Design and Development of Compiler  
for C- Language

**Phase II**

**Design and Implementation**

**of LALR Parser**

**Project Result**

과목명: CSE4120 기초 컴파일러 구성

담당 교수: 서강대학교 컴퓨터공학과 정성원

개발자: 20141589 최광희

제출일: 2017. 5. 12.

1. **개발 목표**

전체 C- Compiler 설계 프로젝트 중 Phase II의 개발 목표는 LALR Parsing of C- Language로, C- Language로 쓰인 소스 코드를 Parsing할 수 있도록 한다. 구체적으로, LALR Parser을 만들어 Abstract Syntax Tree를 출력하는 것을 목표로 한다.

1. **개발 범위 및 내용**

**1. 개발 범위**

Phase II는 주어진 C- 언어 소스 코드를 Syntactic Analysis 과정을 거쳐 Stream of tokens를 Abstract syntax tree로 바꿀 수 있는 Software을 개발한다. Stream of tokens는 이전 Phase I의 Lexical Analysis를 통해 만들어낸 token과 일맥상통한다. 여기서 Parsing이란, 해당 stream of token이 grammatically well-formed인지 확인하는 것을 말한다. 이 과정에서 C- Language의 Context-free Grammar Production rule들로부터 LALR Parsing Table을 생성하는 과정은 bison이라는 Software가 대신 생성하므로, 이번 Phase에서는 bison이 이해할 수 있는 Production Rule form인 Backus-Naur form으로 소스 코드를 작성한 뒤, bison이 해당 코드를 변환시켜준 C 소스 코드를 통해 주어진 소스 코드를 Abstract Syntax Tree로 출력할 수 있도록 한다.

**2. 개발 내용**

C- 언어는 C 언어의 Subset으로써, 단순한 문법 구조를 갖고 있다. 이번 Phase에서는 먼저, 해당 언어의 분석을 통하여 BNF Grammar로 변환시키고, 해당 Grammar을 바탕으로 bison이 이해할 수 있는 Lexical Specification 소스 코드를 작성한다. 해당 BNF Grammar은 Compiler 수업의 교재에 작성되어있으므로, 해당 BNF Grammar을 Backus-Naur form으로 바꿔 cm.y 파일을 작성하면 된다.

위를 완성한 뒤에는, cm.y 파일을 bison이 변환을 한 뒤에 나오는 결과인 cm.tab.c 파일에 생성된 함수를 이용하여, C- 언어로 이뤄진 소스 코드를 Syntactically Analyze한다. 이후 Tree를 출력하기 위해 Phase I의 소스 코드를 이어서 작성하는데, 특히 printTree 함수를 수정하고, 만약 Parsing 시에 오류가 있다면 해당 오류 메세지와 Line number을 출력하는 프로그램을 만든다.

그 과정에서 필요한 C- 언어로 이뤄진 소스 코드 역시 작성한다. 정상 코드와 Syntactic Error가 있는 코드 두 개를 작성해야 한다.

1. **추진 일정 및 개발 방법**

**1. 추진 일정**

|  |
| --- |
| 5/5 ~ 5/7: 개발해야 할 C- Language의 Production Rule의 BNF Grammar 확인  5/7 ~ 5/8: Syntactic Specification 설계  5/8 ~ 5/10: AST Node 자료구조 설계 및 printTree 구현  5/10 ~ 5/11: Test case 작성 및 디버깅과 보고서 작성 |

예상보다 훨씬 빠르게 개발할 수 있었는데, 대부분의 코드가 이미 교과서에 설계되어 있었기 때문이다. Bison을 사용하여 개발해, Specification 구현과 실제 자료구조 구현을 쪼개 구현할 수 있었다.

각 Commit 단위로 쪼갠 구체적인 일정은 다음과 같다.

