**Compiler Project**

**1. Removing ambiguity from the given CFG.**

1.1 Operation Priority

05: EXPR → EXPR addsub EXPR | EXPR multdiv EXPR

06: EXPR → lparen EXPR rparen | id | num

주어진 CFG에서 05번의 rule을 보면, addsub과 multdiv를 같이 parsing 한다는 것을 알 수 있다. 이 경우 곱셈/나눗셈과 덧셈/뺄셈이 합쳐진 statement에서 어느 rule을 먼저 선택하느냐에 따라 두 개 이상의 parse tree가 나올 수 있다. 그러나 실제로 multdiv가 addsub보다 우선순위를 가져야 하므로 parse tree의 아래쪽에 있어야 한다. 따라서 addsub과 multdiv의 rule을 분리해야 한다.

따라서 같은 레벨에 있는 EXPR을 TERM과 EXPR\_TAIL로 분리하여 EXPR\_TAIL에서 addsub을 먼저 분기하도록 하였다. 따라서 EXPR\_TAIL은 TERM과 TERM의 addsub operation을 담당하게 된다. 이후 TERM은 다시 FACTOR와 TERM\_TAIL로 나뉘어 TERM\_TAIL에서 FACTOR과 FACTOR의 multdiv operation을 처리하도록 하였다. 아래와 같은 rule을 사용하면 주어진 statement에서 addsub operation이 parse tree의 위쪽에서 분리되고, 그 아래에서 multdiv operation이 분리되도록 구성할 수 있다.

05: EXPR → TERM EXPR\_TAIL

06: EXPR\_TAIL → addsub TERM EXPR\_TAIL | ϵ

07: TERM → FACTOR TERM\_TAIL

08: TERM\_TAIL → multdiv FACTOR TERM\_TAIL | ϵ

09: FACTOR → lparen EXPR rparen | id | num

1.2 Dangling Else Problem

12: STMT → if lparen COND rparen lbrace BLOCK rbrace ELSE

15: ELSE → else lbrace BLOCK rbrace | ϵ

주어진 CFG의 12번 rule을 보면 if ELSE를 한 번에 처리하는 것을 알 수 있다. 그러나 이 경우, 여러 if else 문이 중첩되어 있을 때 어느 if와 else를 연결해야 하는지 알기 어려워 ambiguity가 발생한다. 예를 들어 if – if – else 의 경우, if-else 문 안에 else가 없는 if 문이 들어가 있는 것으로 생각할 수 있지만, else가 없는 if 문 뒤에 if-else 문이 따라오는 것으로 해석할 수도 있다. 따라서 else를 가장 가까운 if 문과 매치하여 짝이 존재하는 것과 아닌 것을 구분하여 나타내야 한다.

주어진 CFG의 STMT를 IF와 IFELSE로 분리하여 각각 else가 없는 것과 있는 것으로 분리하였다. 그리고 IF는 else가 없는 if문으로, IFELSE는 if와 else의 짝이 존재하도록 각각 parsing 하였다. 아래와 같은 rule을 적용하면, if문 내부 block에 if-else 문이 포함되어 있는 parse tree를 형성하여 ambiguity를 없앨 수 있다.

15: STMT → IF | IFELSE

17: IF → if lparen COND rparen lbrace BLOCK rbrace

18: IFELSE → if lparen COND rparen lbrace BLOCK rbrace else lbrace BLOCK rbrace

1.3 Multiple Comp ambiguity

14: COND → COND comp COND | boolstr

주어진 CFG의 14번 rule에서 COND는 여러 COND 혹은 boolstr로 표현되어 comp operator로 비교되어 나타나진다. 그런데 만약 3 개 이상의 boolstr이 존재하여 비교된다면, 이들 중 어느 것이 먼저 해석되어야 할 지에 대한 priority problem이 발생한다. 예를 들어 a == b && a == c와 같은 statement가 COND로 해석되어야 할 때 ==연산자가 && 연산자보다 먼저 해석되어야 한다. 그러나 주어진 rule로는 무엇을 먼저 분리할 지 알 수 없기 때문에 두 개 이상의 parse tree가 형성될 수 있다. 따라서 이 ambiguity를 제거해야 한다.

