PUSAN NATIONAL UNIVERSITY

REPORT

프로그래밍언어론 과제7

과 목 명 :	프로그래밍언어론
지도교수 :	
학 과 :	전기컴퓨터공학부 정보컴퓨터공학전공
학 번 :	
이 름 :	장 수현
제 출 일 :	2020년 6월 9일

부산대학교

1. Precedence에 따른 tree 표현 그리기

Give the tree representation for the following, assuming C precedence:

a. - A - B / C * D / E / F + G

b. A * B > C + D / E - F

c. ! A & B > C + D

sol)

우선순위	연산자	설명	결합 법칙(방향)
1	x++ x-- ( ) [ ] . ->(자료형){값}	증가 연산자(뒤, 후위) 감소 연산자(뒤, 후위) 함수 호출 배열 첨자 구조체/공용체 멤버 접근 포인터로 구조체/공용체 멤버 접근 복합 리터럴	→
2	++x --x +x -x ! ~ (자료형) *x &x sizeof	증가 연산자(앞, 전위) 감소 연산자(앞, 전위) 단항 덧셈(양의 부호) 단항 뺄셈(음의 부호) 논리 NOT 비트 NOT 자료형 캐스팅(자료형 변환) 포인터 x 역참조 x의 주소 자료형의 크기	←
3	* / %	곱셈 나눗셈 나머지	→
4	+ -	덧셈 뺄셈	→
5	<< >>	비트를 왼쪽으로 시프트 비트를 오른쪽으로 시프트	→
6	< <= > >=	작음 작거나 같음 큼 크거나 같음	→
7	== !=	같음 다름	→
8	&	비트 AND	→

그림 1 C언어 연산자 우선순위 표 일부.

그림 1에서 문제에서 요구하는 연산자들에 대한 우선순위를 확인할 수 있다. 연산 우선순위가 높을수록 먼저 연산이 되고, 우선순위가 낮을수록 나중에 계산이 된다. 만약 우선순위가 같을 경우, 결합 법칙(방향)에 따라 연산 순위가 결정된다. 이를 고려해 tree representation을 작성하면 다음과 같다.

a.

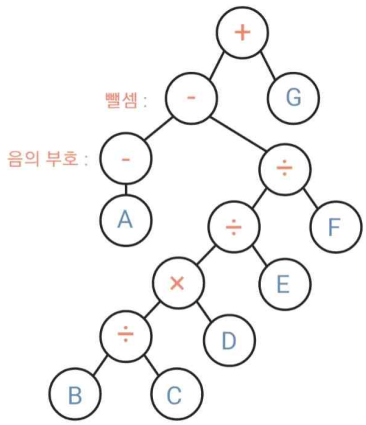

그림 2 '-A-B/C*D/E/F+G'의 tree representation

b.

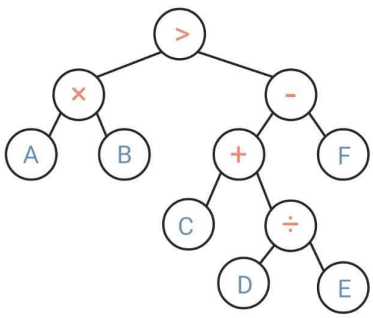

그림 3 'A*B>C+D/E-F'의 tree representation

c.

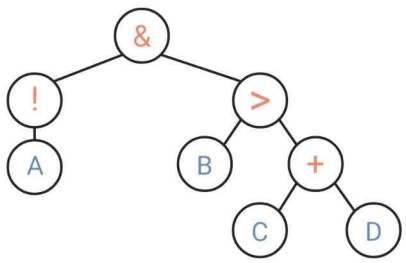

그림 4 '!A&B>C+D'의 tree representation

2. Prime 프로그램 찾기

Give all prime programs consisting of five nodes

sol)