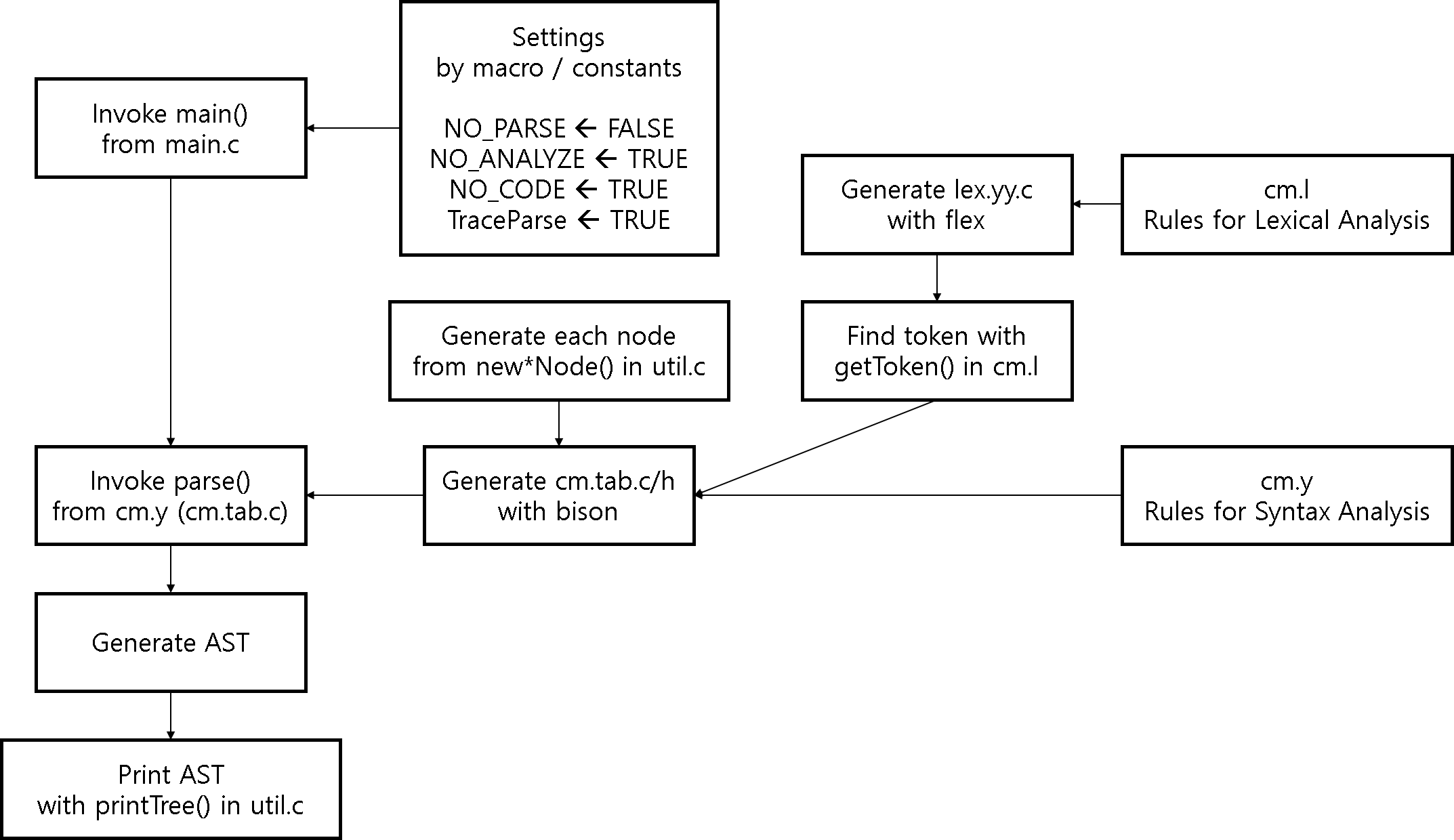
|  |
| --- |
| commit 6af3e4f57763b87f706e011072cb77aed07f69f0  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Mon May 8 03:58:00 2017 -0700  [minor] Rename cminus.l to cm.l  commit 4dc4e28fd635fc043f5bb92be77e4fe45144783f  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Mon May 8 04:37:51 2017 -0700  [minor] Reindent cm.l for consistent spacing  commit 3cccac2b65c7c772201f6a88528e4ac99ed2263a  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Mon May 8 06:26:00 2017 -0700  [minor] Add options at cm.l for removing default flex functions  commit 3fe40b06a19bf0b4cac7e7892327bd11b31a49f1  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Mon May 8 06:42:26 2017 -0700  Commit inital code from subject textbook    This code is from "Compiler Construction: Principles and Practice"  by Kenneth C. Louden.  commit c9301b6bf874be4d94fe63ff6ff943898e57e2df  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Mon May 8 06:43:11 2017 -0700  Change TINY Bison grammar to C- Bison grammar    File is loosely similar to TINY yacc file, which was given, to C-  Bison file. Reference can be found at Appendix A.1 Lexical Conventions  of C-. BNF Grammar of C- is added. Modification credit is stated at the  start of the modified file.  commit 9bda4eb982f1aec5cb8dc86a25c8cccfaf9e98fa  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Mon May 8 20:20:14 2017 -0700  [minor] Change indentation and printing format for printToken()  commit 3972ee28a144c8b941c61a5983952022f21ba6e2  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Mon May 8 20:38:23 2017 -0700  [minor] Exclude parse.c and parse.h to use Bison  commit fa859c9bf658773a9257449078b10b503154b3cd  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Mon May 8 23:00:32 2017 -0700  Prototype bison specifications of C-minus language with pseudocode  Note that this commit will not build successfully, as C-style  comments are located inadequately.  commit f04d210cf702006b0e211bfcd249959247c6da7f  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Wed May 10 03:50:26 2017 -0700  Modify prototype specification to be grammatically correct for bison    Several modification has been done for specification. This spec is  now correctly follows bison grammar, hence building successfully.  Also had to change some C minus grammar to remove conflicts -  dangling else statement conflict can be avoided by adjusting the  priority of if-else higher than if.  commit cf1e2ac33567ec35ce2c967dc59a9ded2c156fe3  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Wed May 10 03:54:41 2017 -0700  Modify data structure for abstract syntax tree node to serve for C minus  As TINY and C- are quite different languages, data structure had to  undergo major fixes.  1. Removed specific node variety, now every node is divided with single  enum NodeKind.  2. Removed child and added named children for each node kind.  commit 281b4a3b7483caf01d961a9cae88922694837aa0  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Wed May 10 04:00:56 2017 -0700  Implement prototype functions from bison specifications    1. Implemented setters for each specification  2. Implemented printable abstract syntax tree  Now this project successfully builds.  commit 04739ac3f56da01140bbd1a40a329f84a16d0a40  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Wed May 10 04:39:00 2017 -0700  [minor] Remove debugging flags  commit 5bffa5b69a0a1fd8ebda00cf49758bb004ec5e48  Author: Kwanghee Choi <juice500ml@gmail.com>  Date: Wed May 10 05:01:49 2017 -0700  [minor] Clean up printing format for printTree |

