Semantic Analysis

Dragon: Ch 6. (Just part of it)

Holub:

What is semantic analysis?

- Semantic validity
  - Parser and Lexer ensure the input has valid structure
  - Need to check if the input has valid meaning (the meaning of a program is the result of the computation)
- Static semantic checking at compile-Time
  - Type checking: if operand types match the operator
  - Flow-of-Control: if having well defined “jumps” (e.g., if there is a “continue”, there should be enclosing iterator)

Limitations of Semantic Analysis

- Lexical & syntactic analysis is advanced
  - Well worked-out theories to provide precise description of aspects of programming language
    - Regular expression and context free grammar
  - It is possible to "compile" these descriptions into lexical and syntactic analyzer automatically
    - `lex` and `yacc`

- Semantic analysis is less advanced
  - A great deal of investigation is going on in formal semantics of programming languages
  - It is difficult to write a precise description of semantics of a programming language (though possible)
  - Automatic compilation of such a description into a semantic analyzer is beyond the state of the art
  - There have been promising researches, but there still is a long way to go

Static and Dynamic Semantics

- One issue is that the semantics of a program is not entirely determined at compile-time
- In a typical programming language,
  - Compiler decides some semantic issues (e.g., correct binding of types and names)
  - Leave some to the object code to be determined at run-time (e.g., out-of-bound array accesses)
    - However, compiler must assure that the semantics is preserved in by the object code
  - The compile-time part: static semantics
    The run-time part: dynamic semantics

Ensuring Static Semantics

- Semantic analysis phase deals with static semantics
  - Should catch all compile-time semantic errors
  - Keep track of types, declarations, scoping, etc for use in code generation

Declarations & Symbol Tables

- Declarations
  - Associate "meaning" with names
  - For example,
    - Variables (name, type, storage class, scope, etc.)
    - Functions (name, arguments, return type, external or not)
    - Types (type class, size, etc.) ....

Declarations & Symbol Tables

- To handle declarations, we use symbol tables
  - A global data structure to map a name with values or attributes at compile-time
    - E.g., `int x;` `x` is an integer variable
  - Symbol tables become complicated when the mapping depends on contextual information (ex: block-structure)
- In our compiler, we have two kinds of symbol table
  - “Flat” symbol table
  - “Scoped” symbol table

“Flat” Symbol Table

- Symbol Table that depicts global name definition
  - No contextual information
  - A simple mapping from name to declarations
  - Used in very simple compilers
    - Assemblers or macro processors
- This is actually a dictionary abstract data type
  - **Insert(Name, Decl)**: map the name with declaration
  - **Lookup(Name)**: get the declaration with that name
  - **enter(Name):** combination of the two (i.e., inserts Name if it is not already there and returns the declaration)

Flat Symbol Table in Lexer

- Our project compiler will have two symbol tables, one of which is flat (i.e., the hash table we used)
  - Lexer enters name of an identifier into this symbol table and returns declaration, which is a pointer to **`struct id`**
- After lexical analysis, **`struct id`** pointers are used to represent identifiers everywhere
  - Actually, they are the *names* in another, more complicated symbol table that handles scoping
  - An advantage is that when we want to check if two identifiers are the same, we can compare their **`struct id`** pointers, instead of string comparison
    - But we compare strings in the hash table, anyway don’t we?

Data Structures for Flat Symbol Table

- Linear search structures
  - Array or linked List: easy to program, OK if list is short
- Binary search tree
  - Good asymptotic *log n* average performance
  - In practice, not used in a compiler symbol table from an engineering viewpoint of programmability and performance
- Hash table
  - If there are many symbols, a hash table is good, and if implemented carefully, almost constant insert/lookup

Block Structure & Scoping

- Block structure is one of the most useful features
  - Statement that can have its own data definitions that disappear after exiting the block
  - Prevents accidental name clashes
  - Ex: {*decls; stmts*} in C
  - Blocks can be nested but cannot otherwise overlap

```
int x;     int y;
{
   float x;
   x += y;  /* float += int */
}
x += ..       /* int */
```

Scope & Extent in Block Structures

- The "scope" of a declaration is the portion of a program text for which the declaration is "visible"
  - Global declaration: entire program
  - Local declaration: procedure or block
  - Some names may have many scopes
- The "extent" means the lifetime of the storage associated with the variable
  - Scope & extent are usually the same
  - Exception (e.g., static locals in C)

Scoped Name Definition

- Compiler symbol tables are concerned exclusively with the scope, not the extent
  - Bind names to attributes depending on the scope in which it occurs
- Scope rule determines which declaration applies to a name instance: **most-closely nested rule**
  - The scope of declaration in a block B includes B
  - If name x is not declared in B, then an instance of x in B is in the scope of the declaration of x in the most closely enclosing block B’

