Design and Development of Compiler for C- Language
(설계 프로젝트 수행 결과)

과목명: [CSE4120] 기초 컴파일러 구성
담당교수: 서강대학교 컴퓨터공학과 정 성 원
개발자: 20151623 한상구
개발기간: 2019. 5. 3. - 2019. 5. 3

각 단계별 결과 보 고 서

프로젝트 제목: Design and Development of Compiler for C- Language:
Phase 2: Design and Implementation of LALR Parser

제출일: 2019. 5. 3.

개발자: 한상구

I. 개발 목표

Bison(or Yacc)를 이용하여 C- language 에 대해 parsing 을 수행할 수 있는 LALR parser 를 구현한다.

C- language 에 관한 세부 내용은 교재[1] Appendix A 를 참조하며, Makefile 을 통해 build 한 parser 는 주어진 *.tny 파일의 syntax tree 를 적절하게 생성할 수 있어야 한다.

BNF grammar 는 A.2 SYNTAX AND SEMANTIX OF C-[2] 에서 주어졌으며, 본 grammar 에서 발생하는 Shift/Reduce Conflict 에 대해 C- Language 에 적절한 방향으로 parsing 할 수 있도록 action 을 설계한다.

II. 개발 범위 및 내용

가. 개발 범위:

1. Bison 을 이용한 parsing, 생성된 syntax tree 를 이용해 적절히 콘솔 상에 출력
2. Shift/Reduce Conflict 해결
3. Syntax error handling

나. 개발 내용:

1. Yacc 을 이용해 C- language 의 BNF grammar 를 구현
2. Shift/Reduce Conflict 발생 시 어떤 action 을 실행 할 것인지 판단 및 구현
3. Syntax error 에 관한 handling aux routine 구현

III. 추진 일정 및 개발 방법

가. 추진 일정:

~2019.05.03 LALR parser 설계 및 구현

나. 개발 방법:

Bison 을 이용하여 BNF grammar parsing routine 작성 (Yacc), 이후 생성된 syntax tree 출력 (C)

