2021-1 PLCOMP

정태명 교수님

Compiler Project

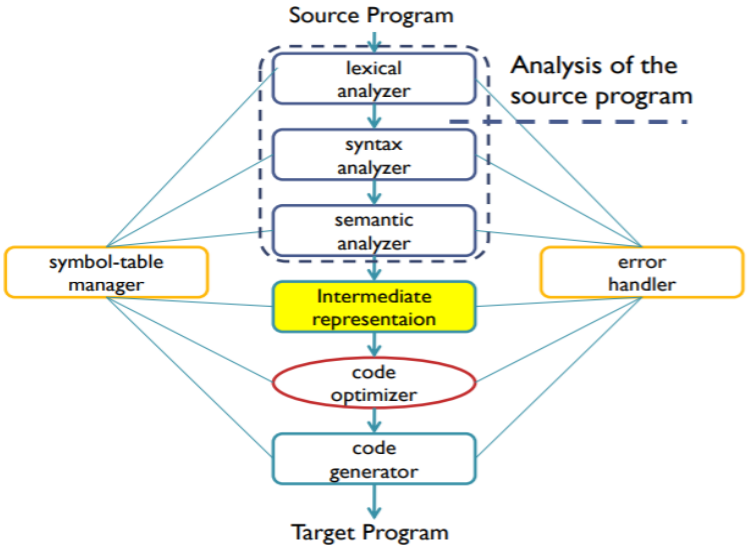
2017311666 윤혜진

2017312170 김도현

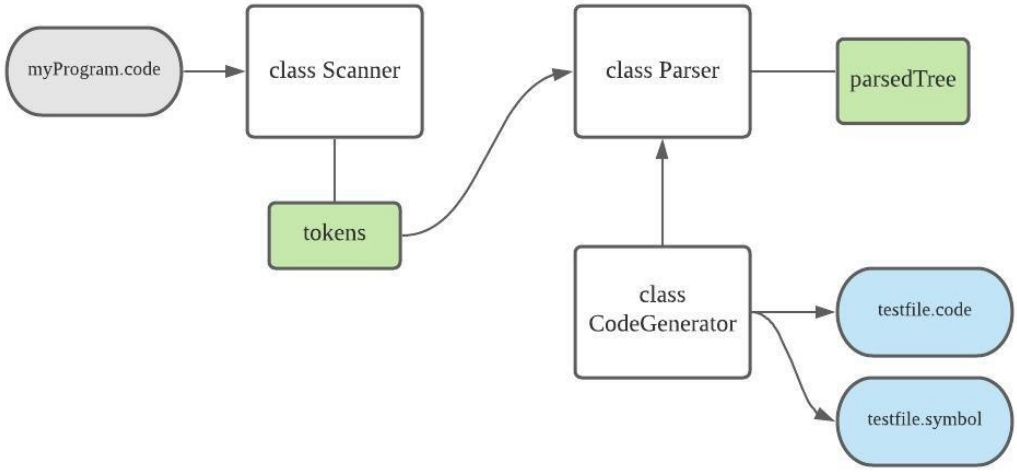
2017312782 김세영

1. 프로젝트 개요

특정한 언어로 쓰여진 프로그램을 target language로 변환해 작성하는 프로그램을 컴파일러라 한다. 소스 코드를 머신 코드로 바꾸어주는 프로그램이라는 의미로 통용되지만, 이외에도 다양한 타겟 언어를 가질 수 있다.



컴파일러의 동작은 위와 같은 단계를 따른다. 먼저 Lexical analyzer(Scanner)가 소스 프로그램을 문법적으로 의미를 가지는 단위인 토큰으로 나눈다. 토큰화 된 프로그램을 Parser가 의미 구조에 맞게 위계화한다. 이 작업이 Syntax/Semantic Analyzer에 해당한다. 만들어진 위계 구조를 Syntax Tree 등 Intermediate Representation으로 나타내 변환 작업을 더욱 수월하게 할 수 있다. 컴파일러가 문자열로 이루어진 소스 프로그램의 의미를 이해하는 과정에서, 다양한 최적화 기법이 적용되어 프로그램 효율성을 증대시키기도 한다. 마지막으로 Code generator가 앞선 단계에서 프로세스된 정보를 바탕으로 타겟 프로그램을 작성한다.



