Compiler Project

2014313303 홍태하

2014313873 하승준

2014312541 장태영

2014310355 김현승

1. Compiler

컴파일러(Compiler)란 source language로 쓰여진 구문을 받아와 동일한 의미를 지니는 target language로 생성해주는 프로그램을 의미한다. 컴파일러가 가지는 특징은 다음과 같다. 첫째, 입력 언어의 구문 구조(Grammar)에 대한 이해를 기반으로 한다. 둘째, 복잡한 알고리즘 매핑을 포함한다. 마지막으로 효율성을 위해서 target code를 생성한다.

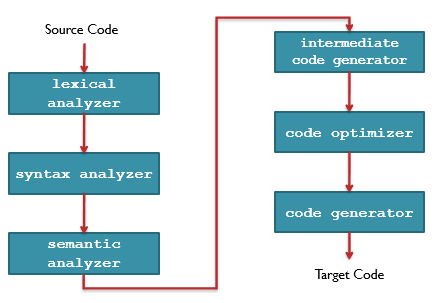


Figure 1. Compile 절차

컴파일러는 [Figure 1]과 같은 과정을 통해 Source code에서 Target Code를 생성한다. 이번 프로젝트에서 우리가 해야 할 과제는 lexical analyzer와 syntax analyzer를 구현하여 컴파일러를 구현하는 것이다.

1.1. Scanner

스캐너(Scanner)란 lexical analysis 과정에서 사용되는 프로그램을 말한다. 스캐너의 역할은 심볼 테이블(symbol table)에서 입력 값을 읽고, 그것을 토큰(Token)으로 그룹화 하는 것이다. 토큰이란 문법적으로 더 이상 나눌 수 없는 기본적인 언어 요소를 가리킨다. 토큰으로 나눠지면 그것을 파서(parser)로 전달하여 다음 컴파일 과정을 진행할 수 있도록 한다. lexical analyzer를 구현하는 방법은 크게 3가지가 있다. 첫째, ‘lex’, ‘flex’와 같은 도구를 사용하여 regular expression을 기반으로 작성하는 방법. 둘째, C, C++과 같은 프로그래밍 언어를 사용하여 작성하는 방법. 마지막으로 어셈블리 언어로 작성하는 방법이다. 이 3가지 방법 중에서 우리는 ‘flex’라는 도구를 이용하여 lexical analyzer를 구현하도록 하겠다.

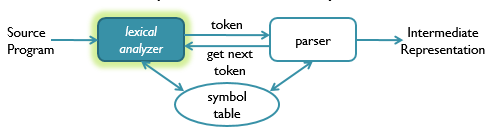


Figure 2. lexical analyzer

1.2. Parser

파서(Parser)란 syntax analysis 과정에서 사용되는 프로그램을 말한다. 파서는 lexical analyzer로부터 토큰을 받아와 parse tree를 생성하고, 언어의 grammar와 일치하는지 판별한다. 즉, 문장의 구조를 알아내는 구문 분석을 행하는 프로그램이다. 파서는 크게 Top-down을 기반으로 하는 LL 파서, Bottom-up을 기반으로 하는 LR 파서로 구분된다. 이번 과제에서는 일반적으로 lexical analyzer를 구현할 수 있는 도구인 ‘flex’와 함께 사용되는 ‘bison’이라는 도구를 이용하여 파서를 구현하도록 하겠다.

