Report

Team 8. 임동영, 한태균

The definition of tokens and their regular expression

Token 정의와 그 정규식

다음과 같이 token들을 정의 하였습니다.

1. <SINGLE CHARACTER> = (‘) (<digit> | <letter> | blank) (‘)
2. <VARIABLE TYPE> = int | char | String | boolean
3. <LITERAL STRING> = (“) (<letter> | <digit> | <white space>) * (“)
4. <SIGNED INTEGER> = 0 | (- | ε) (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) (<digit>) *
5. <BOOLEAN STRING> = true | false
6. <WHITE SPACE> = (₩n | ₩t | blank)⁺
7. <ARITHMETIC OPERATER> = + | - | * | /
8. <ASSIGNMENT OPERATER> = (=)
9. <COMPARISON OPERATER> = < | > | <= | >= | == | !=
10. <TERMINATING SYMBOL> = ;
11. <LPAREN> = (
12. <RPAREN> =)
13. <LBRACE> = {
14. <RBRACE> = }
15. <LBRANKET> = [
16. <RBRANKET> =]
17. <COMMA> = ,
18. <IDENTIFIER> = (_ | <letter>)(_ | <letter> | <digit>) *
19. <KEYWORD> = if | else | while | public | static | main | class | return | void

The DFA transition graph or table for recognizing the regular expressions.

col : state, row : transition, state which red and bold : final state

SINGLE CHARACTER

	'	digit	letter
0	1		
1	4	2	3
2	5		
3	5		
4	5		
5			

VARIABLE TYPE

	i	n	t	c	b	S	a	r	o	l	h	e	g	n
0	1			3	6	12								
1		2												
2			17											
3											4			
4							5							
5								17						
6									7					
7									8					
8										9				
9												10		
10							11							
11														17
12			13											
13								14						
14	15													
15														16
16													17	
17														

SIGNED INTEGER

	0	digit	digit except 0	-
0	3		2	1
1			2	3
2		2		
3				

BOOLEAN STRING

	t	r	u	f	a	l	s	e
0	1			3				
1		2						
2			6					
3					4			
4						5		
5							6	
6								7
7								

WHITE SPACE

	₩n	₩t	blank	₩r
0	1	1	1	1
1	1	1	1	1

ARITHMETIC OPERATOR

	+	-	/	*
0	1	1	1	1
1				

ASSIGNMENT OPERATOR

	=
0	1
1	

COMPARISON OPERATOR

	<	>	=	!
0	1	1	2	2
1			3	
2			3	
3				

TERMINATTING SYMBOL

	;
0	1
1	

LPAREN

	(
0	1
1	

RPAREN

	)
0	1
1	

LBRACE

	{
0	1
1	

RBRACE

	}
0	1
1	

LBRANKET

	[
0	1
1	

RBRANKET

	]
0	1
1	

COMMA

	,
0	1
1	

LITERAL STRING

	“	letter	digit	white space
0	1			
1	5	2	3	4
2	5	3	4	1
3	5	4	1	2
4	5	1	2	3
5				

IDENTIFIER

	_	letter	digit
0	1	2	
1	1	2	3
2	2	3	1
3	3	1	2

All about how your lexical analyzer works for recognizing tokens

Transition Table의 구현

DFA의 state 간 이동을 나타내기 위한 Transition Table을 표현하기 위해 Python의 Dictionary를 사용하였습니다.

```
"Transition Name": {
    <state>: {<transition key>: <next state>}.
    "final": [<list of final state>],
}
```

그리하여, DFA는 예를 들어 다음의 모양을 가지게 됩니다.

```
"SINGLE_CHARACTER": {
    0: {"'": 1},
    1: {
        string.ascii_letters: 3,
        string.digits: 2,
        " ": 4,
    },
    2: {"'": 5},
    3: {"'": 5},
    4: {"'": 5},
    5: {},
    "final": [5],
}
```

또한, 첫 문자가 겹치지 않는 keyword들 간에 한하여 Transition Table을 더 쉽게 만들 수 있게 도와주는 build_transition_table_with_lexemes 함수를 정의하여 사용하였습니다. 이 함수는 명시적인 keyword들을 lexemes 라는 입력으로 받습니다.

