프로그래밍언어

[3장]

Problem Set

과목명	프로그래밍언어
교수명	김철홍
분 반	나반
학 과	컴퓨터학부
학 번	20211799
이 름	유진
제출일	2023.04.01

*/*3장 Problem Set의 다음 문제들을 푼 결과물을 PDF 파일로 제출하세요. (문서편집기로 작업 또는 손으로 연습용지에 문제풀이 후 캡쳐 편집 가능)*/*

(3, 7, 8, 14, 16, 19, 22-(b,c,d) , 23)

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

주어진 문법을 보면, "<expr>"은 "<expr> + <term>" 또는 "<term>"으로 정의되어 있다. 그런데, "<term>"도 "<term> * <factor>" 또는 "<factor>"로 정의되어 있어서 "<expr>"을 해석할 때 어떤 부분을 먼저 계산해야 할지 애매모호한 상황이 발생할 수 있습니다. 따라서 이 문법은 애매모호한 문법이다.

이 문제를 해결하기 위해서는, "<expr>"을 "<term>"이나 "<factor>" 하나로만 구성되도록 정의해야 한다. 하나의 규칙으로만 정의하기 위해서는, "<expr>"과 "<term>"을 합치거나, "<term>"과 "<factor>"를 합치는 방법이 있다.

"<expr>"과 "<term>"을 합치는 방법 선택

```
<assign>  → <id> = <expr>
<id>      → A ¦ B ¦ C
<expr>    → <term> <expr_tail>
<expr_tail> → + <term> <expr_tail> ¦ ε
<term>    → <factor> <term_tail>
<term_tail> → * <factor> <term_tail> ¦ ε
<factor> → ( <expr> ) ¦ <id>
```

여기서 "<expr_tail>"이 "<term>" 다음에 나오는 "+" 연산자와 그 뒤에 나오는 "<expr>"을 표현한다. "+" 연산자와 "<expr>"의 나열이 재귀적으로 반복될 수 있도록 ε(epsilon)도 추가했다. "<term_tail>"도 마찬가지로 "*"

연산자와 "<factor>"의 나열을 표현한다. 이렇게 하면, "<expr>"이나 "<term>"을 계산할 때 어떤 연산을 먼저 수행해야 하는지 명확하게 알 수 있다.

7. Using the grammar in Example 3.4, show a parse tree and a leftmost der- ivation for each of the following statements:

a. A = (A + B) * C

```
assign => id = expr
      => A = expr
      => A = term expr_tail
      => A = factor term_tail expr_tail
      => A = (expr) term_tail expr_tail
      => A = (factor expr_tail) term_tail expr_tail
      => A = ((expr) + factor) term_tail expr_tail
      => A = ((factor + id) + factor) term_tail expr_tail
      => A = ((id + id) + factor) term_tail expr_tail
      => A = ((A + id) + factor) term_tail expr_tail
      => A = ((A + B) + factor) term_tail expr_tail
      => A = ((A + B) * factor) term_tail expr_tail
      => A = ((A + B) * id) term_tail expr_tail
      => A = ((A + B) * C) term_tail expr_tail
      => A = ((A + B) * C) ε expr_tail
      => A = ((A + B) * C)
```

b. A = B + C + A

```
assign => id = expr
      => A = expr
      => A = term expr_tail
      => A = factor term_tail expr_tail
      => A = id term_tail expr_tail
      => A = B term_tail expr_tail
      => A = factor + term term_tail expr_tail
      => A = id + term term_tail expr_tail
      => A = B + term term_tail expr_tail
      => A = B + factor term_tail term_tail expr_tail
      => A = B + id term_tail term_tail expr_tail
```

=> A = B + C term_tail term_tail expr_tail
=> A = B + C term_tail expr_tail
=> A = B + C + term expr_tail
=> A = B + C + factor term_tail expr_tail
=> A = B + C + id term_tail expr_tail
=> A = B + C + A term_tail expr_tail
=> A = B + C + A ε expr_tail
=> A = B + C + A

c. A = A * (B + C)

