中国科学技术大学
University of Science and Technology of China

USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
类型检查

计算机科学与技术学院

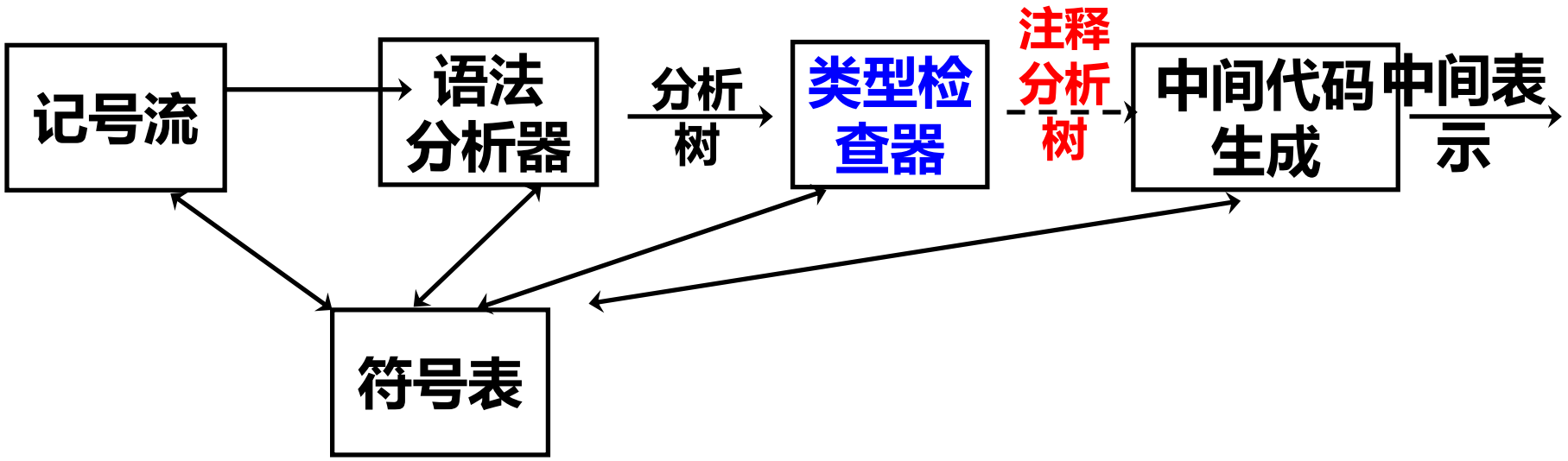
李 诚

5/11/2018

Announcement

- 上周书面作业延期到11.15日与今天的书面作业一起提交，project的提交时间不变。

主要内容



- 静态检查—类型检查
- 描述类型系统的语言
- 简单类型检查器的说明
- 类型表达式的等价
- 函数和算符的重载

程序运行时的执行错误

- 会被捕获的错误（trapped error）
  - 例：非法指令错误、非法内存访问、除数为零
  - 引起计算立即停止
- 不会被捕获的错误(untrapped error)
  - 例：下标变量的访问越过了数组的末端；跳到一个错误的地址，该地址开始的内存正好代表一个指令序列
  - 错误可能会有一段时间未引起注意

希望可执行的程序不存在不会被捕获的错误

安全语言

- 良行为的(well-behaved)程序
  - 没有统一的定义
  - 如: 没有任何不会被捕获的错误的程序
- 安全语言(safe language)
  - 定义: 安全语言的任何合法程序都是良行为的
  - 设计类型系统, 通过静态类型检查拒绝不会被捕获错误
  - 设计正好只拒绝不会被捕获错误的类型系统是困难的
- 禁止错误(forbidden error)
  - 不会被捕获错误集合+ 会被捕获错误的一个子集

类型化的语言

- 变量的类型
  - 限定了变量在程序执行期间的取值范围
- 类型化的语言(typed language)
  - 变量都被给定类型的语言
  - 表达式、语句等程序构造的类型都可以静态确定

  例如，类型boolean的变量x在程序每次运行时的值只 能是布尔值，not (x)总有意义
- 未类型化的语言(untyped language)
  - 不限制变量值范围的语言,如JavaScript、Perl

类型化的语言

❑显式类型化语言

- ❖类型是语法的一部分

❑隐式类型化的语言

- ❖不存在隐式类型化的主流语言，但可能存在忽略类型信息的程序片段,如不需要程序员声明函数的参数类型

类型系统

- 语言的组成部分,其构成成分是一组定型规则(typing rule),用来给各种程序构造指派类型
- 设计目的
  - 用静态检查的方式来保证合法程序在运行时的良行为
- 类型系统的形式化
  - 类型表达式、定型断言、定型规则
- 类型检查算法
  - 通常是静态地完成类型检查

类型可靠的语言

- 良类型的程序(well-typed program)
  - 没有类型错误的程序，也称合法程序
- 类型可靠（type sound）的语言
  - 所有良类型程序（合法程序）都是良行为的
  - 类型可靠的语言一定是安全的语言

语法的和静态的概念	动态的概念
类型化语言	安全语言
良类型程序	良行为的程序

类型检查与推断

- 类型检查
  - Type Checking is the process of verifying fully typed programs
- 类型推断
  - Type Inference is the process of filling in missing type information

类型检查

- 未类型化语言
  - 可以通过运行时的类型推断和检查来排除禁止错误
- 类型化语言
  - 类型检查也可以放在运行时完成，但影响效率
  - 一般都是静态检查，类型系统被用来支持静态检查
  - 通常也需要运行时的检查，如数组访问越界检查

一些编程语言并不安全

禁止错误集合没有囊括所有不会被捕获的错误

- **用C语言的共用体（*union*）来举例**

```
union U { int u1; int *u2;} u;

int *p;

u.u1 = 10;

p = u.u2;

*p = 0;
```

