Project 1: AdvCalc

CMPE 230: Systems Programming, Spring 2023

Yiğit Kağan Poyrazoğlu
Abdullah Umut Hamzaoğulları
1.04.2023

1 Introduction

In this project, we implemented an interpreter for a calculator (AdvCalc) that has the desired functionalities of the assignment using C language and its standard libraries. AdvCalc evaluates expressions and accepts variable assignments.

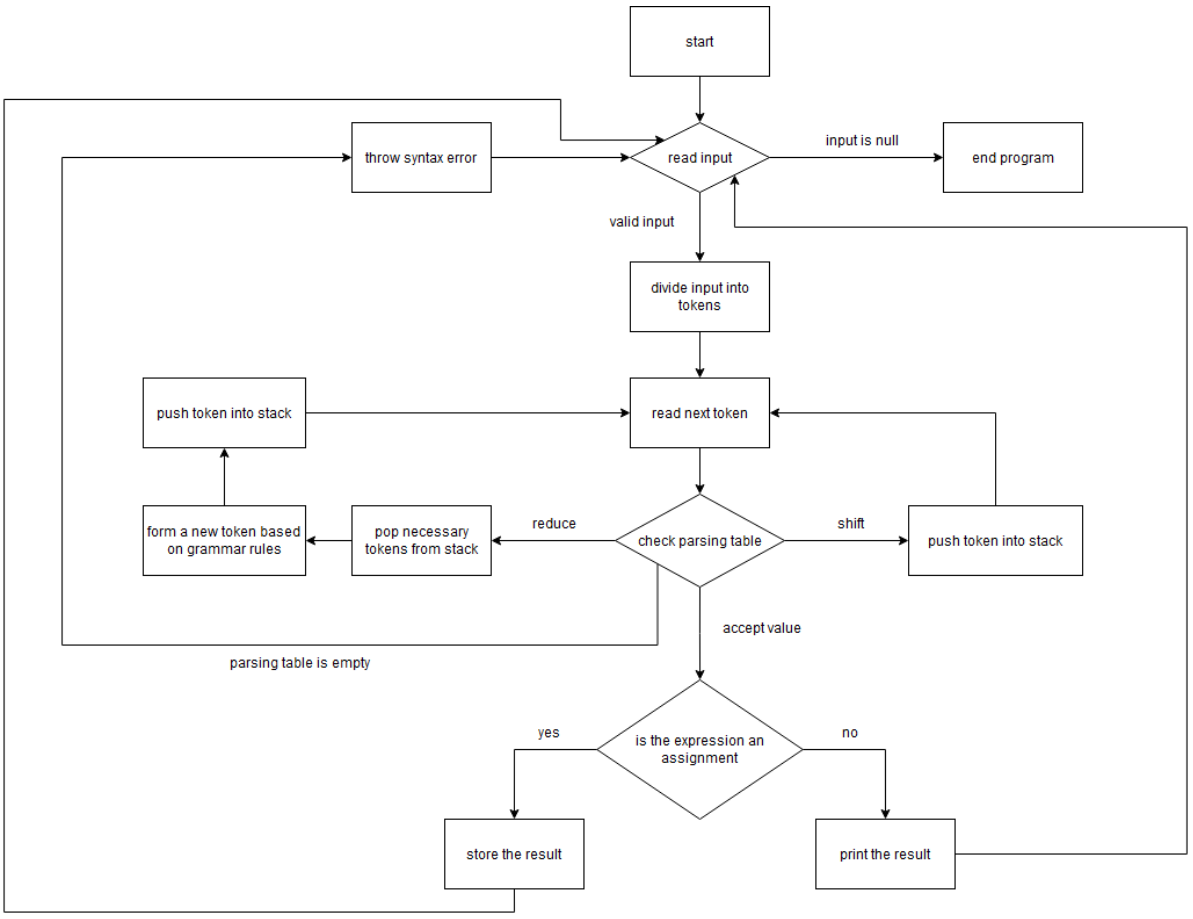
The language implied by the notation is made sense of by explicitly writing out grammar rules, and parsing of any expression is achieved by implementing an LR(1) parser. Input strings are analyzed using regular expressions. Needed data structures like stacks are manually implemented.

2 Design and Architecture

AdvCalc makes use of a primitive parser to parse the input stream. The input as a line is taken from standard input and is divided into a string of tokens and evaluates these tokens from left to right. Based on language grammar, the program then either outputs a result to standard output, or stores the result in a variable (referred to as program memory from now on).

