Assignment 10: Yacc

Date: 2024. 11. 19
Student ID: 201924451
Name: 김태훈

1. Bison의 인터페이스와 동작 구조[1][2]

Bison은 주어진 문법을 바탕으로 구문분석기를 만들어준다. Bison의 인터페이스는 다음과 같다.

```
Yacc source program(mygram.y) --> yacc(or bison) --> y.tab.c, y.tab.h
y.tab.c --> C compiler --> a.out
input stream --> a.out --> sequence of parsing actions
```

Yacc source program은 다음과 같은 구조로 구성되어있다.

```
declarations
%%
translation rules
%%
auxiliary procedures
```

declaration은 토큰과 우선순위 및 좌/우측 결합을 지정한다. %{ %} 사이에 C 변수, 함수 프로토타입 선언을 할 수 있다.

declaration의 예는 다음과 같다:

```
%{
    #include <stdio.h>
    int yylex();
    int yyerror(char *s);
    int a;
%}
%token NUMBER
%left '+'
%right '*'
%nonassoc '<'
```

%token NUMBER는 NUMBER가 토큰임을 나타낸다.

%left '+' 는 '+' 연산자가 좌측결합임을 나타낸다.

%nonassoc '<'은 '<' 가 동일한 우선순위의 연산자와 결합하지 않음을 의미한다. 즉 a<b<c는 에러를 발생시킨다.

+, *, < 순서대로 선언되었으므로, 연산자 우선순위는 '<' 가 가장 높고 '+' 가 가장 낮다.

translation rule에는 production rule과 해당 rule이 구문분석에 사용되었을 때 실행할 C코드로 구성되어있다. 예를 들면 아래와 같다.

```
%%
Exp : Exp '+' NUM {puts("E-> E+N");}
    | NUM {puts("E->N");}
    ;
%%
```

auxiliary procedures 에는 사용자 지정 함수가 정의되어있다. 예를 들면 아래와 같다.

```
int main() {yyparse(); return 0;}
```

Bison은 *.y 파일을 입력받아 shift-reduce 구문분석기를 생성하며 보통 lex와 결합하여 사용한다. shift-reduce 구문분석기는 다음과 같은 동작을 한다.

(1) 토큰을 수신한다.

(2) 토큰 수신 후 4가지 동작

1) shift: 현재 토큰을 push한다.

2) reduce: translation rule을 사용하여 스택에 있는 토큰 일부를 논터미널로 바꾼다.

3) Accept: 입력된 string이 production rule을 만족한다. 즉 마지막에 stack에 start symbol만 존재하고 shift해야할 토큰이 남아있지 않다.

4) Error: 입력된 string은 production rule로부터 생성될 수 없다.

3) 충돌 에러

shift-reduce 에러와 reduce-reduce 에러가 있을 때 발생한다

shift-reduce 에러는 shift와 reduce 둘다 가능할 때 발생하며, bison은 shift를 한다. 아래와 같은 경우 'if E then S else S' string에서 'if E then S' 를 reduce 할지, else S까지 shift해서 reduce할지 모호해진다.

```
S: if E then S
  | if E then S else S
  ;
E: true
  | false
```

reduce-reduce 에러는 여러 개의 reduce가 가능해질 때를 말하며, 아래와 같은 경우 NUM 토큰을 Term과으로 reduce할 수도 있고, Param으로도 reduce할 수 있어 모호해진다.

```
Term : NUM {puts("T--> n");}
     ;
Param : 'a' {puts("P->a");}
          | NUM {puts("P->n");}
```