2.0 Prime Programs

프로그램은 크게 prime program과 composite program으로 분류되고, 이러한 개념은 프로그램 전체에 대한 제어구조를 어떻게 만들고 관리해야 하는지에 대한 일관적 이론 수립을 위해 제시되었다. 우선, proper program이란, 입구와 출구가 각각 하나씩만 존재하고, 입구로부터 모든 노드까지의 경로와 모든 노드로부터 출구까지의 경로가 있는 제어 흐름을 가진 프로그램을 말한다. 그리고 두 개 이상의 노드로 이루어진 proper subprogram을 포함하고 있지 않은 proper program을 Prime Program이라 한다. 즉, 2개의 edge에 대해 cut을 했을 때, 나눠진 두 개의 프로그램이 proper program이 되면 안 된다는 것이다. proper program 중 prime program이 아닌 것은 composite program이다. 이 개념을 가지고 우리가 추구해야 할 것은 최종적으로 proper program을 만들어 복잡한 스파게티 프로그래밍이 아닌 구조적 프로그래밍을 하는 것이다.

2.1 all prime programs consisting of five nodes

모든 프로그램은 function node, decision node, join node로만 구성되는 흐름도로 나타낼 수 있다. 흐름도 내부의 세 노드 간의 구조와 흐름에 따라 proper program, prime program, composite program으로 구분할 수 있다. 이때, function node가 일렬로 늘어선 경우는 하나의 prime으로 취급한다.

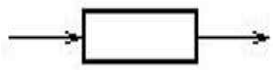

그림 5 function node

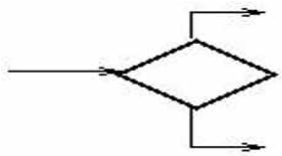

그림 6 decision node

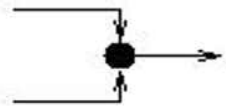

그림 7 join node

5개의 노드를 갖는 prime program들은 그림 8의 4개의 노드를 갖는 prime program에서 proper subprogram을 포함하지 않도록 유지하면서, 한 개의 노드를 추가하는 것으로 생성할 수 있다. 그림 8의 (l) ~ (q) 6개의 프로그램의 5개 edge에 각각 function node를 추가해서 만들어진 그림 9 ~ 그림 14와 function node가 5개 일렬로 늘어선 그림 15까지 총 31개의 program은 모두 5개의 노드를 갖는 prime program이다. 이때, (j)와 (k) 프로그램을 제외한 이유는 어떤 edge에라도 function node를 추가했을 경우, 2개 이상의 노드를 갖는 proper program을 포함하게 되기 때문이다.

