Chapter 6
Semantic Analysis

2022 Spring&Summer

Outline

- Attributes and attribute grammars
- Algorithms for attribute computation
- The symbol table
- Data types and type checking
- A semantic analyzer for the TINY language

6.3 The symbol table

- Semantic checks refer to properties of identifiers in the program -- their scope or type
- Need an environment to store the information about identifiers = symbol table
- Each entry in the symbol table contains
  - the name of an identifier
  - additional information: its kind, its type, if it is constant, …

NAME	KIND	TYPE	ATTRIBUTES
foo	fun	Int x int $\rightarrow$ bool	extern
m	arg	int	
n	arg	int	const
tmp	var	bool	const

6.3.1 The structure of the symbol table

1. Linear List:

   Provide easy and direct implementations of the three basic operations;

   insert operation is performed in constant time

   lookup and *delete* operation are linear time in the size of the list;

   Good for a compiler implementation in which speed is not a major concern

   - a prototype
   - experimental compiler
   - an interpreter for a very small programs.

6.3.1 The structure of the symbol table

2. 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

3. Hash Tables

all three operation can be performed in almost constant time;

most frequently in practice, best choice

6.3.1 The structure of the symbol table

Hash Table

- A *hash function* turns the search key(the identifier name, consisting of a string of characters) into an integer hash value in the index range
- Indexed by an integer range from 0 to the table size minus one
- The item corresponding to the search key is stored in the bucket at this index.

Main problem:

Collision: two keys are mapped to the same index by the hash function.

6.3.1 The structure of the symbol table

Collision resolution

1. Open addressing
   - inserting the collided new items in successive buckets.
   - cause a significant degradation in performance and make delete operation difficult.
2. Separate chaining
   - each bucket is a linear list , collisions are resolved by inserting the new item into the bucket list.
   - it is the best scheme for compiler construction.

6.3.1 The structure of the symbol table

The size of the hash table

- One question:

  How large to make the initial bucket array. This size will be fixed at compiler construction time.

- Typical sizes range from a few hundred to over a thousand.
- The actual size of the bucket array should be chosen to be a *prime number.*

6.3.1 The structure of the symbol table

The process of the hash function

Three-step process:

1. Converts a character string (the identifier name) into a nonnegative integer.
2. These integers are combined in some way to form a single integer.
3. The result integer is scaled to the range *0…size-1*.

6.3.1 The structure of the symbol table

The algorithm of the hash function

1. One simple algorithm : ignore many of the characters and to add together only the value of the first few, or the first, middle, and last, characters.

   This is inadequate for a compiler.

2. Another simple method : add up the values of all the characters.

   all permutations of the same characters will cause collisions.

6.3.1 The structure of the symbol table

The algorithm of the hash function

3. One good solution : repeatedly use a constant number α as a multiplying factor when adding in the value of the next character.

   $h_{i+1} = \alpha h_i + c_i \quad h_0 = 0$

   final hash value $h = h_n \bmod size$

   n is the number of characters in the name being hashed.

   $h = (\alpha^{n-1} c_1 + \alpha^{n-2} c_2 + ... + \alpha\, c_{n-1} + c_n) \bmod size$

The choice of α has a significant effect on the outcome.

A reasonable choice for α is a power of 2, such as 16 or 128, so that the multiplication can be performed as a shift .

6.3.2 Declarations

Four basic kinds of declarations :

- constant declarations,
- type declarations
- variable declarations
- procedure/function declarations

6.3.2 Declarations

Constant Declarations

- Associate values to names.   value binding
- The values that can be bound determine how the compiler treats them.
  - Pascal and Modula-2:the values in a constant declaration be static , computable by the compiler.
  - Such as C and Ada, permit constants to be dynamic. only computable during execution.

6.3.2 Declarations

Type Declarations

- Bind names to newly constructed types and may also create aliases for existing named types.

  Type names are used in conjunction with a type equivalence algorithm (perform type checking of a program)

  type table = array [1..SIZE] of Entry (pascal)

  struct Entry   (C)

  *{char * name;*

  int count;

  *struct Entry *next ;*

  };

  *typedef struct Entry *Entryptr;*

-

6.3.2 Declarations

Variable Declarations

- Bind names to data types

  integer a, b[100]; (C declaration)

  integer a, b(100) (FORTRAN declaration)

- Variable declarations may also bind other attributes implicitly : *scope of a declaration.*
- 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)

6.3.2 Declarations

Procedure/Function Declarations

- Include *explicit* and *implicit* declarations

  Declarations are attached to executable instructions without explicit mention.

