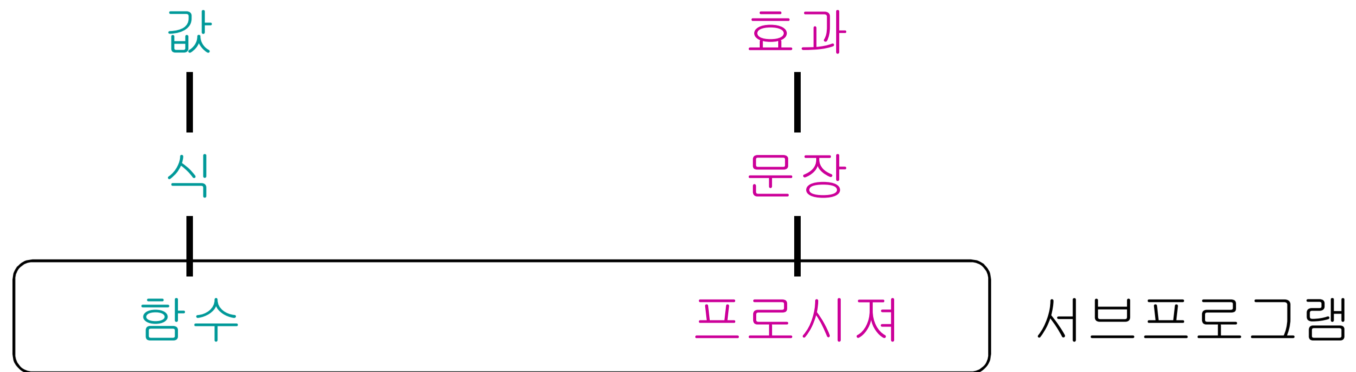
7 장
제어 I— 표현식과 문장

식
조건식과 가드
루프와 WHILE의 변형
GOTO 논쟁
예외 상황 처리

값과 효과들

- 프로그래밍 언어에서 중요한 개념들



- 복잡하게 하는 요소들
  - 부작용을 가지는 식들
  - 부작용을 가지는 함수들

제어 구조

- 암시적인 제어 구조
  - 식에서 피연산자들의 평가 순서
  - 일련의 문장들의 실행
- 명시적인 제어 구조
  - 구조적인 제어 문장들
  - 단일-입구, 단일-출구라는 원칙
- 선점적 제어 메커니즘
  - 예외 처리

식

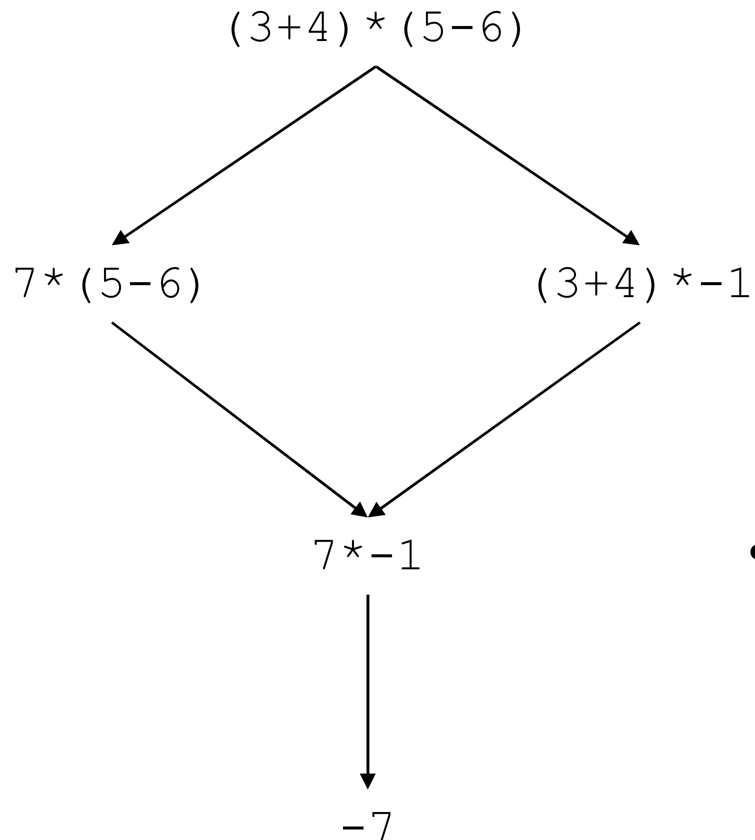
- 식은 다음으로 구성되어 있음
  - 연산자와 피연산자
  - 함수와 매개변수들
- 연산자들의 분류
  - 피연산자들의 개수
    - 단항 `- x`
    - 2항 `x + y`
    - 3항 `x ? y : z`

    ...
  - 연산자의 위치
    - 중위(infix) `x + y` most languages
    - 선위(prefix) `+ x y` LISP
    - 후위(postfix) `x y +` Postscript, FORTH

식의 평가 순서 (중요!)

