语义分析
(3. 符号表与类型检查)

魏恒峰

hfwei@nju.edu.cn

2021 年 12 月 14 日

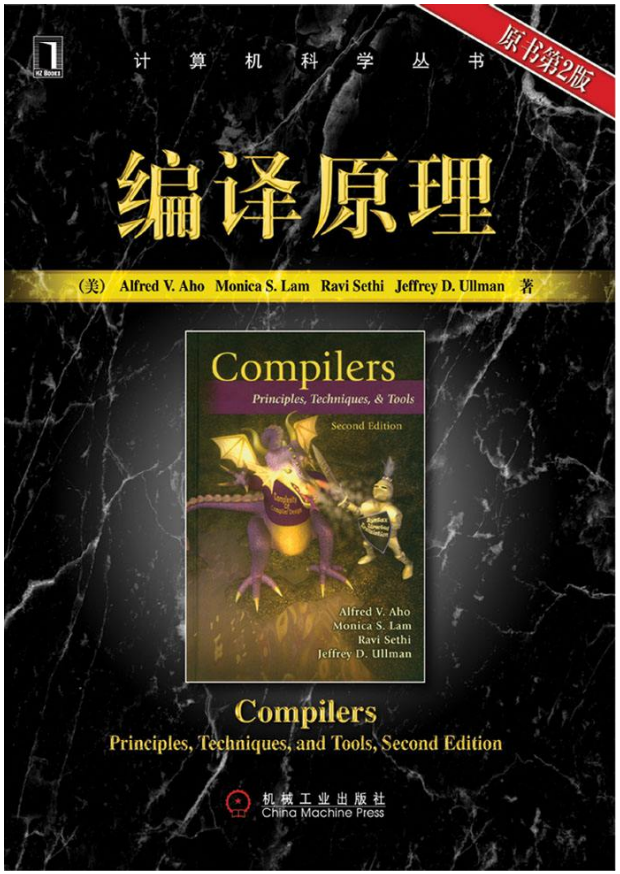
语义分析

语法制导的定义 (SDD)

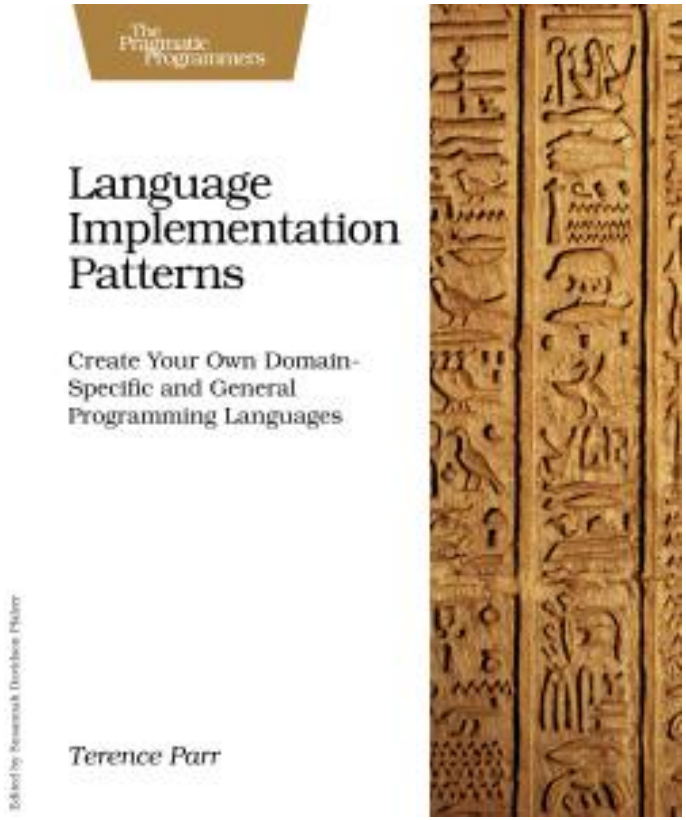
语法制导的翻译 (SDT)

符号表与类型检查

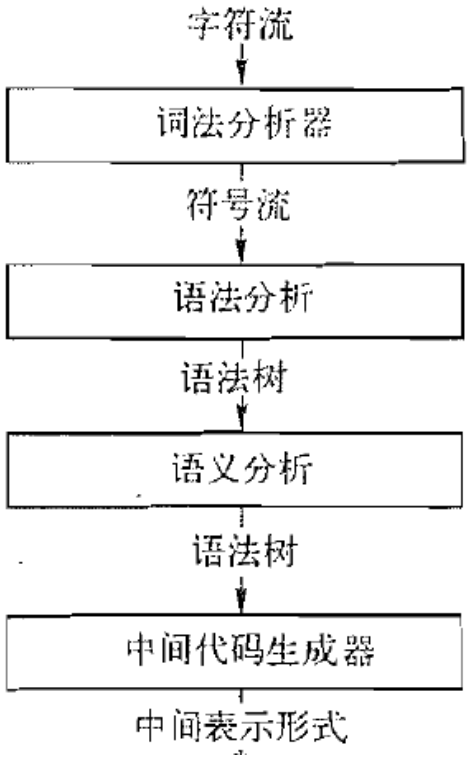
Another big difference is that we discourage the use of actions directly within the grammar because ANTLR 4 automatically generates listeners and visitors for you to use that trigger method calls when some phrases of interest are recognized during a tree walk after parsing. See also Parse Tree Matching and XPath.



