제 7 장 타입 시스템

7.1 타입 오류와 타입 검사

타입 오류

- 문법에는 맞지만 제대로 실행될 수 없는 프로그램들이 많다!
  - 겉모양은 맞지만 속내용이 잘못되어 제대로 실행될 수 없는 프로그램
  - 여러 종류의 잘못이 있겠지만
  - 여기에서 다룰 내용은 데이터 타입이 잘못 사용되는 것(type error)

- 타입 오류(type error) ?
  - 프로그램 실행 중에 수식, 문장, 함수 등의 프로그램 구성요소가
  - 타입에 맞지 않게 잘못 사용되어 발생하는 오류.

- [예제 1]

```
>> int x = 1;
>> x = ! x + 2;
```

← syntax적으로는 문제가 없지만 language S에서는 boolean타입에 대해서만 ! 연산가능

타입 오류

- [예제 2]

```
>> int x = 1;
>> bool y = true;
>> y = x;        (1)   boolean에 int 할당
>> x = x+y;      (2)   boolean과 int의 연산
```

- [예제 3]

```
let int x = 1; bool y = true; in
    if (...) then x = x + 1;
    else x = x + y;    boolean과 int의 연산
end;
```

- [예제 4]

```
fun bool f(int x)
    if (x>0) then return 1;
    else return 0;
```

boolean이 아닌 int를 return함

타입 검사의 필요성

- 프로그램의 안전한(safe) 실행
  - 실행 중에 타입오류로 갑자기 죽는 프로그램을 미리 예방해야 한다.
  - 타입 검사를 하여 실행시간 타입 오류를 미리 예방할 수 있다.
- 타입 검사란 무엇인가?
  - 타입 정보를 이용하여 실행 중에 발생 가능한 타입 오류를 미리 검사하는 것
  - 프로그램에서 타입이 올바르게 사용되고 있는지 검사

타입오류가 있다면 실행하지 못하도록.

- 정적 타입 검사(static type checking)
  - 컴파일 시간에 가능한 타입 오류 미리 검사
  - 프로그램 안전성을 위해 매우 중요하다 !

타입 검사

- 구문 검사(syntax analysis)
  - 프로그램이 구문법(syntax grammar)에 맞는지 검사
  - 1세대 기술로 1970년대에 활발히 연구 개발되었다.
  - 모든 언어에 적용되고 있다.

- 타입 검사(type checking)
  - 프로그램 구성요소가 데이터 타입에 맞게 올바르게 사용되고 있는지 검사
  - 2세대 기술로 1990년대부터 활발히 연구 개발되었다.
  - 안전한 타입 시스템을 갖춘 언어
    - Java, C#, ML, Haskell, Smalltalk

강한 타입 언어 vs 약한 타입 언어

- 강한 타입 언어(strongly typed language)
  - 프로그램 실행 전에 타입 검사를 엄격하게(strict) 하면 할수록 실행시간 타입 오류를 보다 확실하게 예방할 수 있을 것이다.
  - 엄격한 타입 규칙을 적용하여 타입 오류를 찾아 내는 언어
  - 예: Java, ML, C#, Python 등

- 약한 타입 언어(weakly typed language)
  - 느슨한 타입 규칙을 적용하여 타입 검사하는 언어
  - 예: C/C++, PHP, Perl, Java Script 등
  - 타입 규칙을 적용하여 타입 검사를 하더라도 이 검사를 통과한 프로그램이 실행 중에 타입 오류가 발생할 수 있다.

안전한 타입 시스템

↳ 타입규칙의 집합

- 안전한 타입 시스템(sound type system)
  - 이 타입 시스템의 타입 검사를 통과한 프로그램은 타입 오류를 일으키지 않을 것이 보장된다(no false positive).

