|  |  |  |
| --- | --- | --- |
| **Submission** | **2018. 12. 9** | |
| **Department** | **Computer Science** | |
| **Student Id1** | **20141775** | **박진혁** |
| **Student Id2** | **20146703** | **박지호** |
| **Student Id3** | **20144320** | **김경현** |
| **Student Id4** | **20143592** | **최영재** |



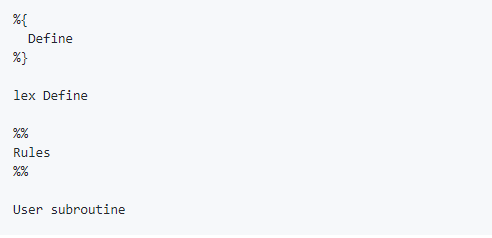
**Pl Project Report**

1. **개발 도구**

Flex : lex의 기능을 개선한 어휘분석기를 생성해주는 소프트웨어

C로 구문 분석 코드를 만들 수 있음

코드를 어휘적으로 토큰화 시킴

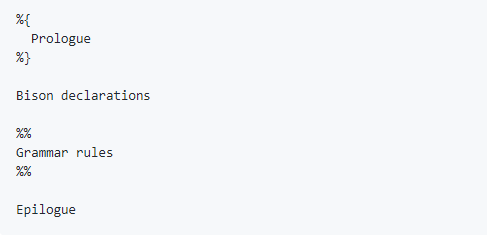


Bison : yacc의 기능을 개선한 GNU 파서를 생성해주는 파서 생성기

LALR 방식으로 작성된 문법을 처리하고 해석해서 C 코드로 만들어 줌

토큰들을 토대로 문법적으로 맞는지 검사

실제 동작하는 알고리즘이나 함수들을 여기에 넣었다.



1. **구현 방식**

* **Lex**

**Lex Define**

* + 숫자와 캐릭터 구분

Letter [ a - zA - z ]

Digit [ 0 - 9 ]

**Rules**

* + Id, Integer, Float 구분

{ Letter } ( { Letter } | { Digit } ) \* -------🡪 Id

{ Digit } + --------------------------------🡪 Integer

{ Digit } + \. { Digit } + -----------------🡪 Float

* + ReservedWord, Operator, Delimeter 구분

“mainprog” ---🡪 return MAINPROG;

“+” ------------🡪 return PLUS;

* + Whitespace 구분

[\n] ----🡪 <newline>

[ \t] ----🡪 <tab> <space>

“#”.\*\n -🡪 Comment

\*\* 구분만하고 토큰화 하지는 않습니다.

\*\* <newline> 부분에 yylineno++를 넣어두어 코드 부분이 몇 번째 줄인지 확인합니다

* **Yacc**

**Prologue**

* + 사용할 변수들이나 함수들을 선언해 놓았습니다.
  + Parse tree를 만들어서 사용하기 위해 struct ast(abstract tree)를 만들었습니다.
  + Symbol table를 만들어서 사용하기 위해 struct symbol, struct symasgn, struct symlist, struct symref를 만들었습니다. 그리고 해당하는 symbol을 찾기 위해서 struct symbol\* lookup을 만들어서 사용하였습니다.

**Bison Declarations**

* + %start program

Non-terminal인 program을 시작으로 문법적으로 맞는 지 검사합니다.

* + %token

lex에서 tokenize된 token들을 lex에서 지정해주었다.

ReservedWord, operation , int, float, 콤마등 token등을 선언해주었다.

* + %type

Grammar에서 사용하는 nonterminal symbol의 type들을 정의해준다. 각 nonterminal들의 타입은 union에서 선언한 타입으로 지정해준다

* + %union

Union은 Grammar 에서 사용할 type들을 나타낸다 이에 맞게 symbol들을 type들을 가지며 정의된 union들을 다음과 같다

Struct ast \* a,

struct symbol \* s,

struct symlist \* sl,

int fn

char c,

double dnum

char \* str

str은 스트링을 나타내고 c는 문자, 그리고 트리를 만들기 위한 ast 구조체 포인터, symbol, symbol list pointer를 선언해주었다. 여기서 value를 dnum으로 선언한 이유는 float로 하였을 때 소수점 자리 값이 정확하지 않아서 정화성을 위해서 double을 이용하였다. 그리고 계산시에 충돌이 나는 것을 방지하기 위해서 int가 아닌 double을 사용하였다. Fn은 어떤 기능을 할지 정해주는 function의 type이다.

**Grammar Rules**

* + Int형, float형 변수 선언

Symbol Table을 만들어서 변수들을 저장하였다.

변수를 새로 선언할 때 lookup을 사용하여 Symbol table에 같은 변수명을 가진 것이 있는지 탐색하고 있을 시에 에러를 발생시키게 하였습니다.

