编译原理与设计

北京理工大学 计算机学院

语义分析：概览

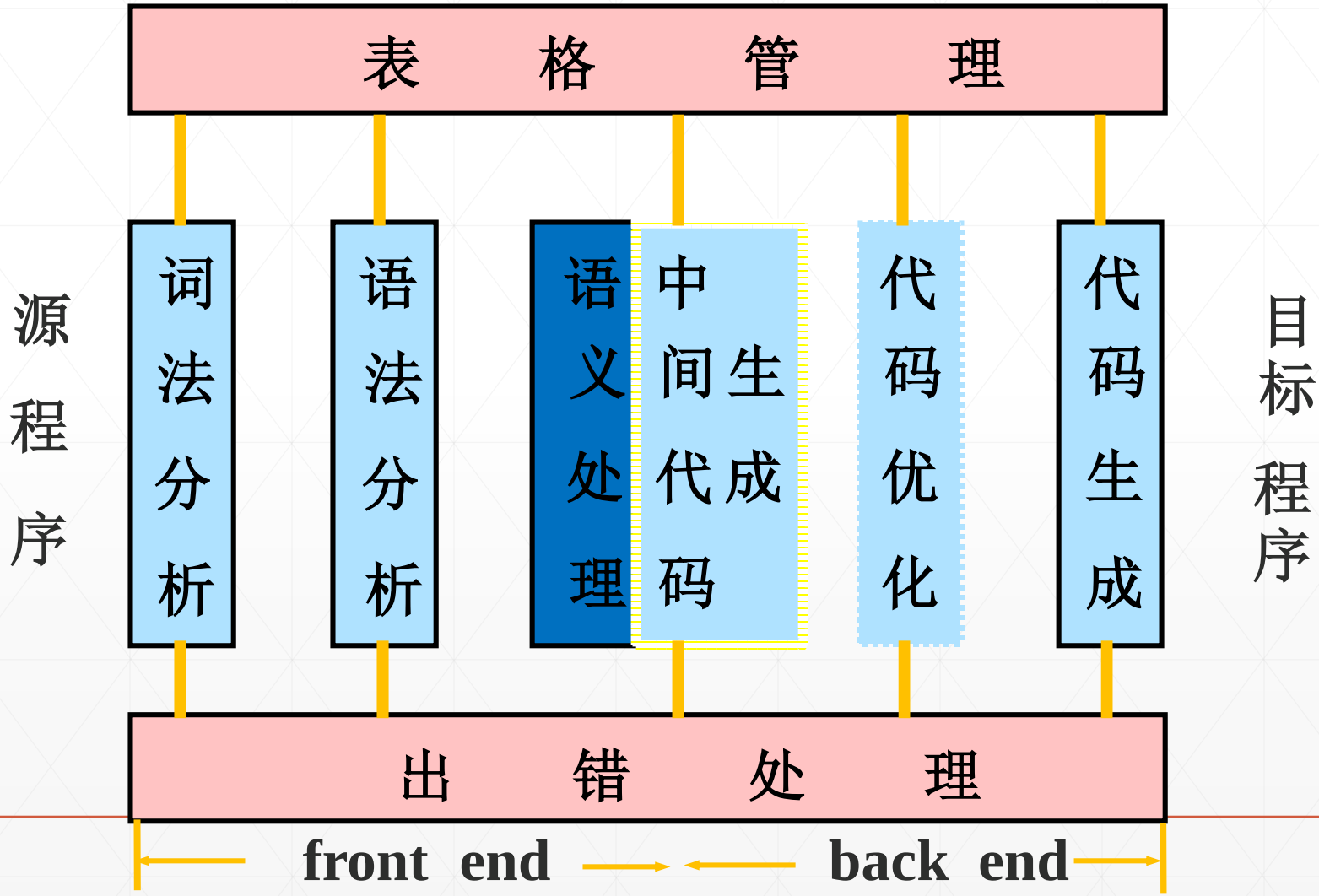
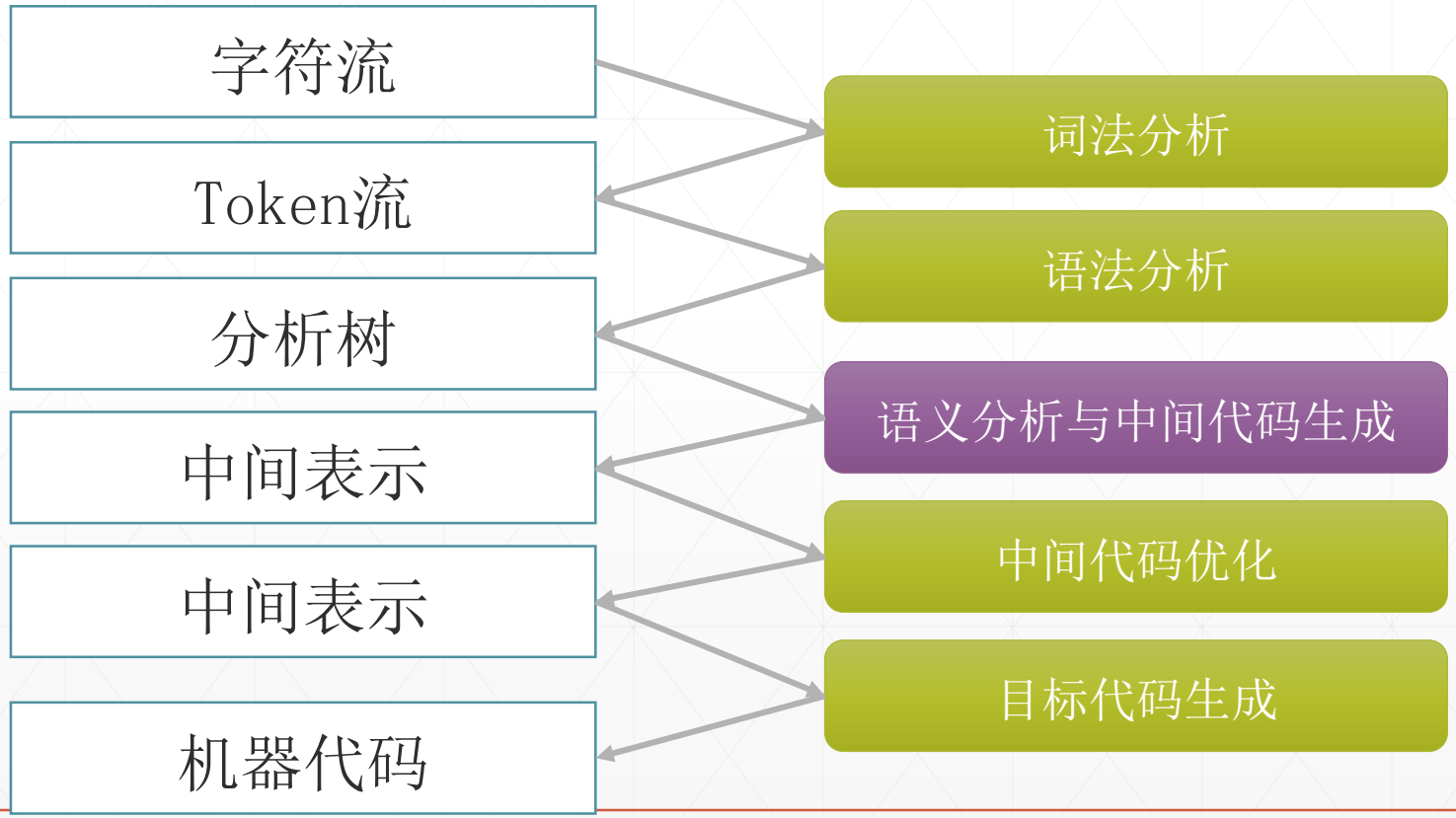


表 格 管 理
源程序
词法分析
语法分析
语义处理
中间代码生成
代码优化
代码生成
目标程序
出 错 处 理
front end
back end

语义分析：概览

- 基本功能


字符流
Token流
分析树
中间表示
中间表示
机器代码
词法分析
语法分析
语义分析与中间代码生成
中间代码优化
目标代码生成

语义分析：概览

- **词法分析**: 程序符合词法规则
  - 找到合法的Token
  - 报告那些不合法的Token
- **语法分析**: 程序符合语法规则
  - 结构是正确的
  - 报告结构不良的问题
- **语义分析**
  - 前端编译的最后一个阶段，最后一道墙
  - 检查并报告所有可能的错误

语义分析：概览

Lexical analysis: Each token is valid?

```
if ift 3 "This"                    /* valid Java tokens */
#a1123                             /* not a token */
```

Syntactic analysis: Tokens appear in the correct order?

```
return 3 + "f";   /* valid Java syntax */
for break         /* invalid syntax */
```

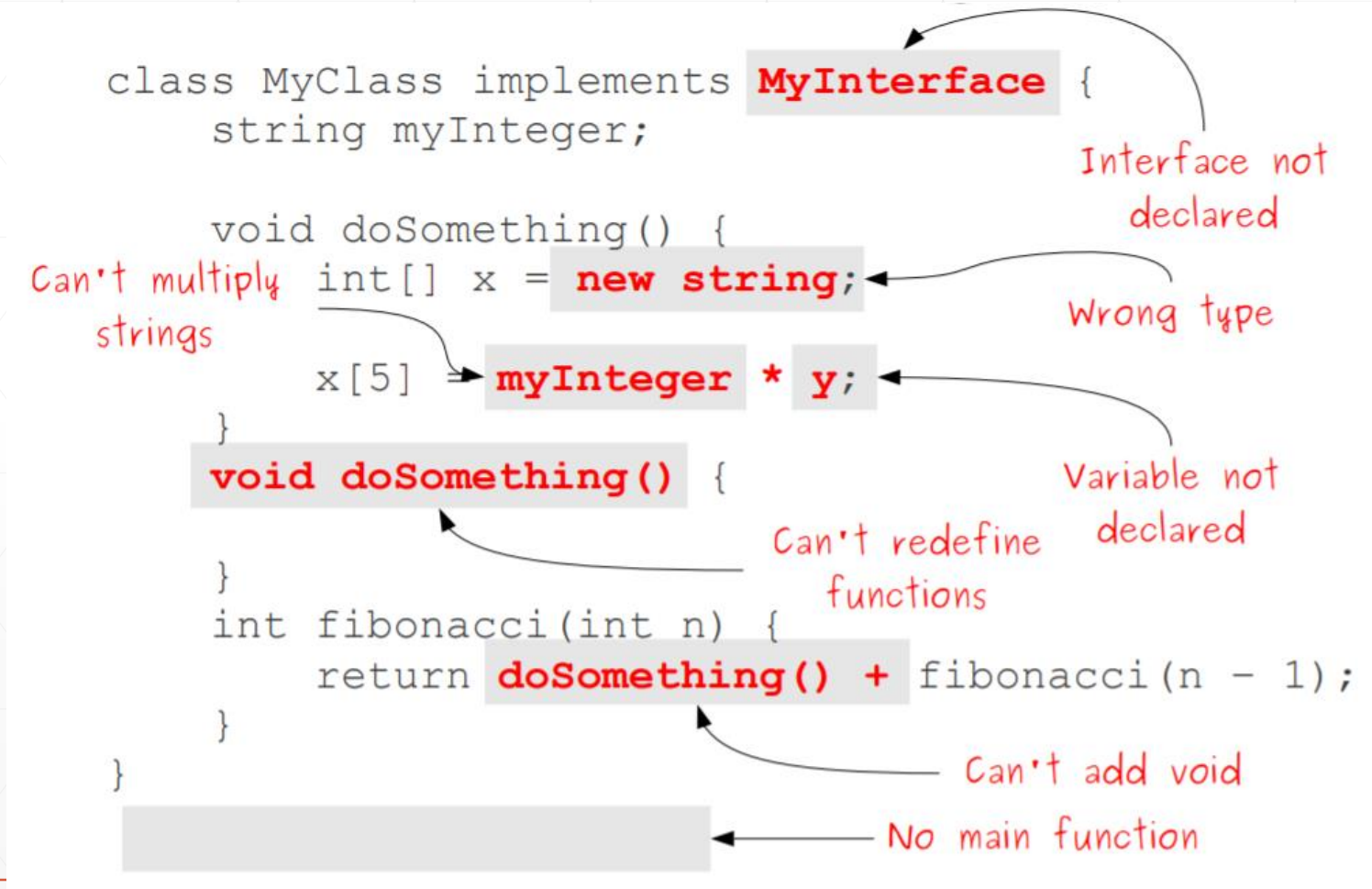
Semantic analysis: Names used correctly? Types consistent?

```
int v = 42 + 13;      /* valid in Java (if v is new) */
return 3 + "f";       /* invalid */
return f + f(3);      /* invalid */
```

语义分析：概览

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

语义分析：概览


class MyClass implements MyInterface {
string myInteger;
void doSomething() {
int[] x = new string;
x[5] = myInteger * y;
}
void doSomething() {
}
int fibonacci(int n) {
return doSomething() + fibonacci(n - 1);
}
}
Interface not declared
Wrong type
Can't multiply strings
Variable not declared
Can't redefine functions
Can't add void
No main function

语义分析：概览

- 语法正确，但是…
  - 未声明的标识符
  - 重复定义的标识符
  - 数组越界访问
  - 类型不兼容的访问
  - Break语句的位置
  - goto的目标不存在
  - …

```
foo(int a, char * s){...}

int bar() {
  int f[3];
  int i, j, k;
  char q, *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 42;
  j = m + k;
  printf("%s,%s.\n",p,q);
  goto label42;
}
```

语义分析：概览

- 一个Java程序

```
public class Name {
    int Name;
    Name Name(Name Name) {
        Name.Name = 137;
        return Name((Name) Name);
    }
}
```

语义分析：概览

- 一个Java程序

```
public class Name {
    int Name;
    Name Name(Name Name) {
        Name.Name = 137;
        return Name((Name) Name);
    }
}
```

语义分析：概览

- 一个C++程序

```
#include <stdio.h>
int main(){
    int x = 137;
    {
        char x = 'a';
        if( x == 'a')
            x = 1;
    }
     if(x == 137)
        printf("Y");
    return 0;
}
```

语义分析：概览

- 语义分析
  - 扫描程序（一遍/多遍）分析程序表达的含义
- 主要内容
  - 关于符号的声明和使用：符号表+作用域检查
  - 关于类型的标记和操作：类型检查

语义分析：概览

Verify names are defined (scope) and are of the right type (type).

```
int i = 5;
int a = z;      /* Error: cannot find symbol */
int b = i[3]; /* Error: array required, but int found */
```

Verify the type of each expression is consistent (type).

```
int j = i + 53;
int k = 3 + "hello";     /* Error: incompatible types */
int l = k(42);           /* Error: k is not a method */
if ("Hello") return 5; /* Error: incompatible types */
String s = "Hello";
int m = s;               /* Error: incompatible types */
```

属性文法

- 上下文相关的要求无法使用上下文无关文法描述
  - 例1: $a^n b^n c^n$
  - 例2: $w\alpha w, w \in \sum^*$

$S \rightarrow aSBC \mid abC$

$CB \rightarrow CD$

$CD \rightarrow BD$

$BD \rightarrow BC$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

$L(G_1) = \{a^n b^n c^n \mid n \geq 1\}$

属性文法

- 对上下文无关文法进行改进以便于验证上下文有关的条件
  - 属性文法可以用于描述一个语言的语法上下文相关的信息
  - 属性文法还可以用于描述程序的操作语义

属性文法

A context-free grammar that has been extended to provide context sensitivity using a set of attributes, assignment of attribute values, evaluation rules, and conditions.

