CS-1319: PLDI - Monsoon 23	**Team Name:** julius-stabs-back
Assignment #2	
Instructor: PPD	**Name:** Gautam Ahuja, Nistha Singh

Note: All of the codes are compiled on `gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0` over the Windows Subsystem for Linux version: 1.2.5.0 using Ubuntu 22.04.2 LTS and flex 2.6.4.

1 Pipeline

We set up a pipeline for the code to flow through as follows:

1.1 `3_A2.c`

This is the `.c` file which contains the `main()` function to text our lexer. We followed the *Assignment Guide* shared to create this file as follows:

```
// Initialize the yylex() function for the lexer
int yylex();
// main to drive the yylex() engine
int main() {
    int token;
    while(1){
        token = yylex();
        // Check if incoming token is INVALID (not in grammar)
        if(token == -1){
            // Stop the program immediately
            return 0;
        }
    }
    return 0;
}      Gautam8387, 18 hours ago • Main Attempted ...
```

The above codes prints the tokens as per the definitions of lexical rules. If it however encounters, -1, which is when lexer encounters an invalid token, the program stops immediately.

1.2 `Makefile`

We generate a `Makefile` to compile the `flex 3_A2.l` file, link the generated `3_A2.yy.c` with `3_A2.c` file and generate `3_A2.out` file. Then we feed it our test file `3_A2.nc` which contains a test `nanoC` program. The `3_A2.out` then gives us the lexical tokens for that test file.

```
all:
    flex -o 3_A2.yy.c 3_A2.l
    gcc 3_A2.yy.c 3_A2.c -ll -o lexer.out
    ./lexer.out < 3_A2.nc

clean:
    rm lexer.out 3_A2.yy.c

build: clean
    flex -o 3_A2.yy.c 3_A2.l
    gcc 3_A2.yy.c 3_A2.c -ll -o lexer.out

test: build
    ./lexer.out < 3_A2.nc
```

The output of `flex -o 3_A2.yy.c 3_A2.l` is the file `3_A2.yy.c` which is the lexer analyser and containing the functionality and rules implemented in `3_A2.l` in form of Discrete Finite Automata.
The rule for `all` always works when you type `make` in terminal. The rule for `clean` runs when you type `make clean`. The rule for `rebuild` forcefully rebuild the executable. The rule for `test` can be used to get the lexical analysis of `3_A2.nc` file. It rebuilds the executable and run the test.
ASSUMPTION: The rule for `rebuild` will run successfully only if a build is already present, ie, `make` has run before. Directly executing `make rebuild` will throw errors.
This then takes the `3_A2.nc` as input and performs the rules to tokenize entities.

1.3 Test File

The file `3_A2.nc` is test file which contains some `nanoC` code. This will be used to test all rules of lexical analyzer.

2 Flex Specification

Our main `3_A2.l` contains the flex code which has all the lexical grammar and regular expressions.

2.1 Keywords

The given keywords are `char, else, for, if, int, return, void`. We write the Regular Expressions for these rules as follows:

```
/* Regular Expressions */
CHARACTER   "char"
ELSE        "else"
FOR         "for"
IF          "if"
INTEGER     "int"
RETURN      "return"
VOID        "void"
```

The Definitions of the above Rules are:

```
{CHARACTER}     {printf("<KEYWORD char>\n");}
{ELSE}          {printf("<KEYWORD else>\n");}
{FOR}           {printf("<KEYWORD for>\n");}
{IF}            {printf("<KEYWORD if>\n");}
{INTEGER}       {printf("<KEYWORD int>\n");}
{RETURN}        {printf("<KEYWORD return>\n");}
{VOID}          {printf("<KEYWORD void>\n");}
```

2.2 Identifiers

Since our valid identifiers grammar include _, a-z, A-Z and 0-9. It all has to be included in the identifier.

```
IDENTIFIER  [_a-zA-Z][_a-zA-Z0-9]*
```

The output of this will be:

```
{IDENTIFIER}    {printf("<IDENTIFIER %s>\n", yytext);}
```

Here, `yytext` will parse the input stream and create a token of output stream of identifiers.

2.3 Constants

We refer to the **Flex & Bison** book by **John Levine** shared and find a few tricks that can be used here.

- {}: If the braces contain a name, they refer to a named pattern by that name.
- `"..."`" Anything within the quotation marks is treated literally. Metacharacters other than C escape sequences lose their meaning. As a matter of style, it's good practice to *quote any punctuation characters intended to be matched literally.*
- `()`: Groups a series of regular expressions together into a new regular expression. Parentheses are useful when building up complex patterns with *, +, ?, and |.

2.3.1 Integer Constants

An integer-constant is defined as:

$$integer - constant = \begin{cases} 0 \\ sign_{opt} nonzero - digit \\ integer - constant - digit \end{cases}$$

This expression can be further broken as $nonzero - digit \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $sign \in \{+, -\}$, and $digit \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Therefore we can use an abstraction technique and write regular expression for each of these lower definitions as:

```
SIGN                [+-]
NONZERO_DIGIT       [1-9]
DIGIT               [0-9]
```

The first `SIGN [+-]` means any characters amongst `+, -`. Later expression denotes any digit from 1 through 9.
Now for $integer - constant$ we can write,

```
INTEGER_CONSTANT   0|({SIGN}?({NONZERO_DIGIT}))({DIGIT})*
```

We use `{, }` to refer to the Regular Expression of $NONZERO_DIGIT$ This means 0 OR a a nonzero-digit followed by a sign and then digits following in kleene closure (zero or more digits).
This will capture all of the Integer Constants as defined.