- 적용 순서 평가(Applicative Order Evaluation) 또는 엄격 평가(Strict Evaluation)
  - 피연산자들이 먼저 평가됨
  - 평가된 피연산자들에 대해 연산자가 적용됨
- 정규 순서 평가(Normal Order Evaluation)
  - 극단의 지연 평가(delayed evaluation) 또는 비엄격 평가(non-strict evaluation) 또는 게으른 평가(lazy evaluation)
  - 연산자가 피연산자보다 먼저 평가됨
  - 연산자가 먼저 평가?
    … 나중 나중에 나오는 슬라이드에서 설명

적용 순서 평가의 예



- 대부분의 언어들에서, 피연산자 평가의 순서는 지정되어 있지 않음

평가 순서와 부작용

- 식이 부작용이 있다면, 프로그램은 틀릴 수 있음
- 예:
  - 오른쪽의 C 프로그램에서, 함수 f는 부작용이 있음
  - 출력은?
- 어떤 연산자는 부작용이 있음
  - C의 배정 연산자 (assignment operators)

```
#include <stdio.h>

int x = 1;

int f(void)
{ x += 1;
  return x;
}

int p( int a, int b)
{ return a + b;
}

main()
{ printf("%d\n",p(x,f()));
  return 0;
}
```

특정 연산자들의 평가 순서

- 시퀀스 연산자 (Sequence Operators)

```
x = (y+1,x+y,x+1)
x = 10,000;
x = (10,000);
```

- 논리적 And 와 논리적 Or
  - 논리적 이항 연산자는 단락 회로일 수 있음
- If 식

```
c ? e1 : e2
```

- Case 식

```
case e0 of
   a => e1 |
   b => e2 |
   c => e3 ;
```

단락 회로 평가 (Short-Circuit Evaluation)

