Compiler term project

Implementation of lexical analyzer

Class by prof. Hyosu Kim

20184395 Myeongwon Choi

20186491 Chihyun Song

Table of Contents

[Abstract 3](#_Toc39674876)

[Language specification 3](#_Toc39674877)

[Regular expression definition 3](#_Toc39674878)

[NFA and DFA definition 4](#_Toc39674879)

[Program implementation 7](#_Toc39674880)

[Building DFA class 7](#_Toc39674881)

[Generating DFA object 8](#_Toc39674882)

[Extra problem 9](#_Toc39674883)

[Parse token using DFA 10](#_Toc39674884)

[Program test 11](#_Toc39674885)

[1. test.c 11](#_Toc39674886)

[2. input.c 12](#_Toc39674887)

[3. error.c 13](#_Toc39674888)

# Abstract

We implemented a syntax analyzer by implementing a DFA class which acts based on the definition introduced on compiler lecture.

And we generated instances of it and defined its behavior based on DFA detail we generated.

# Language specification

## Regular expression definition

First we made regular expressions of given tokens so we can make DFAs with it.

* (positive\_digit): (1|2|3|4|5|6|7|8|9)
* (digit): (0|1|2|3|4|5|6|7|8|9)
* (alphabet): (a|b|c|…|z|A|B|…|Z)

1. Signed integer(INT): ((-|ε)(positive\_digit)(digit\*)|0)
2. Literal string(STRING): “(alphabet| |digit)”
3. Floating-point number(FLOAT):

(-|ε)((positive\_digit)(digit\*)|0).((digit\*)(positive\_digit)|0)

1. An identifier of variables and functions(ID): (\_|alphabet)(alphabet|digit|\_)\*
2. Whitespaces(WHITESPACE): (\t| |\n)\*
3. Variable type(TYPE),

Boolean string(BOOLEAN),

Keywords for special statements(KEYWORD),

Arithmetic operators(ARITHMETIC),

Bitwise operators(BITWISE),

Assignment operator(ASSIGNMENT),

Comparison operators(COMPARISON),

A terminating symbol of statements(SEMICOLON),

A pair of symbols for defining area/scope of variables and functions(BRACE),

A pair of symbols for indicating a function/statement(PARENTHESES),

A symbol for separating input arguments in functions(SEPARATOR)

: all same. Just write all the accepted lexemes and separate them with |.

Actually, most of token definitions are just based on a list of words accepted. So, there is some redundant token definitions that is totally clear. We skipped that kind of token definition.

## NFA and DFA definition

The strict type of DFA must have exactly one transition for one input alphabet for all current state. So DFA follows strict definition has some trap state which cannot derive to final state to represent the halted state of DFA.

But we don’t have to implement our DFA in this way, our casual definition of a DFA will have at most one possible transition for one input alphabet for all current state. If a DFA gets an input alphabet which is not defined as an alphabet, The DFA will simply halt at that moment.

Although we should follow the procedure which converts a regular expression into a NFA and translate it again into a DFA, but we decided to build DFA directly since the language specification is simple so we can make efficient DFA directly.

* Pos:1|2|3|4|5|6|7|8|9
* Dig:1|2|3|4|5|6|7|8|9|0
* Alpha:a|b|c|…|z|A|B|…|Z

INT

A close up of text on a white background

Description automatically generated

STRING

A close up of text on a white background

Description automatically generated

FLOATA close up of text on a white background

Description automatically generated

IDA close up of text on a black background

Description automatically generated

WHITESPACEA close up of text on a black background

Description automatically generated

A picture containing text

Description automatically generatedA close up of text on a white background

Description automatically generated

# Program implementation

## Building DFA class

We defined corresponding DFA. In order to get things work, we need to implement DFA class works based on our DFA designs.

First, a DFA consists of:

* a finite set of states
* a finite set of input symbols called the alphabet
* a transition function
* an initial or start state
* a set of accept states

Among the formal definition of DFA, we need the information of alphabet, transition functions, initial state, and a set of final states. Actually, an information about set of state can be omitted because unreachable sets are useless. And the object definition of DFA has one more field, halted or running.

So constructor will simply get mentioned values, and initialize the field values with input.

Next, we have process function which gets a character or string input and process it. Giving an automaton object string input is equivalent to giving character input sequentially.

First, automaton checks if input character is defined as alphabet. If not, DFA will halt.

And check if the automata is already halted. If so, it will simply do nothing because halted automaton does not accept any character.

Finally a DFA will traverse around transition function list to find a transition function which matches the current state and the input character. If found, change current state. If not found, it means transition function not found. halt DFA.

Actually we can use hash table to store transition function so we can find corresponding transition function immediately. but we did not use a hash table because the language specification is quite simple so list is fast enough to use.

And a DFA has additional operation which real world DFA does not have, setState. It directly sets current state with input parameter, and it will set DFA state to running.

Now our DFA definition is all set. We can make our own DFA object with DFA class. Additionally, we built a simple utility function which builds DFA out of list of lexemes can be accepted. It helps us building pre-defined DFA.

