Document

20186889 권용한

이전과 같이 c언어를 사용하여 과제를 수행하였으며, 소스 코드 및 실행 파일은 이전의 실행 결과도 포함하고 있습니다. 때문에 자바 소스코드를 바로 토큰으로 분류하면서 syntax적으로 문제가 있는지 없는지 판단이 바로 가능합니다.

1. CFG

01: CODE → VDECL CODE | FDECL CODE | CDECL CODE | ε

02: VDECL → vtype id semi | vtype ASSIGN semi

03: ASSIGN → id assign RHS

04: RHS → EXPR | literal | character | boolstr

05: EXPR → EXPR addsub EXPR | EXPR multdiv EXPR

06: EXPR → lparen EXPR rparen | id | num

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

08: ARG → vtype id MOREARGS | ε

09: MOREARGS → comma vtype id MOREARGS | ε

10: BLOCK → STMT BLOCK | ε

11: STMT → VDECL | ASSIGN semi

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

13: STMT → while lparen COND rparen lbrace BLOCK rbrace

14: COND → COND comp COND | boolstr

15: ELSE → else lbrace BLOCK rbrace | ε

16: RETURN → return RHS semi

17: CDECL → class id lbrace ODECL rbrace

18: ODECL → VDECL ODECL | FDECL ODECL | εRETURN -> return RHS semi

여기서 애매한 부분은 expr과 cond로, 두 부분 모두 slr을 만든다면 shift할지 reduce할지 결정할 수 없는 경우가 발생하게 되며, 때문에 이를 구분해 주었습니다.

05: EXPRA → EXPRA addsub EXPRB | EXPRB

EXPRB → EXPRB multdiv EXPRC | EXPRC

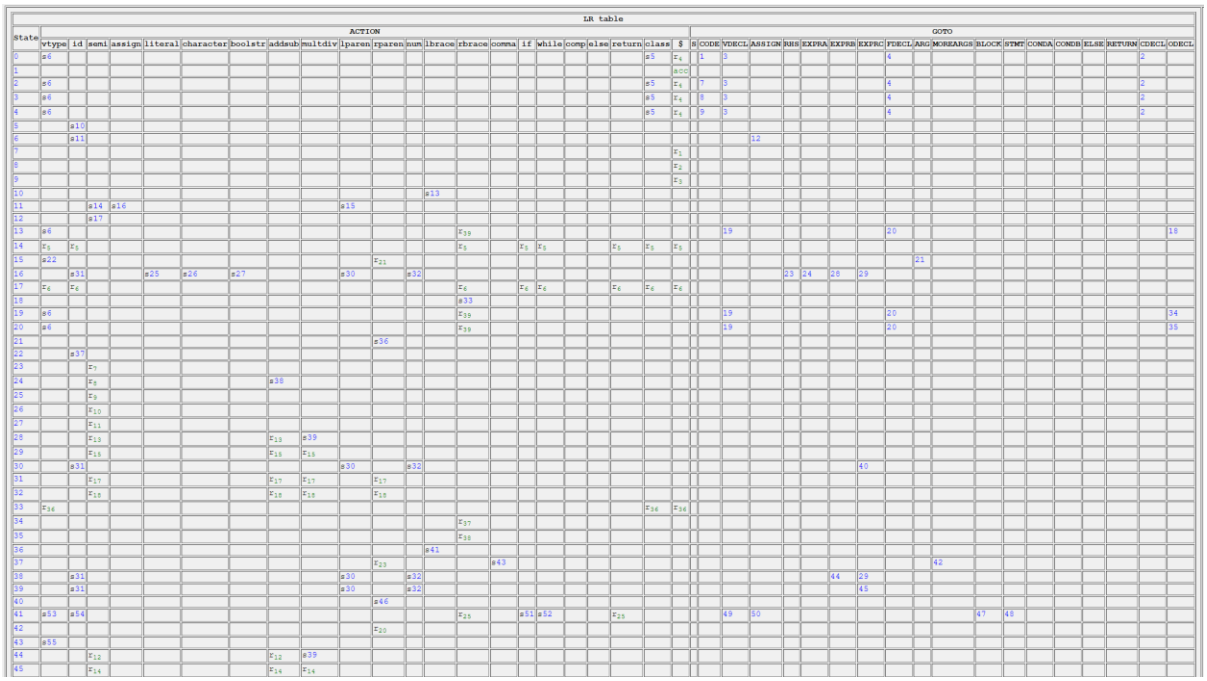
06: EXPRC → lparen EXPRC rparen | id | num

