Chapter 8
Declaration Processing and Symbol Tables

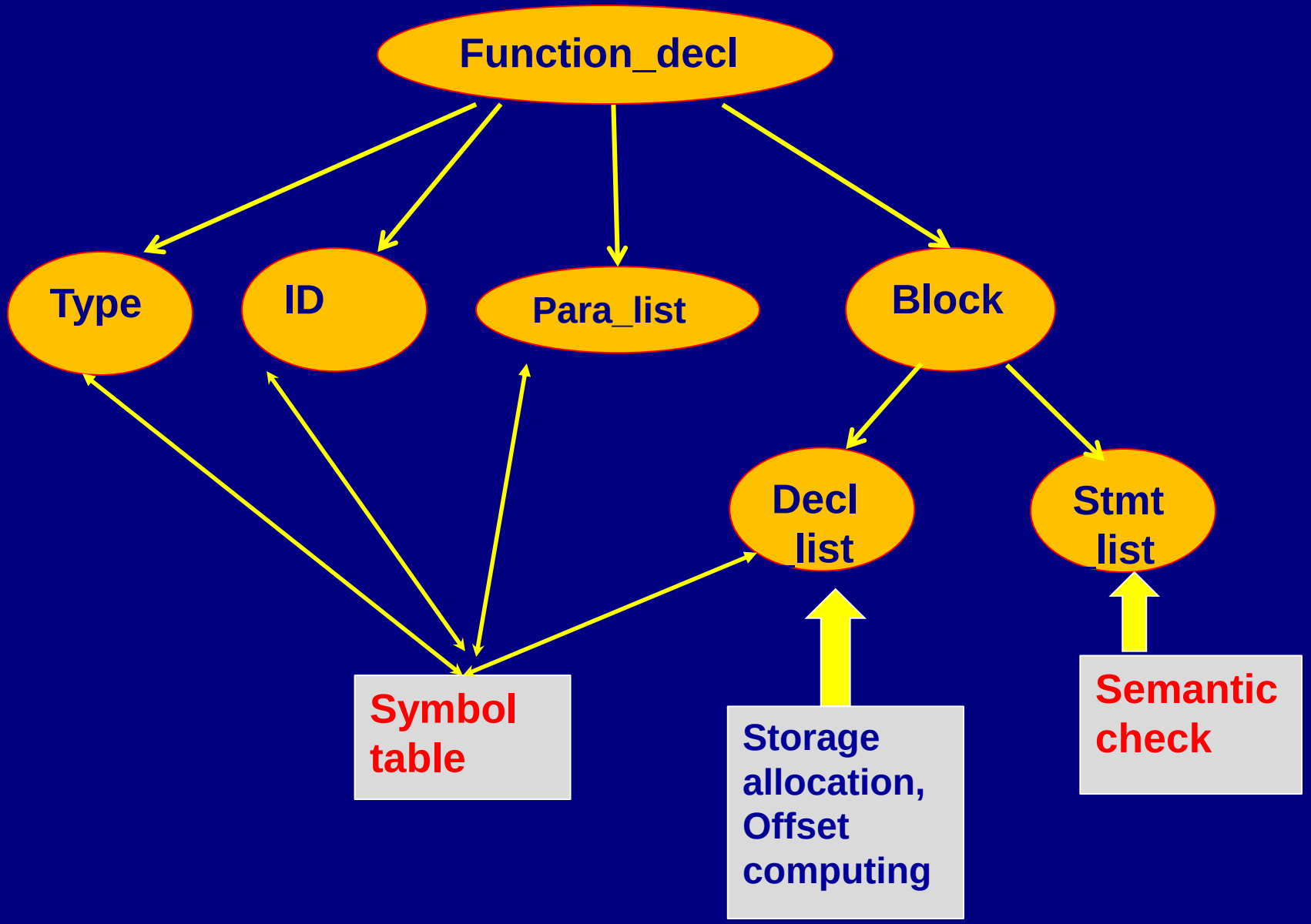
Prof. Farn Wang

Outlines of Chapter 8

- Symbol Tables
  - Constructing a symbol table
  - Block-structured language and scope
  - Basic implementation techniques
  - Advanced features
- Declaration Processing
  - Variable and type declarations
  - Function declarations
- Type Checking

Symbol Table

- A database of the declared *names*
- Attributes associated with a name
  - Class: type, variable, function, label
  - Type: int, float, double, array, boolean, struct
  - Scope
  - Memory address (for code gen)
  - Access attributes (for code gen)
- Attributes are *collected* when a name is declared. Attributes are *checked* when the names are used in statements.


Function_decl
Type
ID
Para_list
Block
Decl list
Stmt list
Symbol table
Storage allocation, Offset computing
Semantic check

```
Int f (float h1, float h2, float h3)
void g(int a)
{
    int w,x;
    {
        float, x,z[100][10];
        f(x,w,z);
    }
}
```

Symbol number	Symbol name	Attributes
1	f	Int function (float, float, float)
2	g	void function (int)
3	w	int
4	x	int
5	x	float
6	z	float; array; two dimensional; 1000w
7	a	int

Static Scoping

- PLs use scopes to confine a symbol in region.
- Scopes may be nested.
  - A name is usually associated with its ***closest containing scope***.
- Variables, types, functions may be declared in a scope.
- A name usually can be declared only once in a scope (overloading is an exception)
- C is a flat language in that it does not allow functions to be declared in another function. (Pascal, ML allow this)