**2. 개발 방법**

개발 환경은 cspro9와 cspro10을 사용하여, Ubuntu 16.04.2 LTS 기준으로 작업한다. vi editor을 이용하여 편집을 하며, Indentation은 GNU Standard를 따른다. 주 개발 언어는 C이나, 이번 Phase의 경우 Syntactic Specification 소스 역시 작성한다. 컴파일은 gcc 5.4.0 을 통해 하며, Makefile을 이용하여 빌드 및 Testing의 자동화를 한다. git을 이용하여 Version Control을 하며, Github에서 remote branch를 관리하여, 매 Commit마다 각 case들에 대해 검증한다.

1. **연구 결과**

**1. 합성 내용**



전체 소프트웨어 분석도

* 교과서에서 주어진 TINY Compiler의 소스 코드를 기본으로 설계를 진행했다.
* Build 과정을 자동화하기 위해 Makefile을 사용하여 빌드 Rule들을 지정해주었다.
* 그 과정에서 쓰이는 프로그램이 flex인데, 해당 프로그램은 .lex 파일을 이용하여 각 Token에 대응되는 Rule들을 정해주면, 해당 Rule들을 통해 FSM을 생성한다.
* 이번 프로젝트에서 저번 프로젝트와 다르게 추가적으로 쓰이는 프로그램이 bison인데, 해당 프로그램은 .y 파일을 이용하여 위에서 발견한 token을 .y 내에 선언되어있는 BNF 문법을 기준으로 내부적으로 parsing을 위한 parsing table을 만들어 해당 table을 이용해 DFA를 만들어준다.
* 해당 DFA를 이용하여 main.c 내부에서 AST를 생성한 뒤, tree를 적절히 출력한다.

