Compiler Principle and Technology

Prof. Dongming LU
Apr. 17th, 2015

6. Semantic Analysis

Contents

6.3 The Symbol Table

- **Symbol table**: major inherited attribute and major data structure in a compiler
- **Principal operations:**
  - Insert: store the information provided by name declarations when processing these declarations
  - Lookup: retrieve the information associated to a name when that name is used in the associated code.
  - Delete: remove the information provided by a declaration when that declaration no longer applies.
- **Information stored:**
  - Include data type information, information on region of applicability (scope), and information on eventual location in memory.

6.3.1 The Structure of The Symbol Table

Typical **dictionary data structure**

- **Linear list**
  - Provide easy and direct implementations of the three basic operations;
  - Operation time is linear in the size of the list;
  - Good for a compiler implementation in which speed is not a major concern such as a prototype or experimental compiler or an interpreter for a very small program.
- **Various search tree structures**
  - (binary search trees, AVL trees, B trees)
  - Don't provide best case efficiency;
  - The delete operation is very complexity;
  - Less useful.

- **Hash tables**
  - All three operation can be performed in almost constant time
  - Most frequently in practice, best choice
- **Main idea**
  - An array of entries (called buckets), indexed by an integer range
  - A hash function turns the search key into an integer hash value in the index range
  - And the item corresponding to the search key is stored in the bucket at this index
  - The efficiency greatly depends on the design of the hash function
- **Main problem**
  - Collision resolution: two keys are mapped to the same index by the hash function

The methods to deal with the collision

- **Open addressing**
  - Inserting the collided new items in successive buckets
  - Will cause a significant degradation in performance and make delete operation difficult
- **Separate chaining**
  - Each bucket is a linear list; collisions are resolved by inserting the new item into the bucket list

Perhaps it is the best scheme for compiler construction

6.3.2 Declarations

Basic kinds of declarations

- ***Constant*** declarations (Value bindings)
  - Associate values to names, is single assignment.
  - const int SIZE = 199 ;
- ***Type*** declarations
  - Bind names to newly constructed types and may also create aliases for existing named types
  - Type names are usually used in conjunction with a type equivalence algorithm to perform type checking of a program

```
Struct Entry
{char * name;
 int count;
struct Entry *next ;};
```

- ***Variable*** declarations
  - Bind names to data types,

    Integer a, b[100]
  - An attribute of variables related to scope

    that is also implicitly or explicitly bound is the ***allocation of memory*** for the declared variable

    and ***the duration*** during execution of the allocation (lifetime or extent of the declaration)

```
int count(void)
{ static int counter = 0;
  return ++counter; }
int count(void)
{ extern int counter = 0;
 return ++counter; }
```

- Procedure/function declarations
  - Include explicit and implicit declarations

- Definition and Declaration

- The strategies:
  - Use one symbol table to hold the names from all the different kinds of declarations.
  - Use a different symbol table for each kind of declaration.

6.3.3 Scope Rules and Block Structure

Two rules:

- Declaration before use
  - Without this rule one-pass compilation is impossible
- The most closely nested rule for block structure
  - Block: any construct that can contain declarations, such as procedure/function declarations.
  - A language is block structured:

    (1) If it permits the nesting of blocks inside other blocks;

    (2) If the scope of declarations in a block are limited to that block and other block contained in that block.