- 포함 관계

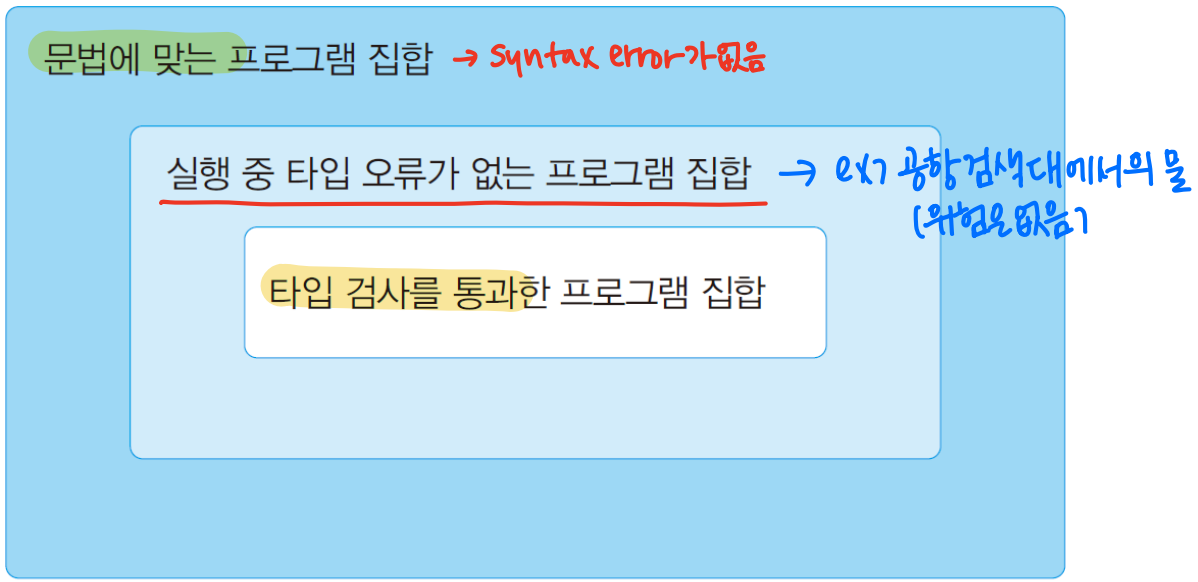


그림 7.1 안전한 타입 시스템으로 타입 검사

타입 검사 종류 (언제하느냐에따라)

- 정적 타입 검사(Static Type Checking)
  - 거의 모든 타입 검사를 컴파일 시간에 한다
  - Java, Pascal, C, C++, ML, Haskell
- 동적 타입 검사(Dynamically typed) → 컴파일을안하는언어
  - 실행 시간에 타입 검사
  - Lisp, Scheme
- 타입 검사 안 함
  - 어떤 종류의 타입 검사도 하지 않음(어셈블리어)

7.2 타입 시스템 개요

지금까지 한 것/앞으로 할 것!

주제	논리	구현
구문법	문법	파서
의미론	의미 함수	인터프리터
타입	타입 규칙 → 타입시스템 (사용)	타입 검사기

타입 규칙과 타입 시스템

- 타입 규칙(typing rule)
  - 수식, 문장, 함수 등과 같은 프로그램의 구성요소의 올바른 타입 사용 규칙

- 타입 시스템(type system)
  - 프로그래밍 언어의 수식, 문장, 함수 등과 같은 프로그램의 구성요소의 타입 규칙으로 구성된 시스템을 그 언어의 타입 시스템이라고 한다.

타입 규칙(Typing Rule)

- 타입 사용에 대한 규칙으로 타입 검사에 사용된다.
  - 논리적 추론 규칙(logical inference rule)으로 표현 P→Q
  - "if X and Y then Z"

- 규칙 예 (Int, String, Boolean 중 Int만 가능 (나머지경우는 모두 error))
  - If E1 has type Int and E2 has type Int, then E1 + E2 has type Int

P→Q형식 (E1 has type Int ∧ (and) E2 has type Int) ⇒ E1 + E2 has type Int

  - (E1 : Int ∧ (and) E2 : Int) ⇒ E1 + E2 : Int
    (P ⇒ Q)

타입 규칙을 추론 규칙으로 표현

- 타입 규칙 예

  $(E1 : Int \wedge E2 : Int) \Rightarrow E1 + E2 : Int$

- 일반적 형태

$$p_1 \wedge \ldots \wedge p_n \rightarrow q$$

$$\Vert$$

$$\text{가정}_1 \wedge \ldots \wedge \text{가정}_n \rightarrow \text{결론}$$

- 추론 규칙

$$\frac{\text{가정}_1 \quad \ldots \quad \text{가정}_n}{\text{결론}}$$

P ↓ Q

왜 분수형태로 표시?
시각적으로 보기 쉽게 하기 위함

타입 규칙 예

- 정수와 + 식에 대한 타입 규칙:

$$\vdash n : int \qquad \frac{\vdash E1:int \quad \vdash E2:int}{\vdash E1 + E2:int}$$

- 식 2 + 5의 타입은 ?

number (상수)

$$\vdash 2 + 5:int$$

$$/ \qquad \backslash$$

$$\vdash 2:int \qquad \vdash 5:int$$

- 그런데 변수를 포함한 수식 x + y의 타입은 ?
  - 변수 x와 y의 타입에 따라 다르다!
  - ✡ 유효한 변수의 타입을 유지하는 타입 환경(type env)이 필요하다!

↳ 항상 변수의 타입정보가 필요함
ex) semantics 에서 수식을 계산할때 변수의 값이 필요한 것처럼
(변수의 값을 모아놓은 state가 필요한 것처럼 타입환경 필요!)

타입 환경

- 타입 환경(Type environment)
  - 각 지점에서 유효한 변수들의 타입 정보를 유지한다.
  - Γ : Identifier → Type
  - 예 $\Gamma = \{x \mapsto int,\ y \mapsto int\}$

- 변수 x의 타입은 ?

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}$$

- 식 x + y의 타입은 ?

$$\frac{\Gamma \vdash x : int \qquad \Gamma \vdash y : int}{\Gamma \vdash x+y : int}$$

타입 안전성

- 안전한 타입 시스템(sound type system)
  - 타입 시스템에 의해 어떤 수식의 타입이 t라고 결정했으면
    수식 E를 실제 실행하여 계산된 값이 반드시 t 타입의 값이어야 한다.
  - 즉 ⊢ E : t이면, 실제 실행에서 E의 계산된 값이 t 타입이다.
  - If ⊢ E : t and V(E) = v, then v : t

- 안전한 타입 시스템 → 증명해야함!
  - 타입 검사에서 수식 E가 오류가 없으면 이 식은 실제 실행에서도 타입 오류가 없다는 것을 의미함!

