COMPILER PROJECT

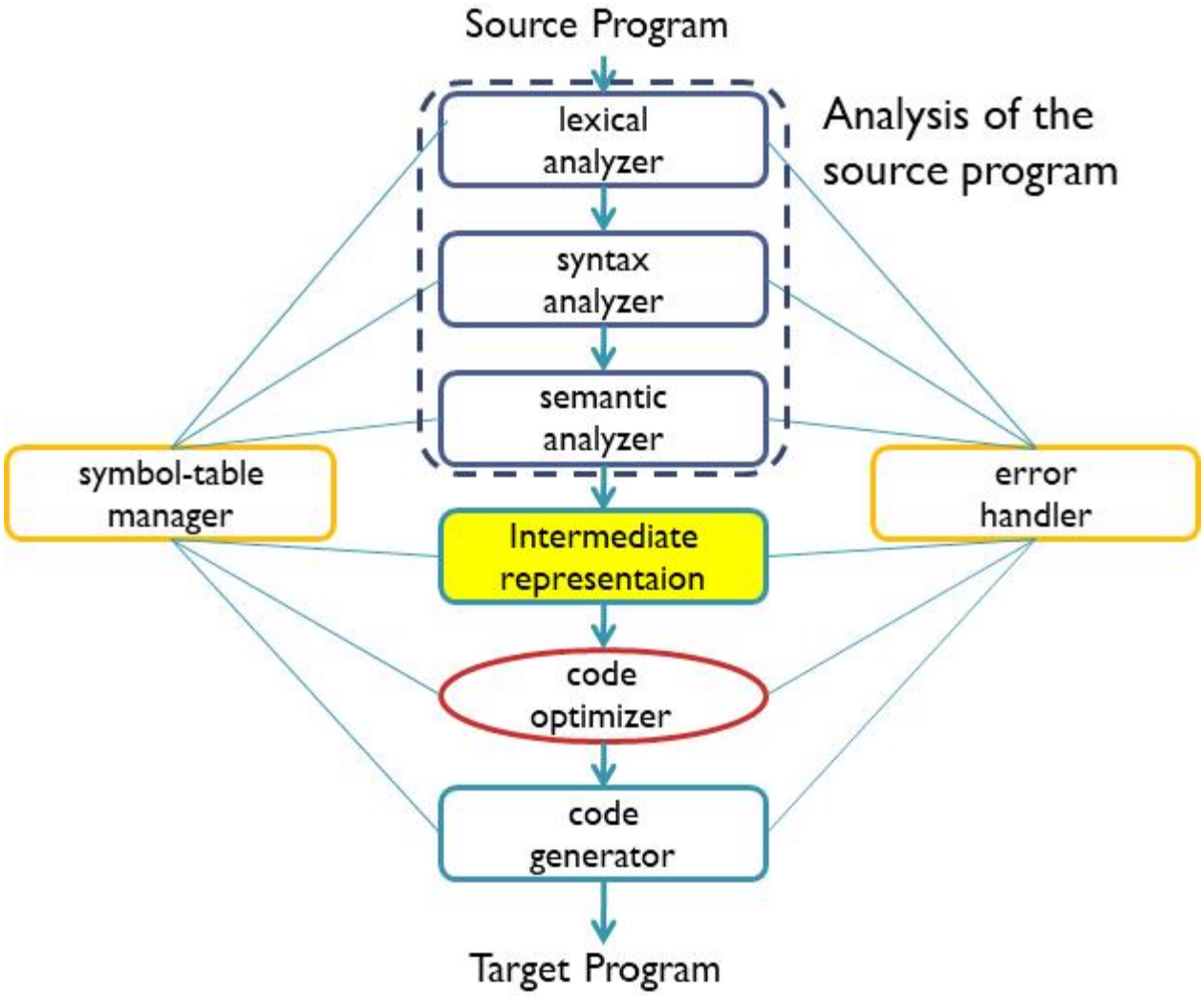
과목 : 프로그래밍 언어
담당교수 : 정태명 교수님

20조
조원 : 2014310530 박강인
2016310586 박상민
2013311898 박재훈
2014313329 유형진

목차

1. 머리말
2. Compiler 설계
3. Compiler 구현
    3.1 Lexer(Scannar)
    3.2 Parser
    3.3 Intermediate Representation
    3.4 Symbol Table
    3.5 Code Generator
4. Compiler 실행
5. 부록

