CS1319 - Monsoon 2023 - Assignment 2

Lexical and Syntactical Debuggers
Aryika Mehrotra and Aaryan Nagpal

This is our document explaining our assignment, the lexer. We have used GCC Version: `11.4.0`, on an Intel Processor.

MakeFile

We first started with creating a MakeFile and making our pipeline ready as we code. Our code in MakeFile is:

```
all: 26_A2.nc
        flex 26_A2.l
        gcc lex.yy.c 26_A2.c -ll -o 26_A2.out
        ./26_A2.out < 26_A2.nc

test: 26_A2.nc
        flex 26_A2.l
        gcc lex.yy.c 26_A2.c -ll -o 26_A2.out
        ./26_A2.out < trial.nc

clean:
rm -f lex.yy.c 26_A2.out
```

Right now, all are empty files. We have first wrote on the `all` section, where we create `lex.yy.cc` file, which simulates the Discrete Finite Automata, after inputting our `.l` lexer file.

After creating `lex.yy.cc`, we run it through GCC, along with our main C file, compiling it and creating the output file. Our output file is finally run after taking our NanoC file as input and outputs the results.

We have additionally created a `clean` command for removing the output files respectively.

Main C File

Our main C file includes the code:

```
1  int yylex();
2
3  int main(){
4      yylex();
```

This works as an entry point for running our lexer. The call to `yylex()` makes the lexer start tokenizing source code and return the recognized tokens based on the given input.

Lexer File

We will follow the order of precedence given in the assignment.

Keyword

We have the first task as finding the keywords. Our code which does that is:

```
CHAR            "char"
ELSE            "else"
FOR             "for"
IF              "if"
INT             "int"
RETURN          "return"
VOID            "void"
```

Using this, we can define the outcomes when we come across these particular tokens and label them as `KEYWORD`. This looks like:

```
{CHAR}          {printf("<KEYWORD, %s>\n", yytext);}
{ELSE}          {printf("<KEYWORD, %s>\n", yytext);}
{FOR}           {printf("<KEYWORD, %s>\n", yytext);}
{IF}            {printf("<KEYWORD, %s>\n", yytext);}
{INT}           {printf("<KEYWORD, %s>\n", yytext);}
{RETURN}        {printf("<KEYWORD, %s>\n", yytext);}
{VOID}          {printf("<KEYWORD, %s>\n", yytext);}
```

Identifier

We follow the rules given in the Assignment file by first defining `non-digit` and `digit`. Afterwards, we define our combination keeping in mind that an `identifier` cannot start from a `digit` but only a `non-digit`. Afterwards, it can either be followed by a `digit`, `non-digit` or empty string, hence the kleene closure. We write the following:

```
DIGIT           [0-9]
NON_DIGIT       [a-zA-Z_]

IDENTIFIER      ({NON_DIGIT})({DIGIT}|{NON_DIGIT})*
```

For this rule, we set the outcome:

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

Constant

To label a `constant`, we either need `integer-constant` or `character-constant`. We write the following:

```
CONSTANT        ({INTEGER_CONSTANT})|({CHARACTER_CONSTANT})
```

For this rule, we set the outcome:

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

We add this after we define Integer Constant and Character Constant

Integer Constant

As we saw in the assignment, we have multiple levels, which are `digit`, `nonzero-digit` and `sign`.

Starting with `sign`, we only need either `+ or -` and in `nonzero-digit`, we need from 1 to 9. I also defined 0 as a separate `zero`. We would not focus about `digit` here since we have defined it earlier.

Now that we have defined our levels, we follow the definition and write the final combination. Since first precedence is `zero`, I mention it first as our `integer-constant` can either be defined as a single `zero`, or an optional `sign`, followed by a `nonzero-digit`. After that, `digit` be repeated, or not hence the kleene closure. Our code looks like:

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

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

Character Constant

Again, we begin by defining `c-char-sequence`, `c-char` and `escape-sequence`.