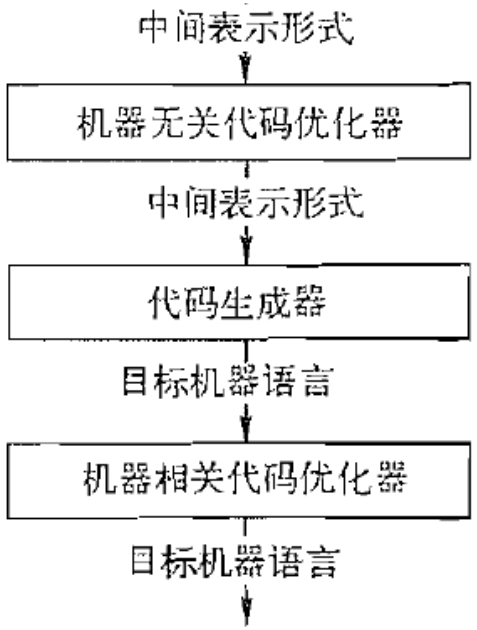
属性文法: 语义动作嵌入到文法中



ANTLR4: 将语义动作与文法分离


字符流
词法分析器
符号流
语法分析
语法树
语义分析
语法树
中间代码生成器
中间表示形式

符号表


中间表示形式
机器无关代码优化器
中间表示形式
代码生成器
目标机器语言
机器相关代码优化器
目标机器语言

Definition (符号表 (Symbol Table))

符号表是用于保存各种信息的**数据结构**。

标识符: 词素、类型、大小、存储位置等

Definition (符号表 (Symbol Table))

符号表是用于保存各种信息的**数据结构**。

标识符: 词素、类型、大小、存储位置等

```
public class ST<Key extends Comparable<Key>, Value>
```

	ST()	*create an empty symbol table*
void	put(Key key, Value val)	*associate val with key*
Value	get(Key key)	*value associated with key*
void	remove(Key key)	*remove key (and its associated value)*
boolean	contains(Key key)	*is there a value associated with key?*
int	size()	*number of key–value pairs*
Iterable<Key>	keys()	*all keys in the symbol table*

可使用 HashTable 或 Red-Black Tree

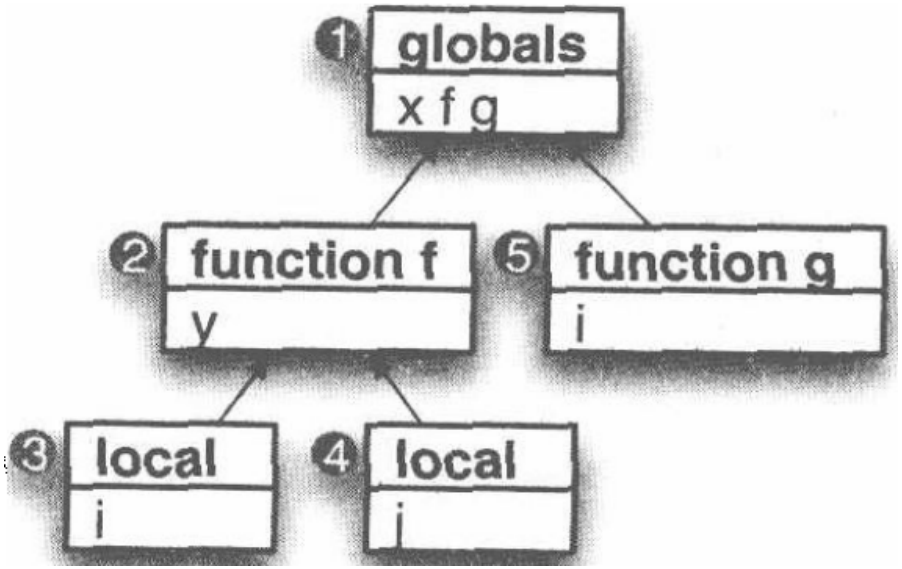
“领域特定语言” (DSL) 通常只有**单作用域** (全局作用域)

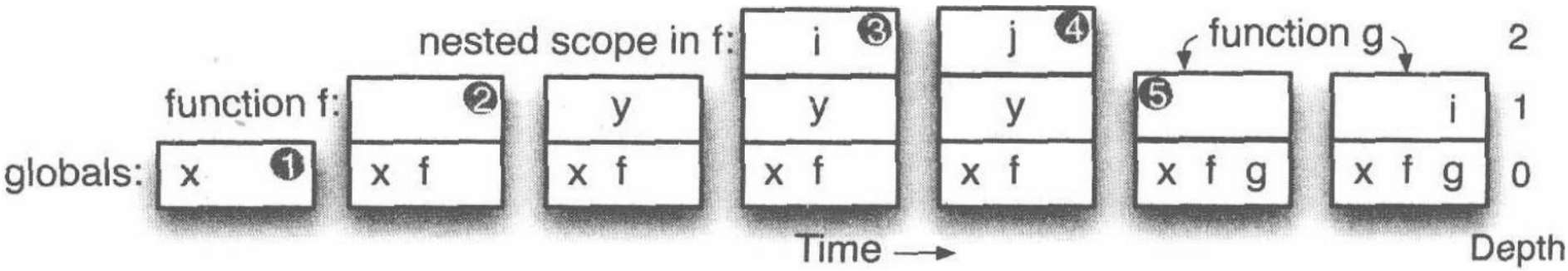
```
host=antlr.org
port=80
webmaster=parrt@antlr.org
```

