ÖREBRO UNIVERSITY

COMPILERS AND INTERPRETERS

Assignment 7

Author:

Linus Baumgärtner lbaumgaertner@culba.de

Marko Reichhart marko.reichhart@web.de

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);
2
       printf("\n");
3
       sm.append(Instruction(halt));
4
5
       sm.showstate();
6
       sm.list_program();
       sm.set_trace(1);
8
       sm.run();
   }
9
10
   void push_to_stack(TreeNode* p) {
11
       if (p == 0)
12
           return;
13
14
       switch (p->type) {
15
            case ';':
16
                push_to_stack(p->args[0]);
17
                push_to_stack(p->args[1]);
18
19
                return;
            case ' ':
20
21
                return;
            case NUM:
22
                printf("%d", p->leaf_value);
23
                sm.append(Instruction(push, p->leaf_value));
24
                return;
25
            case ID:
26
                printf("%s", symtable[p->leaf_value].lexeme);
27
                sm.append(Instruction(rvalue, p->leaf_value));
28
                return;
           case '=':
30
                sm.append(Instruction(lvalue, p->args[0]->leaf_value));
31
                printf("%s", symtable[p->args[0]->leaf_value].lexeme);
32
                push_to_stack(p->args[1]);
33
34
                sm.append(Instruction(assign));
35
                printf("%c", p->type);
36
                return;
37
            case READ:
38
                sm.append(Instruction(lvalue, p->args[0]->leaf_value));
40
                sm.append(Instruction(stackop_read));
41
                sm.append(Instruction(assign));
42
                return;
43
           }
44
45
                sm.append(Instruction(rvalue, p->args[0]->leaf_value));
46
                sm.append(Instruction(stackop_write));
47
                return;
48
           case '?':
49
            case IF:
50
51
                int label1 = lowest_label;
52
                int label2 = lowest_label+1;
53
```

```
lowest_label+=2;
54
                 push_to_stack(p->args[0]);
                                                                        // push condition
55
                 sm.append(Instruction(gofalse, label1));
                                                                  // jumpfalse to label 1
56
                 push_to_stack(p->args[1]);
                                                                        // push if part
                 sm.append(Instruction(jump, label2));
                                                                // jump to label 2 over else part
58
                 sm.append(Instruction(label, label1));
                                                                  // label 1
59
                 push_to_stack(p->args[2]);
                                                                        // push else part
60
                 sm.append(Instruction(label, label2));
                                                                // label 2
61
                 return;
62
             }
63
             case WHILE:
64
65
             {
                 int label1 = lowest_label;
66
                 int label2 = lowest_label+1;
67
                 lowest_label+=2;
                 sm.append(Instruction(label, label1));
                                                                  // label 1
69
                 push_to_stack(p->args[0]);
                                                                        // push condition
70
                 sm.append(Instruction(gofalse, label2));
                                                                // jumpfalse to label 2
71
                 push_to_stack(p->args[1]);
                                                                        // push statement part
72
                 sm.append(Instruction(jump, label1));
                                                                  // jump to label 1
73
                 sm.append(Instruction(label, label2));
                                                                // label 2
74
                 lowest_label++;
75
                 return;
76
            }
77
        }
78
79
        push_to_stack(p->args[0]);
80
        push_to_stack(p->args[1]);
81
        push_to_stack(p->args[2]);
82
        push_to_stack(p->args[3]);
83
        printf("%c", p->type);
84
85
        switch (p->type) {
             case '+':
86
87
                 sm.append(Instruction(plus));
88
                 break;
            case '-':
89
                 sm.append(Instruction(minus));
90
91
                 break:
            case '*':
92
                 sm.append(Instruction(times));
93
                 break:
94
             case '/':
95
                 sm.append(Instruction(divide));
96
                 break;
97
             case '%':
98
                 sm.append(Instruction(modulo));
100
                 break;
             case '&':
101
                 sm.append(Instruction(stackop_and));
102
                 break;
103
             case '|':
104
                 sm.append(Instruction(stackop_or));
105
                 break;
106
             case '<':
107
                 sm.append(Instruction(lt));
108
                 break;
109
             case '>':
110
                 sm.append(Instruction(gt));
111
                 break;
112
113
        }
114
115 }
```

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);
2
        TreeNode* opt_p2 = optimize(opt_p1);
3
        TreeNode* opt_p3 = optimize(opt_p2);
4
        push_to_stack(opt_p3);
5
        printf("\n");
6
        sm.append(Instruction(halt));
7
8
        sm.showstate();
9
        sm.list_program();
10
        sm.set_trace(1);
11
        sm.run();
   }
12
13
   TreeNode* optimize(TreeNode* p) {
14
        if (p == 0)
15
            return 0;
16
17
        switch (p->type) {
18
            case NUM:
19
            case ID:
                return cpnode(p);
            case '+': case '-':
22
                if (p-\rangle args[0]-\rangle type == NUM && p-\rangle args[0]-\rangle leaf_value == 0) {
23
                     return cpnode(p->args[1]);
24
                } else if (p->args[1]->type == NUM && p->args[1]->leaf_value == 0) {
25
                     return cpnode(p->args[0]);
26
                } else if (p->args[0]->type == NUM && p->args[1]->type == NUM) {
27
                     return mkleaf(NUM, execute(p));
28
29
                break;
30
            case '*':
31
                if (p->args[0]->type == NUM && p->args[0]->leaf_value == 1) {
32
                     return cpnode(p->args[1]);
33
                } if (p->args[1]->type == NUM && p->args[1]->leaf_value == 1) {
34
                     return cpnode(p->args[0]);
35
                } else if (p->args[0]->type == NUM && p->args[1]->type == NUM) {
36
                     return mkleaf(NUM, execute(p));
37
38
                break;
39
            case '/':
40
                if (p->args[1]->type == NUM && p->args[1]->leaf_value == 1) {
41
                     return cpnode(p->args[0]);
42
                } else if (p->args[0]->type == NUM && p->args[1]->type == NUM) {
43
                     return mkleaf(NUM, execute(p));
44
45
                break;
46
            case '%':
47
                if (p->args[1]->type == NUM && p->args[1]->leaf_value == 1) {
48
                     return mkleaf(NUM, 0);
49
                } else if (p-\rangle args[0]-\rangle type == NUM && p-\rangle args[1]-\rangle type == NUM) {
50
                     return mkleaf(NUM, execute(p));
51
52
```

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