1. 머리말



컴파일러는 특정한 프로그래밍 언어를 같은 의미를 가지는 타겟 프로그래밍 언어로 바꾸는 프로그램이다. 본 프로젝트는 위와 같은 컴파일러의 6가지의 단계를 lexer, parser, intermediate representation, code generator, 4가지 단계로 묶어서 진행한다. lexical analyzer는 lexer로 syntax analyzer와 semantic analyer는 parser로 inetermediate representation은 그대로 두고 code optimizer는 code generator와 합쳐서 하나의 기능으로 만든다. Lexer는 소스코드를 token으로 분해하고 Parser는 각 token을 받아와 grammar에 대조하여 parsing tree를 만든다. 해당 parsing tree를 타겟 코드를 만들기 위해 필요한 intermediate representation으로 바꾸고 code generator에서는 소스코드와 동등한 의미를 가지고 있는 타겟 코드를 intermediate representation으로부터 생성한다.

2. Compiler 설계

입력받은 Input Program을 Lexer를 통해 Token으로 그룹지어 쪼개어 주어야 한다. LL(1) Parser를 이용하여 Lexer로부터 받은 Token들을 Parse Tree로 구현한 후 Parse Tree를 Intermediate Representation(IR)중 하나인 Abstract Syntax Tree(AST)로 변환하면서 Symbol Table도 같이 작성하도록 한다. 마지막으로 Code Generator를 이용하여 IR를 Instruction set에 맞는 Pseudo Code를 생성하며 Compile이 마무리되게 한다.

Grammar에서는 프로그램에 진입점을 정의하고 있지 않다. 그래서 해당 Grammar를 함수의 집합을 읽는 Grammar로 설정하고 정의된 함수들 중에 main이라는 이름을 가진 함수를 진입점으로 정했다. Parser에서는 함수들의 집합을 인식하고 Parsing Tree를 만들면 Intermediate Representation으로 변환하는 과정에서 각 함수에 대한 Intermediate Representation을 따로 만들고 Code Generator에서 main을 중심으로 Pseudo Code를 생성한다.

Compiler를 구현하기 위해 주어진 Grammar를 다듬어야 한다. LL(1) Parser를 통해 Parse Tree를 생성하기 위해서는 주어진 Grammar가 Left recursion, Left factoring, Ambiguity가 없어야 한다. 주어진 Grammar를 변경한 과정은 다음과 같다.

수정 전	수정 후
Ambiguity와 이해하기 어려운 Grammar 수정	
prog ::= vtype word "(" words ")" block ; decls ::= decls decl | ; decl ::= vtype words ";" ; words ::= words "," word | word ; vtype ::= int | char | ; block ::= "{" decls slist "}" | ; slist ::= slist stat | stat ; stat ::= IF cond THEN block ELSE block | WHILE cond block | word "=" expr ";" | word "(" words ")" ";" | RETURN expr ";" | ; cond ::= expr ">" expr | expr "==" expr ; expr ::= term | term "+" term ; term ::= fact | fact "*" fact ; fact ::= num | word ; word ::= ([a-z] | [A-Z])* ; num ::= [0-9]*	prog -> vtype word (words) block decls -> decls decl | ε // | 이후 빈 공간에 ε 추가 decl -> vtype words ; words -> words , word | word vtype -> int | char // | 이후 빈 공간 제거 (vtype에 ε이 들어갈 경우 Ambiguous Grammar가 됨) block -> { decls slist } // | 이후 빈 공간 제거 (ε이 들어갈 경우 WHILE이나 IF ELSE 구문에 state가 없는 경우를 허용하게 됨) slist -> slist stat | stat stat -> IF cond THEN block ELSE block | WHILE cond block | word = expr ; | word (words) ; | RETURN expr ; // | 이후 빈 공간 제거 (stat에 ε이 들어갈 경우 Ambiguous Grammar가 됨) cond -> expr > expr | expr == expr expr -> term | term + term term -> fact | fact * fact fact -> num | word word -> ([a-z] | [A-Z])* num -> [0-9]*