We write all the escape sequences in `escape-sequence` and since `c-char` can have any character or some escape sequence, except single quote, backslash or new line character, we negate them when writing the rule.

Now, `c-char-sequence` can either be just a `c-char` or more than one, hence we apply positive enclosure. Finally, our final combination has only got `c-char-sequence`, enclosed in single quotes. Our code looks like:

```
ESCAPE_SEQUENCE     [\'\"?\\abfnrtv]
C_CHAR              ({ESCAPE_SEQUENCE})|([^\'\\\n])
CHAR_SEQUENCE       {C_CHAR}+

CHARACTER_CONSTANT  ([\'])({CHAR_SEQUENCE})([\'])
```

String Literal

We define `s-char-sequence` and `s-char`. I would not define `null` as it is included automatically. Since `s-char` can have any character or escape sequences, except double quote, backslash or new line character, we negate them again when writing the rule.

Now, again, `s-char-sequence` can either be just a `s-char` or more than one, hence we apply positive enclosure. Finally, our final combination has only got `s-char-sequence`, enclosed in double quotes. Our code looks like:

```
S_CHAR              ({ESCAPE_SEQUENCE})|([^\"\\\n])
S_CHAR_SEQUENCE     {S_CHAR}+

STRING_LITERAL      ([\"])({S_CHAR_SEQUENCE})([\"])
```

For this rule, we set the outcome:

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

Punctuator

Now, we find the punctuators, which will be defined mostly by escape sequences as followed:

```
PUNCTUATOR (\[|\]|\(|\)|\{|\}|&|\*|\+|-|\/|%|!|\?|<|>|<=|>=|==|!=|&&|\|\||=|->|\,|;|\:)
```

which gives:

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

Comment

To label a `comment`, we either need `single-line-comment` or `multi-line-comment`. We write the following:

```
COMMENT         ({SINGLE_LINE_COMMENT})|({MULTI_LINE_COMMENT})
```

For this rule, we set the outcome:

```
{COMMENT}       /* ignore comment */
```

We add this after we define Single Line Comment and Multi Line Comment

Single Line Comment

Since the `single-line-comment` start with `//`, they continue to ignore every character after that until it encounters the newline character. So, we write it as:

```
SINGLE_LINE_COMMENT     (\/\/)([^\n])*
```

Multi Line Comment

Now, in `multi-line-comment`, `/*` is the starting mark and `*/` is the ending mark, while everything, including newline character, is ignored. We have to make sure that `*` is followed by `/`, then only it will end, otherwise, it will keep taking as a comment.

```
MULTI_LINE_COMMENT      (\/\*)([^\*]|\*[^\/])*(\*\/)
```

White Space*

In the end, we define the rules for white spaces, which we first label, as:

```
WHITESPACE  [ \s\t\n]
```

and write the rule:

```
{WHITESPACE}    /* ignore whitespace */
```

NanoC File

To use as a test, we will use the Binary Search code given in Assignment pdf to check our lexer. It is as follows:

```
int arr[10]; // Sorted array to search
/*
A recursive binary search function. It returns location of x
in given array arr[l..r] is present, otherwise -1
*/

int binarySearch(int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l) / 2;
        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;
        // If element is smaller than mid, then it can only be present in left
        ↪  subarray
        if (arr[mid] > x)
            return binarySearch(l, mid - 1, x);
        // Else the element can only be present in right subarray
        return binarySearch(mid + 1, r, x);
    }
    // We reach here when element is not present in array
    return -1;
}
int main()
{
    int n = 5; // Number of elements
    arr[0] = 2;
    arr[1] = 3;
    arr[2] = 4;
    arr[3] = 10;
    arr[4] = 40;
    int x = 10; // Key to search
    int result = binarySearch(0, n - 1, x);
    if (result == -1)
        printStr("Element is not present in array");
    else
    {
        printStr("Element is present at index ");
        printInt(result);
    }
    return 0;
}
```