**Design and Development of Compiler**

**for C- Language**

Phase 2: Design and Implementation of LALR Parser

Results Report of Phase 2

과목명: [CSE4120] 기초 컴파일러 구성

담당교수: 서강대학교 컴퓨터공학과 정 성 원

개발자: 20141522 박상욱

개발기간: 2017. 4.25 - 2017. 5. 12

각 단계별 결과 보 고 서

프로젝트 제목: Design and Development of Compiler for C-Language:  
Phase 1: Design and Implementation of LALR Parser

**제출일: 2017.** **5.** **12.**

**개발자: 20141522박 상 욱**

**I. 개발 목표**

단계별 제안서 I. 개발 목표 참조

II. 개발 범위 및 내용

단계별 제안서 II. 개발 범위 및 내용 참조

**III. 추진 일정 및 개발 방법**

단계별 제안서 III. 추진 일정 및 개발 방법 참조

**Ⅳ. 연구 결과**

1. **합성 내용**

본 프로젝트는 앞선 Phase 1에서 개발한 Scanner를 사용한다. Phase 1에서 개발한 Scanner를 이용해 입력 스트림을 토큰 스트림으로 바꾸어준다. cminus.y 파일을 통해서 BNF 문법을 통해서 Grammar Rule을 명시한다. Flex 툴과 Bison 툴을 이용해 cminus.l 파일과 cminus.y 파일로부터 lex.yy.c, cminus.tab.h, cminus.tab.c를 생성한다. 이를 최종적으로 나머지 파일들과 합쳐 Compile하여 LALR Parser를 완성한다. LALR Parser로 생성되는Syntax Tree의 다양한 노드들은 globals.h에서 Enum Type을 통해서 관리해준다. 노드의 종류에 따라 저장하고 있는 정보가 다를 수 있도록 설정한다. 테스트 C- Program을 이용해 Syntax Tree를 구축해보기 위해서는, make를 통해 LALR Parser를 생성하고, *./20141522 [C- filename]* 을 통해서 실행한다. 프로그램의 결과는 util.c의 PrintTree함수를 통해서 출력되기에, PrintTree함수를 수정하여 다양한 Expression과 Statement에 대해 출력양식을 달리 해주었다.

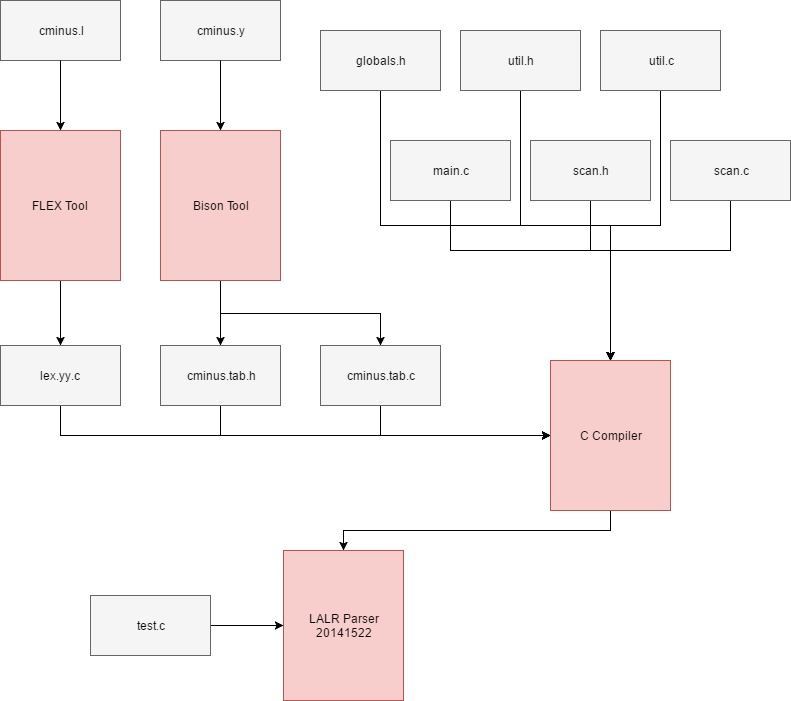


Figure . 프로그램 흐름도

1. **분석 내용**

**C- Grammar Rule**

본 프로젝트에서는 C- Language에 대한 Syntax Tree를 생성하기 위해서 아래와 같은 Grammar Rule을 사용하였다. 아래의 Grammar Rule은 BNF 문법을 이용해 작성하며, 대문자는 token(terminal)의 이름을, 소문자는 State(nonterminal)의 이름을 말한다.