The main structures used throughout the program are a simple hashmap implementation, stacks, and a parsing table (more information is provided below).

Hashmap is used to access program memory. Due to the constraints of the project, program memory is initialized as an array of fixed size and is accessed by a calculation of hash code for the corresponding variable. Stacks are used in parsing the expression tokens. A more detailed functionality is illustrated below with a crude flowchart:



The parsing of tokens is done through a simple LR(1) parser. First, the input string is divided into tokens using regular expressions and then saved in-order as tokens. Then starting from the leftmost token, the tokens are processed in accordance with the states of a parsing table, either pushed to a token stack or reduced according to the rules of the grammar. (For a more detailed explanation, see below.) A correct expression is then expected to reach the end state, and a syntax error will hit an empty spot in the parsing table, thus halting the evaluation.

The PEG (parsing expression grammar) of the language is as follows:

```
S' -> S$
S -> E
S -> V=E
E -> (E)
E -> F(E,E)
E -> not(E)
E -> E&E
E -> E|E
E -> E*E
E -> E+E
E -> E-E
E -> V
E -> I
```

where S' is the starting string,
S is the string non-terminal,
E is the expression non-terminal,
F is the non-unitary function terminal,
V is the variable terminal,
I is the integer terminal,
not is the not function terminal,
&, +, -, |, * are operator terminals,
(,) and "," are symbol terminals.

The theory behind the LR(1) parser is to see our language as a finite-state machine. In our case, that means, in our language, there are finitely next defined behaviors given what had come before, and there are finitely many situations of what may have come before. For example, if what we have is “3 + “, then only another expression or other tokens that will eventually reduce to an expression can come after.

States contain information about the situation of what had come before, and languages are collections of states. For each state, we consider the next token and consider how we should interpret the next token.

There are finite interpretations for a given state, and in this document, the different ways of considering the next token are sometimes referred to as “behaviors”.

LR(1) parser has four behaviors:

1. Reduce according to rule r
2. Shift and go to state s
3. Go to state s
4. Accept

Reduce means to apply the relevant grammatical rule in reverse. For example, in our 5th rule, we define an expression E as F(E,E). Reducing would mean actually computing the result of this function and then labeling the result as just E.

Shifting means reading one more token in our given token list. When we do that, we get to another state of our grammar, how we should consider the next token should be updated, so we also update our state.

Go-to means just updating the state we are in.

Accepting is when we reach the end of the line token. It means to stop parsing and the reduced token is the output of our language.

In this system, when we “update our state”, we cannot simply change our current state, we should also store the previous ones because when we reduce an expression, we will get back to those previous states (from last to first). So they should be stored in a data structure like a stack. Similarly, while reducing,

last-put tokens are taken out and new ones are put in, so they should also be stored in a data structure like a stack as well.

Our states and their associated defined behaviors form a table, called a parsing table.

LR table

State	ACTION														GOTO		
	V	=	(	)	F	,	not	&	|	*	+	-	I	$	S'	S	E
0	s3		s4		s5		s6						s7			1	2
1														acc			
2								s8	s9	s10	s11	s12		r_1			
3		s13						r_{11}	r_{11}	r_{11}	r_{11}	r_{11}		r_{11}			
4	s18		s15		s16		s17						s19				14
5			s20														
6			s21														
7								r_{12}	r_{12}	r_{12}	r_{12}	r_{12}		r_{12}			
8	s23		s4		s5		s6						s7				22
9	s23		s4		s5		s6						s7				24
10	s23		s4		s5		s6						s7				25
11	s23		s4		s5		s6						s7				26
12	s23		s4		s5		s6						s7				27
13	s23		s4		s5		s6						s7				28
14				s29				s30	s31	s32	s33	s34					
15	s18		s15		s16		s17						s19				35
16			s36														
17			s37														
18				r_{11}				r_{11}	r_{11}	r_{11}	r_{11}	r_{11}					
19				r_{12}				r_{12}	r_{12}	r_{12}	r_{12}	r_{12}					
20	s42		s39		s40		s41						s43				38

Part of our parsing table representation. Illustration is done through https://jsmachines.sourceforge.net/machines/lr1.html

Sometimes, there may be conflicting behaviors given a state. This is simply called a conflict. For example, when our expression is “3+4*5” and we have read till “4”, the next token is *, and according to our grammar, we should continue reading to see the end of the operation. But at the same time, we have reached a state where two operands and the operator is read, so again according to our grammar, we should be reducing our expression “3+4”.