여러 개의 Function 인식을 위한 Grammar 추가

```
start -> prog prog_
prog -> vtype word ( words ) block
prog_ -> vtype word ( words ) block prog_ | ε
```

Left recursion && Left factoring 수정

```
start -> prog prog_
prog -> vtype word ( words ) block
prog_ -> vtype word ( words ) block prog_ | ε
decls -> decls decl | ε
// Left recursion 제거 'decls'
decl -> vtype words ;
words -> words , word | word
// Left recursion 제거 'words'
vtype -> int | char
block -> { decls slist }
slist -> slist stat | stat
// Left recursion 제거 'slist'
stat -> IF cond THEN block ELSE block
      | WHILE cond block | word = expr ;
      | word ( words ) ;   | RETURN expr ;
// Left factoring 제거 'word'
cond -> expr > expr | expr == expr
// Left factoring 제거 'expr'
expr -> term | term + term
// Left factoring 제거 'term'
term -> fact | fact * fact
// Left factoring 제거 'fact'
fact -> num | word
```

```
start -> prog prog_
prog -> vtype word ( words ) block
prog_ -> vtype word ( words ) block prog_ | ε
decls -> decls_
decls_ -> decl decls_ | ε
decl -> vtype words ;
words -> word words_
words_ -> , word words_ | ε
vtype -> int | char
block -> { decls slist }
slist -> stat slist_
slist_ -> stat slist_ | ε
stat -> IF cond THEN block ELSE block
      | WHILE cond block | word stat_
      | RETURN expr ;
stat_ -> = expr ; | ( words ) ;
cond -> expr cond_
cond_ -> > expr | == expr
expr -> term expr_
expr_ -> ε | + term
term -> fact term_
term_ -> ε | * fact
fact -> num | word
```

Compiler 구현에는 C++ 언어를 사용하였다.

3. Compiler 구현

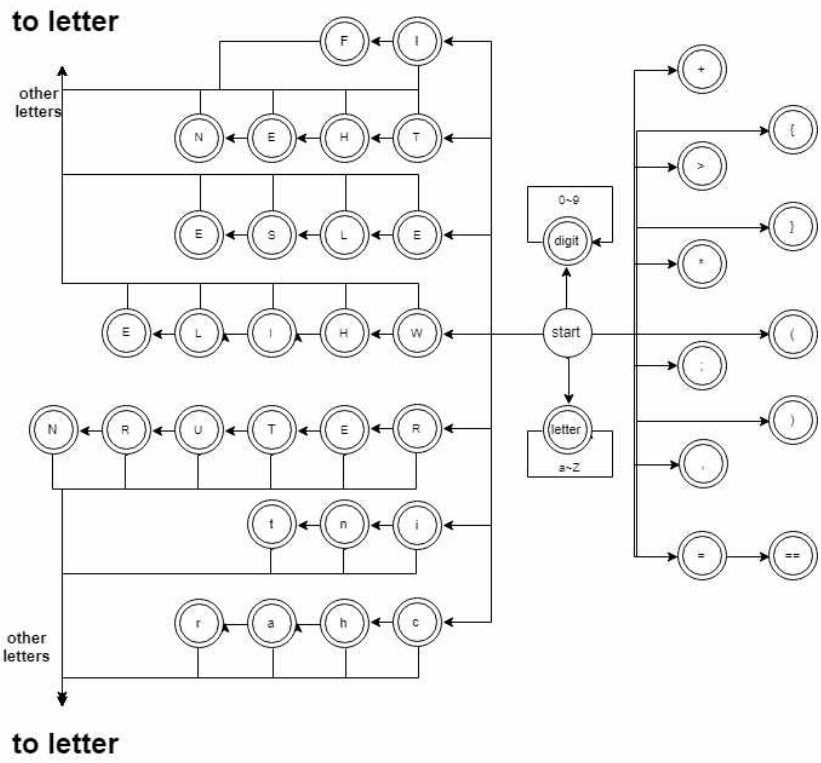
3.1 Lexer(Scannar)

Lexer에서는 소스 코드를 각각 token으로 잘라 Parser에게 보내주기 위해서 미리 정의된 keyword와 operator, seperator, identifier, literal을 regular expression으로 먼저 표현했다.