**2. 분석 내용**

이번 프로젝트에서는 크게 두 부분으로 나누어 개발했다. 먼저, BNF Grammar을 읽어내기 위한 Bison specification의 구현과, 해당 spec을 통해 사용할 수 있는 yyparse() 함수를 통해 실제 접근할 수 있는 Abstract Syntax Tree 자료구조이다. 순서대로 살펴보도록 하겠다.

**실제 개발에 이용된 C minus 언어를 표현하는 BNF Grammar을 교과서에서 발췌하여 각 Grammar rule에 대해 살펴보도록 하겠다. 여기서는 Semantics가 아닌 오직 Syntax만 다루도록 하겠다**.

1. program → declaration-list

2. declaration-list → declaration-list declaration | declaration

3. declaration → var-declaration | fun-declaration

C minus program은 선언의 집합으로 이루어져있다. 최소한 하나 이상의 declaration이 있어야 하고, declaration에는 Variable declaration과 Function declaration이 있다.

4. var-declaration → type-specifier ID ; | type-specifier ID [ NUM ] ;

5. type-specifier → int | void

Variable declaration은 단일 변수와 Array 두 종류가 있다. 각 declaration에는 해당 변수의 id가 있고, Array의 경우 꺽쇠 안에 정수가 있다. 해당 정수는 array의 길이가 된다. Type은 오직 int와 void 두 개만 있다.

6. fun-declaration → type-specifier ID ( params ) compound-stmt

7. params → param-list | void

8. param-list → param-list , param | param

9. param → type-specifier ID | type-specifier ID [ ]

Function declaration은 먼저 해당 함수의 return value의 type, 어떤 parameter가 들어가는지, 실제 함수 내부의 code 총 세 개가 필요하다. parameter의 경우 parameter의 comma-seperated list가 되어야 하나, array가 parameter인 경우 숫자 없이 오직 [ ] 만 사용한다. 내부의 compound statement에 대해서는 바로 아래에서 살펴보도록 하겠다.

10. compound-stmt → { local-declarations statement-list }

11. local-declarations → local-declarations var-declaration | empty

12. statement-list → statement-list statement | empty

Compound statement는 중괄호로 묶여있고, 내부에 local 변수를 전역변수와 마찬가지로 선언할 수 있고, statement의 list로 되어있는데, 선언부와 statement 모두 비어있을 수 있다.

13. statement → expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt

Statement의 종류는 총 다섯 개이다.

14. expession-stmt → expression ; | ;

Expression statement는 비어있어 세미콜론으로만 끝날 수도 있고, 내부에 expression이 들어갈 수도 있다. 주로 변수 assignment나 함수 call에 쓰인다.

15. selection-stmt → if ( expression ) statement | if ( expression ) statement else statement

16. iteration-stmt — while ( expression ) statement

17. return-stmt —> return ; | return expression ;

차례대로 if-else clause, while loop, return statement이다. if-else clause는 내부의 expression을 evaluate하여 if clause에 들어올지 else clause에 들어올지를 결정하는데, else clause는 없을 수도 있다. while loop 역시 내부의 expression을 evaluate하여 loop을 할지 결정한다. return statement는 아무 것도 return하지 않을 수도 있고, expression의 결과를 return할 수도 있다. if-else clause의 경우 conflict가 날 수 밖에 없는데, 이에 대한 자세한 설명은 밑에서 서술하도록 하겠다.

18. expression → var = expression | simple-expression

19. var → ID | ID [ expression ]

Expression이란 Variable reference와 expression이 assignment되거나, simple expression 둘 중 하나이다. Simple expression은 바로 다음에 설명하도록 하겠다.