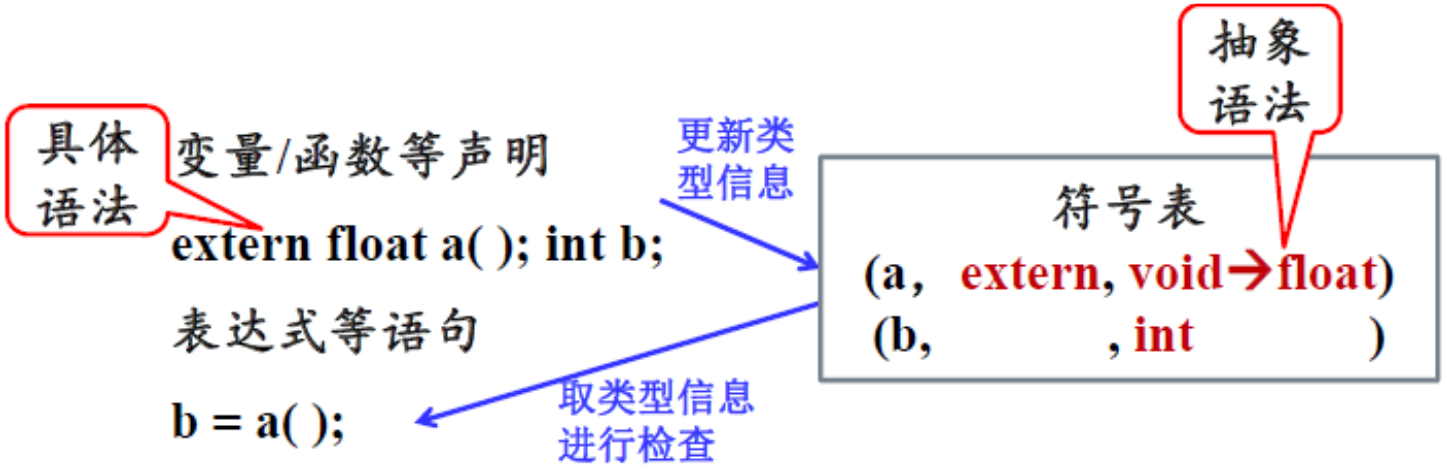
一些编程语言并不安全

- C语言
  - 有很多不安全但被广泛使用的特征，如：指针算术运算、类型强制、参数个数可变
  - 在语言设计的历史上，安全性考虑不足是因为当时强调代码的执行效率
- 在现代语言设计上，安全性的位置越来越重要
  - C的一些问题已经在C++中得以缓和
  - 更多一些问题在Java中已得到解决

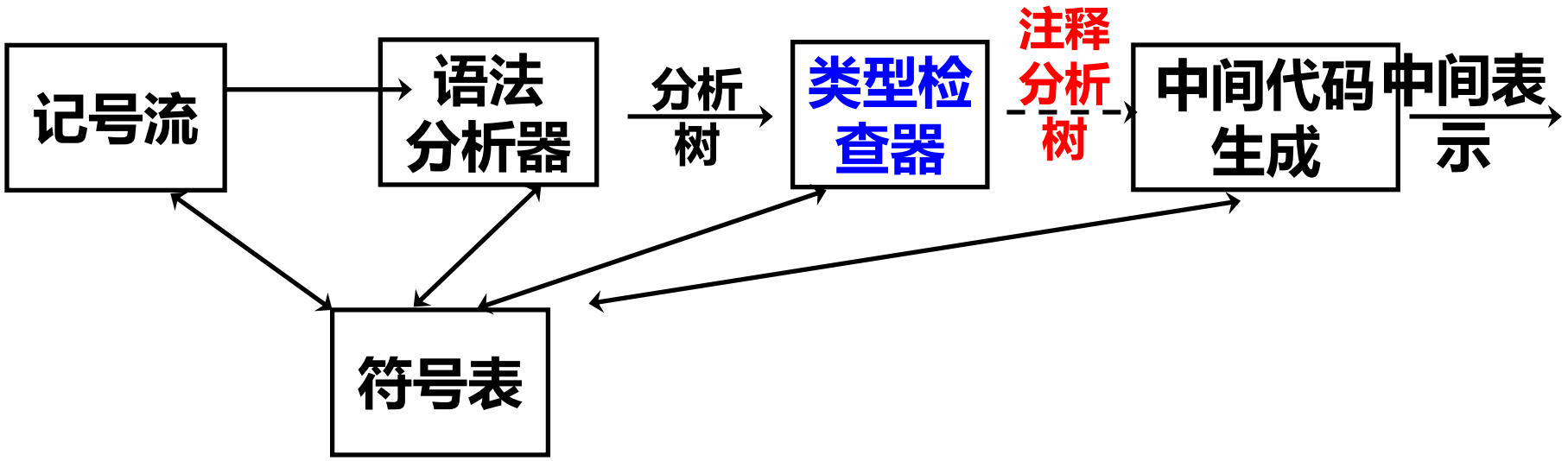
类型化语言的优点

❑从工程的观点看

- ❖开发的实惠：较早发现错误、类型信息具有文档作用
- ❖编译的实惠：程序模块可以相互独立地编译
- ❖运行的实惠：可得到更有效的空间安排和访问方式



主要内容



- ❑静态检查—类型检查
- ❑描述类型系统的语言
  - ❖类型系统的形式化、类型检查与推断
- ❑简单类型检查器的说明
- ❑类型表达式的等价

类型系统的形式化

❑类型系统是一种逻辑系统

有关自然数的逻辑系统

- ❖自然数表达式（需要定义它的语法）

  $a+b, 3$

- ❖良形公式　（逻辑断言，需要定义它的语法）

  $a+b=3, (d=3)\wedge(c<10)$

- ❖推理规则

$$\frac{a < b, \quad b < c}{a < c}$$

前提

结论

类型系统的形式化

❑类型系统是一种逻辑系统

有关自然数的逻辑系统

❖自然数表达式

a+b, 3

❖良形公式

a+b=3, (d=3)∧(c<10)

❖推理规则

$$\frac{a < b, \quad b < c}{a < c}$$

类型系统

❖类型表达式

int, int → int

❖定型断言

x:int |– x+3 : int

❖定型规则

定型环境(符号表)

$$\frac{\Gamma \mid\!- M : \text{int}, \Gamma \mid\!- N : \text{int}}{\Gamma \mid\!- M + N : \text{int}}$$

