Criterion C

Word count: 974

Contents

1 Lexer

Lexer is the part of interpreter that converts text to tokens. Token is a structure holding a string and identifier that is later used in parsing. General idea behind lexer has been shown in criterion B. Below is a concrete example of lexer working from source code showing lexical analysis for variables, mathod names and reserved keywords.

```
tk::Token &Lexer::id()
{
    int id = tk::ID_VAR;
    attr_buffer.push_back(c);

    if(!is_upcase(c)) id = tk::ID_METHOD;

    advance();

    while(std::isalnum(c) || c == '_')
    {
        if(!is_upcase(c)) id = tk::ID_METHOD;

        attr_buffer.push_back(c);
        advance();
    }
    if(tk::lookup_keyword(attr_buffer) > 0)
        id = tk::lookup_keyword(attr_buffer);

    token.mutate(id, attr_buffer, line_num);
    return token;
}
```

The starting point for this lexical analysis of id's are regular expressions for them.

```
ID : [A-Z]([A-Z] | [0-9] | _)*
METHOD_ID : ([A-Z] | [a-z]) ([A-Z] | [a-z] | [0-9] _)*
```

The function holds a state (`id`), which is the identifier for the token returned by the method. The starting state is variable id, because it is a subset of method id, as it is easier to change from subset to superset than the other way around. The loop iterates over the string buffer until it detects character that is not defined for method id. After the loop, the token buffer is checked whether it is a reserved keyword. Method `lookup_keyword` searches a map, whose key is a string, holding identification for reserved keyword. Then token is changed to contain appropriate data and its address in stack is returned.

2 Recursive-descend parser

The role of parser is to make a representiation of source code that can later be processed by the intepreter. In this case parser generates abstract syntax tree (AST) from tokens (terminals). Example of AST has been shown in criterion B. The blueprint for parser is grammar, in this example, grammar for expressions.

```
expr    :: term + term
         | term - term
         | term
```

In recursive-descend parser, each nonterminal has its own method, which is responsible for building a subtree expressing a bit of syntax in the main AST. Below, is such method for above-mentioned expression.

```
ast::AST *Parser::expr()
{
    ast::AST *root, *new_node;
    root = term();

    while(token.id == tk::PLUS
            || token.id == tk::MINUS)
    {
        new_node = new ast::AST(token, ast::BINOP);
        new_node->push_child(root);
        root = new_node;
        eat(token.id);
        new_node->push_child(term());
    }

    return root;
}
```

We start by declaring two variables that will hold poitners to AST nodes. We assign *root* pointer to *term*, as it is the first nonterminal that has to be evaluated in each of cases in the grammar. Then, if the next token is neither *PLUS*, nor *MINUS*, we return the node. In other case we enter the while loop that will turn as long as the token during the check is *PLUS* or *MINUS*. Inside the loop, we assign a pointer to a subroot to variable *new_node*, which is the terminal of binary operation holding the operator. The node currently held by *root* is the left operand of binary operation and is linked to the root by funtion *push_child* that appends the pointer to the vector inside the node holding the children. After *root* reclaiming the root of the tree, we advance to the next token using method *eat*, which checks whether the current token is the same as requested token by a method. Because we already checked if the token is correct, we just pass the id of current token. Lastly, we append another *term* to our root and either repeat the process or break out of the loop and return the poitner to the tree. This method, along with helper functions *term* and *factor* provided an example expression $10 + 100 * 10 - 6$ will output following tree (actual output of the interpreter with flag *-p*):

```
|--[-]
    |--[+]
        |--[10]
        |--[*]
            |--[100]
            |--[10]
    |--[6]
```

3 Tree-walk interpreter

Tree-walk interpreter traverses the AST while executing operations that are described by the terminals within the tree. Additionally it evaluates the semantics of the code and throws run-time errors.

3.1 Evaluating and executing binary operations

Below is code evaluating and executing binary operations.

```
 1 rf::Reference *Interpreter::binop
 2         (rf::Reference *l, rf::Reference *r, int op)
 3 {
 4     int type = check_types(l, r);
 5     rf::Reference *out;
 6         if(type == tk::STRING)
 7         {
 8             if(op == tk::PLUS)
 9             {
10                 out = add(l, r);
11             }
12             else
13             {
14                 delete l;
15                 error("cannot make this type of comparison on strings", r);
16             }
17     }
18     else if(op == tk::MINUS || op == tk::MULT || op == tk::PLUS)
19     {
20         switch(op)
21         {
22             case tk::PLUS: { out = add(l, r); break; }
23             case tk::MINUS:
24                  {
25                     out = new rf::Reference(l->token.val_num - r->token.val_num);
26                     break;
27                  }
28             case tk::MULT:
29                  {
30                     out = new rf::Reference(l->token.val_num * r->token.val_num);
31                     break;
32                  }
33         }
34     }
35     else
36     {
37         out = divide(l, r, op);
38     }
39     delete l; delete r;
40     return out;
41 }
```

In order to evaluate a binary operation, *binop* takes as arguments left and right node of an expression subtree along with the operator. This is because binary operation is bound to the right side of assignment, and because there are more right-bound expressions, there is a function that manages that choice so that functions working with terminals can be cleaner and more readable. The first function, which is called is *check_types*, returns the type of expression, but also has optional side effect of throwing a run-time error when incompatible types are to be computed. When the type check is successful, there are two main branches. If the type is a string, the interpreter is only to execute concatenation, which means that only viable operator is addition, which is tested in 6th line. If the types are numerical, then all binary operations can be performed. Addition, subtraction and multiplication are seperated from divisions and modulo because they do not need checking for zero in right value. If any of the options checks out, a new reference (a field in activation record) is created with literal expression in its constructor. In the end, all redundant heap-allocated objects are disposed of to prevent memory leaks and pointer to reference is returned.

3.2 Executing function calls

The last example are function calls.

```
rf::Reference *Interpreter::method_call(ast::AST *root)
{
    std::string method_name = root->token.val_str;
    std::vector<rf::Reference*> computed_params;
    rf::Reference *return_reference;

    if(!root->children.empty())
        collect_params(root->children[0], &computed_params);

    call_stack.push_AR(method_name, lookup_method(method_name, root));
    ast::AST *method_root = call_stack.peek_for_root();
    init_record(root, &computed_params);

    if(method_root->children.size() == 2)
    {
        return_reference = exec_block(method_root->children[1]);
    }
    else
    {
        return_reference = exec_block(method_root->children[0]);
    }
    call_stack.pop();
    return return_reference;
}
```

Method calls are a bit more complicated, as they require managing memory using a call stack, which holds activation records (AR) holding all variables from each function call. Every call requires an instance of AR pushed onto the stack. When the interpreter enters the subtree of method call, the first thing it does is computing and collecting arguments from function call to a vector of references. After that an instance of activation record is allocated and pushed onto the stack. Function *lookup_method* checks for declaration of called method and returns the pointer to the root of body of the method, which later is assigned to variable *method_root*. Having done that, the record gets populated with references by method *init_record* which assigns values to all variables while also checking arity (number of supplied arguments). The last if statement branches between functions with or without arguments because their structure varies. The *block* node (defined in grammar) is then passed to method executing blocks of statements *exec_block* and assigning pointer to return reference to the *return_reference* variable. After that, the top of the stack is popped and the reference is returned. Recursion is possible owing to *exec_block* method because it can call another method allowing for repeating the whole process with activation record pushed onto the current one.