1. **Calculator(수식계산기) program using recursive-descent parsing techniques of slide:24-25**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <math.h>  **typedef** **struct** \_params { // 토큰 구조체  **char** cur\_char; // 현재 인덱스의 문자  **char** \*m\_input; // 공백 제거 된 수식  **int** cur\_idx; // 현재 인덱스  **int** expect\_parens; // 괄호 카운트  } \_params;  **enum** token{ // 토큰 종류  NUMBER, // 숫자  LP, // (  RP, // )  PLUS, // +  MINUS, // -  MULTIPLY, // \*  DIVIDE, // /  POW, // ^  UNKNOWN // ?  };  **enum** token \_symbol(**char** c) { // 토큰 구분 함수  **switch** (c) {  **case** '0':  **case** '1':  **case** '2':  **case** '3':  **case** '4':  **case** '5':  **case** '6':  **case** '7':  **case** '8':  **case** '9': **return** NUMBER;  **case** '(': **return** LP;  **case** ')': **return** RP;  **case** '+': **return** PLUS;  **case** '-': **return** MINUS;  **case** '\*': **return** MULTIPLY;  **case** '/': **return** DIVIDE;  **case** '^': **return** POW;  **default**: **return** UNKNOWN;  }  };  **char** input[1024] = {0}; // 입력 된 수식  **char** output[1024] = {0}; // 공백 제거 된 수식  **double** \_expr(\_params \*ptr);  **double** \_term(\_params \*ptr);  **double** \_factor(\_params \*ptr);  **enum** token \_peek(\_params \*ptr);  **void** get\_next\_token(\_params \*ptr);  **double** \_exponent(\_params \*ptr);  **double** \_number(\_params \*ptr);  **void** \_error(**char** \*str);  **void** remove\_space(**void**);  **int** main(**void**) {  printf("Input expression: "); // 수식 입력  fgets(input, **sizeof**(input), stdin);  remove\_space(); // 공백 제거  \_params params = { output[0], output, 0, 0 }; // 파라미터 생성  printf("%.f\n", \_expr(&params)); // 출력  }  **double** \_expr(\_params \*ptr) { // 재귀 하향 파서  **double** term = \_term(ptr); // 우선순위(\*, /, ^) 연산  **while**(1){  **if**(\_peek(ptr) == PLUS) { // + 경우  get\_next\_token(ptr); // 토큰 이동  term = term + \_term(ptr); // 결과 갱신  } **else** **if**(\_peek(ptr) == MINUS) { // - 경우  get\_next\_token(ptr); // 토큰 이동  term = term - \_term(ptr); // 결과 갱신  } **else** **return** term; // 결과 반환  }  }  **double** \_term(\_params \*ptr) {  **double** exponent = \_exponent(ptr); // 우선순위(^) 연산  **while**(1) {  **if**(\_peek(ptr) == MULTIPLY) { // \* 경우  get\_next\_token(ptr); // 토큰 이동  exponent = exponent \* \_exponent(ptr); // 결과 갱신  } **else** **if**(\_peek(ptr) == DIVIDE) { // ‘/’ 경우  get\_next\_token(ptr); // 토큰 이동  exponent = exponent / \_exponent(ptr); // 결과 갱신  } **else** **return** exponent; // 결과 반환  }  }  **double** \_factor(\_params \*ptr) {  **int** sign = 1; // 부호 초기화  **double** expression = 0; // 결과  **if** (\_peek(ptr) == MINUS) { // - 경우  get\_next\_token(ptr); // 토큰 이동  sign = -1; // 부호 음수 전환  } **else** sign = 1; // 부호 양수 유지  **if** (\_peek(ptr) == NUMBER) { // 숫자 경우  **return** sign \* \_number(ptr); // 부호에 따른 결과 반환  } **else** **if** (\_peek(ptr) == LP) { // ( 경우  ptr->expect\_parens = 1; // 괄호 카운트 증가  get\_next\_token(ptr); // 토큰 이동  expression = \_expr(ptr); // 내부 수식 연산  **if** (\_peek(ptr) != RP) // ) 가 없는 경우  \_error("Syntax: Expected closed parenthesis");  get\_next\_token(ptr); // 토큰 이동  ptr->expect\_parens = 0; // 괄호 카운트 초기화  **return** sign \* expression; // 결과 반환  } **else** { // 이 외의 경우  \_error("Syntax");  **return** 0;  }  }  **enum** token \_peek(\_params \*ptr) { // 현재 인덱스의 토큰 문자 형식 반환  **return** \_symbol(ptr->m\_input[ptr->cur\_idx]);  }  **void** get\_next\_token(\_params \*ptr) { // 다음 문자 이동  ptr->cur\_idx = ptr->cur\_idx + 1; // 인덱스 증가  ptr->cur\_char = ptr->m\_input[ptr->cur\_idx]; // 문자 갱신  }  **double** \_exponent(\_params \*ptr) { // 지수 연산  **double** factor = \_factor(ptr);  **while** (1) {  **if** (\_peek(ptr) == POW) { // ^ 경우  get\_next\_token(ptr); // 토큰 이동  factor = pow(factor, \_factor(ptr)); // 결과 갱신  } **else** **return** factor; // 결과 반환  }  }  **double** \_number(\_params \*ptr) { // 숫자 판별  **double** result = 0; // 최종 숫자 값  **double** fractional = 0; // 분수 확인  **int** length = 0;  **while**(\_peek(ptr) == NUMBER) { // 숫자 문자 경우  result = result \* 10 + (ptr->cur\_char - '0'); // 숫자 갱신  get\_next\_token(ptr); // 토큰 이동  }  **if** (fractional > 0) // 분수일 경우  result = result + (fractional/pow(10, length)); // 숫자 갱신  **if** (\_peek(ptr) == LP) // ( 경우  \_error("Syntax: Unexpected open parenthesis");  **else** **if** (\_peek(ptr) == RP && !ptr->expect\_parens) // ) 이고 괄호 카운트가 0이 아닌 경우  \_error("Syntax: Unexpected closed parenthesis");  **return** result; // 결과 반환  }  **void** \_error(**char** \*str) { // 오류 구문 출력  fprintf(stderr, "ERROR: %s\n", str);  exit(1);  }  **void** remove\_space(**void**) { // 공백 제거  **int** i, j = 0;  **for** (i = 0; i < strlen(input); i++) {  **if** (input[i]==' ') **continue**;  output[j++]=input[i];  }  output[j]=0; // 끝에 NULL값 추가  } |