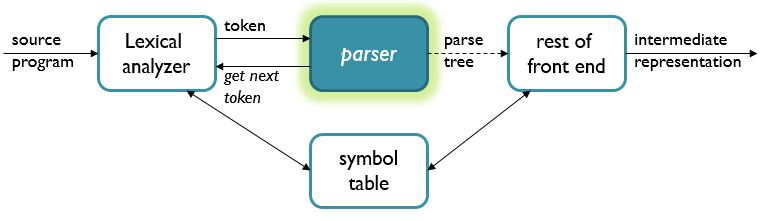


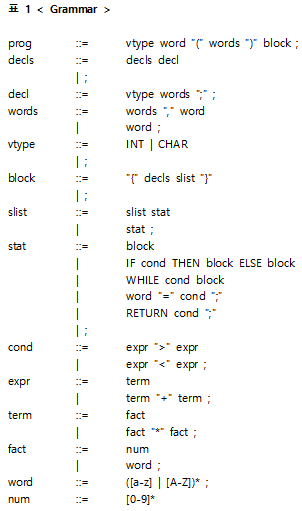
Figure 3. parser

1.3. flex & bison

이번 과제에서 사용하게 될 도구인 flex & bison에 대해서 간략하게 설명하도록 하겠다. flex & bison은 lex & yacc을 기반으로 하여 기존의 도구들을 개선하고 대체하기 위해 만들어졌다. lex & yacc은 벨연구소(Bell Laboratories)에서 70년대에 개발한 UNIX 환경에서 사용되는 컴파일러 작성 도구이다. 이 도구는 컴파일러의 작성을 매우 간단하게 처리할 수 있는 훌륭한 기능을 가지고 있었으며, 계속 발전하여 지금 우리가 사용하는 현대적인 기능을 가진 버전까지 발표되었다. lex는 입력된 문자에서 매칭되는 문자열의 패턴을 이용하여 문자열을 토큰으로 변환한다. 즉, 앞서 설명한 스캐너에 해당한다. flex는 lex의 GNU 법전으로 여기에 F(Fast)를 붙여 이전 버전의 lex와 구분한다. yacc는 ‘Yet Another Compiler Compiler’의 약자로 컴파일러 생성을 위한 프로그램이다. compiler-generator 혹은 compiler-compiler로 불리기도 한다. yacc는 lex로부터 토큰을 받아들이고 이것을 문법룰에 넣고 돌려서 syntax tree를 생성한다. 그리고 다음 단계에서 code generating을 하게 되고, 이는 DFS 기반으로 이루어지게 된다. 이러한 과정에서 lex를 호출하기 때문에 yacc은 lex의 상위에 위치하게 된다. 만들어지는 코드는 시스템의 assembler가 해석할 수 있는 assembly code 또는 machine code가 된다. 이 역시도 앞서 설명한 파서에 해당한다. Bison은 이러한 일을 하는 yacc의 GNU버전이며, 거의 동일한 역할을 수행한다.

2. Project

이번 프로젝트는 프로그래밍 언어 수업시간에 배운 방식을 사용하여 Finite State Automata의 하나인 스캐너와 파서를 구현하는 것이다. 우리는 Linux에서 flex & bison 도구를 사용하여 구현하기로 하였다. 주어진 Grammar는 다음과 같다.



2.1. Modification

과제에서 기존에 주어진 Grammar는 다음과 같다.

prog  ::=  vtype word "(" words ")" block ;

decls  ::=  decls decl

| ;

decl  ::=  vtype words ";" ;

words ::= words "," word

   |  word ;

vtype  ::=  INT | CHAR

| ;

block ::= "{" decls slist "}"

| ;

slist  ::=  slist stat

|  stat ;

stat  ::=  block

|  IF cond THEN block ELSE block

|  WHILE cond block

|  word "=" cond ";"

|  RETURN cond ";"

| ;

cond  ::=  expr ">" expr

|  expr "<" expr ;

expr  ::=  term

|  term "+" term ;

term  ::=  fact

| fact "\*" fact ;

fact  ::=  num

|  word ;

word ::=  ([a-z] | [A-Z])\* ;

num ::= [0-9]\*

이를 다음과 같은 이유로 일부분 수정하였다.

1. ‘expre’ non-terminal 추가

expre ::= expr

| term

| cond

| ( expre )

| num

| word

expr, term, cond의 연산들을 한번에 처리하기 위하여 expre라는 새로운 non-terminal을 추가하였고, num과 word는 return 과정에서 한번에 처리하기 위함이다. 그리고 C언어 기반으로 tool을 이해하다 보니 연산 순서를 명확하게 하기 위해 괄호 내에 있는 expre도 따로 추가하였다.