타입 안전성: 사례

- 안전하지 않은 타입 시스템
  - C, C++
  - 느슨하면서 안전하지 않은 타입 시스템을 갖추고 있음.
  - 주로 타입 캐스트(cast)나 포인터 연산 때문에 안전하지 않음.
- 거의 안전한 타입 시스템
  - Pascal, Ada.
  - 허상 포인터(dangling pointer) 때문에 안전하지는 않다
- 안전한 타입 시스템
  - ML, Java
  - 엄격하면서 안전한 타입 시스템을 갖추고 있다.
  - 컴파일 과정에서 엄격하게 타입 검사를 함으로써 실행시간에 타입으로 인한 오류를 미연에 방지한다.

7.3 언어 S의 타입 시스템

언어 S

Command C → D | S | F

Decl D → T id [=E]

Stmt S → id = E
| S; S
| if E then S [else S]
| while (E) S
| read id
| print E
| let D in S end
| return E → return 뒤만 void가아님!

Type T → int | bool | string | void ← string

Expr E → n | id | true | false | str | f(E)
| E + E | E - E | E * E | E / E | (E)
| E == E | E != E | E < E | E > E | !E

타입 규칙: 상수/변수

- 타입 환경(Type environment)
  - 각 지점에서 유효한 변수들의 타입 정보를 유지한다.
  - Γ:Identifier → Type (mapping)
- 상수를 위한 규칙

$\Gamma \vdash n:int$ $\quad$ $\Gamma \vdash true:bool$ $\quad$ $\Gamma \vdash false:bool$

타입환경 하에서

- 변수를 위한 규칙

$$\frac{\Gamma(id) = t}{\Gamma \vdash id : t}$$

P ↓ Q

→ mapping된것을 확인해야함

타입 규칙: 수식

- 수식을 위한 규칙

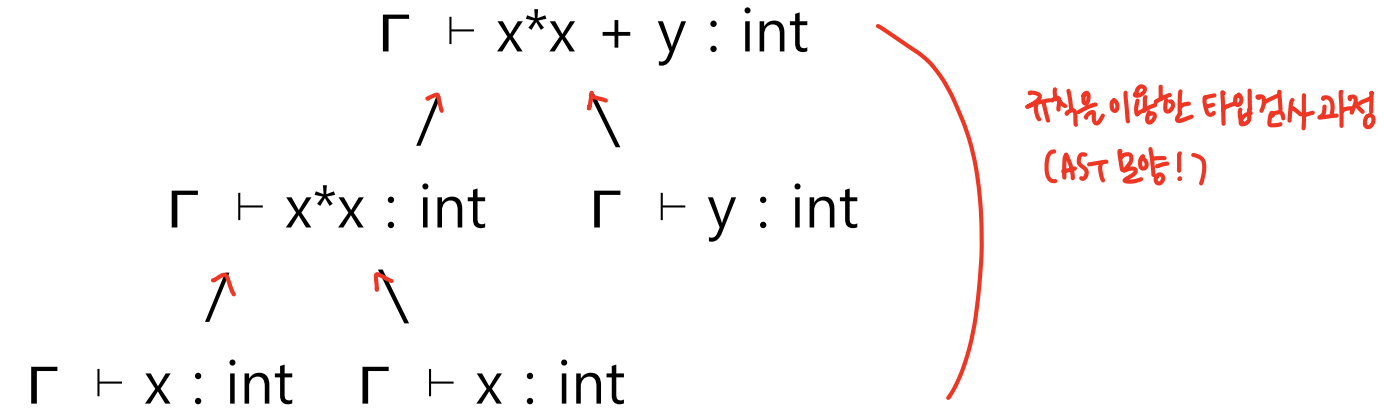
$$\frac{\Gamma \vdash E1 : \texttt{int} \quad \Gamma \vdash E2 : \texttt{int}}{\Gamma \vdash E1 + E2 : \texttt{int}}$$

P ↓ Q

↳ Language S에서 +는 integer만가능함!
(int, string, boolean 3가지 있으므로)

- 예

$\Gamma = \{ x \mapsto \text{int}, y \mapsto \text{int} \}$ → 규칙



규칙을 이용한 타입검사 과정
(AST 모양!)

타입 규칙: 수식

- 관계 수식 → 꼭 integer일필요X 반드시같은type이어야함

$$\frac{\Gamma \vdash E1 : t \qquad \Gamma \vdash E2 : t}{\Gamma \vdash E1 == E2 : \text{bool}}$$

* E!=E , E<E , E>E 도 마찬가지!
* !E의 타입규칙은?

$$\frac{\Gamma \vdash E : \text{bool}}{\Gamma \vdash !E : \text{bool}}$$

- 예

$\Gamma = \{ x \mapsto \text{int}, y \mapsto \text{int} \}$ → 타입규칙

$$\Gamma \vdash x == y : \text{bool}$$

$$\Gamma \vdash x : \text{int} \qquad \Gamma \vdash y : \text{int}$$

타입규칙에의한 타입검사과정

타입 규칙: 대입문

가장 간단한 statement : assignment

- 대입문의 타입 규칙
  - 대입문의 왼쪽 변수는 유효하고 오른쪽 식의 타입과 같아야 한다.