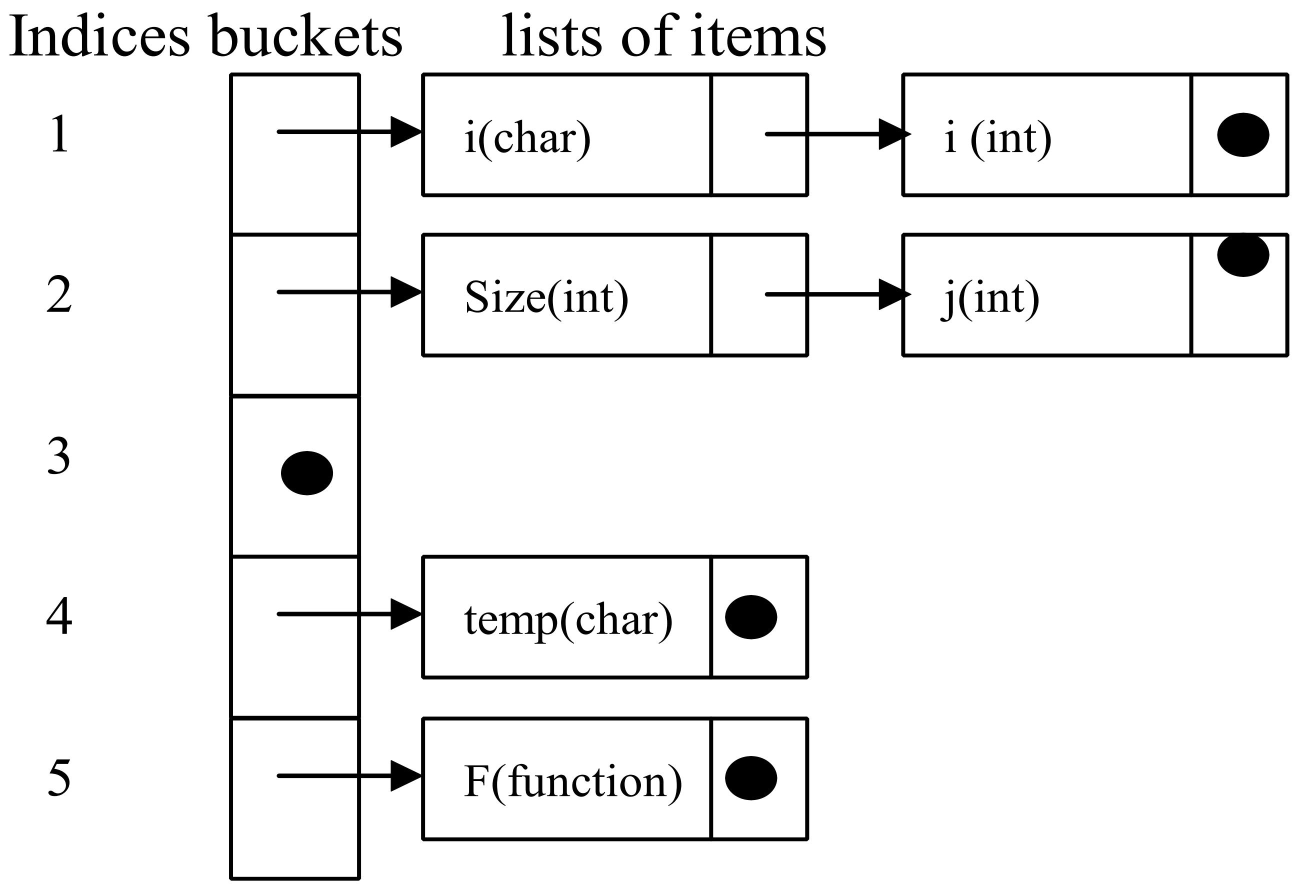
The **most closely nested rule**:

Given several different declarations for the same name, the declaration that applies to a reference is the one in the most closely nested block to the reference.

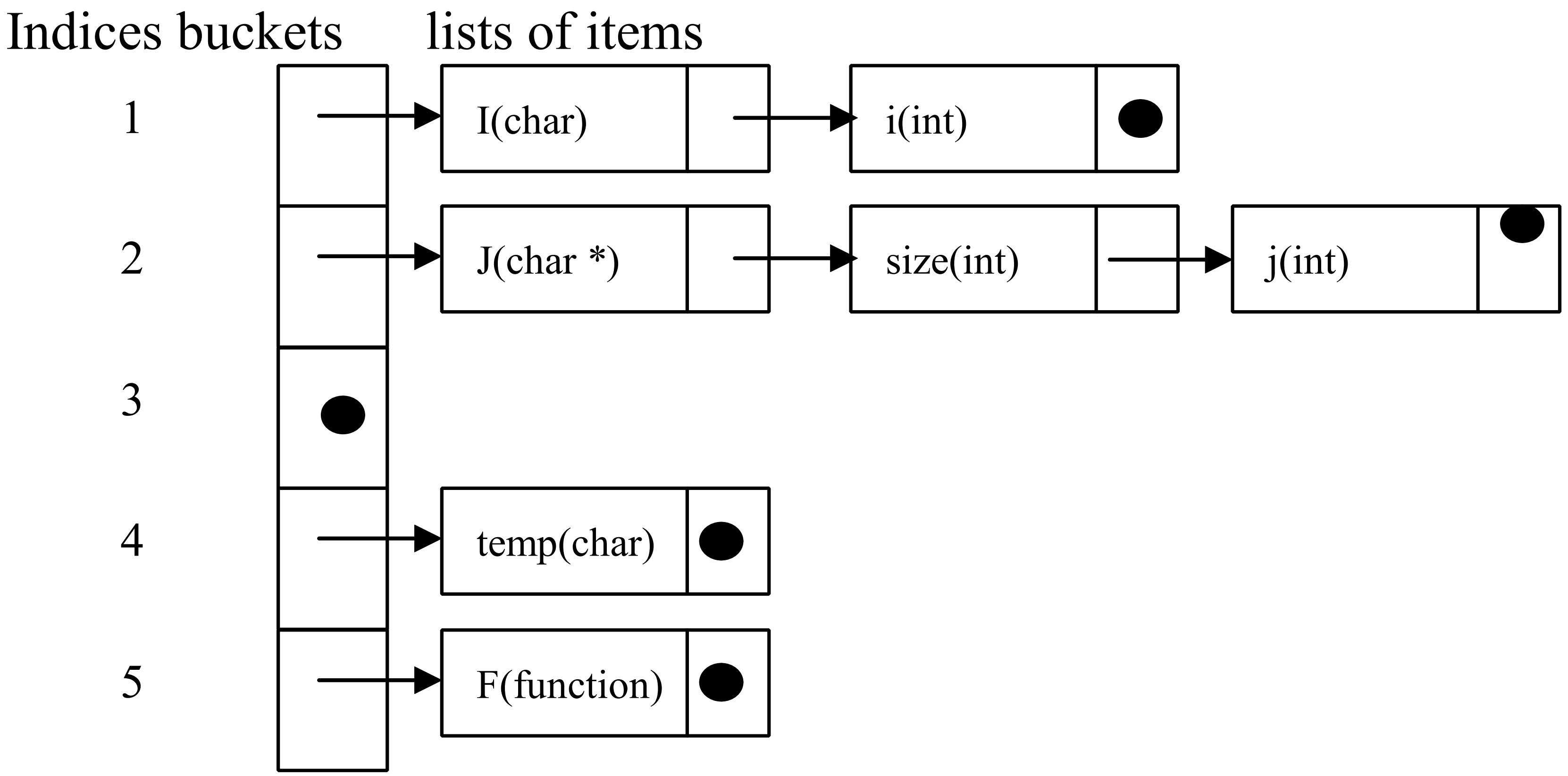
```
Int i, j ;
Int f(int size)
{ char i, temp;
  …
    {double j;
      …
    }
    {char * j;
    …
    }
}
```