20. simple-expression → additive-expression relop additive-expression | additive-expression

21. relop → <= | < | > | >= | == | !=

Simple expression은 relational operator을 사이에 둔 additive expression이거나, additive expression 자체 둘 중에 하나이다. additive expression은 바로 다음에 설명하도록 하겠다. relational operator을 사이에 둔 경우, 해당 값의 참, 거짓을 판별하는데 쓰인다.

22. additive-expression → additive-expression addop term | term

23. addop → + | -

24. term → term mulop factor | factor

25. mulop → \* | /

Additive expression은 일반적으로 흔히 쓰이는 형태의 associativity와 precedence를 갖는 사칙연산 expression을 의미한다.

26. factor → ( expression ) | var | call | NUM

factor은 괄호로 싸인 expression이거나, variable이거나, 함수 call이거나, 상수를 의미한다.

27. call → ID ( args )

28. args → arg-list | empty

29. arg-list → arg-list , expression | expression

call은 function call을 의미하는데, 해당 함수의 ID와 함수에 들어갈 arguments가 필요하다. 다만, argument는 비어있을 수 있고, 만약 있다면 해당 argument에 comma-seperated expression이 들어갈 수 있다.

위에서 소개한 BNF Grammar을 그대로 Bison에 작성하여 넣으면 1개의 Conflict가 일어난다. 해당 Bison에서 출력한 output은 다음과 같다.

|  |
| --- |
| State 99 conflicts: 1 shift/reduce  …  State 99  29 selection\_stmt: IF LPAREN expression RPAREN statement .  30 | IF LPAREN expression RPAREN statement . ELSE statement  ELSE shift, and go to state 102  ELSE [reduce using rule 29 (selection\_stmt)]  $default reduce using rule 29 (selection\_stmt) |

해당 Conflict는 dangling-else problem으로, if문의 expression이 끝나고 난 뒤 else 토큰이 등장했을 때 이전까지의 if statement를 Rule 29를 이용하여 Reduce시킬 것인가, 아니면 Rule 30에 입각하여 else 토큰을 shift할 것인가,의 conflict가 일어난다. 즉, SR Conflict (Shift-Reduce Conflict)가 일어난다. 이 상황에서는, else로 새 node를 시작하는 경우는 없으므로 else 토큰을 shift하는게 맞는 선택이다.

**Precedence Directives with explanations**

이 문제를 해결하기 위해서 %nonassoc directive를 사용하여, LOWER\_ELSE라는 새로운 token을 introduce했다. LOWER\_ELSE가 ELSE보다 윗줄에 써있으므로, ELSE의 precedence가 더 높게 된다. 해당 token을 위에 설명한 대로 ELSE가 붙은 rule이 명시적으로 더 높은 precedence를 같게 하기 위해, LOWER\_ELSE를 ELSE가 붙지 않은 rule에 %prec을 이용하여 해당 rule의 precedence를 결정해주었다. 이렇게 하면, ELSE token이 왔을 때 위의 reduce using rule 29 (selection\_stmt) 대신 ELSE shift, and go to state 102이 결정되어 conflict가 사라진다.

|  |
| --- |
| %nonassoc LOWER\_ELSE  %nonassoc ELSE  …  selection\_stmt  : IF LPAREN expression RPAREN statement  { $$ = newSelectionStatementNode($3, $5, NULL); }  %prec LOWER\_ELSE  | IF LPAREN expression RPAREN statement ELSE statement  { $$ = newSelectionStatementNode($3, $5, $7); }  ; |

실제로 해당 directive들을 추가한 뒤에 해당 State에서 conflict가 사라진 것을 볼 수 있다.

|  |
| --- |
| State 99  29 selection\_stmt: IF LPAREN expression RPAREN statement .  30 | IF LPAREN expression RPAREN statement . ELSE statement  ELSE shift, and go to state 102  $default reduce using rule 29 (selection\_stmt) |

**3. 제작 내용**

전체 C- Compiler 설계 프로젝트 중 Phase II: LALR Parser for C- Language로, C- Language로 쓰인 소스 코드를 Syntax Analysis를 하기 위하여 Abstract Syntax Tree를 생성한 뒤, 해당 Tree를 보기 위하여 Tree를 적절히 Level 별로 indent를 하여 출력하는 프로그램을 작성했다. 실제 구현은 교재에서 제공된 C언어로 되어 있는 TINY Compiler 코드를 base로 하여, 해당 base 위에서 주어진 코드를 수정하여 작성했다.