In this case, we introduce precedence and choose the behavior associated with the more precedent token, and therefore resolve the conflict. If conflicts existed even after the precedences, then that means we have an ambiguous grammar and should fix our grammar rules.

3 Implementation Details

Token types are stored in an enum and are determined accordingly with the grammar of the language:

```
enum type{
    V,
    EQ,
    OBR,
    CBR,
    F,
    COMM,
    NOT,
    AND,
    OR,
    MULT,
    ADD,
    SUB,
    I,
    EOL,
    Sp,
    S,
    E
};
```

Matching with grammar rules seen in Section 2:

EQ is “=”, OBR is “(“, CBR is “)”, COMM is “, ”, NOT is “not”, AND is “&”, OR is “|”, MULT is “*”, ADD is “+”, SUB is “/”, EOL is “$”, Sp is “S’ ”.

Tokens are stored in a struct. Their values and their types are stored as members of the struct:

```
struct token{
    enum type type;
    char value[TOKEN_SIZE+1];
};
```

Due to limitations of the language and implementation convenience, two stacks were created to perform stack operations on different data types. The first one accepts tokens:

```
struct Node{
    struct token *data;
    struct Node *next;
    int state;
};
```

```
struct Stack{
    struct Node *top;
    int size;
};
```

While specified, state and size variables are never used in practice.

The other one accepts integers:

```
struct intNode{
    int data;
    struct intNode *next;
};
```

```
struct intStack{
    struct intNode *top;
    int size;
};
```

Stacks support simple push, pop, and peek operations. They are also initialized through init() functions.

The stacks are used in parsing, one as a token stack and the other as a state stack. Stacks were implemented in a linked-list style because, in our implementation, there didn't seem to exist a limitation on the number of stacks or the number of tokens that will go into the stacks. While implementations of a single stack are possible. two distinct stacks at the time were more convenient to implement.

Program memory, as stated above, was implemented using a hashmap. Since there is a rather small upper bound on the maximum number of variables, the hashmap is initialized to an array of fixed size. For any variable, the name string is hashed using quadratic rolling and stored accordingly. For any uninitialized space in the array, -1 is placed as a sentinel value (initialized at the beginning of the main function).

```
char *keys[TABLE_SIZE];
long long variables[TABLE_SIZE];
```

```
int toHash(char* string){
    long hash = 0;
    int p = 991;
    int powr = 0;
    while (*string != '\0'){
        hash = hash + ((int) *string)*(power(p, powr));
        string++;
        powr++;
    }
    return hash%TABLE_SIZE;
}
```

3.1 Lexer

The lexer acts on the string of the entire line. As stated above, regular expressions are used to divide the string into tokens, then token structs are initialized and stored.

The regular expression used to match tokens is as follows:

```
[a-zA-Z]+|[0-9]+|[^[:alnum:]]
```

The expression is written in POSIX Extended Regular Expressions syntax. It matches words with uppercase or lowercase letters from the English alphabet, numbers, or non-alphanumeric characters of length 1.

To compile and compare this regular expression, regex.h library is used.

```
regexVal = regcomp(&regex, "[a-zA-Z0-9]+|[^[:alnum:]]", REG_EXTENDED);
```

regcomp() function compiles the regex string to an integer and stores it in a variable (named regex in this instance). Then, regexec() function is run to compare the input string with the regular expression:

```
regexVal = regexec(&regex, msgbuf, 1, match, 0);
```

Here regex is the compiled expression, msgbuf is the input string, and match is the array to which buffers for the matching substring are recorded. Since global flag (\g) is not supported by regex.h library, the function must be executed over and over until there are no more matches:

```
regexVal = regexec(&regex, msgbuf + lastPtr, 1, match, 0);
```

Here lastPtr refers to the end index of the first match. Thus the function is executed on the substring to avoid matching the same expression repeatedly.