6.3.2 Declarations

The strategies

1. Use one symbol table to hold the names from all the different kinds of declarations.
2. Use a different symbol table for each kind of declaration.
3. Associate separate symbol tables with different regions of a program and link them together according to the semantic rules of the language.

6.3.3 Scope rules and block structure

- Two Rules:

  Declaration before use

  The most closely nested rule for block structure

6.3.3 Scope rules and block structure

Declaration Before Use

- Used in C and Pascal, that requires a name be declared *prior to* any references to the name.
- Declaration before use permits
  - the symbol table to be built as parsing proceeds.
  - lookups to be performed as soon as a name reference is encountered in the code.
- If the lookup fails
  - a violation of declaration before use has occurred,
  - the compiler will issue an appropriate error massage.

6.3.3 Scope rules and block structure

The most closely nested rule for block structure

- Block: any construct that can contain declarations. such as procedure/function declarations .
- In Pascal,
  - the blocks are the main program
  - procedure/function declarations.
- In C
  - the blocks are the compilation units, procedure/function declarations.
  - the compound statements(surrounded by curly brackets).

6.3.3 Scope rules and block structure

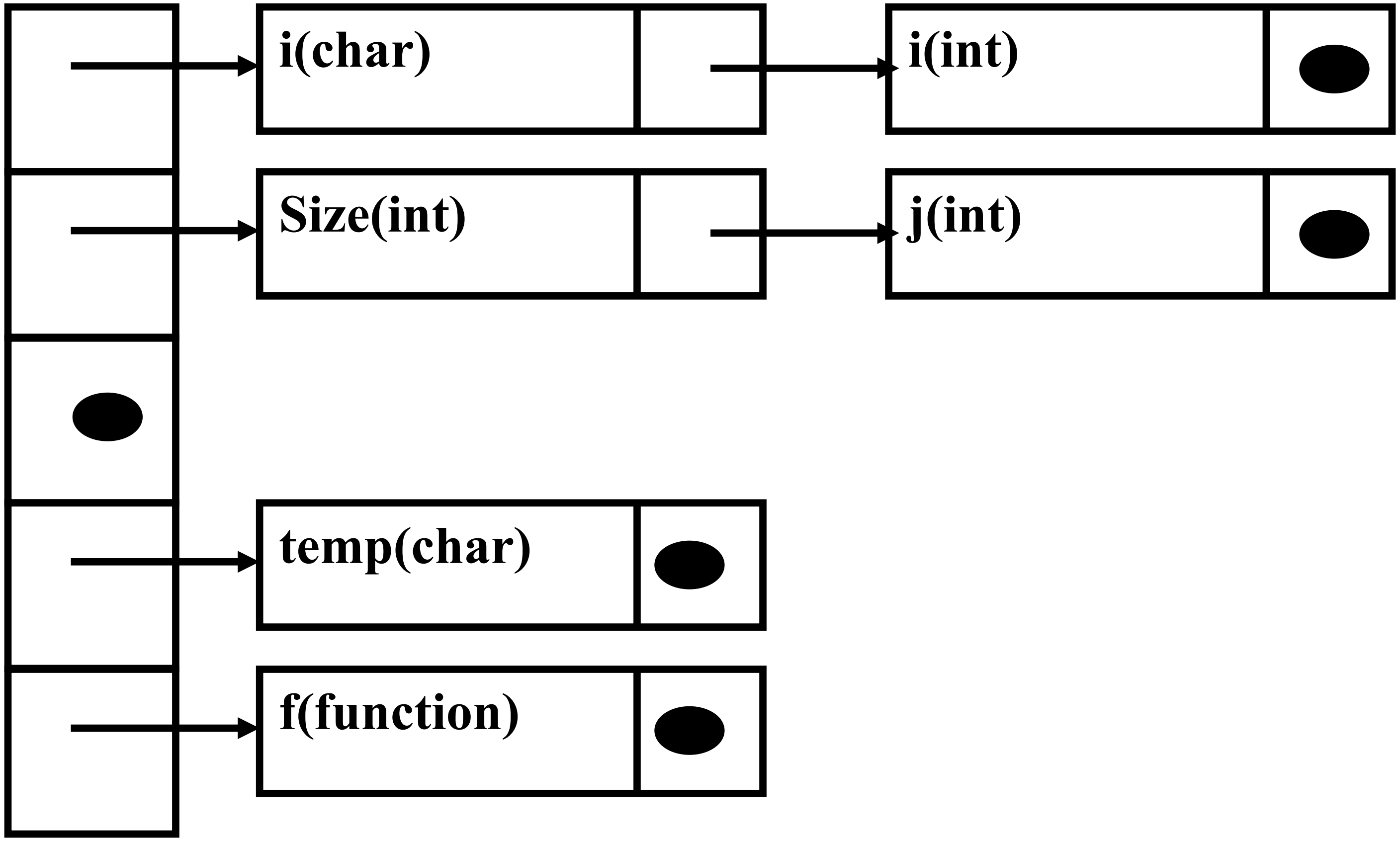
The most closely nested rule for block structure