그리고 lexeme 들의 길이를 이용하여 final state를 계산합니다. 그리고 lexeme 별로 순회하며 lexeme들의 한 글자 한 글자를 transition의 key로 설정하고, 그에 따른 state의 변경을 명시한 후 이를 통해 만들어진 Dictionary 형태의 Transition table을 반환합니다.

```python
def build_transition_table_with_lexemes(lexemes):
    transition_table = {}
    i = 0
    final_state = 1
    for lexeme in lexemes:
        final_state += len(lexeme) - 1

    transition_table["final"] = [final_state]
    transition_table[final_state] = {}

    for lexeme in lexemes:
        for j, c in enumerate(lexeme):
            if j == 0:
                if i == 0:
                    transition_table[0] = {c: i + j + 1}
                    continue
                transition_table[0] = {**transition_table[0], c: i + j + 1}
                continue

            if j + 1 == len(lexeme) and final_state != 0:
                transition_table[i + j] = {c: final_state}
                i += j
                continue

            transition_table[i + j] = {c: i + j + 1}

    return transition_table
```

첫 글자가 겹칠 경우 DFA의 transition table이 아닌 NFA의 transition table이 만들어져 버리기 때문에 개선이 필요한 코드이나. 이번 과제에서는 겹치는 경우가 없어서 일단 사용하였습니다.

DFA의 구현

Lexical Analyzer에서 Token 별로 여러가지 DFA를 생성하여 사용하게 되기 때문에, DFA는 별도의 클래스로 나누어서 작성하였습니다.

DFA의 생성자는 일단 이름, 알파벳, Transition table, 초기 state, 최종 state를 받아서 DFA를 초기화해줍니다. 또한, 추후 DFA가 동작하고 있는 상태인지 확인하기 위하여 running이라는 인스턴스 멤버를 만들어 True로 초기화합니다.

```
def __init__(self, name, alphabet, transition_table, initial_state,
final_state):
        self.name = name
        self.alphabet = alphabet
        self.transition_table = transition_table
        self.state = initial_state
        self.initial_state = initial_state
        self.final_state = final_state
        self.running = True
```

DFA의 주요 동작은 DFA 클래스의 run 메소드에서 이루어집니다. 해당 run 메소드는 inputValue라는 하나의 문자를 입력 받아서 DFA의 state를 변경하는 기능을 수행합니다.

해당 메소드는 현재 DFA의 동작 상태(self.running)을 반환하여 이를 사용하는 Lexical Analyzer에게 동작 상태를 알립니다.

해당 메소드는 다음의 조건을 모두 만족할 때에만 Transition table을 참고하여 state를 변경합니다.

- 현재 DFA가 동작 중(running) 일 때
- 입력 값이 DFA의 알파벳의 범위 안에 있을 때

그렇지 않다면 False를 반환하여 해당 DFA가 동작하지 않음을 알립니다.

```
if not self.running:
    return False

if inputValue not in self.alphabet:
    self.running = False
    return False
```

위의 조건을 모두 만족한다면, Transition Table을 참고하여 state를 변경하는 시도를 합니다.

Transition Table의 첫 번째 깊이의 내용인 이전 state와 transition에 대한 정보를 key-value 형태로 받아옵니다. 만약 DFA의 state가 Transition table의 이전 state에 해당하는 내용과 같다면, 해당 state로부터 수행될 수 있는 transition에 대한 정보를 key-value 형태로 받아옵니다. 그렇게 하여, key와 run 함수의 입력값이 일치할 경우, 다음 state로 DFA의 state를 이동시킵니다. 그리고 해당 DFA가 동작중임을 알립니다.

```
for old_state, transitions in self.transition_table.items():
    if old_state == self.state:
        for key, new_state in transitions.items():
            if inputValue in key:
                self.state = new_state
                return True
```

만약 반복문의 수행 중간에 반복문을 빠져나가지 않을 경우엔 적절한 Transition을 찾지 못하였다는 것이고, 해당 입력값이 해당 Token의 DFA에서는 처리될 수 없다는 뜻이기에 for-else 문을 사용하여 DFA의 동작 상태를 False로 변경하고, 그 사실을 알립니다.

```
else:
    self.running = False
    return False
```