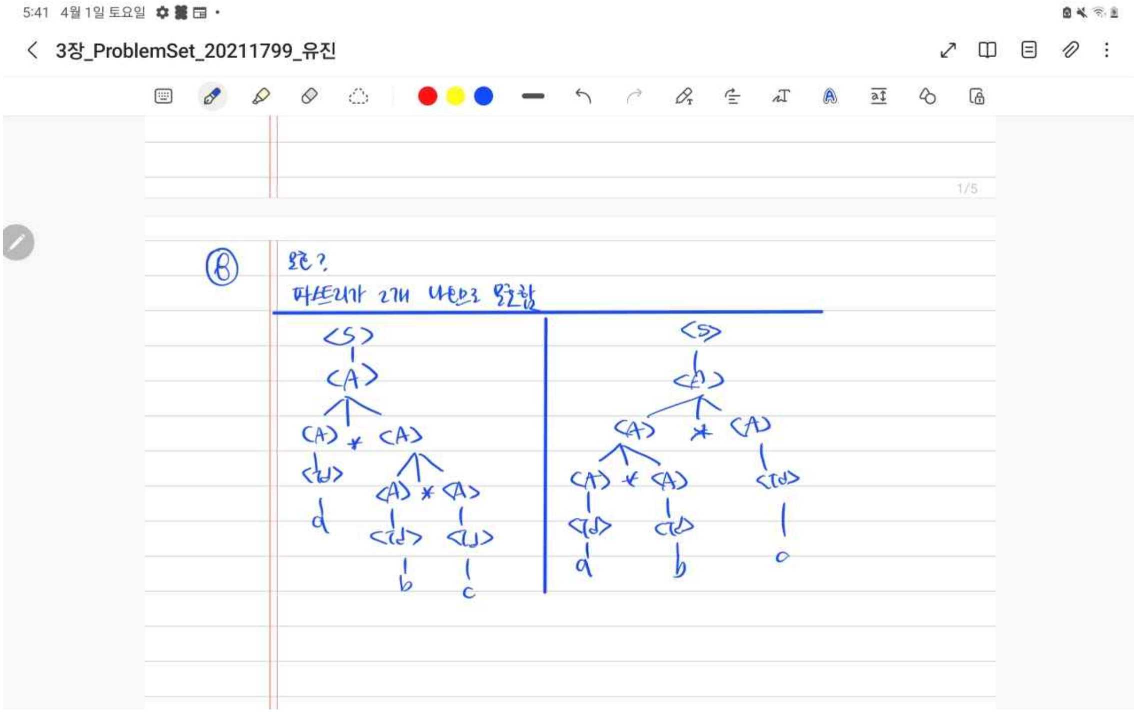
assign -> id = expr -> A = term -> A * factor -> A * (expr)
-> A * (term) -> A * (factor + expr) -> A * (id + expr)
-> A * (B + expr) -> A * (B + term) -> A * (B + factor)
-> A * (B + C)

d. A = B * (C * (A + B)

assign -> id = expr -> A = term -> factor * expr -> B * (expr)
-> B * (term) -> B * (factor * expr) -> B * (factor * term)
-> B * (factor * factor + expr) -> B * (factor * factor + term)
-> B * (factor * factor + factor) -> B * (factor * factor + (expr))
-> B * (factor * factor + (factor + expr)) -> B * (factor * factor + (factor + term))
-> B * (factor * factor + (factor + factor)) -> B * (factor * factor + (factor + (expr)))
-> B * (factor * factor + (factor + (id))) -> B * (factor * factor + (factor + (B)))
-> B * (factor * factor + (factor + (C))) -> B * (factor * factor + (A + B))
-> B * (C * (A + B))

8.Prove that the following grammar is ambiguous:
<S> → <A>
<A> → <A> + <A> ¦ <id>
<id> → a ¦ b ¦ c

모호하다는 것 증명=> 파스트리가 2개 나옴



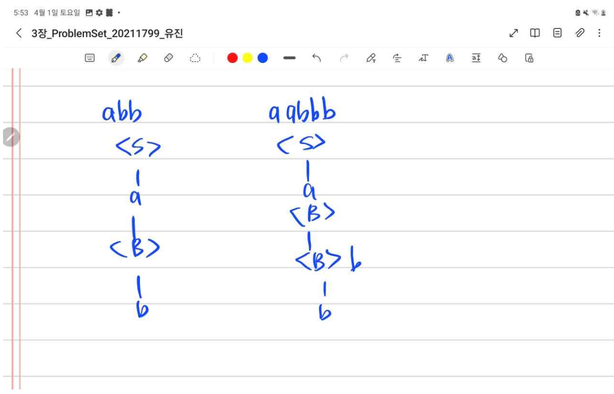
14. Draw parse trees for the sentences abb and aabbb, as derived from the grammar of Problem 13.

문자 a가 n번 반복되고 문자 b가 한번 더 나오는 문자열로 이루어진 언어에 대한 문법을 작성하라. 이 때, n은 0보다 큰 자연수이다. 예를 들어, abb, aaaabbbbb, aaaaaaaaaabbbbbbbbbbb은 해당 언어에 속하지만, a, ab, ba, aaabb는 속하지 않는다.