- A language is block structured
  - if it permits the nesting of blocks inside other blocks
  - if the scope of declarations in a block are limited to that block and other block contained in that block

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

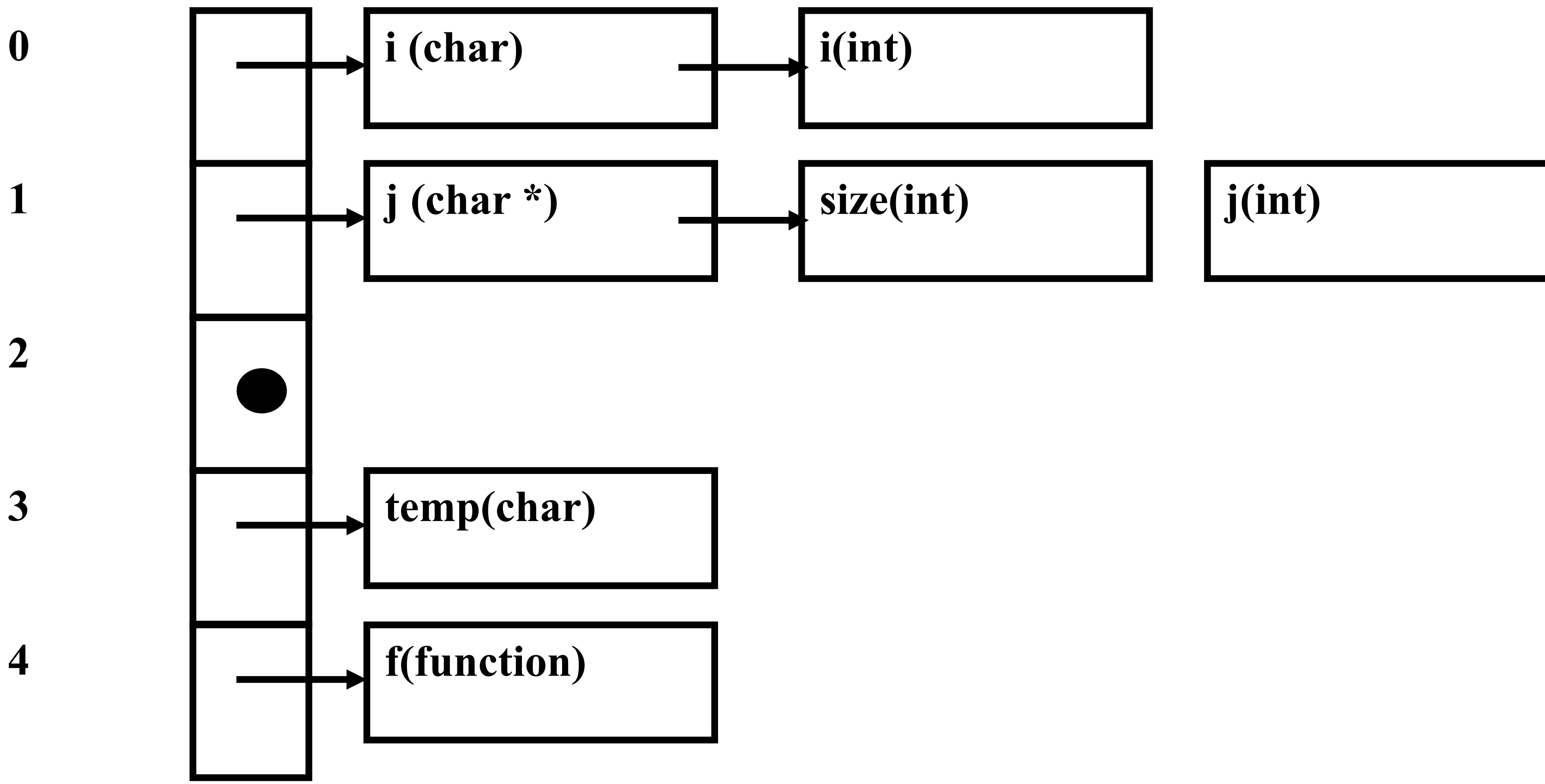
6.3.3 Scope rules and block structure

After processing the declarations of the body of f



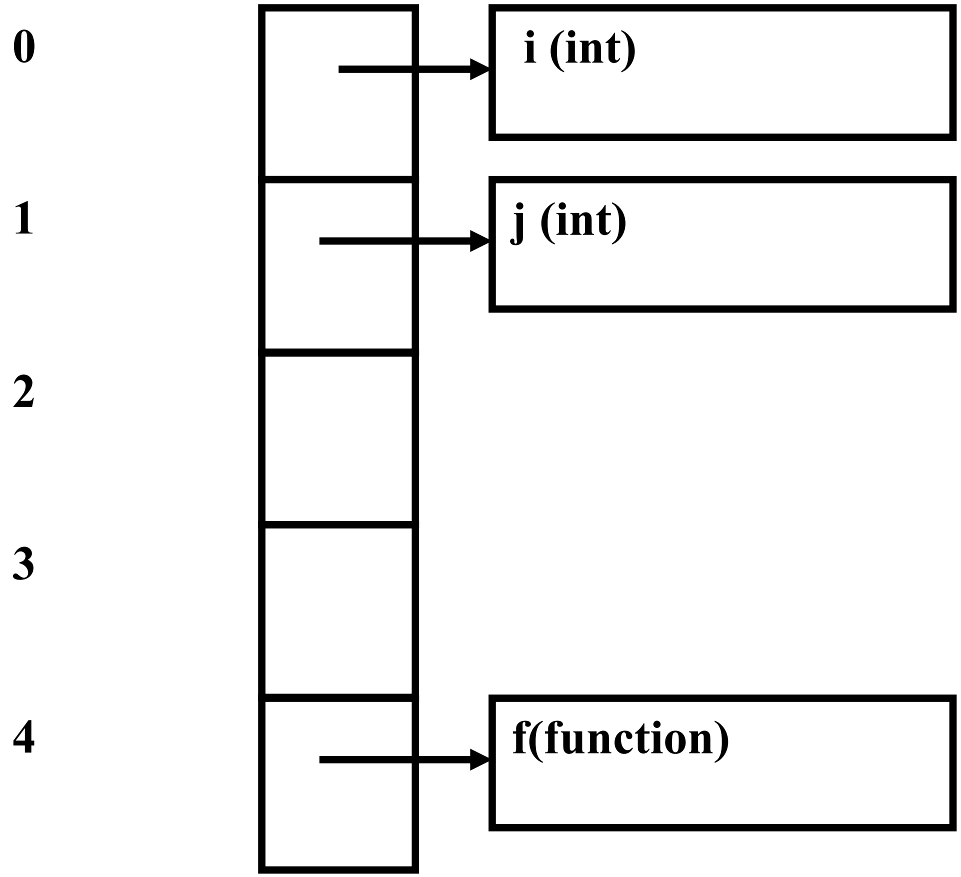
6.3.3 Scope rules and block structure

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



6.3.3 Scope rules and block structure

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



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.

6.3.4 Interaction of same-level declarations

- Lookup before each insert.
- Determine by some mechanism whether any preexisting declarations with the same name are at the same level or not.

