Criterion B

Word count: 466

Contents

1 Technologies

- **C++14** (g++ compiler)
  - implementation language
  - provides abstraction mechanisms without performance loss
- **ASAN (ASan)**
  - debugging tool
  - detects memory corruption (leaks, buffer and stack overflows)
- **CMake**
  - tool for generating makefiles
  - makes building larger C/C++ projects more efficient

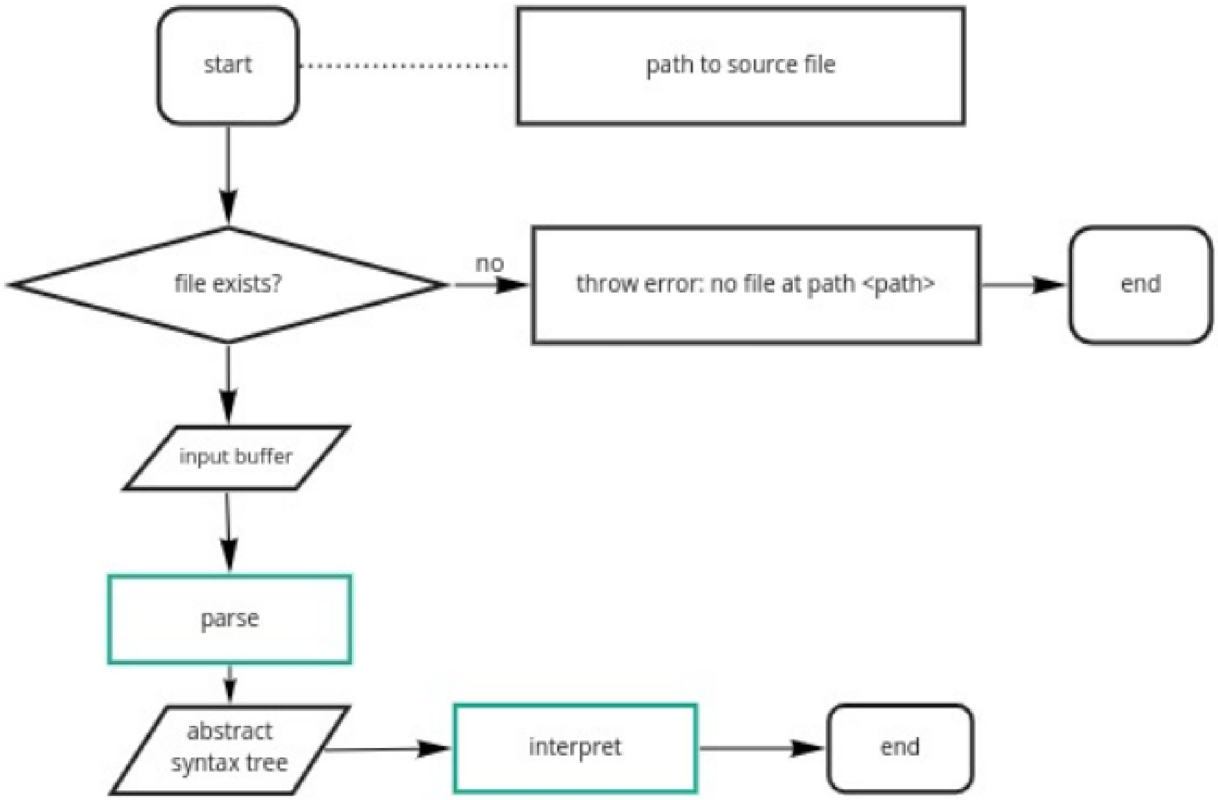
2 Test Plan

In order to test the interpreter, a series of programs testing features described in success criteria have been written and will fed to the interpreter. [See Appendix B for source code of all tests] If the interpreter responses as expected, test is passed. Additionally throughout the development address sanitizer is used to detect and mitigate memory leaks.

3 General flowcharts

Flowcharts in case of an interpreter do not serve very important role in designing an tree-walk interpreter, as the main focus is on grammar, which represented in flowcharts would grow to impractical sizes, similarly with the run-time logic of interpreter. This is why the flowcharts in this section represent general concepts. The program consists of three main stages: lexical analysis, parsing and run-time. In Figure 1 there is a very general flowchart of the program. Green outlines point out that the process is more complex.

Figure 1: General flowchart of the program



In Figure 2 there is flowchart representing general idea behind recursive-descend parser. Parser uses lexer whose flowchart is in Figure 3.

