Scanner Documentation

Lab 1b

Jeler Andrei-Iulian

934

Statement: Considering a small programming language (that we shall call mini-language), you have to write a scanner (lexical analyser)

Task 1: Minilanguage Specification

Deliverables:

1. **Lexic.txt (file containing mini language lexic description; see example)**
2. **token.in (containing the list of all tokens corresponding to the minilanguage)**
3. **Syntax.in - the syntactical rules of the language**

Task 2: Review the mini language specification of a colleague

The minilanguage can be a restricted form of a known programming language, and should contain the following:

- 2 simple data types and a user-defined type

- statements:

- assignment

- input/output

- conditional

- loop

- some conditions will be imposed on the way the identifiers and constants can be formed:

i) Identifiers: no more than 256 characters

ii) constants: corresponding to your types

Lexic

Alphabet

- Uppercase (A - Z) and lowercase (a - z) letters of the english alphabet
- Decimal digits (0 - 9)
- Underscore character “_"

Lexic

1. Special symbols:
    a. Operators: + - * / % < > <= >= == != “and" “or" not
    b. Separators: {} () [] ; , space \n \t
    c. Reserved words: read write if else elif while for integer string bool true false
2. Identifiers

    Combinations of letters, digits and underscore. It can start with an underscore or an letter.

    identifier = non_digit {non_digit | digit}

    non_digit = “_" | letter

    letter = “a” | “b” | ... | “z” | “A” | “B” | ... | “Z”

    digit = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

3. Constants
    a. Integer:

        Integer = zero_digit | [sign] non_zero_digit { digit}

        zero_digit = “0”

        non_zero_digit = “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

        digit = zero_digit | non_zero_digit

        sign = “+” | “-”

    b. Char:

        char = “ ’ “ letter | digit | special_char “ ‘ “

        letter = “a” | “b” | ... | “z” | “A” | “B” | ... | “Z”

        digit = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

special_char = “_” | “.” | “,” | “;" | “:" | “ " | “?" | “!" | “@” | “ ” " |“ /" | “(" | “)" | “|" | “-" | “+" | “=" | “{" | “}" | “*” | “[" | “]" | “$" | “%" | “^"

c. String:

string = “ \” ” {letter | digit | special_char} “ \” “

d. Bool:

bool = true | false

Syntax

Sintactical rules:

program = "program” compound_stmt .

compound_stmt = “{“ stmt_list “}” .

stmt_list = stmt {stmt} .

stmt = simple_stmt | complex_stmt.

simple_stmt = (decl_stmt | assign_stmt | return_stmt | IO_stmt) “;”.

complex_stmt = if_stmt | loop_stmt.

IO_stmt = “read(“ identifier “)” | “write(“ expression ”)” .

decl_stmt = type identifier {“," identifier} | type identifier “=" expr {“," identifier “=" expr} .

type = primary_types | array_types .

primary_types = “integer" | “char" | “string" | “bool" .

array_types = primary_types “[" number “]" .

number = non_zero_digit {digit} .

identifier = non_digit {non_digit | digit} .

non_digit = “_" | letter .

assign_stmt = identifier “=” expression.

expression = expression operator term | term.

operator = “+”

term = term ("*" | “/”) factor | factor.

factor = "(" expression ")" | identifier | identifier “[“ number “]” | constant.

constant = integer | string.

return_stmt = “return” expression .

if_stmt = “if” condition compound_stmt {“elif” condition compound_stmt} [“else” compound_stmt)].

loop_stmt = for_stmt | while_stmt .

for_stmt = “for(“ assign_stmt “;” condition {“;” assign_stmt} “)” comp_stmt .

while_stmt = “while (“ condition” “)” comp_stmt .

condition = [not] [“(“] expression ralational_operator expression {conditional_operator condition} [“)”] .

relational_operator = “>" | “<" |”>=" | “<=" | “==" | “!=" .

conditional_operator = “and” |”or" .

Tokens

Token	Code
identifier	0
constant	1
if	2
else	3
elif	4
while	5
for	6
read	7
write	8
integer	9
char	10
string	11
+	12
-	13
*	14
/	15
%	16
<	17
>	18
<=	19
>=	20
==	21
!=	21
and	22

or	23
{	24
}	25
(	26
)	27
;	28
,	29
Space	30
\n	31
\t	32
[	33
]	34
program	35
not	36

Programs from the first lab

Changes:

Integer main() is changed with program.

The read receives the identifier as a parameter.

Removed the ++ and transformed it to i = i + 1. The same for +=.