“通用程序设计语言” (GPL) 通常需要**嵌套作用域**

```
❶   //开始全局作用域
    int x;              //定义全局作用域中的变量x
❷   void f(){           //定义全局作用域中的函数f
      int y;            //定义f作用域中的变量y
❸     { int i; }        //在嵌套作用域中定出变量i
❹     { int j; }        //在嵌套作用域中定出变量j
    }
❺   void g() {          //定义全局作用域中的函数g
      int i;            //定义g作用域中的变量i
    }
```

```
❶
      int x;
❷     void f(){
        int y;
❸       { int i; }
❹       { int j; }
      }
❺     void g() {
        int i;
      }
```





```
1)  package symbols;                                  // 文件 Env.java
2)  import java.util.*;
3)  public class Env {
4)      private Hashtable table;
5)      protected Env prev;

6)      public Env(Env p) {
7)          table = new Hashtable(); prev = p;
8)      }

9)      public void put(String s, Symbol sym) {
10)         table.put(s, sym);
11)     }

12)     public Symbol get(String s) {
13)         for( Env e = this; e != null; e = e.prev ) {
14)             Symbol found = (Symbol)(e.table.get(s));
15)             if( found != null ) return found;
16)         }
17)         return null;
18)     }
19) }
```

创建新的，旧的“压栈”

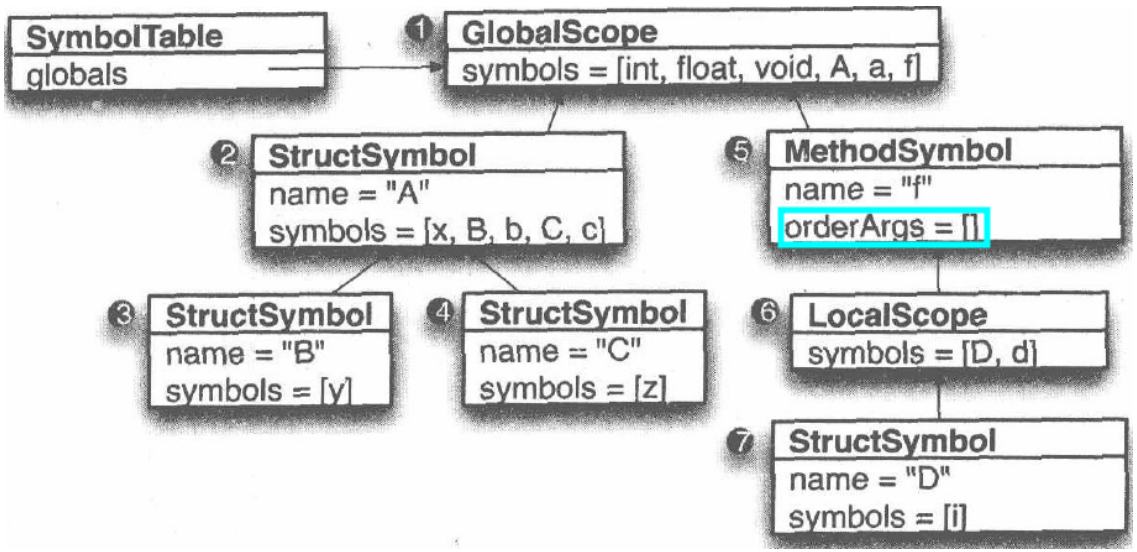
放入当前符号表

倒序搜索符号表链

struct: 类型作用域

```
❶
❷  struct A {
     int x;
❸    struct B { int y; };
     B b;
❹    struct C {int z; };
     C c;
   };
   A a;

❺  void f()
❻  {
❼    struct D {
       int i;
     };
     D d;
     d.i = a.b.y;
   }
```



$d.i$ $a.b.y$

声明: 添加符号与作用域 (“def”)

使用: 解析符号 (“ref”)

先声明后使用

```
{ int x; char y; { bool y; x; y; } x; y; }
```

```
{ int x; char y; { bool y; x; y; } x; y; }
```

翻译任务: 解析每个标识符 (引用处) 的**类型**

```
{ { x:int; y:bool; } x:int; y:char; }
```

Definition (类型表达式 (Type Expressions))

- **基本类型**是类型表达式;
  - char, bool, int, float, double, void, ...
- **类名**是类型表达式;

Definition (类型表达式 (Type Expressions))