❑断言的形式

Γ|– S　　S的所有自由变量都声明在Γ中

其中

- ❖ Γ是一个静态定型环境(编译器实现中的符号表),
  如 $x_1:T_1, \ldots, x_n:T_n$
- ❖S的形式随断言形式的不同而不同
- ❖断言有三种具体形式

断言的种类

❑环境断言

$\Gamma \vdash \Diamond$ 该断言表示Γ是良形的环境

❖将用推理规则来定义环境的语法(而不是用文法)

❑语法断言

$\Gamma \vdash \mathbf{nat}$ 在环境Γ下，nat是类型表达式

❖将用推理规则来定义类型表达式的语法

❑定型断言

$\Gamma \vdash M : T$ 在环境Γ下， M具有类型T

例：$\varnothing \vdash \mathbf{true} : \mathbf{boolean}$ $x : \mathbf{nat} \vdash x+1 : \mathbf{nat}$

❖将用推理规则来确定程序构造实例的类型

断言的有效性

- 有效断言

$$\Gamma \vdash \text{true} : \text{boolean}$$

- 无效断言

$$\Gamma \vdash \text{true} : \text{nat}$$

❑习惯表示法

$$\frac{\Gamma_1 \vdash S_1,\ \ldots,\ \Gamma_n \vdash S_n}{\Gamma \vdash S}$$

- ❖前提、结论
- ❖公理（前提为空）、推理规则

推理规则

（规则名） （注释） 推理规则 （注释）

❑环境规则

(Env ∅)

$$\frac{}{\varnothing \vdash \Diamond}$$

空环境是良形的环境

❑语法规则

(Type Bool)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \text{boolean}}$$

boolean是类型表达式

❑定型规则

(Val +)

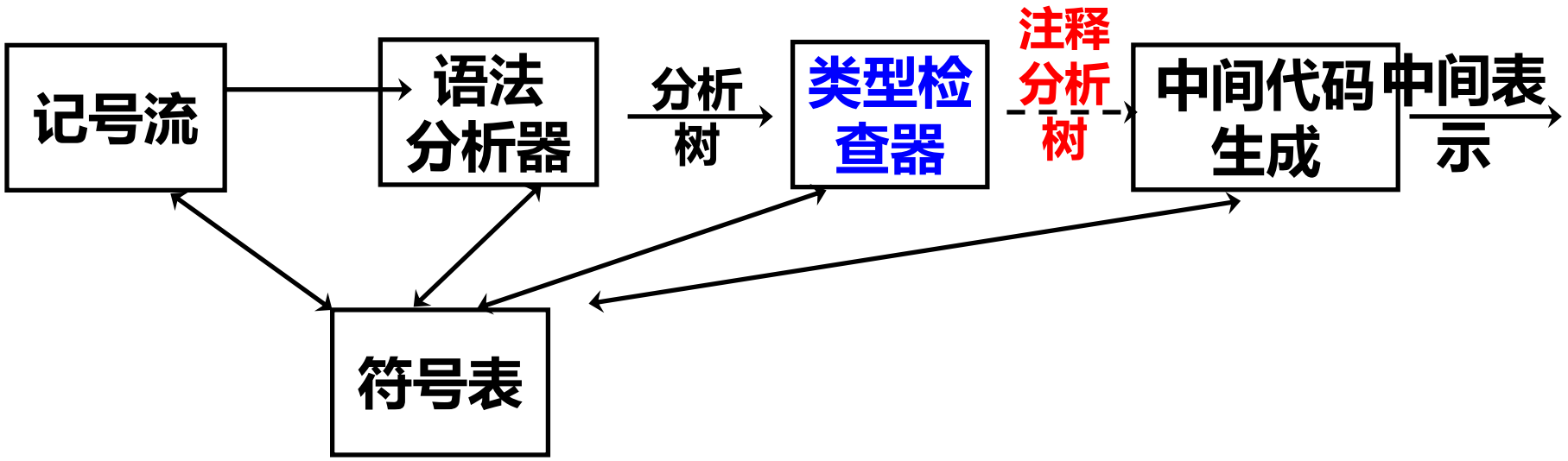
$$\frac{\Gamma \vdash M : \text{int}, \Gamma \vdash N : \text{int}}{\Gamma \vdash M + N : \text{int}}$$

在环境Γ下，M + N是int类型

类型检查与推断

- **类型检查(type checking)**
  - 用语法制导的方式，根据上下文有关的定型规则来判定程序构造是否为良类型的程序构造的过程
  - 可以边解析边检查，也可以在访问AST时进行检查
- **类型推断(type inference)**
  - 类型信息不完全情况下的定型判定问题
  - 例如：f (x : t) = E 和f (x) = E的区别

主要内容



- 静态检查—类型检查
- 描述类型系统的语言
- 简单类型检查器的说明
  - 一个简单的语言及类型系统、语法制导的类型检查
- 类型表达式的等价
- 函数和算符的重载

一个简单的语言

$P \rightarrow D\ ;\ S$

$D \rightarrow D\ ;\ D \mid \text{id} : T$

$T \rightarrow \text{boolean} \mid \text{integer} \mid \text{array [num] of } T \mid$

$\uparrow T \mid T\ \text{‘}\rightarrow\text{’}\ T$

$S \rightarrow \text{id} := E \mid \text{if } E \text{ then } S \mid \text{while } E \text{ do } S \mid S\ ;\ S$

$E \rightarrow \text{truth} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E\ [\ E\ ] \mid$

$E\uparrow \mid E\ (E\ )$

例

```
i : integer;
j : integer;
j := i mod 2000
```

❑环境规则

(Env ∅)

$$\frac{}{\varnothing \vdash \Diamond}$$

dom包含了所有在环境中定义的变量

(Decl Var)

$$\frac{\Gamma \vdash T,\ \mathrm{id} \notin dom\,(\Gamma)}{\Gamma, \mathrm{id} : T \vdash \Diamond}$$

其中id : *T*是该简单语言的一个声明语句

略去了基于程序中所有声明语句来构成整个Γ的规则

❑语法规则

(Type Bool)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textit{boolean}}$$