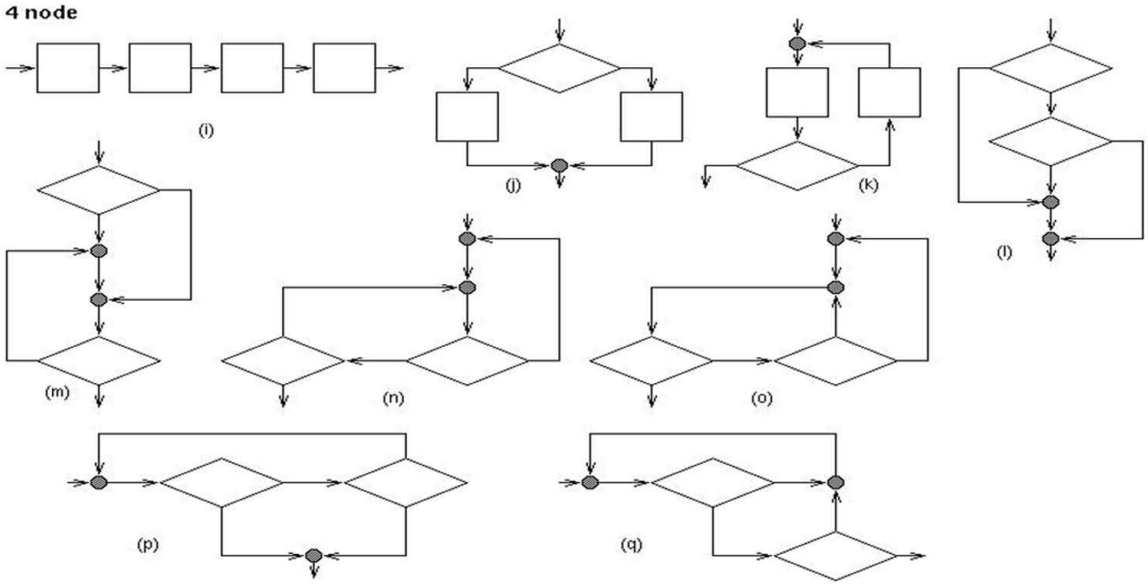

그림 8 4개의 노드를 가지는 prime programs

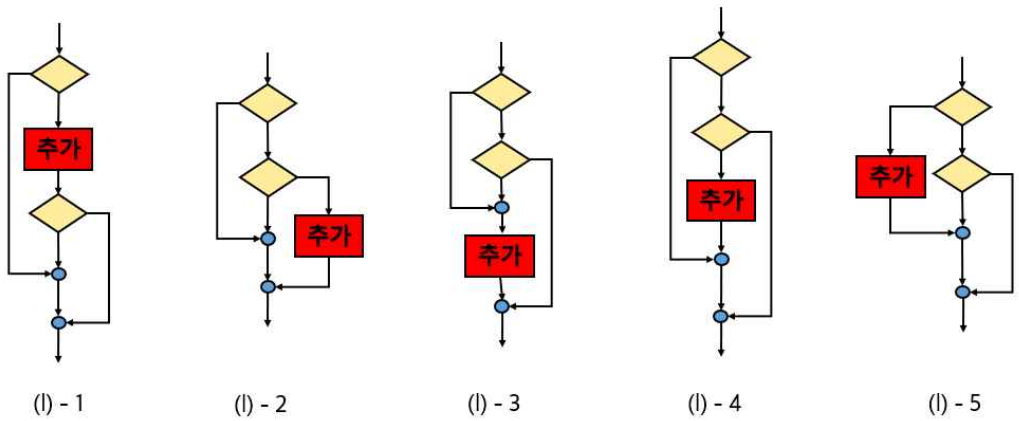

그림 9 그림 8의 (l)에 function node를 추가하여 5개의 노드를 가지는 prime programs

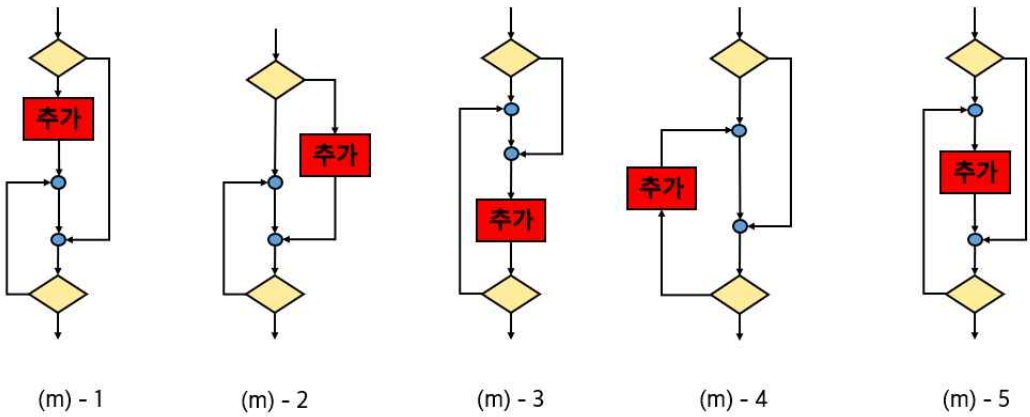

그림 10 그림 8의 (m)에 function node를 추가하여 5개의 노드를 가지는 prime programs