- **基本类型**是类型表达式;
  - char, bool, int, float, double, void, ...
- **类名**是类型表达式;
- 如果 t 是类型表达式, 则 **array(num, t)** 是类型表达式;
- **record(⟨id : t, ...⟩)** 是类型表达式 (C 结构体);

Definition (类型表达式 (Type Expressions))

- **基本类型**是类型表达式;
  - char, bool, int, float, double, void, ...
- **类名**是类型表达式;
- 如果 t 是类型表达式, 则 **array(num, t)** 是类型表达式;
- **record(⟨id : t, ...⟩)** 是类型表达式 (C 结构体);
- 如果 s 和 t 是类型表达式, 则 $s \times t$ 是类型表达式;
- 如果 s 和 t 是类型表达式, 则 $s \to t$ 是类型表达式;

```
program → block                                  { top = null; }

block   → '{'                                    { saved = top;
                                                  top = new Env(top);
                                                  print("{ "); }
          decls stmts '}'                        { top = saved;
                                                  print("} "); }

decls   → decls decl
        | ϵ

decl    → type id ;                              { s = new Symbol;
                                                  s.type = type.lexeme;
                                                  top.put(id.lexeme, s); }

stmts   → stmts stmt
        | ϵ

stmt    → block
        | factor ;                               { print("; "); }

factor  → id                                     { s = top.get(id.lexeme);
                                                  print(id.lexeme);
                                                  print(":");
                                                  print(s.type);}
```

```
{ char y; { bool y; y; } y; }
```

```
{ { y : bool; } y : char; }
```

program	→	*block*	{ *top* = null; }
block	→	'{'	{ *saved* = *top*; *top* = **new** *Env*(*top*); print("{ "); }
		decls stmts '}'	{ *top* = *saved*; print("} "); }
decls	→	*decls decl*	
	|	ϵ	
decl	→	**type id** ;	{ *s* = **new** *Symbol*; *s.type* = **type**.*lexeme*; *top*.*put*(**id**.*lexeme*, *s*); }
stmts	→	*stmts stmt*	
	|	ϵ	
stmt	→	*block*	
	|	*factor* ;	{ print("; "); }
factor	→	**id**	{ *s* = *top*.*get*(**id**.*lexeme*); print(**id**.*lexeme*); print(":"); print(*s.type*);}

局部变量 saved

类型声明

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

符号表中记录标识符的类型、宽度 (width)、偏移地址 (offset)

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ \text{'}\{\text{'}\ D\ \text{'}\}\text{'} \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

需要为每个 **record** 生成单独的符号表

全局变量 t 与 w 将 B 的**类型与宽度**信息传递给产生式 $C \to \epsilon$

(在语法制导定义中, t 与 w 是 C 的**继承属性**)

$T \to B$		$\{ t = B.type; w = B.width; \}$
	C	$\{ T.type = C.type; T.width = C.width \}$
$B \to$	**int**	$\{ B.type = integer; B.width = 4; \}$
$B \to$	**float**	$\{ B.type = float; B.width = 8; \}$
$C \to$	ϵ	$\{ C.type = t; C.width = w; \}$
$C \to$	[**num**] C_1	$\{ C.type = array(\mathbf{num}.value, C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

图 6-15　计算类型及其宽度

`float x`：目前仅关心类型声明 (`float`)