Fig 6.14

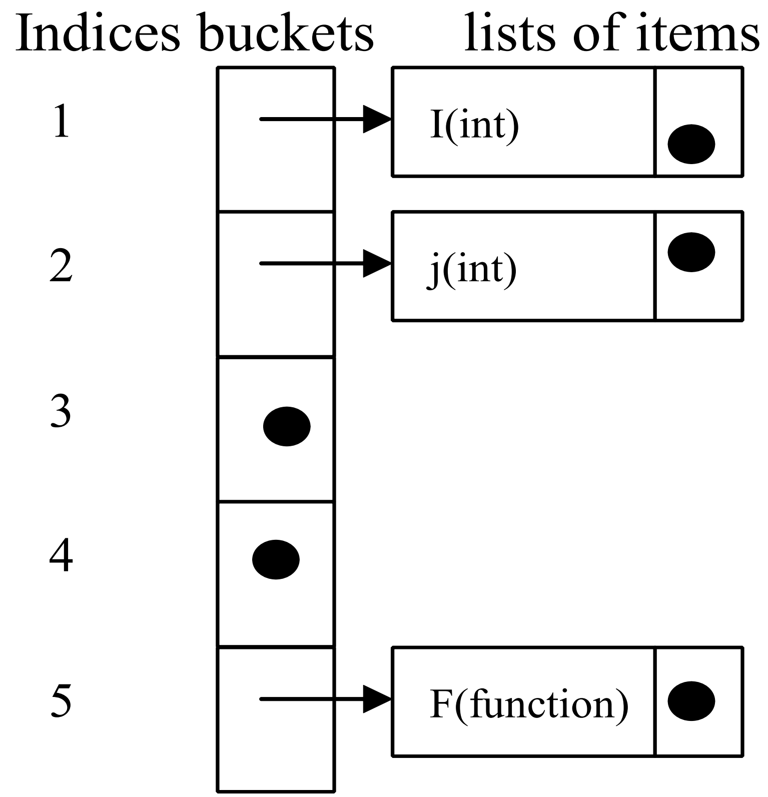
For each name, the linked lists in each bucket behave as a stack for the different declarations of that name.



(a) After processing the declarations of the body of f



(b) After processing the declarations of the second nested compound statement within the body of f

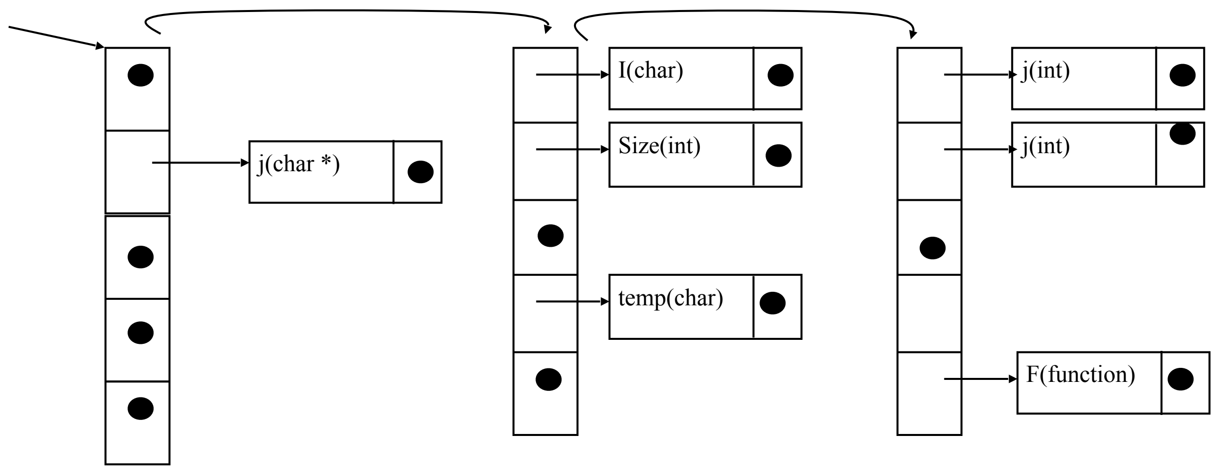


(c) After exiting the body of f (and deleting its declarations)

In fact it is necessary to keep the declarations in the table (at least not to de-allocate their memory), so the delete operation is simply to mark them as no longer active.

