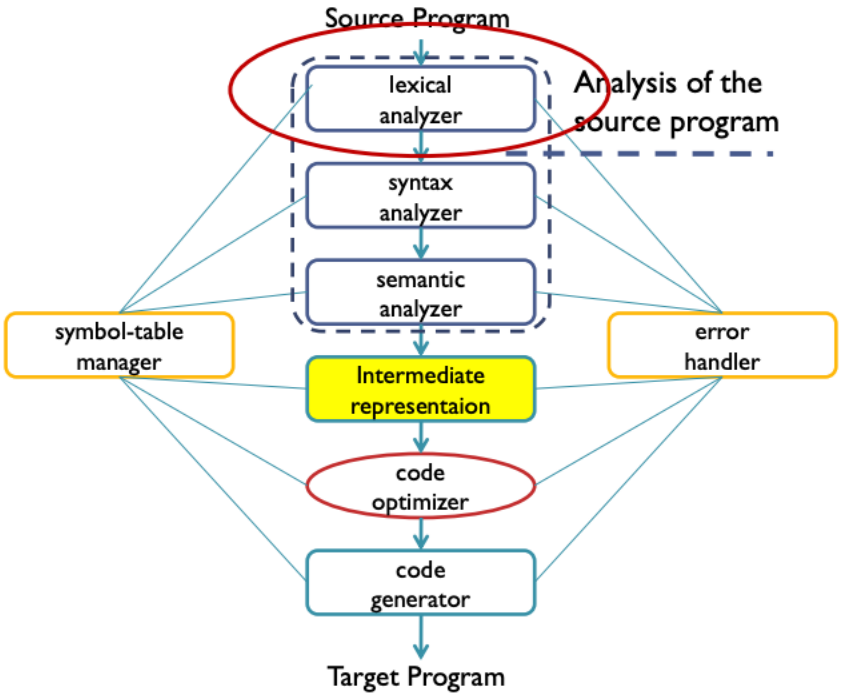
프로그래밍언어 Project2 Lexer 보고서

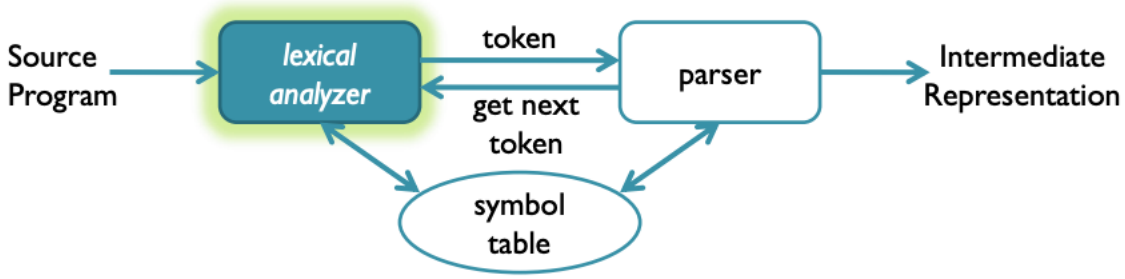
2017313213 박경태

1. 과제 개요

본 과제는 주어진 언어를 컴파일하는 컴파일러의 Lexer, Parser, Code Generator 를 설계하는 팀프로젝트이다. 컴파일러는 아래 사진과 같은 구조를 가지고 있으며, 아래의 사진단계로 설명하고자 한다.



2. Lexical Analyzer 설명



Lexical Analyzer 는 주어진 소스프로그램을 토큰으로 분리하여 parser 에 전달해주는 역할을 담당한다.

Code Stream 을 Token Stream 으로 바꿔야 하는데, 이 과정에서 유의할 점이 더러 있다.

예를 들어, 기존 C++ 문법같은 경우 pair<int,int>> 에서 >> 라는 닫힘 괄호를 "parenthesis"가 아닌 "operator" 역할로 오해하여 parser 에 전달시키면 안될 것이다. 이러한 오해는 Token stream 으로 변환하는 과정에서 해소되게 된다. 그리고 이러한 작업은 lexer 의 역할이다.
주어진 문법은 상대적으로 매우 간단한 편이라, 그런 오해가 벌어질 염려는 적다.

```python
import sys

class scanner():
    def __init__(self, code):
        self.code = code
        self.length = len(code)
        self.type = ['int', 'char']
        self.operator = ['>','==', '+', '*', '=']
        self.bracket = ["(", ")", "{", "}"]
        self.other_character = [",", ";"]
        self.statement = ['IF', 'THEN', 'ELSE', 'WHILE']
        self.exit = ["EXIT"]
        self.tokens = []
```

lexical.py 에서는 input 된 code 를 token 으로 변환하는 작업을 구현한다.
token 은 리스트로 구현되어있으며, 리스트에는 길이 2 의 리스트가 들어가게 된다.