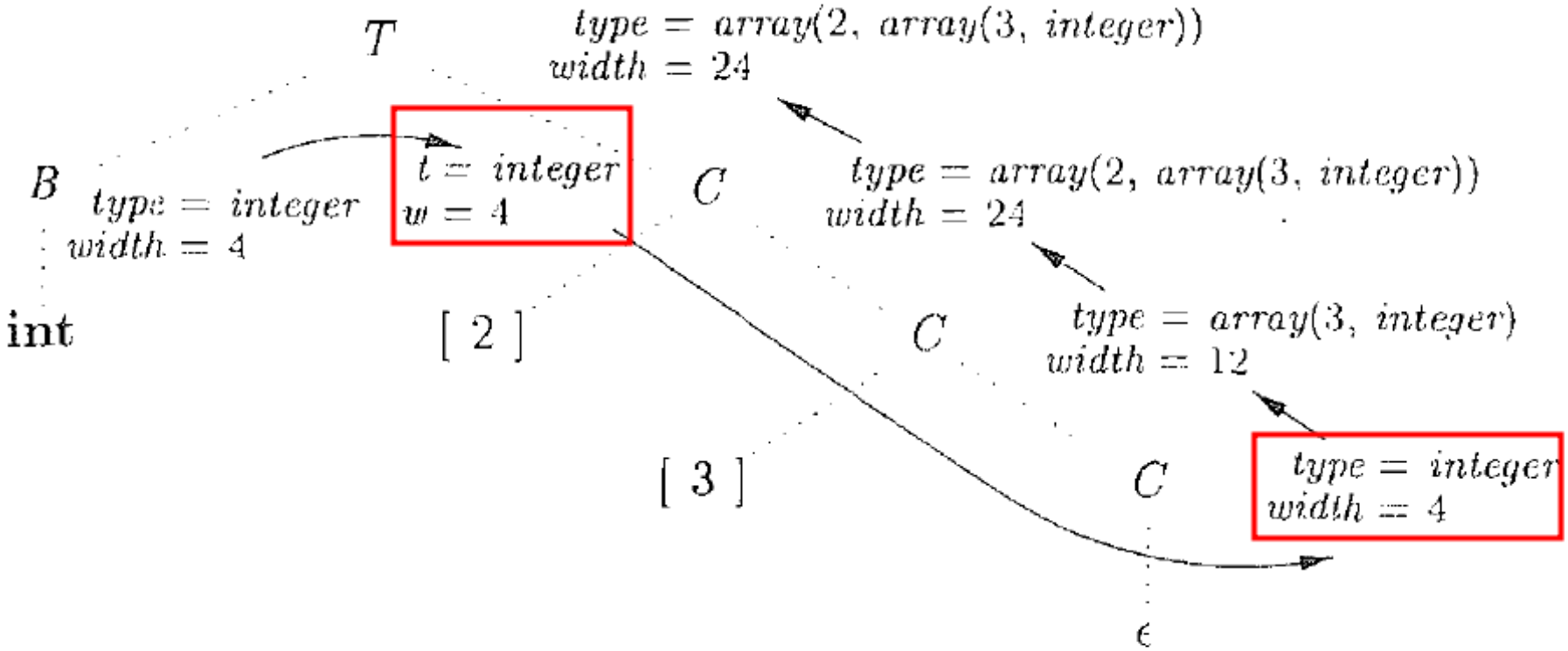


图 6-16　数组类型的语法制导翻译

```
int[2][3]
```

全局变量 offset 表示变量的相对地址

全局变量 top 表示当前的符号表

P	$\rightarrow$	M	$\{\ offset = 0;\ \}$
		D	
D	$\rightarrow$	T **id** ;	$\{\ top.put(\mathbf{id}.lexeme,\ T.type,\ offset);$
			$offset = offset + T.width;\ \}$
		D_1	
D	$\rightarrow$	ϵ	

图 6-17　计算被声明变量的相对地址

```
float x; float y;
```

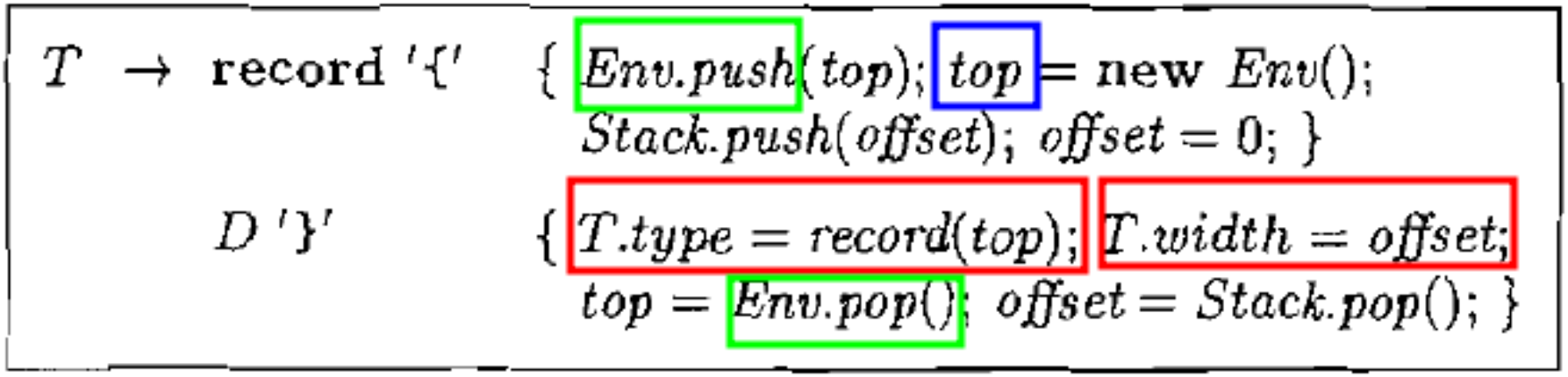


图 6-18　处理记录中的字段名

$$
\begin{array}{lll}
T \rightarrow \textbf{record}\ '\{' & \{\ \mathit{Env.push}(\mathit{top});\ \mathit{top} = \textbf{new}\ \mathit{Env}(); & \\
 & \mathit{Stack.push}(\mathit{offset});\ \mathit{offset} = 0;\ \} & \\
\quad D\ '\}' & \{\ \mathit{T.type} = \mathit{record}(\mathit{top});\ \mathit{T.width} = \mathit{offset}; & \\
 & \mathit{top} = \mathit{Env.pop}();\ \mathit{offset} = \mathit{Stack.pop}();\ \} &
\end{array}
$$

图 6-18　处理记录中的字段名

record 类型表达式: record(top)

字段名的偏移量是相对地址

$$
\begin{array}{lll}
T \rightarrow \textbf{record}\ \text{'\{'} & \{\ \mathit{Env.push}(\mathit{top});\ \mathit{top} = \textbf{new}\ \mathit{Env}(); & \\
 & \mathit{Stack.push}(\mathit{offset});\ \mathit{offset} = 0;\ \} & \\
D\ \text{'\}'} & \{\ \mathit{T.type} = \mathit{record}(\mathit{top});\ \mathit{T.width} = \mathit{offset}; & \\
 & \mathit{top} = \mathit{Env.pop}();\ \mathit{offset} = \mathit{Stack.pop}();\ \} &
\end{array}
$$