IDENTIFIER	([A-Z]|[a-z])+
KEYWORD	IF | THEN | ELSE | WHILE | RETURN | int | char
OPERATOR	+ | * | = | == | >
SEPERATOR	(|) | { | } | ; | ,
LITERAL	([0-9])+

그 후 위 token들을 전부 Accept할 수 있는 DFA를 regular expression으로부터 변환했다. 아래 DFA는 각 상태에 적혀있는 문자가 이전 state에서 넘어올 수 있는 input 문자이다. letter state는 start state로부터 움직일 수 있는 letter state 이외의 다른 state의 input 문자를 제외한 문자가 start state에서의 input이고 letter state에서는 a~Z까지의

모든 문자 input에 대해 다시 letter state로 이동한다. digit state은 0부터 9까지의 자연수를 input으로 받고 digit state로 돌아간다.



위 DFA는 start state를 제외한 모든 state에서 accept가 가능한데, 이 DFA는 모든 소스코드에 대해 accept를 하려는 것이 아니라 각 token별로 accept를 하기 때문에 accept condition을 삽입해야 한다. 현재 state에서 다음 input을 받고 그 input으로 갈 수 있는 state가 없으면 accept하고 input을 buffer에 저장해 둔 뒤 start state로 돌아가서 buffer에 있는 input으로 다시 시작한다. 소스 코드를 읽다보면 필요없는 문자인 개행문자나 수평탭문자, 공백문자들이 등장하는데 해당 문자들은 DFA의 input으로 넣지 않고 전부 버린다. 위 DFA는 저장한 값을 해당 input에 대해 이동할 목표 state로 하여 state(40) x input(73) 행렬로 구현했다. state set은 위 DFA와 같고 inpu set은 아래와 같다.

INPUT SET (73)
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
+ = * > ( ) { } ; ,

input set에 존재하지 않는 공백문자를 제외한 문자를 입력받으면 lexer에서 error를 출력한다. Parser는 매번 다음 token을 Lexer에게 요구하고 Lexer는 DFA를 이용하여 문자를 읽어들이면서 token이 accept되면 Parser에게 넘겨준다.

3.2 Parser

LL(1) Parser를 구현하기 위해서는 Grammar의 FIRST SET과 FOLLOW SET을 구하고, 이를 이용하여 Parsing table을 제작하여야 한다. Grammar의 FIRST SET과 FOLLOW SET은 다음과 같다.

Non-terminal	FIRST SET	FOLLOW SET
start	{ int, char }	{ $ }
prog	{ int, char }	{ $, int, char }
prog_	{ ε, int, char }	{ $ }
decls	{ ε, int, char }	{ IF, WHILE, word, RETURN }
decls_	{ ε, int, char }	{ IF, WHILE, word, RETURN }
decl	{ int, char }	{ int, char, IF, WHILE, word, RETURN }
words	{ word }	{), ; }
words_	{ , , ε }	{), ; }
vtype	{ int, char }	{ word }
block	{ { }	{ $, int, char, ELSE, }, IF, WHILE, word, RETURN }
slist	{ IF, WHILE, word, RETURN }	{ } }
slist_	{ ε, IF, WHILE, word, RETURN }	{ } }
stat	{ IF, WHILE, word, RETURN }	{ }, IF, WHILE, word, RETURN }
stat_	{ =, (}	{ }, IF, WHILE, word, RETURN }
cond	{ num, word }	{THEN, { }
cond_	{ >, == }	{THEN, { }
expr	{ num, word }	{ ;, >, ==, THEN, { }
expr_	{ ε, + }	{ ;, >, ==, THEN, { }
term	{ num, word }	{ ;, >, ==, THEN, {, + }
term_	{ ε, * }	{ ;, >, ==, THEN, {, + }
fact	{ num, word }	{ ;, >, ==, THEN, {, +, * }

구해진 FIRST SET과 FOLLOW SET을 이용하여 Parse Table을 작성하면 <부록-1>와 같다. Parse Table은 int_Stack 타입의 2차원 배열로 저장하였다.