최종적으로 분류하는 가능한 lexeme 속성의 종류는 아래와 같다.
number: 숫자를 의미
type: 자료형을 의미
statement: if, then, else, while 등의 구문을 의미
EXIT: 종료구문을 의미
word: 변수명을 의미
bracket: 열고닫는괄호
operator: 연산자
comma or semicolon: 괄호 및 세미콜론

우리가 a=110; 라는 문법을 받았는데, a=1 로 끝나면 안될 것이다.
따라서, 끝까지 읽어주는 read_full_digit 함수를 만든다.

```python
    def read_full_digit(self, idx):
        ret = 0
        while idx+ret < self.length and self.code[idx+ret].isdigit():
            ret+=1
        assert ret>0
        return self.code[idx:idx+ret]
```

마찬가지로, 우리가 str = alpha 을 입력받았는데 str=a 만 입력되면 안될 것이다.
따라서, read_full_string 함수도 만든다.

```
    def read_full_string(self, idx):
        ret=0
        while idx+ret<self.length and self.code[idx+ret].isalpha():
            ret+=1
        assert ret>0
        return self.code[idx:idx+ret]
```

또한, 비교연산자 ==의 경우 2 글자인 operator 이므로 동일한 operator 인 할당연산자 =와 구별되어야한다. 따라서 반드시 2 글자인 ==인지의 여부가 =인지의 여부의 판단보다 선행되어야만 한다. 그 작업을 is_equal_sign 에서 한다.

```
    def is_equal_sign(self, idx):
        if idx < self.length-1 and self.code[idx:idx+2]== '==':
            return True
```

lexer 역할을 담당하는 lexical 부분은 아래와 같다

```
def lexical(self):
    idx=0
    while idx<self.length:
        if self.code[idx].isdigit(): # 숫자가 들어오면
            token = self.read_full_digit(idx)
            self.tokens.append(['number', token])
            idx+=len(token)
        elif self.code[idx].isalpha(): # 알파벳이 들어오면
            token = self.read_full_string(idx)
            if token in self.type:
                self.tokens.append(['type : ', token])
            elif token in self.statement:
                self.tokens.append(['statement : ', token])
            elif token in self.exit:
                self.tokens.append(['EXIT : ', token])
            else:
                self.tokens.append(['word : ', token])
            idx+=len(token)
        elif self.code[idx] == ' ' or self.code[idx] == '\n' or self.code[idx] == '   ':
            idx+=1
        else:
            if self.code[idx] in self.bracket:
                self.tokens.append(['bracket : ', self.code[idx]])
                idx+=1
            elif self.code[idx] in self.operator:
                if self.is_equal_sign(idx):
                    self.tokens.append((['operator : ', '==']))
                    idx+=2
                else:
                    self.tokens.append(['operator : ', self.code[idx]])
                    idx+=1
            elif self.code[idx] in self.other_character:
                self.tokens.append(['comma, semicolon : ', self.code[idx]])
                idx+=1
            else:
                self.tokens.append(0)
                print("Lexical Error: Wrong Input{}".format(idx))
                sys.exit()
    assert len(self.tokens) > 0
```

숫자가 들어오면 반드시 숫자이므로 (변수는 숫자로 시작할 수 없다) number 로 받아들여준다.
알파벳으로 시작하면 여러 경우의 수가 가능하다.
type 또는 statement 또는 EXIT 또는 word 의 경우가 가능한데, 각각 읽힌 token 이 어느 것이냐에 따라 구분이 가능하다.
공백 또는 \n, \t 의 경우 무시한다. whitespace 는 문법에 무의미하다.
만일 숫자도 아니고 알파벳도 아니라면 연산자 또는 괄호나 특수문자일 것이다.
각각 케이스를 분류해준다.
그리고, tokens 배열에 append 해준 뒤 함수를 종료한다.
이로써 tokens 에는 symbol table 이 저장되게 되며, 이 symbol table 은 parser 로 넘어가게 된다.