图 6-18　处理记录中的字段名

record 类型表达式：record(top)

字段名的偏移量是相对地址

全局变量 top 表示当前的符号表

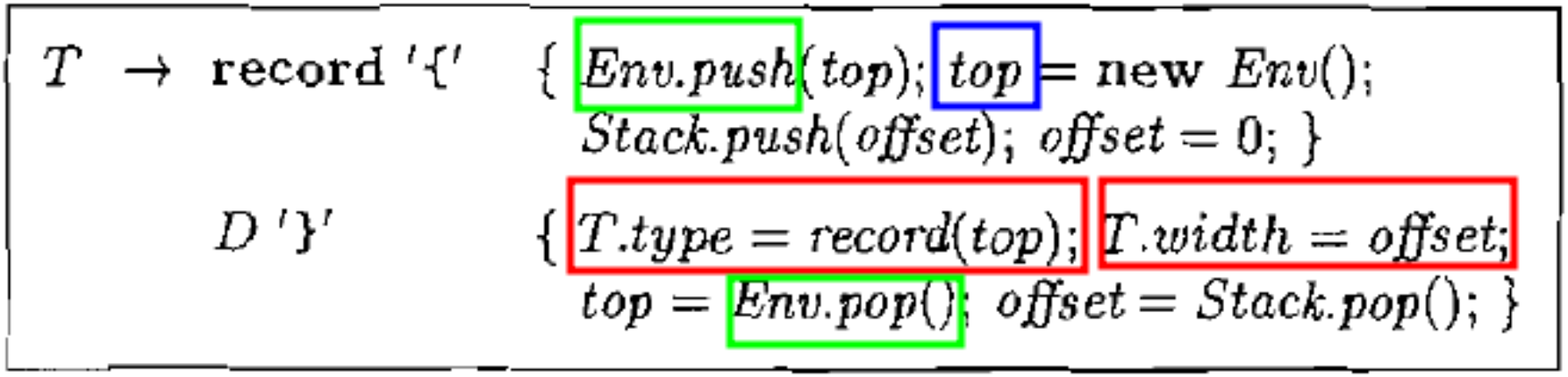


图 6-18　处理记录中的字段名

record 类型表达式: record(top)

字段名的偏移量是相对地址

全局变量 top 表示当前的符号表

全局变量 Env 表示符号表栈

$$
\begin{array}{lll}
T \rightarrow \textbf{record}\ \text{'\{'} & \{\ \mathit{Env.push}(\mathit{top});\ \mathit{top} = \textbf{new}\ \mathit{Env}(); & \\
 & \mathit{Stack.push}(\mathit{offset});\ \mathit{offset} = 0;\ \} & \\
D\ \text{'\}'} & \{\ \mathit{T.type} = \mathit{record}(\mathit{top});\ \mathit{T.width} = \mathit{offset}; & \\
 & \mathit{top} = \mathit{Env.pop}();\ \mathit{offset} = \mathit{Stack.pop}();\ \} &
\end{array}
$$

图 6-18　处理记录中的字段名

record 类型表达式: record(top)

