Franklin’s 13 virtues and Puritan ethics

Temperance Silence
Order Resolution
Frugality Industry
Sincerity Justice
Moderation Cleanliness

(11) Tranquility: Be not disturbed at trifles, or at accidents common or unavoidable .

沉着：不要因为琐事或者难以避免的不测烦恼。

When we are unable to find tranquility within ourselves, it is useless to seek it elsewhere.

(12) Chastity: rarely use venery but for health or offspring, never to dullness, weakness, or the injury of your own or another’s peace or reputation.

贞洁：切忌纵欲过度，以免伤害身体或者有损自己或他人的安宁和名誉。

(13) Humility: Imitate Jesus and Socrates.

谦虚：效法耶稣和苏格拉底。

Compiling and Running of Program

Dr. Zheng Xiaojuan
Professor

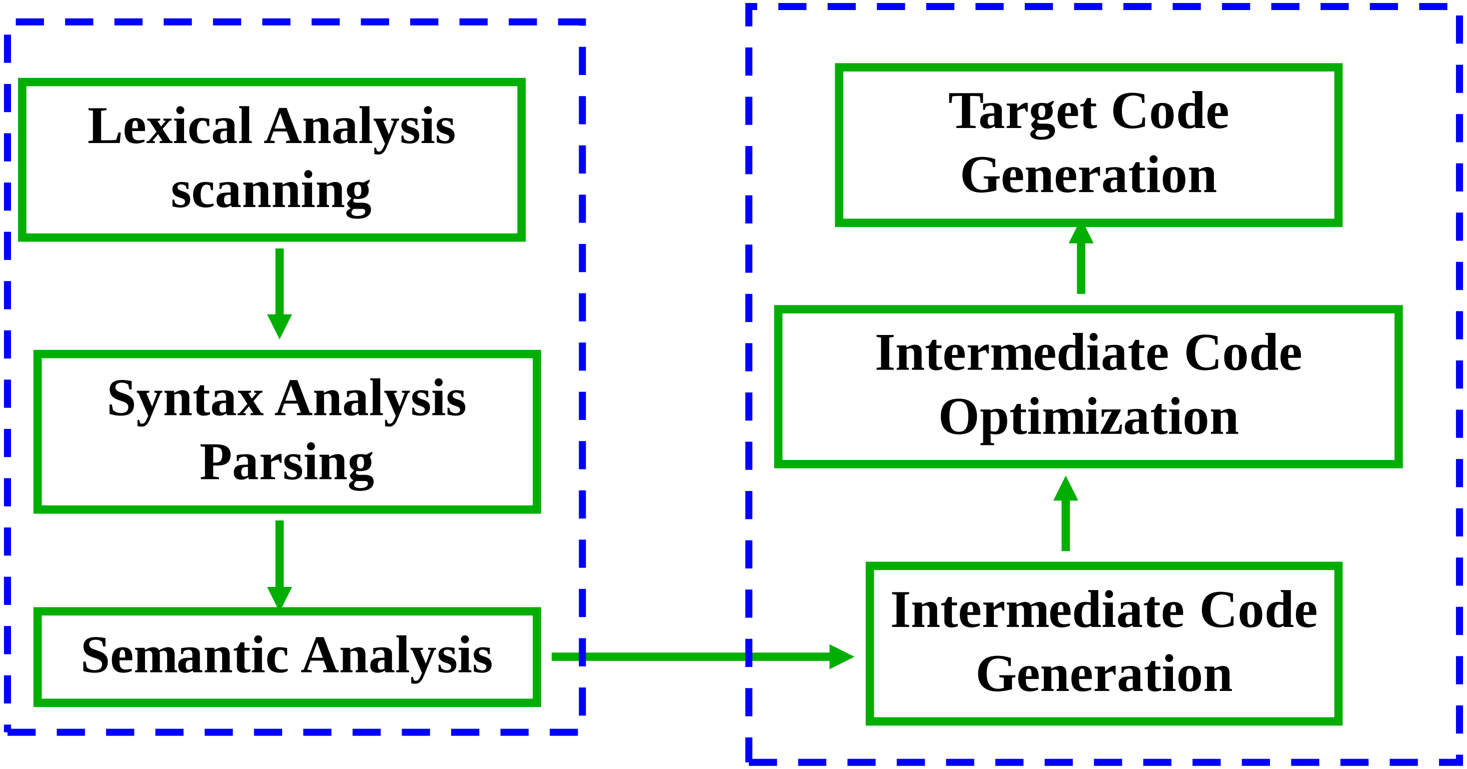
December. 2019

Main Content of Chapter 6

§ 6. Semantic Analysis

- 6.1 Overview of Semantic Analysis
- 6.2 Symbol Table
- 6.3 Semantic Analysis of Types
- 6.4 Semantic Analysis of Declaration
- 6.5 Semantic Analysis of Body
- 6.6 Attribute Grammar and Action Grammar