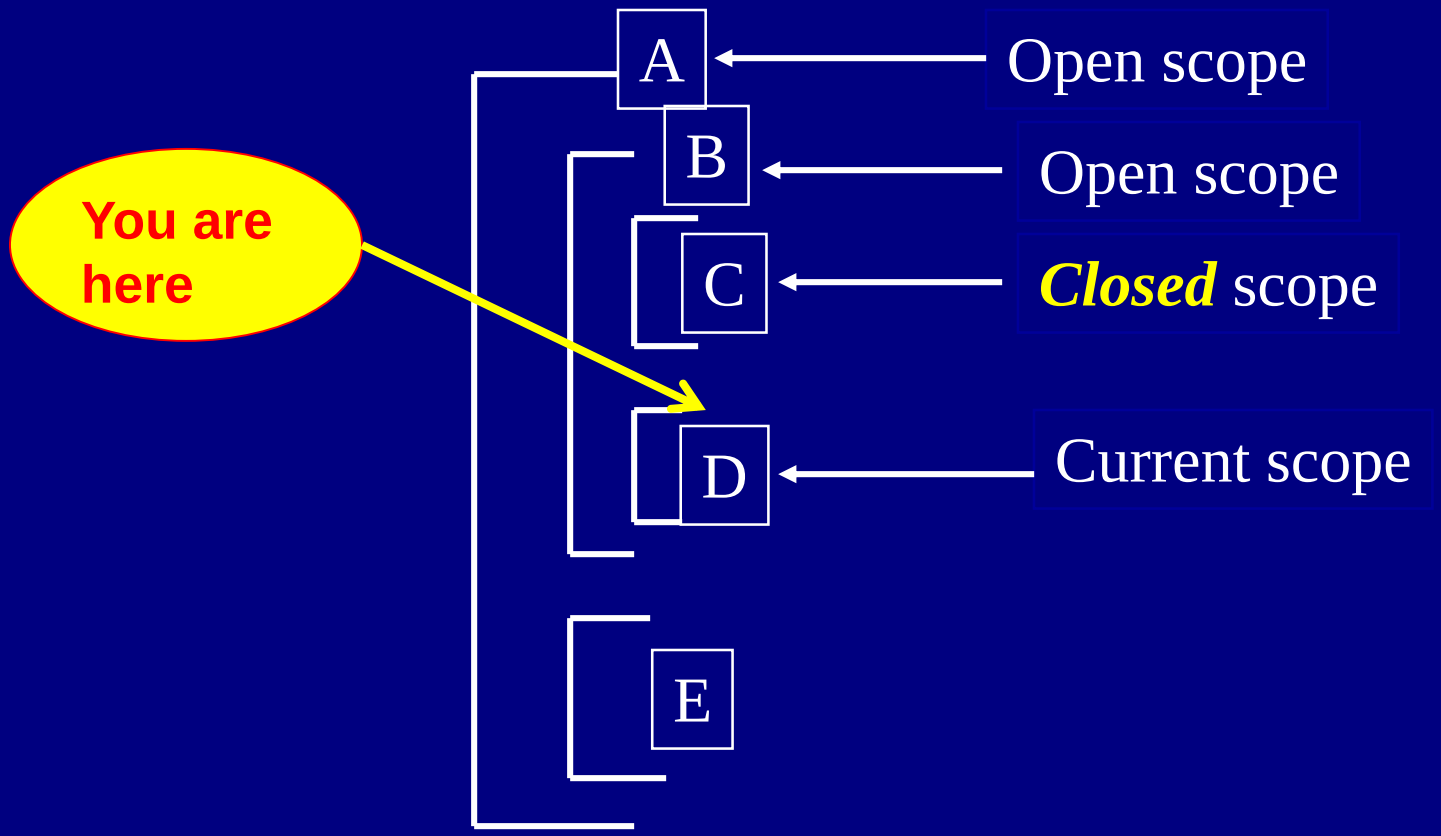
A Symbol Table Interface

- OpenScope()
  - Open a new scope
- CloseScope()
  - Close the current scope, and revert to a previous scope
- EnterSymbol(name, type)
  - Enter the newly declared name into the symbol table for the current scope with the type info.
- RetriveSymbol(name)
- DeclaredLocally(name)

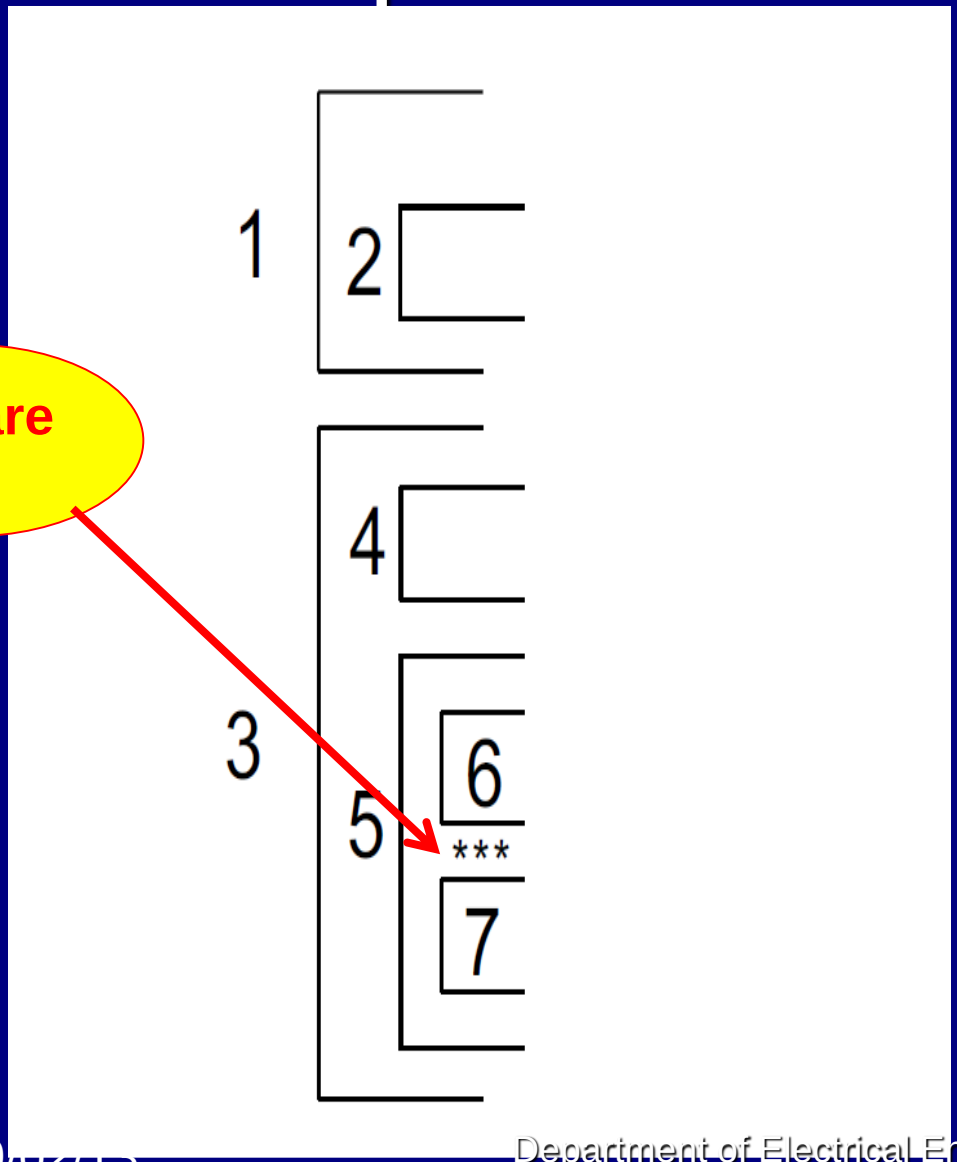
Block Structured Languages and Scopes

- Visibility rules
  - At any point in the text of a program, the accessible names are those that are declared in the *current scope* and in all other *open scopes*.
  - If a name is declared in more than one open scope, then a reference to the name is resolved to the innermost declaration.
  - *New declarations* can be made only in the current scope.