본 프로젝트에서는 간단한 컴파일러의 구현을 통해 강의 내용에 대한 이해를 깊이할 수 있었다. 프로그램은 리눅스 환경에서 Python3 언어로 작성되었다. 모듈의 간략한 설계는 위 다이어그램과 같다. 초록색으로 표현된 구조가 컴파일러에서 프로세싱하는 핵심 데이터이다.

Scanner에서 토큰화 된 소스코드가 tokens:List 형태로 Parser로 전달된다. 그 뒤 SLR Parser가 문법 일치여부를 판단한다. Accept되는 경우 파싱트리가 만들어진다. 트리가 만들어지면, 하위 클래스인 Code Generator가 출력 파일을 생성한다. 프로그램의 출력 파일은 타겟 언어로 작성된 testfile.code와 심볼테이블을 담은 testfile.symbol이다. 프로그램은 각 모듈 파일과 메인모듈인 compiler2021.py, 소스코드를 담은 myProgram.code가 포함된 디렉토리에서 아래와 같은 명령어 입력을 통해 실행할 수 있다.

```
~/Downloads/TeamH$ python3 compiler2021.py myProgram.code
```

2. Changed Grammar

컴파일러에서 SLR 파서를 이용하기 위해 주어진 문법에서 제거한 production rule이 있다.

1) vtype -> ε

decl -> vtype word ";"와 stat -> word "=" expr ";" 에서 발생하는 reduce 시의 **ambiguity**를 없애주기 위해 vtype -> ε 규칙을 삭제했다.

2) stat -> ε

이 규칙이 있을 경우 input string 파싱이 끝나지 않고 slist -> slist stat에서 ε이 무한 생성되는 루프에 빠질 수 있어 제거했다.

3. Scanner

	ID	VALUE
word	1	Value
(	2	-1
)	3	-1
;	4	-1
int	5	-1
char	6	-1
{	7	-1
}	8	-1
IF	9	-1
THEN	10	-1
ELSE	11	-1
=	12	-1
EXIT	13	-1
>	14	-1
+	15	-1
num	16	Value
prog	17	
decls	18	
decl	19	
vtype	20	
block	21	
slist	22	
stat	23	
cond	24	
expr	25	
fact	26	

스캐너에서는 코드의 각각의 요소들을 토큰화 시켜 리스트 형식으로 저장했다. 총 16가지 (word ( ) ; int char { } IF THEN ELSE = EXIT > + num)로 토큰화 하였다. 각각의 토큰들은 id 를 부여 하였으며 word와 num을 제외한 각 토큰의 value 값은 -1을 주었고, num 과 value에 값을 word의 경우 VALUE 에 값을 저장 하였다.

주어진 토큰의 값들이 겹쳐있거나 (ex aa+, aa11), 토큰에 ₩t 이나 ₩n 값이 딸려 나오는 경우 등 주어진 항목에 맞지 않는 값 들을 토큰화 시키는 경우들은 따로 처리 하였고, 나머지 값들을 리스트에 저장하였다.

4. Parser

Parser에서는 Scanner에서 넘겨준 토큰 list를 parsing table을 이용하여 tree의 형태로 바꾸어 Code generator에 전달한다. Parsing table은 아래의 사이트를 참고해 구성했다. 다음 두 사진은 Parsing table을 분할한 것이다.