- 단락 회로 평가를 통해 간략한 표현이 가능함
  - 단락 회로 평가 사용함

```
if (x != 0 && y%x == 1) ...
```

  - 단락 회로 평가 사용 안함

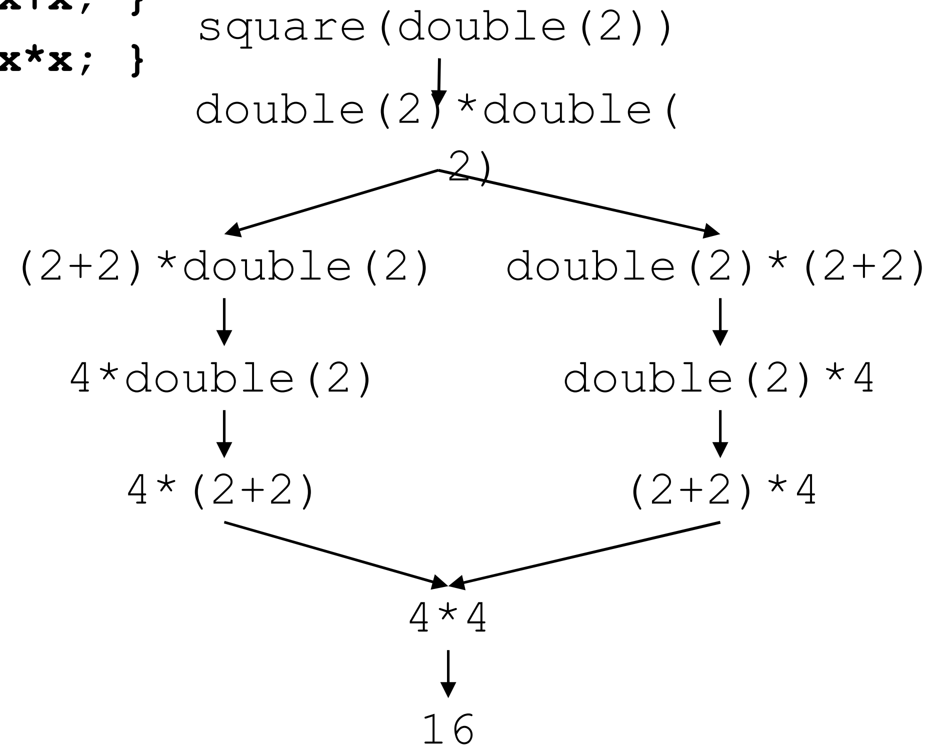
```
if (x != 0) if (y%x == 1) ...
```

- 어떤 언어들은 단락 회로 평가를 사용하는 연산자와 그렇지 않은 부울리언 연산자를 둘 다 지원함
  - Ada,
    - **and**, **or** 단락 회로 아님
    - **and then**, **or else** 단락 회로

정규 순서 평가

- 연산자들의 평가라는 건? ... 연산자 대신 사용자가 정의한 함수들로 생각해 보자.

```
int double(int x){ return x+x; }
int square(int x){ return x*x; }
```



조건문

- 조건문 안에선,

  조건문 내의 문장들은 특정 조건 하에서 한 번 만 실행됨

- 조건문들의 종류
  - 단일 또는 이중 조건: if 문과 if-else 문
  - 다중 선택: case- 문과 switch- 문

가드 If 문 (Guarded If)

- E. W. Dijkstra 가 주장함
- 문장을 비결정적으로 선택할 수 있음

```
if B1 -> S1
 | B2 -> S2
   ...
 | Bn -> Sn
fi
```

B_i 중 하나인 예를 들어 B_k 가 참이면, S_k 가 실행됨

현수-Else (Dangling-Else) 문제

- 어떤 **if** 가 뒤의 **else** 와 연결될까?

  if (*e1*) if (*e2*) *s1* else *s2*

- 해결책
  - 최근접 중첩 규칙(가장 가까운 중첩과 연결한다는 규칙)
    - **else** 는 다른 **else** 와 연결되지 않은 가장 가까운 앞의 **if** 와 연결됨
  - 구간 키워드 (bracketing keyword)
    - if 문은 다른 특정 키워드로 닫혀야 한다

      if … **fi**
    - Ada에서는 중첩 **if** 를 위해 **elsif** 를 지원함 (다음의 예)

```
if e1 then S1
else if e2 then S2
else if e3 then S3
end if ; end if ; end
if;
```

```
if e1 then S1
elsif e2 then S2
elsif e3 then S3
end if ;
```

Case-문과 Switch-문

- 가드가 순서 값(ordinal values) 인 ‘Guarded If’의 한 형태
- C의 Switch 문
  - 그냥 빠져나가는 (‘falls through’) 의미
  - 각 case 값은 컴파일 시간에 하나의 상수로 계산될 수 있어야 함
- Ada의 Case 문
  - case 값들이 그룹지어짐 (즉, case 값들이 수직 막대나, 부분범위를 통해 리스트될 수 있음)
  - case 값들 중 빠진 값, 즉 구멍이 없어야 함 (exhaustive) → 더 안전함
- ML의 Case 식
  - case 구성물이 값을 반환할 수 있어야 함 (그 자체가 식이므로)
  - case 는 값이라기보다는 패턴임
  - 디폴트 캐이스는 와일드 카드 패턴임 (wildcard pattern)

가드 Do (Guarded Do)

- 역시 E. W. Dijkstra
- 모든 가드들이 false일 때까지 문장이 반복됨

```
do B1 -> S1
 | B2 -> S2
   ...
 | Bn -> Sn
od
```

B_i 중 하나인 예를 들어 B_k 가 참이면, S_k 가 실행됨

While 루프

- 하나의 가드만을 가지는 가드 Do
- do-while 루프는 while의 구문적 당의 (syntactic sugar)

```
  do S while (e);
≡ S;
  while (e) S;
```

- break 문을 이용해 여러 개의 종료 포인트를 만들 수 있음 (cf. continue 문)

For 루프

- 카운터로 제어되는 반복문
- 배열 처리에 편함

  Note) ISO Standard C
- 카운터 변수 (인덱스 변수)의 범위는 일반적으로 루프 안으로 제한됨
- 루프의 최적화를 목적으로 인덱스 변수, 예를 들어 **`i`** ,에 대해 제한이 가해지기도 함:
  - `i` 는 루프 내에서 변경될 수 없음
  - `i` 는 루프가 끝나면 정의가 안됨
  - `i` 는 제한된 타입이어야 하며, 다른 방법으로 선언되어서는 안됨

For 루프의 설계 이슈들

- 루프의 경계값은 한번만 계산되어야 하는가?
- 하한(lower bound)이 상한(upper bound)보다 크다면, 루프의 내부 몸체는 전혀 수행되지 말아야 하는가? 아니면, 한번이라도 실행되어야 하는가?
  - C 언어의 do while, while, for
- 루프를 미리 나갈 수 있는가? (exit 또는 break 문)
- 인덱스 변수에 무슨 제한을 두어야 할까?

