level as your `public` and `private` directories), named something like `cs536-P2`. To set the permissions of the directory for the combined code to allow your partner to write into it, change to that directory and type:

```
fs setacl . <login> write
```

using your partner's CS login in place of `<login>`. You should also prevent any other access by typing:

```
fs setacl . system:anyuser none
```

in the new directory that you create (*not* in your top-level directory). To see what the permissions are in your current directory, type:

```
fs listacl
```

Do *not* try to share by letting your partner log in to your account. Departmental and University policy prohibits your revealing your password to anybody else, including your partner.

Handing in

You will be needed to submit the the entire working folder (all the java files, jlex file, Makefile and pdf) as a compressed file. Please look into Handing Instructions here

Grading criteria

General information on program grading criteria can be found on the Assignments page).

For more advice on Java programming style, see Code Conventions for the Java Programming Language. See also the style and commenting standards used in CS 302 and CS 367.