1. expr ::= expre “+” expre

term ::= expre “\*” expre

cond ::= expre “>” expre

| expre “<” expre ;

Expr, term, cond에서 더 많은 경우에서 연산이 가능하도록 expre로 바꿨다.

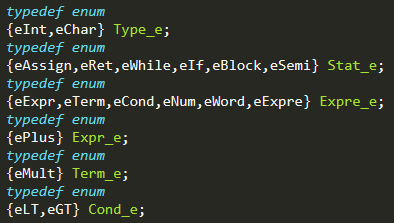
1. If, while, return, assign문 등을 non-terminal로 빼어서 계산하였다. 가독성을 높이기 위해 if, while문의 조건문 자리에 ()를 추가하였다.

2.2. flex part

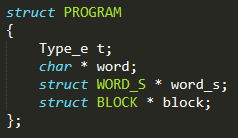
스캐너 part에 해당하는 flex part에 대해서 살펴보도록 하겠다. 크게 살펴볼 것은 .h 파일, .tab.h 파일 그리고 .l 파일이다. .h 파일은 syntax tree에 대한 헤더 파일로 grammar에 일치하도록 tree를 생성할 수 있도록 한다. .tab.h 은 토큰에 대한 각종 선언을 가진 헤더파일로 파서를 통해 생성된다. 마지막으로 .l 파일은 lex 파일로 토큰이 어떤 방식으로 나눌지 결정하고 토큰을 생성하는 파일이다.

2.2.1. AST.h

AST.h는 syntax tree의 구성요소들에 대한 선언을 포함하는 헤더파일이다.



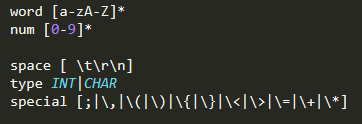
enum으로 구조체 별로 어떻게 사용되는지 구분하여 각각 선언하였다. 예를 들어, stat의 경우 stat 이후에 올 수 있는 것들이 assign, return 등 여러 가지 경우가 있으므로 이 중에 어떤 것이 올지를 구분해야 한다.



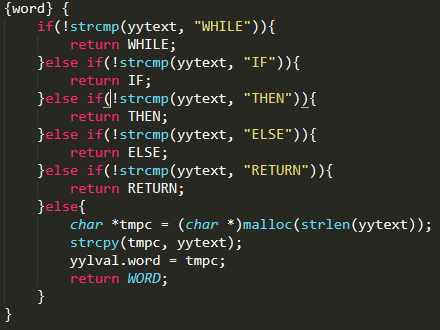
AST.h 내에 선언된 program구조체의 예시이다. 이 구조체들은 syntax tree에서 각각의 노드를 의미한다.

2.2.2. lex file

컴파일러의 lex 파일로 파일 이름은 taeha.l 로 지정하였다. 코드는 다음과 같다.



taeha.l 에서 어떠한 input이 있는지에 대한 정보로서 definition 부분에 해당한다. word는 대소문자 알파벳의 나열 즉 문자열, num은 숫자를 나타낸다. space는 공백들의 집합, type은 해당 input이 INT형인지, CHAR형인지 등의 type을 나타낸다. 마지막으로 special은 ;, +, \* 등의 terminal을 표현한다.



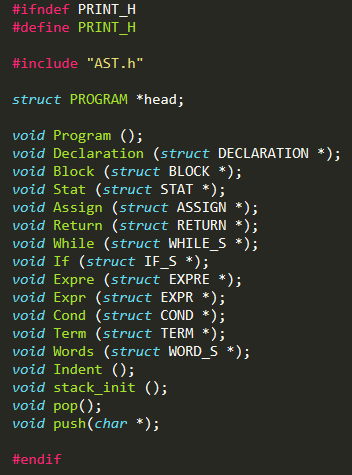
위의 코드는 taeha.l 코드의 일부다. yytext는 lex에서 사용되는 입력 텍스트에 대한 global pointer이다. 우선, WHILE, IF, THEN, ELSE, RETURN의 특수한 경우에는 input이 일치할 경우 각각 따로 토큰으로 생성된다. 그 이외에는 따로 처리해야 하지 않는 일반적인 문자열로서 WORD라는 구조체 자체를 반환하여 토큰을 생성하도록 한다.