반복자(Iterator)

- CLU 같은 언어들의 Iterator
  - for 루프 구성물에 대한 일반적인 형태를 제시함
  - 과거에 민감한 함수와 비슷함
  - 일부 객체지향 언어들에 의해 이터레이터 객체로 적용됨

```
...
for c: char in stringchars(s) do
...
stringchars = iter (s: string) yields (char);
  index: int := 1;
  limit: int := string$size(s);
  while index <= limit do
    yield (string$fetch(s, index
    index := index + 1;
  end;
end stringchars;
```

이터레이터 stringchars 는 주어진 매개변수 스트링에서 문자를 하나씩 출력

C++ STL 또는 Java의 반복자 (중요)

- Java – 데모
- C++ – 2008년 2학기 고급객체지향 프로그래밍 제16장

GOTO 로 인한 스파게티 코드

- 다음 두 개의 코드를 비교해 보자

```
   IF (X.GT.0) GOTO 10
   IF (X.LT.0) GOTO 20
   X = 1
   GOTO 30
10 X = X + 1
   GOTO 30
20 X = -X
   GOTO 10
30 CONTINUE
```

```
if (x > 0)
    x = x + 1;
else if (x < 0)
{   x = -x;
    x = x + 1;
}
else
    x = 1;
```

이론적인 근거

- 구조적인 제어에 대한 이론적 토대
  - Böhm 과 Jacopini, 1996
  - 모든 가능한 제어 구조는 구조적인 제어 구성물로 표현될 수 있음
    - 시퀀스 구조
    - 선택 구조
    - 반복 구조

GOTO 논쟁

"GO TO Statement Considered Harmful"

— Dijkstra, 1968

"'GOTO considered harmful' considered harmful"