**program : dclr\_list**

**dclr\_list : dclr\_list dclr | dclr**

**dclr : var\_dclr | func\_dclr**

**var\_dclr : type\_spcf ID SEMI | type\_spcf ID LSBRACE NUM RSBRACE SEMI**

**type\_spcf : INT | VOID**

**func\_dclr : type\_spcf ID LPAREN params RPAREN compstmt**

**params : para\_list | VOID**

**para\_list : para\_list COMMA param | param**

**param : type\_spcf ID | type\_spcf ID LSBRACE RSBRACE**

**compstmt : LQBRACE local\_dclr stmt\_list RQBRACE**

**local\_dclr : local\_dclr var\_dclr | /\* empty \*/**

**stmt\_list : stmt\_list stmt | /\* empty \*/**

**stmt : expstmt | compstmt | selstmt | iterstmt | retstmt**

**expstmt : exp SEMI | SEMI**

**selstmt : IF LPAREN exp RPAREN stmt**

**| IF LPAREN exp RPAREN stmt ELSE stmt**

**iterstmt : WHILE LPAREN exp RPAREN stmt**

**retstmt : RETURN SEMI | RETURN exp SEMI**

**exp : var ASSIGN exp | smpl\_exp**

**var : ID | ID LSBRACE exp RSBRACE**

**smpl\_exp : addexp relop addexp | addexp**

**relop : LE | LT | GT | GE | SAME | DIFF**

**addexp : addexp addop term | term**

**addop : PLUS | MINUS**

**term : term mulop factor | factor**

**mulop : TIMES | OVER**

**factor : LPAREN exp RPAREN | var | call | NUM**

**call : ID LPAREN args RPAREN**

**args : arg\_list | /\* empty \*/**

**arg\_list : arg\_list COMMA exp | exp**

**Semantic Action**

본 프로젝트에서 사용할 BNF 문법은 Bison 툴을 이용하여 C- Language의 Syntax Tree 생성에 사용한다. BNF 문법은 Bison Grammar File에 작성하면 되는데, Bison Grammar File은 .y 확장자를 가지며 Prologue, Bison declarations, Grammar rules, Epilogue로 이루어져 있다. 대략적인 파일의 구성은 아래를 참조하자.

%{

*Prologue*

%}

*Bison declarations*

%%

*Grammar rules*

%%

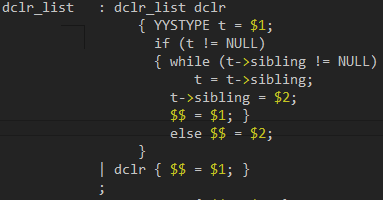
*Epilogue*

Grammar Rule에서 C언어를 이용해 Semantic Action을 정의할 수 있다. 이러한 Semantic Action은 해당 문법의 Token Stream이 나타나면, 정해진 코드를 수행한다. Bison File에서의 Semantic Action은 다음과 같이 사용한다.

exp:

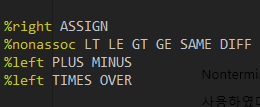
…

| exp '+' exp { $$ = $1 + $3; }

본 프로젝트에서는 Semantic Action을 Syntax Tree의 노드를 생성하고 필요한 정보를 저장하는데 사용하였다. 위의 Grammar Rule을 이용해 C- Language의 Syntax Tree를 생성한다. C- Language에서 Syntax Tree를 생성하기 위해서 사용한 Semantic Action을 설명하자면 Terminal의 경우, ID, NUM과 같은 Token이 입력되는 경우에 노드를 생성하고 Variable의 이름, Type, Value와 같은 정보를 저장해준다. ID혹은 NUM의 경우에는 yylex()를 이용해 LEX 파일에서 Token을 읽을 때, string을 savedName 혹은 savedNum를 이용해 Bison 파일로 가져와 현재 노드에 그 값을 저장하도록 한다. Nonterminal Token이 입력되는 경우 현재 노드의 Child로 저장해준다. 그리고 Compound Statment와 같이 다수의 Nonterminal Token이 입력되어 여러 개의 Child가 필요한 경우에는 새로운 노드를 생성하고 그 노드의 Child로 저장한다. 또한 Nonterminal Token이 입력될 때 Left Recursion을 해결하기 위해서 아래와 같은 코드를 사용하였다.

C- Grammar Rule에서는 Nonterminal Token이 3개 이상 나타나는 경우가 없었고, 따라서 MAXCHILDREN의 수는 3으로 설정한다.