2.3. bison part

파서 part에 해당하는 bison part에 대해서 살펴보도록 하겠다. 크게 parser의 역할을 하는 .y 파일이 있고, 그 결과를 토대로 symbol table과 output을 만드는 print.c 파일이 있다.

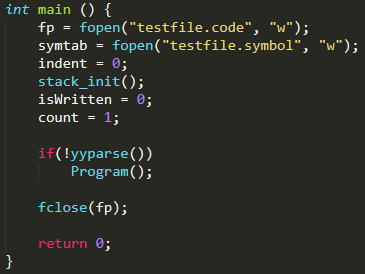
2. 3. 1. print.h

print.c 에서 구현하게 될 함수들과 program이 흐름에 따라 진행되도록 구조체의 포인터를 선언해둔 헤더파일이다.

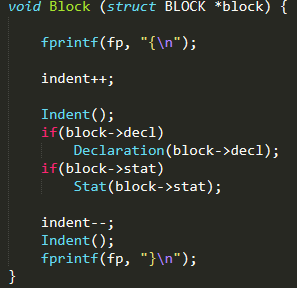


2.3.2. print.c

input과 파서를 통해 나온 output을 코드와 symbol table로 만들어주는 c파일이다. 코드는 다음과 같다.



print.c 의 main 함수이다. yyparse 함수가 있는데 이것은 bison에서 제공하는 내장함수로 구문분석을 시작할 것을 의미한다. yyparse를 통해 program 함수를 실행하여 시작한다.

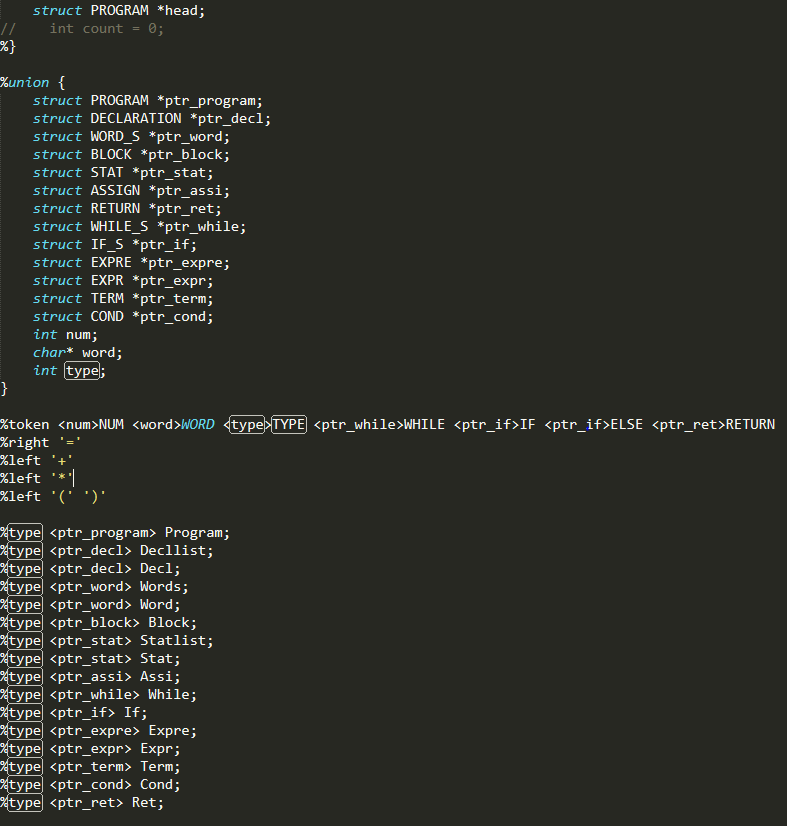


print.c 내에서 구현한 함수 중 하나인 Block 함수이다. 이 함수를 통해 fp가 가리키는 파일에 code를 작성한다. Block을 시작하므로 ‘{‘를 적고 엔터를 한번 한다. 그리고 block 내에서 indent가 하나 늘어나므로 indent를 1 늘려주고, Indent()를 호출하여 “\t”를 통해 들여쓰기가 되도록 한다. block 내에는 decls가 오고 그 이후에 statlist가 올 수 있는 grammar 구조이므로 순서에 맞게 작성한다. 그리고 block을 빠져 나오므로 다시 indent를 1 줄여주고, “}”를 출력하고 함수를 종료한다.