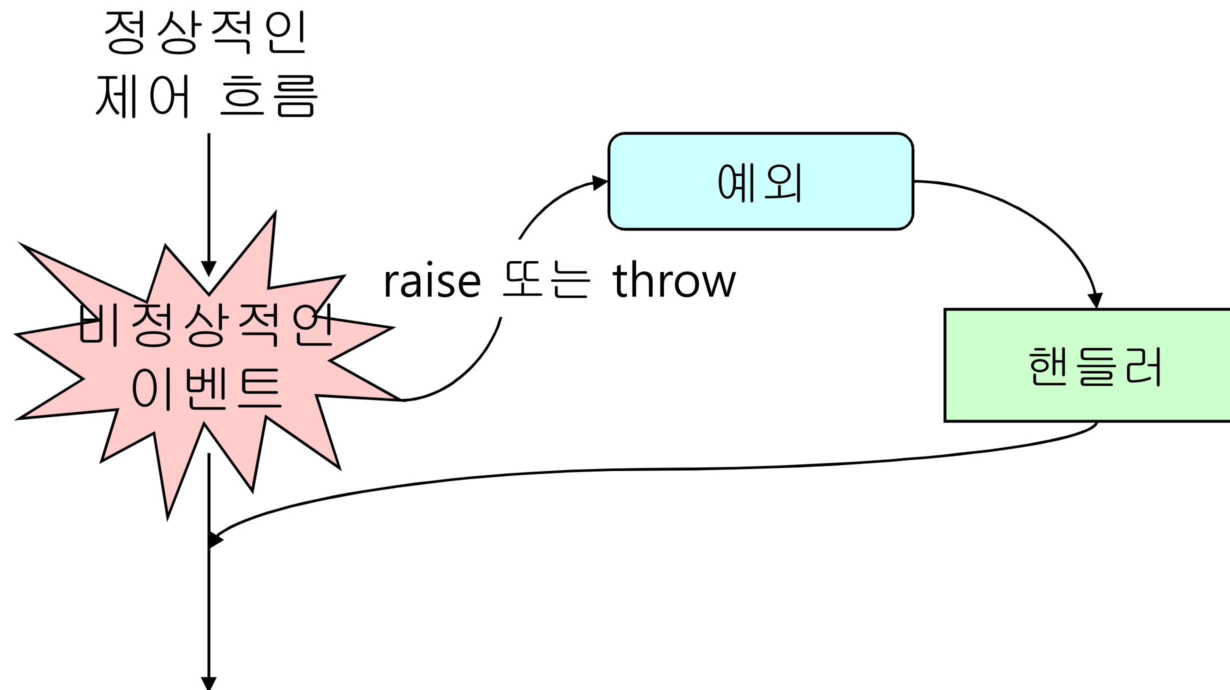
— Rubin, 1987

- 결과적으로…
  - GOTO의 사용에 심각한 제한이 가해졌다.
  - Java는 라벨이 있는 **`break`** 와 **`continue`** 로 변경했다.
  - Java에서 **`goto`** 는 (쓸 수 없는) 예약어이다.

예외 처리

- 예외들의 속성
  - 암시적인 제어 구조: 제어의 흐름에 대한 문법적인 표시가 없음
  - 선점적인 제어 구조: 일반적인 프로그램의 제어가 멈춰지고, 예외의 경우로 넘어감
- 예외들의 종류
  - 비동기적 예외
    - 언제든지 발생함
    - 하드웨어 실패, 통신 실패
  - 동기적 예외
    - 프로그램에 의해 직접 반응되어서 발생함
    - 파일 오픈 실패, 0으로 나누어짐

예외 처리 시나리오



예외 처리에서의 설계 이슈

- 예외
  - 어떤 예외들이 이미 정의되었는가?
  - 예외들이 사용 불가가 가능한가?
  - 사용자가 정의한 예외를 만들 수 있는가?
  - 예외의 영역은 어떻게 되는가?
- 핸들러
  - 어떻게 핸들러가 정의되는가?
  - 무슨 디폴트 핸들러가 제공되는가?
  - 이미 정의된 핸들러가 바뀔 수 있는가?
- 제어
  - 어떻게 핸들러에 제어가 넘어가는가?
  - 핸들러가 실행되고 나서 제어는 어디로 가는가?
  - 핸들러가 실행되고 나서 어떤 실행 환경이 남는가?

예외 (언어의 예들)

	예외 선언	예외에 추가적인 데이터	영역 규칙
ML	가능함	가능함	다른 선언들과 동일한 영역 규칙
Ada	가능함	불가능함	
C++	불가능하지만, 어떤 타입이든 예외로 던져짐	≈ 거의 가능함	

핸들러들 (언어의 예)

	핸들러 문법	기본 핸들러를 재정의?

C++	`try {`
	`  …`
	`}`
	`catch (type1 var1){…}`
	`catch (type2 var2){…}`
		가능함

Ada	`begin`
	`  …`
	`exception`
	`  when exception1 => …`
	`  when exception2 => …`
		불가능하나 기본 핸들러가 안되게 할 수는 있음

ML	`val name =`
	`  …`
	`handle`
	`  exception1 => …`
	`  exception2 => …`
		불가능함

제어

- 예외 발생 (Exception Raise)
  - 시스템이 정의한 예외: 자동 발생하거나, 수동 발생함
  - 사용자가 정의한 예외: 무조건 수동적으로 발생함
- 예외 전파 (Exception Propagation)
  - 만일 핸들러가 없다면, 그 코드를 덮고 있는 블록의 핸들러 부분을 참조함
  - 만일 가장 바깥의 블록에 다다를 때까지, 어떤 핸들러도 찾을 수 없다면, 디폴트 핸들러가 호출됨
- 핸들러로부터의 반환 (Return from Handler)
  - 속개 모델(Resumption Model): 제어가 처음 예외가 발생했던 포인트로 돌아옴
  - 종료 모델(Termination Model): 제어가 실행된예외 핸들러 다음 부분의 코드로 돌아옴 (대부분의 경우)

예외의 명세

- 예외의 명세
  - 서브프로그램에 붙어있는 예외들의 리스트
  - 서브프로그램은 예외 명세에 있는 예외만 발생시킨 수 있음
- 예외 명세의 장점
  - 컴파일러는 이 정보를 통해 최적화를 할 수 있음
  - 컴파일러는 필요없는 예외 핸들러를 미리 알 수 있음
- 예외 처리의 예
  - See Fig. 7.11 (Recursive Descent Calculator in C++)