[코드 1] c\_recursiveparse.c

* 실행 결과

측정기이(가) 표시된 사진

자동 생성된 설명

측정기이(가) 표시된 사진

자동 생성된 설명

1. **Problem(연습문제) set of chapter-3(p.181-184)**
2. **Write EBNF descriptions for the following.**
3. A Java class definition header statement.

<method\_head> → [public][(abstract|final)]class <id>[extends <id>][implements<id>{,<id>}]

1. A C/C++/Java switch statement

<switch> → <switch>(<expr>){case <literal> : <stmt\_list>}[default : <stmt\_list>]}

1. A C/C++/Java for-loop statement

<for> → for([[<type>]<id> = <expr>{,[<type>]<id> = <expr>}]; [<expr>]; [<expr>{,<expr>}]){stmt\_list>}

1. **Rewrite the BNF of Example 3.4 to give + precedence over \* and force + to be right associative.**

<assign> → <id> = <expr>

<id> →A | B | C

<expr> → <expr > \* <term> | <term>

<term> → <term> + <factor> | <factor>

<factor> → (<expr>) | <id>

1. **Rewrite the BNF of Example 3.4 to add the ++ and – unary operators of Java.**

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term> | <term>

<term> → <term> \* <factor> | <factor>

<factor> → (expr) | <id> | <id>++ | <id>--

**6. Using the grammar in Example 3.2, show a parse tree and a left most derivation for each of the following statements.**

1. 텍스트이(가) 표시된 사진

   자동 생성된 설명A = A \* (B + (C \* A))

<assign> → <id> = <expr>

→ A = <expr>

→ A = <id> \* <expr>

→ A = A \* (<expr>)

→ A = A \* (<id> + <expr>)

→ A = A \* (B + <expr>)

→ A = A \* (B + (<expr>))

→ A = A \* (B + (<id> \* <expr>))

→ A = A \* (B + (C \* <expr>))

텍스트이(가) 표시된 사진

자동 생성된 설명 → A = A \* (B + (C \* <id>))

→ A = A \* (B + (C \* A))

1. B = C \* (A \* C + B)

<assign> → <id> = <expr>

→ B = <expr>

→ B = <id> \* <expr>

→ B = C \* <expr>

→ B = C \* (<expr>)

→ B = C \* (<id> \* <expr>)

→ B = C \* (A \* <expr>)

→ B = C \* (A \* <id> + <expr>)

→ B = C \* (A \* C + <expr>)

→ B = C \* (A \* C + <id>)

→ B = C \* (A \* C + B)