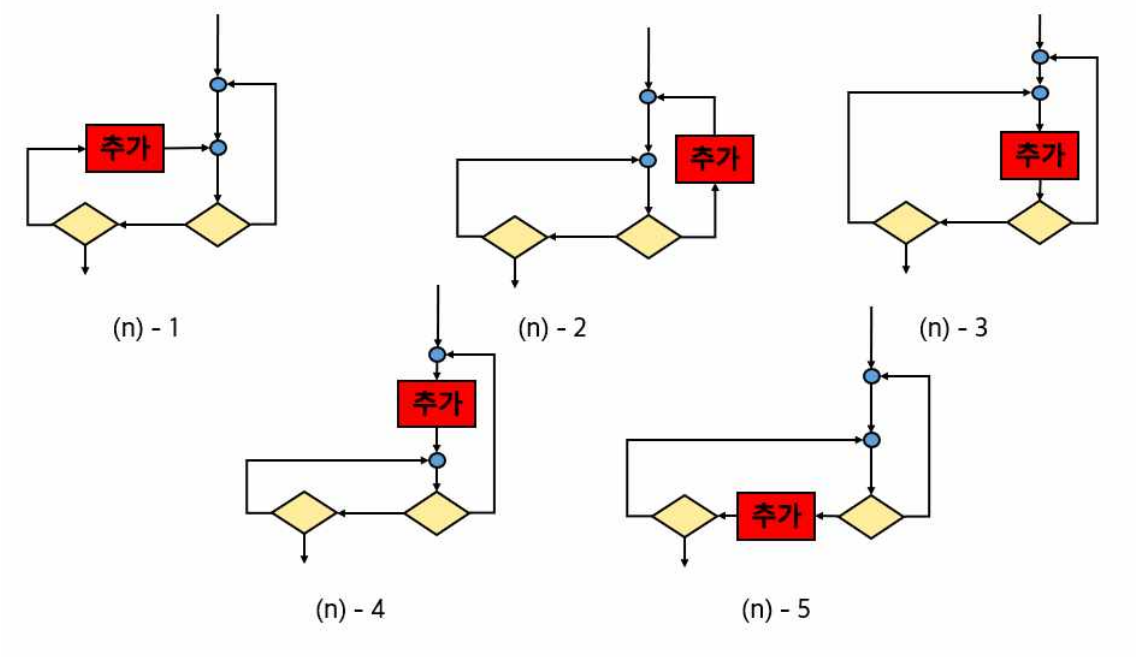

그림 11 그림 8의 (n)에 function node를 추가하여 5개의 노드를 가지는 prime programs

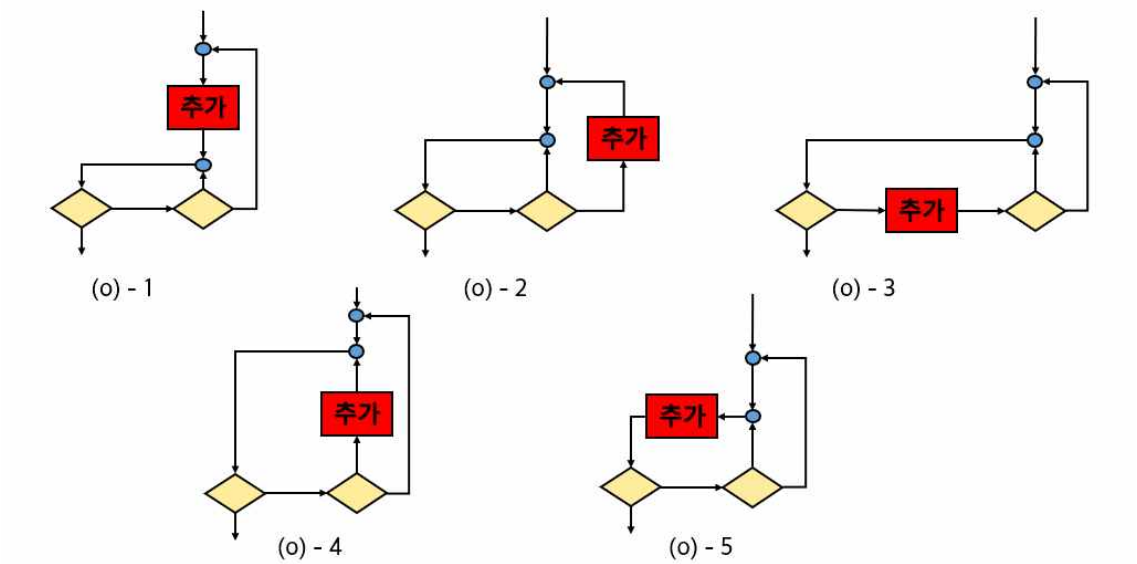

그림 12 그림 8의 (0)에 function node를 추가하여 5개의 노드를 가지는 prime programs

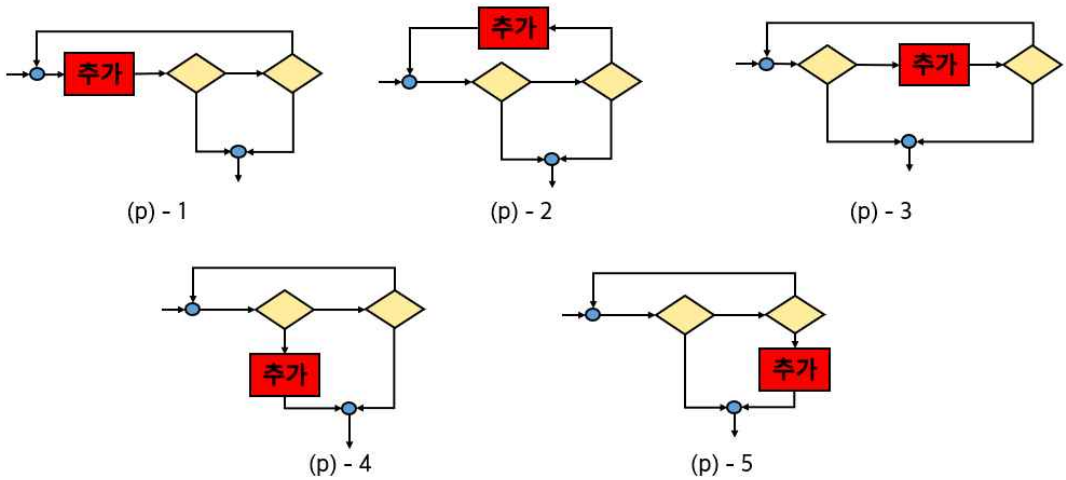