P1 : maximul a 3 numere

```
program
{
        integer a, b, c;
        string printMessage = “is the biggest number";

        read(a);
        read(b);
        read(c);

        if (a > b and a > c)
                {
                        write(“a", printMessage);
```

```
        }
    elif (b > a and b > c)
        {
            write(“b", printMessage);
        }
    else
        {
            write(“c", printMessage);
        }
    return 0;
}
```

P1 lexic errors

```
program
{
    integer 1a, 2b, 3c;

    read(a);
    read(b);
    read(c);

    if (a > b && a > c) then
        #
            write(“A is the biggest number");
        #
    elif (b > a && b > c) then
        #
            write(“B is is the biggest number ");
```

```
        #
    else
        #
            write("C is the biggest number");
        #
    return 0;
}
```

Explanations:

- An identifier can't start with a digit
- && is not part of the lexic (the login uses words like and, or, not)
- # is not defined in the lexic, { and } are used to delimit some compound statements

P2: Greatest common divisor of 2 numbers

```
program
{
    Integer a, b;

    read(a);
    read(b);

    While ( a != b )
    {
        If (a > b)
        {
            a = a – b;
        }
        else
        {
```

```
                b = b – a;
            }
        }

        write(“The greatest common divisor is", b);
        return 0;
}
```

P3 Sum of an array

```
program
{
        integer result = 0;
        integer[10] array = {1, 2, 3, 4 ,5 , 6, 7, 8 ,9);

        for (int i = 0; i < 9; i =i+1)
            {
                result = result + array[i];
            }

        write(result);
        return 0;
}
```

Lab 2

Statement: Implement the Symbol Table (ST) as the specified data structure, with the corresponding operations

Deliverables: class ST(source code) + documentation.

UPLOAD documentation, on the first line link to git for source code

4. Symbol Table:

- unique for identifiers and constants (create one instance of ST)

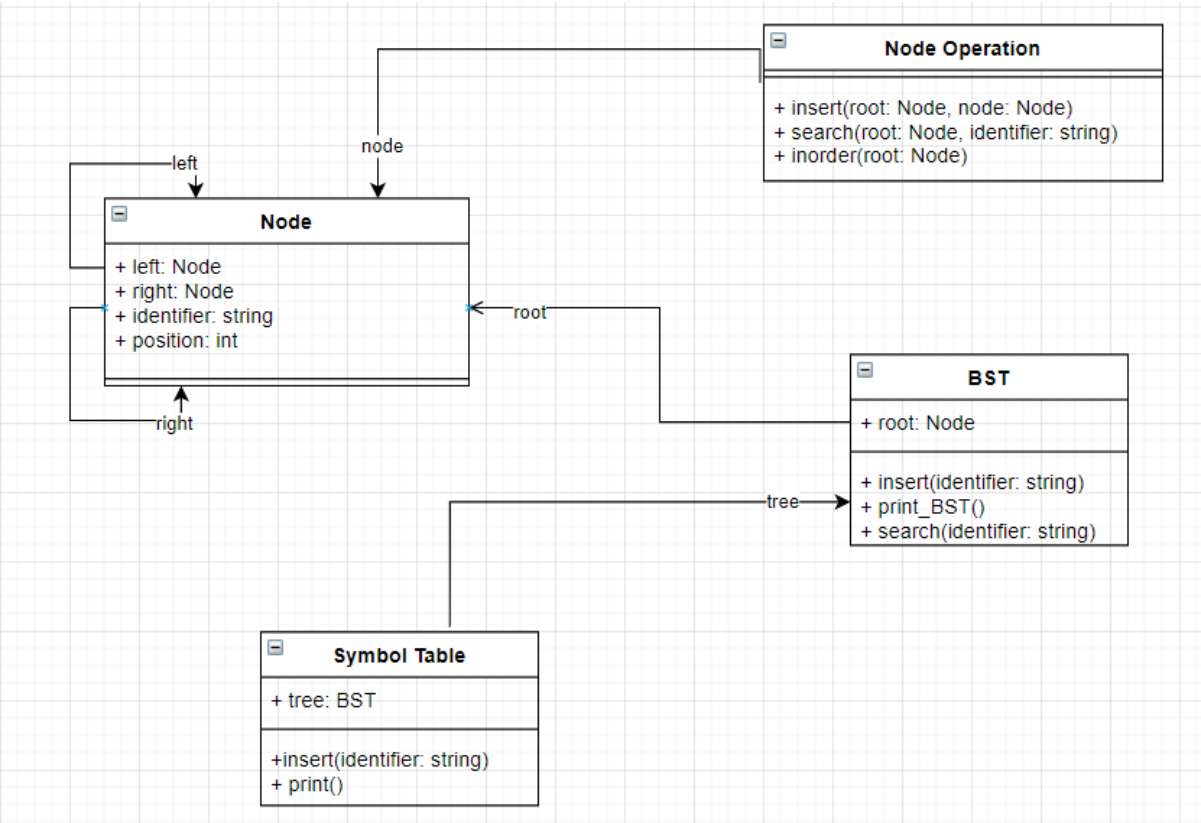
5. Symbol Table (you need to implement the data structure and required operations) :

- alphabeticallybinary search tree

https://github.com/SummerRolls99/FLCD/tree/main/lab2

The chosen programming language is python. I have choosed to use the binary search tree as the data structure for the symbol table. The position is represented as an integer which is increased when a new node is inserted, and the node is receiving the current position, if the identifier already exists, a new position will not be generated.