Role of Semantic Analysis in a Compiler



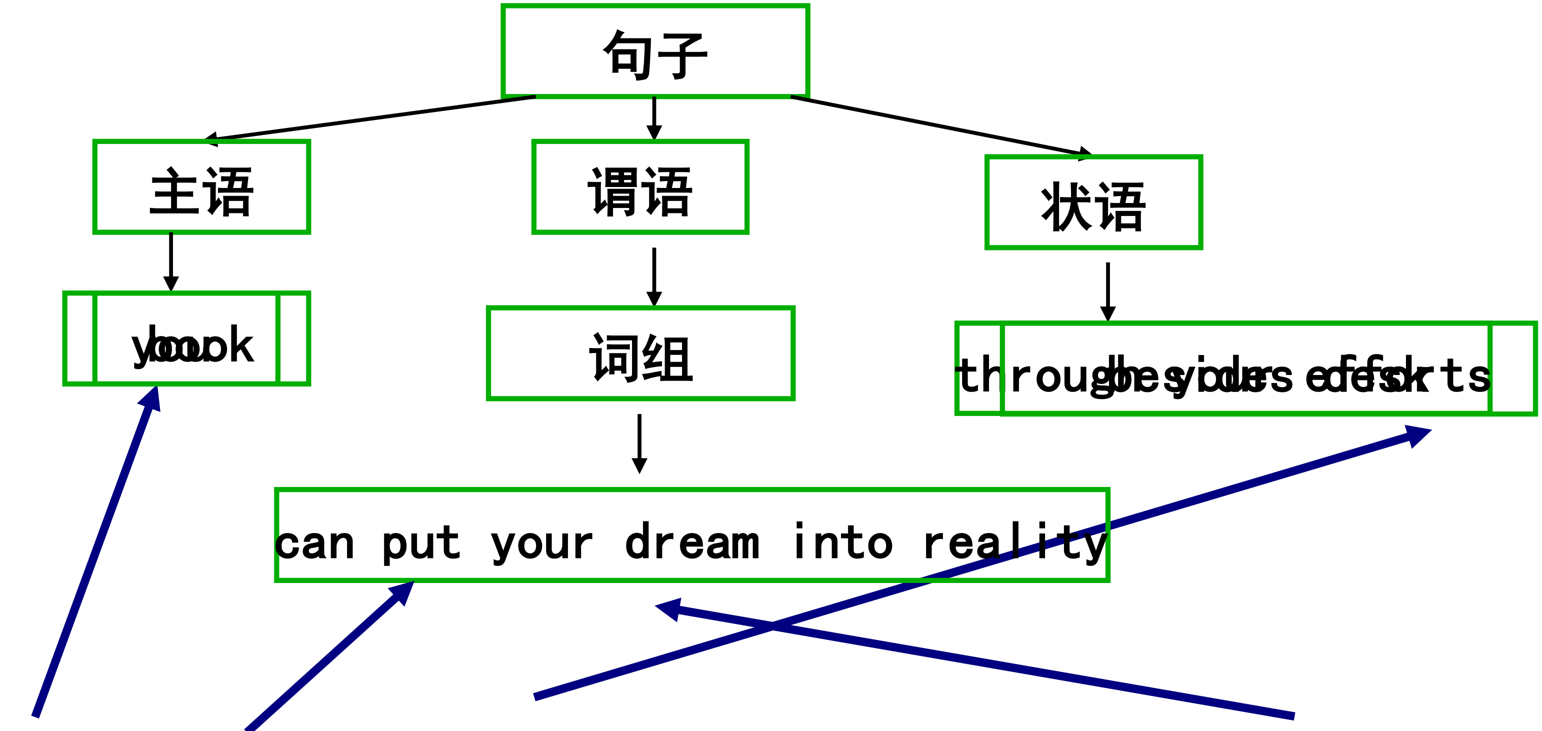
analysis/front end

synthesis/back end

Review the process of Natural Language Translation

Informa
Northea

You can put your dream into reality through your efforts!



句子

主语

谓语

状语

you

book

词组

through your efforts

besides desk

can put your dream into reality

你　能够　通过你的努力　实现你的梦想！

6.1 Overview of Semantic Analysis

- The differences between Syntax and Semantics
- The necessity of semantic analysis
- Classification of Semantics for Programming Languages
- How to describe semantics of a programming language formally?
- The main task of Semantic Analysis

Differences between Syntax & Semantics

- **Syntax:**
  - **Rules for describing the *structure* of a well-defined program;**
- **Semantics:**
  - **The *meaning* of syntactic structures of a well-defined program;**

The Necessity of Semantic Analysis

- **A syntactically correct program can not guarantee that it is *meaningful*!**
- **The semantic errors for programming languages**
  - **Undeclared identifier**
  - **The type of operands does not match for their operator;**
  - **…..**

Classification of Semantics for Programming Languages

- Static semantics
  - Can be checked during compilation
  - For instance： Undeclared identifier
- Dynamic semantics
  - Can only be checked during execution
  - For instance： divide zero

Formal Description of Semantics

- Formal Semantics for Programming Languages
  - Attribute Grammar (for static semantics)
  - Operational Semantics
  - Denotational Semantics
  - Algebra Semantics
  - Axiomatic Semantics
- Not as mature as Syntax Analysis
- Postgraduate Course

Main Task of Semantic Analysis

- **Establish *Symbol Table* from declaration part**
  - **Store *attributes of identifiers* for checking semantic errors and code generation;**
- ***Check semantic errors* throughout the whole program**
  - **Use after declaration**
  - **Type related**
- **Normally Semantic Analysis will be done along with parsing;**

Semantic Errors

- Use after Declaration（ Declaration related ）
  - 标识符没有声明;
  - 重复声明;
- 如何检查?
  - 每当遇到新声明的标识符, 查符号表
    - 如果当前有效的所有标识符中有相同名字的, 则是重复声明错误;
    - 否则生成它的属性信息, 保存到符号表中;
  - 每当遇到标识符的使用, 查符号表
    - 如果没有找到, 说明该标识符没有声明;
    - 否则， 得到该标识符的属性, 进行进一步分析;

Semantic Errors

- Type related errors（ Type related ）
  - 各种条件表达式的类型不是布尔类型;
  - 运算符的分量类型不相容;
  - 赋值语句左右类型不相容;
  - 形实参类型不相容;
  - 函数调用参数个数不同;
  - f(.....)中f不是函数标识符;
  - 下标表达式(数组)不是合法类型;
  - 函数说明和函数返回类型不相容;
  - ……

Main Content of Chapter 6

§ 6. Semantic Analysis

- 6.1 Overview of Semantic Analysis
- 6.2 Symbol Table
- 6.3 Semantic Analysis of Types
- 6.4 Semantic Analysis of Declaration
- 6.5 Semantic Analysis of Body
- 6.6 Attribute Grammar and Action Grammar

6.2 Symbol Table

- **What is a symbol table?**
  - **Symbol table can be seen as a mapping from identifier name to its attributes;**
  - **stating the identifiers declared in a program and their attributes;**

Identifier Name	Attributes
x	int, variable, ……
p	void, function, (int i), ……
…	…….

