Programming Language Design and Implementation

CS - 1319 - 1 `python-shivam` **Group: 7**
Monsoon 2023 **Saptarishi Dhanuka**
Assignment 2 **Shivam Kedia**

1 Flex Specifications

1.1 Regular Expression Definitions

1. Keyword: First we defined regular expressions for each keyword that matches exactly that keyword by using double quotes. Then we defined KEYWORD as matching any of the previously defined keywords with

   ```
   {INT}|{CHAR}|{ELSE}|{FOR}|{IF}|{RETURN}|{VOID}
   ```

2. Identifier: To help our regex definition of ID we first defined some other rules that would make up the definition for ID. We did this since it's easier than putting all the regex for ID into just one pattern altogether. Since we defined digit and idnondigit earlier, we can simply follow the recursive definition that's given in the assignment to construction the regex for ID. The first character of ID, as per the definition, has to be an underscore or a letter (an idnondigit), which can be followed by zero or more digits or idnondigits

3. Constant: Again we defined few "helper" regular expressions that would eventually make up the regexes for integer and character constants. A constant matches either an integer or character constant as follows:

   (a) Integer Constant: As per the given specification, if an integer constant has a sign, then it must start with a nonzero digit, followed by zero or more digits. In case there is no sign there it can be made of 0 or more digits

   (b) Char Constant: As per the specs, a c-char can be anything from the char set except single quote, backslash or a newline character. The last restrictions means we cannot have a character spanning multiple lines. Or it could be an escape sequence, defined earlier. The final charconst is simply one or more c-chars enclosed by single quotes.

4. String Literal: The regular expression for this is quite similar to a character constant, except that we can allow empty strings and we cannot allow double quotes as an s-char.

5. Punctuator: Listed the exact matches of the given punctuators separated by OR operators so that PUNC would match any of them.

6. White Spaces: Check either of single white space ( ), tab ($\backslash t$), or newline ($\backslash n$)

7. Comments:

   - Single Line Comments: We defined the regex for single line comments in the rules section itself. It first matches "//" literally followed by zero or more instances of anything except the newline.

- Multi Line Comments: We used Start Conditions for handling multiline comments. While a regular expression for this was possible, it was easier for us to use start conditions to define this. Also it makes it easier to find an unterminated comment if we use start conditions. In the definitions sections, we wrote %x MCOMMENT since we are defining an exclusive start condition where only rules prefixed with the MCOMMENT qualifier will be active.

1.2 Rules and Actions

The order of the translation rules is important since Flex will choose the earlier one in case of a clash so we have written the order as follows:

```
{KEYWORD}               {printf("<KEYWORD, %s>\n", yytext);}
{CONST}                 {printf("<CONSTANT, %s>\n", yytext);}
{STRING-LITERAL}        {printf("STRING-LITERAL, %s>\n", yytext);}
{ID}                    {printf("<IDENTIFIER, %s>\n", yytext);}
{PUNC}                  {printf("<PUNCTUATOR, %s>\n", yytext);}
"/*"                    {BEGIN(MCOMMENT);}
<MCOMMENT>"*/"          {BEGIN(0);}
<MCOMMENT>(\n)+|.       {;}
<MCOMMENT><<EOF>>       {printf("<INVALID INPUT, UNTERMINATED \"/*\">\n");
                        return yywrap();}

"//".*                  {;}

{WS}                    {;}
.                       {printf("<INVALID INPUT: %s>\n", yytext); return yywrap();}
```

This ensures that any keyword is not mistaken for an identifier or any other bad case that may occur.