(Type Int)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textit{integer}}$$

(Type Void)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textit{void}}$$

***void*用于表示语句类型**

表明编程语言和定型断言的类型表达式并非完全一致

❑语法规则

(Type Ref) $(T \neq void)$

$$\frac{\Gamma \vdash T}{\Gamma \vdash pointer(T)}$$

具体语法：$\uparrow T$

(Type Array) $(T \neq void)$

$$\frac{\Gamma \vdash T,\ \Gamma \vdash N : integer}{\Gamma \vdash array(N, T)} (N>0)$$

具体语法：array [N] of T

(Type Function) $(T_1, T_2 \neq void)$

$$\frac{\Gamma \vdash T_1,\ \Gamma \vdash T_2}{\Gamma \vdash T_1 \rightarrow T_2}$$

定型断言中的类型表达式用的是抽象语法

类型系统-- 定型规则

❑定型规则——表达式

(Exp Truth)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \text{truth} : \textit{boolean}}$$

(Exp Num)

$$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \text{num} : \textit{integer}}$$

(Exp Id)

$$\frac{\Gamma_1, \text{id} : T, \Gamma_2 \vdash \Diamond}{\Gamma_1, \text{id} : T, \Gamma_2 \vdash \text{id} : T}$$

❑定型规则——表达式

(Exp Mod)

$$\frac{\Gamma \vdash E_1 : integer,\ \Gamma \vdash E_2 : integer}{\Gamma \vdash E_1 \bmod E_2 : integer}$$

(Exp Index)

$$\frac{\Gamma \vdash E_1 : array(N,T),\ \Gamma \vdash E_2 : integer}{\Gamma \vdash E_1[E_2] : T}$$

$$(0 \le E_2 \le N-1)$$

(Exp Deref)

$$\frac{\Gamma \vdash E : pointer(T)}{\Gamma \vdash E\uparrow : T}$$

定型规则——表达式

(Exp FunCall)

$$\frac{\Gamma \vdash E_1 : T_1 \to T_2, \quad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1\,(E_2) : T_2}$$

定型规则——语句

(State Assign) (*T*=*boolean* or *T*= *integer*)

$$\frac{\Gamma \vdash \mathrm{id} : T,\ \Gamma \vdash E : T}{\Gamma \vdash \mathrm{id} := E : \mathit{void}}$$

(State If)

$$\frac{\Gamma \vdash E : \mathit{boolean},\ \Gamma \vdash S : \mathit{void}}{\Gamma \vdash \mathrm{if}\ E\ \mathrm{then}\ S : \mathit{void}}$$

(State While)

$$\frac{\Gamma \vdash E : \mathit{boolean},\ \Gamma \vdash S : \mathit{void}}{\Gamma \vdash \mathrm{while}\ E\ \mathrm{do}\ S : \mathit{void}}$$

定型规则——语句

(State Seq)

$$\frac{\Gamma \vdash S_1 : void,\ \Gamma \vdash S_2 : void}{\Gamma \vdash S_1; S_2 : void}$$

❑设计语法制导的类型检查器

- ❖设计依据是上节的类型系统
- ❖类型环境Γ的信息进入符号表
- ❖对类型表达式采用抽象语法

  具体：**array** [*N*] **of** *T*　　抽象：*array* (*N*, *T*)

  ↑*T*　　*pointer* (*T*)

- ❖考虑到报错的需要，增加了类型*type_error*

类型检查——声明语句

$D \rightarrow D; D$

$D \rightarrow \text{id} : T \ \{addtype\ (\text{id}.entry, T.type)\}$

addtype：把类型信息填入符号表

类型检查——声明语句

$D \rightarrow D; D$

$D \rightarrow \text{id} : T \ \{addtype\ (\text{id}.entry, T.type)\}$

$T \rightarrow \text{boolean} \quad \{T.type = boolean\}$

$T \rightarrow \text{integer} \quad \{T.type = integer\}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type)\}$

类型检查——声明语句

$D \rightarrow D; D$

$D \rightarrow \text{id} : T \ \{addtype\ (\text{id}.entry, T.type)\}$

$T \rightarrow \text{boolean} \quad \{T.type = boolean\}$

$T \rightarrow \text{integer} \quad \{T.type = integer\}$

$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type)\}$

$T \rightarrow \text{array [num] of } T_1$

$\{T.type = array(\text{num}.val, T_1.type)\}$

类型检查——声明语句

$D \rightarrow D;\ D$

$D \rightarrow \text{id} : T$ {*addtype* (id.*entry*, *T.type*)}

$T \rightarrow \text{boolean}$ {*T.type* = *boolean*}

$T \rightarrow \text{integer}$ {*T.type* = *integer*}

$T \rightarrow \uparrow T_1$ {*T.type* = *pointer*(T_1.*type*)}

$T \rightarrow \text{array [num] of } T_1$

{*T.type* = *array*(num.*val*, T_1.*type*)}

$T \rightarrow T_1$ '$\rightarrow$' T_2 {*T.type* = T_1.*type* $\rightarrow$ T_2.*type* }

类型检查——表达式

$E \rightarrow \text{truth}$ $\{E.type = boolean\}$

$E \rightarrow \text{num}$ $\{E.type = integer\}$

$E \rightarrow \text{id}$ $\{E.type = lookup(\text{id}.entry)\}$

类型检查——表达式

$E \rightarrow$ truth $\quad \{E.type = boolean\}$

$E \rightarrow$ num $\quad \{E.type = integer\}$

$E \rightarrow$ id $\quad \{E.type = lookup(\text{id}.entry)\}$

$E \rightarrow E_1 \bmod E_2$

$\{E.type =$ if $E_1.type == integer$ and

$E_2.type == integer$ then $integer$

else $type\_error\}$

类型检查——表达式

$$E \rightarrow E_1\ [E_2\ ]\ \{E.type = \text{if } E_2.\ type == integer \text{ and } E_1.\ type == array(s,\ t) \text{ then } t \text{ else } type\_error\ \}$$