- 每个属性都有相应的值域，例如整数、字符或者字符串，或者更加复杂的类型
- 将输入的程序看做是一个语法树，属性文法可以使用综合属性从子节点向父节点传递属性值，也可以使用继承属性从当前节点向子节点传递属性值
- 语法树对应的节点可以为设置、修改或者检查属性值

属性文法：Attribute Grammar

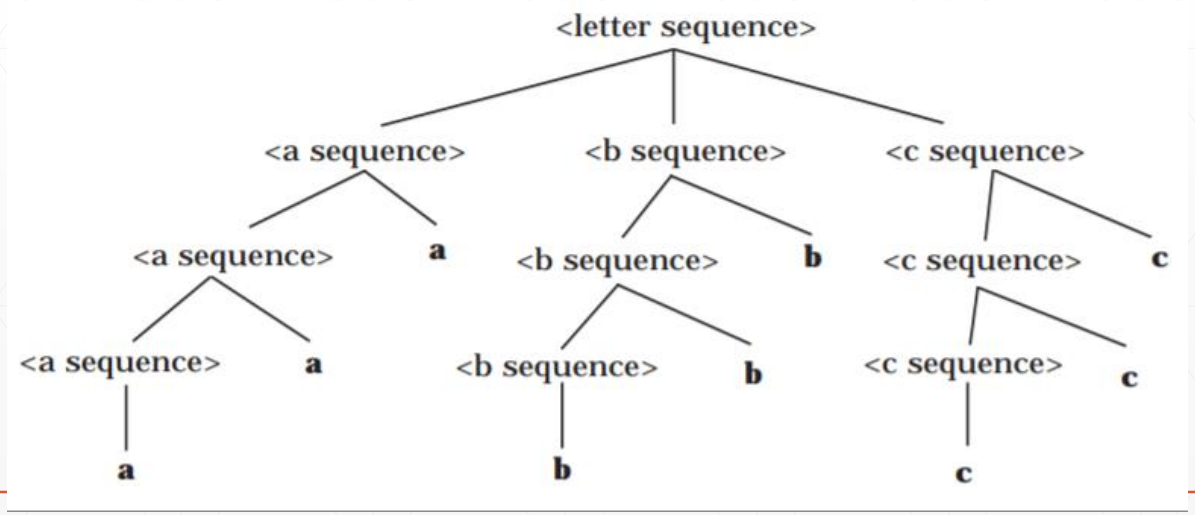
- 属性文法举例

<letter sequence> ::= <a sequence> <b sequence> <c sequence>

<asequence> ::= **a** | <a sequence>**a**

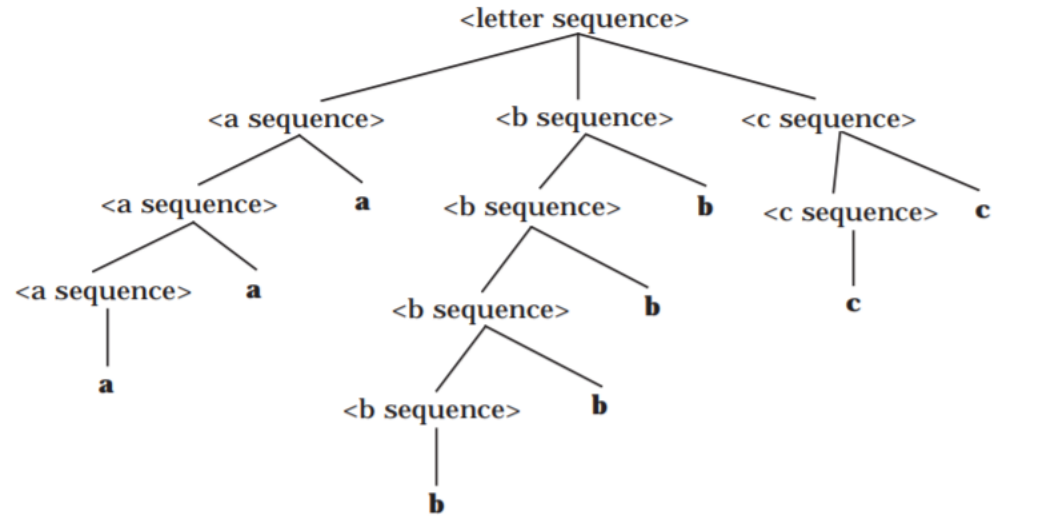
<bsequence> ::= **b** | <bsequence> **b**

<csequence> ::= **c** | <csequence> **c**



属性文法：Attribute Grammar

- 属性文法举例
  - 上下文无关文法无法描述
  - 上下文有关文法可以描述



属性文法：Attribute Grammar

- 属性文法
  - 属性文法提供了另一种定义上下文相关信息的方式
  - 对每个非终结符加一个size属性
  - 从下自上计算size属性
  - 在根节点检查三个序列的size是否相等

<letter sequence> ::= <a sequence> <b sequence> <c sequence>

<asequence> ::= **a** | <a sequence>**a**

<bsequence> ::= **b** | <bsequence> **b**

<csequence> ::= **c** | <csequence> **c**

属性文法：Attribute Grammar

- 属性文法举例

<lettersequence> ::= <asequence> <bsequence> <csequence>
condition :
Size (<asequence>) = *Size* (<bsequence>) = *Size* (<csequence>)

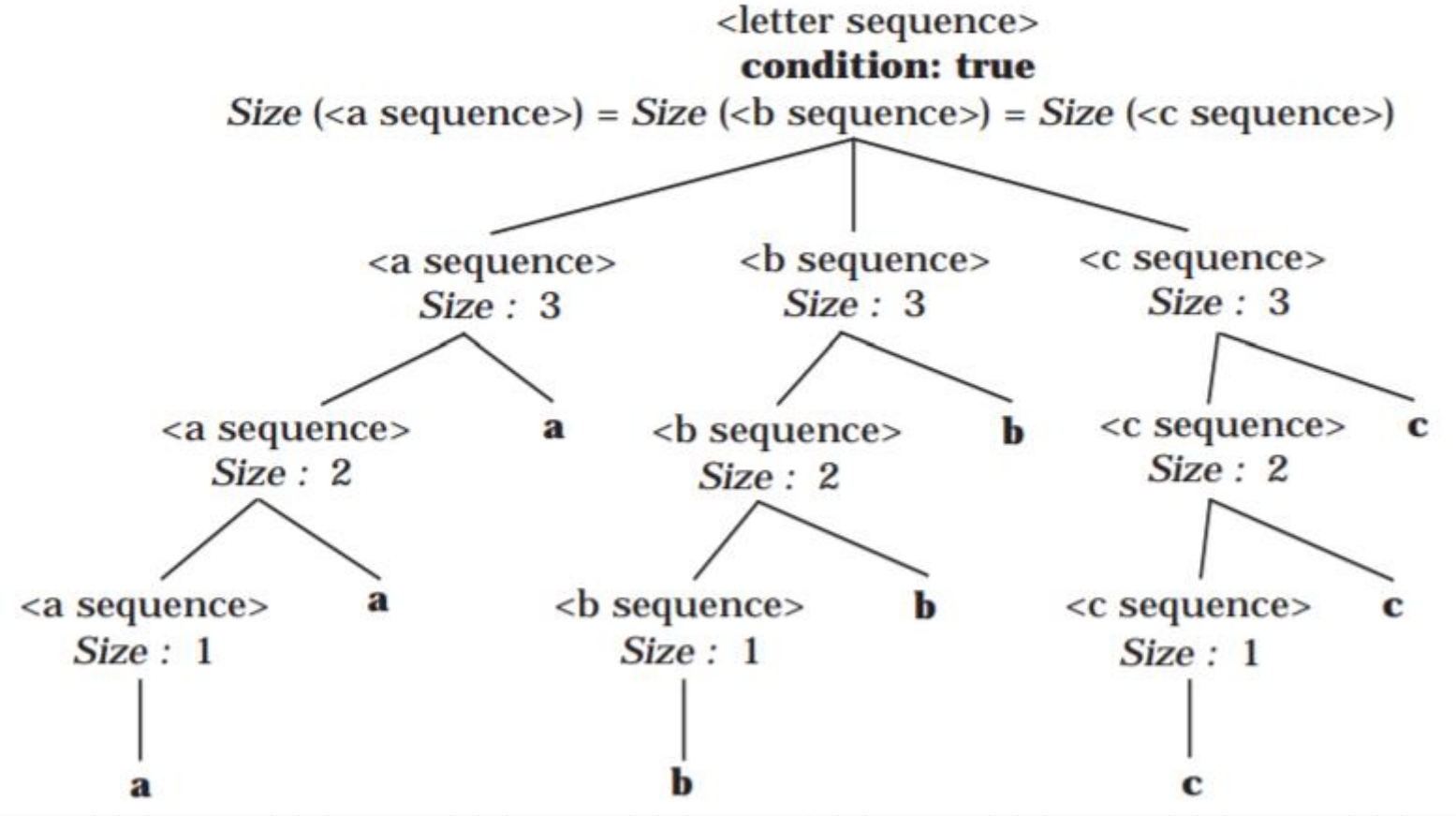
<asequence> ::= **a**
Size (<asequence>) ← 1
| <asequence> $_2$ **a**
Size (<asequence>) ← *Size* (<asequence> $_2$) + 1

<bsequence> ::= **b**
Size (<bsequence>) ← 1
| <bsequence> $_2$ **b**
Size (<bsequence>) ← *Size* (<bsequence> $_2$) + 1

<csequence> ::= **c**
Size (<csequence>) ← 1
| <csequence> $_2$ **c**
Size (<csequence>) ←*Size* (<csequence> $_2$) + 1

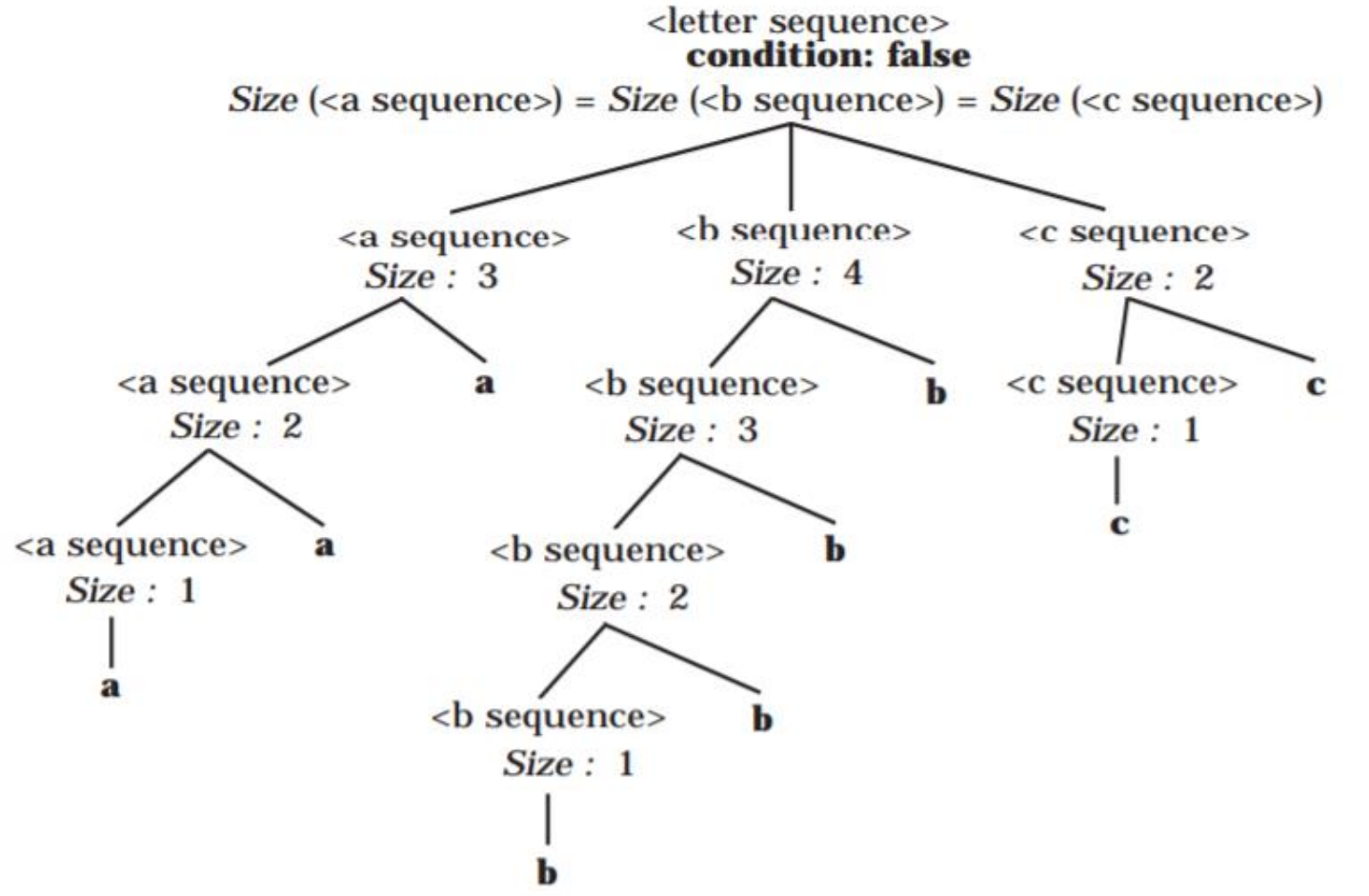
属性文法：Attribute Grammar

- 属性文法举例



属性文法：Attribute Grammar

- 属性文法举例



属性文法：Attribute Grammar

- 属性文法举例
  - 自底向上计算a分支的size属性
  - 将其传递给b和c分支的InhSize属性
  - 沿着语法树对b和c分支自顶向下继承传递InhSize属性
  - b和c分支每遇到一个b或者c就对InhSize减1

属性文法：Attribute Grammar

- 属性文法举例

<lettersequence> ::= <asequence> <bsequence> <csequence>
InhSize (<bsequence>) ← *Size* (<asequence>)
InhSize (<csequence>) ← *Size* (<asequence>)