**Precedence and Associativity**

Parsing을 진행하며, Precedence와 Association을 제대로 설정해주지 않는 경우, Ambiguous Syntax Tree가 생성될 수 있다. 이러한 문제를 해결해 주기 위해서, Ambiguous한 Grammar에 대하여 Precedence와 Associativity를 정의해주도록 한다. 이 문제를 해결하기 위해서는 Bison Declaration 부분에서 Bison 문법을 통해서 Operator간의 Precedence 및 Associativity를 설정해 준다.

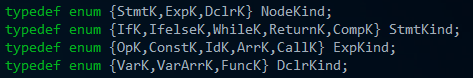
**Conflict**

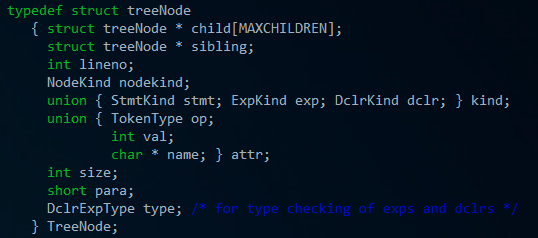
LALR(1) 문법에서 Ambiguous 한 부분이 있다면 Conflict가 발생할 수 있다. 발생 가능한 Conflict는 Shift/Reduce Conflict와 Reduce/Reduce Conflict가 있다. 이러한 Conflict가 발생하더라도 Bison은 자동적으로 Rule을 선택해 Conflict 발생을 해결한다. Shift/Reduce Conflict의 경우 Bison은 기본적으로 Shift를 선택해 Conflict를 해결하며, Reduce/Reduce Conflict의 경우에 처음으로 나타나는 Rule을 선택해 Conflict를 해결한다. 다만 Reduce/Reduce Conflict의 경우는 해결이 되지 않는 경우도 있어, Conflict가 발생하는 경우에 대해 자세히 살펴보고 수정해야 할 필요가 있을 경우 Grammar를 변경해주는 것이 좋다.

1. **제작 내용**
   1. **main.c**

C- Language Parser를 제작하기 위해서 NO\_ANALYZE 값을 TRUE로 설정해준다. 앞서 Phase 1에서 Scanner를 제작하며 그 결과를 보기 위해 출력했던 부분을 삭제해준다. NO\_PARSE를 FALSE로 변경해주며 TraceScan을 FALSE로 설정한다. 그리고 Parsing 결과를 보기 위해서 TraceParse를 True로 변경해준다.

* 1. **globals.h**

C- 문법에 해당하는 Token은 Bison File에서 정의해주기 때문에, Phase 1에서 사용했던 globals.h에서의 Token에 대한 Enum 데이터는 삭제한다. Syntax Tree의 노드를 생성하기 위해서 노드에 필요한 정보를 추가해준다. 새로운 타입의 노드를 위해서 enum을 통해서 그 종류를 정의해준다.

Syntax Tree를 생성하는 데에 있어 노드에는 다양한 정보가 필요했다. 따라서 그 정보들을 저장해주기 위해 노드를 아래와 같이 정의하였다.

위와 같이 노드의 종류에 Declare node를 추가해 주고, Array 노드에서 필요한 Size 정보와 parameter인지 확인하기 위한 para flag를 선언해주었다.

* 1. **util.c**

Syntax Tree의 노드를 생성하고, 완성된 Tree 전체를 출력해줄 수 있는 함수를 작성하였다. DclrKind의 노드를 생성하기 위해 newDclrNode 함수를 제작하였다. printTree함수를 수정하여, 노드의 종류에 따라 출력양식이 다르게 해주었다.

* 1. **cminus.l**

Bison 파일cminus.y에서 Token을 정의하고, 특히 ID와 NUM에 대해서 받은 String을 Tree의 노드에 저장하기 위해서 다음과 같이 처리해주었다.

받아진 savedNum 또는 savedName은 노드에서 attr.val 혹은 attr.name 에 저장하였다. 또한 Comment의 경우에, Grammar Rule에 Comment Token이 존재하지 않아 에러가 발생하였다. 이에 Comment가 들어오는 경우, Token을 Return하지 않고, 단순히 Comment Error인 경우에만 Error 출력을 해주도록 변경하였다.