Parser의 Stack은 ParseTree_Node Type으로, Stack이 빌 때까지 한 Node씩 Pop하여 처리한다. 해당 Node가 Terminal이면, Terminal의 Data를 행으로 Lexer로부터 받은 Input Token의 데이터를 열로 사용하여 Parse Table에 접근한다. 만약 Parse Table의 Stack이 비었다면 더 이상 진행 할 수 있는 Transition이 없는 것이기 때문에 Reject하며 Syntax 에러를 호출한다. 이외의 경우 Parse Table의 Stack의 내용을 Parser의 Stack에 Node를 생성하여 담는다. 이 후 새로 생성한 Node들을 Child Node로 연결시켜 주어 Parse Tree 구조를 형성한다.

Pop한 Node가 NonTerminal이면, 해당 Node의 Data에 Input Token의 Data를 저장하고 (AST 및 Symbol Table 구현을 위함) 다음 Token을 Lexer로부터 받아온다. 모든 Input을 처리하고(Input Token의 마지막 값은 $이다) Parser의 Stack이 비게 되면 Parsing을 종료한다.

3.3 Symbol Table

Symbol Table은 type, scope, block level, data size, address 총 5가지의 정보를 저장하고 STL의 hash map을 사용하여 구현하였다. block level은 inner block의 깊이 정보를 담고 있다. block level 값이 작을수록 global 영역에 더 가깝고 값이 클수록 내부에 있다. 구현에는 STL에서 제공하는 hash map인 unordered map을 이용했다. key값을 변수명으로 하여 검색하는데 다른 scope에 존재하는 동명의 변수를 구분하기 위해서 hash map의 value 값으로 STL에서 제공하는 이진탐색트리인 map을 삽입했다. map의 key값으로는 block id를 이용한다. parsing하는 과정에서 각 block에 id값을 부여하여 변수의

scope를 block단위로 알 수 있게 하고 block id와 block level을 이용해 상위 scope에 있는 변수를 이용할 수 있게 하고 다른 scope에 있는 변수를 제한한다.

3.4 Intermediate Representation

Intermediate Represntation으로는 Abstract Syntax Tree를 이용했다. AST에선 start -> prog + prog_로 이루어진 각 함수들에 대해 각자 다른 AST를 구한다. Parser에서 보낸 parsing tree를 전부 탐색하면서 non terminal을 전부 지우고 leaf node에는 terminal 중 변수명인 word나 상수인 num으로 채우고 나머지 노드들은 operator나 함수명인 word 또는 IF, WHILE의 jump문을 채워서 만든다. 또 AST를 만드는 과정에서 block의 depth를 계산하여 Symbol Table에 변수와 함수 symbol들을 삽입한다. AST를 만드는 과정에선 두 가지 error checking을 한다. 하나는 변수가 scoping rule을 만족하는 지 확인하는 것이고 다른 하나는 정의되지 않은 변수나 함수이름을 사용했는지 확인하는 것이다(<부록-2>참조). 위 Grammar에서는 함수의 선언을 정의하지 않기 때문에 함수는 무조건 소스코드에 상위에 정의된 함수만 사용할 수 있다. 소스코드에 하위의 작성된 함수는 아직 인식하지 못했기 때문에 not defined에러를 호출한다.

3.5 Code Generator

Code generator를 제작할 때 기존의 Instruction Set에서 조금 수정하여 진행하였다.

기존 Instruction Set	
LD Reg#1, addr(or num)	Load var (or num) into the Reg#1
ST Reg#1, addr	Store value of Reg#1 into var
ADD Reg#1, Reg#2, Reg#3	Reg#1 = Reg#2 + Reg#3
MUL Reg#1, Reg#2, Reg#3	Reg#1 = Reg#2 x Reg#3
LT Reg#1, Reg#2, Reg#3	1 if (Reg#2 < Reg#3), 0 otherwise , store into Reg#1
CALL function_name	Invoke function name
JUMPF Reg#1 label	Jump to label if Reg#1 contains 0
JUMPT Reg#1 label	Jump to label if Reg#1 contains Non-0
JUMP label	Jump to label without condition
수정한 Instruction Set	
LD Reg#1, addr(or num)	Load var (or num) into the Reg#1
ST Reg#1, addr	Store value of Reg#1 into var
ADD Reg#1, Reg#2, Reg#3	Reg#1 = Reg#2 + Reg#3
MUL Reg#1, Reg#2, Reg#3	Reg#1 = Reg#2 x Reg#3
LT Reg#1, Reg#2, Reg#3	1 if (Reg#2 < Reg#3), 0 otherwise , store into Reg#1
CALL function_name label	Invoke function name
JUMPF Reg#1 label	Jump to label if Reg#1 contains 0
JUMPT Reg#1 label	Jump to label if Reg#1 contains Non-0
JUMP label(Return_Adress)	Jump to label(Return_Adress) without condition
END	End of main function