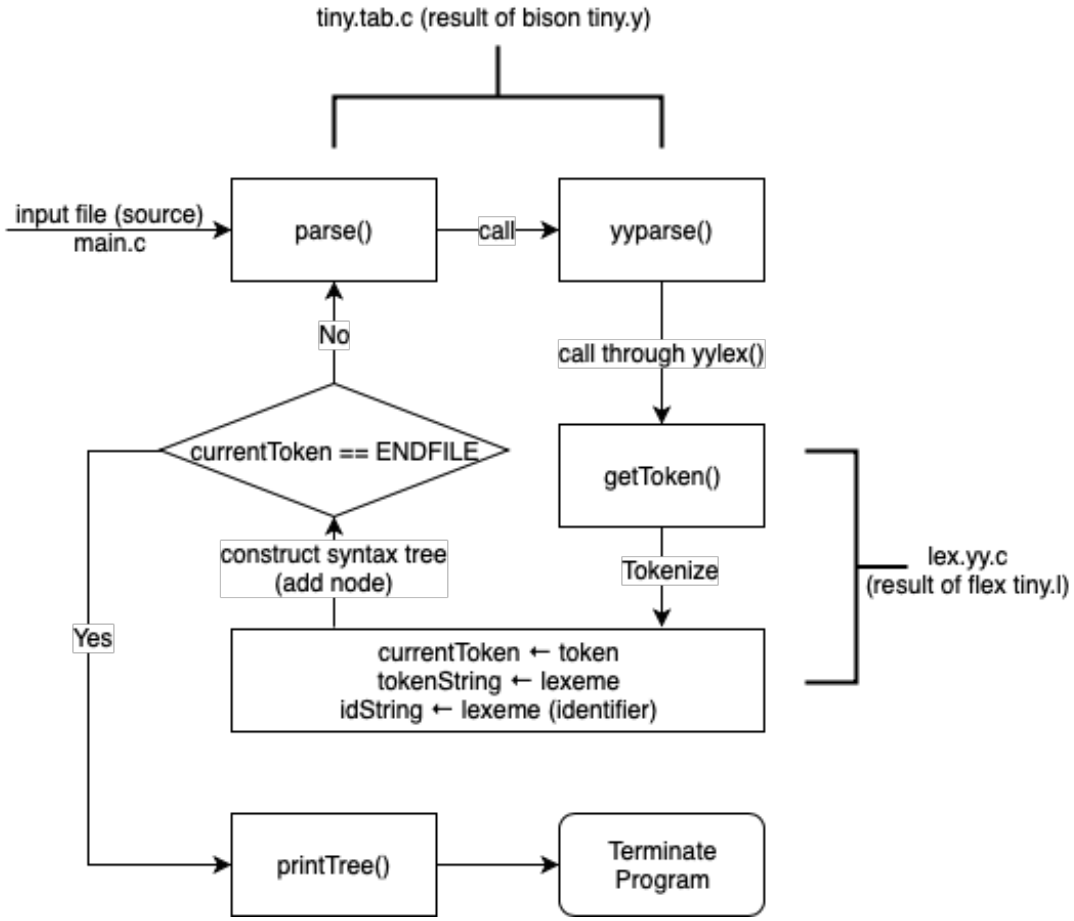
---

[1] Compiler Construction Principles and Practice, K. C. Louden

[2] Compiler Construction Principles and Practice, K. C. Louden , p.492

IV. 연구 결과

1. 합성 내용:



<그림 1 - C- LALR parser 의 간단한 flow chart>

tiny.y 의 aux subroutine 에 정의된 parse()를 기점으로 동작하는 LALR parser 입니다. 큰 관점에서 Bison 은 Flex 의 위에서 동작한다고 볼 수 있는데, Flex 를 이용하여 주어진 source 를 digest 하고, 그 과정에서 발생하는 token 을 기반으로 tiny.tab.c 에 구현된 LALR parsing table 에 따라 parsing 을 진행하기 때문입니다.

2. 분석 내용:

본 프로젝트에서 요구한 파일은 총 5 개입니다. 위 flow chart 와 연계하여 각 구성 요소를 설명하겠습니다.

- main.c

Source file 을 읽고, parse()를 호출하여 parsing 을 initiate 시킵니다. parse()는 parsing 이 정상적으로 종료되었을 시 syntax tree 를 반환하고, 이 tree 를 출력하는 printTree()를 호출한 뒤 프로그램을 종료시킵니다. 위 flow chart 에서 시작하는 부분, printTree()를 호출하고 프로그램을 종료하는 부분이 본 코드에 구현되어 있습니다.

- globals.h

위 flow chart 에서 전체적으로 사용되는 변수 및 타입, 매크로를 가지고 있는 헤더파일 입니다. syntax tree 의 node 구조, token type (프로젝트 1 에서 직접 선언한 것과 달리 Bison 의 output 인 tiny.tab.h 에 정의된 token type 을 가져와서 사용) 등이 선언되어 있습니다.

- util.c, util.h

위 flow chart 에서 syntax tree 를 출력하는 코드와 LALR parser 가 syntax tree 를 구축하기 위한 코드가 구현되어 있습니다.

- tiny.y

Bison 문법을 이용해 BNF grammar of C- language 의 parsing 을 구현한 코드입니다 (production rule 명시 및 해당 rule 에서 parser 가 할 action 을 구현).

parse()를 통해 yyparse()를 initiate, syntax tree 를 구축하며 이를 반환합니다.

3. 제작 내용:

BNF grammar 를 Bison 문법에 맞게 옮긴 뒤, 각 rule 에 맞는 semantic action 을 설정하여 syntax tree 를 구성할 수 있도록 했습니다. 주어진 BNF grammar of C- language 는 Shift/Reduce Conflict 가 하나 존재하며, 이는 selection-stmt 에서 발생하는 `dangling else[3]` 가 원인입니다.

C- language 에서 원하는 해결 방안은 shift 를 action 으로 선택하여 해결하는 것입니다. 실제 Bison 구현은 Shift/Reduce Conflict 발생 시 기본적으로 shift 를 action 으로 취하도록 되어있기 때문에[4], 따로 수정이나 추가 명시는 필요하지 않았습니다.