1. 텍스트이(가) 표시된 사진

   자동 생성된 설명A = A \* (B + ( C ))

<assign> → <id> = <expr>

→ A = <expr>

→ A = <id> \* <expr>

→ A = A \* <expr>

→ A = A \* (<expr>)

→ A = A \* (<id> + <expr>)

→ A = A \* (B + <expr>)

→ A = A \* (B + (<expr>)

→ A = A \* (B + (<id>))

→ A = A \* (B + ( C ))

**8. Prove that the following grammar is ambiguous.**

다음의 두가지 경우를 통하여 이 문법이 모호함을 증명 할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**11. Consider the following grammar.**

1. b a b b : 가능

<s> → <A> a <B> b

→ b a <B> b

→ b a <B> b

→ b a b b

1. b b b a b b : 가능

<s> → <A> a <B> b

→ <A> b a <B> b

→ <A> b b a <B> b

→ b b b a <B> b

→ b b b a b b

1. b b a a a a a b c : 불가능, c를 출현하게 할 가능성이 전무함
2. a a a a a a : 불가능, 최소한 하나의 b가 포함되어야 함

**14. Draw parse trees for the sentences a b b and a a b b b, as derived from the grammar of Problem 13.**

<s> → a <s> b | b

텍스트이(가) 표시된 사진

자동 생성된 설명

**17. Convert the following EBNF grammar to BNF.**

S → S b A | A

A → a a | a b A

**19. Write an attribute grammar whose BNF basis is that of Example 3.6 in Section 3.4.5 but whose language rules are as follows: Data types cannot be mixed in expression, but assignment statements need not have the same types on both sides of the assignment operator.**

1. Syntax rule: <assign> → <var> = <expr>
2. Syntax rule: <expr> → <var>[2] + <var>[3]

Predicate: <var>[2].actual\_type == <var>[3].actual\_type

1. Syntax rule: <expr> → <var>
2. Syntax rule: <var> → A | B | C

Semantic rule: <var>.actual\_type ← lookup(<var>.string)

**20. Write an attribute grammar whose BNF basis is that of Example 3.2 and whose type rules are the same as for the assignment statement example of Section 3.4.5.**

1. Syntax rule: <assign> → <id> = <expr>

Semantic rule: <expr>.expected\_type ← <id>.actual\_type

Predicate: <expr>[1].actual\_type == <expr>[1].expected\_type

1. Syntax rule: <expr>[1] → <id> + <expr>[2]

Sematic rule: <expr>[1].actual\_type ←

if(<id>.actual\_type = int and <expr>[2].actual\_type = int)

then int

else float

end if

Predicate: <var>[2].actual\_type == <var>[3].actual\_type

1. Syntax rule: <expr>[1] → <id> \* <expr>[2]

Semantic rule: <expr>[1].actual\_type ←

if(<id>.actual\_type = int and <expr>[2].actual\_type = int)

then int

else float

end if

1. Syntax rule: <expr>[1] → (<expr>[2])

Semantic rule: <expr>[1].actual\_type ← <expr>[2].actual\_type

1. Syntax rule: <expr> → <id>

Semantic rule: <expr>.actual\_type ← <id>.actual\_type

1. Syntax rule: <id> → A | B | C

Sematic rule: <id>.actual\_type ← lookup(<id>.string)

1. **Correctness proof(증명):Slide:34**

텍스트이(가) 표시된 사진

자동 생성된 설명

1. x>0, y>0이라는 precondition에서 <1>명령문을 실행하면 u>0, v>0, u=x, v=y, z=0이 됨을 알 수 있다. 따라서 z+u\*v = x\*y이고 0+u\*v = u\*v이므로 condition 2가 참임을 증명하였다.
2. while문에서 u != 0이라고 가정할 때, U가 홀수 일 경우, u는 정수형 타입이므로 2로 나누어도 짝수(u-1)과 같은 몫을 가지므로 u/2 = (u-1)/2가 참이다. 따라서 명령문 <2>, <3>을 수행하면 z+2v+u/2\*2v = z+2v+(u-1\_/2\*2v = z+uv = xy가 되므로 condition 2는 참이다.
3. u가 짝수일 경우, z+u\*v = x\*y가 되고 명령문 <3> 수행 시 z+u/2\*2v = x\*y이고 결과적으로 z+uv = xy가 되므로 condition 2는 참이다.
4. while문에서u=0이라고 가정하면, z+u\*v=x\*y이고 z+0\*v=x\*y이고 최종적으로z=x\*y가 되므로 이 프로그램은 정확하다.