The other solutions to the implementation of nested scopes:

- Build a new symbol table for each scope and to link the tables from inner to outer scopes together.
- Lookup operation will automatically continue the search with an enclosing table if it fails to find a name in the current table.
- Leaving a scope simply requires resetting the access pointer to the next outermost scope.



Additional processing and attribute computation may also be necessary during the construction of the symbol table **depending on the language** and the details of the **compiler operations**.

For example in the following Pascal code:

```
Program Ex;
Var i, j: integer;
Function f(size:integer):integer;
Var i, temp:char;
        Procedure g
        Var j: real;
        Begin
                ...
        end;
```

```
            Procedure h;
            Var j:^char;
            Begin
                        …
            End
Begin
            …
end
begin (*main program*)
            …
end
```

Identify each scope by a name and to prefix each name declared within a scope by its accumulated nested scope names, Thus, all occurrence of the name j would be distinguished as

Ex.j Ex.f.g.j and Ex.f.h.j

assign a nesting level or nesting depth to each scope and record in each symbol table entry the nesting level of each name.

- **Static scope**:
  - The standard scope rule that follows the textual structure of the program
- **Dynamic scope**:
  - Requires the application of nested scopes follow the execution path, rather than the textual layout of the program.

Example:

```
#include <stdio.h>
int I = 1;
void f(void)
        {           printf(“%d\n”,i);
        }
void main(void)
{       int I=2;
        f();
        return 0;
}
```

Static: 1

Dynamic: 2

6.3.4 Interaction of Same-Level Declarations

Interaction among declarations at the same nesting level can vary with the kind of declaration and with the language being translated

- One typical requirement:
    - **No reuse of the same name** in declarations at the same level

```
Typedef int i;
Int i;
```

    - The above declaration will cause an error.

- Solution:
    - Lookup before each insert and determine by some mechanism.

```
Int i = 1;
Void f(void)
{ int i = 2 , j = i+1
… }
```

j should be initialized to the value 2 or 3?

- Solution:
  - By the most closely nested rule, the local declaration is used, j should be 3.
  - **Sequential declaration**, each declaration is added to the symbol table as it is processed.

- **Collateral declaration**:
  - Declarations not be added immediately to the existing symbol table, but accumulated in a new table (or temporary structure),
  - And then added to the existing table after all declarations have been processed

- **Recursive declaration**:
  - Declaration may refer to themselves or each other. (particularly necessary for procedure/function declarations)

```
Int gcd(int n, int m)
{if (m = = 0) return n;
                else return gcd(m , n%m)
}
```

  - The compiler should add the function name gcd to the symbol table before processing the body of the function.

```
Void f(viod)
{ … g() …}
Void g(viod)
{ … f() …}
```

- **It needs a scope modifier, adds a function prototype** declaration to extent the scope of the name g to include f. Just as follows.

```
Void g(void);  /* function prototype declaration */
Void f(void)
{ … g() …}
Void g(void)
{ … f() …}
```

- **Other solutions to mutual recursion**
  - 3 (The compiler should add the procedure/function name to the symbol table before processing the body of the procedure/function.)
  - 3 Provide forward declaration to extend procedure/function scope;
  - 3 The scope rule must be extended to the entire block of the declarations.

6.3.5 An Extended Example of An Attribute Grammar Using a Symbol Table

An example:

- S→exp
- exp→(exp) | exp + exp | id | num | let dec-list in exp
- dec-list→dec-list, decl | decl
- decl → id = exp

- Assume the ambiguities have been dealt with at parser time and the syntax tree has been constructed
  - exp → let dec-list in exp
- The declarations inside a let expression represent a kind of constant declaration. Such as
  - let x = 2+1, y=3+4 in x+y

Scope rules:

1. No redeclaration of the same name within the same let expression, such as

   let x = 2, x= 3 in x+1 (illegal)

2. Any name not declared in some surrounding let expression caused an error, such as

   let x = 2 in x+y

3. The scope of each declaration in a let expression extends over the body of the let according to the most closely nested rule for block structure. Such as

   let x = 2 in (let x = 3 in x) , the value is 3

4. Each declaration uses the previous declarations to resolve names within its own expression. Such as

   let x = 2, y = x+1 in (let x = x+y, y = x+y in y)

   the first y is 3, second x is 5, second y is 8

- Attribute equations that use a symbol table to keep track of the declarations in let expressions and that express the scope rules and interactions just described are as follows.(Only use the symbol table to determine whether an expression is erroneous or not)

- **Three attributes:**
  - **Err**: Synthesize attribute, represent whether the expression is erroneous;
  - **Symbol**: Inherited attribute, represent the symbol table;
  - **Nestlevel**: Inherited attribute, nonnegative integer, represent the current nesting level of the let blocks.