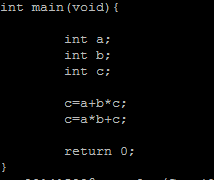
* 1. **cminus.y**

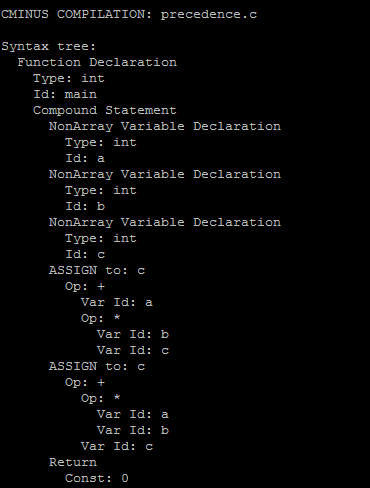
Bison 파일로서 앞서 설명한 C- Grammar를 BNF문법으로 작성하였다. Phase 1에서 globals.h에 작성한 Token들을 이 파일에서 Bison Declaration부분에서 정의해주었다. Bison 툴은 이 파일에 작성한 문법을 기반으로 cminus.tab.h와 cminus.tab.c 파일을 생성한다. Semantic Action을 통해서 Syntax Tree를 생성했고, lex파일에서 받은 savedNum, savedName을 사용해 노드에 필요한 정보를 저장해주었다.

1. **시험**

본 프로젝트, LALR Parser를 시험하기 위해서 C- Language를 따르는 Test File을 제작하였으며, 그 File은 아래와 같다.

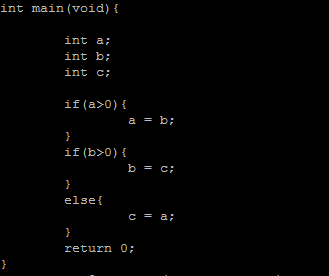
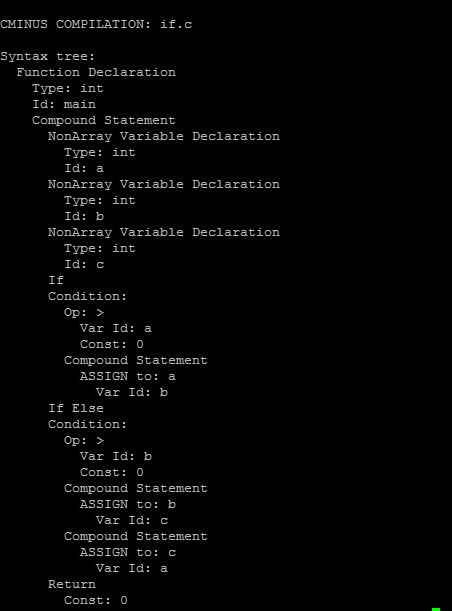
* 1. precedence.c

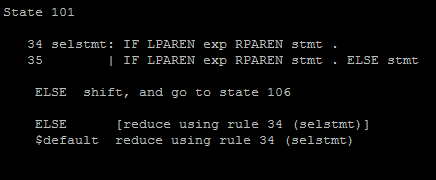
연산자 간의 precedence를 살펴보기 위한 코드이다. 앞서 Bison Declaration에서 Precedence를 정의해준 것처럼 \*의 precedence가 +보다 높은 것을 확인 할 수 있다.



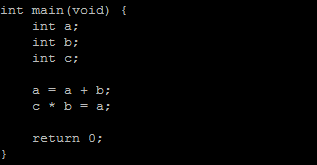
* 1. if.c

if구문과 if-else 구문에서 shift-reduce conflict가 발생하는지 테스트 하기 위한 코드이다. 앞서 말했듯이, Bison 툴은 Shif-Reduce Conflict가 발생하는 경우 기본적으로 Shift를 선택하게 된다. 따라서 if, if-else에서도 Ambiguous하지 않고 제대로 Parsing이 되는 것을 확인 할 수 있다. 별다른 수정 없이 Conflict 해결이 가능하다.[[1]](#footnote-1)



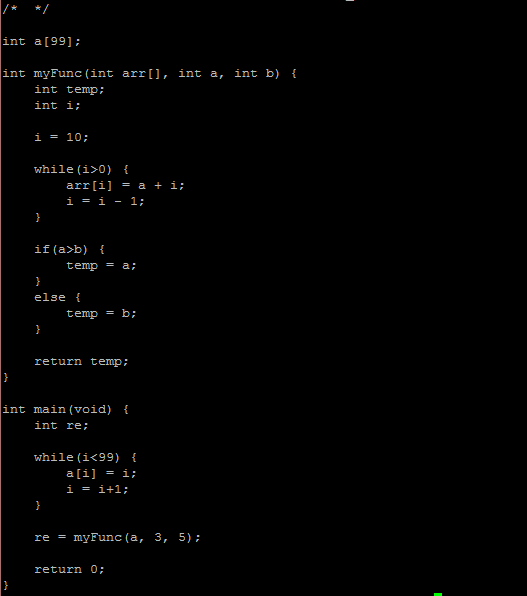
if, if-else 문법에서 Shift-Reduce Conflict가 실제로 발생하는지 알아보기 위해 cminus.output 파일을 열어보면 다음과 같이 확인할 수 있다. State 101에서 Conflict가 발생하고, State 101에서 Shift를 우선 선택하는 것을 확인 할 수 있다.