Scope Rules Example



Scope Rules Example



1 Closed

2 Closed

3 Open

4 Closed

5 Current

6 Closed

Scope Rules in C--

- Global variables, local variables, formal parameters
  - Example:

```
int i,j,k;
int f1(int a, int j)
{
    float k;
        while (a < 1)
          { int k;
                i= j + k; }
}
```

Scope 0 → i,j,k

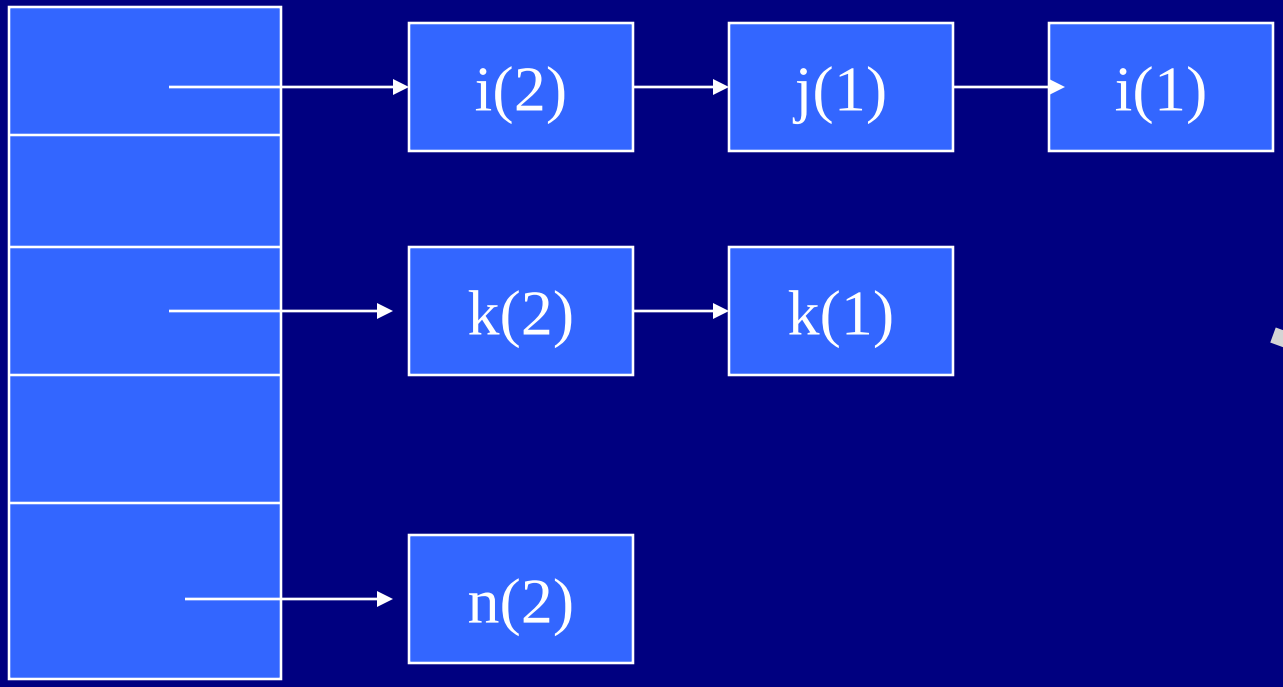
Scope 1 → a,j,k

Scope 2 → k

Which scope is f1 in?

Symbol Table Organization

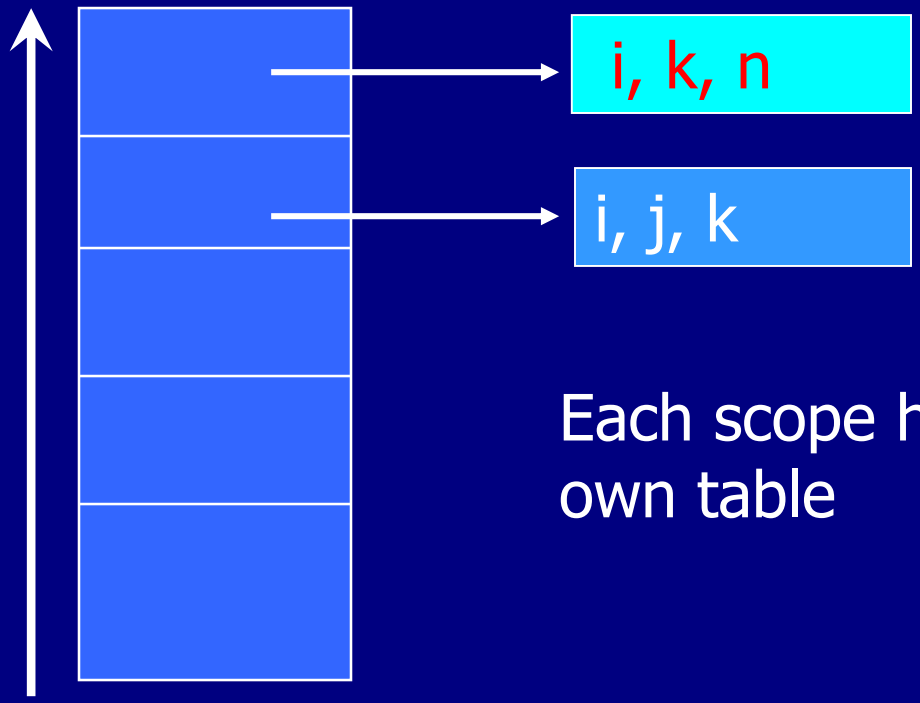
- A single symbol table implementation
  - The pair <name, scope #> should be unique
  - To close a scope is more expensive



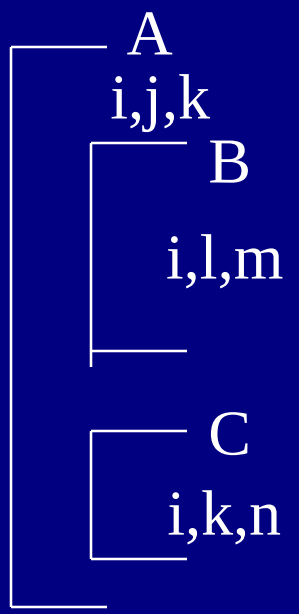
Q: where are B's names (i,l,m)?