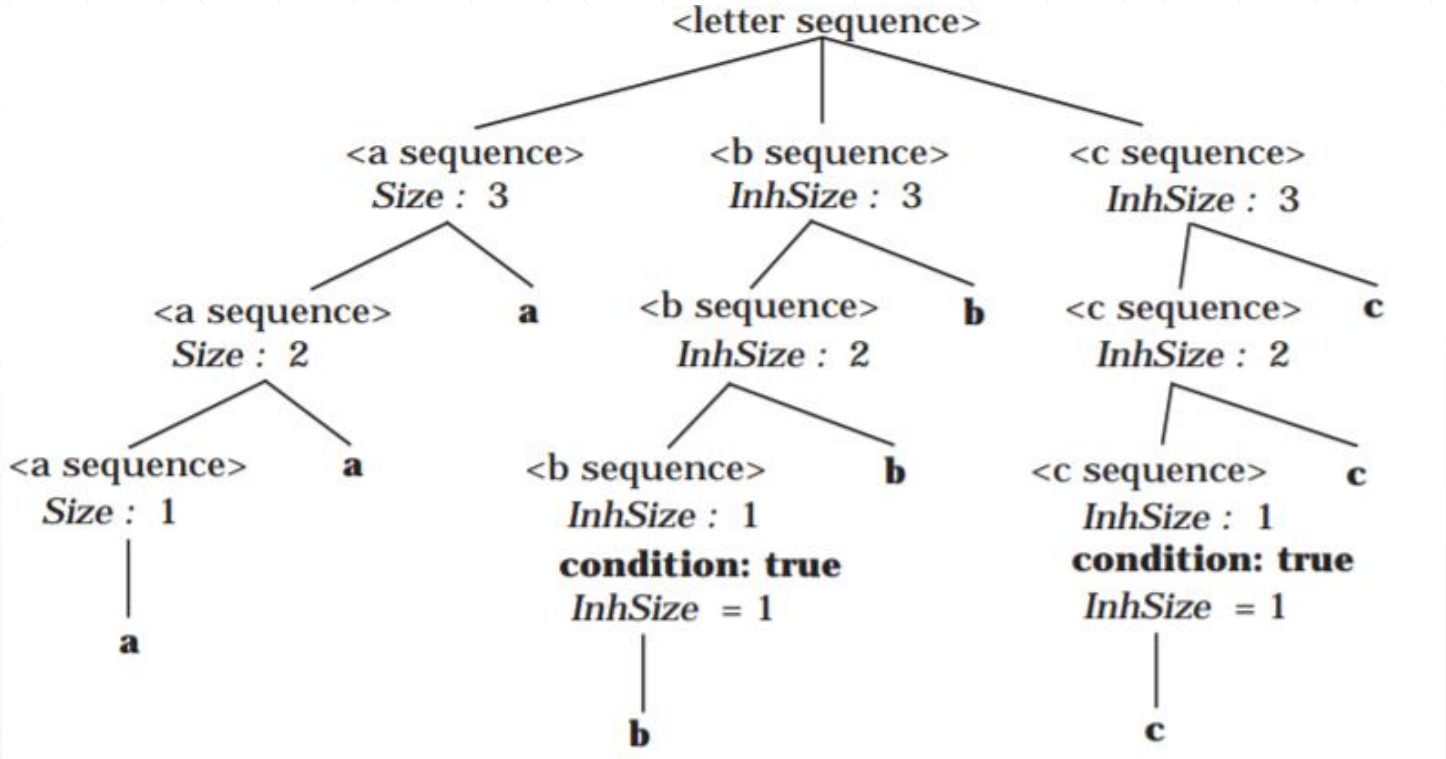
<asequence> ::= **a**
Size (<asequence>) ← 1
| <asequence> $_2$ **a**
Size (<asequence>) ← *Size* (<asequence> $_2$) + 1

<bsequence> ::= **b**
condition: *InhSize* (<bsequence>) = 1
| <bsequence> $_2$ **b**
InhSize (<bsequence> $_2$) ← *InhSize* (<bsequence>) – 1

<csequence> ::= **c**
condition: *InhSize* (<csequence>) = 1
| <csequence> $_2$ **c**
InhSize (<csequence> $_2$) ← *InhSize* (<csequence>) – 1

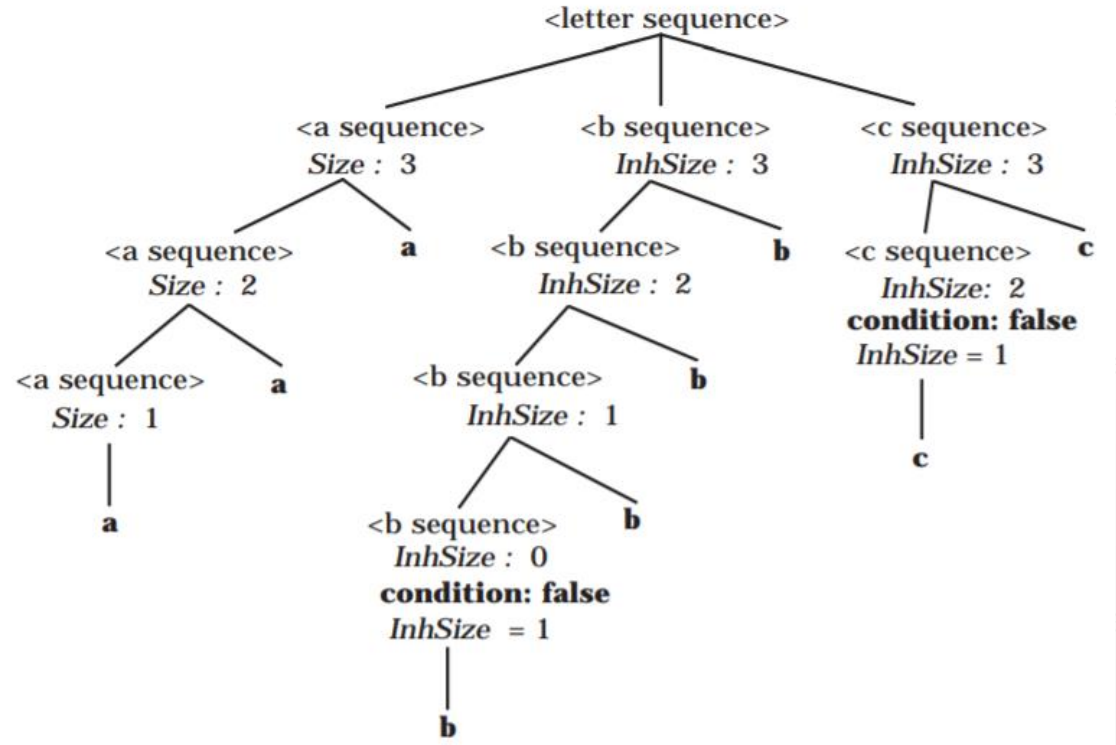
属性文法：Attribute Grammar

- 属性文法举例



属性文法：Attribute Grammar

- 属性文法举例



属性文法：Attribute Grammar

- 定义语义举例
  - use of attribute grammars to **specify meaning** by developing the semantics of binary numerals

Nonterminals	Synthesized Attributes	Inherited Attributes
<binary numeral>	*Val*	—
<binary digits>	*Val, Len*	—
<bit>	*Val*	—

属性文法：Attribute Grammar

- 定义语义举例

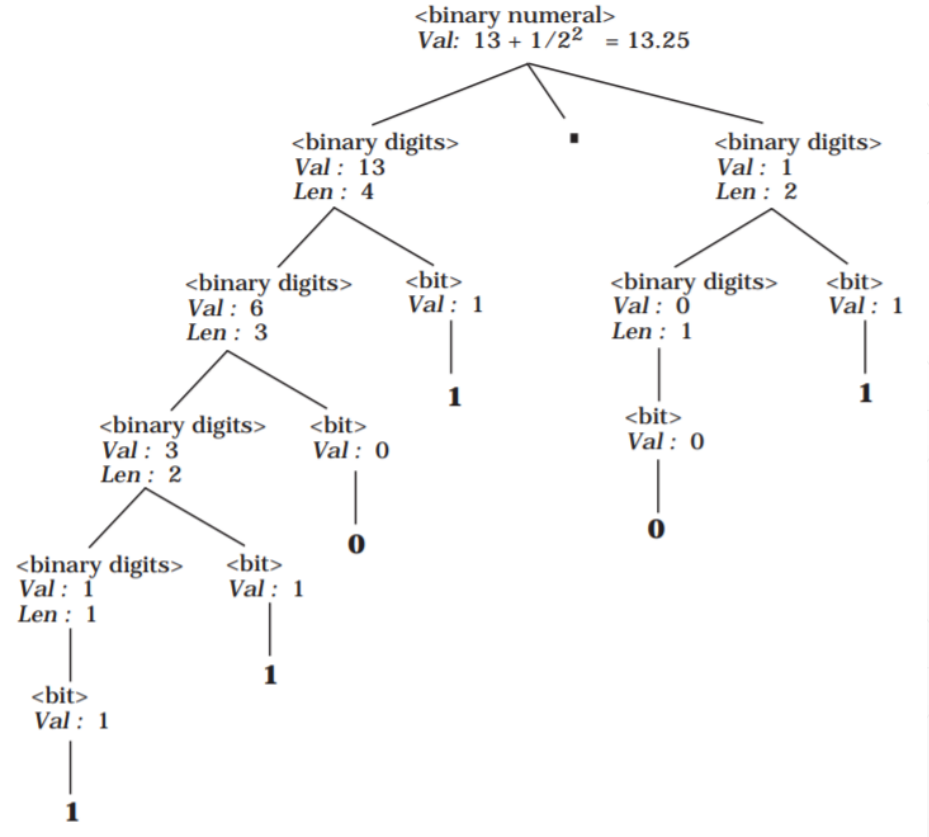
<binary numeral> ::= <binary digits>$_1$. <binary digits>$_2$
Val (<binary numeral>) ← *Val* (<binary digits>$_1$) +
Val (<binary digits>$_2$) / $2^{Len\ (<\text{binary digits}>_2)}$

<binary digits> ::=
<binary digits>$_2$ <bit>
Val (<binary digits>) ← 2 • *Val* (<binary digits>$_2$) + *Val* (<bit>)
Len (<binary digits>) ← *Len* (<binary digits>$_2$) + 1
| <bit>
Val (<binary digits>) ← *Val* (<bit>)
Len (<binary digits>) ← 1

<bit> ::=
0
Val (<bit>) ← 0
| **1**
Val (<bit>) ← 1

属性文法：Attribute Grammar

- 定义语义举例



属性文法：Attribute Grammar

- 定义语义举例
  - 前述例子没有使用位置信息

$$123.45 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

and then in base 2,

$$110.101 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$
$$= 6.625 \text{ (base 10).}$$

属性文法：Attribute Grammar

- 定义语义举例

Nonterminals	Synthesized Attributes	Inherited Attributes
<binary numeral>	*Val*	—
<binary digits>	*Val*	*Pos*
<fraction digits>	*Val*, *Len*	—
<bit>	*Val*	*Pos*

<binary numeral> ::= <binary digits> . <fraction digits>

Val (<binary numeral>) ← *Val* (<binary digits>)+*Val* (<fraction digits>)

Pos (<binary digits>) ← 0

属性文法：Attribute Grammar

- 定义语义举例

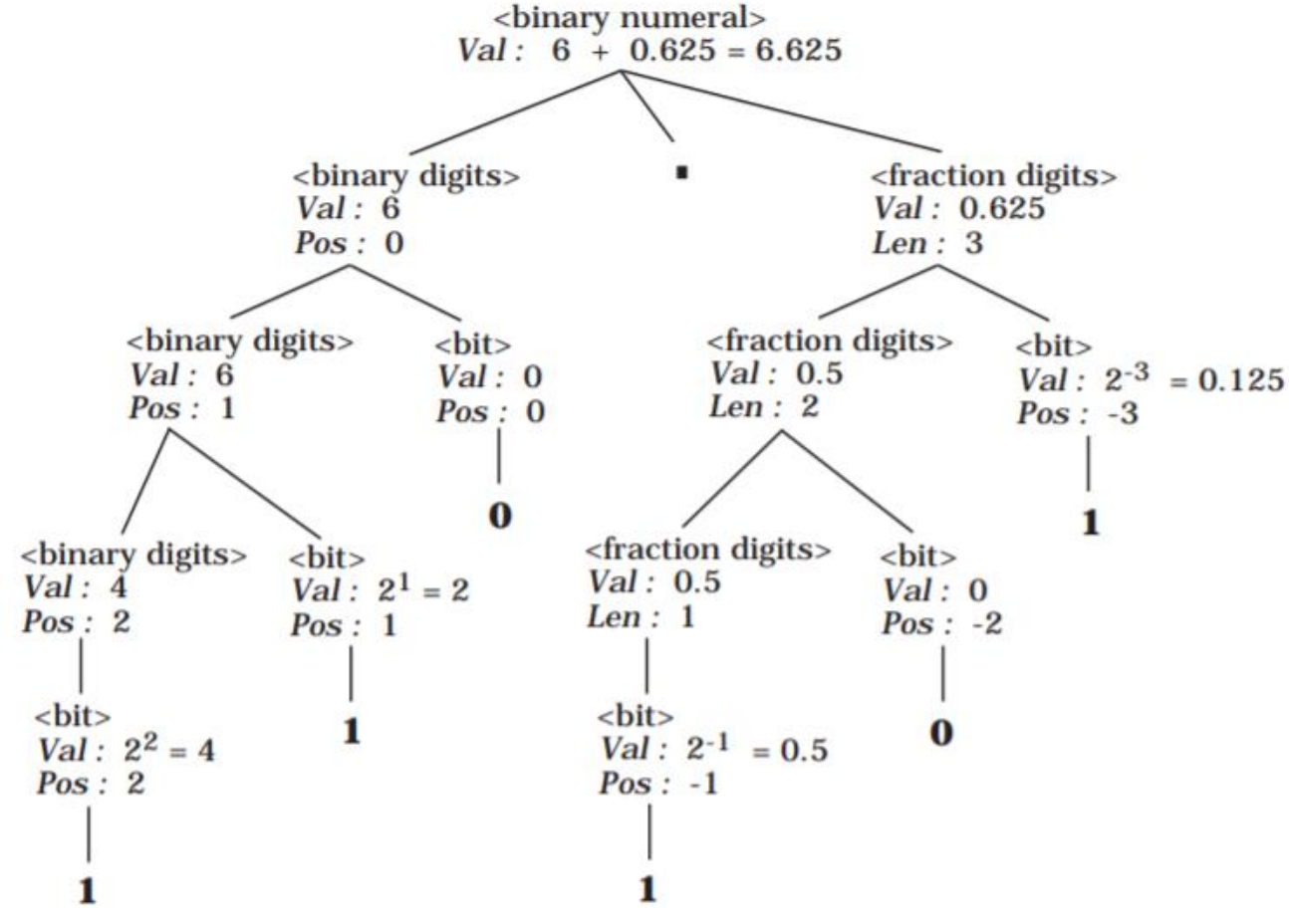
<binary digits> ::=
    <binary digits>$_2$ <bit>
        Val (<binary digits>) ← *Val* (<binary digits>$_2$) + *Val* (<bit>)
        Pos (<binary digits>$_2$) ← *Pos* (<binary digits>) + 1
        Pos (<bit>) ← *Pos* (<binary digits>)
    | <bit>
        Val (<binary digits>) ← *Val* (<bit>)
        Pos (<bit>) ← *Pos* (<binary digits>)