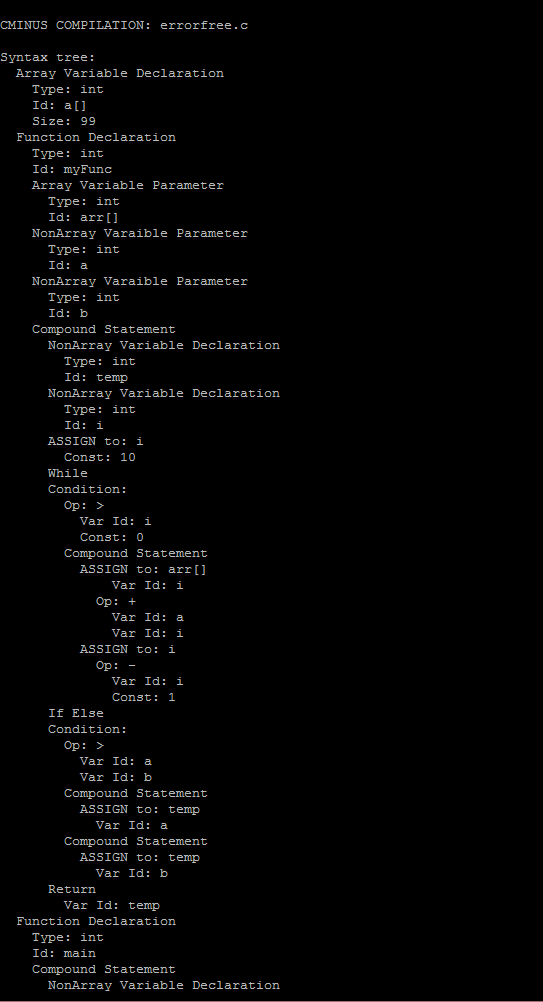
* 1. error.c

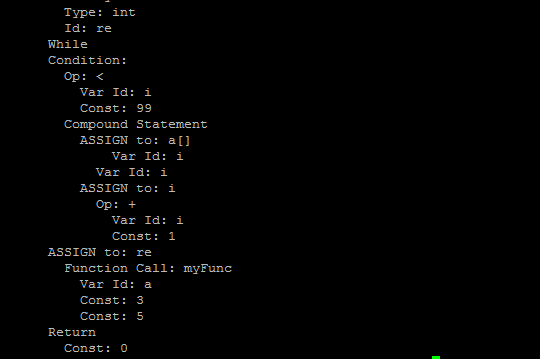
Parsing Error가 있는 코드로, Error가 잘 검출되는지 시험해보았다.



* 1. errorfree.c

정상적인 C- 코드로 Syntax Tree가 잘 생성되는지 확인하였다.





1. **평가 내용**

본 프로젝트는 2. 분석 내용에서 확인할 수 있는 C- Grammar Rule에 따라 작성되었다. 여러 가지의 Test에 대해서 Error가 없는 코드인 경우에 대해 문제없이 Parsing 되어 Syntax Tree 생성이 됨을 확인 할 수 있었다. Precedence와 Associativity, 그리고 if와 if-else에서의 shift-reduce conflict 또한 문제 없이 해결됨을 확인 할 수 있었다. Appendix A의 C- Language 문법을 정확히 따라 제작되었다. 따라서 앞서 Test한 C- Language로 작성된 프로그램들을 정확히 분석하는 것을 확인 할 수 있었다.

**V. 기타**

1. **자체 평가**

제작한 LALR Parser는 의도한 C- Language에 대해서 훌륭한 Parser 역할을 한다. 시험을 위해 제작했던 Test File들을 문제없이 통과하였다. 프로젝트를 진행하는 과정에 있어 GitHub를 사용해 코드를 관리하여, 신뢰성을 높였다.

1. **느낀 점**

LALR Parser를 제작하게 되면서, CFG를 BNF문법을 이용해서 작성할 수 있음을 알았다. 또한 Bison 툴에 대해서 알게되었고, 그 사용법을 배울 수 있어 좋았다.

1. http://www.gnu.org/software/bison/manual/html\_node/Shift\_002fReduce.html#Shift\_002fReduce [↑](#footnote-ref-1)