6.3.4 Interaction of same-level declarations

- How much information the declarations in a sequence at the same level have available about each other

  int i = 1;

  Void f(void)

  { int i = 2 , j = i+1;

  ... }

  Question： j=?

- By the most closely nested rule, the local declaration is used, *j* should be *3*.

6.3.4 Interaction of same-level declarations

- 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
  - Accumulated in a new table(or temporary structure)
  - Added to the existing table after all declarations have been processed.
- Recursive declaration:

  declaration may refer to themselves or each other.

6.3.4 Interaction of same-level declarations

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

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

6.3.5 An extended example of an attribute grammar using a symbol table

S → *exp*

exp → ***(****exp****)*** | *exp* + *exp* | ***id*** | ***num*** | ***let*** *dec-list* ***in*** *exp*

dec-list → *dec-list**,** decl* | *decl*

decl → ***id*** = *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.*
- The *let* expressions represent the blocks of this language.
- The grammar allows arbitrary nesting of *let* expressions inside each other.

6.3.5 An extended example of an attribute grammar using a symbol table

Attribute

- Use a symbol table
  - keep track of the declarations in let expressions
  - 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.

6.3.5 An extended example of an attribute grammar using a symbol table

Semantic Rules

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

6.3.5 An extended example of an attribute grammar using a symbol table

Semantic Rules

Grammar Rule	Semantic Rules
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

6.3.5 An extended example of an attribute grammar using a symbol table

Semantic Rules

Grammar Rule	Semantic Rules
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
- Type checking

  the principal tasks of a compiler

6.4 Data types and type checking

- Type checking = set of rules that ensure the type consistency of different constructs in the program
- Examples:
  - The type of a variable must match the type from its declaration
  - The operands of arithmetic expressions (+, *, -, /) must have integer types; the result has integer type
  - The operands of comparison expressions (==, !=) must have integer or string types; the result has boolean type

6.4 Data types and type checking

- More examples:
  - For each assignment statement, the type of the updated variable must match the type of the expression being assigned
  - For each call statement *foo(v1, ..., vn)*, the type of each actual argument *vi* must match the type of the corresponding formal argument *fi* from the declaration of function *foo*
  - The type of the return value must match the return type from the declaration of the function

6.4.1 Type expressions and type constructors

- Data type forms:

  a set of values with certain operations on those values.

- Type information can be explicit and implicit.

  For instance *var x: array[1..10] of real* (explicit)

  const greeting = "Hello" (implicitly array [1..6] of char)

- A data type is a set of values, or more precisely, a set of values with certain operations on those values.
- These sets are usually described by a type expression.

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 and *enumerated* types.

6.4.1 Type expressions and type constructors

Structured type

- New data types can be created using type constructors.
- Such constructors can be viewed as functions :

  take existing types as parameters .

  return new types with a structure that depends on the constructor.
- Array :Type parameter:

  index type

  component type.

6.4.1 Type expressions and type constructors

Array

- Arrays are commonly allocated contiguous storage from smaller to larger indexes.
- Allow for the use of automatic offset calculations during execution.
- The amount of memory needed is $n * size$.

6.4.1 Type expressions and type constructors

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 .
- The names are used to access the different components.

6.4.1 Type expressions and type constructors

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, but never both.
- Allocate memory in parallel for each component.

6.4.1 Type expressions and type constructors

Pointer

- Values that are references to values of another type. Most useful in describibling recursive types.
- A value of a pointer type is a memory address whose location holds a value of its base type.

  ^*integer*

  **integer*
- Allocated space based on the address size of the target machine.

6.4.1 Type expressions and type constructors

Function

- An array can be viewed as a function from its index set to its component set.
- Many language have a more general ability to describe function types.
- The allocated space depend 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

  Environment pointer.