Code generator는 진입점인 main함수를 찾는다. main함수가 존재하지 않으면 Error를 호출한다. main함수의 AST를 탐색하면서 Leaf노드에 도달하면 해당 데이터를 Register에 Load하는 것을 재귀적인 방식으로 target 코드를 저장한다. 탐색을 하는 과정은 Operator 부분과 IF, WHILE, function CALL과 같은 JUMP가 필요한 부분이 별개의 재귀 루틴으로 진행한다. AST는 각각의 sentence마다 Tree형식으로 자식을 가지고 있고 Next 노드로 다음 sentence의 노드를 가리키고 있기 때문에, Register 분배는 각각 Tree마다 Sethi-Ullman Numbering 방식을 사용하였다. 각각의 Operator 별로 처리한 방식은 다음과 같다.

Operator	처리 루틴	
+	Left 노드루틴 탐색 결과 target코드 Right 노드루틴 탐색 결과 target코드 ADD Reg#, Reg#, Reg#	
*	Left 노드루틴 탐색 결과 target코드 Right 노드루틴 탐색 결과 target코드 MUL Reg#, Reg#, Reg#	
=	Right 노드루틴 탐색 결과 target코드 ST Reg#, addr	
==	Left 노드루틴 탐색 결과 target코드 Right 노드루틴 탐색 결과 target코드 LT Reg#c, Reg#a, Reg#b LT Reg#d, Reg#b, Reg#a ADD Reg#c, Reg#c, Reg#d JUMPF Reg#c label	Equal을 LT를 활용해 a<b 의 값과 b>a의 값을 구하고, ADD로 두 값을 더했을 때 그 값이 0이면 점프를 하도록 구현한다.
>	Left 노드루틴 탐색 결과 target코드 Right 노드루틴 탐색 결과 target코드 LT Reg#c, Reg#a, Reg#b JUMPT Reg#c label	
IF	Left 노드루틴 탐색 결과 target코드 false 노드루틴 탐색 결과 target코드 JUMP label# labelT:: true 노드루틴 탐색 결과 target코드 label#::	조건이 True이면 True라벨로 JUMP해서 IF블록을 실행하고 그대로 이후 Instruction을 진행하며, False면 그대로 진행하는 것이 False 블록이므로 ELSE블록 실행 후 이후 Instruction의 label로 JUMP하여 진행한다.
WHILE	JUMP label# labelT:: true 노드루틴 탐색 결과 target코드 label#:: Left 노드루틴 탐색 결과 target코드	조건을 체크하는 부분을 아래에 두어 JUMP를 통해 조건을 체크하여 False면 이후 instruction을 진행하고 True면 labelT 즉 WHILE블록을 실행한 후 다시 조건을 체크하며 진행한다.
Function call	Left 노드루틴 AReg 저장 target코드 CALL function_name label	AReg라는 함수 인자 저장용 레지스터에 인자를 저장하고 함수를 CALL한다. (실제로 시스템차원에서는 AReg의 데이터를 메모리에 백업한 후 함수를 호출했다가 호출한 함수가 종료되면 백업데이터를 다시 AReg에 복구할 것이 가정하였다.)

main 함수에 대한 target코드 생성이 끝난 이후, main의 끝을 알리는 END instruction 코드를 생성하고, 나머지 function의 AST도 탐색하면서 target코드를 생성한다. 새로운 Register를 사용하면 번호를 1씩 올려서 사용했기 때문에 Register 번호 중 가장 큰 번호 +1 이 사용한 레지스터의 개수가 된다.(Reg, AReg 모두 집계)

4. Compiler 실행

compiler2018 testfile // 와 같이 Compiler를 구동하게 되면, testfile.code와 testfile.symbol이 Rusult 폴더 안에 생성된다. Compiler가 체킹해준 에러는 다음과 같다.