<fraction digits> ::=
    <fraction digits>$_2$ <bit>
        Val (<fraction digits>) ← *Val* (<fraction digits>$_2$) + *Val* (<bit>)
        Len (<fraction digits>) ← *Len* (<fraction digits>$_2$) + 1
        Pos (<bit>) ← - *Len* (<fraction digits>)
    | <bit>
        Val (<fraction digits>) ← *Val* (<bit>)
        Len (<fraction digits>) ← 1
        Pos (<bit>) ← - 1

<bit> ::=
    0
        Val (<bit>) ← 0
    | **1**
        Val (<bit>) ← $2^{Pos\ (<bit>)}$

属性文法：Attribute Grammar

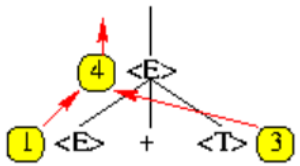
- 定义语义举例



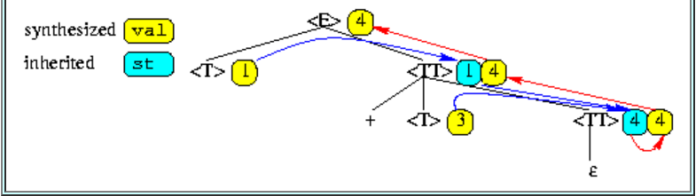
属性文法：Attribute Grammar

- 综合属性 **Synthesized Attributes:**结点的属性值是通过子结点的属性值计算得到
- 继承属性 **Inherited Attributes:**结点的属性值是由该结点的父结点和(或)兄弟结点的属性定义的

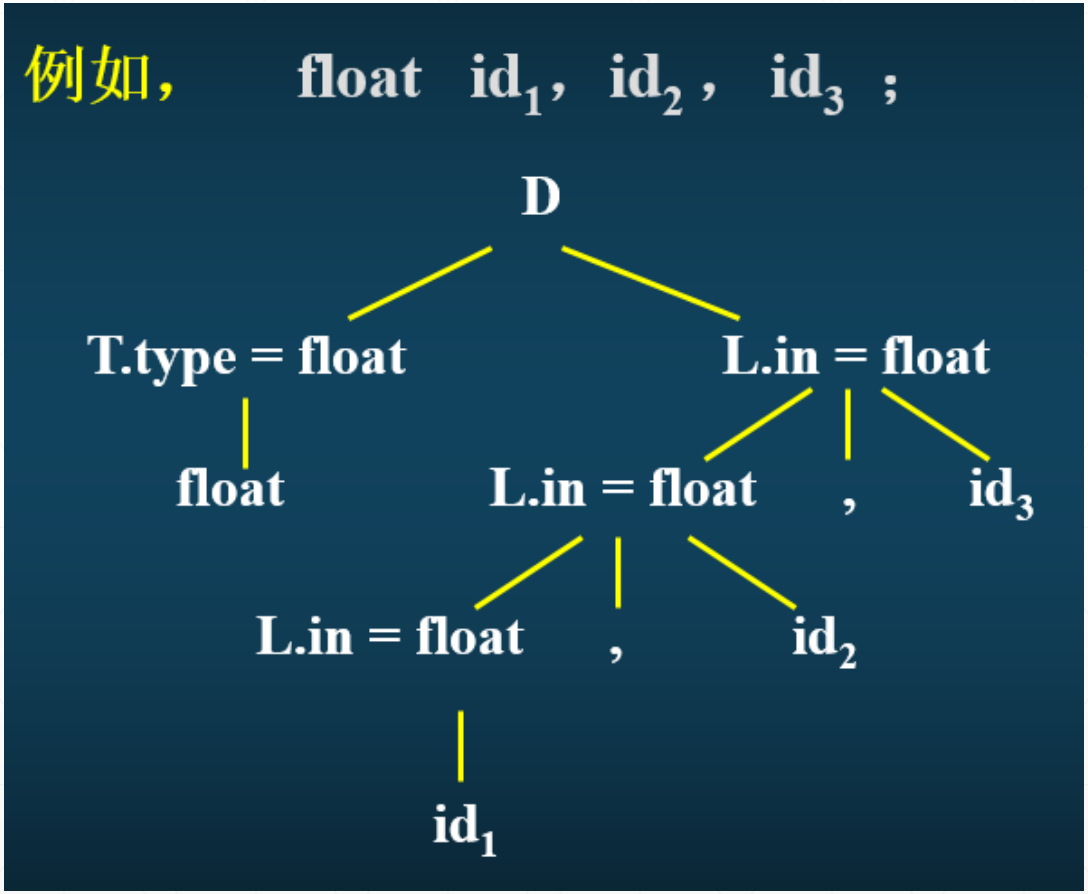
Production	Semantic rule
$\langle E_1 \rangle \rightarrow \langle E_2 \rangle + \langle T \rangle$	$E_1.val := E_2.val + T.val$



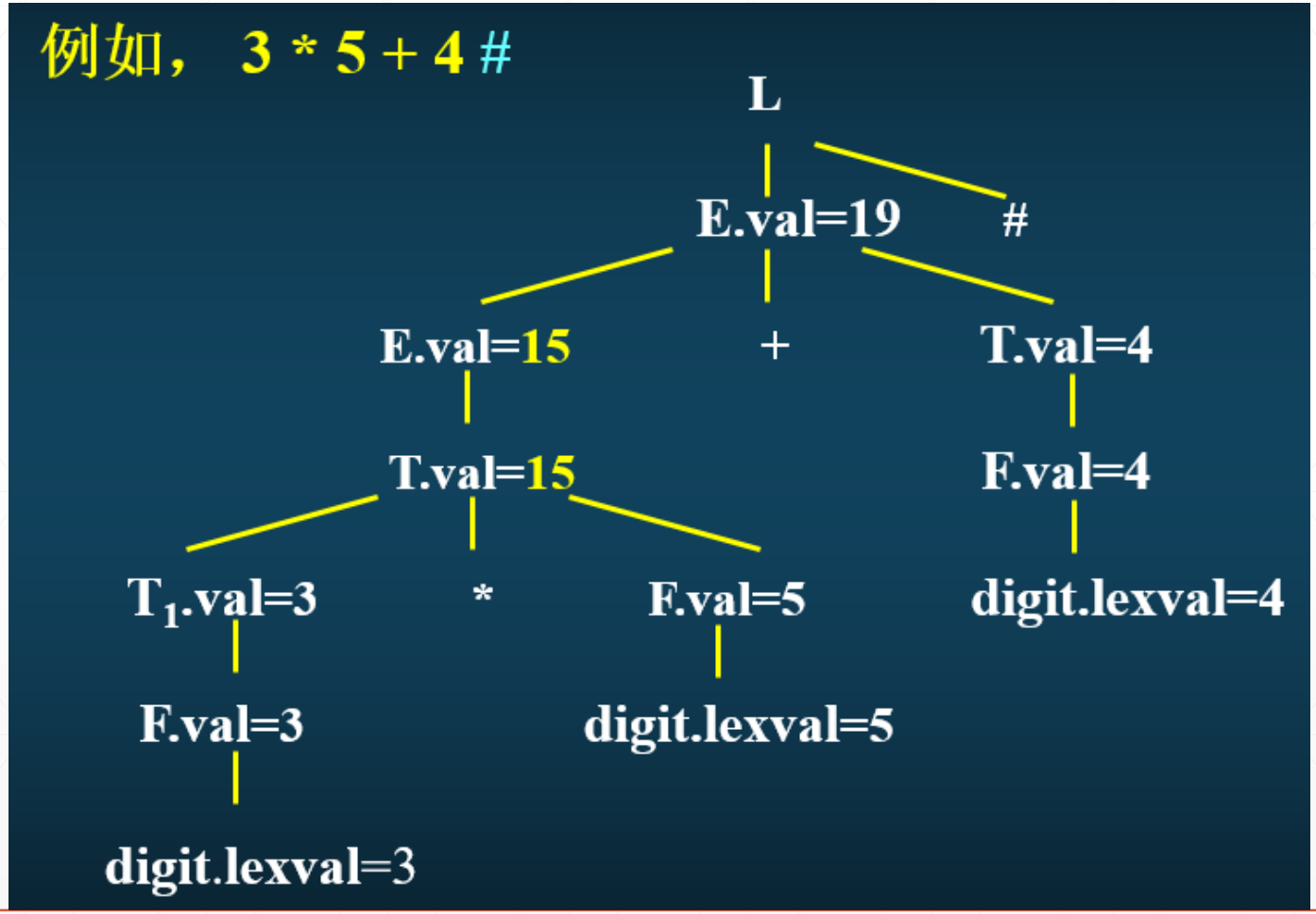
Production	Semantic rule
$\langle E \rangle \rightarrow \langle T \rangle \langle TT \rangle$	$TT.st := T.val;\ E.val := TT.val$
$\langle TT_1 \rangle \rightarrow + \langle T \rangle \langle TT_2 \rangle$	$TT_2.st := TT_1.st + T.val;$ $TT_1.val := TT_2.val$
$\langle TT \rangle \rightarrow \varepsilon$	$TT.val := TT.st$



属性文法：Attribute Grammar

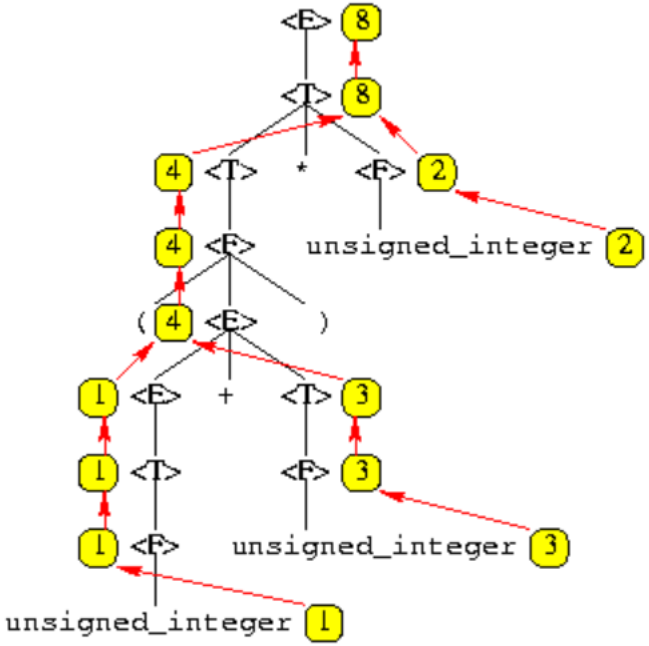


属性文法：Attribute Grammar



属性文法：Attribute Grammar

- Decorated Parse Trees
  - A parser produces a parse tree that is decorated with the attribute values



属性文法：Attribute Grammar

- 属性文法举例

Production	*Semantic rule*
$\langle E_1 \rangle \rightarrow \langle E_2 \rangle + \langle T \rangle$	$E_1.val := E_2.val + T.val$
$\langle E_1 \rangle \rightarrow \langle E_2 \rangle - \langle T \rangle$	$E_1.val := E_2.val - T.val$
$\langle E \rangle \rightarrow \langle T \rangle$	$E.val := T.val$
$\langle T_1 \rangle \rightarrow \langle T_2 \rangle * \langle F \rangle$	$T_1.val := T_2.val * F.val$
$\langle T_1 \rangle \rightarrow \langle T_2 \rangle / \langle F \rangle$	$T_1.val := T_2.val / F.val$
$\langle T \rangle \rightarrow \langle F \rangle$	$T.val := F.val$
$\langle F_1 \rangle \rightarrow - \langle F_2 \rangle$	$F_1.val := -F_2.val$
$\langle F \rangle \rightarrow (\langle E \rangle)$	$F.val := E.val$
$\langle F \rangle \rightarrow$ unsigned_int	$F.val :=$ unsigned_int.val

属性文法：Attribute Grammar

- 符号表举例
  - 将声明信息收集放入一个属性Symbol-table中
  - Symbol-table从声明侧综合而来，在其他语句侧是继承而来
  - 在语句块级进行传播转移

属性文法：Attribute Grammar

- 符号表示例

```
program p is
    var x, y : integer;
    var a : boolean;
begin
    :
end
```

Attribute	Value Types
Type	{ *integer*, *boolean*, *program*, *undefined* }
Name	String of letters or digits
Var-list	Sequence of Name values
Symbol-table	Set of pairs of the form [Name, Type]

属性文法：Attribute Grammar

Nonterminals	Synthesized Attributes	Inherited Attributes
<block>	—	*Symbol-table*
<declarationsequence>	*Symbol-table*	—
<declaration>	*Symbol-table*	—
<variable list>	*Var-list*	—
<type>	*Type*	—
<commandsequence>	—	*Symbol-table*
<command>	—	*Symbol-table*
<expr>	—	*Symbol-table, Type*
<integer expr>	—	*Symbol-table, Type*
<term>	—	*Symbol-table, Type*
<element>	—	*Symbol-table, Type*
<boolean expr>	—	*Symbol-table, Type*
<boolean term>	—	*Symbol-table, Type*
<boolean element>	—	*Symbol-table, Type*
<comparison>	—	*Symbol-table*
<variable>	*Name*	—
<identifier>	*Name*	—
<letter>	*Name*	—
<digit>	*Name*	—

Figure 3.10: Attributes Associated with Nonterminal Symbols

属性文法：Attribute Grammar

- 符号表示例

```
program p is
    var x, y : integer;
    var a : boolean;
begin
    :
end
```

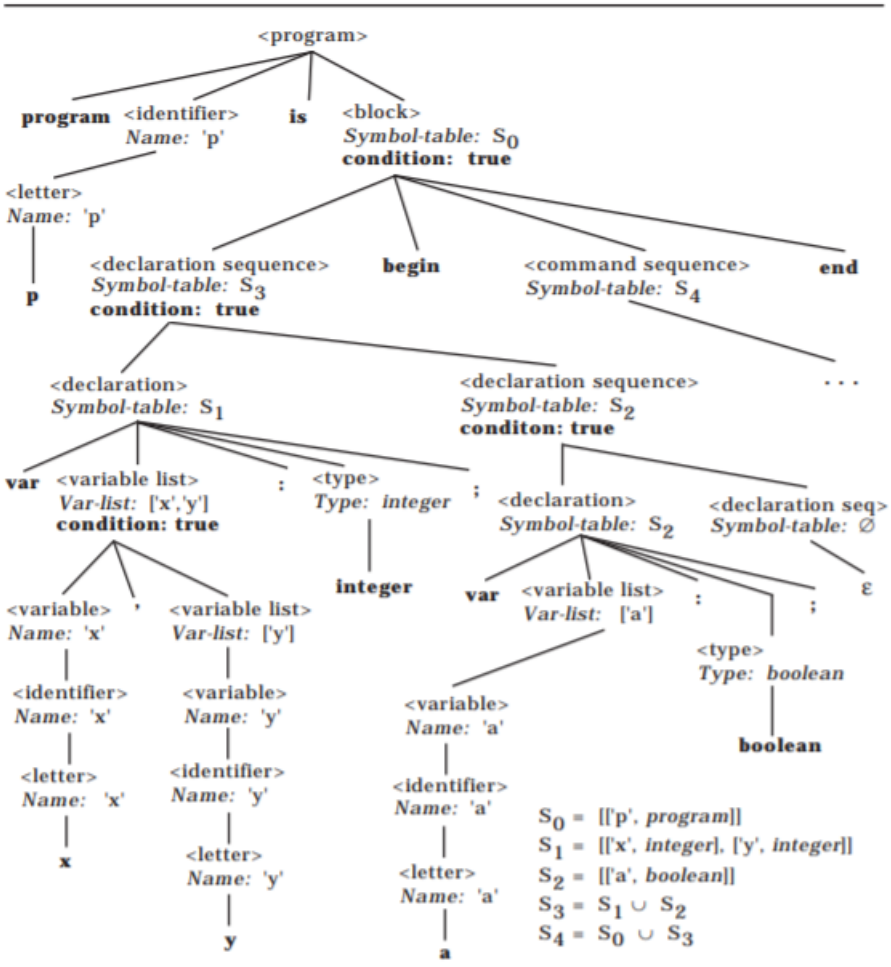


Figure 3.11: Decorated Parse Tree for Wren Program Fragment

[[‘p’, *program*], [‘x’, *integer*], [‘y’, *integer*], [‘a’, *boolean*]].