- 计算思维的典型方法
  - □ 理论可实现 vs. 实际可实现
  - □ 理论研究重在探寻问题求解的方法，对于理论成果的研究运用又需要在能力和运用中作出**权衡**

程序设计语言采用上下文无关文法

- 程序设计语言不是上下文无关语言，甚至不是上下文有关语言
- L={αcα| α∈{a, b}*}不能由上下文无关文法产生，甚至连上下文有关文法也不能产生，只能由0型文法产生
  - 标识符引用
  - 过程调用过程中，“形-实参数的对应性"(如个数，顺序和类型一致性)
- 对于现今程序设计语言，在编译程序中，仍然采用上下文无关文法来描述其语言结构

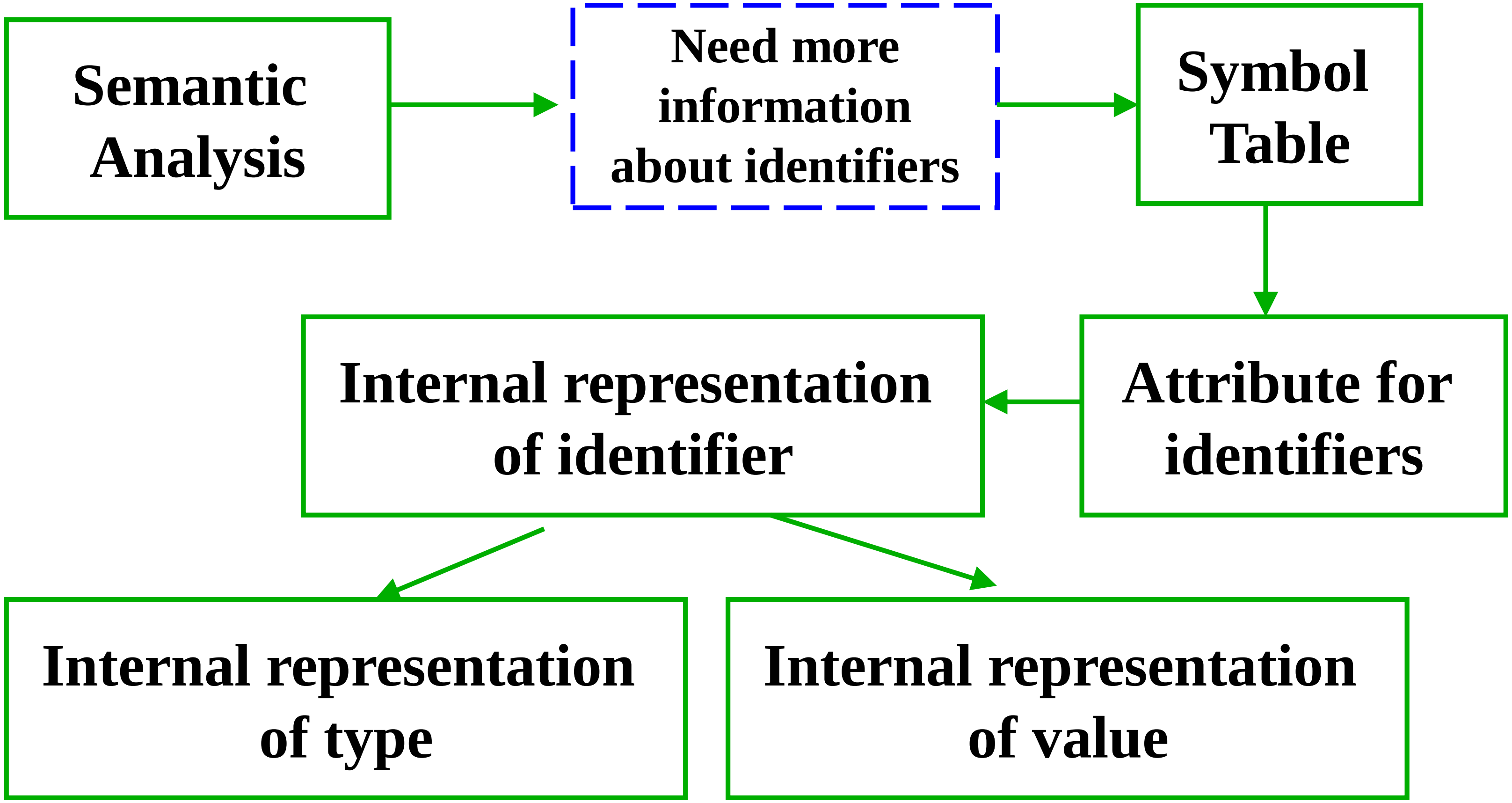
6.2 Symbol Table

- **Why we need symbol table during Semantic Analysis?**
  - **From the token definition of an identifier, we only know the *name of the identifier*, but have no idea about its attributes such as type, what kind of identifier it is, ……**
  - ***More information* about identifiers are needed to do semantic analysis, to check semantic errors;**

6.2 Symbol Table

- **In order to represent the attributes of identifiers, we need to establish**
  - **Internal representation of Identifiers**
  - **Internal representation of Types**
  - **Internal representation of Values**
- **Organization of Symbol Table**
  - **Scope of identifiers**
  - **Localized Symbol Table**
  - **Globalized Symbol Table**
  - **Interface Functions for Symbol Table**

Knowledge Relation Graph



Semantic Analysis

Need more information about identifiers

Symbol Table

Attribute for identifiers

Internal representation of identifier

Internal representation of type

Internal representation of value

Internal representation of Identifiers

- **Different Types of Identifiers**
  - Constant name
  - Type name
  - Variable name
  - Function name
  - Procedure name
  - Field name

```
typedef enum {constKind, typeKind, varKind,
                          procKind, funcKind, fieldKind} idKid;
```

Internal representation of Identifiers

- **Attributes of Different Types of Identifiers**
  - **Constant: (类型，值)**
  - **Type: (类型）**
  - **Variable: (类型， 层数，偏移)**
  - **Function: (返回类型， 形参定义， 代码地址， 空间大小, …….)**
  - **Procedure: (形参定义， 代码地址， 空间大小, …….)**
  - **Field: (类型， 偏移， 所在结构类型)**

Attributes for Constant Identifier

TypePtr	Kind	Value
	constKind	

- **TypePtr**
  - **point to the internal representation of the type of the constant;**
- **Kind**
  - **The kind of identifier, here should be constKind;**
- **Value**
  - **The value of the constant;**

Attributes for Type Identifier

TypePtr	Kind
	typeKind

- **TypePtr**
  - **point to the internal representation of the type;**
- **Kind**
  - **The kind of identifier, here should be typeKind;**

Attributes for Variable Identifier

TypePtr	Kind	Access	Level	Offset
	varKind			

- **TypePtr**
  - **point to the internal representation of the type of the variable;**
- **Kind**
  - **The kind of identifier, here should be varKind;**
- **Access: (dir, indir);**
- **Level: 层数**
- **Offset:偏移量**