그림 13 그림 8의 (p)에 function node를 추가하여 5개의 노드를 가지는 prime programs

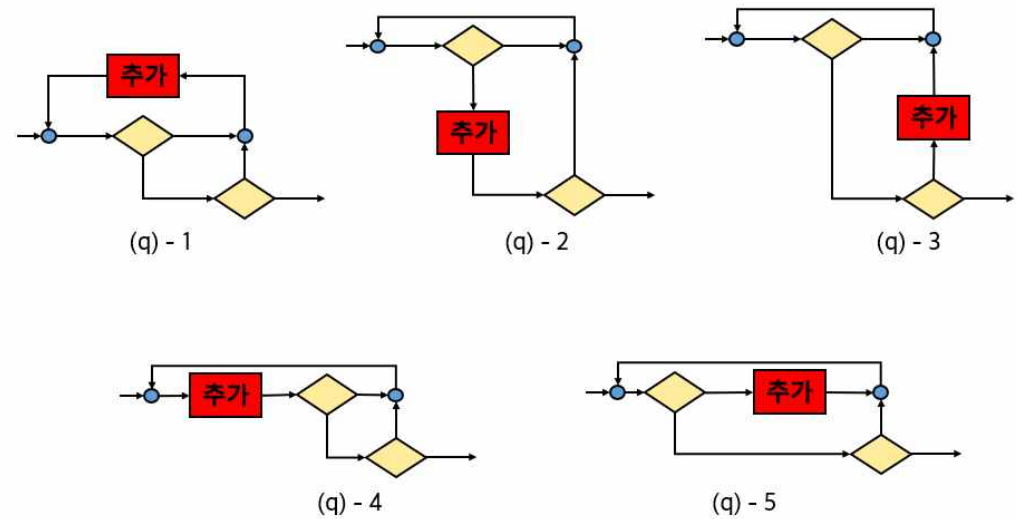

그림 14 그림 8의 (q)에 function node를 추가하여 5개의 노드를 가지는 prime programs

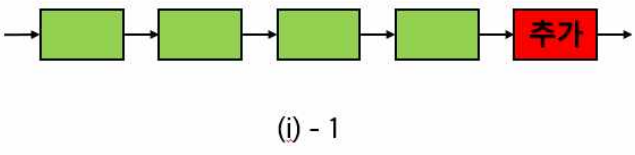

그림 15 그림 8의 (i)에 function node를 추가하여 5개의 노드를 가지는, 일련의 function node, prime programs

