2021 spring semester final project

1. Lexical

prog ::= word "(" ")" block ;  
decls ::= decls decl  
| ;  
decl ::= vtype word ";" ;  
vtype ::= int | char  
| ;  
block ::= "{" decls slist "}"  
| ;  
slist ::= slist stat  
| stat ;  
stat ::= IF cond THEN block ELSE block  
| word "=" expr ";"  
| EXIT expr ";"  
| ;  
cond ::= expr ">" expr ;  
expr ::= expr "+" fact  
| fact ;  
fact ::= num  
| word ;  
word ::= ([a-z] | [A-Z])\* ;  
num ::= [0-9]\*

The given grammar was put in grammar.txt. But we cannot use this grammar right away. We need to make a parse table for better approach to text that follow this grammar. To make **parse-table**, we need **FIRST** sets and **FOLLOW** sets. But these things do not just come. There are some traps that prevents us from getting it. One of them is the **left-recursion.**

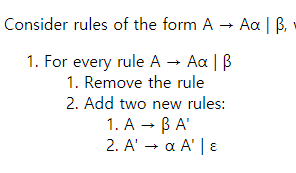
cycle을 제거하기 위해 grammar를 아래와 같이 바꾸었다.

prog ::= word "(" ")" block ;  
decls ::= decls decl  
| ;  
decl ::= vtype word ";" ;  
vtype ::= int | char  
| ;  
block ::= "{" decls slist "}"  
| ;  
slist ::= slist stat  
| ;  
stat ::= IF cond THEN block ELSE block  
| word "=" expr ";"  
| EXIT expr ";" ;  
cond ::= expr ">" expr ;  
expr ::= expr "+" fact  
| fact ;  
fact ::= num  
| word ;  
word ::= ([a-z] | [A-Z])\* ;  
num ::= [0-9]\*

slist와 stat이 변경되었다.

def remove\_recursion(raw\_grammar):  
 grammar = {}  
 for key in raw\_grammar.keys():  
 idx = [0, 0]  
 check = [False, False]  
 for i, value in enumerate(raw\_grammar[key]):  
 if key == value[0]:  
 idx[0] = i  
 check[0] = True  
 elif len(value) == 1:  
 idx[1] = i  
 check[1] = True  
 if check[0] and check[1]:  
 if raw\_grammar[key][idx[1]][0] == '':  
 grammar[key] = [[key+'\_1']]  
 else:  
 grammar[key] = [[raw\_grammar[key][idx[1]][0], key+'\_1']]  
 grammar[key+'\_1'] = [raw\_grammar[key][idx[0]][1:] + [key+'\_1'], ['']]  
 else:  
 grammar[key] = raw\_grammar[key]  
  
 return grammar

But before we find the **FIRST** set and **FOLLOW** set, we had to remove the left-recursion. If there is one, we have a cycle, that will always have non-terminals at the first. To remove that endless cycle, we must make additional non-terminal symbol.



By using, this method, **left recursion** was removed. Instead, we had new non-terminals for each **left recursion**. After that, we are now able to get the first set for each non-terminals. Since we had the grammar, we could have just got the sets by hand, but we feared we might make mistakes and thought it would be nice to have some functions that automatically makes one,

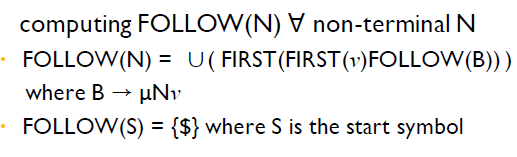
def find\_first\_sub(key, grammar):  
 first\_list = []  
 if key not in grammar.keys():  
 return [key]  
 else:  
 for values in grammar[key]:  
 first = find\_first\_sub(values[0], grammar)  
 if type(first) is list:  
 first\_list = first\_list + first  
 else:  
 first\_list.append(first)  
 return first\_list  
  
  
def find\_first(grammar):  
 first\_dic = {}  
 for key in grammar.keys():  
  
 first\_dic[key] = set()  
 for values in grammar[key]:  
 first = find\_first\_sub(values[0], grammar)  
  
 if first != '':  
 first\_dic[key] = first\_dic[key].union(set(first))  
 else:  
 first\_dic[key].add('')  
  
 return first\_dic

After getting the **First** sets for each non-terminals, we can get the **FOLLOW** sets. The FOLLOW sets are the sets of terminals that comes after any given non-terminals. So lets say, we have grammar that is

A -> aBC

Then since C comes after B, which is equal to C follows B, every terminals that comes after C and, C itself include the First sets are also following B. So **FIRST**(C) is included in the **FOLLOW**(B).

Likewise,

is the simplified rule.

def find\_follow\_sub(m\_key, grammar, first):  
 key\_list = list(grammar.keys())  
 follow = set()  
 for key in key\_list:  
 for value in grammar[key]:  
 for i in range(len(value)):  
 if value[i] == m\_key:  
 if i + 1 < len(value):  
 next\_f = set()  
 if value[i + 1] in key\_list:  
 next\_f = first[value[i + 1]]  
 else:  
 next\_f.add(value[i + 1])  
  
 if '' in next\_f and m\_key != key:  
 fol = find\_follow\_sub(key, grammar, first)  
 follow = follow.union(fol)  
 follow = follow.union(next\_f-{''})  
 else:  
 if m\_key != key:  
 fol = find\_follow\_sub(key, grammar, first)  
 if len(fol) == 0:  
 fol.add('$')  
 follow = follow.union(fol)  
 return follow  
  
  
def find\_follow(grammar, first):  
 follow\_dic = {}  
 key\_list = list(grammar.keys())  
 for m\_key in key\_list:  
 follow = find\_follow\_sub(m\_key, grammar, first)  
 if len(follow) == 0:  
 follow.add('$')  
 follow\_dic[m\_key] = follow  
  
 for i in follow\_dic:  
 print(i, follow\_dic[i])  
  
 return follow\_dic

By using the above method and rules, we know have the follow sets for the non-terminals. Both FIRST sets and FOLLOW sets includes only terminals.

