Compiler Design Lab (CS 511)

Autumn 2021

Assignment 4

Consider the following set of fully parenthesized expressions over non-negative integers with addition (+) and multiplication (∗) operators, and reading input (r) from the stdin.

1. Every non-negative integer (32-bit) is an expression.

2. r is an expression. Its value is the integer read from the stdin.

3. If e1 and e2 are expressions, then so are (e1 + e2) and (e1 ∗ e2).

4. Nothing else is an expression.

As an example, the expression ((12+r)\*5) on **input** 3 from stdin is evaluated to 75. Every expression is followed by EOF.  The input-output looks as follows.

$ ./a.out

((12+r)\*5)

:3

Value: 75

Write a C program to implement a scanner, a parser and an interpreter for such expressions. The parser builds an abstract syntax tree (AST) of the input expression. The interpreter evaluates the expressions by inorder traversal on the AST. You may follow the following instructions. You are not allowed to use any available software or library for scanner, parser or interpreter.

1. In the scanner there are two files lex.h and lex.c. The header may be as follows:

*// lex.h the header file for the scannar*

*#include <stdio.h>*

*#define END 256*

*#define NUM 257*

*typedef struct { int tokenClass; int val; } token\_t;*

*extern token\_t token;*

*extern void getNextToken(void);*

A token is of type token\_t with two fields. The **tokenClass** has values END (end-of-file), NUM (non-negative integer) or ASCII code of any other character except white space (blank, \n, \t), which are ignored. The **val** is used to store the value of a number (tokenClass = NUM). The function getNextToken(), when called by the parser, updates the content of token with the next token value before it is accessed by the parser.

2. In the parser also there are two files parser.h and parser.c. The header of parser.h may be as follows:

*// parse.h header file for the parser*

*#ifndef PARSER\_H*

*#define PARSER\_H*

*typedef struct node {*

*char type; // I: internal, D: data, R: read unsigned*

*int val; // for a node of type D*

*struct node \*left, \*right; // pointers to left and right subtrees for a node of type I*

*char op; // the operator in node type I*

*} ASTnode\_t;*

*extern int parser(ASTnode\_t \*\*); // returns 1 on success, returns 0 on failure.*

*#endif*

ASTnode\_t is the type of every AST node. The field type indicates the type of a node - internal node (I) with an operator (op), a pointer to the left sub-expression tree (left) and a pointer to the right sub-expression tree (right). Leaf node (D) with data in val or a leaf node (R) to read a data from the stdin.

The main parser function int parser(ASTnode\_t \*\*tpp), defined in parser.c, takes a pointer to pointer to an AST node (tpp) as argument and returns 1 when the AST of the expression is constructed successfully and pointed by \*tpp. Otherwise it returns 0. This function calls scanner function void getNextToken(void) when the next token is required.

Parsing an expression is done by the recursive function int parseExp(ASTnode\_t \*\*tpp). Following is the outline of its definition. The pointer \*tpp is the address of root of the AST (if successfully created). The return value is for success or failure.

* If the next-token is NUM, a leaf-node of type ‘D’ is created with the value of the number.
* If the next token is ‘r’, a leaf node of type ‘R’ is created that will be subsequently used by the interpreter to read data from stdin.
* If the next token is ‘(’, an internal node of type ‘I’ is created for expression of the form (e1 + e2) or (e1 ∗ e2). The left and right subtrees corresponding to e1 and e2 are created by calling parseExp() recursively. The left and the right pointers are updated.

When the token for the ‘+’ or ‘\*’ is encountered, it is put in the op field of the internal node. The final ‘)’ completes the expression.

3. The function main() calls int parser(ASTnode\_t \*\*tpp) with pointer to pointer to an AST node as argument. If the return value is 1 i.e. a successful construction of the AST, it calls the backend interpreter.

4. The backend function present in backend.c takes the pointer to the AST as argument and interprets it by traversing the tree inorder. Its header is available in backend.h

5. There are several files to compile. So it is necessary to prepare a Makefile as follows:

*objfiles = main.o parser.o lex.o backend.o*

*a.out: $(objfiles)*

*cc $(objfiles)*

*main.o: main.c*

*cc -c -Wall main.c*

*parser.o: parser.c*

*cc -c -Wall parser.c*

*lex.o: lex.c*

*cc -Wall -c lex.c*

*backend.o: backend.c*

*cc -Wall -c backend.c*

*clean :*

*rm a.out $(objfiles)*