C언어에서는 assign한 결과 반환값도 가지지만 language S의 assign은 단순히 저장하는 역할!

$$\frac{\Gamma \vdash id : t \quad \Gamma \vdash E : t}{\Gamma \vdash id = E : void}$$

→ 같은 type이어야 assign 가능!

(empty)

- [예제 6]

```
let bool x; in
    x = 1;
end;
```

→ bool에 int를 assign (오류)

[예제 7]

```
let int x = 0; in
    x = x + 1;
end;
```

→ int에 int assign

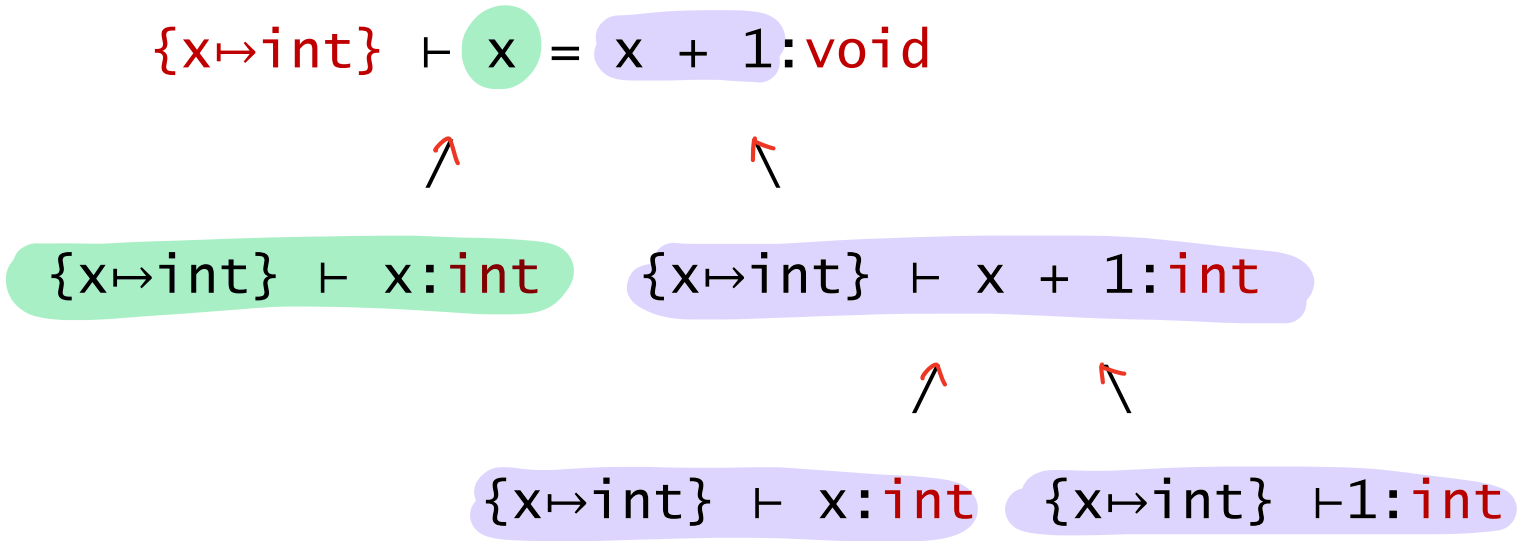
타입 검사: 대입문

- [예제 7]

  변수선언 → 변수가유효해짐 → 타입정보는 타입 environment에 추가

```
0 let int x = 0; in
1     x = x + 1;
2 end
```

- 타입 검사



타입 규칙: let-문

- let 문의 타입 규칙
  - let 문의 변수 선언에서 초기화 수식은 변수의 타입과 같아야 한다.
  - let 문의 문장 S가 타입 오류가 없어야 한다.

→ Stack에 추가했다고 생각하기

$$\frac{\Gamma \vdash E : t \qquad \Gamma[id \mapsto t] \vdash S : t'}{\Gamma \vdash \text{let } t\ id = E \text{ in } S \text{ end} : t'}$$