- **Functions:**
  - **Insert(s,n,l):**

    Returns a new symbol table containing all the information from symbol table s and in addition associating the name n with the nesting level l, without changing s
  - **Isin(s,n):**

    Return a Boolean value depending on whether n is in symbol table s or not.
  - **Lookup(s,n):**

    Return an integer value giving the nesting level of the most recent declaration of n, if it exists , or –1, if n is not in the symbol table s.

Initial symbol table that has no entries is emptytable.

Grammar Rule	Semantic Rules
S→exp	exp.symtab = emptytable
	Exp.nestlevel = 0
	S.err = exp.err
Exp1→exp2+exp3	exp2.symtab=exp1.symtab
	Exp3.symtab=exp1.symtab
	Exp2.nestlevel=exp1.nestlevel
	Exp3.nestlevel=exp1.nestlevel
	Exp1.err = exp2.err or exp3.err
Exp1→(exp2)	Exp2.symtab=exp1.symtab
	Exp2.nestlevel=exp1.nestlevel
	Exp1.err = exp2.err
Exp→ id	exp.err = not isin(exp.symtab, id.name)
Exp→num	exp.err = false

Exp1→let dec-list in exp2	dec-list.intab=exp1.symtab
	Dec-list.nestlevel=exp1.nestlevel+1
	Exp2.symtab=dec-list.outtab
	Exp2.nestlevel=dec-list.nestlevel
	Exp1.err = (dec-list.outtab=errtab) or exp2.err
Dec-list1→dec-list2,decl	dec-list2.intab= dec-list1.intab
	Dec-list2.nestlevel=dec-list1.nestlevel
	Decl.intab=dec-list2.outtab
	decl.nestlevel=dec-list2.nestlevel
	decl-list1.outtab=decl.outtab
dec-list → decl	decl.intab = dec-list.intab
	decl.nestlevel=dec-list.nestlevel
	dec-list.outtab=decl.outtab
decl→id = exp	exp.symtab = decl.intab
	exp.nestlevel=decl.nestlevel
	decl.outtab =
	if(decl.intab = errtab)or exp.err
	then errtab
	else if (lookup(decl.intab,id.name)= decl.nestlevel)
	then errtab
	else insert(decl.intab,id.name,decl.nestlevel)

6.4 Data Types and Type Checking

- **Type inference and type checking is one of the principal tasks of a compiler**
  - Closely related, performed together and referred to simply as type checking

- **Data type information can be:**

1. **Static**: such as in C, Pascal. Primarily,
   - Used as the principal mechanism for checking the correctness of a program prior to execution
   - Determine the size of memory needed for the allocation and the way that the memory can be accessed
   - Simplify the runtime environment
2. **Dynamic**: such as in LISP
3. **A mixture of the two**

- **Data type forms:**
  - A set of values with certain operations on those values

- These sets can be described by ***a type expression***:
  - A **type name**: integer
  - A **structured expression**: array [1..10] of real

- Type information can be explicit and implicit
  - For instance var x: array[1..10] or real (explicit)
  - Const greeting = “Hello”   (implicitly array [1..6] of char)

- Type information is maintained in the symbol table and retrieved by the type checker

6.4.1 Type Expressions and Type Constructors

- **Simple types:**
  - Such as int, double, boolean, char.
  - The values exhibit no explicit internal structure, and the typical representation is also simple and predefined.
- **Void**: has no value, represent the empty set
- **New simply type** defined such as **subrange** types (Type digit = 0...9)
- **enumerated types** (typedef enum {red, green, blue} color):
  - Treated as integers or using a smaller amount of memory .

- **Structured type**

1. *Array:*

- Two type parameter:
    - Index type (often limited to so-called ordinal types: types for which every value has an immediate predecessor and an immediate successor)
    - and component type

- Produce: a new array type, array [Color] of char
    - Array[color] of char;
    - Create an array type whose index type is color and whose component type is char.

- An array represents values that are sequences of values of the component type, indexed by values of the index type

- Arrays are commonly **allocated contiguous storage** from smaller to larger indexes, to allow for the use of automatic offset calculations during execution
- **The amount of memory** needed is n * size (n is the number of values in the index type and size is the amount of memory needed for a value of the component type)

- **Multi-dimensioned arrays**:
  - Array[0..9] of array[color] of integer
  - Array[0..9, color] of integer;
  - The sequence of values can be organized in different ways in memory depending on whether the indexing is done first on the first index, and then on the second index, or vice versa.
- **Open-indexed array**: whose index range is unspecified, especially useful in the declaration of array parameters to functions

2．*Record*

A record or structure type constructor takes a list of names and associated types and constructs a new type.

```
        Struct
        {    double r;
             int i;}
```

Different types may be combined and the names are used to access the different components.

Cartesian product type constructors: int*real

The standard implementation method is to allocate memory sequentially, with a block of memory for each component type.

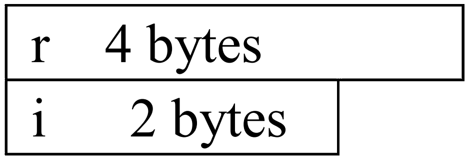
4bytes	2bytes
r	i

3． Union

Correspond to the set union operation

```
Union
{double r;
int i;}
```

Disjoint union, each value is viewed as either a real or an integer.
Allocate memory in parallel for each component.