2.3.2 Character Constant

A character-constant is defined as $character - constant =' c - char - sequence'$. Where $c - char - sequence$ is a Positive closure of $c - char$. And $c - char$ is any character from from $escape - sequence$ or any member of the source character set except $\backslash', \backslash\backslash, \backslash n$.
We can start from the smallest unit here $escape - sequence$ and build the Regular Expression upwards. We define $escape - sequence$ as:

```
/* escape-sequence: any one of the \', \'', \?, \\, \a, \b, \f, \n, \r, \t, \v */
ESCAPE_SEQUENCE    "\'"|"\""|"\?"|"\\a"|"\\b"|"\\f"|"\\n"|"\\r"|"\\t"|"\\v"
```

We use `"..."` as suggested in book to literally match the escape sequences.
Next we make an expression for $c - char$ which is either $escape - sequence$ or any member of the source character set except $\backslash', \backslash\backslash, \backslash n$ as:

```
/* c-char: escape-sequence or any character except single quote ', backslash \, or new line */
C_CHAR              ({ESCAPE_SEQUENCE})|([^\'\\\n])
```

We use `{,}` to refer to the Regular Expression of $ESCAPE\_SEQUENCE$.
Next we define regular expression for $c - char - sequence$ as a Positive Closure of $c - char$ as:

```
/* c-char-sequence: c-char | c-char-sequence c-char */
CHAR_SEQUENCE       {C_CHAR}+
```

Now, at last $character - constant$ is just $c - char - sequence$ within single quotes. Therefore expression for it is:

```
/* character-constant: 'c-char-sequence' */
CHARACTER_CONSTANT  ([\'])({CHAR_SEQUENCE})([\'])
```

Here first group ([\′]) matches for opening single quote followed by the $c - char - sequence$ and closing single quotes.

Now we have a regular expression for both Integer-Constant and Character-Constant. We just combine the two sequences in group with an `OR` to get the result for `constant` token as follows:

```
/* CONSTANT: integer-constant or character-constant */
CONSTANT            ({INTEGER_CONSTANT})|({CHARACTER_CONSTANT})
```

The output of the above complete expression will be:

```
{CONSTANT}  {printf("<CONSTANT, %s>\n", yytext);}
```

Where `yytext` will parse the input stream and create a token of output stream of constants.

2.4 String Literals

The structure of $string - literal$ is more or less similar to $character - constant$.
A $string - literal$ is a $s - char - sequence_{opt}$ within double quotes.
A $s - char - sequenceopt$ is a positive closure on $s - char$.
A $s - char$ is is either $escape - sequence$ or any member of the source character set except \”, \\, $\backslash n$.

Since we already have our $escape - sequence$ defined, we define our $s - char$ as:

```
/* S-Char: escape-sequence or any character except double quote ", backslash \, or new line */
S_CHAR              ({ESCAPE_SEQUENCE})|([^\"\\\n])
```

We use {,} to refer to the Regular Expression of $ESCAPE\_SEQUENCE$.
Next we define regular expression for $s - char - sequence$ as a Positive Closure of $s - char$ as:

```
/* S-Char-Sequence: S-Char | S-Char-Sequence S-Char */
S_CHAR_SEQUENCE     {S_CHAR}+
```

Now, at last $string - literal$ is just $s - char - sequence$ within double quotes. Therefore expression for it is:

```
/* String-Literal: S-Char-Sequence_opt. Terminated by null = '\0' */
STRING_LITERAL      ([\"])({S_CHAR_SEQUENCE})([\"])
```

Again, here first group ([\”]) matches for opening double quote followed by the $s - char - sequence$ and closing double quotes.
Now the output of the above expression will be:

```
{STRING_LITERAL}    {printf("<STRING_LITERAL, %s>\n", yytext);}
```

Again, here `yytext` will parse the input stream and create a token of output stream of string-literals.

2.5 Punctuators

The grammar for punctuators is just any one of the following:
`[ ] ( ) { } -> & * + - / % ! ? < > <= >= == != && || = : ; ,`
Again, since we are dealing with literals and escape sequences, we are better off using `"..."` for each punctuation seperated by `OR` as follows:

```
/* Punctuators: one of [ ] ( ) { } -> & * + - / % ! ? < > <= >= == != && || = : ; ,*/
PUNCTUATORS     "["|"]"|"("|")"|"{"|"}"|"->"|"&"|"*"|"+"|"-"|"/"|"%"|"!"|"?"|"<"|">"|"<="|">="|"=="|"!="|"&&"|"||"|"="|":"|";"|","
```

The output of above will be as follows:

```
{PUNCTUATORS}   {printf("<PUNCTUATOR, %s>\n", yytext);}
```

Where `yytext` will parse the input stream and create a token of output stream of punctuators.