Now since we have both the **FIRST** and **FOLLOW** sets, we can easily construct a **parse-table**. The row of the parse-table is the non-terminals that appears in the grammar, and the columns are the terminals, in this case, alphabets(upper and lower), digits, parenthesis, etc.

def make\_parsing\_table(grammar, terminal, non\_terminal, follow):  
 parsing\_table = []  
 for i in range(len(terminal)):  
 initial = []  
 for j in range(len(non\_terminal)):  
 initial.append(None)  
 parsing\_table.append(initial)  
 for key in grammar.keys():  
 idx1 = terminal.index(key)  
 fol = follow[key]  
 for value in grammar[key]:  
 first = find\_first\_sub(value[0], grammar)  
 for first\_val in first:  
 if first\_val != '':  
 parsing\_table[idx1][non\_terminal.index(first\_val)] = value  
 if '' in first:  
 for f in fol:  
 parsing\_table[idx1][non\_terminal.index(f)] = value  
 print(non\_terminal)  
 for i in parsing\_table:  
 print(i)  
 return parsing\_table

And inside the corresponding slots of table, we enter the grammar. For example,

B -> AC

In this case, A is the most left on the right side. And if non-terminal A has FIRST set as { + , alphabets}, then in the slot of coordinate (A, +) and (A, alphabets), the grammar ‘B-> AC’ is located.

After doing all of these and similar to the FOLLOW sets, we can complete the LL(1) parse-table. LL(1) mean, L : (reading input from Left to right), L : (Left most derivation) and digit 1 between parenthesis means that we will read one symbols at a time.

1. Cut and tokenize the text

main(){  
 int a;  
 int c;  
 int f;  
 char b;  
  
 b = 4;  
 a = b + 14;  
 f = 3 ;  
 c = 5;  
  
 IF a + 1 > c THEN {  
 a=4;  
 }  
 ELSE {  
 int k;  
 a = 22;  
 }  
 EXIT a;  
}

1. Make AST(Abstract Syntax Tree)

Since we know have both LL(1) parse table and tokenized code, we can construct AST, abstract syntax tree. But before we start using these, vital information, we need a node to construct any kind of tree.

class Node:  
 def \_\_init\_\_(self, data, parent, sibling\_order, val=None):  
 self.data = data  
 self.parent = parent  
 self.sibling\_order = sibling\_order  
 self.children = []  
 self.val = val  
  
  
 def make\_children(self, data):   
 for i, dat in enumerate(data):  
 self.children.append(Node(dat, self, i))  
 return self.children[0]  
  
 def get\_next(self):  
 sibling\_order = self.sibling\_order  
 node = self  
 parent = self.parent  
 grand\_parent = None  
 while True:  
 if parent is not None:  
 if len(parent.children)-1 == sibling\_order:  
 sibling\_order = parent.sibling\_order  
 node = parent  
 grand\_parent = parent.parent  
 else:  
 break  
 else:  
 return node  
 parent = grand\_parent.children[sibling\_order+1]  
 while len(parent.children) != 0:  
 node = parent.children[0]  
 return node  
  
 def get\_left(self):  
 node = self  
 if node.sibling\_order == 0:  
 node = node.parent  
 return node.parent.children[node.sibling\_order - 1]  
  
 def get\_right(self):  
 node = self  
 if node.sibling\_order > len(node.parent.children):  
 node = node.parent  
 return node.parent.children[node.sibling\_order + 1]  
  
 def get\_root(self):  
 node = self  
 while node.parent is not None:  
 node = node.parent  
 return node

The given code is class functioning as nodes of a tree. We have 5 attributes in the Node class. Data, parent, children, sibling order, val. Each attribute is used in controlling the nodes that changes as the data which came from analyzing the code.

First we make two stacks. One stack is for parser stack and other for the code. We put starting symbol and ‘$’ inside the parser stack. In the code stack, we insert all the tokens.

From start, we see the top of both stack, which has non-terminal at the parser stack and terminal at the code stack. Then using dose two as the coordinate, we find which grammar should be used.

If we find the grammar than we pop the parser stack and insert the corresponding non-terminalsl

We continue to do so until, the top of two stack are same, which is both terminals. Then we pop both side

We continue this sequence until both side is empty. For specifically when parser stack has only ‘$’ and emptied the code stack.

1. IR(Intermediat Representation)

Generator class로 만들었다.

1. Final Result(Assembly Code)

먼저, grammar를 LL(1)이 되게 수정한 뒤에

grammar의 recursion을 제거해주고

FIRST와 FOLLOW를 구해 parsing table을 만들었다.

그리고나서 이것을 통해 parsing tree를 만들었고,

이것으로 Intermediat Representation를 만들었다.

Tree가 완성되었다는 말은 사실상 모든게 끝이 났다.

어셈블리 코드로 바꿔주기 위해서는 각 statement에 대해 if문을 걸어서 ADD인지, IF인지 등등 구분을 해서 출력하면된다. 특히 goto 는 JUMP로 바뀔 것이다.

tree 및 parsing table이 정상적으로 만들어졌고,

Intermediat Representation 역시 정상적으로 만들어졌지만 시간상의 문제로 인해 어셈블리 코드로 변경까진 하지 못하엿다.

main함수를 실행시키면 std output으로 출력되는 부분으로 구현한 결과를 확인할 수 있다.

symbol table 함수에 보면 다 구현했는데 잘 안된다.