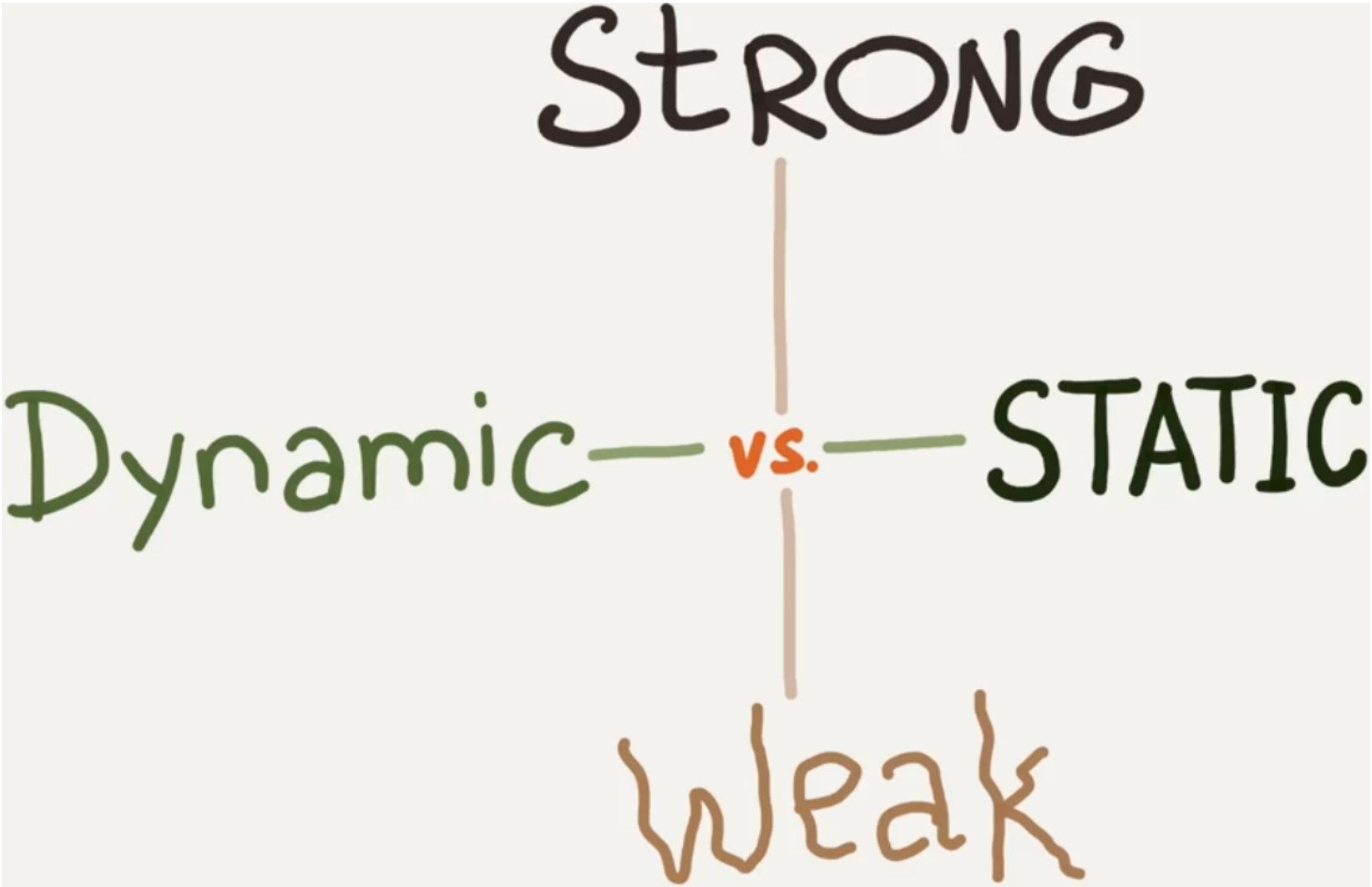
字段名的偏移量是相对地址

全局变量 top 表示当前的符号表

全局变量 Env 表示符号表栈

```
record { float x; float y; } p;
```

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```



https://youtu.be/C5fr0LZLMAs

类型检查的常见形式

if two type expressions are equivalent
then return a given type
else return **type_error**

```
typedef struct {
        int data[100];
        int count;
        } Stack;

typedef struct {
        int data[100];
        int count;
        } Set;

Stack x, y;
Set r, s;
```

Definition (结构等价 (Structurally Equivalent))

两种类型**结构等价**当且仅当以下任一条件为真:

- 它们是相同的基本类型;
- 它们是将相同的类型构造算子应用于结构等价的类型而构造得到;
- **一个类型是另一个类型表达式的名字**。

Definition (结构等价 (Structurally Equivalent))

两种类型**结构等价**当且仅当以下任一条件为真:

- 它们是相同的基本类型;
- 它们是将相同的类型构造算子应用于结构等价的类型而构造得到;
- **一个类型是另一个类型表达式的名字**。

Definition (名等价 (Name Equivalent))

两种类型**名等价**当且仅当以下任一条件为真:

- 它们是相同的基本类型;
- 它们是将相同的类型构造算子应用于结构等价的类型而构造得到。

结构等价中的“结构”又是什么意思?

$$array(n, t) \qquad array(m, t)$$

结构等价中的“结构”又是什么意思?

$$array(n, t) \qquad array(m, t)$$

```
type R3 = record
    a : integer;
    b : integer
end;
```

```
type R4 = record
    b : integer;
    a : integer
end;
```

结构等价中的“结构”又是什么意思?

$$array(n, t) \qquad array(m, t)$$

```
type R3 = record
     a : integer;
     b : integer
end;
```

```
type R4 = record
     b : integer;
     a : integer
end;
```

不同的语言有不同的设计方案

类型综合: 根据子表达式的类型确定表达式的类型

if f 的类型为 $s \to t$ 且 x 的类型为 s

then 表达式 $f(x)$ 的类型为 t

$$E_1 + E_2$$

```
void err() { ... }
void err(String s) { ... }
```

重载函数的类型综合规则

if f 可能的类型为 $s_i \to t_i (1 \leqslant i \leqslant n)$，其中，$s_i \neq s_j (i \neq j)$