“Scopd” Symbol Table

- Abstract scoping operations: use stack paradigm
  - push_scope(): start a new scope which becomes the “current scope”
  - pop_scope(): return to the previous state; restore symbol table before the last **push_scope()**
  - insert(name, decl): basically the same, but it inserts the definition in the “current scope”
  - lookup(name) must now search for the variable in all of pushed but not popped scopes in reverse order in which they were pushed; it returns the first definition

Example

```
{                               push_scope()
    int x;                      insert(x, var int decl)
    int y;                      insert(y, var int decl)
    {                           push_scope()
        float x;                insert(x, var float decl)

        x += y;                 lookup(x) : float; lookup(y): int;
     }                          pop_scope()
     x += ..                    lookup(x) : int
}                               pop_scope()
```

Implementation of Scoped Symbol Table

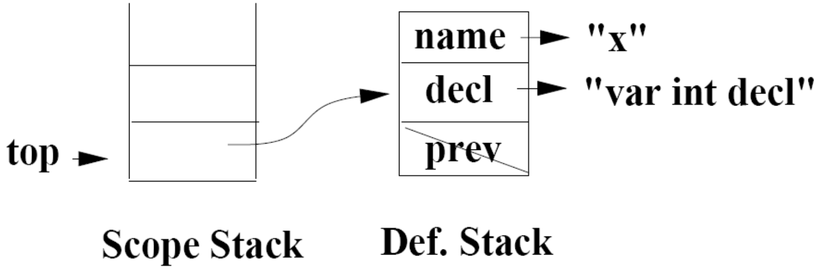
- Stack of flat tables
  - Implement a stack of array elements and each array element is a flat symbol table
  - **push_scope()** and **pop_scope()** literally push and pop a flat table
  - **insert()** inserts in the current scope (table) and **lookup()** does a flat-table lookup in each element of the array from the top
  - Problems: scopes with not many definitions either waste space or require complex implementation

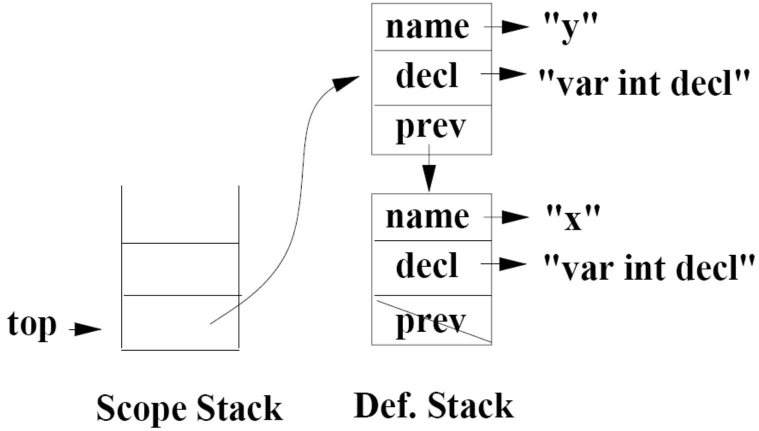
Our Implementation Choice

- Stack of definitions
  - Keep a stack of individual definitions (not scopes) and mark scope boundaries so that **pop_scope()** knows how many definitions to remove from the top of the stack
  - Two methods to make the boundary
    - Inserts a pseudo definition that is recognized as a marker
    - Maintain a separate scope stack which points the to the top of stack when a scope was pushed: we can take this approach
  - **insert()** always inserts to the top of definition stack
  - **lookup()** searches *backwards* in the table

