Örebro University

Compilers and Interpreters

Assignment 5

Authors:

Linus Baumgärtner
lbaumgaertner@culba.de

Marko Reichhart
marko.reichhart@web.de

October 20, 2022

Part A

In listing 1 the grammar for a list of statements can be seen. The previous grammar was extended by the non-terminals *stmt* and *stmt_list*. The functions that were suggested in the lab were used to create the syntax tree. To connect the statements a semicolon node is introduced.

Listing 1: Grammar for list of statements

```
start:
    list DONE           { print_syntax_tree($1); }
    ;
list:
      stmt list         { $$ = mknode(';', $1, $2); }
    |                   { $$ = mknode(' '); }
    ;
stmt:
      ';'                                           { $$ = mknode(';'); }
    | expr ';'                                      { $$ = $1; }
    | ID '=' expr ';'                               { $$ = mknode('=', mkleaf(ID, $1), $3); }
    | PRINT '(' ID ')' ';'                          { $$ = mknode(PRINT, mkleaf(ID, $3)); }
    | READ '(' ID ')' ';'                           { $$ = mknode(READ, mkleaf(ID, $3)); }
    | WHILE '(' expr ')' stmt                       { $$ = mknode(WHILE, $3, $5); }
    | IF '(' expr ')' stmt %prec IFX                { $$ = mknode(IF, $3, $5); }
    | IF '(' expr ')' stmt ELSE stmt                { $$ = mknode(IF, $3, $5, $7); }
    | '{' stmt_list '}'                             { $$ = $2; }
    | '{' '}'                                       { $$ = mknode(' ');}
    ;
stmt_list:
      stmt              { $$ = $1; }
    | stmt_list stmt    { $$ = mknode(';', $1, $2); }
    ;
expr :
      NUM                     { $$ = mkleaf(NUM, $1); }
    | ID                      { $$ = mkleaf(ID, $1); }
    | expr '+' expr           { $$ = mknode('+', $1, $3); }
    | expr '-' expr           { $$ = mknode('-', $1, $3); }
    | expr '*' expr           { $$ = mknode('*', $1, $3); }
    | expr '/' expr           { $$ = mknode('/', $1, $3); }
    | expr '%' expr           { $$ = mknode('%', $1, $3); }
    | expr '^' expr           { $$ = mknode('^', $1, $3); }
    | expr '&' expr           { $$ = mknode('&', $1, $3); }
    | expr '|' expr           { $$ = mknode('|', $1, $3); }
    | expr '<' expr           { $$ = mknode('<', $1, $3); }
    | expr '>' expr           { $$ = mknode('>', $1, $3); }
    | '(' expr ')'            { $$ = $2; }
    | expr '?' expr ':' expr  { $$ = mknode('?', $1, $3, $5); }
    ;

%%
```

Part B

To print the tree, the recursive printout function from the lab was used and extended by all occurring cases and can be seen in listing 2. The function is called when the program is aborted by ctrl+d.

Listing 2: Syntax tree printout function

```
void print_syntax_tree(TreeNode* p) {
  printf("\nSyntax tree:\n");
  printtree1(p, 0);
}
```

```c

void printtree1(TreeNode* p, int level) {
    if (p==0)
        return;
    printf("%*s", level, "");
    switch (p->type) {
        case ' ':
            printf("{}\n");
            break;
        case ID:
            printf("%s\n", symtable[p->leaf_value].lexeme);
            break;
        case NUM:
            printf("%d\n", p->leaf_value);
            break;
        case '=':
            printf("=\n");
            printtree1(p->args[0], level + 4);
            printtree1(p->args[1], level + 4);
            break;
        case IF:
            printf("if\n");
            printtree1(p->args[0], level + 4);
            printtree1(p->args[1], level + 4);
            printtree1(p->args[2], level + 4);
            break;
        case WHILE:
            printf("while\n");
            printtree1(p->args[0], level + 4);
            printtree1(p->args[1], level + 4);
            break;
        case PRINT:
            printf("print\n");
            printtree1(p->args[0], level + 4);
            break;
        case READ:
            printf("read\n");
            printtree1(p->args[0], level + 4);
            break;
        case '+':  case '-': case '*': case '/': case '%': case '^': case '&': case '|': case '<': case '>':
            printf("%c\n", p->type);
            printtree1(p->args[0], level + 4);
            printtree1(p->args[1], level + 4);
            break;
        case '?':
            printf("%c\n", p->type);
            printtree1(p->args[0], level + 4);
            printtree1(p->args[1], level + 4);
            printtree1(p->args[2], level + 4);
            break;
        case ';':
            printf(";\n");
            printtree1(p->args[0], level + 4);
            if (p->args[1] != 0) {
                printtree1(p->args[1], level + 4);
            }
            break;
    }
}
```

Part C

The implementation of the statements if, while, print and read can be seen in listing 1 and listing 2. They are inserted into the grammar as an option for the non-terminal *stmt*.