Figure 2: Flowchart of recursive-descend parser

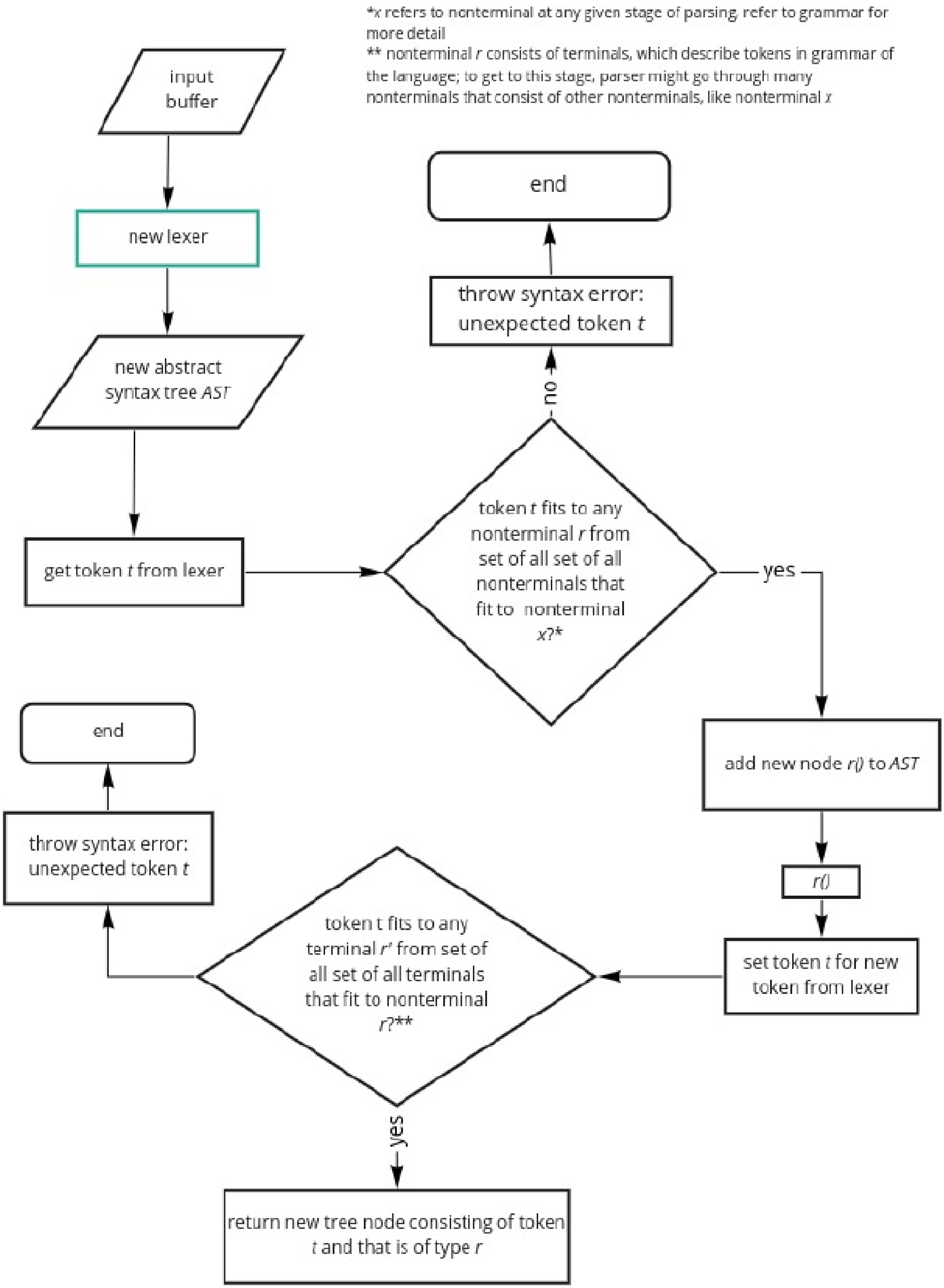
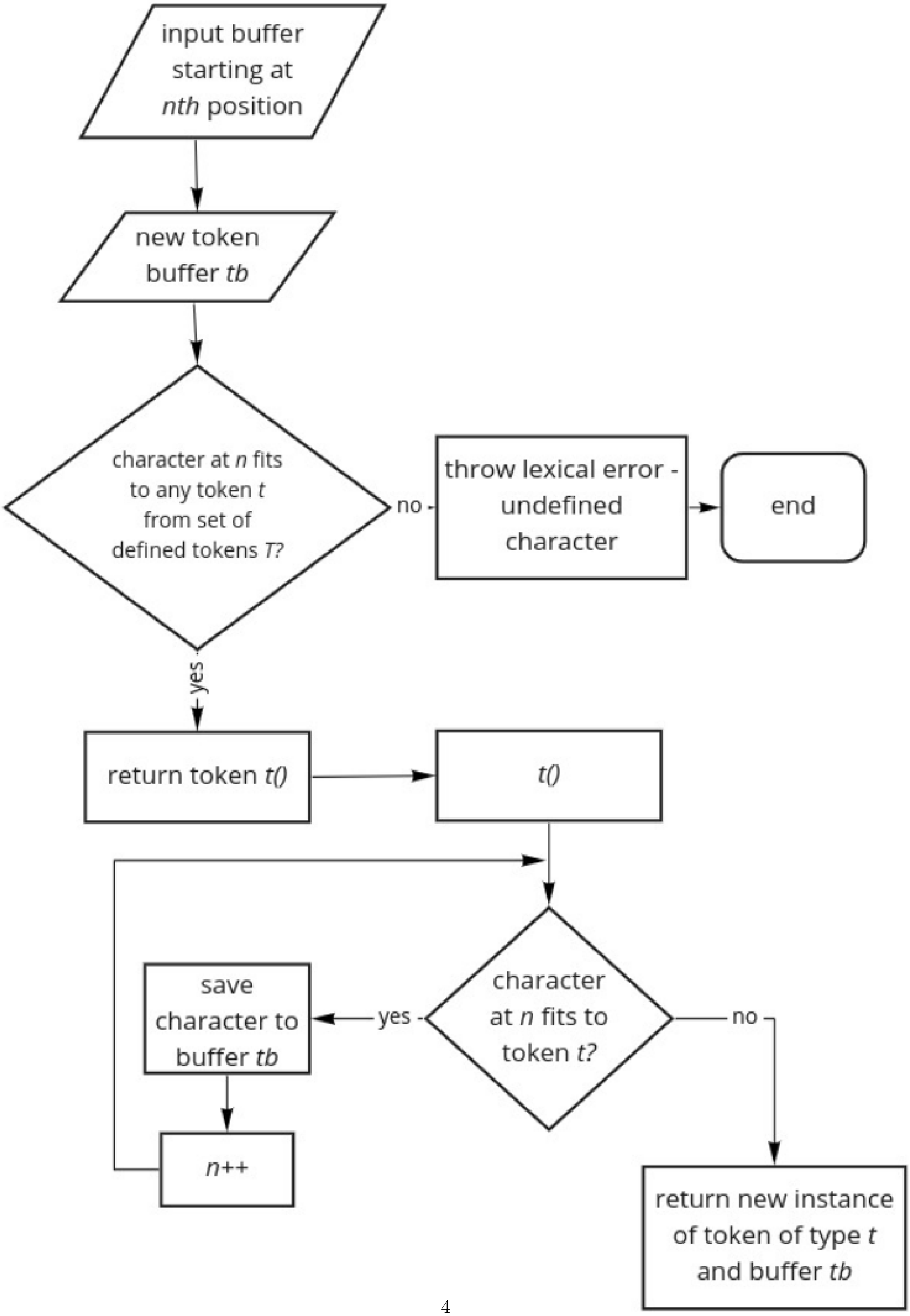