이를 제거하기 위해 COND를 SIMPLECOND와 COND\_TAIL로 분리하여 가장 앞의 boolstr부터 차례로 해석하도록 하였다. SIMPLECOND는 하나의 boolstr 혹은 paren으로 감싸진 COND를 분리해내고, 그 것과 연결된 comp와 나머지 COND가 COND\_TAIL에서 해석된다. 따라서 이 rule을 사용하면 COND를 가장 앞에 있는 boolstr부터 하나씩 분리하여 해석할 수 있다. 이는 실제 연산자 우선순위와는 다르지만 해당 CFG는 simplified C이므로 연산자 우선순위까지 구현할 필요는 없다고 생각하여 이처럼 가장 간단한 방식으로 ambiguity를 제거하였다.

19: COND → SIMPLECOND COND\_TAIL

20: SIMPLECOND → boolstr | lparen COND rparen

21: COND\_TAIL → comp SIMPLECOND COND\_TAIL | ϵ

1.4 Revised CFG

따라서 세 가지의 ambiguity를 모두 수정한 최종 CFG는 아래와 같다.

01: CODE → VDECL CODE | FDECL CODE | ϵ

02: VDECL → vtype id semi | vtype ASSIGN semi

03: ASSIGN → id assign RHS

04: RHS → EXPR | literal | character | boolstr

05: EXPR → TERM EXPR\_TAIL

06: EXPR\_TAIL → addsub TERM EXPR\_TAIL | ϵ

07: TERM → FACTOR TERM\_TAIL

08: TERM\_TAIL → multdiv FACTOR TERM\_TAIL | ϵ

09: FACTOR → lparen EXPR rparen | id | num

10: FDECL → vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace

11: ARG → vtype id MOREARGS | ϵ

12: MOREARGS → comma vtype id MOREARGS | ϵ

13: BLOCK → STMT BLOCK | ϵ

14: STMT → VDECL | ASSIGN semi

15: STMT → IF | IFELSE

16: STMT → while lparen COND rparen lbrace BLOCK rbrace

17: IF → if lparen COND rparen lbrace BLOCK rbrace

18: IFELSE → if lparen COND rparen lbrace BLOCK rbrace else lbrace BLOCK rbrace

19: COND → SIMPLECOND COND\_TAIL

20: SIMPLECOND → boolstr | lparen COND rparen

21: COND\_TAIL → comp SIMPLECOND COND\_TAIL | ϵ

22: RETURN → return RHS semi

**2. SLR Parsing Table**

상기의 non-ambiguous grammar를 통해 SLR parsing table을 만들 수 있다. 과제 description에 명시된 웹사이트를 이용하기 위해서는 ϵ와 화살표를 ''와 ->로 바꾸고, | 를 제거하는 작업이 필요하다. 그리고 New start symbol E’ -> CODE를 추가해주어야 한다. 이 과정을 거친 CFG는 아래와 같다.

E’ -> CODE

CODE -> VDECL CODE

CODE -> FDECL CODE

CODE -> ''

VDECL -> vtype id semi

VDECL -> vtype ASSIGN semi

ASSIGN -> id assign RHS

RHS -> EXPR

RHS -> literal

RHS -> character

RHS -> boolstr

EXPR -> TERM EXPR\_TAIL

EXPR\_TAIL -> addsub TERM EXPR\_TAIL

EXPR\_TAIL -> ''

TERM -> FACTOR TERM\_TAIL

TERM\_TAIL -> multdiv FACTOR TERM\_TAIL

TERM\_TAIL -> ''

FACTOR -> lparen EXPR rparen

FACTOR -> id

FACTOR -> num

FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace

ARG -> vtype id MOREARGS

ARG -> ''

MOREARGS -> comma vtype id MOREARGS

MOREARGS -> ''

BLOCK -> STMT BLOCK

BLOCK -> ''

STMT -> VDECL

STMT -> ASSIGN semi

STMT -> IF

STMT -> IFELSE

STMT -> while lparen COND rparen lbrace BLOCK rbrace

IF -> if lparen COND rparen lbrace BLOCK rbrace

IFELSE -> if lparen COND rparen lbrace BLOCK rbrace else lbrace BLOCK rbrace

COND -> SIMPLECOND COND\_TAIL

SIMPLECOND -> boolstr

SIMPLECOND -> lparen COND rparen

COND\_TAIL -> comp SIMPLECOND COND\_TAIL

COND\_TAIL -> ''

RETURN -> return RHS semi

이를 해당 웹사이트에 입력하면 SLR parsing table을 구할 수 있다.