After matching an expression, a corresponding token is created. The token has two attributes: type and value. Type is the enum corresponding to the terminal/non-terminal attributed to the token, and value is the string matched by regular expression.

```
      //check for beginning of the comment
const char *comment = "%";
int cmp = strcmp(tokenStr, comment);
if (cmp == 0) break;
//tokenize and add to array
if (!isspace((int) tokenStr[0])) {
    struct token *token = tokenize(tokenStr, len);
    tokens[i] = *token;
    i++;
}
```

```
struct token *tokenize(char *string, int len){
    enum type type = getType(string, len);
    struct token *token = calloc(1, sizeof(struct token));
    token->type = type;
    strncpy(token->value, string, len+1);
    return token;
}
```

Since whitespace is not defined in the enum (or is not present in language grammar) it must be checked explicitly to avoid any token initialization errors. Also, since the comment character was not specified during the building of the grammar, it is explicitly checked. After such checks, the token is added to an array of tokens, preserving the line order. From this array, the tokens are passed to the parser.

3.2 Parser

As stated, LR(1) parser system is used to parse. A generator of the resulting finite state machine (and thus, the resulting table) from the grammar rules is used to get the parsing table, which shows how the next character should be interpreted.[1] There are 80 states and 17 behaviors defined to them. Then, the resulting table was converted to a 3-d array named parsingTable using a simple Python code. The structure of parsingTable is like this:

parsingTable is an array of states.
A state is an array of behaviors each associated with a token.
A behavior is a two-element int array determining what to do next, i.e. to reduce. The following table will show how to understand any behavior array:

First element	Second element	What it means
-1	-1	accept stop parsing and exit the program.
0	0	The behavior is not defined - the language does not allow this input and cannot parse it.
1	s	Shift and go to state s.
2	r	Reduce the expressions according to r'th grammar rule
3	s	Go to state s.

As an example, the first element of the parsingTable, the state 0 is below:

```
{{1, 3}, {0,0}, {1, 4}, {0,0}, {1, 5}, {0,0}, {1, 6}, {0,0}, {0,0}, {0,0}, {0,0},
{0,0}, {1, 7}, {0,0}, {0,0}, {3, 1}, {3, 2}}
```

In our LR(1) implementation, there are two stacks. One stack, named stateStack stores the states as integers, and the other, named tokenStack stores the tokens.

```
struct intStack *stateStack;
struct Stack *tokenStack;
```

In the main loop, after the input is lexed, if we haven't reached the end of the line and if the input string was not empty, we consider the next token and the current state we are in and look at the table for our next move.

```
while (condition) {
   if ((step == 0) && (tokens[0].type == EOL)){
       printf("> ");
       break;
   }
   if (tokenIndex == TOKEN_SIZE-1) {
       //printf("Error!\n");
       break;
   }
   struct token *nextToken;
   if (reduced) {
       nextToken = ((struct token *) peek(tokenStack));
   } else {
       nextToken = &tokens[tokenIndex];
   }

   int type = (*nextToken).type;
   int currentState = i_pop(stateStack);

   int action = parsingTable[currentState][type][0];
   int targetState = parsingTable[currentState][type][1];


   reduced = 0;
   step++;
   switch (action) {
       //accept the statement
       case -1: {
           if (isAssigned == 0) printf("%s\n", ((struct token *)
peek(tokenStack))->value);
           condition = 0;
           printf("> ");
           break;
       }
       //error
       case 0:
           printf("Error!\n");
           printf("> ");
           condition = 0;
           break;
       //shift
       case 1:
           i_push(stateStack, currentState);
           shift(targetState, nextToken);
           nextToken = NULL;
           tokenIndex++;
```

```
            break;
        //reduce
        case 2:
            reduce(targetState);
            reduced = 1;
            break;
        //goto
        case 3:
            i_push(stateStack, currentState);
            goTo(targetState);
            break;
    }
}
```

In our table, there are situations where we shouldn't touch the token stack, but just push a new state to the stateStack, which is called a "go-to" move. It is implemented in the goTo() function as

```
void goTo(int state){
   i_push(stateStack, state);
}
```

shift() function just needs to take in one more token and go-to a state. It is implemented as

```
void shift(int state, struct token *token){
   struct token *newtoken = token;
   push(tokenStack, newtoken);
   token = NULL;
   goTo(state);
}
```

reduce() function takes in the rule we should reduce our expression according to. Depending on the rule, it pops out the correct amount of tokens from the tokenStack. Then, it either changes the token type of a token or creates a new kind of token which is the reduced expression. (As discussed in Section 2, the rule is applied backward when reducing.)