类型检查——表达式

$$E \rightarrow E_1\,[E_2\,]\ \{E.type = \text{if } E_2.type == integer \text{ and } E_1.type == array(s, t) \text{ then } t \text{ else } type\_error\ \}$$

$$E \rightarrow E_1\uparrow\ \{E.type = \text{if } E_1.type == pointer(t) \text{ then } t \text{ else } type\_error\ \}$$

类型检查——表达式

$E \rightarrow E_1\,[E_2\,]$ {*E.type* = if E_2. *type* == *integer* and
E_1. *type* == *array*(*s*, *t*) then *t*
else *type_error* }

$E \rightarrow E_1\uparrow$ {*E.type* = if E_1.*type* == *pointer*(*t*) then *t*
else *type_error* }

$E \rightarrow E_1\,(E_2\,)$ {*E*. *type* = if E_2 . *type* == *s* and
E_1. *type* == $s \rightarrow t$ then *t*
else *type_error* }

类型检查——语句

$$S \rightarrow \text{id} := E \{ \text{if } (id.type == E.type \;\&\&\; E.type \in \{boolean, integer\}) \; S.type = void;$$

$$\text{else } S.type = type\_error;\}$$

类型检查——语句

$S \rightarrow$ id := E { if ($id.type == E.type$ && $E.type \in$ {*boolean*, *integer*}) $S.type = void$;

else $S.type = type\_error$;}

$S \rightarrow$ if E then S_1 {$S.type$ = if $E.type == boolean$ then $S_1.type$ else $type\_error$ }

$S \rightarrow$ **while** E **do** S_1

$$\{S.type = \text{if } E.type == boolean \text{ then } S_1.type$$

$$\text{else } type\_error \}$$

类型检查——语句

$S \rightarrow$ **while** E **do** S_1

{*S.type* = **if** *E.type* == *boolean* **then** S_1. *type*

else *type_error* }

$S \rightarrow S_1; S_2$

{*S. type* = **if** S_1. *type* == *void* **and**

S_2. *type* == *void* **then** *void*

else *type_error* }

类型检查——程序

$P \rightarrow D; S$

{*P*. *type* = if *S*. *type* == *void* then *void*

else *type_error* }

类型转换

$E \rightarrow E_1$ op E_2

{*E.type* = if E_1.*type* == *integer* and E_2.*type* == *integer*
then *integer*
else if E_1.*type* == *integer* and E_2.*type* == *real*
then *real*
else if E_1.*type* == *real* and E_2.*type* == *integer*
then *real*
else if E_1.*type* == *real* and E_2.*type* == *real*
then *real*
else *type_error* }

例题1

- 考虑C语言中数组double a[10][20]，写出a、a[0]、a[0][0]、&a、&(a[0])、&(a[0][0])的类型表达式

a[0][0]:

a[0]:

a:

&(a[0][0]):

&(a[0]):

&a:

例题1

- 考虑C语言中数组double a[10][20]，写出a、a[0]、a[0][0]、&a、&(a[0])、&(a[0][0])的类型表达式

a[0][0]: double

a[0]: array(20,double);
pointer(double)

a: array(10,array(20,double));
pointer(array(20,double))

&(a[0][0]): pointer(double)

&(a[0]): pointer(array(20,double))

&a: pointer(array(10,array(20,double)))

例题2

- 编译时的控制流检查的例子

```
main() {
printf("\n%ld\n",gcd(4,12));
continue;
}
```

  - 编译时的报错如下:

```
continue.c: In function 'main':
continue.c:3: continue statement not within a loop
```

例题3

❑编译时的唯一性检查的例子

```
main() {
  int i;
  switch(i){
  case 10: printf("%d\n", 10); break;
  case 20: printf("%d\n", 20); break;
  case 10: printf("%d\n", 10); break;
  }
}
```

❖编译时的报错如下：

```
switch.c: In function 'main':
switch.c:6: duplicate case value
switch.c:4: this is the first entry for that value
```

例题4

- C语言
  - 称&为地址运算符，&a为变量a的地址
  - 数组名代表数组第一个元素的地址
- 问题：
  - 如果a是一个数组名，那么表达式a和&a的值都是数组a第一个元素的地址，它们的使用是否有区别？
  - 用四个C文件的编译报错或运行结果来提示

例题4


```
typedef int A[10][20];
    A a;
    A *fun() {
    return(a);
}
```

该函数在Linux上用gcc编译，报告的错误如下:

第5行：warning: return from incompatible pointer type

例题4


```
typedef int A[10][20];
  A a;
  A *fun() {
  return(&a);
}
```

该函数在Linux上用gcc编译时，没有错误

例题4


```
typedef int A[10][20];

typedef int B[20];

  A a;

  B *fun() {

  return(a);

}
```

该函数在Linux上用gcc编译时，没有错误

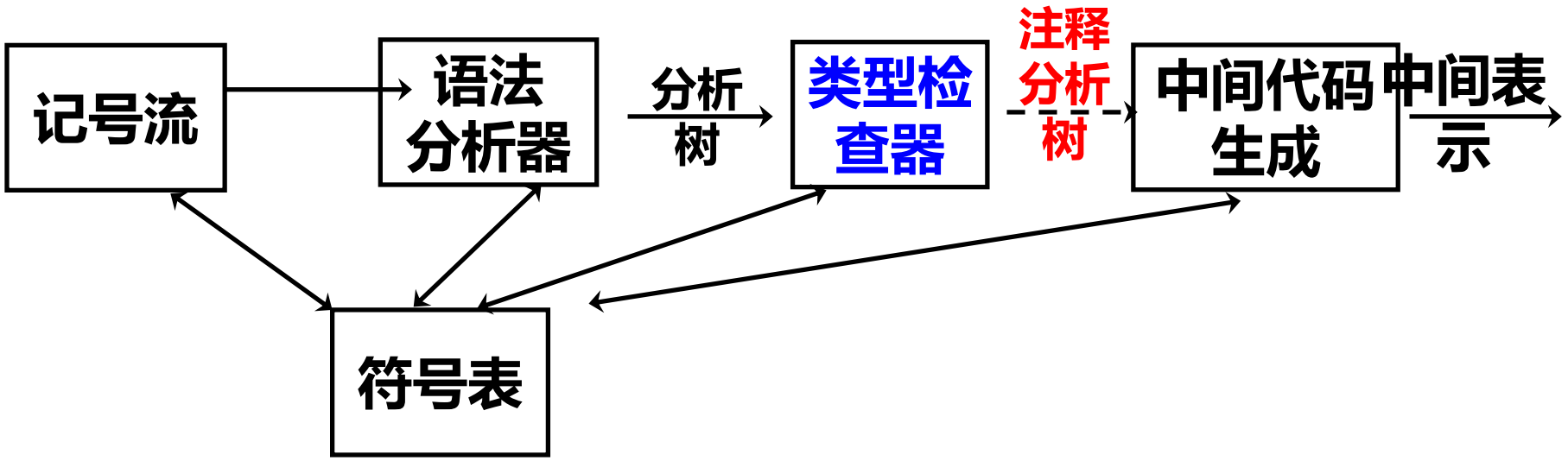
例题4


```
typedef int A[10][20];
  A a;
fun() { printf("%d,%d,%d\n", a, a+1, &a+1);}
main() {
  fun();
}
```