http://jsmachines.sourceforge.net/machines/slr.html

state	ACTION															
	$	word	(	)	;	int	char	{	}	IF	THEN	ELSE	"="	EXIT	>	"+"	num
0		shift 1															
1			shift 2														
2				shift 3													
3	reduce 7	reduce 7						shift 5	reduce 7	reduce 7		reduce 7		reduce 7			
4	accept																
5		reduce 2				reduce 2	reduce 2			reduce 2				reduce 2			
6		shift 12				shift 14	shift 15			shift 11				shift 13			
7		shift 12							shift 16	shift 11				shift 13			
8		reduce 1				reduce 1	reduce 1			reduce 1				reduce 1			
9		reduce 9							reduce 9	reduce 9				reduce 9			
10		shift 18															
11		shift 23															shift 22
12													shift 24				
13		shift 23															shift 22
14		reduce 4															
15		reduce 5															
16	reduce 6	reduce 6							reduce 6	reduce 6		reduce 6		reduce 6			
17		reduce 8							reduce 8	reduce 8				reduce 8			
18					shift 26												
19											shift 27						
20															shift 28	shift 29	
21					reduce 15						reduce 15				reduce 15	reduce 15	
22					reduce 16						reduce 16				reduce 16	reduce 16	
23					reduce 17						reduce 17				reduce 17	reduce 17	
24		shift 23															shift 22
25					shift 31											shift 29	
26		reduce 3				reduce 3	reduce 3			reduce 3				reduce 3			
27	reduce 7	reduce 7						shift 5	reduce 7	reduce 7		reduce 7		reduce 7			
28		shift 23															shift 22
29		shift 23															shift 22
30					shift 35											shift 29	
31		reduce 12							reduce 12	reduce 12				reduce 12			
32												shift 36					
33											reduce 13					shift 29	
34					reduce 14						reduce 14				reduce 14	reduce 14	
35		reduce 11							reduce 11	reduce 11				reduce 11			
36	reduce 7	reduce 7						shift 5	reduce 7	reduce 7		reduce 7		reduce 7			
37		reduce 10							reduce 10	reduce 10				reduce 10			

state	GOTO									
	prog	decls	decl	vtype	block	slist	stat	cond	expr	fact
0										
1										
2										
3					4					
4										
5		6								
6			8	10		7	9			
7							17			
8										
9										
10										
11								19	20	21
12										
13									25	21
14										
15										
16										
17										
18										
19										
20										
21										
22										
23										
24									30	21
25										
26										
27					32					
28									33	21
29										34
30										
31										
32										
33										
34										
35										
36					37					
37										

Parsing table에서의 Reduction list는 다음과 같다.

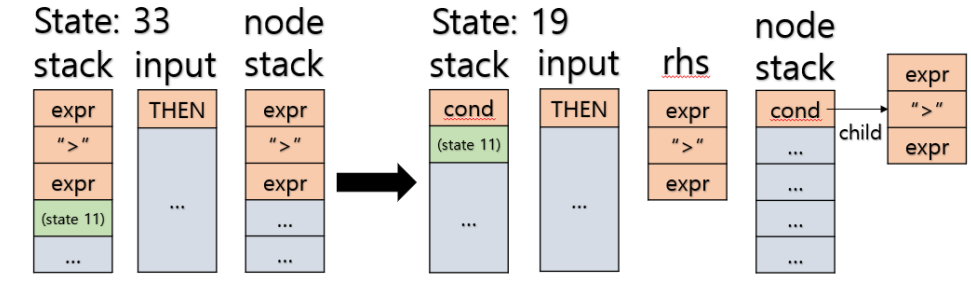
0. prog -> word aopen aclose block
1. decls -> decls decl
2. decls -> ε
3. decls -> vtype word ";"
4. vtype -> int
5. vtype -> char
6. block -> bopen decls slist bclose
7. block -> ε
8. slist -> slist stat
9. slist -> stat
10. stat -> IF cond THEN block ELSE block
11. stat -> word "=" expr ";"
12. stat -> EXIT expr ";"
13. cond -> expr ">" expr
14. expr -> expr "+" fact
15. expr -> fact
16. fact -> num
17. fact -> word

메인에서 parser.parse()를 실행하면, Scanner에서 넘겨받은 input list를 탐색한다. 현재 state 번호와 현재 input 종류의 번호를 parsingTable의 row, col 인자로 하였을 때의 결과값에 따라 reduce/shift/error&accept 처리한다.

parsingTable의 값은 -1은 reduce 0을 실행하고 accept, 0은 error, 1이상 99 이하는 그 state num으로 shift, 100초과는 reduce (값-100)을 실행한다.

shift는 [state, input]을 stack에 넣고, input에서 제거한다.

reduce a는 a번째 reduction을 실행합니다. stack에서 rhs의 개수만큼 pop해주고, 트리를 생성하기 위한 노드 스택에 저장되어있던 top들과 비교한다. 만약 다르다면 error로 취급한다. rhs를 lhs 노드의 자식노드들로 만들고, 노드 스택에 push한다. 마지막으로 stack에 parsingTable[stack.top.state][lhs number]를 state로 하는 lhs 정보를 push한다. 아래는 reduce의 예시이다.



error는 parser.error를 True로 바꿔주며, input 탐색을 멈춘다. 이 경우 5의 code generator는 parser.accept == false임을 확인하고 타겟 코드를 작성하는 대신 에러 메세지를 출력한다. accept는 reduce 0을 실행한 뒤, parser.accept를 True로 바꿔주며, input 탐색을 그만 둔다. 모든 input을 탐색한 후에는 parsedTree를 노드 스택의 top으로 바꾸어 준다.