또한. DFA 클래스는 Token 하나의 파싱이 끝날 경우 모두 초기화되어야 하기 때문에, reset 메소드를 제공합니다. reset 메소드는 DFA 인스턴스의 state를 초기 state로 초기화하고, DFA의 동작 상태를 running으로 변경하여 다시 입력 값을 받을 수 있는 상태로 만들어줍니다.

```
def reset(self):
    self.state = self.initial_state
    self.running = True
```

마지막으로, DFA 클래스는 DFA가 최종 state에 도달했는지와 DFA가 동작중인지에 대한 정보를 이용하여 DFA의 Token 인식이 끝났는지 확인하는 isDone 메소드를 제공합니다.

```
def isDone(self):
    return self.running and self.state in self.final_state
```

Lexical Analyzer의 구현

Lexical analyzer는 다음과 같은 DFA들의 리스트입니다. DFA의 순서가 곧 그들의 우선순위를 나타내 줍니다. 예를 들어 comparison operator는 assignment operator보다 앞에 위치합니다. 따라서 '=='를 검사할 때 comparison operator를 먼저 검사할 수 있습니다.

```
lexical_analyzer = [
    keyword,
    singleChar,
    variableType,
    signedInteger,
    literalString,
    booleanString,
    whiteSpace,
    arithmeticOperator,
    comparisonOperator,
    assignmentOperator,
    terminatingSymbol,
    lparen,
    rparen,
    lbrace,
    rbrace,
    lbranket,
    rbranket,
    comma,
    identifier,
]
```

output, getReady 두 리스트의 역할은 각각 결과값과 종료할 수 있는 DFA 목록을 저장하는 것입니다.

```
# output of lexical analyzer
output = []
# list of pass dfa : appended when dfa reached final state
getReady = []
value = ""
```

기본적인 분석기의 로직은 다음과 같이 동작합니다.

1. 파일을 엽니다.
2. 모든 DFA를 현재 받아온 한 글자에 대해 구동합니다.

   2-1) 만약 getReady 안에 있는 DFA가 구동하지 못했다면 4. 로 이동합니다.

   2-2) 만약 현 DFA가 종료될 수 있다면 getReady에 넣습니다.
3. Input value를 value 문자열에 더합니다 다음 글자를 받아 2. 로 이동합니다.

4. output 리스트에 토큰과 value 문자열을 넣습니다. 모든 DFA와 getReady 리스트를 초기화합니다.
5. 현 input value에 대해 다시 한번 DFA를 구동하고 2-2를 수행합니다. 다음 글자를 받아 2. 로 이동합니다.

파일을 전부 읽으면 종료합니다. 이 로직은 마지막 value에 대한 output 저장을 할 수 없는 단점이 있지만 만약 오류가 없다면 getReady에 그에 상응하는 DFA가 들어가 있기에 다음과 같이 쓸 수 있습니다.

```
# last value handling
if getReady:
    output.append(("<" + getReady.pop().name + ">", value))
else:
    print(value, 'occurred error')
```

혹은 getReady에 DFA가 없다면 이는 value에 상응하는 DFA가 없었다는 뜻이 되고 여기서부터 에러가 발생했다는 것을 알 수 있습니다.

다음은 Laxical analyzer의 기본적인 구조입니다.

```
for input in f.read():
    # to check earn output
    flag = False
    for dfa in lexical_analyzer:
        success = dfa.run(input)
        if not success and dfa in getReady:
            output.append(("<" + dfa.name + ">", value))
            value = ""
            getReady.clear()
            # earn output
            flag = True
            for reset_dfa in lexical_analyzer:
                reset_dfa.reset()
            break
        if dfa.isDone() and dfa not in getReady:
            getReady.append(dfa)
    if flag:
        for dfa in lexical_analyzer:
            dfa.run(input)
            if dfa.isDone():
                getReady.append(dfa)
    value += input
```

flag는 우리가 token을 얻었는지를 표시해 줍니다. 다음 input을 검사해 그 이전까지의 value를 결정하는 형식이므로 만약 token을 얻었다면, 현 input에 대한 재검사를 진행해야만 합니다. 그

렇다면 우리는 다음 input이 뭐가 나올지에 따라서 지금까지의 value를 보류해야만 하는 상황이 올지도 모릅니다. 이러한 예외 케이스를 살펴보도록 하겠습니다.