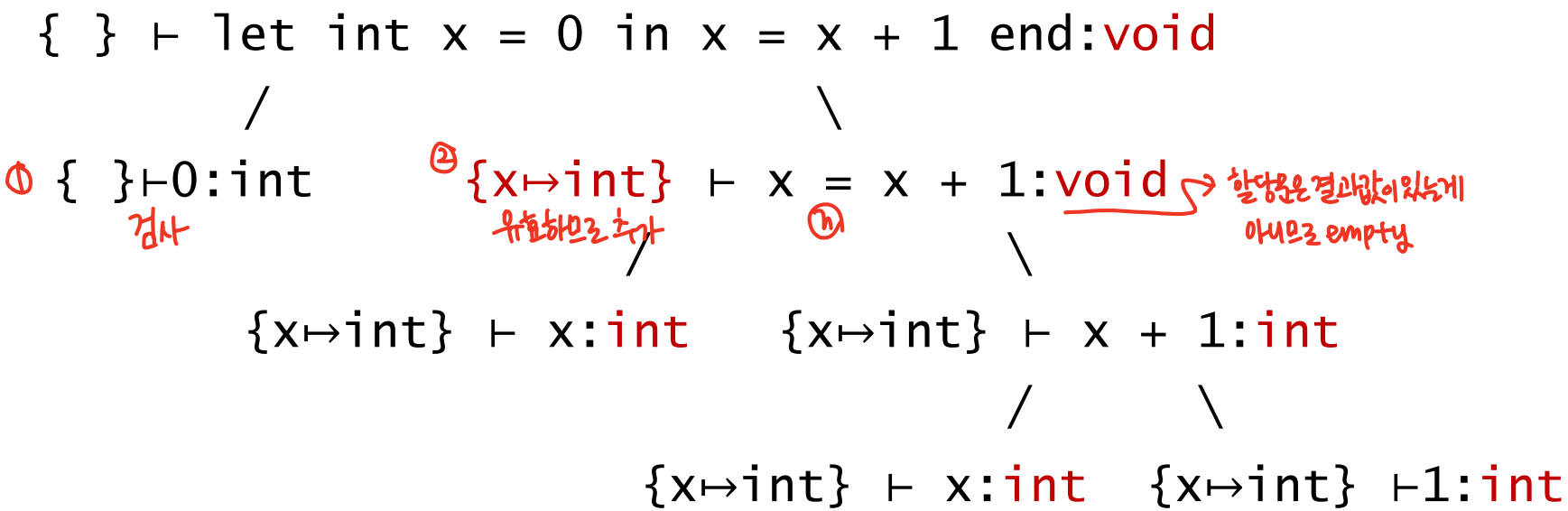
① id가 t타입이므로 초기화수식 E도 t타입
② 변수가 선언됐으므로 id가 유효해짐
→ id가 t타입이라는 것을 타입환경에 추가
③ S의 타입검사

- 예

```
let int x = 0; in
   x = x + 1;
end
```

타입 검사: let 문

- 타입 검사 과정
  1. 초기화 수식 E가 변수의 타입과 같은지 검사한다.
  2. 블록이 시작되면 선언된 변수의 타입 정보를 추가한다.
  3. 이 타입 환경에서 S의 타입을 검사하고 결과 타입이 정해진다.
     - 이 결과 타입이 let 문의 타입이 된다.



타입 규칙: 리턴문

- 리턴문의 타입 규칙
  - 반환 값의 타입은 E의 타입이 된다.

$$\frac{\Gamma \vdash E : t}{\Gamma \vdash \texttt{return } E : t}$$

- 타입 검사 예

```
fun int square(int x)
    return x * x;
```

$$\{x \mapsto \texttt{int}\} \vdash \texttt{return } x*x : \texttt{int}$$

$$\{x \mapsto \texttt{int}\} \vdash x*x : \texttt{int}$$

타입 규칙: 조건문

- 조건문를 위한 규칙
  - 조건식 E는 bool 타입이어야 하고
  - then 문장 S1과 else 문장 S2는 같은 타입이어야 한다.

$$\frac{\Gamma \vdash E:\mathtt{bool} \quad \Gamma \vdash S1:t \quad \Gamma \vdash S2:t}{\Gamma \vdash \mathtt{if}\ E\ \mathtt{then}\ S1\ \mathtt{else}\ S2:t}$$

+) 하나가 return안하면 다른것도 return하면안됨! (둘다 void)

- [예제 8]

```
fun bool f(int x)
    if (x > 0) then return 1;
    else return false;
```

int / bool → 잘못사용!

[예제 9]