Ex:

```
x.r=2.0;
printf(“%d”,x.i);
```

Will not cause a compilation error, but will not print the value 2.

The insecurity in union type has been addressed by many different language design.

In pascal, for instance

```
Record case isReal: boolean of
                    True: (r: real);
                    False: (I: integer);
End
```

Whenever a union component is assigned, the discriminant must be simultaneously assigned with a value that can be checked for legality.

A different approach is taken by functional languages such as the language ML:

```
        IsReal of real | IsInteger of int
```

Value constructor IsReal and IsInteger must be used whenever a value of this type is used.

4. Pointer

Values that are references to values of another type

A value of a pointer type is a memory address whose location holds a value of its base type.

^integer	in Pascal;
int*	in C.

Allocated space based on the address size of the target machine

It is most useful in describing recursive types.

5. Function

An array can be viewed as a function from its index set to its component set.

Many languages have a more general ability to describe function types.

```
Var f: procedure (integer): integer;   in Modula-2
Int  (*f)  (int);                           in C
```

The allocated space depends on the address size of the target machine. According to the language and the organization of the runtime environment,

It should allocate for a code pointer alone or a code pointer and an environment pointer.

6. Class

- Similar to a record declaration, except it includes the definition of operations (methods or member functions)
- Beyond type system such as inheritance and dynamic binding, must be maintained by separate data structures

6.4.2 Type Names, Type Declarations and Recursive Types

- Type declarations (type definition): **mechanism for a programmer to assign names to type expressions**.
  - Such as: typedef, = , associated directly with a struct or union constructor

```
typedef struct
        {double r;
        int I;
} RealIntRec;   (C)

type RealIntRec = real * int   (ML)

struct RealIntRec
        {double r;
        int I;
}          (C)
```

- Type declarations cause the **declared type names to be entered into the symbol table** just as variable declarations.
- The type names **can't be reused as variable names** (except allowed by the scope nesting rules),
- The name associated to struct or union declarations can be reused as typedef names.

```
        Struct RealIntRec
        {double r;
                int I;
        };
```

- Typedef struct RealIntRec RealIntRec; (legal)

- **Recursive data types** are extremely important include lists, trees, and many other structures.
  Permit the direct use of recursion in type declarations such as ML.

- Datatype intBST = Nil | Node of int*intBST*intBST

- The union of the value Nil with the Cartesian product of the integers with two copies of intBST itself
  - (one for the left subtree and one for the right subtree)

  The equivalent C declaration (in slightly altered form) is

```
        Struct intBST
        {          int isNull;
                   int val;
                   struct intBST left, right;
        };
```

- The declaration will generate **an error message in C.**

- Allow recursion only indirectly through pointers. Such as C.

```
Struc intBST
{int val;
  struct intBST *left, *right;
}
typedef struct  intBST * intBST;
```

- or

```
typedef struct  intBST * intBST;
Struc intBST
{int val;
  intBST  left, right;
}
```

6.4.3 Type Equivalence

Type equivalence: two type expressions represent the same type

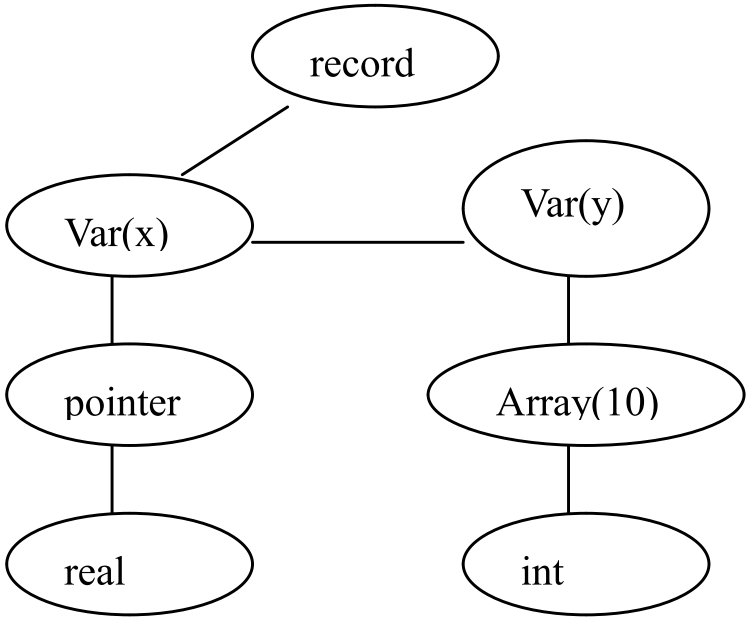
    Function typeEqual(t1,t2:TypeExp): Boolean

Type representation: use a syntax tree representation to express type expressions.