Attributes for Field Identifier

TypePtr	Kind	Offset	Ptr
	fieldKind		

- **TypePtr**
  - point to the internal representation of the type of the field;
- **Kind**
  - The kind of identifier, here should be fieldKind;
- **Offset: 该域名针对所在结构类型的偏移量;**
- **Ptr: refers to next field;**

Attributes for Real Proc/Func Identifier (实在过程或者函数）

TypePtr	Kind	Class	Level	Param	Code	Size	Forward
	routKind	actual					

- TypePtr: point to the internal representation of the type of func(void for proc);
- Kind: the kind of identifier, here should be routKind;
- Class: actual refers to real procedure or function;
- Level: 层数where the identifier defined;
- Param: refers to a list of information for formal parameters;
- Code: refers to the start address of the target code for the procedure or function;
- Size: the size of the space;
- Forward: true for forward declaration;

Attributes for Formal Proc/Func Identifier (形参过程或者函数)

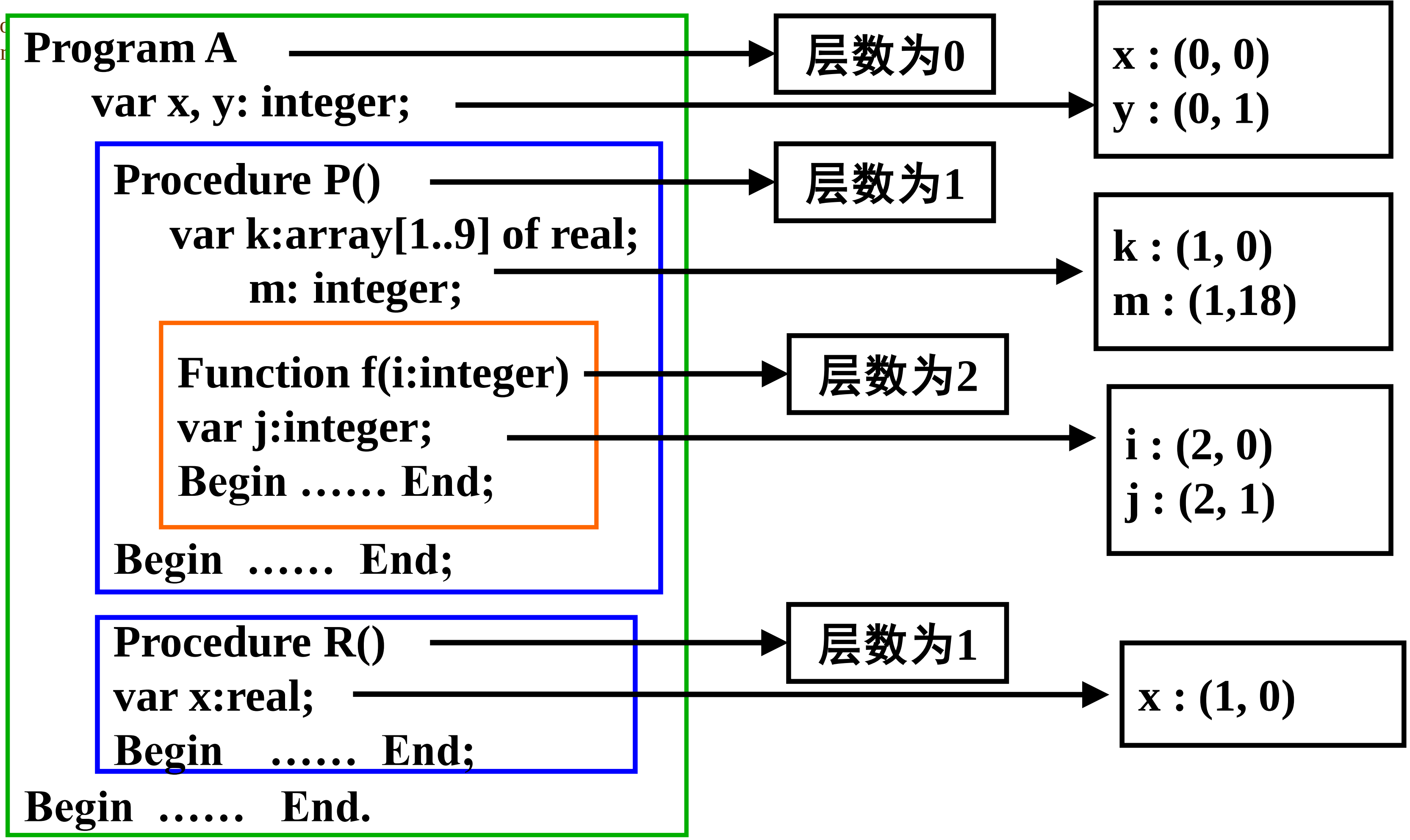
TypePtr	Kind	Class	Level	Param	Offset
	routKind	formal			

- TypePtr: point to the internal representation of the type of func(void for proc);
- Kind: the kind of identifier, here should be routKind;
- Class: formal refers to formal procedure or function;
- Level: 层数where the identifier defined;
- Param: refers to a list of information for formal parameters;
- Offset: 在形参列表中的偏移;

Levels and offsets for Variables (including parameters)

- 层数(level)
  - 过程/函数的定义可以嵌套
  - 主程序的层数为0;
- 偏移量(offset)
  - 执行过程/函数的调用时，需要为其中的变量分配空间;
  - 偏移量指的是变量针对其所在过程/函数的空间的首地址的偏移量;

Levels and offsets for Variables (including parameters)



```
Program A
    var x, y: integer;
    Procedure P()
        var k:array[1..9] of real;
            m: integer;
        Function f(i:integer)
        var j:integer;
        Begin …… End;
    Begin  ……  End;
    Procedure R()
    var x:real;
    Begin   ……  End;
Begin  ……   End.
```

层数为0

x : (0, 0)
y : (0, 1)

层数为1

k : (1, 0)
m : (1,18)