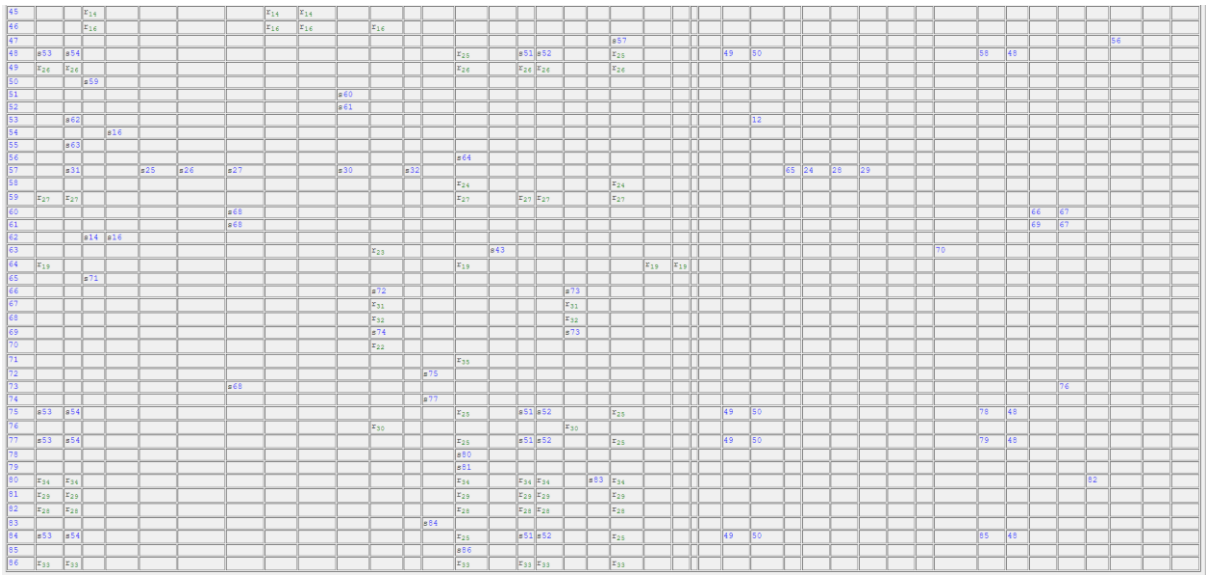
14: CONDA → CONDA comp CONDB | CONDB

CONDB → boolstr

이렇게 구분해줘서 shift하는 경우와 reduce하는 경우를 확실히 구분해 주었고, 이를 통해서 주어진 cfg에서 안정적으로 slr table을 구성 할 수 있었습니다.

2. SLR TABLE





위 사진은 웹사이트를 통해 구한 slr table로, 해당 cfg에서 expr과 cond파트를 더 세분화하고, 시작 노드를 추가하여 구성하였습니다.

FIRST / FOLLOW table		
Nonterminal	**FIRST**	**FOLLOW**
S	{'',class,vtype}	{$}
CODE	{'',class,vtype}	{$}
VDECL	{vtype}	{$,class,vtype,return,id,if,while,rbrace}
ASSIGN	{id}	{semi}
RHS	{literal,character,boolstr,lparen,id,num}	{semi}
EXPRA	{lparen,id,num}	{semi,addsub}
EXPRB	{lparen,id,num}	{semi,addsub,multdiv}
EXPRC	{lparen,id,num}	{semi,addsub,multdiv,rparen}
FDECL	{vtype}	{$,class,vtype,rbrace}
ARG	{vtype,''}	{rparen}
MOREARGS	{comma,''}	{rparen}
BLOCK	{'',vtype,id,if,while}	{return,rbrace}
STMT	{vtype,id,if,while}	{return,vtype,id,if,while,rbrace}
CONDA	{boolstr}	{rparen,comp}
CONDB	{boolstr}	{rparen,comp}
ELSE	{else,''}	{return,vtype,id,if,while,rbrace}
RETURN	{return}	{rbrace}
CDECL	{class}	{$,class,vtype}
ODECL	{vtype,''}	{rbrace}

SLR table을 구성하기 위해 필요한 FIRST/FOLLOW table입니다.

마지막으로 소스코드의 경우 bottom up parsing의 경우 결국 stack을 사용할 수 밖에 없으며, 프로그램으로 사용하기 위해서 해당 테이블의 정보를 정리해 필요한 정보를 직접 찾을 수 밖에 없다고 생각했습니다. 때문에 action과 go_to라는 이름의 함수를 만들어 action과 goto table의 정보가 담겨있습니다.

또한 get_terminal 함수를 통해 이전 lexical analyzer에서 구한 결과물을 해당 cfg에 맞도록 고쳐 정보를 받아오고, reduct 함수에서는 만일 reduction이 발생한다면 현재 어떤 상황에서 어떤 상황으로 변화하는지(goto 함수 이용), 또한 결과 nonterminal이 무엇인지 판단해 다음 state를 구하게 됩니다.

Stack의 경우엔 교재에서는 int형으로 사용하였지만, 결국 제대로 사용하기 위해선 state정보 뿐만이 아니라 이전의 terminal이나 nonterminal 정보도 함께 필요하다고 생각되어 int형과 char*으로 이루어진 element형을 구성하여 stack의 기본 형으로 사용하였습니다. 하지만 제작해본 결과 길이정보와 변화하는 nonterminal의 정보를 얻을 수 있다면 int형으로 사용하여도 같은 결과를 얻을 수 있다고 생각했습니다.

Action 함수를 통해 s(shift)인지 r(reduction)인지 판단할 수 있으며, shift인 경우 stack에 데이터를 추가하고 다음 terminal을 받아와 다시 action을 진행합니다.

Reduction인 경우엔 reduct함수를 실행하고, reduct 함수에서는 현재 state와 terminal 또는 nonterminal 정보를 통해 다음 state와 nonterminal 정보를 얻어옵니다. 얻어온 정보를 바탕으로 state는 변화시키지만, 다음 terminal의 경우 shift시키지 않았기 때문에 다음 terminal을 받아오지 않고, 이전에 받아온(직전에 다음 터미널이라고 판단한) 터미널과 현재 state를 통해 다시 action 함수를 진행하게 됩니다.

이렇게 $가 등장할 때까지 반복하게 되며, $가 등장한다면 action 함수에서 현재 state가 1이면서 다음 terminal이 $라면 accept 하게 됩니다.

만약 진행 과정 중, table에 존재하지 않는 경우에 대한 action을 요구하거나 goto 정보를 요구하게 된다면, 이는 문법적으로 틀린 문장이기 때문에, 해당 경우엔 바로 reject를 출력하고 프로그램을 종료합니다.