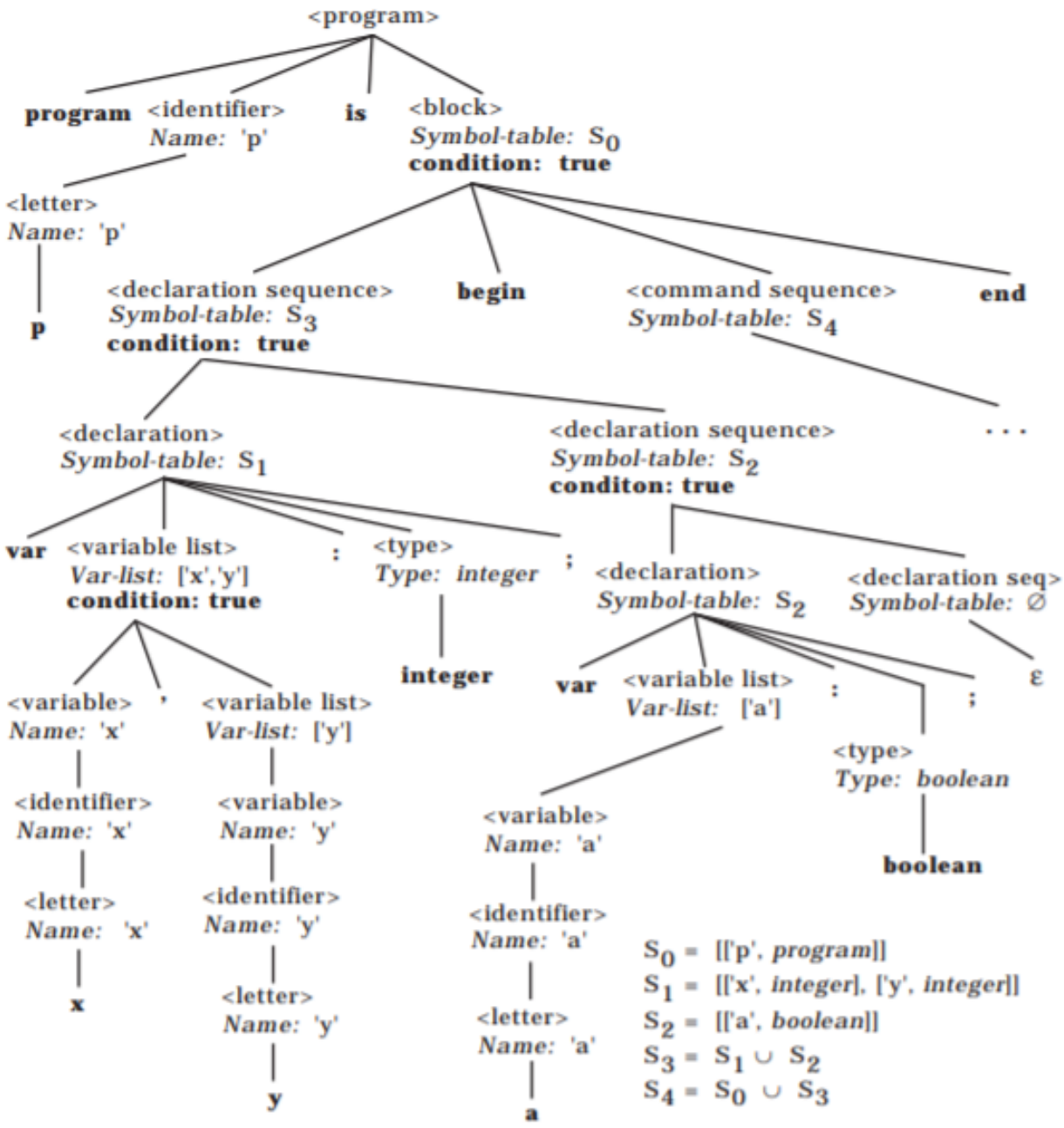


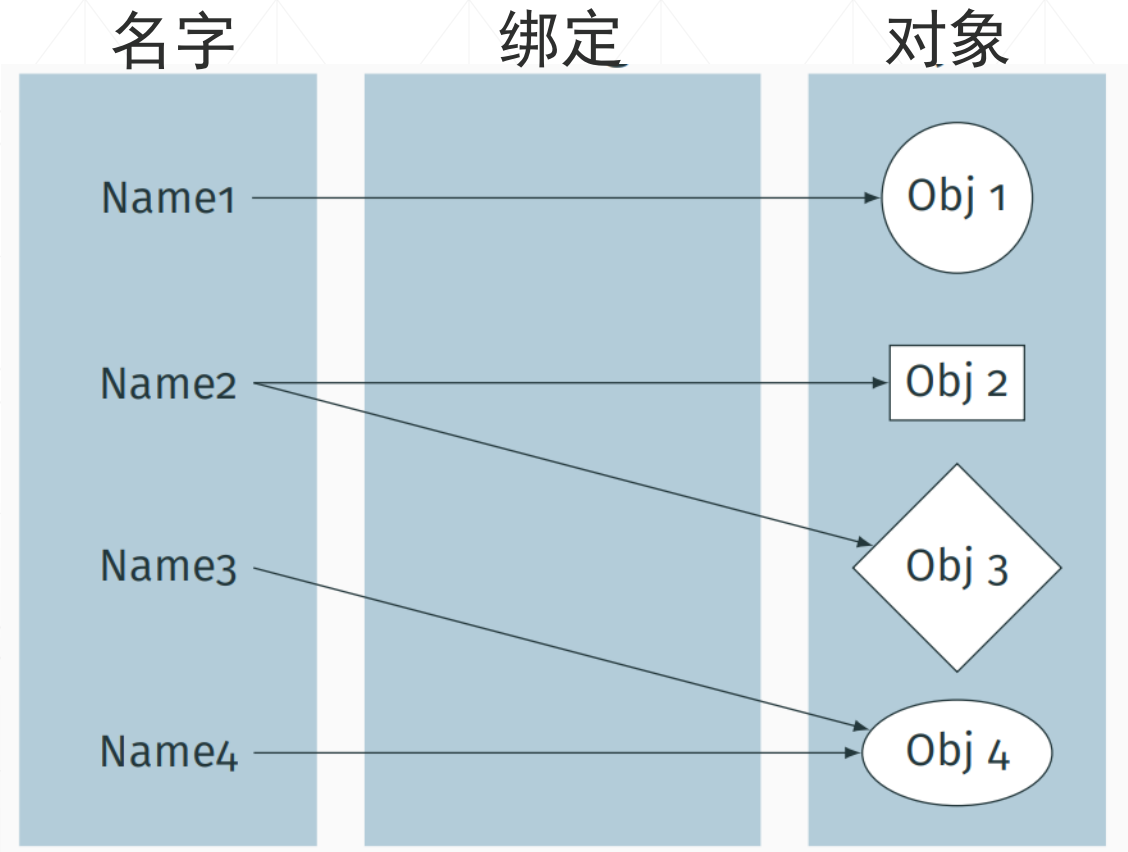
Figure 3.11: Decorated Parse Tree for Wren Program Fragment

属性文法：Attribute Grammar

- 符号表示例

<program> ::= **program** <identifier> **is** <block>
    Symbol-table(<block>) ←
        add-item((*Name*(<identifier>), *program*), empty-table)
<declaration> ::= **var** <variable list> **:** <type>**;**
    Symbol-table(<declaration>) ←
            build-symbol-table(*Var-list*(<variable list>), *Type*(<type>))
<variable list> ::=
      <variable>
        Var-list(<variable list>) ←
                cons(*Name*(<variable>), empty-list)
    | <variable> , <variable list>$_2$
        Var-list(<variable list>) ←
                cons(*Name*(<variable>), *Var-list*(<variable list>$_2$))

作用域与符号表



作用域与符号表

- 作用域
  - 定义了变量的可见范围和存活区间，将变量的使用和声明相关联
  - 名字和数据对象绑定的时间和位置
- 词法作用域/lexical scoping
  - 根据代码的结构和嵌套层次查找对应的符号
  - 作用域可以是嵌套的，block可能对应一个新的作用域
  - 使用符号表在编译时查找绑定关系
- 动态作用域/Dynamic scoping

作用域与符号表

- C/C++和Java等的静态作用域

```
void foo()
{
    int x;
}
```

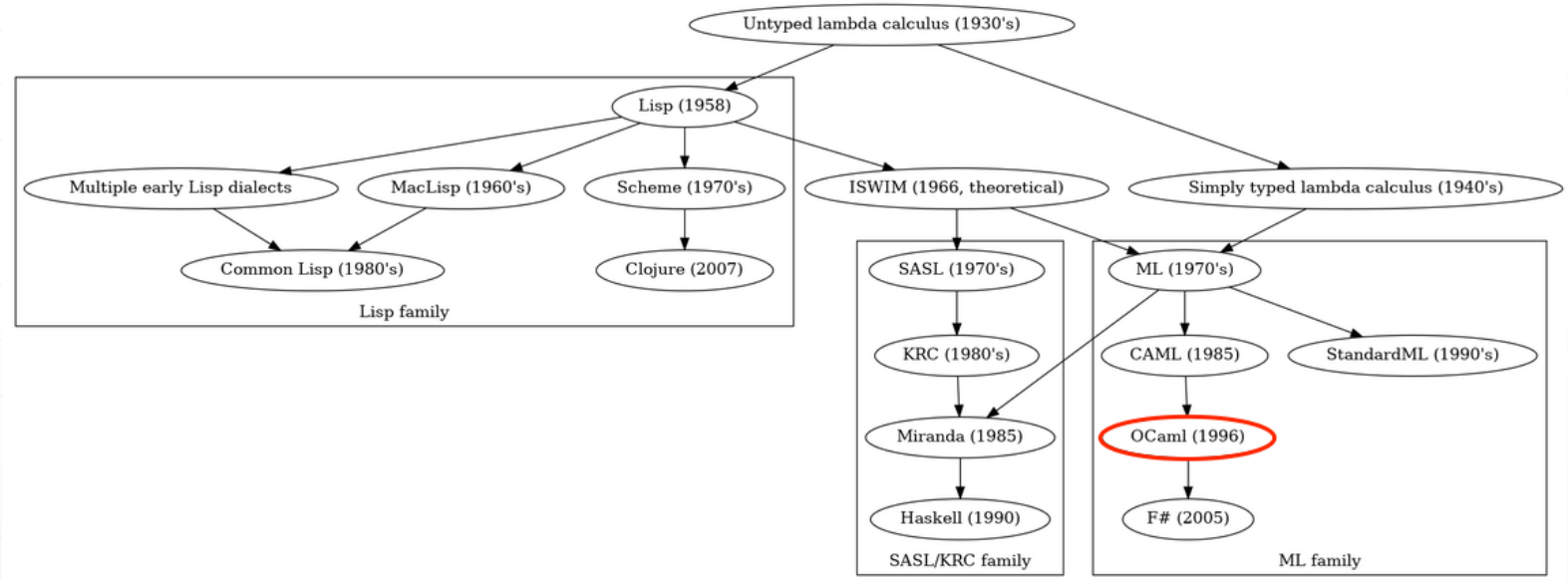
作用域与符号表

- C/C++和Java等的静态作用域

```
void foo()
{
  int x;

  while ( a < 10 ) {
    int x;

  }

}
```