**Explanation on how to run this program**

AST를 생성할 때 BNF Grammar을 간편한 build를 위해 Makefile을 제작하여 해당 프로젝트 폴더에서 make라고만 치면 자동으로 dependency를 검사하여 빌드를 한다.

실제 예시는 errorfree.tny, error.tny로 하였다. 빌드 후 해당 예시 코드에 대해 parsing을 하려면, ./20141589 errorfree.tny 와 ./20141589 error.tny 로 할 수 있다.

**4. 시험 내용**

해당 LALR Parser을 시험하기 위해 총 **두 Test C- program**을 작성하였다. 먼저 error.c는 C- 언어 상에서 Syntax Error을 return해야 하는 코드이고, errorfree.c는 C- 언어에서 Syntax Error가 없는 코드이다. 각 Test program code와 구현한 Parser의 output을 살펴보면 다음과 같다.

|  |
| --- |
| /\* Global variable declaration \*/  int i;  int a[101];  /\* Function declaration \*/  int func(int n, int arr[])  {  /\* Local variables \*/  int locali;  int locala[14];  /\* while and if \*/  while(locali < 10) {  /\* Expressions \*/  locali = (3 + locala[3]) \* 4 + 2 / 1  if (i >= 3) {  if (locali > 1) {  i = i + locali;  }  else {  return 3;  }  }  }  return locali \* n;  } |
| error.tny  “locali = (3 + locala[3]) \* 4 + 2 / 1” Line 16에서 semicolon이 빠져있다. |

|  |
| --- |
| Syntax error at line 16: syntax error  Current token: IF if  Syntax tree:  (null) |
| error.tny를 parsing한 결과  Line 15에서 세미콜론이 빠져있으므로 Line 16에서 if문이 시작되자 에러를 내는 것을 볼 수 있다. |

|  |
| --- |
| /\* Global variable declaration \*/  int i;  int a[101];  /\* Function declaration \*/  int func(int n, int arr[])  {  /\* Local variables \*/  int locali;  int locala[14];  /\* while and if \*/  while(locali < 10) {  /\* Expressions \*/  locali = (3 + locala[3]) \* 4 + 2 / 1;  if (i >= 3) {  if (locali > 1) {  i = i + locali;  }  else {  return 3;  }  }  }  return locali \* n;  }  void main(void) {  i = a[func(i, a)];  return;  } |
| errorfree.tny  C-minus 문법에 맞는 코드이다. |