## Generating DFA object

positive = '123456789'  
INT = DFA(string.digits + "-", [(0, '0', 1), (0, positive, 3), (0, '-', 2), (2, positive, 3), (3, string.digits, 3)], 0, [1, 3])

FLOAT = DFA(string.digits + '-.', [(0, '-', 1), (0, '0', 2), (0, positive, 3), (1, '0', 2), (1, positive, 3), (3, string.digits, 3), (3, '.', 4), (2, '.', 4), (4, string.digits, 5), (5, positive, 5), (5, '0', 6), (6, positive, 5), (6, '0', 6), (5, positive, 5)], 0, [5])

STRING = DFA(string.digits + string.ascii\_letters + ' ' + '"', [(0, '"', 1), (1, string.digits + string.ascii\_letters + ' ', 1), (1, '"', 2)], 0, [2])

ID = DFA(string.ascii\_letters + string.digits + '\_', [(0, string.ascii\_letters + '\_', 1), (1, string.ascii\_letters + string.digits + '\_', 1)], 0, [1])

WHITESPACE = DFA('\t\n ', [(0, '\t\n ', 1), (1, '\t\n ', 1)], 0, [1])  
  
TYPE = generate\_dfa\_out\_of\_list(['int', 'char', 'float', 'bool'])  
BOOLEAN = generate\_dfa\_out\_of\_list(['true', 'false'])  
KEYWORD = generate\_dfa\_out\_of\_list(['if', 'else', 'while', 'for', 'return'])  
ARITHMETIC = generate\_dfa\_out\_of\_list(['+', '-', '\*', '/'])  
BITWISE = generate\_dfa\_out\_of\_list(['<<', '>>', '&', '|'])  
ASSIGNMENT = generate\_dfa\_out\_of\_list(['='])  
COMPARISON = generate\_dfa\_out\_of\_list(['<', '>', '==', '!=', '<=', '>='])  
SEMICOLON = generate\_dfa\_out\_of\_list([';'])  
BRACE = generate\_dfa\_out\_of\_list(['{', '}'])  
PARENTHESES = generate\_dfa\_out\_of\_list(['(', ')'])  
SEPARATOR = generate\_dfa\_out\_of\_list([','])

Code above shows our DFA definition of tokens specified as the language specification.

## Extra problem

We noticed that the -(hyphen) can be ambiguous because hyphen can be recognized as minus sign(int or float) or subtraction operator at the same time.

We tried to resolve this problem with our DFA, but it was impossible since this ambiguity is a syntax manner ambiguity. So DFA cannot solve this problem because DFA has not enough power to solve this problem.

But we can solve this ambiguity problem if we always recognize hyphen as a subtraction operator, then it will be totally fine because subtraction operator simply invert subtrahend’s sign and add. Actually this would be what we wanted to do, so there will be no problem.

But language specification does not designed this way, so we’ll just include it on our program but will not activate it.

# Solution section for solving ambiguity problem  
FLOAT\_SOLUTION = DFA(string.digits + ".", [(0, '0', 2), (0, positive, 3),  
 (3, string.digits, 3), (3, '.', 4), (2, '.', 4), (4, string.digits, 5),  
 (5, positive, 5), (5, '0', 6), (6, positive, 5), (6, '0', 6), (5, positive, 5)], 0,  
 [5])  
INT\_SOLUTION = DFA(string.digits,  
 [(0, '0', 1), (0, positive, 3), (3, string.digits, 3)], 0, [1, 3])  
DFAs\_solution = [INT\_SOLUTION, FLOAT\_SOLUTION, STRING, WHITESPACE, TYPE, BOOLEAN, KEYWORD, ARITHMETIC, BITWISE, ASSIGNMENT, COMPARISON, SEMICOLON,  
 BRACE, PARENTHESES, SEPARATOR, ID]  
# Solution section for solving ambiguity problem end

# tokens = lexically\_analyze(file\_txt, DFAs\_solution)  
# if you want to parse token with no ambiguity then uncomment

## Parse token using DFA

We have two rules when parsing the text.

1. Longest pattern matching: If there are several DFAs accept the string, we use DFA matches the longest lexeme.
2. Priority: If there are multiple DFAs match the string with the same length, we use DFA at the highst priority.

We defined function *lexically\_analyze(string\_to\_scan, dfa\_list)* to lexically analyze sequence *string\_to\_scan* using dfas in *dfa\_list*.

First we initialize variables such as lexemes which stores sequence of lexemes scanned so far.

And we traverse the input text all the way to the end, every iteration we let all of our DFAs process the character. and we keep track of candidate DFA which can accept the longest string from end of previous recognized token.

When every DFAs in the list halts, it means all DFAs cannot accept any character anymore. So we stop scanning and determine the candidate DFA as token and add the option value, token name and state for additional information.

If we couldn’t find appropriate DFA from the end of previous token, it means there is no DFA can accept the remaining string. It is error of input string. So print the error message and terminate the program.

# Program test

We used module tabulate to make table out of our lexemes list. So we generated requirement.txt file to indicate the module used. Further information about executing the program is in README.md file.

## 1. test.c

int func(int a) { return 0; }

A screenshot of a video game

Description automatically generated

As you can see in this execution output, the lexer processes given input text perfectly.

## 2. input.c

int func (char c, bool b, float f){  
 int a = 0 + 1200000003 + -1;  
 a = 0-0;  
 char \_\_c1 = "hello world this is an example 123 ";  
 bool b\_1234 = true false;  
 float f\_ = 0.000012301 + -1233.3 + 33000000003+ 0.0;  
  
 if(b\_1234 != false) {  
 return 12 > 34;  
 } else if {  
 return 1 << 3;  
 }  
}  
Ture\_\_\_r  
0 1 22 123 56 001 "asdfagwet"  
0.5, -0.0, -10.0, 100.00001

A screenshot of a computer

Description automatically generated

Gave list option to represent output data in list format. This means we can use this program as a part of a compiler.

## 3. error.c

0.0-0 does not make error  
if you try to use undefined character such as  
+-+  
+-+ hash sign you""ll get error  
  
or if you could not find transition function  
0.0.0.0.0

A screenshot of a video screen

Description automatically generated

If the program gets unexpected character or there is no transition function can derive remaining DFA, the program will print error message contains line of error character and location and terminate.