作用域与符号表

- OCaml的静态作用域



作用域与符号表

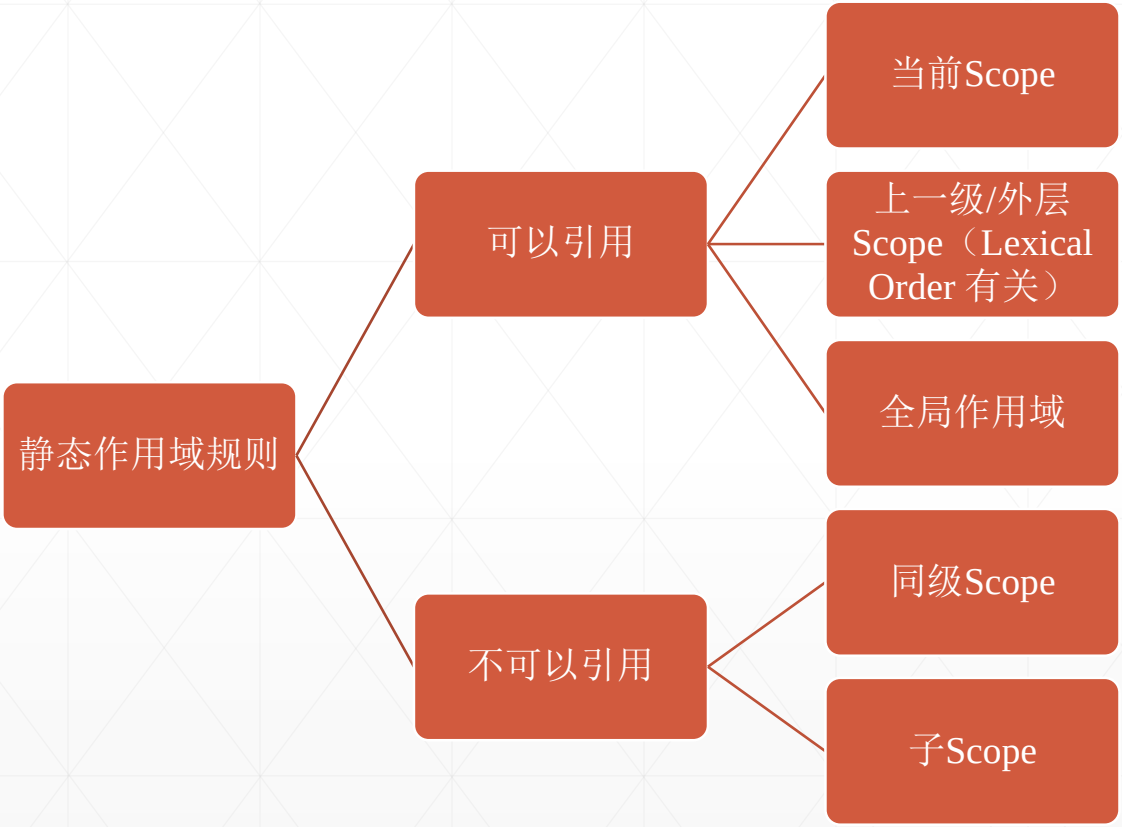
- OCaml的静态作用域

```
let x = 8 in
let x = x + 1 in
```

Returns the pair (12, 8):

```
let x = 8 in
(let x = x + 2 in
 x + 2),
x
```

作用域与符号表


静态作用域规则
可以引用
当前Scope
上一级/外层Scope（Lexical Order 有关）
全局作用域
不可以引用
同级Scope
子Scope

作用域与符号表

- 符号表主要用于对标识符的声明和使用进行描述和分析
- 分析的内容主要包括
  - 作用域
  - 声明和使用的顺序
  - 重复声明
- 每个程序点对应的符号表

i	local	int
done	local	boolean
insert	method	...
List	class	...
x	formal	List
⋮	⋮	⋮

作用域与符号表

- 以面向对象程序为例
  - 类的层次结构
    - 定义了哪些类?
    - 是什么样的继承层次?
    - 继承关系是否正确
  - 类本身
    - 定义了哪些属性成员?
    - 定义了哪些方法?
    - 方法对应的签名是什么?
  - 标识符
    - 标识符是否重复定义?
    - 使用时是否定义过了?
    - 标识符的使用是否正确?

i	local	int
done	local	boolean
insert	method	...
List	class	...
x	formal	List
⋮	⋮	⋮

作用域与符号表

- 符号表
  - 用于跟踪记录符号绑定关系的数据结构
  - 嵌套的Scope
    - 跟踪记录当前、外层、全局的符号绑定关系
  - 实现
    - 每个Scope对应一个符号表
    - List和Hash等

作用域与符号表

- 符号表举例: C-style

```
int x;
int main() {
  int a = 1;
  int b = 1; {
    float b = 2;
    for (int i = 0; i < b; i++) {
      int b = i;
      ...
    }
  }
  b + x;
}
```

作用域与符号表

- 符号表举例: C-style
  - Reach a declaration: Add entry to current table

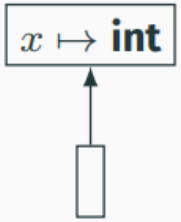
```
int x;
int main() {
  int a = 1;
  int b = 1; {
    float b = 2;
    for (int i = 0; i < b; i++) {
      int b = i;
      ...
    }
  }
  b + x;
}
```

$x \mapsto$ **int**

作用域与符号表

- 符号表举例: C-style
  - Enter a “block”: New symbol table; point to previous

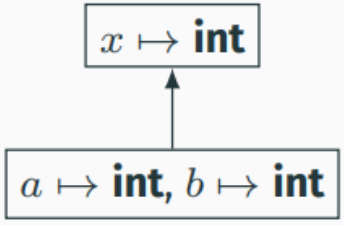
```
int x;
int main() {
  int a = 1;
  int b = 1; {
    float b = 2;
    for (int i = 0; i < b; i++) {
      int b = i;
      ...
    }
  }
  b + x;
}
```



作用域与符号表

- 符号表举例: C-style
  - Enter a “block”: New symbol table; point to previous

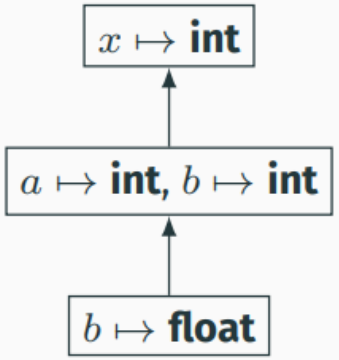
```
int x;
int main() {
  int a = 1;
  int b = 1; {
    float b = 2;
    for (int i = 0; i < b; i++) {
      int b = i;
      ...
    }
  }
  b + x;
}
```



作用域与符号表

- 符号表举例: C-style
  - Enter a “block”: New symbol table; point to previous

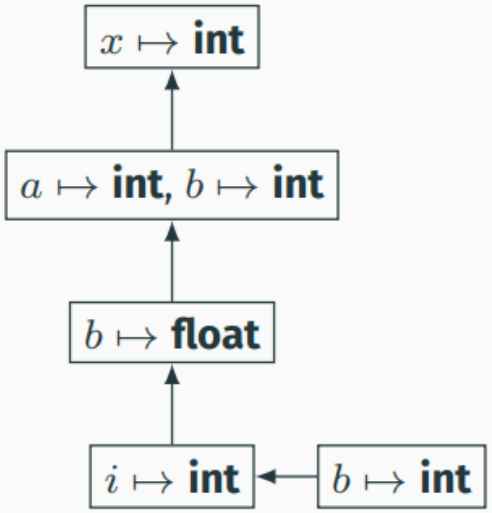
```
int x;
int main() {
  int a = 1;
  int b = 1; {
    float b = 2;
    for (int i = 0; i < b; i++) {
      int b = i;
      ...
    }
  }
  b + x;
}
```



作用域与符号表

- 符号表举例: C-style
  - reach an identifier: lookup in chain of scope

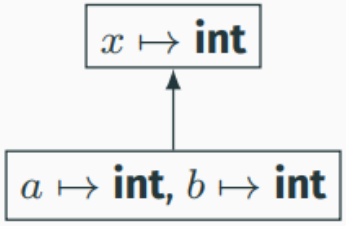
```
int x;
int main() {
  int a = 1;
  int b = 1; {
    float b = 2;
    for (int i = 0; i < b; i++) {
      int b = i;
      ...
    }
  }
  b + x;
}
```



作用域与符号表

- 符号表举例: C-style
  - Leave a block: Local symbol table disappears

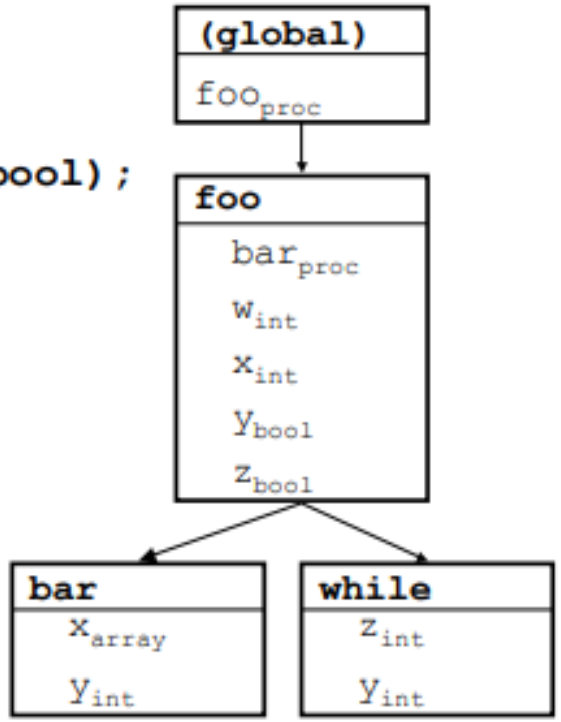
```
int x;
int main() {
  int a = 1;
  int b = 1; {
    float b = 2;
    for (int i = 0; i < b; i++) {
      int b = i;
      ...
    }
  }
  b + x;
}
```



作用域与符号表

- 词法作用域与符号表
  - 每个作用域对应一个符号表，但给定的程序点只有部分绑定关系可见

```
procedure foo(x:int, w:int);
  var z:bool;
  const y:bool = true;
  procedure bar(x:array[5] of bool);
    var y:int;
  begin
    x[y] := z;
  end bar;
begin
  while z do
    var z:int, y:int;
    y := z * x; end;
  output := x + y;
end foo;
```



作用域与符号表

- 嵌套的作用域基本操作
  - 语义分析中遇到一个新的Scope
    - 创建一个新的空符号表
    - 其父级Scope(enclosing block)是当前Scope
    - 新的Scope变为当前的Scope
  - 每当遇到一个声明
    - 将符号加入到符号表中
    - 检查当前scope中是否有重复声明
  - 每当遇到一个符号使用
    - 按照次序搜索scope找到声明：当前、父级、父级的父级
  - 每当退出一个Scope
    - 将父级Scope设置为当前的Scope

作用域与符号表

- 符号表实现

```
class SymTabScope {
public:
  SymTabScope(SymTabScope* enclosingScope);

  void enter(SymTabEntry* newSymbol);
  SymtabEntry* lookup(char* name);
  SymtabEntry* lookup(char* name,
                      SymTabScope*& retScope);
  …
}
```

作用域检查

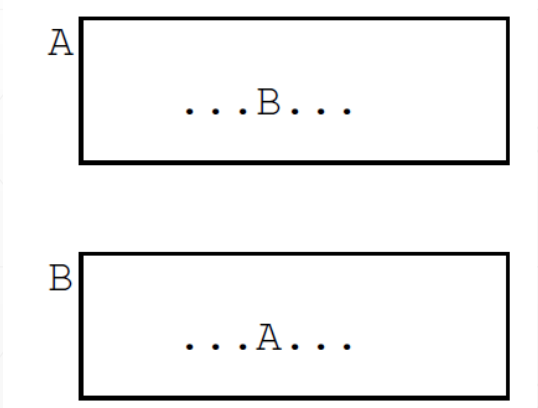
- 符号表实现

A Fancier Symbol Table

• enter_scope()	start/push a new nested scope
• find_symbol(x)	finds current x (or null)
• add_symbol(x)	add a symbol x to the table
• check_scope(x)	true if x defined in current scope
• exit_scope()	exits/pops the current scope

作用域与符号表

- 一遍扫描AST的符号表使用
  - 假设声明总是在使用之前遇到
  - 声明：向符号表中添加语句
  - 使用：将符号链接到最近的声明
  - 无法适用于互相引用、类的属性成员定义



作用域与符号表

- 动态作用域
  - 比较少用，程序的推理更加困难，bash, LaTeX, old Lisp
  - 根据程序状态消解符号，按照调用链条查找

Static scoping

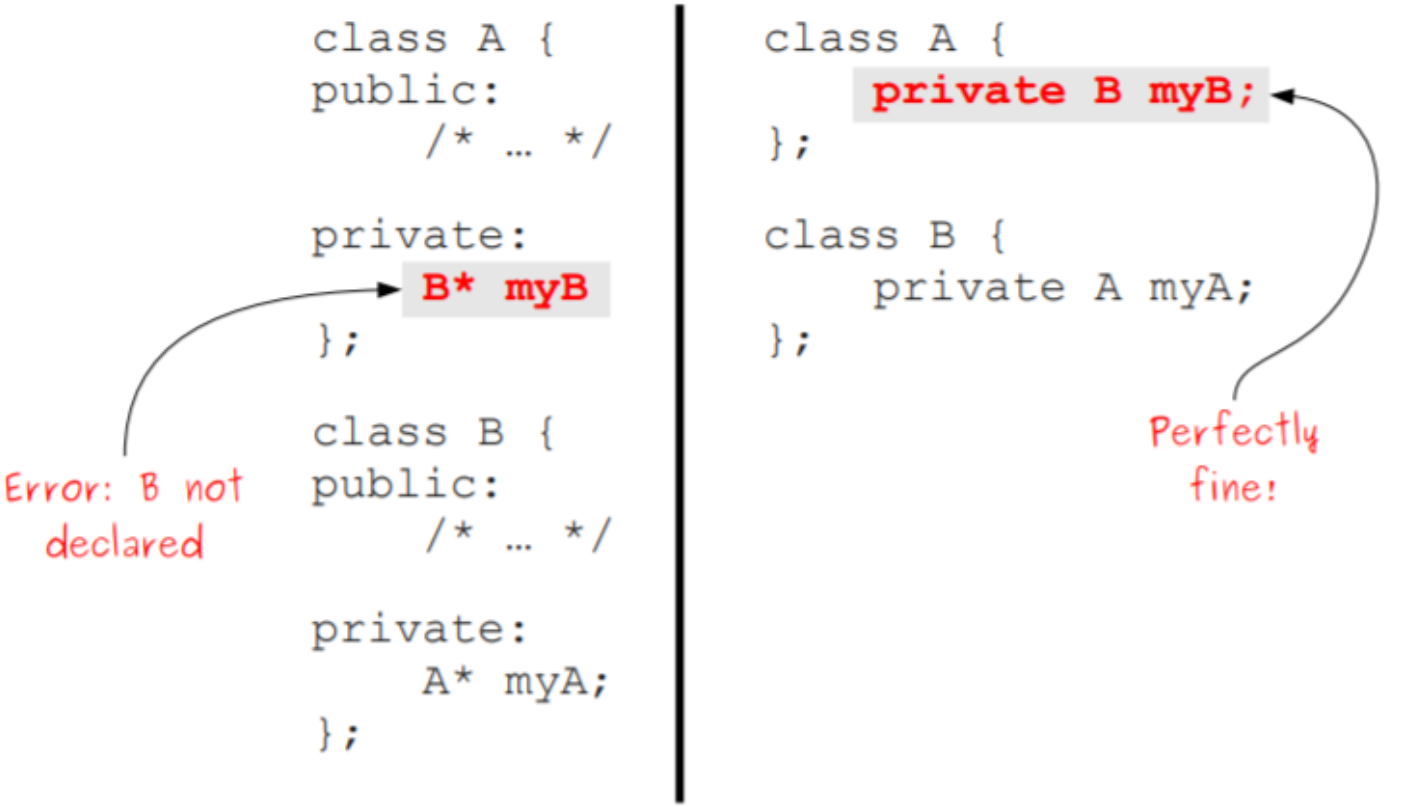
```
int b = 10;

func foo() {
    return b;
}

func main() {
    print foo(); // prints 10
    int b = 5;
    print foo(); // prints 10
}
```