Here is an example of reducing according to the first rule of our grammar:

```
void reduce(int rule){
   switch (rule){
       case 0:{
           struct token* token = (struct token*) pop(tokenStack);
           token->type = Sp;
           push(tokenStack, token);
           break;
       }
```

When the rule is about functions that take two inputs, we send the necessary information to another function called evaluate(). A similar case is with binary operations and the arithmetic() function.

```
 case 4:{
            pop(tokenStack);
            struct token* rightOperand = (struct token*) pop(tokenStack);
            pop(tokenStack);
            struct token* leftOperand = (struct token*) pop(tokenStack);
            pop(tokenStack);
            struct token* function = (struct token*) pop(tokenStack);
            char value[TOKEN_SIZE+1];
            sprintf(value, "%lld", evaluate(function, leftOperand, rightOperand));
            int len = strlen(value);
            struct token *newtoken = tokenize(value, len);
            newtoken->type = E;
            push(tokenStack, newtoken);
            i_pop(stateStack);
            i_pop(stateStack);
            i_pop(stateStack);
            i_pop(stateStack);
            i_pop(stateStack);
            break;
         }
```

evaluate and arithmetic functions do the necessary operations and return the result. They have similar structures, and here is the evaluate function:

```
long long evaluate(struct token* function, struct token* leftoperand, struct
token* rightoperand){
   long long leftVal = strtoll(leftoperand->value, NULL, 10);
   if (rightoperand == NULL){
       return ~leftVal;
   }
   long long rightVal = strtoll(rightoperand->value, NULL, 10);

   long long result;
   switch (getFunction(function->value)){
       case 0:
           break;
```

```
            case 1:
                result = leftVal ^ rightVal;
                break;
            case 2:
                result = leftVal << rightVal;
                break;
            case 3:
                result = (leftVal << rightVal)|(leftVal >> (64 - rightVal));
                break;
            case 4:
                result = leftVal >> rightVal;
                break;
            case 5:
                result = (leftVal >> rightVal)|(leftVal << (64 - rightVal));
                break;
        }
        return result;
    }
```

With the structures described above, tokens are parsed and eventually are reduced to a single expression, and that is the moment our table points us to “accept”, and finish parsing. If the tokens contained undefined behaviors in our language, then a syntax error is printed instead.

4 Testing and Validation

Since the grammar is almost completely specified, most syntax errors do not even need separate handling. During testing, three cases were found to be uncovered by the specified grammar:

-empty line/whitespace line
-commenting
-whitespaces as tokens

Empty lines are checked before beginning the parsing process. The handling of the other two cases is discussed above.

While the program does not cover a broad possibility of faulty inputs (such as Unicode incompatibility), the assignment clarifies that the program will not be tested with such dirty inputs. For any input with which the program will be tested, it is expected to run without any problems.

Similarly, the code has the same limitations of tokens and variables as the

assignment's upper limit for those.

Testing is done as with the example inputs of the assignments and the possible edge cases that were posted in the course's forum, and the outputs were identical to what they should be.

5 Usage and Examples

The usage is complicit with the assignment's requirements. How it should be compiled is determined with a makefile,

```
AdvCalc: main.o lexer.o stacks.o
   gcc main.o lexer.o stacks.o -o AdvCalc

main.o: main.c lexer.h stacks.h
   gcc -c main.c

lexer.o: lexer.c lexer.h
   gcc -c lexer.c

stacks.o: stacks.c stacks.h
   gcc -c stacks.c
```

Below are some example usages from the terminal:

```
ultiminati@Ultiminati:/mnt/c/Users/omen/ClionProjects/CMPE230_Project1$ make
gcc -c main.c
gcc -c lexer.c
gcc -c stacks.c
gcc main.o lexer.o stacks.o -o AdvCalc
ultiminati@Ultiminati:/mnt/c/Users/omen/ClionProjects/CMPE230_Project1$ ./AdvCalc
> 14+24*46
1118
> 4 & 5
4
> x + + 1
Error!
> xor(111,222)
177
> x = xor(111,222)
```

6 Conclusion

The first project of the Systems Programming course, AdvCalc, is an interpreter based on a compact grammar. By first specifying the grammar, then generating the parsing table and constructing the lexer and parser algorithms, it is implemented in a holistic, theory-compliant manner to provide insight into compiler/interpreter design and language parsing.

Throughout this project, our skills in C programming were greatly improved, and we believe that we have learned a lot about methods of string parsing, interpreter design, and as extra, finite state machines.