Örebro University

Compilers and Interpreters

Assignment 2

Author:

Linus Baumgärtner
lbaumgaertner@culba.de

September 23, 2022

Part A

The grammar for both assignments and expressions can be seen in figure 1. It is not possible to write a predictive recursive-descent parser for this grammar because there would be a FIRST conflict. Because FIRST(assignment) and FIRST(expr) both can start with id, the parser can not tell apart which of the non-terminals to use.

```
start -> list eof
list -> assignment ; list
      | expr ; list
      | empty
assignment -> id { print(id.lexeme) } = expr { print('=') }
expr -> term moreterms
moreterms -> + term { print('+') } moreterms
      | - term { print('-') } moreterms
      | empty
term -> factor morefactors
morefactors -> * factor { print('*') } morefactors
      | / factor { print('/') } morefactors
      | div factor { print('DIV') } morefactors
      | mod factor { print('MOD') } morefactors
      | empty
factor -> ( expr )
      | id { print(id.lexeme) }
      | num { print(num.value) }
```

Figure 1: Grammar for both assignments and expressions

Part B

To modify the parser, so that it understands the grammar with assignments first there has to be a assignment function added to the parser. This function works like the described in the grammar. Then, instead of the expr function the assignment function has to be called in list. This changes can be seen in listing 1.

Listing 1: Modificated version of the parser

```
1  void list()
2  {
3      if (lookahead == ID) {
4          assignment(); match(';'); list();
5      }
6      else {
7          /* Empty */
8      }
9  }
10
11 void assignment()
12 {
13     if (lookahead == ID) {
14         int id_number = token_value;
15         match(ID); emit(ID, id_number);
16         match('=');
17         expr();
18         emit('=', token_value);
19     } else {
20         error ("syntax error in assignment");
21     }
22 }
```

Part C

To calculate the results in the file *calc.c* a stack was implemented with which the operations can be executed. The file can be seen in listing 2. It is called after each token is processed by the parser. If the token is a number is pushed onto the stack, if the token is an operation the two top values of the stack are used for the calculation and the result pushed to the stack. If the token is a variable, there are two cases how this is processed. If the stack is empty, the id of the variable in the symtable is pushed onto the stack. Else the value of the variable in the symtable is pushed onto the stack. For the equals operator the top value in the stack is popped and written into the symtable of the tokens id.

Listing 2: calc.c

```c
#include "global.h"

int top = -1; /* top of stack */
int stack[N];

void calc(int token_type, int token_value), push(int t);
int pop();

int assignment_token_id;

void calc (int token_type, int token_value)  /* calculates expressions */
{
    int x;
    switch(token_type) {
    case '+' :
        x = pop() + pop(); push(x); break;
    case '-' :
    {
        int a, b;
        a = pop(); b = pop(); x = b - a;
        push(x); break;
    }
    case '*' :
        x = pop() * pop(); push(x); break;
    case '/':
    {
        int a, b;
        a = pop(); b = pop(); x = b / a;
        push(x);
        break;
    }
    case '^':
    {
        int i, a, b;
        a = pop(); b = pop(); x = 1;
        for (i=0; i<a; i++) {
            x *= b;
        }
        push(x); break;
    }
    case '=':
    {
        int token_id;
        // pop expression value;
        x = pop();
        printf("%d\n", x);

        // pop variable symtable id;
        token_id = pop();

```

```
        symtable[token_id].value = x;
        symtable[token_id].value_initialized = true;
        break;
    }
    case NUM:
        push(token_value); break;
    case ID:
    {
        if (top == -1) {
            // for first variable push id into stack
            push(token_value);
        } else {
            // for all other variables check if value is initialized else error()
            if (symtable[token_value].value_initialized) {
                // push value of id onto stack;
                push(symtable[token_value].value);
            } else {
                error("Variable not initialized.");
            }
        }
        break;

    }
    default:
        printf("[Unknown token %d, with value %d]\n", token_type, token_value);
    }
}

void push(int t) {
    if (top == N-1) {
        error("Stack overflow");
    } else {
        top += 1;
        stack[top] = t;
    }
}

int pop(){
    if(top == -1)
        printf("Underflow State: Stack already empty, can't remove any element\n");
    else{
        int x = stack[top];
        top-=1;
        return x;
    }
    return -1;
}
```

Part D

First, to include exponentials we change the grammar to:

```
start -> list eof
list -> assignment ; list
      | empty
assignment -> id { print(id.lexeme) } = expr { print('=') }
expr -> term moreterms
moreterms -> + term { print('+') } moreterms
      | - term { print('-') } moreterms
      | empty
term -> exponential morefactors
morefactors -> * exponential { print('*') } morefactors
      | / exponential { print('/') } morefactors
      | div exponential { print('DIV') } morefactors
      | mod exponential { print('MOD') } morefactors
      | empty
exponential -> factor moreexponential
moreexponential -> ^ factor { print('^') } moreexponential
factor -> ( expr )
      | id { print(id.lexeme) }
      | num { print(num.value) }
```

This can be implemented by creating the two functions *exponential* and *moreexponential* that are called instead of *factor* in the *term* and *morefactors* function. Also in the *calc* function the case for the exponential has to be added.

Listing 3: exponential and moreexponential functions

```
1  void exponential()
2  {
3      factor(); moreexponential();
4  }
5
6  void moreexponential()
7  {
8      if (lookahead == '^') {
9          match('^');
10         factor();
11         emit('^', token_value);
12         moreexponential();
13     } else {
14     }
15 }
```