--verbose flag 를 통해 tiny.output 파일을 생성하여 확인해보면 shift 가 action 으로 취해져 있는 것을 확인할 수 있습니다.

```
State 101

   34 select_stmt: IF LPAREN expr RPAREN stmt .
   35            | IF LPAREN expr RPAREN stmt . ELSE stmt

    ELSE  shift, and go to state 106

    ELSE      [reduce using rule 34 (select_stmt)]
    $default  reduce using rule 34 (select_stmt)
```

<그림 2 - tiny.output 에서 Shift/Reduce Conflict 가 발생한 State>

syntax tree 는 모든 leaf node 가 최대한 atomic 하도록 설계했으며, 그 결과 다음과 같이 설계할 수 있었습니다.

A. ExprKind - expression kind; 변수, 상수, 함수등을 이용한 연산 정보를 담음

(1) leaf node (atomic node, 0 child) - IdK, ConstK, TypeK

각각 이름, 상수값, 타입을 담는 노드입니다. syntax tree 를 구성하는 노드 중 atomic 한 노드들이며, 이 3 개의 노드를 조합한다면 현재 syntax tree 구성에 필요한 모든 종류의 노드를 구성할 수 있습니다. 부모 노드가 이 노드를 참조하여 자신의 타입, 이름, 값을 가져올 수 있도록 했습니다.

(2) node with 2 children - OpK, ArrSubK, FunCallK

---

[3] https://en.wikipedia.org/wiki/Dangling_else

[4] … Bison is designed to resolve these conflicts by choosing to shift …, Bison Document 5.2 (https://www.gnu.org/software/bison/manual/html_node/Shift_002fReduce.html#Shift_002fReduce)

OpK 는 두 변수(혹은 변수와 상수)간의 operation 을 담는 노드입니다. 2 개의 child 를 가지며, 각 노드에는 또 다른 OpK 형식의 노드를 갖거나 위 leaf node 중 IdK 혹은 ConstK 를 갖습니다. (변수간의 연산, 변수-상수간의 연산) 아직 Semantic analysis 를 진행하지 않기 때문에 type 은 고려하지 않았습니다.

ArrSubK 는 array 에서 특정 index 의 값을 담는 노드입니다. 첫번째 child 에는 array 의 이름을 갖는 leaf node, 두번째 child 에는 index 의 정보를 담는 expression kind node 를 갖도록 설계했습니다.

FunCallK 는 function 을 호출하는 파트를 담는 노드입니다. 첫번째 child 는 function 의 이름을 담을 leaf node(IdK 를 이용), 두번째 child 는 argument list 를 담기위한 expression kind node 를 갖도록 설계했습니다.

B. DeclKind - declaration kind; 변수, 함수의 선언 정보를 담음

(1) variable declaration node - VarK

array 가 아닌 변수를 선언하는 정보를 담은 노드입니다. 변수 선언에는 타입, 이름이 필요하기 때문에 각 정보를 담은 2 개의 leaf node(TypeK, IdK)를 child 로 가지도록 설계했습니다.

(2) array declaration node - ArrayK

array 의 경우 타입, 이름과 추가로 크기까지 저장해야하기 때문에, 크기 정보를 담을 수 있는 leaf node(ConstK)를 추가로 가지도록 설계했습니다.

(3) function declaration node - FunK

function 의 경우 타입, 이름을 위한 2 개의 leaf node(TypeK, IdK), parameter 를 위한 1 개의 expression kind node, body 정보를 위한 1 개의 compound statement node(CompK)를 child 로 가지도록 설계했습니다.

C. StmtKind - statement kind; 프로그램 블록단위의 정보를 담음

(1) return statement - ReturnK

return 관련 정보를 담은 노드입니다. function type 이 int 인 경우, 뒤따라오는 expression 의 정보를 담기 위한 노드를 child 로 가지며, void 인 경우 child 없이 return 임을 명시하는 정보만을 들고 있습니다.

(2) if statement - IfK

if-else(if any) 관련 정보를 담은 노드입니다. condition 을 담기 위한 expression kind node 를 첫번째 child 로, condition 이 만족될 경우 실행할 body 파트의 정보를 담기 위한 statement kind node 를 두번째 child 로 갖도록 설계했습니다. else 파트가 존재한다면 else 의 body 파트 정보를 담기 위한 statement kind node 를 세번째 child 로 가지게 됩니다.

(3) while statement - WhileK

while loop 관련 정보를 담은 노드입니다. condition 을 담기 위한 expression kind node 를 첫번째 child 로, loop body 의 정보를 담기 위한 statement kind node 를 두번째 child 로 갖도록 설계했습니다.