该程序的运行结果是：

134518112, 134518192, 134518912

主要内容



- 静态检查—类型检查
- 描述类型系统的语言
- 简单类型检查器的说明
  - 一个简单的语言及类型系统、语法制导的类型检查
- 类型表达式的等价
- 函数和算符的重载

类型表达式的等价

当允许对类型表达式命名后：

- 类型表达式是否相同就有了不同的解释
- 出现了结构等价和名字等价两个不同的概念

```
type link = ↑cell;
var     next   : link;
        last   : link;
        p      : ↑cell;
        q, r   : ↑cell;
```

类型表达式的等价

- 两个类型表达式完全相同（当无类型名时）

```
type link = ↑cell;
var    next  : link;
       last  : link;
       p     : ↑cell;
       q, r  : ↑cell;
```

类型表达式的等价

- 两个类型表达式完全相同（当无类型名时）
  - 类型表达式树一样

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

类型表达式的等价

- 两个类型表达式完全相同（当无类型名时）
  - 类型表达式树一样
  - 相同的类型构造符作用于相同的子表达式

```
type link = ↑cell;

var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

类型表达式的等价

- 两个类型表达式完全相同（当无类型名时）
- 有类型名时，用它们所定义的类型表达式代换它们，所得表达式完全相同（类型定义无环时）

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

↑cell

↑cell

next, last, p, q和r结构等价

类型表达式的等价

类型表达式的结构等价测试sequiv(s, t) (无类型名时)

if s 和 t 是相同的基本类型then
  return true
else if s == *array*(s_1, s_2) and t == *array*(t_1, t_2) then
  return sequiv(s_1, t_1) and sequiv(s_2, t_2)
else if s == $s_1 \times s_2$ and t == $t_1 \times t_2$ then
  return sequiv(s_1, t_1) and sequiv(s_2, t_2)
else if s == *pointer* (s_1) and t == *pointer*(t_1) then
  return sequiv(s_1, t_1)
else if s == $s_1 \rightarrow s_2$ and t == $t_1 \rightarrow t_2$ then
  return squiv(s_1, t_1) and sequiv(s_2, t_2)
else return false

类型表达式的等价

- 把每个类型名看成是一个可区别的类型

```
type link = ↑cell;
var    next  : link;
       last  : link;
       p     : ↑cell;
       q, r  : ↑cell;
```

类型表达式的等价

- 把每个类型名看成是一个可区别的类型
- 两个类型表达式名字等价当且仅当这两个类型表达式不进行名字代换就能结构等价

```
type link = ↑cell;
var   next  : link;
      last  : link;
      p     : ↑cell;
      q, r  : ↑cell;
```

next和last名字等价

p，q和r名字等价

类型表达式的等价

Pascal的许多实现用隐含的类型名和每个声明的标识符联系起来

```
type link = ↑cell;
var  next : link;
     last : link;
     p    : ↑cell;
     q, r : ↑cell;
```

```
type link = ↑cell;
     np = ↑cell;
     nqr = ↑cell;
var next : link;
     last : link;
     p : np;
     q : nqr;
     r : nqr;
```

这时，p与q和r不是名字等价

类型表达式的等价

中国科学技术大学
University of Science and Technology of China

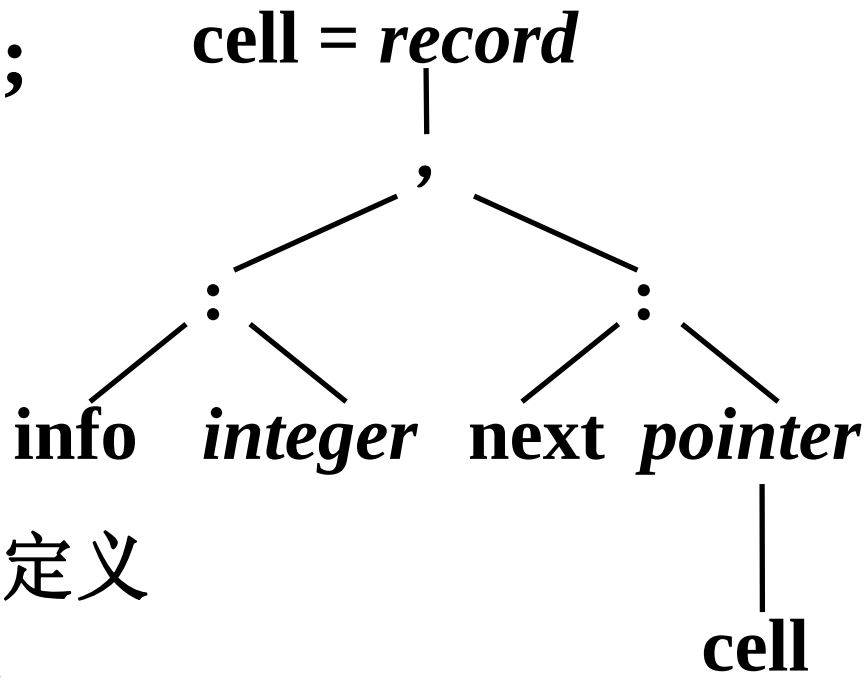
注意：

类型名字的引入只是类型表达式的一个语法约定问题，它并不像引入类型构造符或类型变量那样能丰富所能表达的类型

类型表达式的等价

❑类型表示中的环