먼저 First & Follow set을 명시한 table은 아래와 같다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

전체 ACTION & GOTO table도 아래와 같다.

텍스트, 번호, 평행, 라인이(가) 표시된 사진

자동 생성된 설명라인, 텍스트, 평행, 번호이(가) 표시된 사진

자동 생성된 설명

전체 table은 첨부한 excel 파일에서 확인할 수 있다.

텍스트, 스크린샷, 번호, 라인이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 도표, 평행이(가) 표시된 사진

자동 생성된 설명

제대로 작동하는지 확인하기 위해 예문을 입력하여 parse tree를 확인해보았다. 올바른 결과를 넣을 경우 잘 작동하는 것을 확인할 수 있다.

**3. Implementation of SLR parser**

3.1 Code description

Non-ambiguous 한 CFG로 Action & Goto parsing table을 만들었으므로 이제 이를 사용하여 SLR parser를 구현할 수 있다.

가장 먼저 각 action / goto의 state가 가리키는 상태와 변환 규칙을 위해 CFG를 string과 string vector의 pair 형태로 productions vector에 저장하였다. 그리고 input token에 대한 행동을 지시하기 위해 각 state와 token을 index로 하여 next action을 결정하도록 ACTION 과 GOTO table을 만들었다. (85개의 state에 대해 각 state 마다 존재하는 input token과 terminal을 일일이 기입하다 보니 실수가 많았고 대부분의 버그가 비롯된 곳이 바로 해당 테이블이었다)

그리고 각 action에 대해 다음 행동을 정의하기 위해 enum class와 class Action을 만들어 shift / reduce에 대해 각각 next state과 production을 int 값으로 가지도록 하였다.

Parsing을 담당하는 것은 Parser class에서 각 table을 형성하고(initializeTable), input.txt에서 token sequence를 받아 parse function에서 action / goto table의 state를 보고 stack에 push / pop 하는 방식으로 파싱을 진행하도록 하였다.

이 때 출력 값은 output.txt에 parse tree 형태로 나타나야 하므로 이를 형성하는 class treeNode를 만들고 stack에 각 트리 노드를 push 하는 방식으로 진행하였다.

Parser class에서 실제로 parsing을 담당하는 것은 bool parse() function으로 state과 input token sequence로 Action & Goto table에 저장된 값을 보고 ActionType에 정의된 SHIFT, REDUCE, ACCEPT에 각각에 대해 알맞은 행동을 취한다.

SHIFT라면 해당 state와 token을 push 하고, REDUCE라면 stack에서 CFG에 정의되어 있는 production만큼을 pop하고, 해당 non terminal들을 reduce 한 것을 다시 stack에 push한다. 이를 반복하여 ACCEPT가 되면 parsing 과정은 종료된다. 이 때 각각 상황에서 만약 table에 정의되지 않은 값이 입력되면 error로 판정하여 오류메시지를 output.txt에 출력한다.

(해당 부분에서 epsilon을 reduce하여 어떤 non-terminal이 나올 때 stack에서는 epsilon을 pop, 즉 아무것도 꺼내지 않고 non-terminal을 push 해야 하는데 개수로 확인하여 pop 하다 보니 무조건 하나는 pop 하게 되어 epsilon production에 대해 오류가 발생하였다. 따라서 이를 if문으로 분리하여 CFG의 production rule이 epsilon = “” 에서 어떤 non-terminal로 reduce 된다면 stack의 pop\_count를 0으로 하여 해당 오류를 해결하였다)

본 프로그램은 input.txt에 정의된 token sequence를 입력으로 받아 parsing에 성공하면 parse tree, 실패한다면 error message를 output.txt에 출력한다. 해당 프로그래밍은 linux 환경에서 진행하였으므로 첨부된 Makefile을 통해 소스코드와 원하는 input.txt 파일이 있는 곳에서 컴파일하면 output.txt인 결과를 얻을 수 있다.