Dynamic scoping

```
int b = 10;

func foo() {
    return b;
}

func main() {
    print foo(); // prints 10
    int b = 5;
    print foo(); // prints 5
}
```

作用域检查

- C++和Java中的作用域

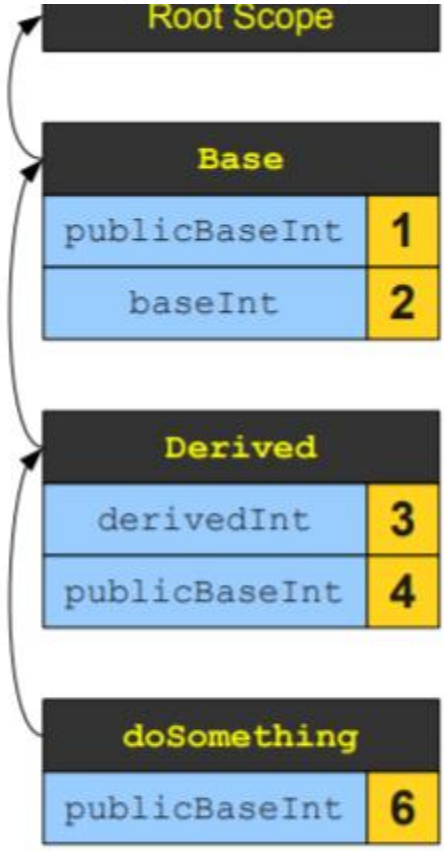


作用域检查

- 继承与作用域

```
public class Base {
    public int publicBaseInt = 1;
    protected int baseInt = 2;
}

public class Derived extends Base {
    public int derivedInt = 3;
    public int publicBaseInt = 4;

    public void doSomething() {
        System.out.println(publicBaseInt);
        System.out.println(baseInt);
        System.out.println(derivedInt);

        int publicBaseInt = 6;
        System.out.println(publicBaseInt);
    }
}
```

```
> 4
  2
  3
  6
```

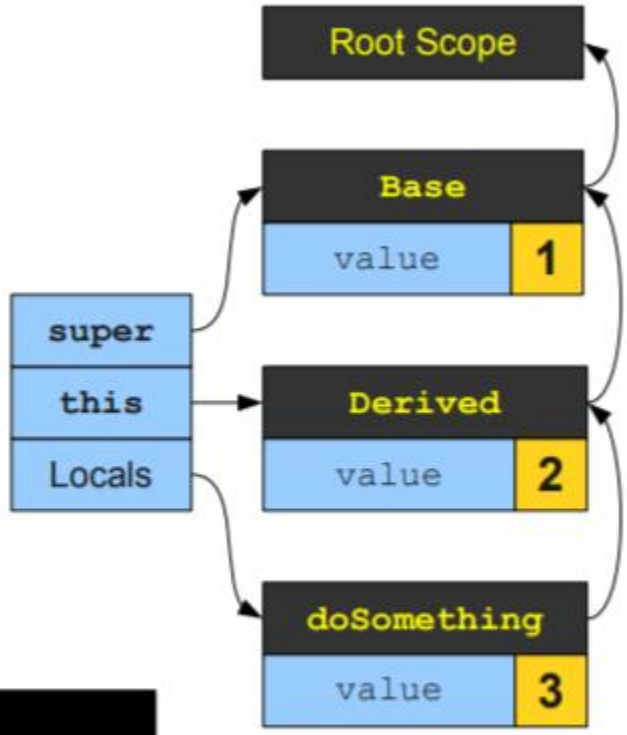


作用域检查

- 继承与作用域

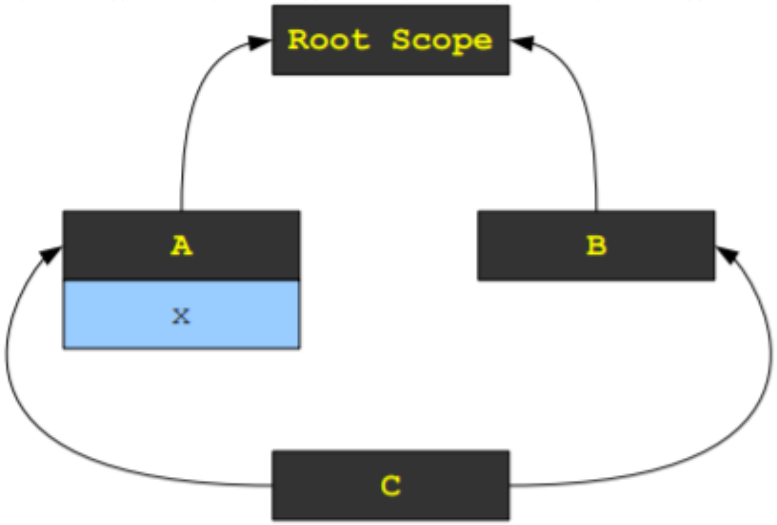
```
public class Base {
    public int value = 1;
}

public class Derived extends Base {
    public int value = 2;

    public void doSomething() {
        int value = 3;
        System.out.println(value);
        System.out.println(this.value);
        System.out.println(super.value);
    }
}
```

```
> 3
  2
  1
```



作用域检查

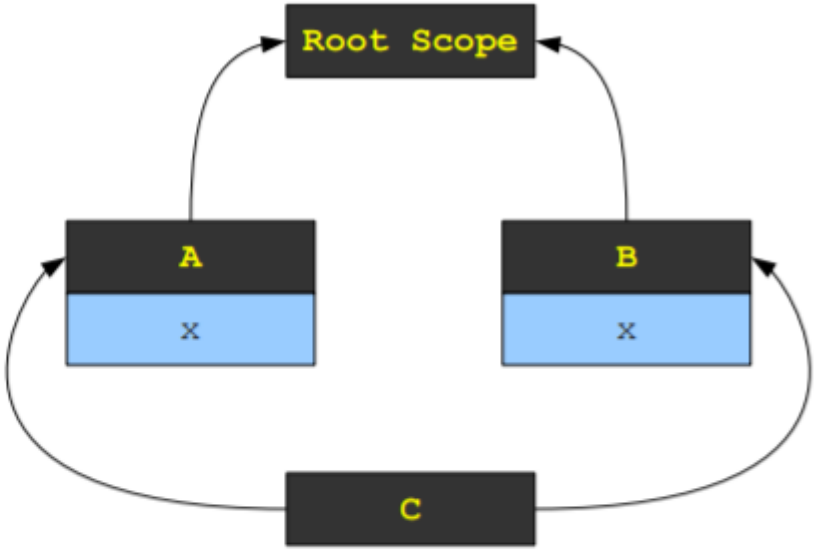
- 多继承与作用域




```
class A {
public:
    int x;
};

class B {


};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

作用域检查

- 多继承与作用域

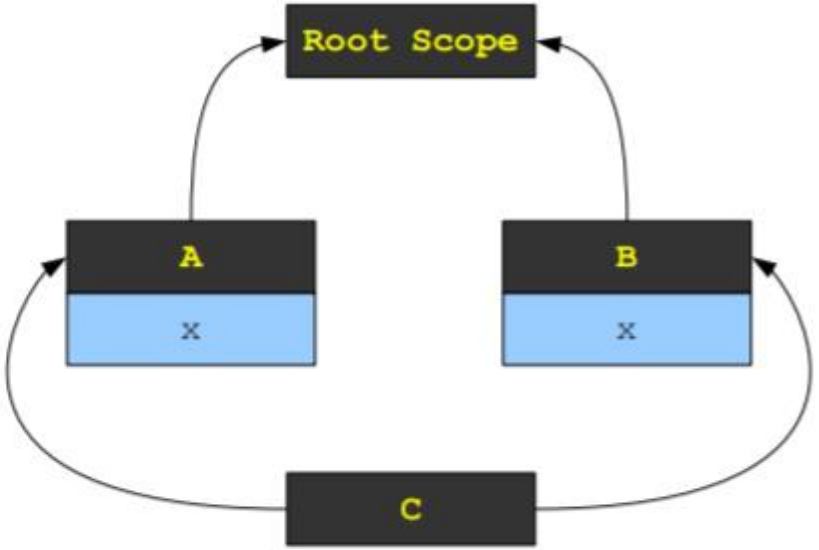



```
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << x << endl;
    }
}
```

Ambiguous – which x?

作用域检查

- 多继承与作用域




```
class A {
public:
    int x;
};

class B {
public:
    int x;
};

class C: public A, public B {
public:
    void doSomething() {
        cout << A::x << endl;
    }
}
```

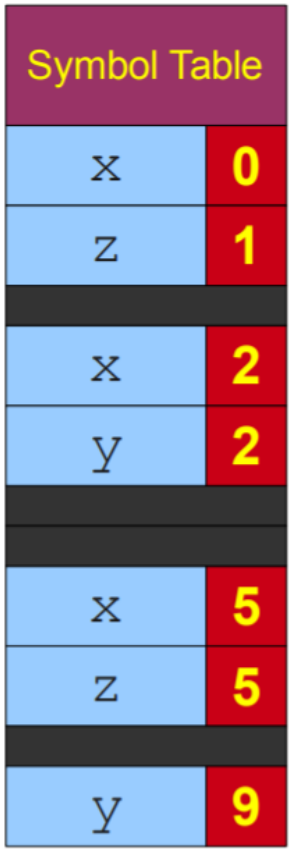
作用域检查

- 作用域举例

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

作用域检查

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z = y@2;
 7:     x = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```



作用域检查

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z = y@2;
 7:     x = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

作用域检查

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z = y@2;
 7:     x = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x@5, y@2, z@5);
16:   }
17: }
```

类型检查/Type Checking

- 类型
  - 考虑下面的代码

```
addi  $r1, $r2, $r3
```

  - 一个函数指针加一个整型？
  - 两个整型相加？

类型检查

- 类型
  - 一个值的集合以及一个可以施加在这些值上的操作集合
  - A data type is a set of values together with a set of operations that can be performed on them.
- 分类
  - **基本类型/Primitive (built-in) types**: 程序设计语言提供的数据类型
  - **复合类型/Composite types:** 使用基本类型组合的类型

类型检查

- 类型
  - Numerics
  - Booleans
  - User-defined enumerations
  - Arrays (static, stack dynamic, heap dynamic)
  - Unions
  - Structures/Records/Objects
  - Tuples/Lists
  - References/Pointers
  - etc.

类型检查

- 类型体系
  - 类型构建方法、类型等价性定义、类型推断规则、类型检查规则等
- 类型构建
  - 基于集合理论：笛卡尔积、并、超集、子集…

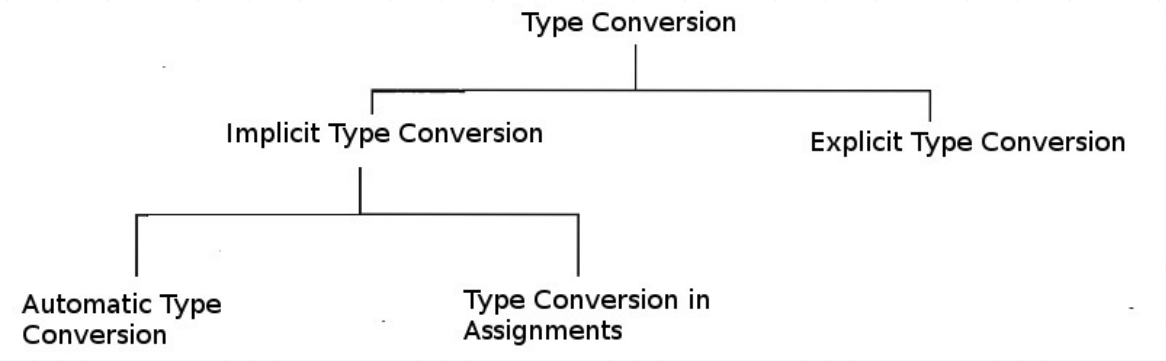
```
struct IntCharReal{
    int i;
    char c;
    double r;
};
```

```
union IntOrReal{
    int i;
    double r;
};
```

$U \times V = \{(u, v) \mid u \text{ is in } U \text{ and } v \text{ is in } V\}$

类型检查

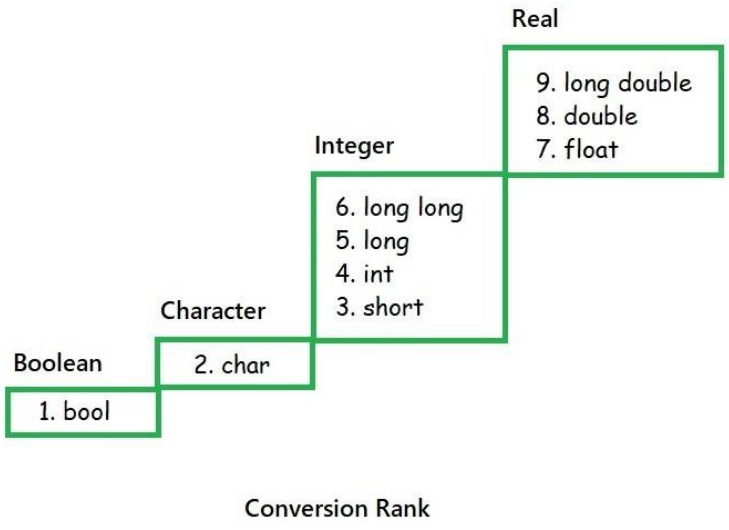
- 类型转换
  - Type Coercion: 自动/隐式类型转换
  - Type Casting: 显式强制转换




```
double  d;
long    l;
int     i;

if (d > i)    d = i;
if (i > l)    l = i;
if (d == l)   d *= 2;
```

类型检查

- 类型提升/Type Promotion
  - automatically converted into another data type representing a superset of the original type



类型检查

- 类型提升/Type Promotion

Conditions for type conversion

Conditions Met	Conversion
Either operand is of type `long double`.	Other operand is converted to type `long double`.
Preceding condition not met and either operand is of type `double`.	Other operand is converted to type `double`.
Preceding conditions not met and either operand is of type `float`.	Other operand is converted to type `float`.
Preceding conditions not met (none of the operands are of floating types).	Operands get integral promotions as follows: - If either operand is of type `unsigned long`, the other operand is converted to type `unsigned long`. - If preceding condition not met, and if either operand is of type `long` and the other of type `unsigned int`, both operands are converted to type `unsigned long`. - If the preceding two conditions aren't met, and if either operand is of type `long`, the other operand is converted to type `long`. - If the preceding three conditions aren't met, and if either operand is of type `unsigned int`, the other operand is converted to type `unsigned int`. - If none of the preceding conditions are met, both operands are converted to type `int`.

https://docs.microsoft.com/en-us/cpp/cpp/standard-conversions?view=msvc-170

类型检查

- 类型等价
  - 名字类型等价
    - views distinct type names as distinct types: two types are name equivalent if and only if they have the same type

```
int i = 42; int j = 51;
```

  - 结构类型等价
    - Two type expressions are structurally equivalent if they have the same structure, i.e., if both apply the same type constructor to structurally equivalent type expressions.