```
type link = ↑ cell ;
cell = record
        info : integer ;
        next : link
end;
```

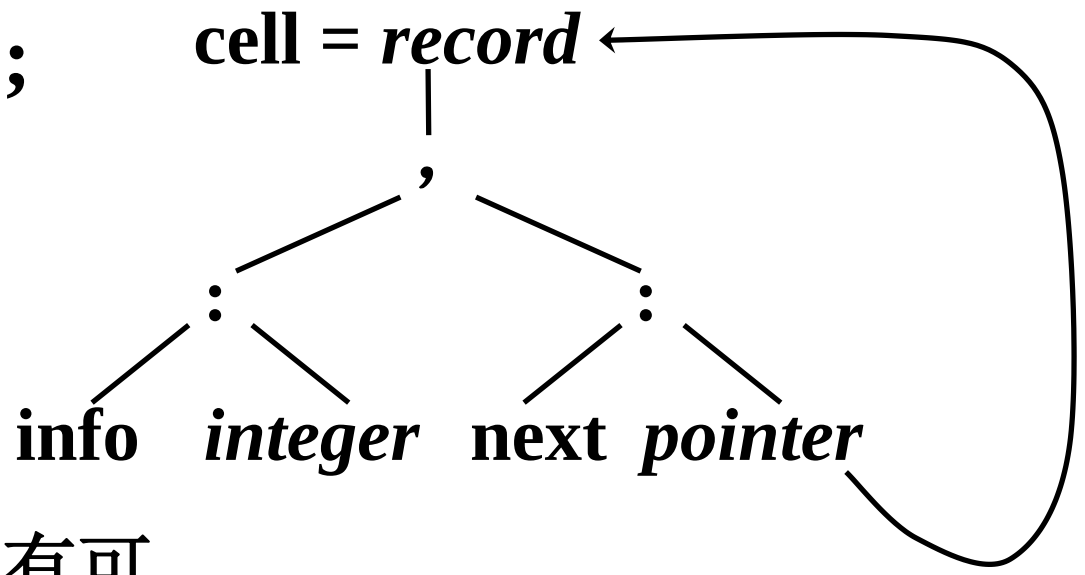


引入环的话，递归定义的类型名可以替换掉

类型表达式的等价

❑类型表示中的环

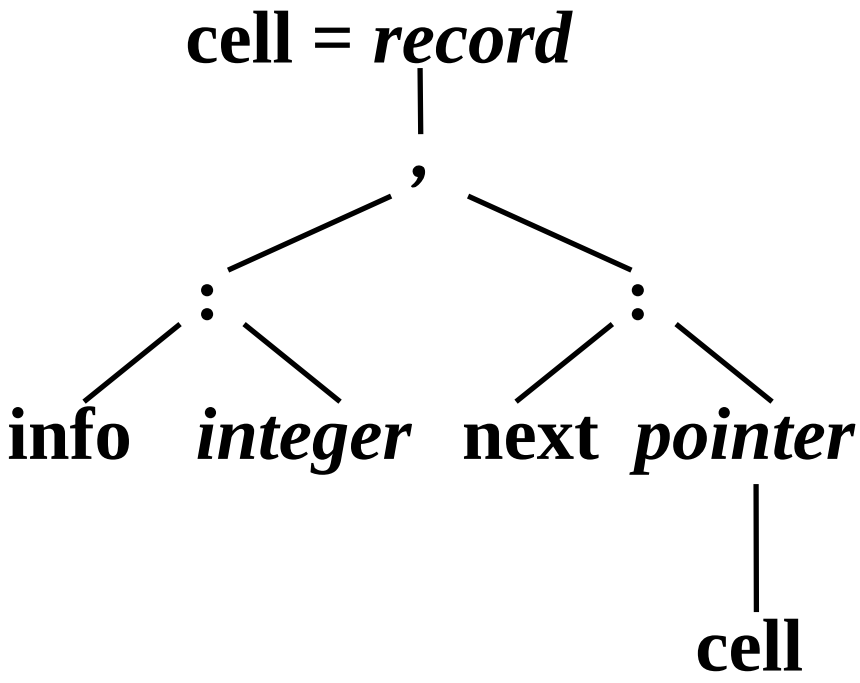
```
type link = ↑cell ;
cell = record
        info : integer ;
        next : link
end;
```



结构等价测试过程有可能不终止

类型表达式的等价

C语言对除记录（结构体）以外的所有类型使用结构等价，而记录类型用的是名字等价，以避免类型图中的环



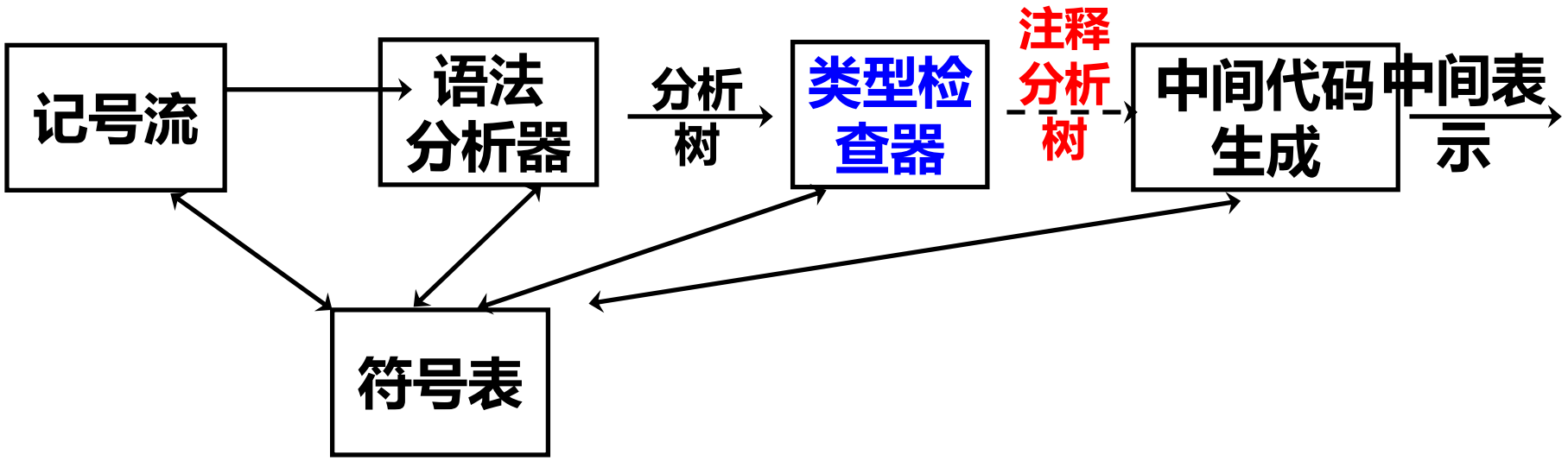
在X86/Linux机器上，编译器报告最后一行有错误：

incompatible types in return