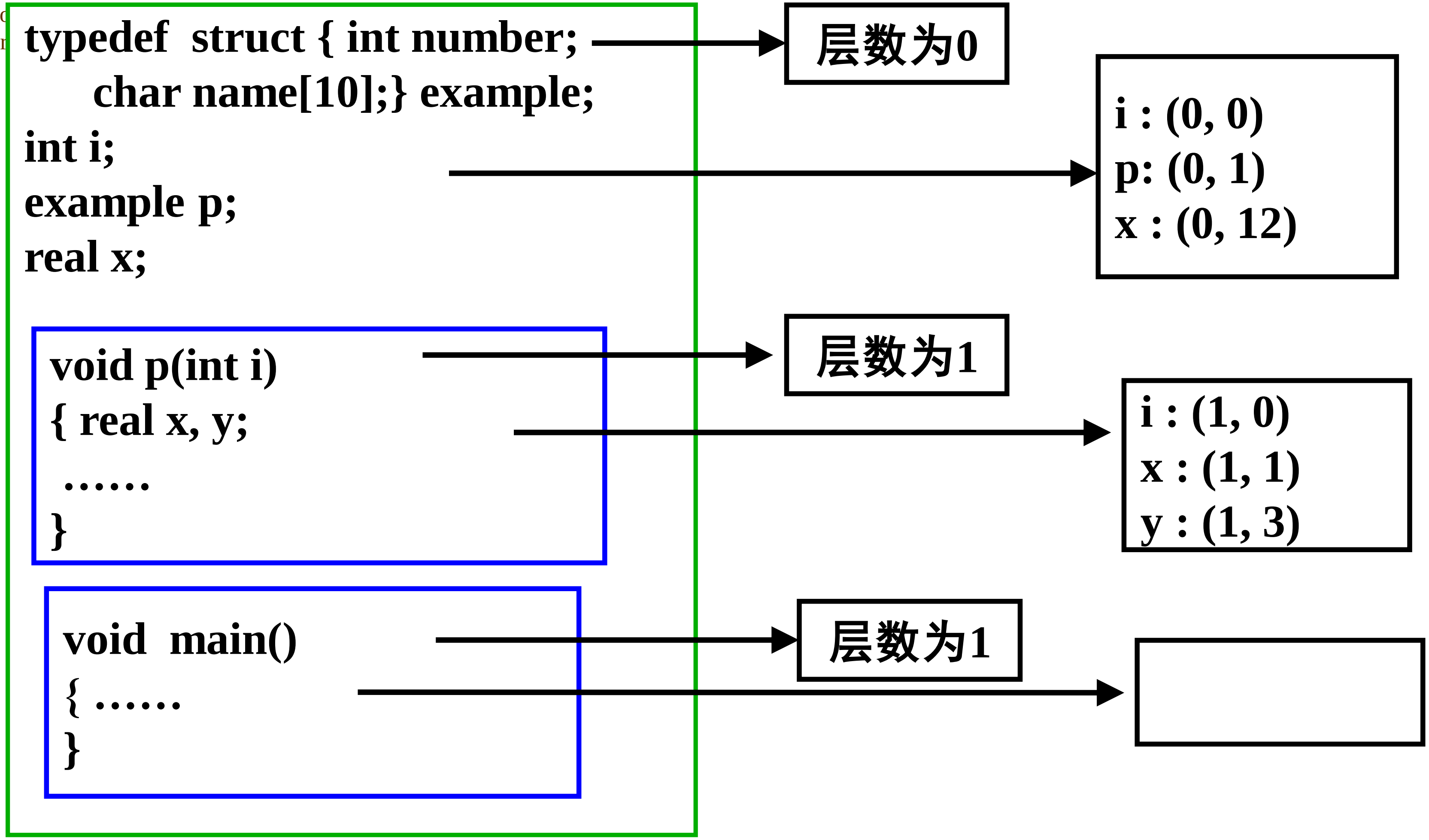
层数为2

i : (2, 0)
j : (2, 1)

层数为1

x : (1, 0)

Levels and offsets for Variables (including parameters)



```
typedef  struct { int number;
      char name[10];} example;
int i;
example p;
real x;

void p(int i)
{ real x, y;
 ……
}

void  main()
{ ……
}
```

层数为0

i : (0, 0)
p: (0, 1)
x : (0, 12)

层数为1

i : (1, 0)
x : (1, 1)
y : (1, 3)

层数为1

Internal representation of Types

Why we need Internal Representation for Types?

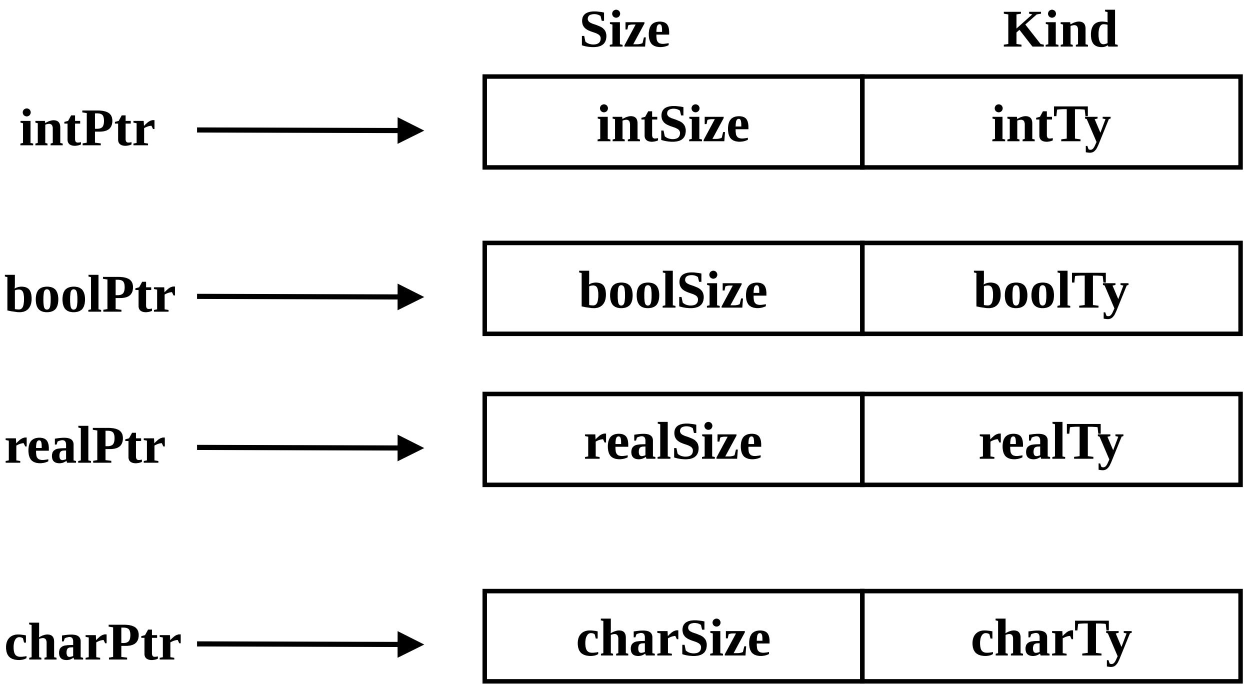
-- Type is the important attribute for ID;