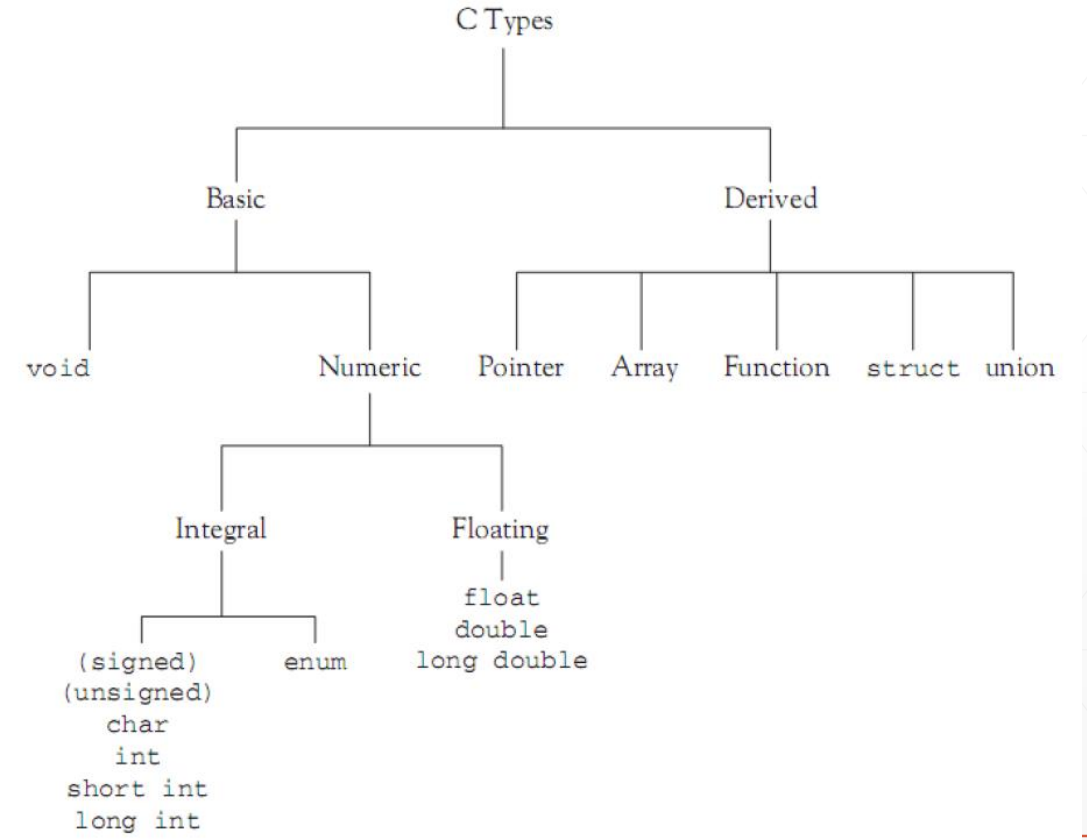
```
struct A { int a; int b;};
struct B { int c; int d;};
```

类型检查

- 根据类型系统对语言分类
  - 静态类型
    - 类型检查在编译时完成(C, Java)
  - 动态强类型
    - 类型检查在动态运行时完成（Scheme, Python）
  - 无类型
    - 不做类型检查（机器语言）

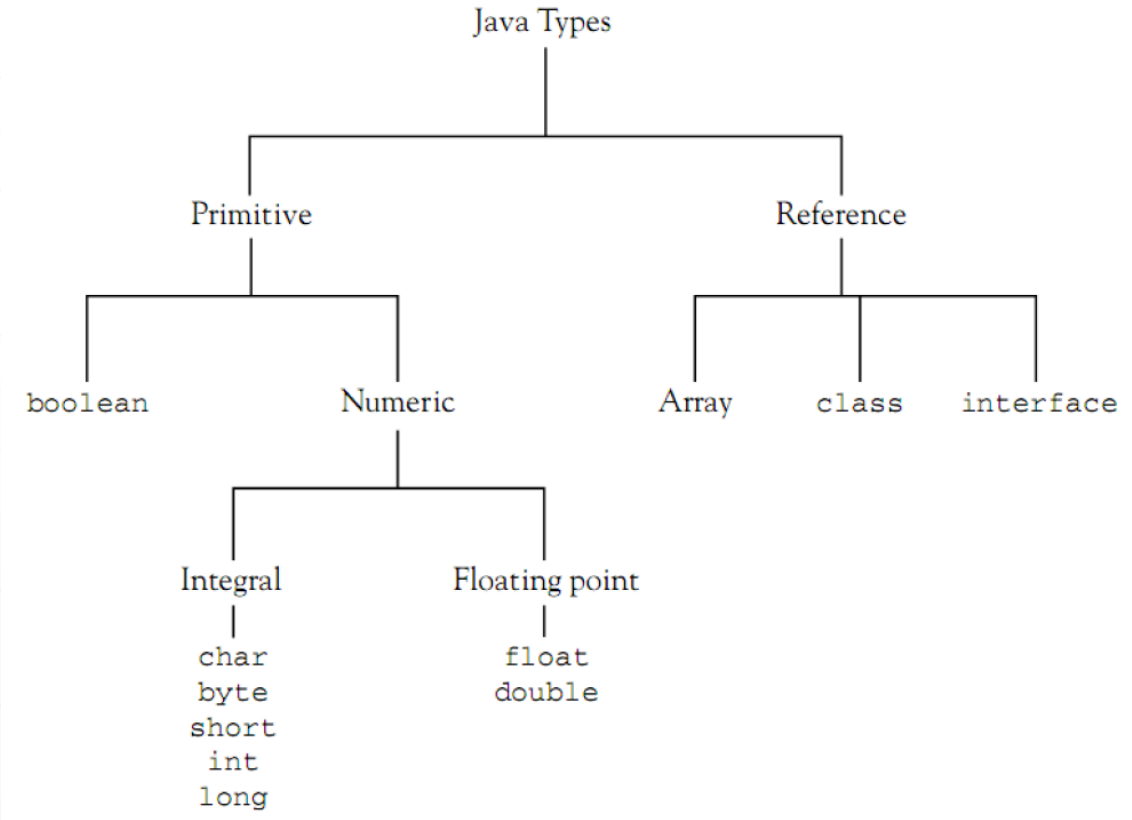
类型检查

- C语言


C Types
Basic
Derived
void
Numeric
Pointer
Array
Function
struct
union
Integral
Floating
(signed)
(unsigned)
char
int
short int
long int
enum
float
double
long double

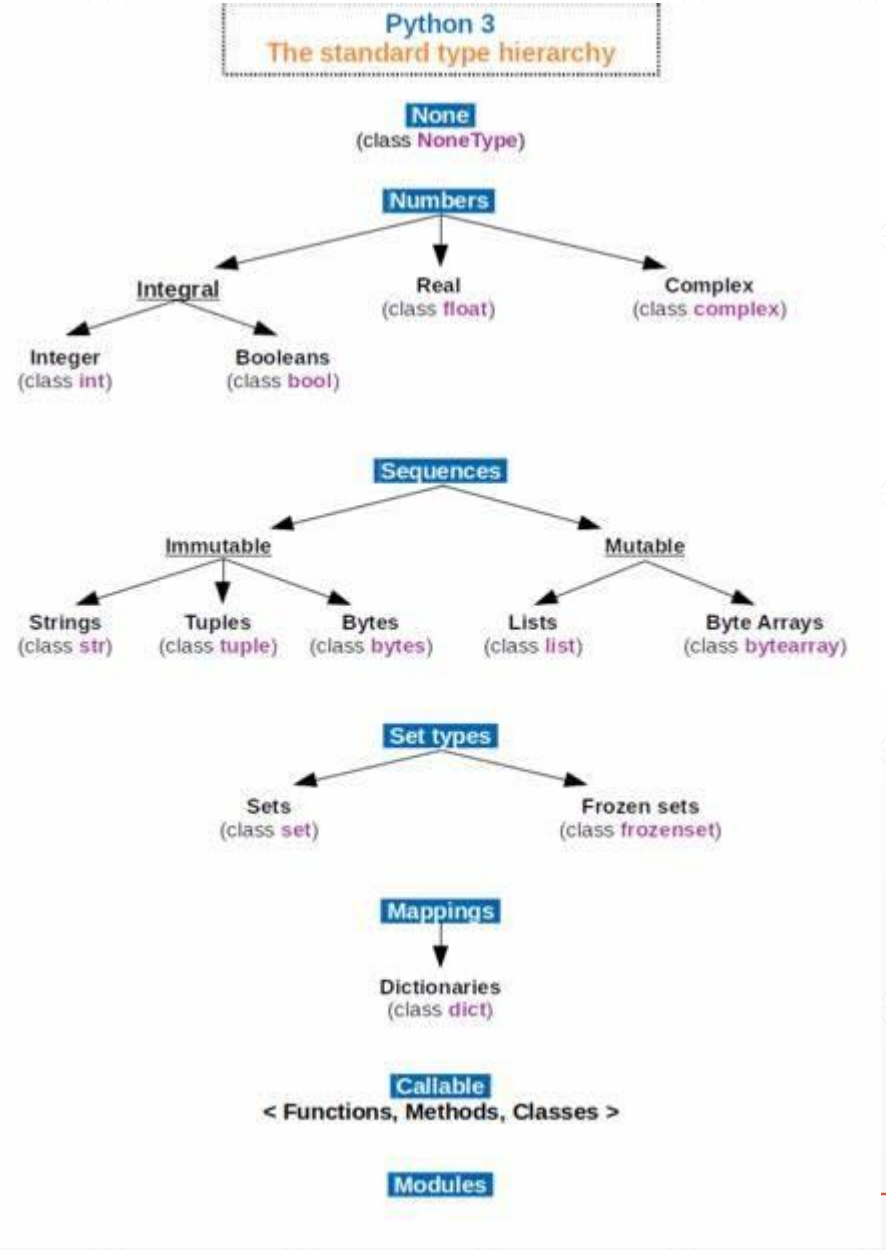
类型检查

- Java语言


Java Types
Primitive
Reference
boolean
Numeric
Array
class
interface
Integral
Floating point
char
byte
short
int
long
float
double

类型检查

- Python语言



类型检查

- 使用静态类型的理由
  - **执行效率**：编译器可以更加有效地分配内存
  - **翻译效率**：编译出来的代码规模更小
  - **安全可靠**：编译阶段发现更多的错误，减少运行时错误
  - **可读性**：便于阅读理解和减少歧义性
  - **其他**：方便工具对程序进行验证

类型检查

- Type Checking
  - the activity of ensuring that the operands of an operator are of compatible types
- A compatible type is one that is
  - either legal for the operator
  - or allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type

类型检查

- 类型检查
  - Type checking aims to verify that operations in a program code are, in fact, permissible on their operand values.
- 类型推断
  - The language provides a set of base types and a set of type constructors;
  - The compiler uses type expressions to represent types definable by the language.

类型检查

- 类型断言

$$\Gamma \mid\!- \exp : \tau$$

- Γ 类型环境，$\Gamma=\{ x:\sigma , . . . \}$ 指定变量的类型
- exp 是程序中的表达式
- τ 表示exp的类型

$$\Gamma \vdash M : A \qquad \Gamma \vdash \diamond \qquad \Gamma, \Gamma' \vdash e$$

类型检查

- 类型规则

$$\frac{(Rule\ name)\ \ (Annotations)\ \Gamma_1 \vdash \mathfrak{I}_1 \ldots \Gamma_n \vdash \mathfrak{I}_n\ (Annotations)}{\Gamma \vdash \mathfrak{I}}$$

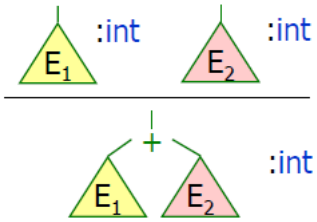
前提(premise)

结论(conclusion)

$$\frac{(Val\ n)\ \ (n = 0,\ 1,\ \ldots)\ \Gamma \vdash \diamond}{\Gamma \vdash n : Nat} \qquad \frac{(Val\ +)\ \Gamma \vdash M : Nat \quad \Gamma \vdash N : Nat}{\Gamma \vdash M + N : Nat}$$

$$\frac{}{A \mid\!- \textbf{true} : bool}$$

$$\frac{A \mid\!- E_1 : int \quad A \mid\!- E_2 : int}{A \mid\!- E_1 + E_2 : int}\ (+)$$



$$\frac{A \mid\!- E : bool \quad A \mid\!- E_1 : T \quad A \mid\!- E_2 : T}{A \mid\!- (E\ ?\ E_1 : E_2) : T}\ (\text{if-rule})$$

Premises (a.k.a., antecedant)

Conclusion (a.k.a., consequent)

类型检查

- 类型规则
  - if-else语句

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}$$

  - 函数调用语句

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

类型检查

- 类型导出

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash \diamond}}{\varnothing \vdash 1 : Nat} \quad \dfrac{\dfrac{}{\varnothing \vdash \diamond}}{\varnothing \vdash 2 : Nat}}{\varnothing \vdash 1 + 2 : Nat}$$

Γ形式良好(well formed)

$$\dfrac{\dfrac{}{A1 \vdash b : bool}}{A1 \vdash !b : bool} \quad \dfrac{\dfrac{}{A1 \vdash 2 : int} \quad \dfrac{}{A1 \vdash 3 : int}}{A1 \vdash 2+3 : int} \quad \dfrac{}{A1 \vdash x : int}$$

$$A1 \vdash (!b \ ? \ 2+3 \ : \ x) : int$$

类型检查

- 常见问题

E : T?

E : _?

_ : T?

类型检查

```
char *X;                    main(){
char **f(){                    printf("%c\n", (*f())[2]);
   X = "GOTCHA!"                                  /* legal??? */
   return &X;               }
}
```

Program Code	***Type Expression***	***Rule***
f	**()→ptr(ptr(char))**	**symbol table lookup**
f()	**ptr(ptr(char))**	**if** *e* : $T_1 \rightarrow T_2$ and e_1 : T_1 **then** $e(e_1)$: T_2
***f()**	**ptr(char)**	**if** *e* : *ptr*(*T*) **then** **e* : *T*
***f()**	**array(char)**	**if** *e* : *ptr*(*T*) **then** *e* : *array*(*T*)
(*f())	**array(char)**	**if** *e* : *T* **then** (*e*) : *T*
2		**int** base type
(*f())[2]	**char**	**if** e_1 : *array*(*T*) and e_2 **:int then** $e_1[e_2]$: *T*