```
typedef int A1[10];                | A2 *fun1( ) {
typedef int A2[10];                |     return(&a);
A1 a;                              | }
typedef struct {int i;}S1;         | S2  fun2( ) {
typedef struct {int i;}S2;         |     return(s);
S1 s;                              | }
```

在C语言中，数组和结构体都是构造类型，为什么上面第2个函数有类型错误，而第1个函数却没有?

主要内容



- 静态检查—类型检查
- 描述类型系统的语言
- 简单类型检查器的说明
  - 一个简单的语言及类型系统、语法制导的类型检查
- 类型表达式的等价
- **函数和算符的重载**

函数和算符的重载

❑重载符号

- ❖有多个含义，但在每个引用点的含义都是唯一的

❑例如：

- ❖加法算符+可用于不同类型，"+"是多个函数的名字，而不是一个多态函数的名字
- ❖在Ada中，( ) 是重载的，A(I) 有不同含义

❑重载的消除

- ❖在重载符号的引用点，其含义能确定到唯一

子表达式的可能类型集合

❑例 Ada语言的声明:

function “*” (i, j : integer) return complex;

function “*” (x, y : complex) return complex;

使得算符*重载，可能的类型包括:

integer × integer → integer --这是预定义的类型

integer × integer → complex

complex × complex → complex

2 * (3 * 5)

(3 * 5) * z z是复型

❑以函数应用为例，考虑类型检查

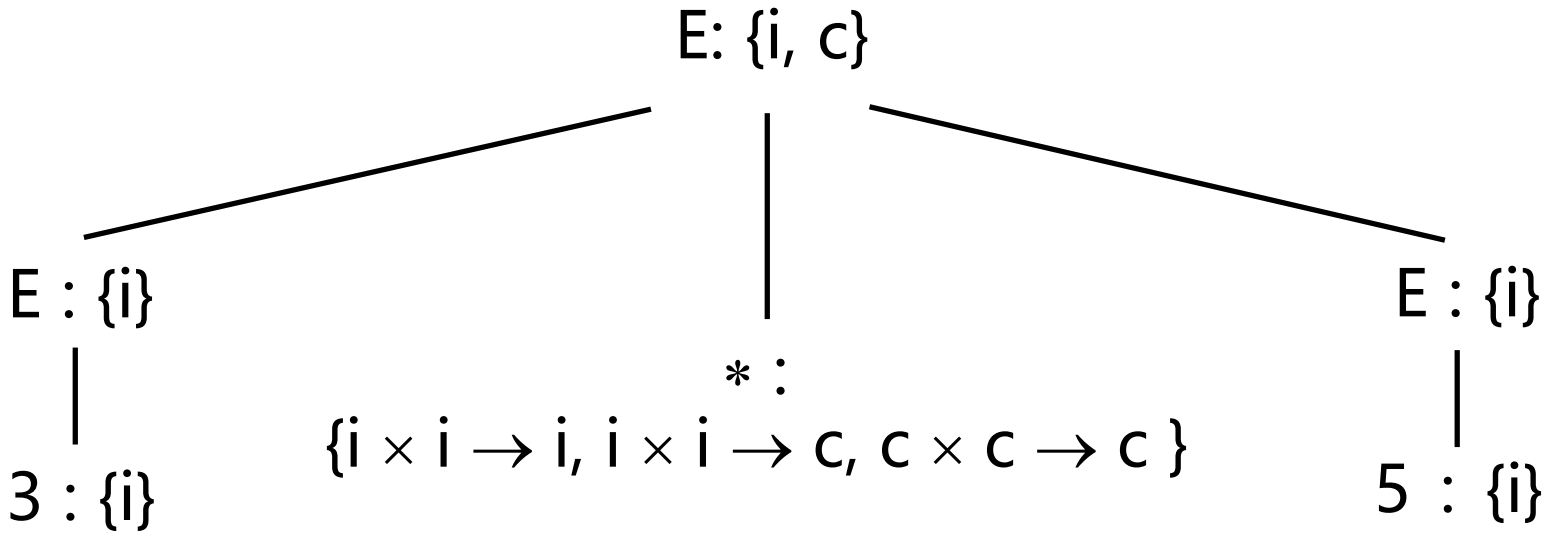
❖在每个表达式都有唯一的类型时，函数应用的类型检查是：

E → $E_1(E_2)$ { E.type = if E_2.type == s and E_1.type == s → t then t else type_ error }

❖确定表达式可能类型的集合（类型可能不唯一）

产 生 式	语 义 规 则
E′ → E	E′.types = E. types
E → id	E. types = lookup(id. entry)
E → E_1 (E_2)	E. types = {t | E_2. types中存在一个s，使得s → t属于E_1.types }

- 例：表达式3 * 5可能的类型集合



中国科学技术大学
University of Science and Technology of China

USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
类型检查

The purpose of computing is insight, not numbers.

—— *Richard Hamming (Turing award, 1968)*