-- Type checking is most important part of Semantic Analysis;

-- The structure of a type is necessary for type checking;

- **Types for Programming Languages**
  - **Basic type**
    - **Integer**
    - **Real**
    - **Bool**
    - **Char**
  - **Structural type**
    - **Array**
    - **Structure**
    - **Union**
  - **Enumeration(枚举类型)**
  - **Pointer**

Internal representation of Basic Types



	Size	Kind
intPtr →	intSize	intTy
boolPtr →	boolSize	boolTy
realPtr →	realSize	realTy
charPtr →	charSize	charTy

Internal representation of Array

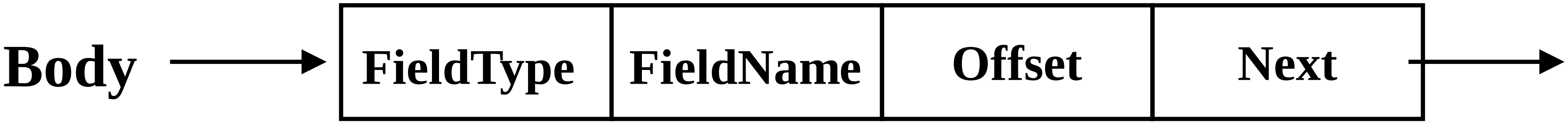
Size	Kind	Low	Up	ElemTy
	arrayTy			

- size = sizeof(ElemTyp)× (Up-Low+1)
- ElemTy: pointer to its element type;
- array [2..9] of integer;
- (8, arrayTy, 2, 9, intPtr)
- int [7];
- (7, arrayTy, 0, 6, intPtr)

Internal representation of Structure

Size	Kind	Body
	strutTy	

- Body: 指向结构体中域定义链表
- Size: 所有域的类型的size的总和;

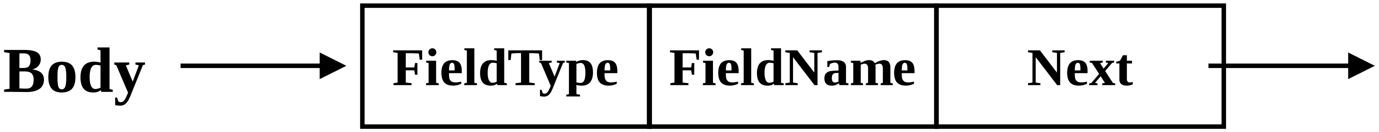


```
typedef struct { int  i;   char name[10]; real  x; }example;
```

Internal representation of Union

Size	Kind	Body
	unionTy	

- Body: 指向联合体中域定义链表
- Size: 所有域的类型的size中最大值;



```
typedef union{ int  i;   char name[10]; real  x; } test;
```

Internal representation of Pointer

Size	Kind	BaseType
	ptrTy	

- BaseType: 指向指针的基类型;
- Size:指针的空间大小(通常为一个单元);