Symbol Table Organization

- Multiple Symbol Table Implementation
  - Push a new table when a new scope is entered
  - Pop a symbol table when a scope is closed



Each scope has its own table



Scope Stack

Many small table vs. One big Table

Many small table	One Big Table
Simpler	More complicated (how to locate all names in a scope)
Slower search (for global or undefined variables)	Faster search
Less efficient storage usage (no sharing among small tables)	Efficient storage usage (but needs to store scope #)
Hash table or search trees	Hash table only
Good for keeping symbol tables of closed scopes	Good for 1-pass compilers (entries may be discarded after a scope is closed)

Figure 8.6
A symbol table entry

Name	Type	Var	Level	Hash	Depth

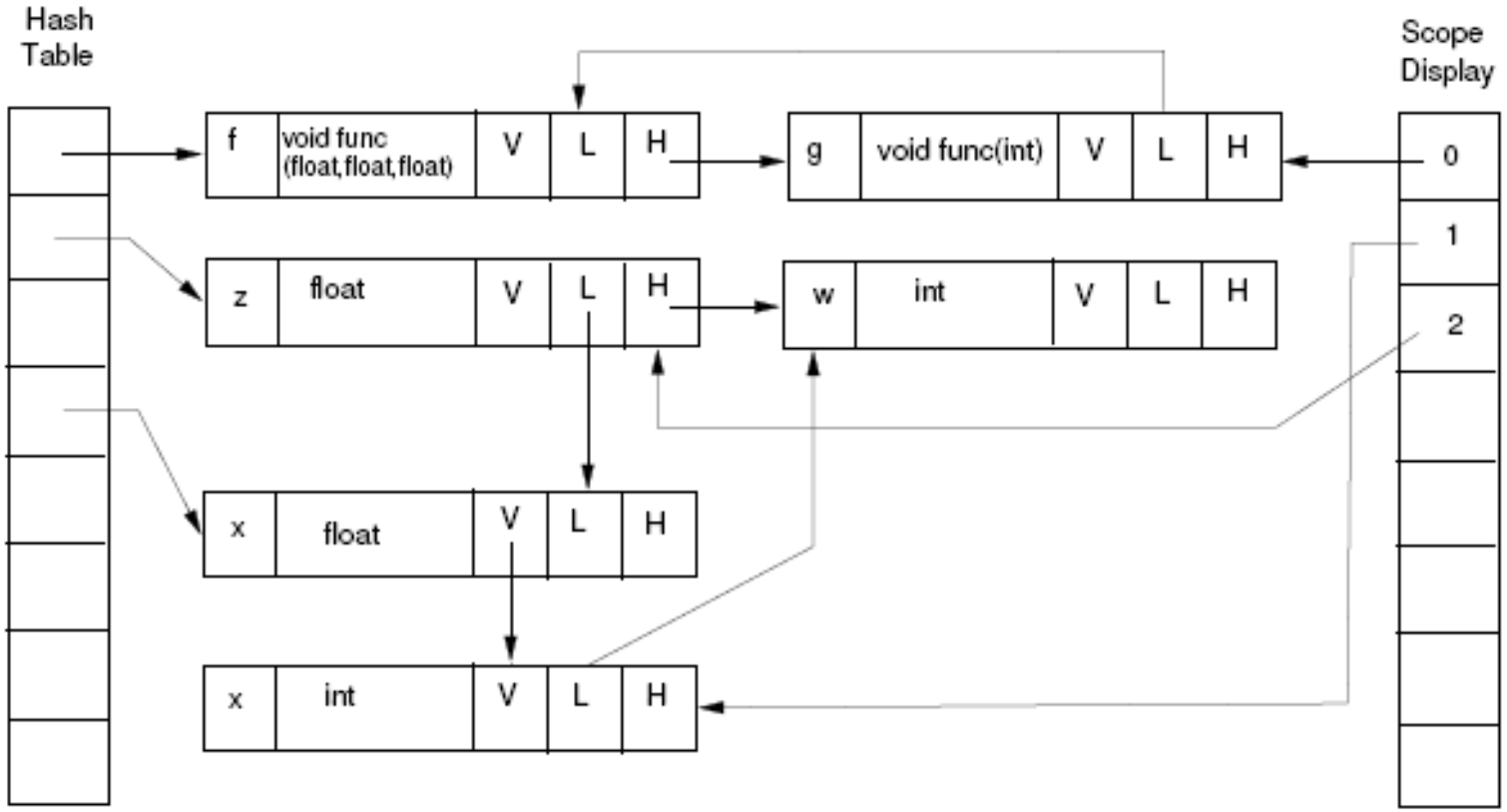


Figure 8.8: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively

Symbol Table Example

```
symtab *hashtable[ TABLE_SIZE ];
struct symtab {
        char *name;
        int scope;      /* or store the VAR info */
        int level;      /* threads symbols in the same scope */
        Type   type;        /* tag → what type is it? */
        union {
                ARRAY_desc      *array_rec;
                FUNC_desc       *func_rec;
                STRUCT_desc     *struct_rec;
                } symtab_urec;
        struct symtab *next; }
```

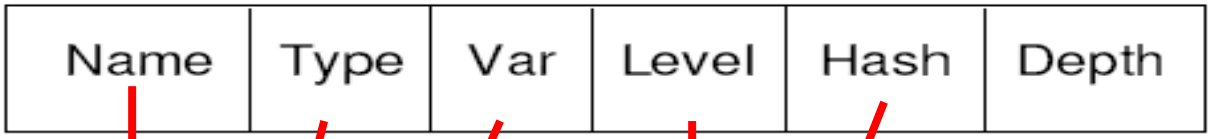


Figure 8.6: A symbol table entry

```
char *name;
int scope;      /* or store the VAR info */
int level;      /* threads symbols in the same scope */
Type   type;        /* tag → what type is it? */
union {
        ARRAY_desc      *array_rec;
        FUNC_desc  *func_rec;
        STRUCT_desc     *struct_rec;
        } symtab_urec;
struct symtab *next; }
```

8.3 Basic Implementation Techniques

- Unordered list
- Ordered list: binary search
- Binary search tree
- Hash tables
  - Constant time search
  - Most commonly used
  - Conflicts resolution: chaining is commonly used

What is a good Hash function ?

In early years, V % P, P is a prime number
e.g. 31,61,127,257,509,1021,…

Name Space

- Should we store the identifier's name in the symbol table?
  - Name length differ significantly, space is wasted if space of max size is reserved.
- Solution
  - Store the identifiers in a name space, and store the index and its length in the symbol table.