문법:

<S> → a <B>

<B> → b ¦ <B> b



16. Convert the BNF of Example 3.3 to EBNF

Example 3.3

```
An Ambiguous Grammar for Simple Assignment Statements
<assign> → <id> = <expr>
<id> → A ¦ B ¦ C
<expr> → <expr> + <expr>
¦  <expr> * <expr>
¦ ( <expr>)
¦  <id>
```

```
<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::= <term> { (+ | *) <term> }
<term> ::= (<expr>) | <id>
```

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 expressions, but assignment statements need not have the same types on both sides of the assignment operator.

Example 3.6

```
An Attribute Grammar for Simple Assignment Statements
1.  Syntax rule: <assign> → <var> = <expr>
Semantic rule: <expr>.expected_type ← <var>.actual_type
2.  Syntax rule: <expr> → <var>[2] + <var>[3]
Semantic rule: <expr>.actual_type ←
                                              if (<var>[2].actual_type = int) and
(<var>[3].actual_type = int)
                                              then int
                                         else real
                                         end if
Predicate:        <expr>.actual_type == <expr>.expected_type
3.  Syntax rule:     <expr> → <var>
Semantic rule: <expr>.actual_type ← <var>.actual_type
Predicate:        <expr>.actual_type == <expr>.expected_type
4.  Syntax rule:     <var> → A ¦ B ¦ C
```

```
Semantic rule: <var>.actual_type ← look-up(<var>.string)
The look-up function looks up a given variable name in the symbol table and
returns the variable’s type.
```

Example 3.6의 BNF를 기반으로 하되, 다음과 같은 언어 규칙을 갖는 속성 문법을 작성하십시오: 데이터 타입은 표현식에서 혼합될 수 없지만, 할당문은 할당 연산자 양쪽에 동일한 타입이 필요하지는 않습니다.

```
1. Synax rule: <assign> → <var> = <expr>
Semantic rules:
<expr>.expected_type ← <var>.actual_type
if <var>.actual_type == <expr>.actual_type then
<assign>.actual_type ← <var>.actual_type
else
raise SemanticError("Type mismatch in assignment statement")
end if


2.Syntax rule: <expr> → <var>[2] + <var>[3]
Semantic rules:
<var>[2].expected_type ← <var>[2].actual_type
<var>[3].expected_type ← <var>[3].actual_type
<expr>.expected_type ← <var>[2].actual_type (which is also equal to
<var>[3].actual_type)
if (<var>[2].actual_type == int) and (<var>[3].actual_type == int) then
<expr>.actual_type ← int
else
<expr>.actual_type ← real
end if
if <expr>.actual_type == <expr>.expected_type then
<var>[2].is_used ← true
<var>[3].is_used ← true
else
raise SemanticError("Type mismatch in expression")
end if


3. Syntax rule: <expr> → <var>
Semantic rules:
```

```
<expr>.expected_type ← <var>.actual_type
<expr>.actual_type ← <var>.actual_type
if <expr>.actual_type == <expr>.expected_type then
<var>.is_used ← true
else
raise SemanticError("Type mismatch in expression")
end if


4. Syntax rule: <var> → A | B | C
Semantic rule:
<var>.actual_type ← look-up(<var>.string)
<var>.is_used ← false
```

22 – b,c,d

Write a denotational semantics mapping function for the following statements:

b. Java do-while

```
do {
   <body>
} while (<condition>);
```

```
semDo(body, cond) =
    body; if (cond) semDo(body, cond) else skip endif
```

c. Java Boolean expressions

```
semBool(true) = 1
semBool(false) = 0
semBool(not b) = 1 - semBool(b)
semBool(b1 and b2) = semBool(b1) * semBool(b2)
semBool(b1 or b2) = semBool(b1) + semBool(b2) - semBool(b1 and b2)
```

d. Java for

```
for (<init>; <cond>; <update>) {
```

```
   <body>
}
```

```
semFor(init, cond, update, body) =
    init; if (cond) body; update; semFor(init, cond, update, body) else skip endif
```

23. Compute the weakest precondition for each of the following assignment
statements and postconditions:

각 대입문과 후조건에 대해 가장 약한 사전조건(weakest precondition):

a. a = 2 * (b - 1) - 1 {a > 0}

wp(a > 0, a = 2 * (b - 1) - 1)
= b > 0 ∧ a > 0 ∧ a > 2 * b – 3

b. b = (c + 10) / 3 {b > 6}

wp(b > 6, b = (c + 10) / 3)
= c > 13

c. a = a + 2 * b - 1 {a > 1}

wp(a > 1, a = a + 2 * b - 1)
= a - 2 * b + 1 > 1
= a > 2 * b – 1

d. x = 2 * y + x - 1 {x > 11}

wp(x > 11, x = 2 * y + x - 1)
= y > 5