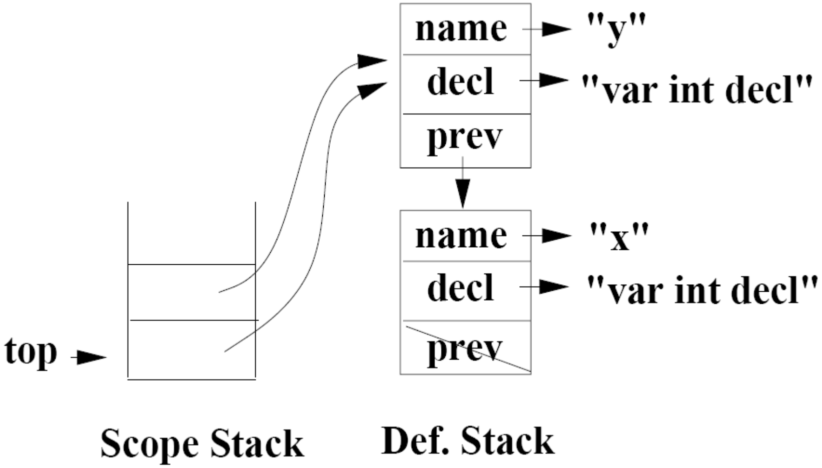
Example

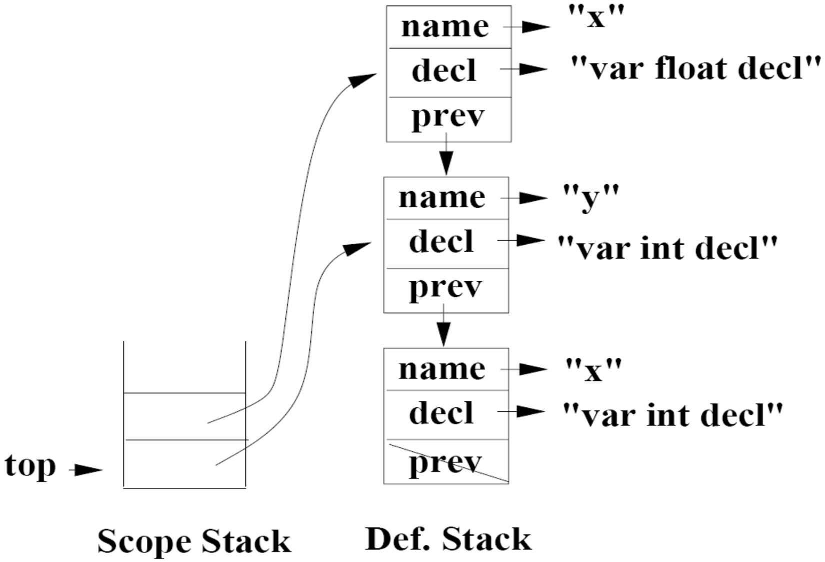
After push _scope(), insert(x, var int decl) → After insert(y, var int decl)





After push_scope() ⟶ After insert(x, var float)





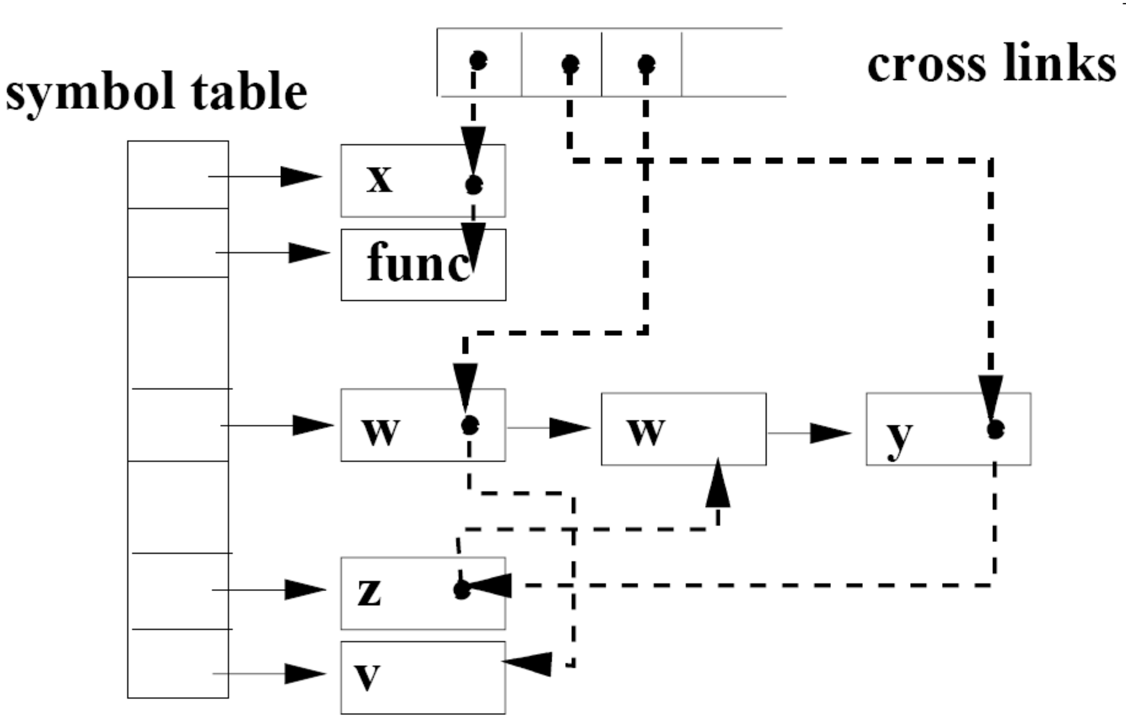
Another Implementation: Hash Table

- Holub: pp. 485–488
  - Maintain a single hash table that implements open hashing
  - A name is hashed and inserted at the beginning of the linked list of that hash slot