이를 위하여 readTokensFromFile과 writeOutputToFile 함수를 각각 만들어 입력 / 출력을 담당하였다.

Main 함수에서는 프로그램의 이름, input.txt, output.txt를 argv로 받아 처리하고 parser를 생성하여 input.txt에 저장된 token sequence에 대해 parsing을 진행한다. 이 때 input.txt는 과제에서 예시된 것처럼 띄어쓰기를 사용해 입력되어야 한다. (eg. vtype id semi)

3.2 Compile

해당 과제는 리눅스 시스템에서 작동하여야 하므로 Makefile을 사용하여 compile 하였다. 따라서 주어진 소스코드와 Makefile, 원하는 token sequence를 입력한 input.txt가 있는 디렉토리에서 make

make run

make clean

을 순서대로 타이핑하면 결과가 저장된 output.txt 파일이 생성된다. 또한 디버깅을 위해 각 production step에서 stack content와 next token 등이 console창에 출력된다.

3.3 Example cases

Input.txt에 아래와 같은 token sequence를 입력하였다.

vtype id lparen vtype id comma vtype id comma vtype id comma vtype id rparen lbrace vtype id semi id assign boolstr semi if lparen boolstr rparen lbrace rbrace if lparen boolstr rparen lbrace rbrace else lbrace vtype id semi rbrace while lparen boolstr comp lparen boolstr comp boolstr rparen rparen lbrace id assign literal semi rbrace return literal semi rbrace

해당 input의 결과로 output.txt에 생성된 parse tree는 아래와 같다.

Parsing successful!

CODE

FDECL

vtype

id

lparen

ARG

vtype

id

MOREARGS

comma

vtype

id

MOREARGS

comma

vtype

id

MOREARGS

comma

vtype

id

MOREARGS

rparen

lbrace

BLOCK

STMT

VDECL

vtype

id

semi

BLOCK

STMT

ASSIGN

id

assign

RHS

boolstr

semi

BLOCK

STMT

IF

if

lparen

COND

SIMPLECOND

boolstr

COND\_TAIL

rparen

lbrace

BLOCK

rbrace

BLOCK

STMT

IFELSE

if

lparen

COND

SIMPLECOND

boolstr

COND\_TAIL

rparen

lbrace

BLOCK

rbrace

else

lbrace

BLOCK

STMT

VDECL

vtype

id

semi

BLOCK

rbrace

BLOCK

STMT

while

lparen

COND

SIMPLECOND

boolstr

COND\_TAIL

comp

SIMPLECOND

lparen

COND

SIMPLECOND

boolstr

COND\_TAIL

comp

SIMPLECOND

boolstr

COND\_TAIL

rparen

COND\_TAIL

rparen

lbrace

BLOCK

STMT

ASSIGN

id

assign

RHS

literal

semi

BLOCK

rbrace

BLOCK

RETURN

return

RHS

literal

semi

rbrace

CODE

같은 input을 과제 파일에 명시된 웹사이트에 입력하여도 동일한 parse tree가 나오는 것을 확인할 수 있다.

스크린샷, 도표, 라인, 평면도이(가) 표시된 사진

자동 생성된 설명

또한, reject 되는 경우를 확인해보기 위해 과제에 예시된 오류 경우를 입력해보았다.

vtype id semi vtype id lparen rparen lbrace if lparen boolstr comp boolstr rparen lbrace rbrace

해당 input에 대한 결과는 아래와 같다.

Error: Unexpected token '$' at position 16

Error: No ACTION entry for state 81 and token '$'

해당 input은 FDECL에서 lbrace 다음 RETURN을 입력하지 않고 바로 rbrace로 sequence가 종료되었다. 따라서 input을 다 읽었음에도 알맞은 reduce 규칙을 찾지 못하고 sequence의 끝에 다다라 오류가 출력된 것을 확인할 수 있다.