5. Code generator

```
40  class Node(object):
41      def __init__(self, type):
42          self.type = type
43          self.children = []
44
45      def add_child(self, children):
46          for i in range(0, len(children)):
47              self.children.append(children[i])
```

입력이 accept 되었다면, 파서가 만든 Intermediate Representation인 parsedTree를 이용해 코드 제너레이터가 testfile.code를 작성한다. parsedTree는 항상 starting symbol인 prog를 루트로 가진다. Parser.py에서 정의된 트리 노드는 위와 같이 구성되어 있으며, type에서 노드가 Grammar의 어떤 Non-terminal-prog, decls, decl 등-에 속하는지 알 수 있다. 이를 활용해, 문법이 규정한대로 변수와 동작을 구분하며 레지스터를 사용해 파싱된 스트링의 의미를 주어진 pseudo-instruction으로 재구성 할 수 있다. codegenerator 클래스는 다음과 같은 변수들을 가진다. registers와 scope, symbolTable은 프로그램 실행에 있어 의미있는 역할을 한다.

```
class codegenerator(parser):

    def __init__(self, tokens):
        super(codegenerator, self).__init__(tokens)

        self.registers = [ 0 ]      # occupied 1 or not 0
        self.totalRegs = 1          # counts the total used registers

        self.scope = [ "global" ]   # stores current scope info
        self.symbolTable = {}       # stores the symbol info
                                    # varname: { scope1: { type: #, register: # }, scope2: { type, register }, ... }
        self.dtype = { 5: "int ", 6: "char" }   # for testfile.symbol.write()
```

```
# returns register# that is available at the current point
def regAlloc(self):…

# free registers that are no longer needed
def regFree(self, regNum):…

# initial call for traversing the parsedTree
def targetCall(self, t, s):
    if self.accept == False:
        print("\nError: Parser rejected the input string")
        quit()
    self.targetCode(self.parsedTree, t, s)
```

regAlloc()과 regFree(regNum)은 레지스터 사용 최적화를 돕는 함수이다. self.registers 리스트를 이용해 현재 레지스터의 점유/사용 여부를 관리하고, regAlloc은 새로운 변수가 들어왔을 때 사용할 레지스터 번호를 리턴해준다. regFree는 더 이상 사용되지 않는 변수를 담은 레지스터를 free해준다.

실제 code generation은 targetCall, targetCode 함수에서 일어나는데, targetCode는 트리

루트, targetcode과 symbol의 두 출력파일 디스크립터 t, s를 인자로 받으며, 재귀적 dfs 방식으로 children 노드들을 탐색한다. 노드의 type에 따라 동작이 달라지는데, 그 각각에 대한 pseudo code는 다음과 같다.

1. **prog** t.write("BEGIN word₩n") → self.scope.append(word) → targetCode(node.children[3]==block) → s.write(prog info) → t.write("END word₩n")
2. **block, decls, slist** for child in children: targetCode(child, t, s)
3. **decl** regAlloc for word → t.write(LD Reg value) → s.write(word info)
4. **stat**

    4-1. **IF cond THEN block ELSE block**

    표현의 논리적 흐름을 따라 라벨과 JUMP를 사용해 타겟코드를 작성했다.

    cond evaluation: regAlloc for cond → **exprEval()** for expr on the both side → t.write(LT, JUMPT, JUMPF)

    Label construction for THEN and ELSE block: t.write("L1/L2") + 2. → free variables inside the block(scope)

    4-2. **word = expr**

    regAlloc or regSearch for expr → t.write(ST)