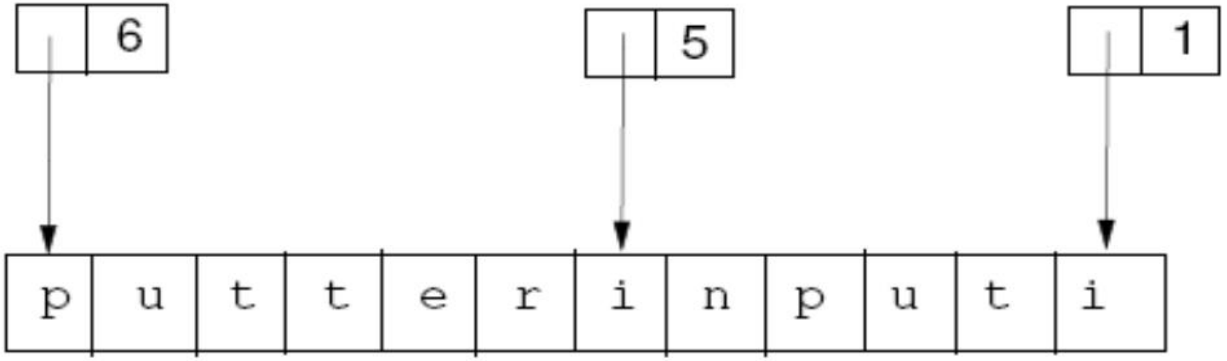
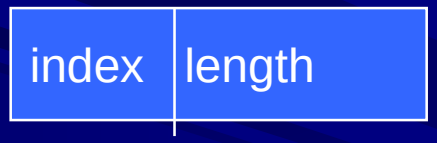


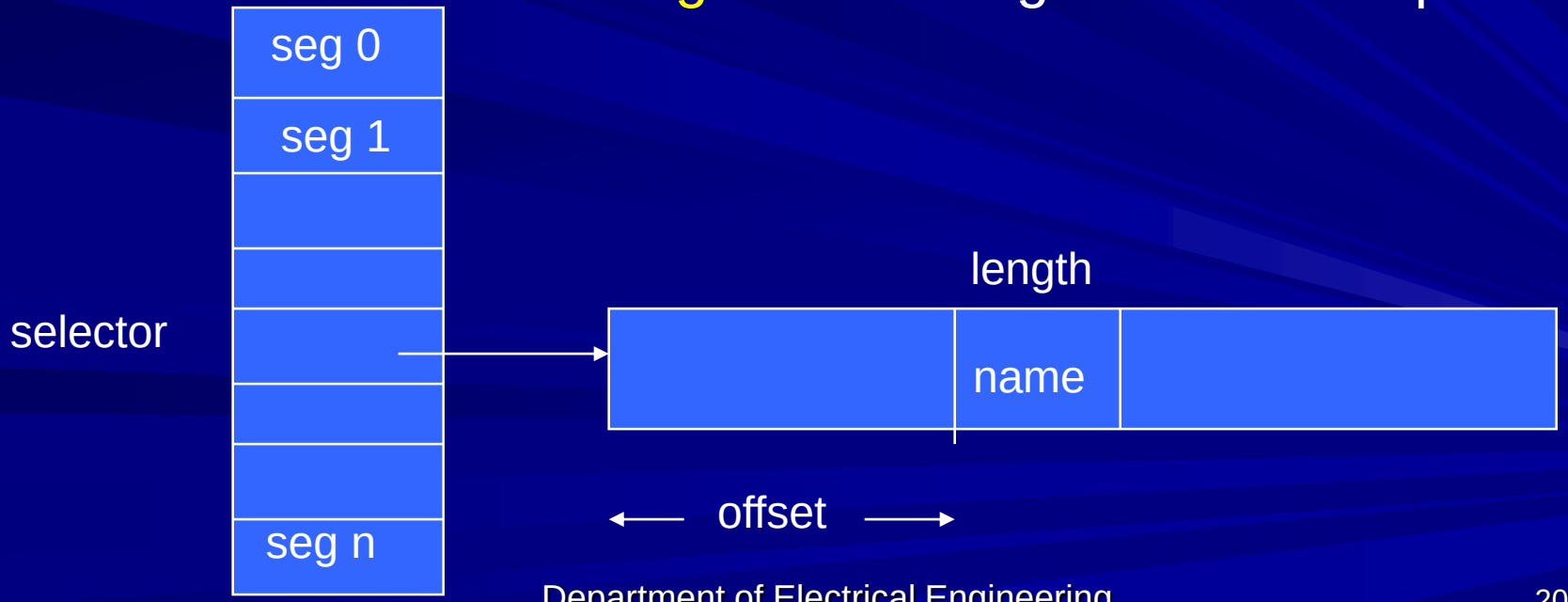
Figure 8.5: Name space for symbols *putter*, *input*, and *i*

Segmented Name Space Index

Segmented name space: string table can be a collection of fixed size string segments, dynamically allocated.



Selector = index div slength
Offset = index mod slength
slength is the length of each seqment



Semantic record in Parser.y layout

%{

C declarations

%}

Yacc/Bison declarations

%%

Grammar rules (including action routines)

%%

Additional C code

Parser.y layout

```
%union {
    int num;
    char* lexeme; …}
%token  <lexeme> ID
%token  <num> const
%type  <num> expr term
%%
expr : expr ‘+’ term      {$$=eval(‘+’,$1,$3)};
%%
int eval (char c, int a,b) {}
```

Set up token attributes

- Step 1: define the %union in parser.y
- Step 2: modify the lexer.l to pass token attributes to the parser.

In lexer.l, insert code as follows:

Name Space?

```
{ID}          {yylval.lexeme = strdup(yytext);
               return ID; }
{int_constant}  {yylval.value = atoi(yytext);
   …}
```

But constants can be integer constants, float point constants and string constants, how to handle constants other than integers ?

Set up token attributes

- Step 1: define the %union in parser.y
- Step 2: modify the lexer.l to pass token attributes to the parser.