먼저 '=='와 같은 경우 첫 '='에 <ASSIGNMENT OPERATOR>가 먼저 getReady에 들어갈 것입니다. <COMPARISON OPERATOR>는 아직 final state에 도달하지 못했기 때문입니다. 그 다음 '='가 등장하면 우리의 로직으로는 기존에 있던 value('=')를 assignment operator로 output에 넣으려고 할 것입니다. 이러한 사항들을 막아줘야 합니다.

```
# skip ASSIGNMENT OPERATOR when input value is '=' to recognize '=='
if dfa.name == "ASSIGNMENT OPERATOR" and input == "=":
    continue
# skip KEYWORD when input value is in alphabet in IDENTIFIER
elif dfa.name == "KEYWORD" and input in string.digits + \
      string.ascii_letters + '_':
    getReady.remove(dfa)
    continue
```

continue하게 되면 다음 input에서 comparison operator가 assignment operator보다 우선순위가 높으므로 '=='가 정상적으로 output에 들어가게 됩니다. 마찬가지로 keyword를 포함하는 identifier도 잡아낼 수 있습니다.

'-' 기호는 좀 더 섬세하게 다뤄져야 합니다. 먼저 '-'가 정수부에 들어가는 경우는 크게 세가지로 분류될 수 있습니다.

1. 기호 앞에 '=' (w.s) 가 오는 경우
2. 기호 앞에 <ARITHMETIC OPERATER> (w.s) 가 오는 경우
3. 기호 앞에 ',' (w.s) 가 오는 경우

모든 경우 사이엔 공백(white space)이 올 수 있음을 염두해야 합니다. 그래서 일단 '-'는 모두 산술 연산자로 받고, 그 뒤에 정수가 올 때에 이전의 토큰을 확인했습니다.

```
if (dfa.name == "SIGNED INTEGER" and output[-1][0] ==\
    "<ARITHMETIC OPERATOR>") and output[-1][1] == '-':
      if len(output) > 1:
            # ignore <WHITE SPACE>
            if output[-2][0] == "<WHITE SPACE>":
                  idx = -3
            else:
                  idx = -2
            if output[idx][0] == "<ASSIGNMENT OPERATOR>" or output[idx][0]\
               == "<ARITHMETIC OPERATOR>" or output[idx][0] == "<COMMA>":
                  output.pop()
                  value = '-' + value
```

만약 '-'이 정수부에 포함된다면 기존에 output에 있던 산술연산자를 pop하고 정수에 '-'를 추가 했습니다. 초반에 언급했던 마지막 value 처리 때문에 음수가 제대로 처리되지 않을수도 있기에 마지막으로 처리해주는 코드를 추가 했습니다.

```
if len(output) > 1:
    if output[-1][0] == "<SIGNED INTEGER>" and output[-2][1] == '-':
        v = output.pop()[1]
        output.pop()
        output.append(("<SIGNED INTEGER>", '-' + v))
```

이렇게 예외처리가 된 output 리스트를 공백 토큰을 제외하고 파일에 옮겨 적습니다.

```
f = open("output.txt", 'w')
for token, v in output:
    # skip white space
    if token == "<WHITE SPACE>":
        continue
    f.write(token + " " + v + "\n")
```

사용한 test.java 파일

```
public class test {

        public static int main(String[] args) {
                int _aa_a = 0;
                char b_b123 = '8';
                boolean __cc = true;
                String d__d_0 = "Hello World 123";
                char if_123if = ' ';
                int _123if0 =-1 -0;
                int[] publice = {100, -10, 2};

                if(_aa_a <= 0) {
                        _aa_a = 3923 * 41 - -1- -1- 1-1 / 10 +2;
                }else if(_aa_a == 0){
                        b_b123 = '3';
                }

                while(__cc) {
                        __cc = false != false;
                }

                return 0;
        }

}
```

