Project 1 Report

1. Basic Requirement

1. Basic parser
2. Read file via input args
3. Report lexical error(type A) and syntax error(type B)

Since these basic features are quite simple, we just skip them.

2. Optional features

1. Supporting single-line and multi-line comments.

   Solution

   For single-line comment, when program encounter "//" , then it just eat the hold line:

   Single-line comment

```
Line 1  "//".*"\n" {
    -       yylineno++;
    -   }
```

   For multi-line comment, here I use start state to implement this feature:

   Single-line comment

```
Line 1  %x COMMENT
    -   "/*" {
    -       BEGIN(COMMENT);
    -   }
    5   <COMMENT>"*/" {
    -       BEGIN(INITIAL);
    -   }
    -   <COMMENT><<EOF>> {
    -       fprintf(out, "Error COMMENT at Line %d: missing '*/'\n", yylineno);
   10   }
    -   <COMMENT>. {
    -   }
    -   <COMMENT>\n {
    -       yylineno++;
   15   }
```

   When first encounter "/*", begin the state COMMENT, then we just eat all content except "*/" string or «EOF». If there is no corresponding "*/", we can report an error.

   Actually, when first time I implement this feature, I just use regular expression to handle it however it is quite complex and hard to implement non-greedy behavior(we just need to match first "*/" after "/*", but quite difficult). Then I get this answer from stack overflow[1], the following features are all implemented by start state.

2. Supporting hexadecimal representation of integers and report lexical errors. To recognize a

   Solution

To recognize a correct hexadecimal representation of integers is so easy, we just need to define the characters of hex, every time match the string starting with "0x" connected with hex characters.

However, to detect a error is hard. If we just recognize the string start with "0x" and following with invalid hex characters, this could be a problem. For example, 0x5f+f is an valid form.

By using start state, the problem will be much simpler:

Support hex and report error

```
hex [0-9a-fA-F]
struct [ .;,=<>!=+\-*/&|()\[\]{}]
%x INT_HEX
"0x" {
    BEGIN(INT_HEX);
}
<INT_HEX>{hex} {
    ...
}
<INT_HEX>({struct}|"\n") {
    ...
    if(wrong == 1){
        ...
        return UNKNOW;
    }
    ...
    return INT;
    }
<INT_HEX>. {
    wrong = 1;
}
```

Here I define a state INT_STATE to represent we already encounter "0x". The {struct} contain the characters such as operator and other structural symbol. We want these structural symbol to split these character. Start state has a good advantage: we can represent the complement of {hex} and {struct} and \n easily: since flex match from top to down when these rules match same length string. If we detect some thing is in the complement of {hex}, {struct}, \n , then we know there are some invalid characters, thus we set $wrong = 1$;. When program detect {struct} or \n, it will end this state because we already recognize a string. The variable $wrong$ will tell us whether this string is valid or invalid. If $wrong == 1$ we can report an error.

3. Supporting hex-form characters, and report lexical errors.

   Solution

   The same method as features 2.

4. Detecting nested multi-line comments, and report a syntax error.

   Solution

   Use start state to solve it again, if we encounter another "/*" when we are in the state <COMMENT>, we know there is a nested comment or there is an error. Then we turn into state <NESTED_COMMENT>.

Support hex and report error

```
%x NESTED_COMMENT
<COMMENT>"/*" {
    BEGIN(NESTED_COMMENT);
```

```
}
<NESTED_COMMENT>"*/" {
    print error ...
    BEGIN(COMMENT);
}
<NESTED_COMMENT><<EOF>> {
    print error
}
<NESTED_COMMENT>. {
}
<NESTED_COMMENT>\n {
}
```

In the state <NESTED_COMMENT>, if we encounter "*/" then we detect the nested comment, report a error and turn to state <COMMENT> again so that we will not affect the running of program, otherwise just eat inputted character.

3. Bonus features

1. macros

   Solution

   This feature is quite hard, the specific details is complex, thus I just briefly explain the principle.

   There are two type of macros: with parameters and without parameters. For latter case, it implement is simpler.

   Firstly, I detect "#define", recognize the following macro and the text behind the macro name, put them into a linked list.

   Secondly, when there is a inputted English character, program will judge it if string start by this character can match the macro name in the linked list. If match, program use input() function to clean this string, and then use unput() to put the macro text into input buffer.

   However, the case with parameters need a extra steps: judge whether parameter is valid, whether there is an empty parameter, and how many parameters in brackets. Before we unput() the macro text into input stream, we need to replace corresponding parameters in the macro text.

   The grammar I am referring to is http://c.biancheng.net/view/446.html [2]. Since the grammar defined by the project document is weak, thus I just complete two main features.

   For more features, please see the test cases in the project.

2. file include

   Solution

   Quite easy feature, we just need to detect "#include" head and the file name behind it, and then return a INCLUDE token, bison file will add INCLUDE in the IncludeList.

3. else-if statement

   Solution

   Basic features are already supported.

4. for statement

   Solution

   Add a FOR token, and add for statement format into Stmt.

References

[1] difficulty getting c-style comments in flex/lex

https://stackoverflow.com/questions/2130097/difficulty-getting-c-style-comments-in-flex-lex

[2] C Language Macro

http://c.biancheng.net/view/446.html