Figure 3: Flowchart of lexer



4 Language design

4.1 Tokens

Tokens represented in regular expressions, which also doubles as a blueprint for lexer.

```
ID : [A-Z]([A-Z] | [0-9] | _)*
METHOD_ID : ([A-Z] | [a-z])[a-z]+ ([A-Z] | [a-z] | _)*
NUMBER : [0-9]+\.?[0-9]*
STRING : "."

RESERVED KEYWORDS / SYMBOLS:

method
if else then loop
while from to end
AND OR
+ - * / div %
< > >= <= == !=
( ) [ ]
```

4.2 Grammar

Backaus-Naur form (BNF) of IB pseudocode developed with reference to pseudocode agenda made by IB[1] . Since the parser is of recursive-descent type, each terminal reflects how the parser builds abtract syntax tree.

```
program :: block
         | block method
         | empty

method_decl :: METHOD METHOD_ID (param_decl) block END METHOD

param_decl  :: ID
             | ID, param_decl
             | empty

block   :: stmt block
         | stmt
         | empty

stmt    :: assignment
         | if
         | for loop
         | while loop
         | method_call
         | expr
         | return
         | std_method
         | BREAK

assignment  :: ID = expr
             | ID = condition
             | ID = arr_decl
             | ID = arr_empty
             | ID = std_method
             | ID = STACK
             | ID = QUEUE
```

[1]Link to source: https://ib.compscihub.net/wp-content/uploads/2015/04/IB-Pseudocode-rules.pdf

```
if  :: IF condition THEN block END IF
     | IF condition THEN block ELSE if END IF
     | IF condition THEN block ELSE if ELSE block END IF
     | IF condition THEN block ELSE block

for_loop    :: LOOP id FROM expr TO expr block END LOOP

while_loop  :: LOOP WHILE condition block END LOOP

method_call :: method_id(params)

return  :: RETURN expr
         | RETURN condition
         | RETURN

params  :: list_of_elements

condition   :: cmp AND cmp
             | cmp OR cmp
             | cmp

cmp :: expr > expr
     | expr < expr
     | expr >= expr
     | expr <= expr
     | expr == expr
     | expr != expr

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

term    :: factor * factor
         | factor / factor
         | factor DIV factor
         | factor MOD factor
         | factor

factor  :: NUM
         | STRING
         | ID
         | method_call
         | std_method
         | arr_acc
         | (expr)

arr_decl    :: [list_of_elements]
             | [arr_decl, arr_decl]

list_of_elements    :: expr
                     | expr, list_of_elements
                     | empty

arr_empty   :: ARRAY(list_of_elements)

arr_acc :: ID accessor
```

```
accessor     :: [NUM]
              | [NUM] accessor
              | empty

std_method   :: OUTPUT(params)
              | INPUT(STRING)
              | container_method

container_method    :: ID.length()
                     | ID.push()
                     | ID.pop(params)
                     | ID.enqueue(params)
                     | ID.dequeue()
                     | ID.getNext()
```

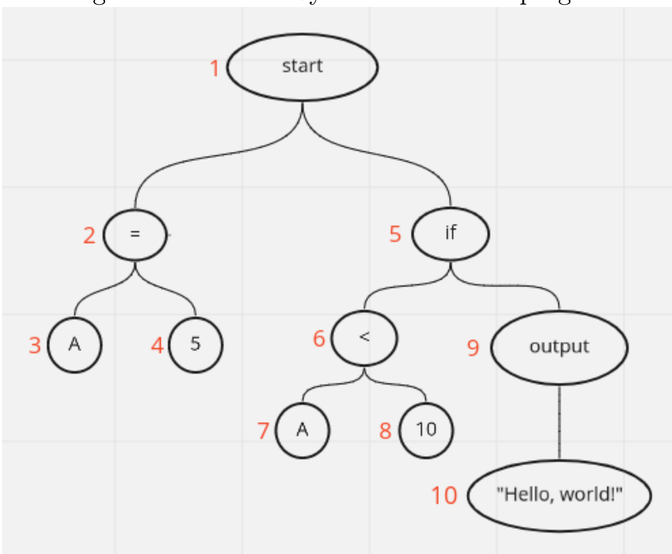
5 Inductive example of interpreter working with Abstract Syntax Tree (AST)

Suppose we have a very simple program:

```
A = 5
if A == 5 then
    output("Hello, world!")
end if
```

After parsing, we would get the following AST - Figure 4. The red numbers beside nodes signify which node is being processed.

Figure 4: Abstract Syntax Tree of the program



Below is a "transcript" of what the interpreter does with the number referring to the number in the Figure 4

1. start
2. assign variable from right node to value from right node
3. check if variable A exists in activation record, if it does not, create a new field; return reference to the field

4. return value reference to numerical value 5
5. check condition from right-most node, if it is true, then execute other nodes
6. if value from right node is lower than the one from left node, return true
7. - check the type of node (here: variable)
   - check activation record for variable A - if it exists and is numerical
   - return reference to numerical value of A
8. - check the type of node (numerical)
   - return reference to numerical value 10
9. print contents of the child node to the console
10. return reference to a string *"Hello, world!"*

From this short example, it is apparent why flowcharts are not very applicable in design of the program, as each type of node holds a bit of different logic behind it.