and x 的类型为 $s_k (1 \leqslant k \leqslant n)$

then 表达式 $f(x)$ 的类型为 t_k

类型推导: 根据某语言结构的使用方式确定表达式的类型

if $f(x)$是一个表达式，

then 对某些 α 和 β，f 的类型为 $\alpha \rightarrow \beta$ 且 x 的类型为 α

$null(x)$: x 是一个列表, 它的元素类型未知

类型转换

```
t1 = (float) 2
t2 = t1 * 3.14
```

类型转换

```
t1 = (float) 2
t2 = t1 * 3.14
```

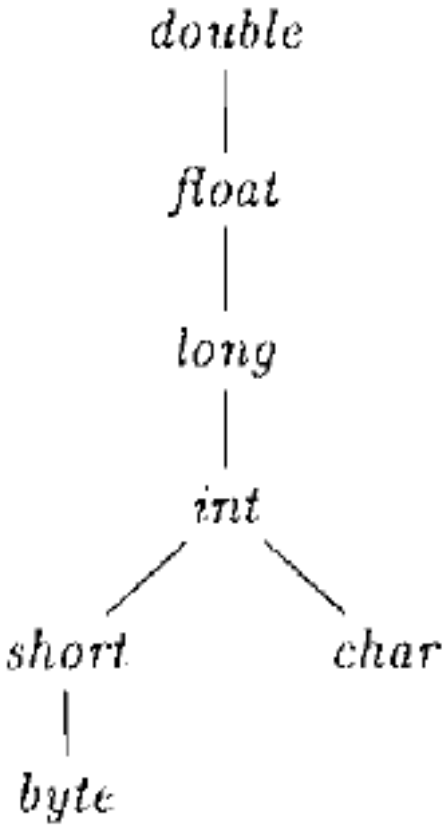
if ($E_1.type = integer$ **and** $E_2.type = integer$) $E.type = integer$;
else if ($E_1.type = float$ **and** $E_2.type = integer$) $\cdots$
$\cdots$

类型转换

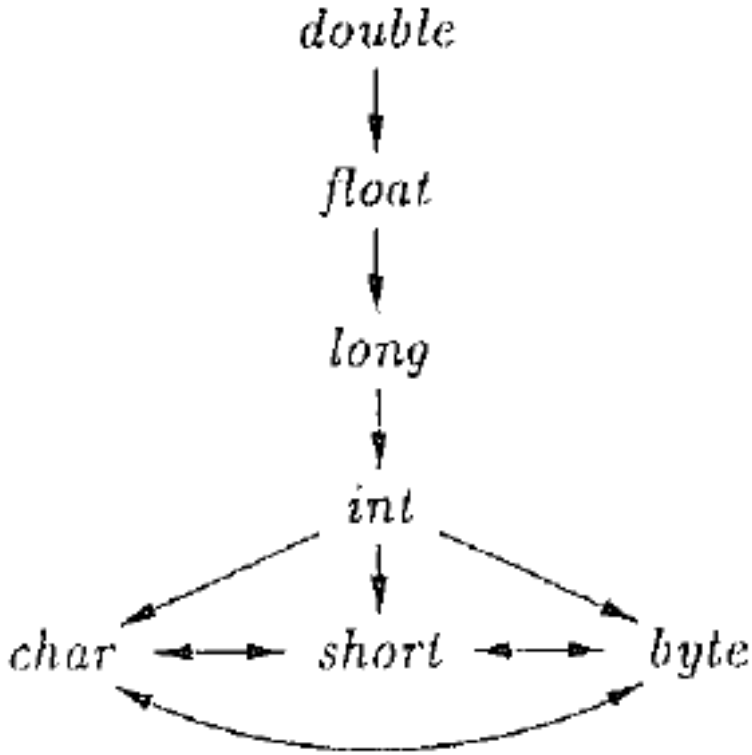
```
t1 = (float) 2
t2 = t1 * 3.14
```

if ($E_1.type = integer$ and $E_2.type = integer$) $E.type = integer$;
else if ($E_1.type = float$ and $E_2.type = integer$) ...
...

不要写这样的代码!!!



a) 拓宽类型转换



b) 窄化类型转换

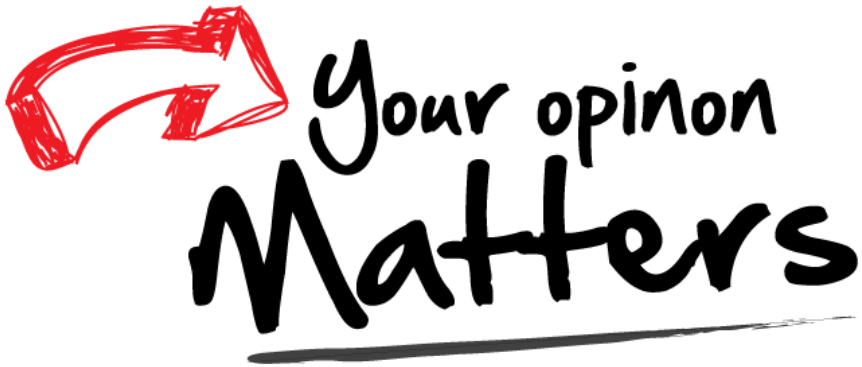
```
E → E1 + E2   { E.type = max(E1.type, E2.type);
                a1 = widen(E1.addr, E1.type, E.type);
                a2 = widen(E2.addr, E2.type, E.type);
                E.addr = new Temp();
                gen(E.addr '=' a1 '+' a2); }
```

图 6-27　在表达式求值中引入类型转换

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

https://youtu.be/SWTWkYbcWUO


Thank
You!



Office 926

hfwei@nju.edu.cn