For example: the type expression

```
Record
    X: pointer to real;
    Y: array[10] of int;
End
```

This type expression can be represented by the syntax tree



A simple grammar for type expressions:

var-decls→ var-decls; var-decl | var-decl

var-decl→ **id:** type-exp

type-exp→ simple-type|structured-type

simple-type →**int|bool|real|char|void**

structured-type→ **array [num]** of type-exp|

record var-decls **end**|

union var-decls **end**|

pointer to type-exp|

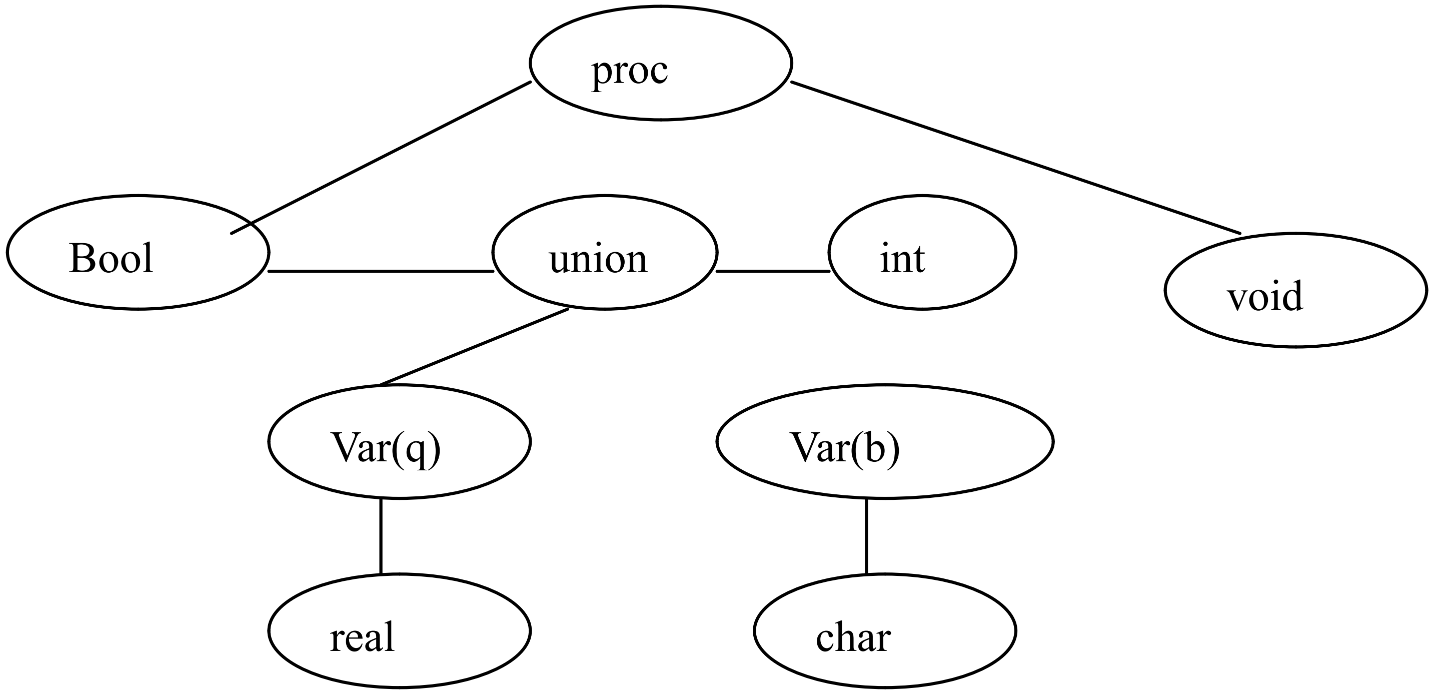
Proc(type-exps) type-exp

type-exps→ type-exps, type-exp| type-exp

Another type expression:

Proc (bool, union a: real; b: char end, int): void

can be represented by the following syntax tree:



The first type of equivalence is structural equivalence

(In the absence of names for types)

If and only if they have syntax trees that are identical in structure

The pseudocode for a typeEqual function

```
Function typeEqual(t1,t2:TypeExp):Boolean:
Var temp: boolean;
    P1, p2: TypeExp;
Begin
    If t1 and t2 are of simple type then return t1=t2;
    Else if t1.kind = array and t2.kind = array then
        Return t1.size = t2.size and typeEqual(t1.child1,t2.child1)
    Else if t1.kind = record and t2.kond = record
        Or t1.kind = union and t2.kond = union then
    Begin
        P1 :=t1.child1;
        P2 :=t2.child1;
        Temp :=true;
        While temp and p1 != nil and P2!=nil do
            If p1.name !=p2.name then
                Temp := false
            Else if not typeEqual(p1.child1, p2.child1)
                Then temp :=false
            Else begin
                P1:=p1.sibling;
                P2:=p2.sibling;
                End;
        Return temp and p1 = nil and p2 = nil;
    End
    Else if t1.kind = pointer and t2.kind = pointer then
        Return typeEqual(t1.child1,t2.child1)
    Else if t1.kind = proc and t2.kind = proc then
    Begin
        P1 :=t1.child1;
        P2 := t2.child1;
        Temp :=true;
        While temp and p1 !=nil and p2 !=nil do
            If not typeEqual(p1.child1,p2.child1)
            Then temp :=false
            Else begin
                P1:=p1.sibling;
                P2:=p2.sibling;
                End;
        Return temp and p1 = nil and p2 = nil
            and typeEqual(t1.child2,t2.child2);
    End
    Else return false
End;
```