Output.txt

```
<KEYWORD> public
<KEYWORD> class
<IDENTIFIER> test
<LBRACE> {
<KEYWORD> public
<KEYWORD> static
<VARIABLE TYPE> int
<KEYWORD> main
<LPAREN> (
<VARIABLE TYPE> String
<LBRANKET> [
<RBRANKET> ]
<IDENTIFIER> args
<RPAREN> )
<LBRACE> {
<VARIABLE TYPE> int
<IDENTIFIER> _aa_a
<ASSIGNMENT OPERATOR> =
<SIGNED INTEGER> 0
<TERMINATING SYMBOL> ;
<VARIABLE TYPE> char
<IDENTIFIER> b_b123
<ASSIGNMENT OPERATOR> =
<SINGLE CHARACTER> '8'
<TERMINATING SYMBOL> ;
<VARIABLE TYPE> boolean
<IDENTIFIER> __cc
<ASSIGNMENT OPERATOR> =
<BOOLEAN STRING> true
<TERMINATING SYMBOL> ;
<VARIABLE TYPE> String
<IDENTIFIER> d__d_0
<ASSIGNMENT OPERATOR> =
<LITERAL STRING> "Hello World 123"
<TERMINATING SYMBOL> ;
<VARIABLE TYPE> char
<IDENTIFIER> if_123if
<ASSIGNMENT OPERATOR> =
<SINGLE CHARACTER> ' '
<TERMINATING SYMBOL> ;
<VARIABLE TYPE> int
<IDENTIFIER> _123if0
<ASSIGNMENT OPERATOR> =
<SIGNED INTEGER> -1
<ARITHMETIC OPERATOR> -
<SIGNED INTEGER> 0
<TERMINATING SYMBOL> ;
<VARIABLE TYPE> int
<LBRANKET> [
<RBRANKET> ]
<IDENTIFIER> publice
<ASSIGNMENT OPERATOR> =
<LBRACE> {
<TERMINATING SYMBOL> ;
<KEYWORD> if
<LPAREN> (
<IDENTIFIER> _aa_a
<COMPARISON OPERATOR> <=
<SIGNED INTEGER> 0
<RPAREN> )
<LBRACE> {
<IDENTIFIER> _aa_a
<ASSIGNMENT OPERATOR> =
<SIGNED INTEGER> 3923
<ARITHMETIC OPERATOR> *
<SIGNED INTEGER> 41
<ARITHMETIC OPERATOR> -
<SIGNED INTEGER> -1
<ARITHMETIC OPERATOR> -
<SIGNED INTEGER> -1
<ARITHMETIC OPERATOR> -
<SIGNED INTEGER> 1
<ARITHMETIC OPERATOR> -
<SIGNED INTEGER> 1
<ARITHMETIC OPERATOR> /
<SIGNED INTEGER> 10
<ARITHMETIC OPERATOR> +
<SIGNED INTEGER> 2
<TERMINATING SYMBOL> ;
<RBRACE> }
<KEYWORD> else
<KEYWORD> if
<LPAREN> (
<IDENTIFIER> _aa_a
<COMPARISON OPERATOR> ==
<SIGNED INTEGER> 0
<RPAREN> )
<LBRACE> {
<IDENTIFIER> b_b123
<ASSIGNMENT OPERATOR> =
<SINGLE CHARACTER> '3'
<TERMINATING SYMBOL> ;
<RBRACE> }
<KEYWORD> while
<LPAREN> (
<IDENTIFIER> __cc
<RPAREN> )
<LBRACE> {
<IDENTIFIER> __cc
<ASSIGNMENT OPERATOR> =
<BOOLEAN STRING> false
<COMPARISON OPERATOR> !=
<BOOLEAN STRING> false
<TERMINATING SYMBOL> ;
<RBRACE> }
<KEYWORD> return
```

```
<SIGNED INTEGER> 100
<COMMA> ,
<SIGNED INTEGER> -10
<COMMA> ,
<SIGNED INTEGER> 2
<RBRACE> }
```

```
<SIGNED INTEGER> 0
<TERMINATING SYMBOL> ;
<RBRACE> }
<RBRACE> }
```

수기로 고안했던 흔적들