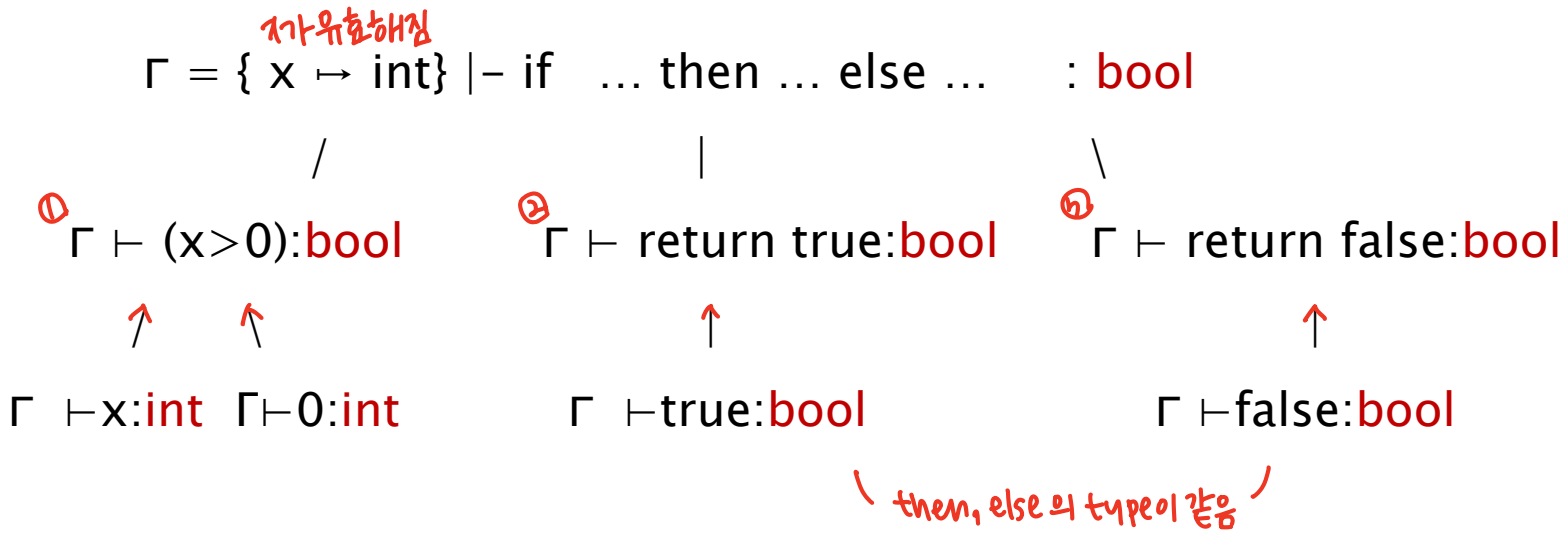
```
fun bool f(int x)
    if (x > 0) then return true;
    else return false;
```

같은 type

타입 검사: 조건문

- [예제 9]

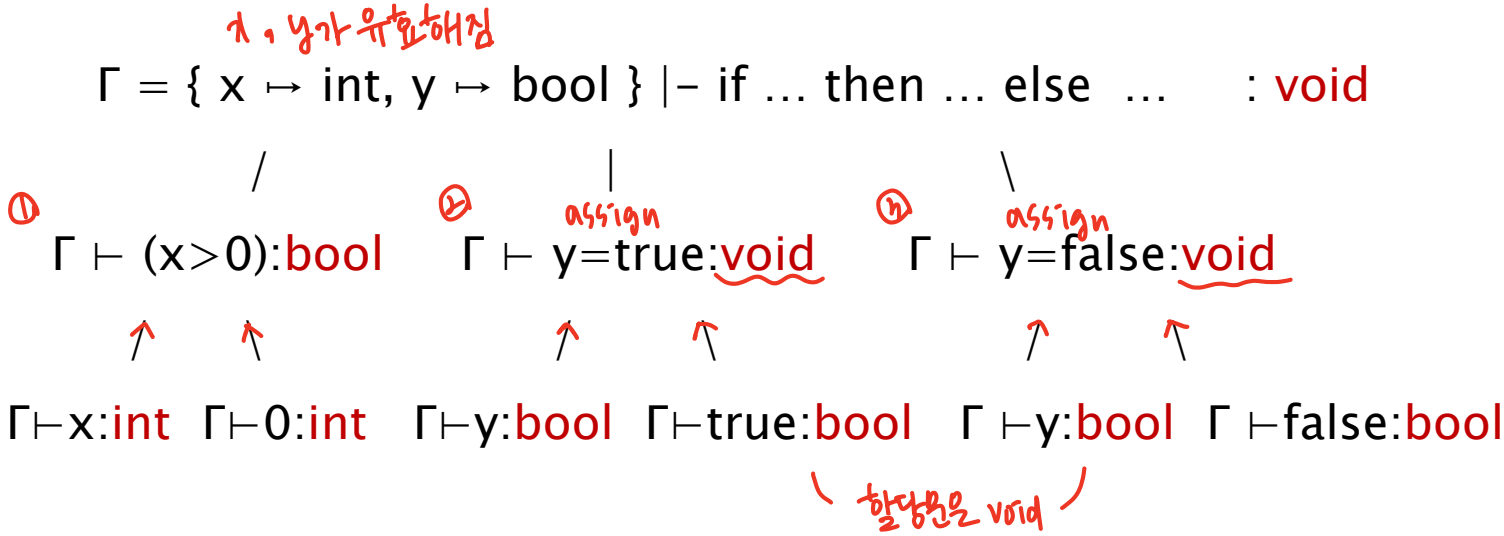
```
fun bool f(int x)
    if (x > 0) then return true;
    else return false;
```



타입 검사: 조건문

- [예제 10]

```
let int x; bool y; in
    if (x>0) then y=true;
    else y=false;
end;
```



타입 규칙: 복합문

- 복합문을 위한 규칙
  - return-문이 오고 그 다음에 다른 문장이 오면 안 된다.

$$\frac{\Gamma \vdash S1:\texttt{void} \quad \Gamma \vdash S2:t}{\Gamma \vdash S1;\ S2:t}$$

Language S에서는 return을 제외하면 모두 void?

  - (1) 첫 번째 문장 S1은 반드시 void 타입이어야 한다.
  - (2) 두 번째 문장 S2는 임의의 타입이 가능하며
  - 이 타입이 복합문의 결과 타입이 된다.