    4-3. **EXIT**

    EXIT this scope and free registers

expr가 등장하는 경우에는 이를 처리하는 별도의 함수를 만들어 이용했다. 레지스터 할당과 표현의 연산을 가독성있게 나타내기 위해서이다. Expressioon evaluation 결과는 IF문의 조건이나 변수에 저장 될 값으로 이용되기 때문에 해당 레지스터 번호를 리턴한다.

```
# evaluates expression and returns the result register
def exprEval(self, exprNode, t, s):

    # expr -> fact
    if exprNode.children[0].type[0] == 26:
        # fact -> num | word
        newReg = self.regAlloc()
        t.write(f"LD\tReg#{newReg}, {exprNode.children[0].children[0].type[1]}\n")
        return newReg

    # expr -> expr + fact
    else:

        tReg = self.exprEval(exprNode.children[0], t, s)
        fReg = self.regAlloc()
        t.write(f"LD\tReg#{fReg}, {exprNode.children[2].children[0].type[1]}\n")
        t.write(f"ADD\tReg#{tReg}, Reg#{tReg}, Reg#{fReg}\n")
        return tReg
```

6. Test Outputs

컴파일러의 실행결과이다. 주어진 문법으로 example()을 작성하여 우분투 vscode에서 1.에 언급한 명령어를 통해 프로그램을 실행했다. 가독성을 위해 input/output 코드에 줄바꿈을 추가했다.

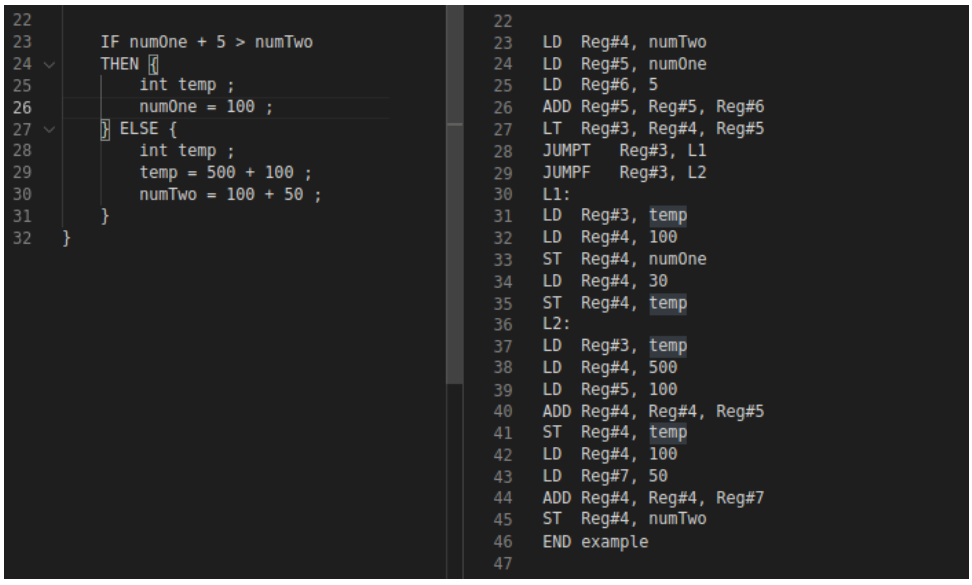
myProgram.code

```
example ( ) {
    int numOne ;
    int numTwo ;
    char temp ;

    numOne = 10 ;
    numTwo = 15 ;
    numOne = 5 ;
    temp = hello ;
```

testfile.code

```
BEGIN example
LD  Reg#0, numOne
LD  Reg#1, numTwo
LD  Reg#2, temp

LD  Reg#3, 10
ST  Reg#3, numOne

LD  Reg#3, 15
ST  Reg#3, numTwo

LD  Reg#3, 5
ST  Reg#3, numOne

LD  Reg#3, hello
ST  Reg#3, temp

LD  Reg#3, bye
ST  Reg#3, temp
```