6.4.1 Type expressions and type constructors

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 type

- 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)

6.4.2 Type names, type declarations and recursive type

- Type declarations cause the declared type names to be entered into the symbol table just as variable declarations.
- Usually the type names can't be reused as variable names.
- The C language has a small exception to this rule in that names associated to *struct* or *union* declarations can be reused as *typedef* names.

```
struct RealIntRec
        { double r;
          int i;
        };
typedef struct RealIntRec RealIntRec;
```

6.4.2 Type names, type declarations and recursive type

- Since type names can appear in type expressions, questions arise about the recursive use of type names.
- Such recursive data types are extremely important in modern programming languages include lists, trees, and many other structures.

```
direct  use of recursion (ML language):
datatype intBST = Nil | Node of int*intBST*intBST
```

```
indirect  use of recursion  (C language):
struct intBST
    {  int val;
       struct intBST *left, *right;
    };
typedef struct intBST  *intBST;
```

6.4.3 Type equivalence

- Type equivalence: two type expression represent the same type.
- There are many possible ways for type equivalence to be defined by a language.
- Represent type equivalence as it would be in a compiler semantic analyzer.

  function typeEqual (t1,t2:TypeExp): boolean;

6.4.3 Type equivalence

A simple grammar for type expressions:

var-decls $\rightarrow$ *var-decls* ; *var-decl* | *var-decl*

var-decl $\rightarrow$ *id* : *type-exp*

type-exp $\rightarrow$ *simple-type* | *structured-type*

simple-type $\rightarrow$ *int* | *bool* | *real* | *char* | *void*

structure-type $\rightarrow$ *array [num] of type-exp*|
record var-decls end |
union var-decls end |
pointer to type-exp |
proc (type-exps) type-exp

type-exps $\rightarrow$ *type-exps*, *type-exp* | *type-exp*

6.4.3 Type equivalence

The type expression can be represented by a syntax tree .

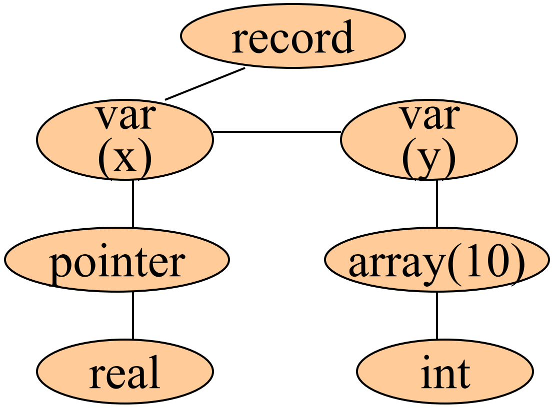
The type expression:

record

x: pointer to real;

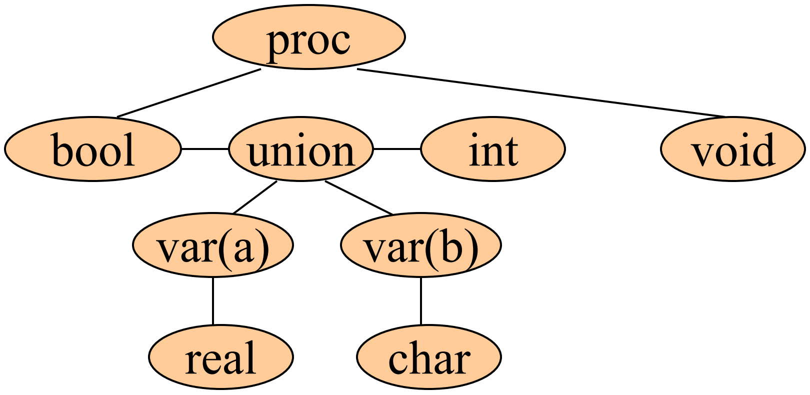
y: array [10] of int

end



The type expression:

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



6.4.3 Type equivalence

Classification of type equivalence

1、Structural equivalence

2、Name equivalence

3、Declaration equivalence

6.4.3 Type equivalence

Structural equivalence

- Two types are the same if and only if they have the same structure.
- Two types are the same if and only if they have syntax trees that are identical in structure.

```
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.kind = record
                or t1.kind = union and t2.kind = union then
               begin
               p1 :=t1.child1;
               p2 :=t2.child1;
               temp :=true;
```

6.4.3 Type equivalence

Structural equivalence

```
        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)
```

6.4.3 Type equivalence

Structural equivalence

```
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;
```

6.4.3 Type equivalence

- Two arrays are equivalent: the same size and component type.
- Two records are equivalent: the same components with the same names and in the same order.

Different choices:

- The size of the array can be ignored
- The components of a structure or union can be in a different order.