Example Hashed Symbol Table

Example:

```
int x;
func(int y, int z)
{
    int w;
    while(expr) {
        int w, v;
    }
}
```



Declarations

- **Name definitions** associate "semantic something" with a name, which is a data structure representing the declaration
  - Processing declaration depends on the language semantics
  - Declarations and names are completely independent things and the only association occurs in the symbol table
  - Association may change in the context and one name may be associated with many declarations
- There are many ways to process declarations and build a symbol table, and we will present one way that is relatively simple yet is directly applicable to processing **C** declaration

An Example of *subc.h*

- Data formats and structures in “subc.h”
  - IDs, symbol table entries, and declarations

```
struct id {
    char          *name;
    int           lextype;
};

struct ste {
    struct id          *name;
    struct decl        *decl;
    struct ste         *prev;
};
```

```
struct decl {
    int             declclass;    /* DECL Class: VAR, CONST, FUNC, TYPE   */
    struct decl     *type;        /* VAR, CONST: pointer to its type decl */
    int             value;        /* CONST: value of integer const        */
    float           real_value;   /* CONST: value of float const          */
    struct ste      *formals;     /* FUNC: ptr to formals list            */
    struct decl     *returntype;  /* FUNC: ptr to return TYPE decl        */
    int             typeclass;    /* TYPE: type class: int, array, ptr    */
    struct decl      *elementvar  /* TYPE (array): ptr to element VAR decl */
    int             num_index     /* TYPE (array): number of elements     */
    struct ste      *fieldlist    /* TYPE (struct): ptr to field list     */
    struct decl     *ptrto        /* TYPE (pointer): type of the pointer  */
    int             size          /* ALL: size in bytes                   */
    struct ste      **scope;      /* VAR: scope when VAR declared         */
    struct decl     *next;        /* For list_of_variables declarations   */
};                                /* Or parameter check of function call  */
```

An Example Declaration in *subc.y*

```
%union yystacktype
{
    int           intval;
    double        flatval;
    char          *stringval;
    struct id     *idptr;
    struct decl   *declptr;
    struct ste    *steptr;
}
%type         <declptr>        type type_id var var_list …
%nonassoc     <idptr>          ID
%nonassoc     <intval>         INTEGER-CONST
%nonassoc     <floatval>       FLOAT-CONST
%nonassoc     <stringval>      STRING-CONST
```

An Example of *init_type()*

```
init_type() {
    inttype         = maketypedecl(INT);
    floattype       = maketypedecl(FLOAT);
    voidtype        = maketypedecl(VOID);
    ..
    declare(enter(ID, “int”, 3), inttype);
    declare(enter(ID, “float”, 5), floattype);
    returnid = enter(ID, “*return”, 7);
```

- In this example, type specifiers like `int` are regarded as a token **ID** instead of a token **TYPE** and the lexer will give **idptr** to **yylval**; later yacc will look through the linked list of the symbol table to determine the declaration that was inserted during the initialization

An Example *subc.l*

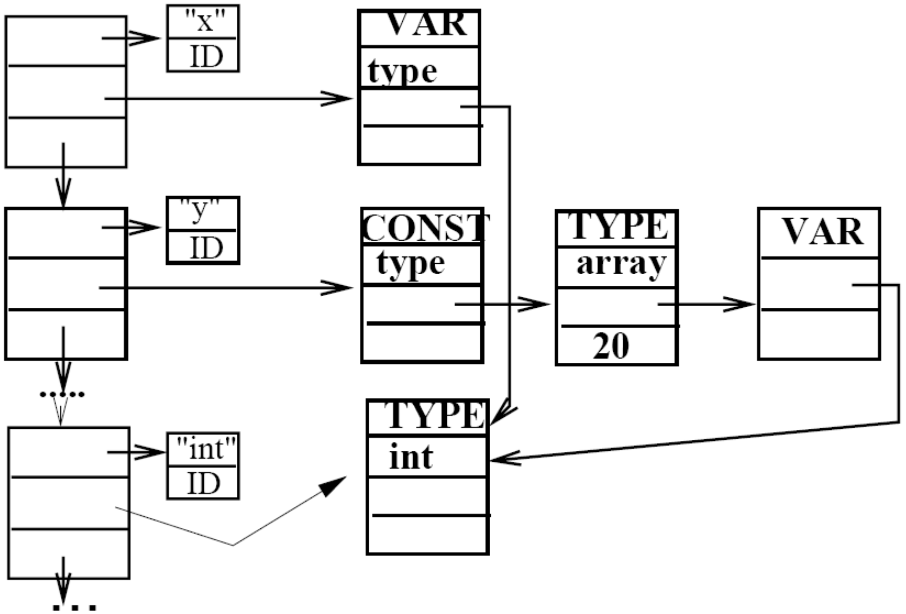
```
…
<norm>{ID} {
    yylval.idptr = enter(ID, yytext, yyleng);
    return (yylval.idptr → lextype);
}

<norm>{DEC_INTEGER} {
    yylval.intval = (int) strtol(yytext, (char**) NULL, 10);
    return (INTEGER-CONST);
}

<norm>{REAL} {
    sscanf(yytext, “%lf”, &yylval.floatval);
    return (FLOAT-CONST);
}
```