- Lexer에서, Input set에서 정의되지 않은 input이 들어오면 error를 반환한다.

AST는 Parsing Tree를 순회하면서 정의를 만날 때마다 symbol table에 해당 symbol과 소속되어 있는 block id를 삽입한다. 매번 symbol의 정의나 참조를 할 때 symbol table을 확인하여 참조할 때에는 symbol table에 없으면 error를 반환하고 반대로 정의를 할 때 symbool table에 있으면 재정의 오류로 error를 반환한다. 다음의 4가지 에러를 반환한다.

- not defined variable used : 정의되지 않은 변수 참조
- not defined function used : 정의되지 않은 함수 참조
- redefinition of function : 함수 재정의
- redefinition of varaible : 변수 재정의

- Parser에서 Parse Table을 참조하였는데 해당 index의 stack이 비어있는 경우, 진행할 수 있는 Transition이 존재하지 않는 것이므로 Input program이 Grammar를 만족하지 않는 것이므로 Syntax Error를 반환시킨다.

제출한 과제에는 testfile1과 testfile2 두 개의 예시 input 코드를 작성해보았다.

5. 부록

부록-1. Parse Table

	word	(	)	;	,	int	char
start						start -> prog prog_	start -> prog prog_
prog						prog -> vtype word (words) block	prog -> vtype word (words) block
prog_						prog_ -> vtype word (words) block prog_	prog_ -> vtype word (words) block prog_
decls	decls -> decls_					decls -> decls_	decls -> decls_
decls_	decls_ -> ε					decls_ -> decl decls_	decls_ -> decl decls_
decl						decl -> vtype words ;	decl -> vtype words ;
words	words -> word words_						
words_			words_ -> ε	words_ -> ε	words_ -> , word words_		
vtype						vtype -> int	vtype -> char
block							
slist	slist -> stat slist_						
slist_	slist_ -> stat slist_						
stat	stat -> word stat_						
stat_		stat_ -> (words) ;					
cond	cond -> expr cond_						
cond_							
expr	expr -> term expr_						
expr_				expr_ -> ε			
term	term -> fact term_						
term_				term_ -> ε			
fact	fact -> word						

(계속)

	{	}	IF	THEN	ELSE	WHILE	RETURN
start							
prog							
prog_							
decls			decls -> decls_			decls -> decls_	decls -> decls_
decls_			decls_ -> ε			decls_ -> ε	decls_ -> ε
decl							
words							
words_							
vtype							
block	block -> { decls slist }						
slist			slist -> stat slist_			slist -> stat slist_	slist -> stat slist_
slist_		slist_ -> ε	slist_ -> stat slist_			slist_ -> stat slist_	slist_ -> stat slist_
stat			stat -> IF cond THEN block ELSE block			stat -> WHILE cond block	stat -> RETURN expr ;
stat_							
cond							
cond_							
expr							
expr_	expr_ -> ε			expr_ -> ε			
term							
term_	term_ -> ε			term_ -> ε			
fact							

(계속)

	=	>	==	+	*	num	$
start							
prog							
prog_							prog_ -> ε
decls							
decls_							
decl							
words							
words_							
vtype							
block							
slist							
slist_							
stat							
stat_	stat_ -> = expr ;						
cond						cond -> expr cond_	
cond_		cond_ -> > expr	cond_ -> == expr				
expr						expr -> term expr_	
expr_		expr_ -> ε	expr_ -> ε	expr_ -> + term			
term						term -> fact term_	
term_		term_ -> ε	term_ -> ε	term_ -> ε	term_ -> * fact		
fact						fact -> num	

부록-2. Scoping rule

현재 scope	해당 block안에 선언된 변수 사용가능
상위 scope	상위 scope에 있는 변수는 최상위 scope인 함수 block의 변수만 접근가능
하위 scope	하위 scope에 있는 변수 접근 불가