Internal representation of Enumeration

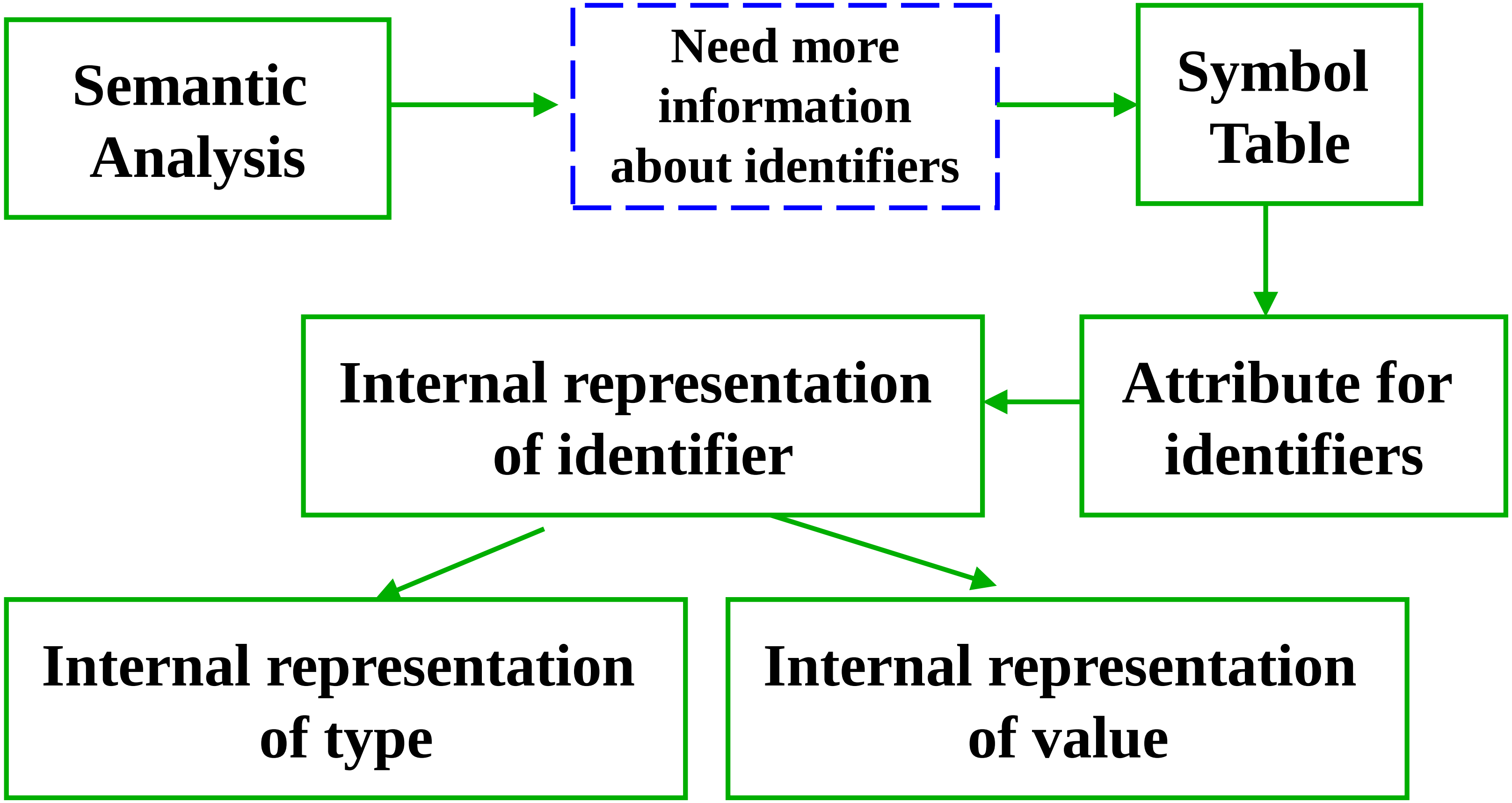
Size	Kind	EList
	enumTy	

- EList: 枚举常量链表;
- Size:枚举类型值的空间大小(通常为一个单元);

Internal representation of Values

- 可表示的值
  - Integer
  - Real
  - False, true(通常对应0和1)
  - char: ASCII码值
  - 枚举类型：对应整数
- 结构值
  - 数组
  - 结构
  - 联合
  - 指针

Knowledge Relation Graph


Semantic Analysis
Need more information about identifiers
Symbol Table
Attribute for identifiers
Internal representation of identifier
Internal representation of type
Internal representation of value

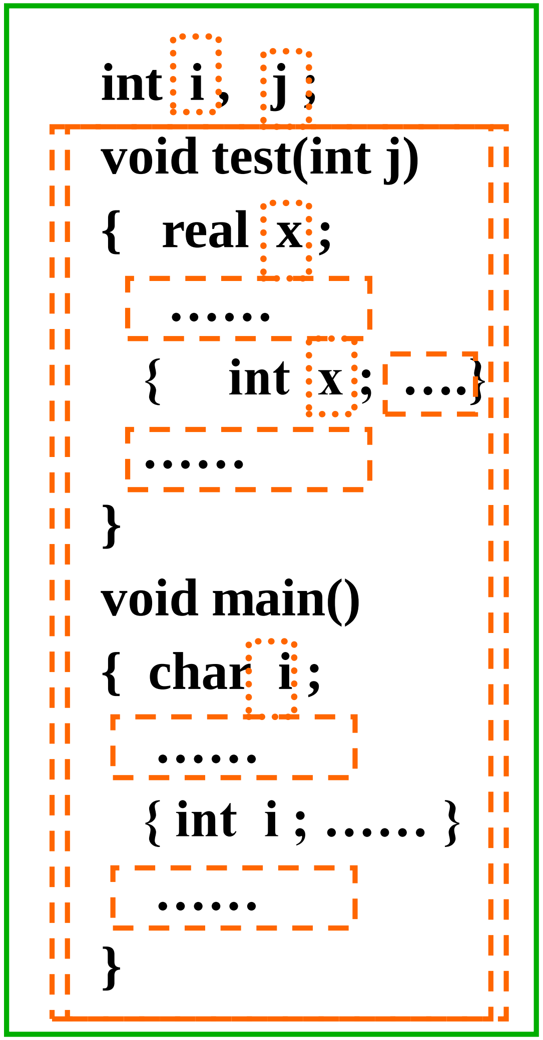
Organization of Symbol Table

- **Scope of identifiers (作用域)**
- **Localized Symbol Table (符号表的局部化)**
- **Globalized Symbol Table (符号表的全局化)**
- **Interface Functions for Symbol Table (接口函数)**

Scope of Identifier

- 作用域(scope)
  - An identifier has its scope in a program;
  - A scope for an identifier is a piece of program where it is visible or effective;
  - 一个标识符的作用域是该标识符有效的一段程序，称为程序的局部化单位，通常一个程序局部化单位是一个子程序（函数）或者分程序；
  - 一个标识符的作用域从声明该标识符的位置开始到其所在的局部化单位的结束（其中要去掉其内部声明的同名标识符的作用域）；
  - 特别地，域名的作用域是包含该域名的结构或者联合体；

Scope of Identifier



```
int i, j;
void test(int j)
{   real x;
    ......
   {    int x; ....}
   ......
}
void main()
{  char  i;
    ......
   { int  i ; ...... }
    ......
}
```

Question:
what about "test" and "main"?

Dealing with Symbol Table

```
int  i , j ;
void test ( int j )
{   real  x ;
     … j … .
   {  int  x ;  ….}
   ……
}
void main()
{  char  i ;
     …..
   { int  i ; …… }
     ……
}
```

i: (intPtr, varKind, 0, 0, dir)
j: (intPtr, varKind, 0, 1, dir)
test: (voidPtr, routKind, 0, …)
main: (voidPtr, routKind, 0, …)
i: (charPtr, varKind, 1, 0, dir)
i: (intPtr, varKind, 1, 1, dir)

Dealing with Symbol Table During Semantic Analysis

- **Identifier declaration(标识符声明)**
  - **Check whether declared already by *searching symbol table*;**
  - **If yes, error;**
  - **If no, establish internal representation, *insert into symbol table*;**
- **Identifier usage(标识符使用)**
  - **Check whether declared by *searching symbol table*;**
  - **If yes, get its attributes for further semantic analysis;**
  - **If no, error;**
- **Exit from 局部化单位, *delete identifiers* that are declared in it;**

Organization of Symbol Table

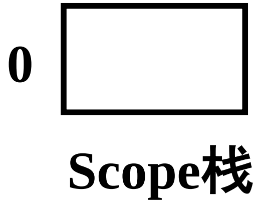
- Easy for searching
  - 顺序查找
  - 折半查找
  - 散列表(hash table)
- Reflect the scope of identifiers, guarantee that each time the effective attributes for the identifiers can be found;
  - 局部化：每个局部化单位的符号表作为一个独立的表处理，即把每个局部化单位的符号表作为建表和查表单位；
  - 全局化：把整个程序的符号表统一处理；