6.4.3 Type equivalence

Name equivalence

- Restricted variable declarations and type subexpressions to simple types and type names.

t1 = array [10] of int;
t2 = array [10] of int;
t3 = record
x: t1;
y: t2
end

•Two type expressions are *equivalent* if and only if they are either the same simple type or are the same type name.

t 1 = int;

t2 = int;

t1 and t2 are not equivalent.

6.4.3 Type equivalence

Name equivalence

```
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;
```

6.4.3 Type equivalence

Name equivalence

One complication in name equivalence:

- Type expressions can be allowed in variable declarations or subexpressions of type expressions.
- A type expression may have no explicit name given to it, a compiler will have to generate an internal name for the type expression that is different from any other names.

x:array [10] of int;
y:array [10] of int;

The variable *x* and *y* are assigned different (and unique) type names corresponding to the type expression.
It is possible to retain structural equivalence in the present of type names.

6.4.3 Type equivalence

Declaration equivalence

- weaker version of name equivalence

  t2 = t1; are interpreted as establishing type aliases, rather than new types.

- 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.

  t1 = array [10] of int;

  t2 = array [10] of int;

  t3 = t1;

  type names *t1* and *t3* are equivalent under declaration equivalence, but neither is equivalent to *t2*.

6.4.3 Type equivalence

- Pascal uniformly uses declaration equivalence
- C uses declaration equivalence for structures and unions, but structural equivalence for pointers and arrays.
- A language will offer a choice of structural, declaration or name equivalence.

6.4.4 Type inference and type checking

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*

Table 6.10 (p.330)
Attributes grammar for type checking of this grammar

6.4.4 Type inference and type checking

1. Declarations: cause the type of an identifier to be entered into the symbol table.

   Insert (id.name, type-exp.type);

2. Statements: substructures will need to be checked for type correctness.

   if not typeEqual(exp.type,boolean)

   then type-error(stmt)

3. Expression:

6.4.4 Type inference and type checking

- The behavior of such a type checker in the presence of errors:
  - The primary issues are when to generate an error message.
  - How to continue to check types in the presence of errors.

6.4.5 Additional topics in type checking

- Overloading: the same operator name is used for two different operations.

  procedure max(x,y: integer):integer;

  procedure max(x,y: real):real;

  In C and Pascal : illegal (redeclaration)

  In Ada and C++ : legal

- Type conversion and coercion

  allow arithmetic expressions of mixed type.

  There are two approaches a language can take to such conversions.

  Require the programmer supply a conversion function (Modula-2)

  The type checker supply the conversion automatically. (C) (coercion)

6.4.5 Additional topics in type checking

- Polymorphic typing

  Allow language constructs to have more than one type.

  procedure swap (var x,y: anytype);

  var x, y: integer;
  a, b: char;

  .....

  swap(x,y);
  swap(a,b);
  swap(x,a);

  A type checker must in every situation where swap is used determine an actual type that matches this type pattern or declare a type error. (involve sophisticated pattern matching techniques)

Homework of Chapter 6

6.7 Consider the following grammar for simple Pascal-style declarations:

decl → var-list **:** type

var-list → var-list **, id** | **id**

type → **integer** | **real**

Write an attribute grammar for the type of a variable.

6.8 Consider the grammar of Exercise 6.7.Rewrite the grammar so that the type of a variable can be defined as a purely synthesized attribute, and give a new attribute grammar for the type

that has this property.

Homework of Chapter 6

6.13 Consider the following attribute grammar:

Grammar Rule	Semantic Rules
S → A B C	B.u = S.u
	A,u = B.v + C.v
	S.v = A.v
A → a	A.v = 2 * A.u
B → b	B.v = B.u
C → c	C.v = 1

a. Draw the parse tree for the string abc (the only string in the language),and draw the dependency graph for the associated attributes. Describe a correct order for the evaluation of the attributes.

b. Suppose that S.u is assigned the value 3 before attribute evaluation begins. What is the value of S.v when evaluation has fineshed?

Homework of Chapter 6

- c. Suppose the attribute equations are modified as follows:

Grammar Rule	Semantic Rules
S → A B C	B.u = S.u
	C.u = A.v
	A,u = B.v + C.v
	S.v = A.v
A → a	A.v = 2 * A.u
B → b	B.v = B.u
C → c	C.v = C.u -2

What value does S.v have after attribute evaluation, if S.u = 3 before evaluation begins?