- [예제 11]

```
fun bool f(int x) {
    if (x > 0) then return true;
    else return false;
    x = x - 1;
}
```

return문 뒤의 문장은 실행되지 않음 (unreachable) → 타입오류

[예제 12]

```
fun bool f(int x) {
    x = x - 1;
    if (x > 0) then return true;
    else return false;
}
```

타입 규칙: 반복문

- 반복문을 위한 규칙
  - 조건식은 bool 타입이어야 하고
  - ✱ 반복문 내에서 return 하면 안됨.

$$\frac{\Gamma \vdash E:\texttt{bool} \quad \Gamma \vdash S:\texttt{void}}{\Gamma \vdash \texttt{while } E \texttt{ do } S:\texttt{void}}$$

Language S는 return문만 void타입이아님
→ void타입이다 == return문이 아니다!

- [예제 14]

```
fun int f(int x)
let int y = 1;
in     while (x > 0) {
           y = y * x;
           x = x –1;
           return y;
       }
end;
```

↳ while문내에서 return : 한번실행하면 끝나버리거나(x>0) return하지않음(x≤0) (오류의소지)

[예제 15]

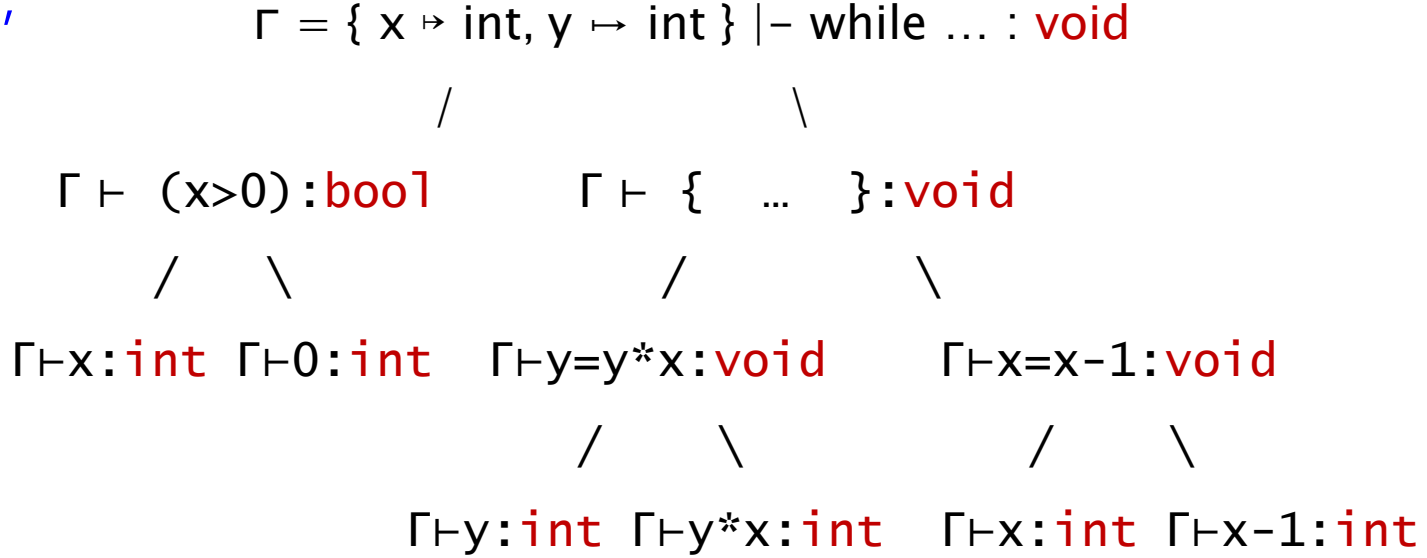
```
fun int f(int x)
let  int y = 1;
     while (x > 0) {
         y = y * x;
         x = x –1;
     }
     return y;
end;
```

✓ 어떤 경우든 return가능

타입 검사: 반복문

- [예제 15]

```
fun int f(int x)
let int y = 1;
in
    while (x > 0) {
        y = y * x;
        x = x –1;
    }
    return y;
end;
```



타입 규칙: 입출력

- 입출력을 위한 규칙

$$\frac{\Gamma \vdash \texttt{id}:\texttt{t}}{\Gamma \vdash \texttt{read id}:\texttt{void}} \qquad \frac{\Gamma \vdash \texttt{E}:\texttt{t}}{\Gamma \vdash \texttt{print E}:\texttt{void}}$$

7.4 타입 검사 구현

[언어 S의 문법]

EBNF

<program> → {<decl> | <stmt> | <function>}
<decl> → <type> id [= <expr>];
<stmt> → id = <expr>;
| '{' <stmts> '}'
| if (<expr>) then <stmt> [else <stmt>]
| while (<expr>) <stmt>
| read id;
| print <expr>;
| let <decls> in <stmts> end;
<stmts> → {<stmt>}
<decls> → {<decl>}
<type> → int | bool | string