|  |
| --- |
| Syntax tree:  Variable Declaration  Token : int  Variable Id : i  Array Declaration  Token : int  Variable Id : a  Constant : 101  Function Declaration  Token : int  Variable Id : func  > Parameters :  Parameter (Variable)  Token : int  Variable Id : n  Parameter (Array)  Token : int  Variable Id : arr  > Function Block :  Compound Statement  > Local Declarations :  Variable Declaration  Token : int  Variable Id : locali  Array Declaration  Token : int  Variable Id : locala  Constant : 14  > Local Statements :  Iteration Statement  > Expression inside while(\*) :  Comparison Expression  Token : <  > Left expression compared :  Variable Id : locali  > Right expression compared :  Constant : 10  > Statements inside while clause :  Compound Statement  > Local Declarations :  (null)  > Local Statements :  Expression Statement  > Expression :  Assignment Expression  > Variable associated to assignment :  Variable Id : locali  > Value assigned :  Additive Expression  Token : +  > Left expression added / subtracted :  Multiplicative Expression  Token : \*  > Left expression multiplied / divided :  Additive Expression  Token : +  > Left expression added / subtracted :  Constant : 3  > Right expression added / subtracted :  Variable Id : (null)  > Right expression multiplied / divided :  Constant : 4  > Right expression added / subtracted :  Multiplicative Expression  Token : /  > Left expression multiplied / divided :  Constant : 2  > Right expression multiplied / divided :  Constant : 1  Selection Statement  > Expression inside if(\*) :  Comparison Expression  Token : >=  > Left expression compared :  Variable Id : i  > Right expression compared :  Constant : 3  > Statements inside if clause :  Compound Statement  > Local Declarations :  (null)  > Local Statements :  Selection Statement  > Expression inside if(\*) :  Comparison Expression  Token : >  > Left expression compared :  Variable Id : locali  > Right expression compared :  Constant : 1  > Statements inside if clause :  Compound Statement  > Local Declarations :  (null)  > Local Statements :  Expression Statement  > Expression :  Assignment Expression  > Variable associated to assignment :  Variable Id : i  > Value assigned :  Additive Expression  Token : +  > Left expression added / subtracted :  Variable Id : i  > Right expression added / subtracted :  Variable Id : locali  > Statements inside else clause :  Compound Statement  > Local Declarations :  (null)  > Local Statements :  Return Statement  > Returning expression :  Constant : 3  > Statements inside else clause :  (null)  Return Statement  > Returning expression :  Multiplicative Expression  Token : \*  > Left expression multiplied / divided :  Variable Id : locali  > Right expression multiplied / divided :  Variable Id : n  Function Declaration  Token : void  Variable Id : main  > Parameters :  (null)  > Function Block :  Compound Statement  > Local Declarations :  (null)  > Local Statements :  Expression Statement  > Expression :  Assignment Expression  > Variable associated to assignment :  Variable Id : i  > Value assigned :  Variable Id : (null)  Return Statement  > Returning expression :  (null) |
| errorfree.tny를 파싱한 결과  C-minus 문법에 맞는 코드이므로, 코드에 대응되는 알맞는 AST가 출력됨을 알 수 있다. |

**5. 평가 내용**

bison을 처음 사용하여, 각종 directive와 bison grammar에 익숙하지 않아 적응하는 데에 오래 걸렸으나, 적응한 뒤에 실제 AST를 생성하는 코드와 연결할 때는 매우 명확하고 편리하게 할 수 있었다. 또한, bison의 verbose output을 통해 conflict를 발견하고 분석하여 해결책을 만들어내고, 이전 project 1에서의 lexical analyzer에 약간의 수정을 가하여 해당 코드가 정확히 어디서 오류가 생겼는지 명시를 하는 등 여러 문제가 있었으나 기존에 존재하는 규칙들에 어긋나지 않을 수 있도록 고민하여 design 결정을 했고, 그 과정에서 여러 예외에 대해 안정적으로 처리할 수 있는 LALR Parser을 설계할 수 있었다.

개발시에 Git을 사용하여 Commit을 잘게 쪼개 각 변경마다 어떤 사항이 변경되었는가를 Tracking 하도록 하여, 신뢰성 있는 Code 작성 및 Continuous Integration 뿐만 아니라 과제를 제출한 뒤에 지속적으로 타인들과 함께 Open Source Contribution을 할 수 있도록 했다. 또한 문제가 생겼을 때 Revert나 Cherry-pick 등 여러 Version Control 기능을 사용할 수 있어, 개발을 훨씬 안정적으로 할 수 있었다. 해당 코드를 Github에 올림으로써, Local Repository에서만 관리하는 것이 아닌 Remote Repository를 두어, 한 쪽 Code가 날아가더라도 전혀 지장이 없도록 했다.

1. **기타**

위의 5. 평가에서 언급된 바와 같이, C- 언어의 Syntax가 명확하지 않은 부분이 있어, 기존에 존재하는 textbook과 C Language family가 주로 갖게 되는 규칙들에 어긋나지 않도록 design 결정을 했고, 그 과정에서 여러 예외에 대해 안정적으로 처리할 수 있는 LALR Parser을 설계할 수 있었다. 또한, 개발시에 Git을 사용하여 신뢰성 있는 Continuous Integration 뿐만 아니라 이후에도 지속적으로 Open Source Contribution을 할 수 있도록 했다. 문제가 생기더라도 여러 Version Control 기능을 통해 안정적인 개발을 할 수 있었다. 해당 코드는 Github를 통해 Remote Repository를 두어 한 쪽 Repo에 문제가 생기더라도 문제 없도록 했다.