2. Bison 입력 코드와 설명

```
%{
    #include <stdio.h>
    int yylex();
    int yyerror(char *s);
%}
%token NUMBER
%%
Exp :Exp '+'Term {printf("E -> E + T\n");}
    |Exp '-'Term {printf("E -> E - T\n");}
    |Term {printf("E -> T\n");}
    ;
Term :Term '*'Factor {printf("T -> T * F\n");}
    |Term '/'Factor {printf("T -> T / F\n");}
    |Factor {printf("T -> F\n");}
    ;
Factor :NUMBER {printf("F -> n\n");}
%%
int main(){
    yyparse();
}
int yyerror(char *s){
    printf("%s\n",s);
}
```

declaration 부분에는 flex 파일로부터 정의될 yylex 함수의 프로토타입을 선언한다. 그리고 NUMBER는 token 임을 선언해준다. 이는 어휘분석기(lex)와 공유된다.

translation rule 부분에는 production rule과 해당 rule 이 reduce시 사용되었을 때 실행할 코드를 지정한다. 코드는 해당 rule이 사용되었음을 출력하는 코드이다. 해당 production rule은 사칙연산을 정의하며, '*', '/' 연산자가 '+', '-' 연산자보다 우선순위를 높게 지정한다.

숫자와 +,-,*,/를 감지하는 어휘분석기와 결합하여 사용하면 '10+11*12' 입력에 다음과 같은 출력이 나타난다.

```
F -> n
T -> F
E -> T
F -> n
T -> F
F -> n
T -> T * F
E -> E + T
```

이를 통해 입력된 문자열이 어떤 과정으로 구문 분석되고 있는지 확인할 수 있다.

3. 리스트를 포함한 데이터를 표현하기 위한 Yacc 문법

```
%{
    #include <stdio.h>
    int yylex();
    int yyerror(char *s);
%}
%token NUMBER
%%
Exp :Exp '+'Term {printf("E -> E + T\n");}
    |Exp '-'Term {printf("E -> E - T\n");}
    |Term {printf("E -> T\n");}
    ;
Term :Term '*'Factor {printf("T -> T * F\n");}
    |Term '/'Factor {printf("T -> T / F\n");}
    |Factor {printf("T -> F\n");}
    ;
Factor :NUMBER {printf("F -> n\n");}
        |'('List ')'  {printf("F -> (L)\n");}
        |'(')'{printf("F -> NIL\n");}
        ;
List:Factor {printf("L -> F\n");}
        |List '.'Factor {printf("L -> L . F\n");}
        ;
%%
int main(){
    yyparse();
}
int yyerror(char *s){
    printf("%s\n",s);
}
```

리스트도 표현할 수 있게 하기 위해 Factor 논터미널이 NUMBER 토큰뿐만 아니라 괄호로 묶인 List나 빈 리스트로 바뀔 수 있게 하였다. List 논터미널은 Factor 논터미널이나 List . Factor로 바뀔 수 있다. 이는 아래와 같은 문장을 표현할 수 있게 한다(좌재귀는 shift-reduce에서는 문제되지 않는다).

(3 . 4 . 5)
(3 . (4 . 5))

(3 . 4 . 5)는 아래와 같이 reduce 된다.

```
F -> n
L -> F
F -> n
L -> L . F
F -> n
L -> L . F
F -> (L)
T -> F
E -> T
```

(3 . (4 . 5))는 아래와 같이 reduce 된다.

```
F -> n
L -> F
F -> n
L -> F
F -> n
L -> L . F
F -> (L)
L -> L . F
F -> (L)
T -> F
E -> T
```

참고) 리스트를 지원하기 위한 lex 파일

```
%{
    #include <stdlib.h>
    #include "y.tab.h"
%}
%%
[0-9]+  return(NUMBER);
[ \t]   ;
\+  return ('+');
\-  return ('-');
\*  return ('*');
\/  return ('/');
\(  return ('(');
\)  return (')');
\.  return ('.');
\n  return (0);
.   {printf("'%c': illegal character\n"),yytext[0]; exit(-1);}
%%
int yywrap()    {return 1;}
```

[1] 박두순 저. “컴파일러의 이해”. 한빛아카데미. 2020

[2] Compiler Lab 09: Using Bison with Flex https://youtu.be/pDXGDOexfpc