타입 환경

- 타입 환경(Type environment)
  - 프로그램 각 지점에서 유효한 변수/함수의 타입 정보
  - 타입 환경 Γ : Identifier → Type

    Identifier = 식별자(변수 혹은 함수 이름) 집합

    Type = 타입 집합

  - 예

    chap09에서

    Γ = {y ↦ int, f ↦ int → bool }

- 타입 환경 구현
  - 일종의 심볼 테이블로 Stack을 확장해서 구현한다.
  - public class TypeEnv extends Stack<Entry> { … }

타입 환경 구현

- 일종의 심볼 테이블로 Stack으로 구현한다.

```
class TypeEnv extends Stack<Entry> {
    public TypeEnv(Identifier id, Type t) {
        push(id, t);
    }
    public TypeEnv push(Identifier id, Type t) {
        super.push(new Entry(id, t));
        return this;
    }
    // 식별자가 타입 환경에 포함되어 있는가?
    public boolean contains (Identifier v) { ... }
    // 타입 환경의 탑에서부터 식별자를 찾아 타입 가져오기
    public Type get(Identifier v) { ... }
}
```

변수하나에 타입정보를 담음

```
class Entry {
    Identifier id;
    Type type;
    Entry (Identifier id, Type t) {
        this.id = id;
        this.type = t;
    }
}
```

→ true/false

타입 검사기 구현

- 타입 검사기 구현

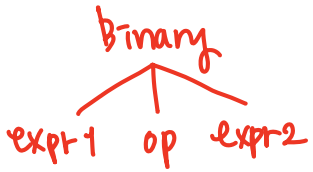
  파싱이 끝났으니까

  - 입력 프로그램의 AST를 순회하면서 Stmt와 Expr를 만날 때마다
  - 해당 타입 규칙(type rule)을 적용하여 타입 검사를 구현
  - 타입 오류에 대해서 오류 메시지 출력

이항 연산의 타입 검사

AST

```
static Type Check(Binary b, TypeEnv te) {
    Type t1 = Check(b.expr1, te);
    Type t2 = Check(b.expr2, te);
    switch(b.op.val) {
① case "+": case "-": case "*": case "/":
        if (t1 == Type.INT && t2 == Type.INT)
            b.type = Type.INT;
        else error(b, "type error for " + b.op);
        break;
② case "<": case "<=": ...
        if (t1 == t2)
            b.type = Type.BOOL;
        else  error(b, "type error for " + b.op);
        break;
            . . .
    }
    return b.type;
}
```



→ int가아닌경우들 모두 error

→ 같은 type이 아닌 경우 모두 error

문장의 타입 검사

```
public static Type Check(Stmt s, TypeEnv te) {
    if (s instanceof Assignment)
        return Check((Assignment) s, te);
    if (s instanceof Stmts)
        return Check((Stmts) s, te);
    if (s instanceof If)
        return Check((If) s, te);
    if (s instanceof While)
        return Check((While) s, te);
    if (s instanceof Let)
        return Check((Let) s, te);
    if (s instanceof Call)
        return Check((Call) s, te);
        ...
}
```

대입문의 타입 검사

```
static Type Check(Assignment a, TypeEnv te) {
    if (!te.contains(a.id)) {
        error(a, " undefined variable in assignment: " + a.id);
        return Type.ERROR;
    }
    Type t1 = te.get(a.id);              // 변수 타입 찾기
    Type t2 = Check(a.expr, te);         // 수식 타입 검사
    if (t1 == t2)
        a.type = Type.VOID;
    else
        error(a, "mixed mode assignment to " + a.id);
    return a.type;
}
```

id=<expr>

→ assign가능

→ void

if 문의 타입 검사

AST

if <expr> then <stmt>
else <stmt>

```
static Type Check(If c, TypeEnv te) {
    Type t = Check(c.expr,te);
    Type t1 = Check(c.stmt1, te);
    Type t2 = Check(c.stmt2, te);
    if (t == Type.BOOL)
        if (t1 == t2)
            c.type = t1;
        else
            error(c, "non-equal type in two branches");
    else
        error(c, "non-bool test in condition");
    return c.type;
}
```

While 문의 타입 검사

```
static Type Check(While l, TypeEnv te) {
    Type t = Check(l.expr,te);
    Type t1 = Check(l.stmt, te);
    if (t == Type.BOOL)
        if (t1 == Type.VOID)
            l.type = t1;
        else
            error(l, "return in loop..");
    else
        error(l, "non-bool test in loop");
    return l.type;
}
```

while <expr>
<stmt>

→ void가 아닌 경우!

let 문의 타입 검사

```
static Type Check(Let l, TypeEnv te) {
    addType(l.decls, te);
    l.type = Check(l.stmts, te);
    deleteType(l.decls, te);
    return l.type;
}

public static TypeEnv addType(Decls ds, TypeEnv te) {
    if (ds == null)
    for (Decl d : ds)
        Check(d, te);
    return te;
}
```

변수가 유효해졌으므로 추가

→ 더이상 유효하지 않으므로

→ stack 자체를 return ; 참조변수를 사용하므로
(변수의 type정보가 필요한 것이므로)

Let <decls> → addType
in
<stmt>
end → deleteType