decls에서 정의된 변수에 레지스터가 할당되는 것을 확인할 수 있고, regFree 함수로 인해 연속된 word = expr 실행이 Reg#3만을 사용해 이루어진다는 점을 알 수 있다.



```
    IF numOne + 5 > numTwo
    THEN {
        int temp ;
        numOne = 100 ;
    } ELSE {
        int temp ;
        temp = 500 + 100 ;
        numTwo = 100 + 50 ;
    }
}
```

```
LD  Reg#4, numTwo
LD  Reg#5, numOne
LD  Reg#6, 5
ADD Reg#5, Reg#5, Reg#6
LT  Reg#3, Reg#4, Reg#5
JUMPT   Reg#3, L1
JUMPF   Reg#3, L2
L1:
LD  Reg#3, temp
LD  Reg#4, 100
ST  Reg#4, numOne
LD  Reg#4, 30
ST  Reg#4, temp
L2:
LD  Reg#3, temp
LD  Reg#4, 500
LD  Reg#5, 100
ADD Reg#4, Reg#4, Reg#5
ST  Reg#4, temp
LD  Reg#4, 100
LD  Reg#7, 50
ADD Reg#4, Reg#4, Reg#7
ST  Reg#4, numTwo
END example
```

다음은 IF cond THEN block ELSE block이다. line 23~27까지 cond에 대한 로드, 연산, 평가 작업이 이루어지고 있다. L1과 L2는 각각 THEN { }, ELSE { } 스코프의 코드 동작을 담은 라벨이다. temp 변수는 이름이 중복되지만 스코프에 따라 새롭게 레지스터에 로드/프리되며 사용된다. 아래 심볼테이블에서 스코프 별 temp 변수를 확인할 수 있다.

testfile.symbol

```
********************************************************************************
Name            Type            Scope
********************************************************************************
example         prog            ['global', 'example']
numOne          int             ['global', 'example']
numTwo          int             ['global', 'example']
temp            char            ['global', 'example']
temp            int             ['global', 'example', 'IF-THEN']
temp            int             ['global', 'example', 'IF-ELSE']
```

마지막으로 console에 표시되는 output이다. 실행이 끝나면 output file들이 생성되고, 콘솔에는 토큰화된 리스트와 파싱 과정, 사용된 레지스터 개수가 출력된다. input string이 reject된 경우 에러 메세지가 출력된다.

```
state num: 24 goto_num: 26 sNum_to_stack: 21
state num: 24 goto_num: 25 sNum_to_stack: 30
state num: 6 goto_num: 23 sNum_to_stack: 9
state num: 6 goto_num: 22 sNum_to_stack: 7
state num: 24 goto_num: 26 sNum_to_stack: 21
state num: 24 goto_num: 25 sNum_to_stack: 30
state num: 7 goto_num: 23 sNum_to_stack: 17
state num: 6 goto_num: 22 sNum_to_stack: 7
state num: 27 goto_num: 21 sNum_to_stack: 32
state num: 5 goto_num: 18 sNum_to_stack: 6
state num: 6 goto_num: 20 sNum_to_stack: 10
state num: 6 goto_num: 19 sNum_to_stack: 8
state num: 5 goto_num: 18 sNum_to_stack: 6
state num: 24 goto_num: 26 sNum_to_stack: 21
state num: 24 goto_num: 25 sNum_to_stack: 30
state num: 29 goto_num: 26 sNum_to_stack: 34
state num: 24 goto_num: 25 sNum_to_stack: 30
state num: 6 goto_num: 23 sNum_to_stack: 9
state num: 6 goto_num: 22 sNum_to_stack: 7
state num: 24 goto_num: 26 sNum_to_stack: 21
state num: 24 goto_num: 25 sNum_to_stack: 30
state num: 29 goto_num: 26 sNum_to_stack: 34
state num: 24 goto_num: 25 sNum_to_stack: 30
state num: 7 goto_num: 23 sNum_to_stack: 17
state num: 6 goto_num: 22 sNum_to_stack: 7
state num: 13 goto_num: 26 sNum_to_stack: 21
state num: 13 goto_num: 25 sNum_to_stack: 25
state num: 7 goto_num: 23 sNum_to_stack: 17
state num: 6 goto_num: 22 sNum_to_stack: 7
state num: 36 goto_num: 21 sNum_to_stack: 37
state num: 7 goto_num: 23 sNum_to_stack: 17
state num: 6 goto_num: 22 sNum_to_stack: 7
state num: 3 goto_num: 21 sNum_to_stack: 4
state num: 0 goto_num: 17 sNum_to_stack: 0
accept: True

Check the target code in testfile.code
Check the symbol table in testfile.symbol

Total 8 registers have been used for execution
```