2.6 Comments

The comments in `nanoC` are of two types: Multi-line Comment and Single-line Comment.

2.6.1 Multi-line Comment

These comments starts with a `/*` and end with a `*/`. Everything in between is commented out. This comment is not nested.
To derive a Regular Expression for this comment we may thing about it as follows:

- A comment start with a `/*` and end with a `*/`.
- All of the comment in between do not contain a a `*/` because if it does, the comment ends. So it forms two cases for `*`:
  - There is not a `*`. Hence all other characters are in a comment. OR
  - There is a `*` but it does not follow a `/`, because otherwise the comment will end. Therefore `*` is followed by any character but `/`.
- Therefore out Regular Expression for above two condition becomes ([^*]|[^\/]) under a Kleene Closure (as comment can be empty string).

Therefore the regular expression for Multi-line Comment becomes:

```
/* Multi-line comments :Start \/\*, End \*\/, In betwen everything is ignored */
/* In between, if there is a * followed by /, then it must be end of comment */
/* If middle sequence is [^\*] not star, it can have any character */
/* If middle sequence is [\*] a star, it must NOT follow a [/] for comment to not end*/
MULTI_LINE_COMMENT      (\/\*)([^\*]|\*[^\/])*(\*\/)
```

2.6.2 Single-line Comment

These comments starts with a `//` and end with a $\backslash n$. Therefore a Kleene Closure over all characters following `//` which are `NOT` $\backslash n$ will be a valid Regular Expression for this as follows:

```
/* Single line comments: Start //, End \n, In between everything is ignored */
SINGLE_LINE_COMMENT    (\/\/)([^\n])*
```

Now we have a regular expression for both Single-line Comment and Multi-line Comment. We just combine the two sequences in group with an OR to get the result for comment token as follows:

```
/* Comments: Multi-line or Single-line */
COMMENT     ({MULTI_LINE_COMMENT})|({SINGLE_LINE_COMMENT})
```

The output of above expression will be as follows:

```
{COMMENT}       {printf("<COMMENT, %s>\n", yytext);}
```

Where `yytext` will parse the input stream and create a token of output stream of comments.

2.7 White-Spaces*

Since there is no formal rules given for White spaces which have a regular expression of:

```
/* Ignore Whitespace */
WHITESPACE  [ \t\n\r]
```

We assume white spaces as space, tab, new line and carriage return ($\backslash t, \backslash n, \backslash r$).We have chosen to ignore the white spaces. Hence there is no definition of rules and actions for white spaces.

```
{WHITESPACE}    /*Ignore whitespace*/
```

2.8 Other Tokens

We have defined our grammar and rules. But since they are limited and not everything can be tokenized, we create another rule for whatever is leftover as follows:

```
.               {printf("<INVALID_TOKEN, %s\n>", yytext); return -1;}
```

This then checks if anything is left out, is not assigned a token and is invalid. In such case it prints the invalid token and return the `yywarp()` with -1. This then checked by `main()` of `3_A2.c` and it stops the lexer at that point.

3 Testing

To test the lexer, we take a program `Bubble Sort` in `nanoC`.

```
/* Bubble Sort Algorithm in nanoC language.
This test program (3_A2.nc) that will check all the lexical rules for all tokens:
keyword, identifier, constant, string-literal, punctuator, white-space* and invalid-tokens.
Team: julius-stabs-back
Members: Gautam Ahuja, Nistha Singh
*/

// Forward declarations
void swap(int *p, int *q);
void readArray(int size);
void printArray(int size);
void bubbleSort(int n);

int arr[20]; // Global array

// Driver program to test above functions
int main()
{
    int n;
    printStr("Input array size: \n");
    readInt(&n);
    printStr("Input array elements: \n");
    readArray(n);
    printStr("Input array: \n");
    printArray(n);
    bubbleSort(n);
    printStr("Sorted array: \n");
    printArray(n);
    return 0;
}

void swap(int *p, int *q)
{ /* Swap two numbers */
    int t = *p;
    *p = *q;
    *q = t;
}
void readArray(int size)
{ /* Function to read an array */
    int i;
    for (i = 0; i < size; i = i + 1)
    {
        printStr("Input next element\n");
```

```
        readInt(&arr[i]);
    }
}
void printArray(int size)
{ /* Function to print an array */
    int i;
    for (i = 0; i < size; i = i + 1)
    {
        printInt(arr[i]);
        printStr(" ");
    }
    printStr("\n");
}
void bubbleSort(int n)
{ /* A function to implement bubble sort */
    int i;
    int j;
    for (i = 0; i < n - 1; i = i + 1)
        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j = j + 1)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
#include
/* This is the ending multi-line comment.
    It will not be read by the lexer as it stops above when it
    ↪ encounters the above invalid token '#'. The next work to #
    ↪ (include) will also be not read. */
```

After running the `Makefile`, the lexer code checked all the lexical rules and generated output for all token classes: keyword, identifier, constant, string-literal, punctuator, white-space* and invalid-tokens.