3. end-if의 효용성

The Pascal if statement has the syntax

if Boolean_expression then statement else statement

whereas the Ada if is

if Boolean_expression then statement else statement end if

Comment on the relative merits of having an explicit terminator to the statement, such as the end if of Ada

sol)

Ada와 같이 explicit terminator는 문장들을 그룹화시키고, 각 decision에 대한 범주를 명확하게 구분 짓게 한다. 이것으로 인해 코드의 가독성도 높아지고, 코드의 흐름이 의도대로 흘러가게 하는 구조적 프로그래밍이 가능해진다. 만일 이러한 explicit terminator가 없다면, dangling else를 발생시키게 된다. C언어로 예를 들어보자.

```c
#include <stdio.h>

int main(void)
{
    int a = 6, b = 4;
    if(a > 5)
        if(b < 3)
            printf("b < 3");
    else
        printf("b > 3");

    return 0;
}
```

위의 코드를 보면, else가 첫 번째 if에 대한 else인지, 두 번째 if에 대한 else인지 명확하지가 않다. 이 경우 dangling else라고 하는데, 자칫하면 프로그래머의 의도대로 실행 결과가 나오지 않는 logical error를 발생시킨다. 그래서 C언어에서는 이러한 경우를 예방하고자 문법을 정해두었는데, 위와 같은 상황에서는 else의 바로 위에 있는 if에 대응되도록 정했다. 그래서 위 프로그램의 실행 결과는 그림 16과 같다. 만일 첫 번째 if에 대응되었다면, 아무것도 출력되지 않았을 것이다. 그런데 출력 결과를 보면, else가 바로 위에 있는 두 번째 if에 대응되었음을 알 수 있다.

```
b > 3
--------------------------------
Process exited after 0.04817 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```

그림 16 프로그램 실행 결과. else가 두 번째 if에 대응되었다.

그런데 if가 else의 바로 위에 없다면 어떻게 될까?

```c
#include <stdio.h>

int main(void)
{
    int a = 6, b = 4;
    if(a > 5)
        if(b < 3)
            printf("b < 3");
            printf("I'm in second if block.");
    else
        printf("b > 3");

    return 0;
}
```

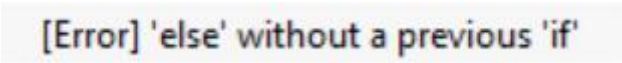

그림 17 프로그램 실행 결과. 구문 오류가 발생했다.

위의 코드 실행 결과는 그림 17과 같다. 구문 오류가 발생했다. C언어는 else 바로 위에서 if를 찾지 못하게 되면 구문 오류를 발생시키도록 정해두었다. 그래서 if-else 사이에는 어떠한 문장도 올 수 없다. 그렇다면 if 조건문 안에 여러 문장을 넣기 위해서는 어떻게 해야 할까? 블록을 지정해줘야 하는데, C언어에서는 { }를 이용한다. { } 내부의 문장들이 모두 if 조건문 내부의 문장임을 명시해주는 것이다.

```c
#include <stdio.h>

int main(void)
{
    int a = 6, b = 4;
    if(a > 5) {
        if(b < 3) {
            printf("b < 3");
            printf("I'm in second if block.");
        }
        else {
            printf("b > 3");
        }
    }

    return 0;
}
```

위의 코드는 { }를 이용해 블록을 명확히 구분 지었다. 그래서 dangling else 문제를 발생시키지 않고, 정상적으로 실행이 된다. C언어에서는 }가 Ada에서의 end if와 같은 역할을 한다. explicit terminate가 되어 가독성을 높이고, 오류 없이 돌아가는 구조적 프로그래밍을 가능하게 해준다.

추가적으로 Phython의 경우에는 따로 explicit terminate가 없지만, indent를 엄격히 따져 마치 explicit terminate가 있는 것과 같은 효과를 낸다.