1. For KEYWORD, CONST, STRING-LITERAL, ID and PUNC we just output the corresponding token which contains the yytext.
2. Multiline comments:
   - When we encounter the the exact string '/*', we begin the MCOMMENT state to indicate that we are in a multiline comment. This does not lead to the case where a comment could start inside a string or a character since we have dealt with STRING-LITERALs and CONSTs earlier on in the Rules.
   - When we encounter the exact string '*/' **AND** we are in the MCOMMENT state already, this indicates the end of the multiline comment and we can go into the 0 state. BEGIN(0) just goes back to the original state where the active rules were the ones without any start conditions prefixed to them. We do this since we have ended the comment and want to recognise the other rules in the rest of the file.
   - To eat up everything within a comment, we use a regex. If we are in the MCOMMENT state, it recognises one or more newlines or anything other than a newline

and the action is doing nothing since we are just ignoring whatever is in the comment.
Comments can't nest since when we start a comment we are not in the MCOMMENT state.
Moreover, we don't run into the case where we eat up the '*/' since its rule lies before the eating up rule, so when a comment termination occurs, we can detect it and end the comment. This also takes care of the greedy problem.

- If we don't find a comment termination and we are in the MCOMMENT state and we have matched the end-of-file, then we raise an error. This is a design choice we have made and the alternatives could have been to either recognise / and * as punctuators or treat everything after the /* as a comment and ignore them. In our design since we are using start conditions, when we see a /*, we start the MCOMMENT state and since there is no */, we never exit out of it.

3. Single-line comments: For this we used a simple regex which matches "//" literally, followed by zero or more instances of anything except newline, and we just ignore whatever is in the comment.

4. Ignore whitespace

5. Anything else which has not matched the above rules is marked as invalid input.

2 Main Function and Makefile

2.1 Makefile Rules

- An empty `.SUFFIXES` clause is used to disable the implicit rules for all suffixes. We didn't pick and choose here because there were too many. We just wanted `flex` to not overwrite `7_A2.c` after only `7_A2.l` was touched.

- `.PHONY = clean` tells `make` that clean is not a file but a fake target of sorts which doesn't need to be checked for latest changes

- `clean` Removes the intermediate files, viz. `.o`, `lex.yy.c` and the executable `lexer`

- `build` Forcefully recompiles the files to produce `lexer` as the executable. It needs to have `7_A2.c` and `lex.yy.c` which is obtained from the target `lex.yy.c`. Then a similar clean is performed except `lexer` is preserved.

- `lex.yy.c` is the obtained by running `7_A2.l` through `flex`. Each time the `.l` file changes the target is recreated. Additionally since `lex.yy.c` is cleared each time after compilation, `build` manages to forcefully recompile because `lex.yy.c` is only created from this target and immediately removed afterwards.

2.2 Main Function

It's in the separate file as per the guidelines and it just calls yylex().

3 Test File: 7_A2.nc

We have first added all the 9 or so programs mentioned in the Assignment PDF. Besides checking normal functioning (like the respective token classes being scanned correctly), our custom test cases check for the following edge cases (all these are explained by comments in the `.nc` file itself):

- Integer Constants:
  - Shouldn't be +0... or -0...
  - can be like 0001239213, (unsigned starting with 0)
  - or +9321000, -12321, etc. (signed, not starting with 0)
- Character Constants:
  - Single quotes, Multiple lines, String literals within a constant aren't allowed
  - Multi-character constants including (escaped characters) are legal.
- String Literals: Similar constraints to character constants, `" " "` isn't allowed, but `"\""` is fine. Single quotes, escaped characters within strings are okay.
- Multi-line comments:
  - Can't be nested, start within a single line comment but end on another line
  - Can't begin in a string, or character constants
  - The matching is supposed to not be greedy, i.e,
    `/* hehe, commented!!! */ but this is not commented */`
  - `*` and `/` when separated by a space, or if they randomly occur in the comment don't act as terminating sequences
  - If the line ends before we find a closing sequence `*/` we don't consider the lexemes `/*` onwards as tokens, we just claim Invalid Input because it was mentioned that the starting sequence "introduces a comment". We've interpreted this as entering the comment state, hence the decision.
- Single-line comments:
  - End at the same line
  - Don't start within strings, or character constants
  - Anything after a \\ is ignored
  - We don't deal with `/` as continuing a line comment unlike some standard implementations.