Class diagram:



Lab 3 – 4

Statement: Implement a scanner (lexical analyzer): Implement the scanning algorithm and use ST from lab 2 for the symbol table.

Input: Programs p1/p2/p3/p1err and token.in (see Lab 1a)

Output: PIF.out, ST.out, message “lexically correct” or “lexical error + location”

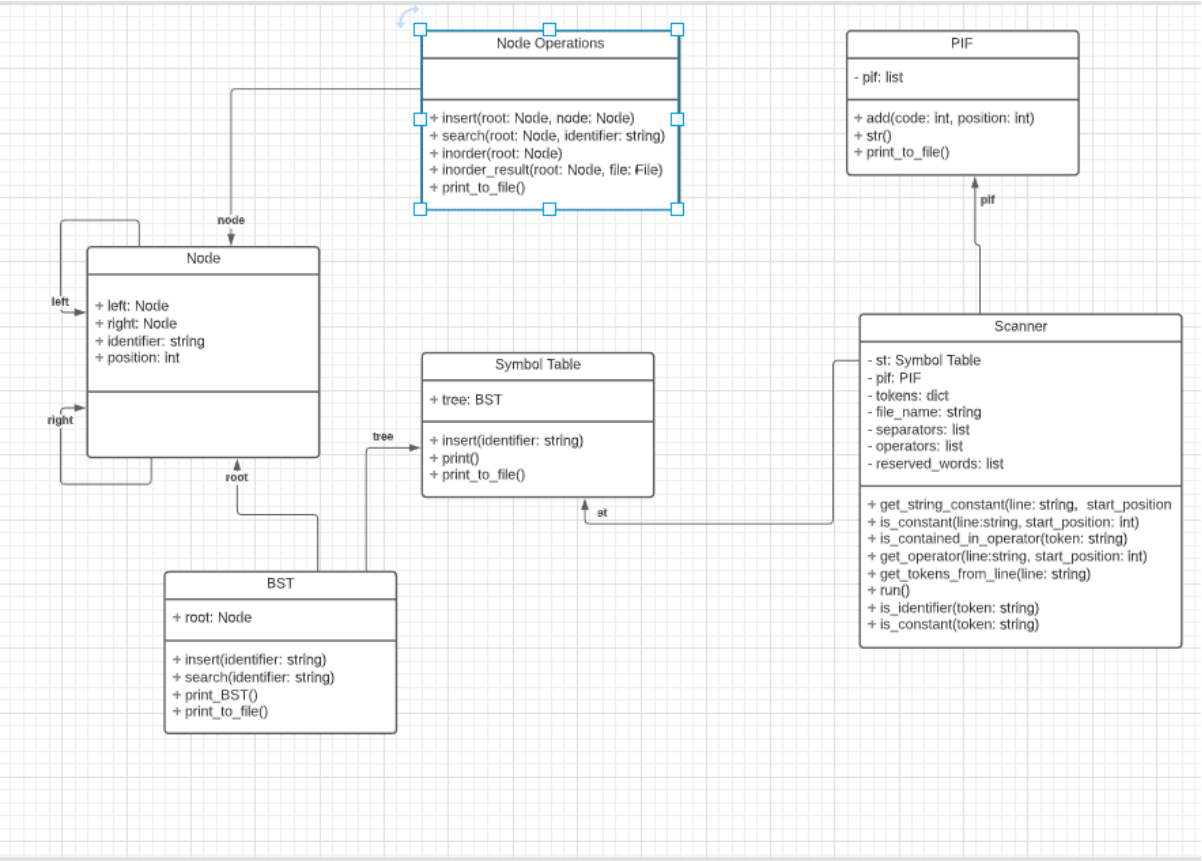
Deliverables: input, output, source code, documentation

Details:

- ST.out should give information about the data structure used in representation
- If there exists an error the program should give a description and the location (line and token)

Github link:

https://github.com/SummerRolls99/FLCD/tree/main/lab3



In order to extract the tokens from the program, the algorithm takes the code line by line. Each line is scanned letter by letter, checking for each kind of token. First it checked if a " appears, if one appears it will continue scanning the line until the string constant is closed. If a char is part of an operator it will check if, in continuation of the line, an operator is really there. After extracting the tokens of the line, we will insert in the pif all the tokens (excepting space and new line). During this process it will also check if it is a valid identifier or constant, using regular expressions. If a token is invalid, it will be printed with yellow, and at the end the PIF and the ST will not be printed. In case of success, the PIF and the ST will be printed on screen and written in pif.out and st.out