(4) compound statement - CompK

위 (1), (2), (3)과 다르게 여러 statement 및 declaration 정보를 관리하기 위한 노드입니다. C-language 에서는 scope 최상단에 declaration 이 위치하고, 이후 이를 사용하는 statement 가 등장하기 때문에 declaration kind node 를 첫번째 child 로, statement kind node 를 두번째 child 로 갖도록 설계했습니다.

위와 같이 syntax tree node 를 설계하게 된다면, syntax tree 를 출력하기 위한 로직은 다음과 같이 단순해집니다.

"순차적으로 tree 의 root 부터 self - child - sibling 으로 탐색하며 출력"

self - child 로 이어지는 탐색으로 통해 본인의 모든 정보를 획득 및 출력하며, 이후 sibling 을 출력하는 순서로 이어지게 된다면 본 프로젝트에서 원하는 syntax tree 의 정상적인 출력이 완료됩니다.

실제 util.c 의 printTree()를 확인해보면, leaf node 에 해당하는 case 이외에는 본인의 역할만을 출력하고 있음을 확인할 수 있습니다 (OpK 는 예외로 operator 를 출력합니다) - 이를 통해 atomic 한 노드를 잘 선별, 이를 이용해 syntax tree 를 적절하게 구성했음을 확인할 수 있습니다.

프로젝트 1 에서 통일했던 interface 에서 ENDFILE 을 제외해야 했는데, 이는 Bison 에서 implicit 하게 EOF 를 ENDFILE 이라는 이름으로 사용하고 있어 발생한 이슈입니다. 이를 제외하고는 tiny.y 에서 %token 뒤에 선언한 순서대로 token 의 번호가 매겨지는 것에 착안, token type 을 tiny.tab.h 에서 가져와도 프로젝트 1 에서 사용하던 interface 를 그대로 사용할 수 있도록 코드를 작성했습니다 (IF 가 가장 먼저 선언한 token 이기 때문에, currentToken - IF 를 하면 enum-name table 상의 본인 index 를 구할 수 있습니다).

4. 시험 내용:

교재 Appendix 에 나와있는 C- language syntax 를 따른 sort 파일 (이하 sort.tny)과, C- language syntax 를 따르지 않는 파일(변수의 선언과 초기화를 동시에 하는 라인이 포함됨, 이하 error.tny), dangling else 가 포함된 파일 (이하 dangle.tny)을 만들어 시험했습니다.

C- language syntax 를 따르는 두 파일 sort.tny, dangle.tny 에 대해서는 Shift/Reduce Conflict 도 적절히 resolve 한 syntax tree 를 생성, 출력하는 것을 확인 할 수 있었습니다.

에러가 있는 파일인 error.tny 에 대해서는 에러가 존재하는 라인과 토큰을 출력하고, syntax tree 는 출력하지 않고 종료하는 것을 확인할 수 있었습니다.

5. 평가 내용:

본 프로젝트에서 요구하는 기능 모두를 구현하였습니다. 추가로 atomic 한 node 를 통한 node composing, enum-name table 을 이용하여 최대한 간결한 switch 구문을 작성하였습니다. 이를 통해 보다 가독성이 뛰어난 코드를 작성할 수 있었습니다.

V. 기타

1. 자체 평가:

본 프로젝트가 요구하는 사항의 만족 이외에도 여러 부분을 고민하며 코드를 작성했습니다.

주로 코드의 직관성 및 간결함에 강점을 둘 수 있도록 고민한 결과, 다음과 같은 결과를 얻을 수 있었습니다.

token type 의 변경에도, 기존 프로젝트 1 에서 따랐던 coding convention 을 유지하며 가독성을 높일 수 있었습니다. atomic 한 element 를 조합하여 필요한 element 를 만들어내는 작업과, 이전 프로젝트의 interface 를 깨뜨리지 않고 이어서 사용할 수 있도록 작성함에 따라 보다 간결하고 직관적인 코드를 작성할 수 있었습니다.

2. 기타 자유 서술:

Bison 의 동작 방식 및 문법, Bison-Flex 간 연계, 그리고 syntax tree construction 에 대한 전반적인 시야를 얻을 수 있었습니다.