In lexer.l, insert code as follows:

```
{ID}          {yylval.lexeme = strdup(yytext);
                return ID; }
{int_constant}  {yylval.value = atoi(yytext);
   …}
```

```
factor  : ( relop_expr )
        : CONST
        : var_ref
        : .....
```

Example

```
%union {
        char* lexeme;
        Const_Type *const_rec;
        TYPE  type;             /* general semantic rec type */
        int             intval; /* evaluate const exp */
        List            *list;          /* handle all lists */
        ArrayInfo       *arr_info;
        VarRef          *var_ref;
        …. }
```

- Token attribute type

```
%token  <lexeme> ID
%token  <const_rec> CONST
```

```
Typedef struct {
C_type  ctype;
Union {
    int intval;
    double fval;
    char *sc; } const_u;
} Const_Type;
```

Set up token attributes

In lexer.l, insert code as follows:

```
{ID}            {yylval.lexeme = strdup(yytext);
                  return ID; }
{int_constant}          {Const_type *p; p=Allocate(Const_type);
                         p->ctype = INT;
                         p->const_u.intval=atoi(yytext);
                         yylval.const_rec=p;
                         return(CONST);
                         }
{float_const}  {…. p->ctype=FLT; p->const_u.fval=atof(yytext);
    …..}
{s-const}       {…. p->ctype=STRING,…}
```

Scope Implementation in 1-pass

- funct_decl → type ID (param_list) { block }

Inserting actions:

funct_decl → type ID ({scope++;}

param_list)

{EnterSymbolparam($5);

EnterSymbolfunc($1,$2,$5);}

{ block } {scope--}

1 2 3 4 5 6

Scope = 0

- Stmt → (block)

stmt → ({scope++;} block) {scope--;}

Building a List (e.g. parameters)

```
%type <plist> param_list
%type <p_var> param
p_list_type         *plist;
param_type          *p_var;

param_list :
      param_list “,” param  {$$=merge_list($1,$3);}
      | param {creat_list($1)};
      ;
```

Merge_list → should $3 be insert at the front or at the end of $1 ?

Extensions

- Fields and Records

```
Struct A {
int a, b;
float c;
double d;
char e;
}
```

Two possible implementations:

1) Allocate a symbol table for each struct.
2) Each struct is assigned a unique number. Each field name is stored in the global table associated with its struct number.

Scope of Struct Type

```
Struct S {
        Struct T {
                int a;
                int b;
                }  x;
        double z1, z2, z3;
        Struct T y;
}
Struct T xyz;
```

OK in C!
But not in C++

Innermost containing block

```
Struct S {
        Struct S next;
        int a;…}
```

illegal!

```
Struct S {
        Struct S {int a;int b;}  x;…
```

illegal!

```
Struct S {
        int a;
        Struct S *next;…
```

OK!

Handling Array Declaration in 1-pass

- Var_list : id
  | Var_list ‘,’ id
  | id dim_decl
  | Var_list ‘,’ id dim_decl

Adding action routines:

Var_list : Var_list ‘,’ id dim_decl
{ addtype($3, scope, current_type, $4);}
| id dim_decl
{ addtype($1, scope, current_type, $2);}

dim_decl : ‘[‘ cexpr ‘]’ | dim_decl ‘[‘ cexpr ‘]’ {get dim info}

cexpr means constant expressions

For each array declaration, we need to record *the number of dimensions* and *the size of each dimension*.

A[3x5+6]
cexpr

Handle Array Declarations in AST

- Var_list : id dim_decl
  | Var_list ‘,’ id dim_decl

```
AST_Node *temp;
Type_Array *Array_info;
Array_info = (Type_Array*) malloc(sizeof(Type_Array));
Array_info->dim=0; /* init */
While(temp->type==DIM_DECL){
        dim_number=do_cexpr(temp->child);
        if (dim_number <= 0) {error_msg(); }
        arrinfo->dim_limit[arrinfo->dim]=dim_number;
        arrinfo->dim=arrinfo->dim+1;
       temp = temp->sibling;
    }
```

Attributes collection during AST tree walk

Attribute Collection in Parser

- Example: Array Declaration

```
%type <num> cexpr
%type <dim_rec> dim_decl
 dim_decl : '[' cexpr ']'
      {$$->n =1; $$->dimsize[0]=$2}
            | dim_decl '[' cexpr ']'
      {$$->n = i = $1->n +1; $$->dimsize[i]=$2}

 cexpr : cexpr '+' cexpr {$$ = $1 + $3}
       | cexpr '*' cexpr  {$$ = $1 * $3}
```

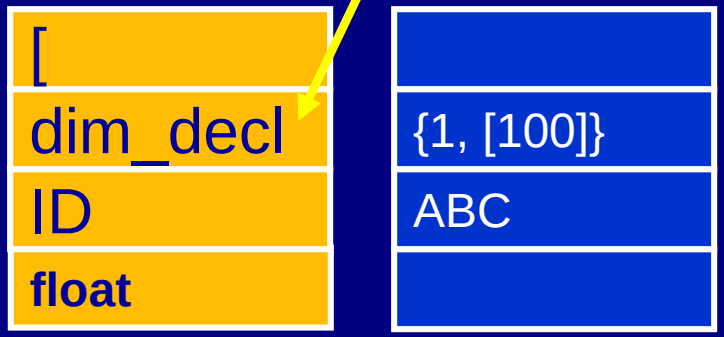
dim_decl : ‘[‘ cexpr ‘]’

{$$->n =1; $$->dimsize[0]=$2}

| dim_decl ‘[‘ cexpr ‘]’

{$$->n = i = $1->n +1; $$->dimsize[i]=$2}

float ABC [100][200][30];



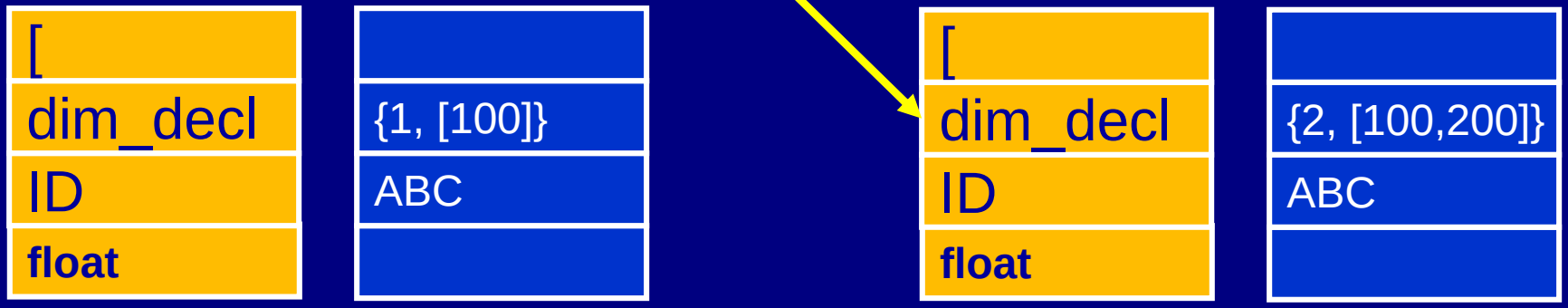
dim_decl : ‘[‘ cexpr ‘]’

{$$->n =1; $$->dimsize[0]=$2}

| dim_decl ‘[‘ cexpr ‘]’

{$$->n = i = $1->n +1; $$->dimsize[i]=$2}

float ABC [100][200][30];



Overloading

- Same symbols may mean more than one thing
  - In Pascal, the same identifier can denote the name of a function and its return value

    f := f(0,1) + 1; f on the left-hand side denotes the return variable, f on the right-hand side denote the function call.
  - In Ada, the name of a procedure, function, operator can be overloaded.
- The symbol table must provide all the possible meanings of a name.
- The key is to link together all possible definitions.