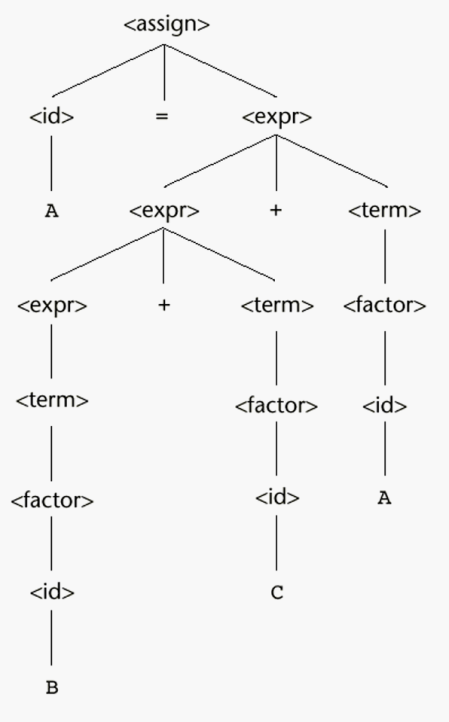
* + 변수 선언

Lookup()함수에서 symbol table에 선언하려는 변수가 있는지 확인하고 변수가 존재 하지 않다면 변수를 선언 할 수 있다. Symbol 구조체에 변수에 관한 정보를 넣어준다.

* + 타입 결정

Symbol 구조체에서 type의 값을 int형은 0, double형은 1 값을 갖도록 하여 구분하였다.

* + 그 이외 print, if, while, 연산

모두 parse tree에 넣은 후 구현한 eval 함수를 사용하여 동작하도록 하였습니다.

문법에 맞게 분기하여 tree를 만드는 함수를 이용하여 tree 구조를 만들어주고 만들어진 tree를 eval() 함수를 이용하여 맨 위의 문법 즉, program시작 부분에서 parse tree의 포인터를 따라가서 tree의 function에 맞게 각각 실행되도록 하였다.

1. **Suggested features**

Main에서 변수를 선언부에서 자유롭게 선언 할 수 있고 값의 할당도 잘 된다.

print문에서는 변수의 출력, 상수의 출력, 변수들의 연산한 결과의 출력(condition)이 모두 잘 실행이 된다.

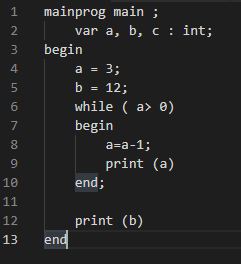
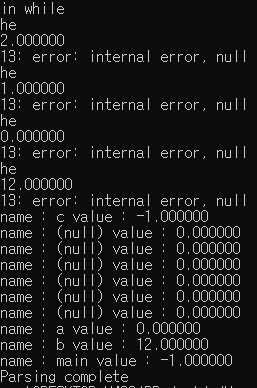
if문도 if문의 condition을 판단하고 조건의 true false, 유무에 맞게 잘 분기하여 실행이 된다.

while문은 parse tree를 만들어서 조건에 만족하는 동안 while문 안에서 statement들을 실행하게 하였다. if문과 while문 모두 newflow 함수를 이용하여 왼쪽 parse tree와 오른쪽 parse tree를 pointer를 생성하여 다시 돌아가서 실행 할 수 있도록 하였다.

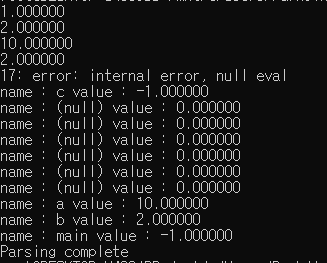
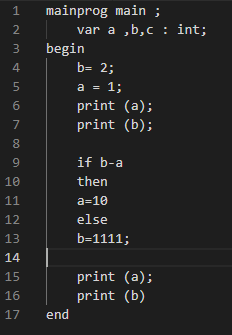
1. **Limit**

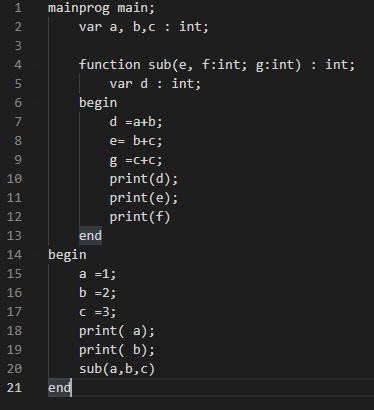
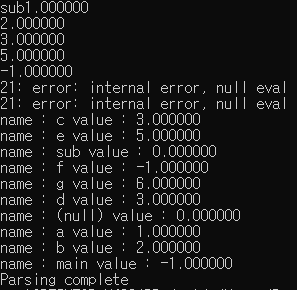
처음에는 하나의 구조체를 선언하여 그 구조체를 가지고 변수를 선언하고 배열을 선언하여 출력하는 부분을 하였다. 하지만 이 구조체 하나만 가지고는 if문은 flag를 이용하여 분기가 가능하지만 while문 같은 경우는 이와 같은 구조로는 while문을 실행할 수 없다는 것을 알게 되었다. Parse tree를 만들어서 statement들을 tree로 만들어 놓고 연결된 tree들을 좇아가면서 while문을 실행 할 수 있도록 바꾸었다. 위와 같은 과정에서 시간을 많이 허비해서 변수의 scoping rule를 구현하지 못하였고 배열의 선언까지는 구현 하였지만 할당까지는 하지 못하였다. 처음 배우고 해보는 언어였기에 체득하는데 시간이 오래 걸려서 완성도를 높이지 못한부분이 아쉽다.

1. **Sample programs for testing**

< Test\_while.txt > ----------------🡪 < cmd 결과 >

< Test\_if.txt > ----------------🡪 < cmd 결과 >



< Test\_sub.txt > -------------------🡪 < cmd 결과 >