局部化符号表

- Scope栈保存当前所有局部化单位符号表的首地址；
- 局部化实现原理：
  - 进入局部化单位，建立一个新的空符号表，并将地址压入Scope栈；
  - 遇到定义性标识符（声明），查当前符号表判定是否有重复定义，如果没有则将其属性登记到当前符号表中；
  - 遇到使用性标识符，查符号表（从当前符号表查，如果没有，再依次查scope栈中下一个符号表，如果都没有，没有声明错；否则，找到对应的属性）；
  - 结束一个局部化单位时，删除当前符号表，弹出scope栈顶元素；
- 每个局部化单位的符号表可以是
  - 线性表；二叉树；散列表

局部化符号表

```
int  i , j ;
void test ( int j )
{   real  x ;
     … j …
   {  int  x ;  ….}
   ……
}
void main()
{  char  i ;
     ……
   { int  i ; …… }
     ……
}
```



全局化符号表

- 全局化实现原理
  - 整个程序用一个符号表，该符号表的组织可以是
    - 线性表；二叉树；散列表
  - 每个局部化单位对应一个唯一的局部化编号num;
  - 符号表的表项为(num, id, attributes);
  - 初始化：CurrentNum = 0;
  - 进入局部化单位，CurrentNum++;
  - 遇到定义性标识符(声明),检查所有对应CurrentNum的表项,判定是否有重复定义,如果没有则将其属性及其CurrentNum登记到符号表中;
  - 遇到使用性标识符,查符号表,如果都没有,没有声明错;否则,找到对应的属性;
  - 结束一个局部化单位时,删除所有CurrentNum对应的表项,CurrentNum --;

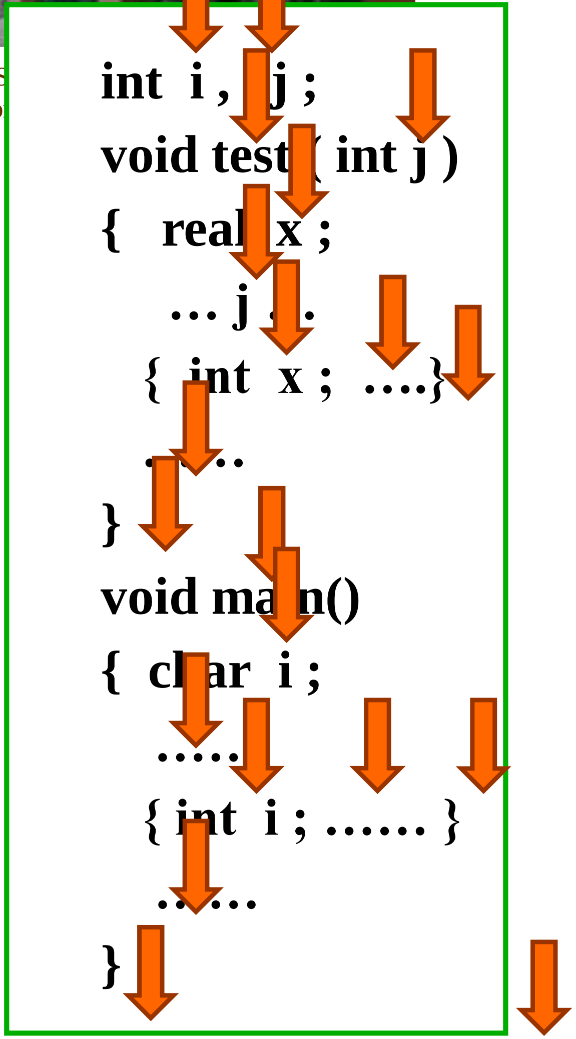
全局化符号表

- **Search for attributes for identifier *X***
  - **Depend on how to organize the symbol table?**
  - **The main idea**
    - **(1)Num = CurrentNum;**
    - **(2) search for (*X*, Num, attr) item in Symbol table,**
      - **If does exist{ return attr;}**
      - **If not exist, {Num--;}**

        if Num<0 { return false}

        else goto (2);

全部化符号表



```
int  i , j ;
void test ( int j )
{   real  x ;
     … j …
   {  int   x ;   ….}
   ……
}
void main()
{  char  i ;
    ……
   { int  i ; …… }
    ……
}
```

(0, id, attributes)

Symbol Table

Interfaces for Symbol Table

- Create symbol table
- Destroy symbol table
- Append one item to symbol table
- Find entry to an identifier
- Find attributes for an identifier

符号表的数据结构

```
struct  symbtable
{
   char  idname[10];
   AttributeIR  attrIR;
   struct symbtable  *  next;
}SymbTable;
```

```
typedef struct
{   struct typeIR          * idtype;          /*指向标识符的类型内部表示*/
    IdKind         kind;                      /*标识符的类型*/
    union
    {struct
        {       AccessKind         access;
                 int         level;
                 int         off;
        }VarAttr;                                /*变量标识符的属性*/
    struct
        {
                 int        level;
                 ParamTable  * param;    /*参数表*/
                 int        code;
                 int        size;
        }ProcAttr;                             /*过程名标识符的属性*/

    }More;                           /*标识符的不同类型有不同的属性*/
}AttributeIR;
typedef IdKind={typeKind, varKind, procKind}
```

```
struct  typeIR
{
        int  size;                    /*类型所占空间大小*/
        TypeKind           kind;
        union
           {  struct
                   { struct typeIR   * indexTy;
                     struct typeIR   * elemTy;
                   }ArrayAttr;
              fieldChain * body; /*记录类型中的域链*/
           } More;
}TypeIR;
typedef Typekind={intTy,charTy ,arrayTy, recordTy, boolTy }
```

Assignment

(1) please write down internal representation of following types?

```
typedef struct {char name[10]; int age; }person;
typedef person   List[10];
typedef union {int data; real length; char name[4];}
```

Assignment

(2) Please write down the process of creating symbol table when analyzing following program?

(localized symbol table and globalized symbol table)

```
typedef struct {char name[30]; int age;} Student;

Student ① Lee;

 int i ;

 int GetAge(Student S) ②  { return S.age;}

 void SetAge( int i)  ③  {Lee.age=i; }

 void main() { student  Lee; SetAge(10); {int i = 20;  ④  Lee.age = i;}}
```