Forward References

- Handling forward references

  there is a risk that a forward reference to a name may be resolved to a non-local definition.

  Example:

```
type T = integer;
procedure foo;
        type   P = ↑T;
               T = real;
```

  P should be a pointer to real, but it is easy to incorrectly interpret P as a pointer to integer.

- The above problem can be resolved by chaining together all references to a pointer to type T until the end of the type section is reached. All type names can then be looked up and resolved.
- Forward references to labels in goto's can also be resolved in a similar way. Once the label address is known, we can back filled the branch/jump instruction.
  - In Pascal, labels must be declared before they can be defined.
- If unrestricted forward references are allowed, the only way to handle them is to make more than one pass over the program.

Type Checking

- Type checking makes sure the source program follows semantic rules.
- Examples of type checking
  - An object is defined exactly once
  - An operator is applied to compatible operands
  - De-referencing is applied only to a pointer
  - Indexing is done only on array objects
  - Function is called with the correct number and type of arguments

Static and Dynamic Type Checking

- Static type check: type checking done at compile time
- Dynamic type check: type checking done at run time. Runtime objects have tags.
- A strongly typed language eliminates the need for dynamic checking of types
- Some type checking can only be performed at runtime
  - Array index for array [0..255] of integer

Type checking of expressions (AST)

```
CheckExpr (AST_Node* exprnode)
{
    AST_node* leftop = exprNode->child;
    AST_node* rightop = leftop->rightSibling;
    if (leftop->datatype != rightop->datatype)
        Error….
}
```

But how do we know the type of each sub-expression ?

a) recursive traverse down

b) bottom up traverse to compute synthesized attributes

Type Conversion

- Implicit type conversions called *coercions*
  - In assignments
  - In expression evaluation
  - In parameter passing
  - In function return
- Coercion is often a symptom of weak typing
- Explicit type conversions are performed by programmers such as type casting.

Type Expressions

integer is a basic type

array (1..100) of integer is a ***type expression***

struct { integer a; float b;} is a type expression

* integer is a type expression

int f(integer a; float b;} is a type expression

struct S { integer a[100][200];
float *b;
struct { integer a; float c;} d; } is a type expression

Equivalence of Type Expressions

- Name Equivalence

  two variables have the same type if they have the same associated type expressions

- Structural Equivalence

  two variables have the same type if their associated type expressions are *structurally equivalent*, i.e., two type expressions are either the same basic type, or are formed by applying the same constructor to *structurally equivalent* types.

A, B : array (1..100) of integer;

C, D : array (1..100) of integer;

A and B are of the same type, C and D are of the same type, but the two types A and C may be incompatible.

Ada, Pascal, Modula-2 are using this *name equivalence* rule.

Algol68 uses *structural equivalence* rule, so A,B,C,D are equivalent.

C uses *name equivalence* for structs and unions, and *structural equivalence* for arrays

A, B : array (1..100) of integer;

C, D : array (1..100) of integer;

A and B are of the same type, C and D are of the same type, but the two types may be incompatible.

Question:

The two type expressions are actually identical ! So why is A not compatible with C?

They are not the same ***name***

(implicit name for type expression)

Modern languages tend to use name equivalence rules, because

1) Type comparison is just a comparison of pointers
2) Programmers can get full benefit of type checking
3) Useful for type-inferred aliasing analysis
   - less possible aliasing
4) It is hard (time consuming) to implement structural equivalence.

C type equivalence test

```
Char *p, *q;
Short j;
Short int k;
Int a[100], b[200], c[100];
Int d[]; /* as in function parameter list or extern */
```

The types of p and q are the same
The types of j and k are the same
The types of a and b are different
The types of a and c are the same
The types of a and d are the same

C type equivalence test

```
Struct {int a,b;} x, y;
Struct S {int a,b;} w;
Typedef Struct S T1,T2;
Struct S u;
T1 z;
T2 zz;
```

Q1: x, y and w are compatible T or F F

Q2: z, zz and u are compatible T or F T

C type equivalence test

```
Struct {int a,b;} x, y;
Struct S {int a,b;} w;
Typedef int A[3];
A B[10];
Int C[10][3];
```

x = w; — No

C = B; — No

Implicit conversion of constants

- Assuming X is an array of float.

  In the following For loop

```
for (I=0; I< n; I++)
        X(I) = 1;
```

  the compiler can automatically convert it to fp constant with no runtime cvt operations

```
for (I=0; I< n; I++)
        X(I) = 1.0;
```

  This speeds up the loop by 9 times

  (some machines are slower on fp-int conversions)

Overloading

- An overloaded symbol is one that has different meanings depending on its context.

  Example:

  - Arithmetic operators +,-,*,/
- The key to handling overloaded names in the symbol table is to link together all possible definitions.
- Overloading is resolved when a unique meaning can be determined.

  function “+” (x,y: complex) return complex

  function “+” (x,y: integer) return complex

  function “+” (x,y: integer) return integer

  A,B: complex then A+B is not ambiguous

  but what if A,B are integer?

Overloading

function “+” (x,y: complex) return complex

function “+” (x,y: integer) return complex

function “+” (x,y: integer) return integer

R = A + B + C what is the type of R if A, B, C are integer?

R = (A+B) + (C+D) what is the type for (A+B) and (C+D) if A,B,C,D are integer?

Ada requires a complete expression to have a unique type, we can use the unique type infor to narrow down the type choices for each subexpression.

Type inferred aliasing analysis

Pointer analysis, aliase analysis, heap analysis

```
{
    a = f(x);
    *p = b;
    c = a + c;
}
```

Register Allocation

Can we reuse a ?

Code Scheduling

Can we schedule the load of c ahead of the store of *p?

*Yes, if data type of *p is different from a and c*

Expression type equivalence checking

How to tell if two type expressions are identical?

```
struct { int a; float b[5];
         struct { int c; double d;} x,y;
       }
struct { int a; float bb[5];
         struct { int cc; double dd;} x,y;
       }
```

Are these two type expressions identical?

They are structural equivalent, not ***name*** equivalent.

How to test if two type expressions are identical, if structural equivalence is used?

```
struct { int a; struct { int c; double d;} x,y;
                double b[5];
            }
struct { int a; struct { int cc; double dd;} x,y;
                float bb[5];
            }
```

A fast encoding would tell they are NOT equivalent.