A grammar for type expressions with type declarations:

Var-decls→ var-decls; var-decl | var-decl

Var-decl→ **id:** simpe-type-exp

Type-decls→ type-decls; type-decl | type-decl

Type-decl→ **id:** type-exp

Type-exp→ simple-type-exp|structured-type

Simple-Type-exp→ simple-type|**id**

Simple-type →**int|bool|real|char|void**

Structured-type→ **array [num]** of simple-type-exp|
Record var-decls **end**|
Union var-decls **end**|
Pointer to simple-type-exp|
Proc(type-exps) simple-type-exp

Type-exps→ type-exps, simple-type-exp| simple-type-exp

- **Name equivalence:**
- Two type expressions are equivalent if and only if they are either the same simple type or are the same type name.

  For example:

  T1=int;

  T2=int;
- The types of t1 and t2 are not equivalence and not equivalence to int.

```
Function typeEqual(t1,t2:typeexp):Boolean:
Var temp: boolean;
        P1, p2: typeExp;
Begin
If t1 and t2 are of simple type then
        Return t1 = t2
Else if t1 and t2 are type names then
Return t1 = t2
Else return false;
End;
```

- It is possible to retain structural equivalence in the presence of type name. By adding the following case to the above mentioned code

```
      Else if t1 and t2 are type name then
        Return typeEqual(getTypeExp(t1),getTypeExp(t2)).
```

- Care must be taken in implementing structural equivalence when recursive type references are possible, since the algorithm just described can result in an infinite loop.

  For example:

```
T1 = record
        X :int;
        T: pointer to t2;
End;
T2 =record
        X:int;
        T:pointer to t1;
End;
```

- **Modify typeEqual(t1,t2) to include the assumption that they are already potentially equal.**

- **Declaration equivalence:**

  (Pascal and struct and union in C)
- Every type name is equivalent to some base type name, which is either a predefined type or is given by a type expression resulting from the application of a type constructor.

  For example:

  ```
  T1=int;
  T2=int;
  ```
- Both t1 and t2 are equivalent to int.

  For example:

  ```
  T1=array [10] of int;
  T2=array [10] of int;
  T3=t1;
  ```
- T1 and t3 are equivalence, but neither is equivalence to t2

- **Pascal uses declaration equivalence while C uses declaration equivalence for structures and unions and structural equivalence for pointers and arrays**

- Occasionally, a language will offer a choice of structural, declarations, or name equivalence, where different type declarations are used for different forms of equivalence.

6.4.4 Type Inference and Type Checking

Type checker is based on a representation of types and a typeEqual operations.
Consider a simple grammar as follows:

```
program → var-decls; stmts
var-decls →var-decls; var-decl | var-decl
var-decl→id: type-exp
type-exp→int |bool|array [num] of type-exp
stmts→stmts; stmt|stmt
stmt → if exp then stmt | id :=exp
```

1. Declarations: cause the type of an identifier to be entered into the symbol table
2. Statements: substructures will need to be checked for type correctness
3. Expression: variable name have their types determined by a lookup in the symbol table, the other expression are formed by operators.

Note: How about the behavior of type checker in presence of errors.

Grammar Rule	Semantic Rules
Var-decl →id : type-exp	insert(id.name, type-exp.type)
Type-exp→int	type-exp.type :=integer
Type-exp→bool	type-exp.type :=boolean
Type-exp1→array[num] of type-exp2	type-exp1.type:= Maketypenode(array, num.size,type-exp2.type)
Stmt→if exp then stmt	if not typequal(exp.type, boolean) Then type-error(stmt)
Stmt→id:=exp	if not typequal(lookup(id.name), exp.type) Then type-error(stmt)

Exp1→exp2+exp3	if not typequal(exp2.type,integer)
	and typequal(exp3.type,integer)
	Then type-error(exp1)
	Exp1.type :=integer
Exp1→exp2 or exp3	if not typequal(exp2.type,boolean)
	and typequal(exp3.type,boolean)
	Then type-error(exp1)
	Exp1.type :=boolean
Exp→num	exp.type :=integer
Exp→true	exp.type := boolean
Exp→false	exp.type :=boolean
Exp→id	exp.type :=lookup(id.name)

6.4.5 Additional Topics in Type Checking

1. Overloading

  - Procedure max(x,y: integer): integer;
  - Procedure max(x,y: real): integer.

- Solutions:
  - Augment the lookup procedure of the symbol table with type parameters, allowing the symbol table to find the right match;
  - For the symbol table to maintain sets of possible types for names and to return the set to the type checker for disambiguation

2. Type conversion and coercion

- For example:

  2.1 + 3

- Solutions:
  - operations must be applied to convert the values at runtime to the appropriate representations before applying the operator;
  - supply the conversion operation automatically, based on the types of the sub-expressions (coercion)

3. Polymorphic typing

```
Procedure swap( var x, y:anytype);
var x,y : integer;
    a ,b: char;
…
swap(x,y);
swap(a,b);
swap(a,x);
```

- The type checking will involve sophisticated pattern matching techniques.

End of Part Three

THANKS