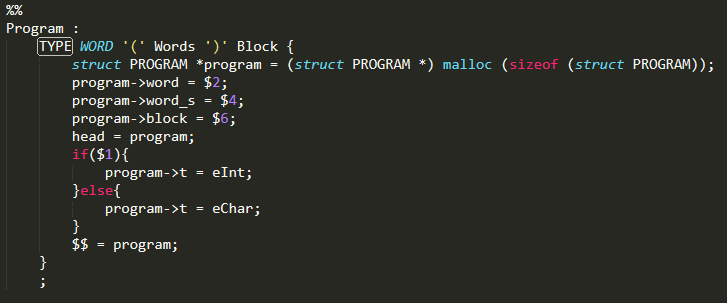
2. 3. 3. yacc file

파서의 역할을 하는 yacc 파일로서 파일 이름은 taeha.y로 지정하였다. 코드는 다음과 같다.

아래의 코드는 Definition 부분으로 yacc 파일에서 사용되는 구조체와 변수들을 선언한다. 그리고 %token 이후로 사용되는 토큰들을 선언하였다. 아래에 right와 left는 operation이 right-most인지, left-most인지 표현한다.



아래의 코드는 taeha.y 코드의 일부로 Program이라는 토큰을 어떻게 처리할지 나타낸 것이다. Program은 또 다시 TYPE WORD ‘(‘ Words ‘)’ Block 이 된다. 여기서 앞에서부터 레지스터의 번호를 1로 매겨 TYPE의 레지스터 번호가 $1, Block의 레지스터 번호가 $7이 된다. Program이라는 토큰이 어떤 다른 토큰들로 이루어져 있는지, 또 이 토큰들을 어떤 방식으로 처리하여 parsing을 할 것인지 결정하게 된다.



3. Test

다음은 input에 대해서 code와 symbol table이 어떻게 작성되는지 보여주는 예시이다.

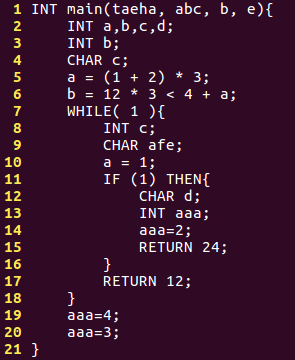


Figure 4 Input

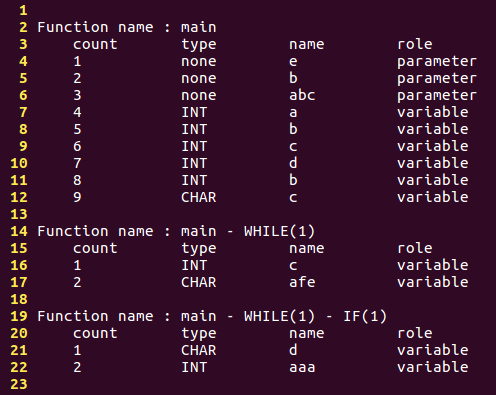


Figure 5 Symbol Table

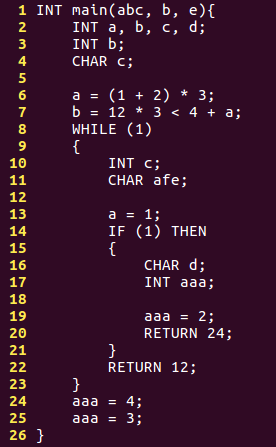


Figure 6 Output Code