Efficient test for structural Inequivalence

- Type expressions can be encoded as an *integer value*. A simple integer comparison can tell structural inequivalence immediately.

1) Basic types: boolean: 0, char 1, integer 2, real 3
2) Type constructors: pointer 1, array 2, freturns 3

- Type expressions
  - char 0 0 0 1
  - freturn(char) 0 0 3 1
  - pointer(freturn(char)) 0 1 3 1
  - array(pointer(freturn(char))) 2 1 3 1

How to test if two type expressions are identical, if structural equivalence is used?

struct { int a; float b[5];
struct { int c; struct {.. Int k[3] …} w;}
x,y;}
struct { int a; float bb[5];
struct { int cc; struct {.. Int k[4] …} z;}
x,y;}

A fast encoding would fail to tell if they are equivalent. So a more expensive recursive algorithm must be used.

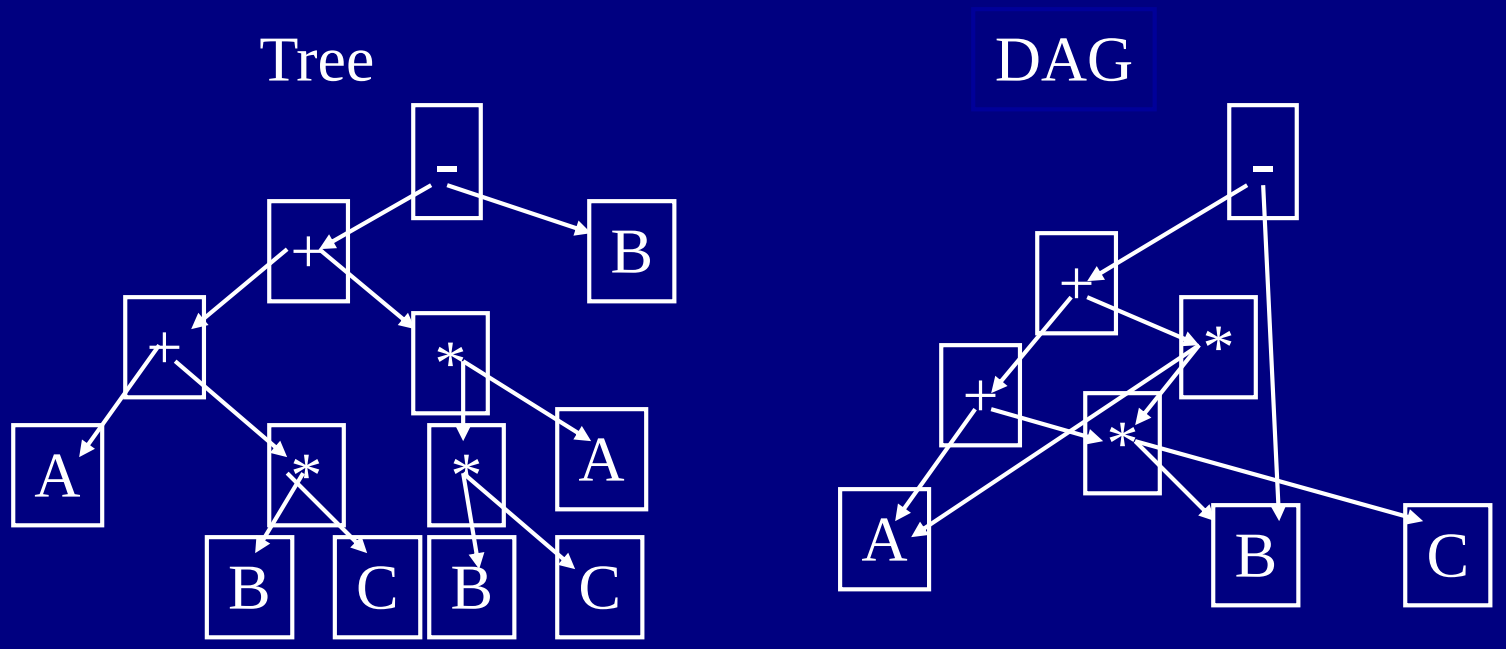
How to tell if two expressions are identical?

```
x = (A+B*C);
.......
y = (A + B*C);
```

- Are these two expressions identical?
- Do they generate the same value?

DAG for Expressions

- A Directed Acyclic Graph (DAG) for an expression identifies the Common SubExpressions (CSE) in the expression.
- For expression A+(B*C) + (B*C*A) - B



Value Numbering

- Suppose each node is stored in an array, and is referenced by its index (called value number).

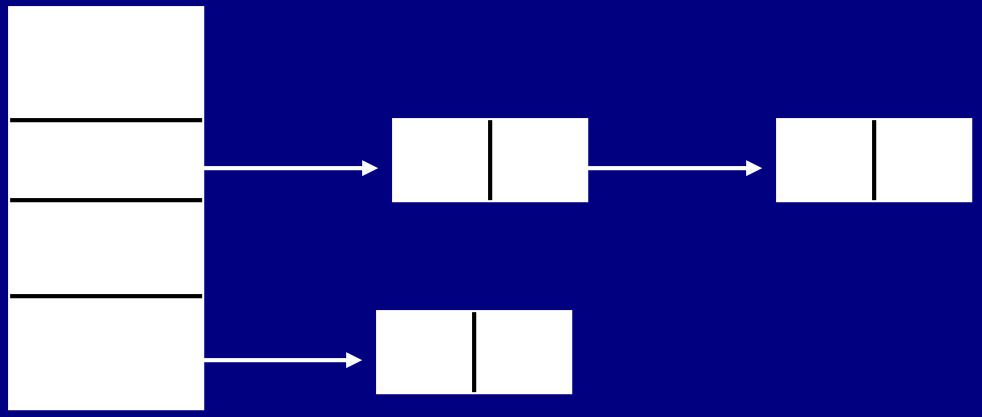
I := I+10

1	id	Pointer to symtab	
2	num	10	
3	+	1	2
4	:=	1	3
5	…		

- The signature of an operator node be a triple <*op*,*l*,*r*> consisting of its label *op*, left child *l*, and right child, *r*.
- Search the array for a node *m* with <*op*,*l*,*r*>, if found, return *m*, else create a new node.

Value Numbering

- An efficient way to search for <op,l,r> is to use hashing.
- All nodes are kept in k lists, called buckets.
- A hash function computes the number of bucket from the signature <op,l,r>.



Example

- (A+B*C) – D *(A+B*C)
- Assume A,B,C are stored in entry 4,5,6.
  B*C is stored in entry 15
  A+B*C in entry 16
  when the 2nd A+B*C is processed, we will search for <*,5,6> first, then <+,4,15>