Örebro University

Compilers and Interpreters

# Assignment 7

*Author:*

Linus Baumgärtner
lbaumgaertner@culba.de

Marko Reichhart
marko.reichhart@web.de

October 20, 2022

# Part A

The function `push_to_stack` as seen in listing 1 generates the stack machine code recursively according to the type of the node. The tree is thereby traversed so that the children of a node are evaluated before the node itself, so that we build a postfix code. The function is called by the function `execute_in_stack_machine`, which is called by the input of the statement `!exe`.

**Listing 1: Generating stack machine code**

```
void execute_in_stack_machine(TreeNode* p) {
    push_to_stack(p);
    printf("\n");
    sm.append(Instruction(halt));
    sm.showstate();
    sm.list_program();
    sm.set_trace(1);
    sm.run();
}

void push_to_stack(TreeNode* p) {
    if (p == 0)
        return;

    switch (p->type) {
        case ';':
            push_to_stack(p->args[0]);
            push_to_stack(p->args[1]);
            return;
        case ' ':
            return;
        case NUM:
            printf("%d", p->leaf_value);
            sm.append(Instruction(push, p->leaf_value));
            return;
        case ID:
            printf("%s", symtable[p->leaf_value].lexeme);
            sm.append(Instruction(rvalue, p->leaf_value));
            return;
        case '=':
            sm.append(Instruction(lvalue, p->args[0]->leaf_value));
            printf("%s", symtable[p->args[0]->leaf_value].lexeme);
            push_to_stack(p->args[1]);

            sm.append(Instruction(assign));
            printf("%c", p->type);
            return;
        case READ:
        {
            sm.append(Instruction(lvalue, p->args[0]->leaf_value));
            sm.append(Instruction(stackop_read));
            sm.append(Instruction(assign));
            return;
        }
        case PRINT:
            sm.append(Instruction(rvalue, p->args[0]->leaf_value));
            sm.append(Instruction(stackop_write));
            return;
        case '?':
        case IF:
        {
            int label1 = lowest_label;
            int label2 = lowest_label+1;
```

```
            lowest_label+=2;
            push_to_stack(p->args[0]);                          // push condition
            sm.append(Instruction(gofalse, label1));      // jumpfalse to label 1
            push_to_stack(p->args[1]);                          // push if part
            sm.append(Instruction(jump, label2));       // jump to label 2 over else part
            sm.append(Instruction(label, label1));        // label 1
            push_to_stack(p->args[2]);                          // push else part
            sm.append(Instruction(label, label2));      // label 2
            return;
        }
        case WHILE:
        {
            int label1 = lowest_label;
            int label2 = lowest_label+1;
            lowest_label+=2;
            sm.append(Instruction(label, label1));        // label 1
            push_to_stack(p->args[0]);                          // push condition
            sm.append(Instruction(gofalse, label2));    // jumpfalse to label 2
            push_to_stack(p->args[1]);                          // push statement part
            sm.append(Instruction(jump, label1));         // jump to label 1
            sm.append(Instruction(label, label2));      // label 2
            lowest_label++;
            return;
        }
    }

    push_to_stack(p->args[0]);
    push_to_stack(p->args[1]);
    push_to_stack(p->args[2]);
    push_to_stack(p->args[3]);
    printf("%c", p->type);
    switch (p->type) {
        case '+':
            sm.append(Instruction(plus));
            break;
        case '-':
            sm.append(Instruction(minus));
            break;
        case '*':
            sm.append(Instruction(times));
            break;
        case '/':
            sm.append(Instruction(divide));
            break;
        case '%':
            sm.append(Instruction(modulo));
            break;
        case '&':
            sm.append(Instruction(stackop_and));
            break;
        case '|':
            sm.append(Instruction(stackop_or));
            break;
        case '<':
            sm.append(Instruction(lt));
            break;
        case '>':
            sm.append(Instruction(gt));
            break;
    }

}
```

# Part B

Listing 2 shows the `optimize` function which recursively traverses the tree while building a second tree that simplifies some constant operations or useless statements like $5+0$ by already evaluating this part of the tree and substituting it by a leaf node. Also if operations with constant conditions are evaluated and only the statement that will be called is returned as a node. The optimize function can also be called multiple times as seen in the `execute_in_stack_machine` function. It is placed before creating the stack machine code with `push_to_stack`.

**Listing 2: Optimizing of the syntax tree**

```
void execute_in_stack_machine(TreeNode* p) {
    TreeNode* opt_p1 = optimize(p);
    TreeNode* opt_p2 = optimize(opt_p1);
    TreeNode* opt_p3 = optimize(opt_p2);
    push_to_stack(opt_p3);
    printf("\n");
    sm.append(Instruction(halt));
    sm.showstate();
    sm.list_program();
    sm.set_trace(1);
    sm.run();
}

TreeNode* optimize(TreeNode* p) {
    if (p == 0)
        return 0;

    switch (p->type) {
        case NUM:
        case ID:
            return cpnode(p);
        case '+': case '-':
            if (p->args[0]->type == NUM && p->args[0]->leaf_value == 0) {
                return cpnode(p->args[1]);
            } else if (p->args[1]->type == NUM && p->args[1]->leaf_value == 0) {
                return cpnode(p->args[0]);
            } else if (p->args[0]->type == NUM && p->args[1]->type == NUM) {
                return mkleaf(NUM, execute(p));
            }
            break;
        case '*':
            if (p->args[0]->type == NUM && p->args[0]->leaf_value == 1) {
                return cpnode(p->args[1]);
            } if (p->args[1]->type == NUM && p->args[1]->leaf_value == 1) {
                return cpnode(p->args[0]);
            } else if (p->args[0]->type == NUM && p->args[1]->type == NUM) {
                return mkleaf(NUM, execute(p));
            }
            break;
        case '/':
            if (p->args[1]->type == NUM && p->args[1]->leaf_value == 1) {
                return cpnode(p->args[0]);
            } else if (p->args[0]->type == NUM && p->args[1]->type == NUM) {
                return mkleaf(NUM, execute(p));
            }
            break;
        case '%':
            if (p->args[1]->type == NUM && p->args[1]->leaf_value == 1) {
                return mkleaf(NUM, 0);
            } else if (p->args[0]->type == NUM && p->args[1]->type == NUM) {
                return mkleaf(NUM, execute(p));
            }
```

```
53              break;
54          case '?':
55          case IF:
56          {
57              TreeNode* cond = p->args[0];
58              if (cond->args[0]->type == NUM && cond->args[1]->type == NUM) {
59                  int condition = execute(cond->args[0]);
60                  if (condition)
61                      return cpnode(p->args[1]);
62                  else
63                      return cpnode(p->args[2]);
64              }
65              break;
66          }
67
68      }
69      TreeNode* new_node = new TreeNode();
70      new_node->type = p->type;
71      for (int i=0; i<4; i++) {
72          new_node->args[i] = optimize(p->args[i]);
73      }
74      return new_node;
75
76  }
```