Example: Simple Variable Declarations

```
int x
int y[20];
```

<u>grammar</u>

```
var_decl : type ID “;”
         | type ID “[“ const_expr “]”
type     : type_id
         | …
type_id  : ID
```



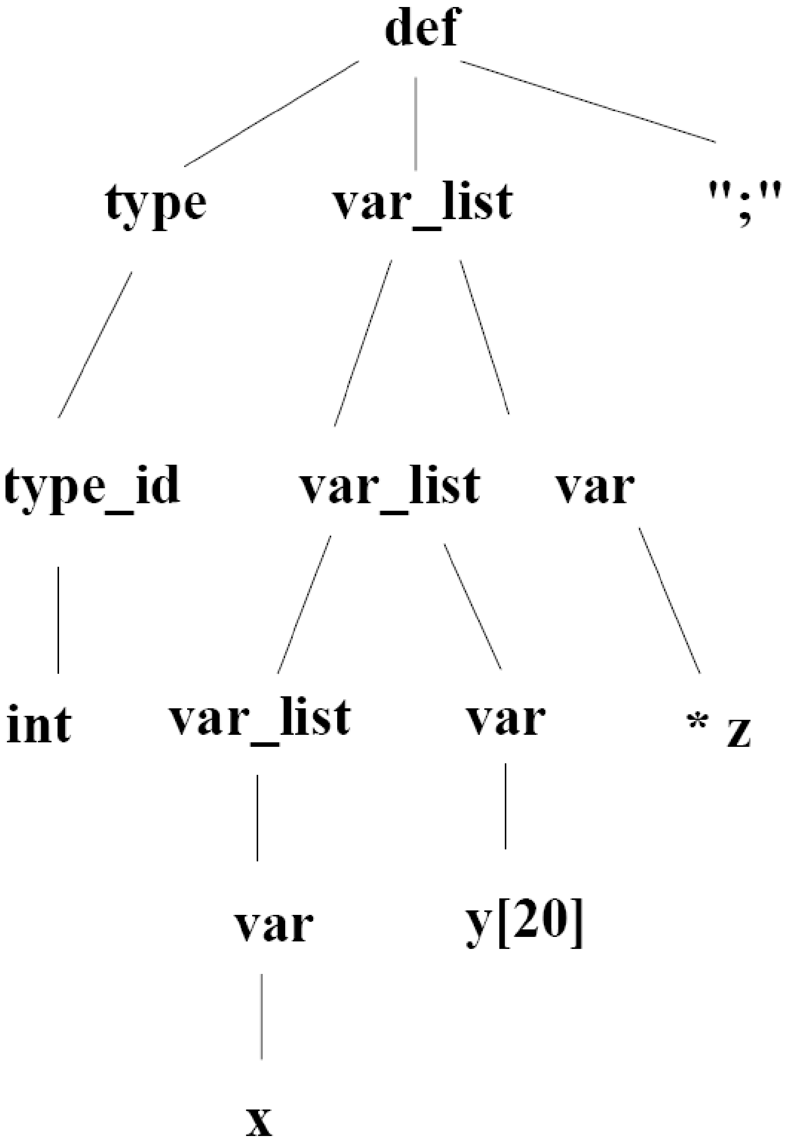
```
var_decl : type ID “;” { declare($2, makevardecl($1); }
           | type ID “[” const_expr “]” “;”
            {declare($2,makeconstdecl(makearraydecl($4,makevardecl($1))));}
type       : type_id {$$ = $1}
           | struct_specifier {$$=$1}
type_id   : ID
            { struct decl *declptr = findcurrentdecl($1);
              check_is_type(declptr);
              $$ = declptr;
            }
```

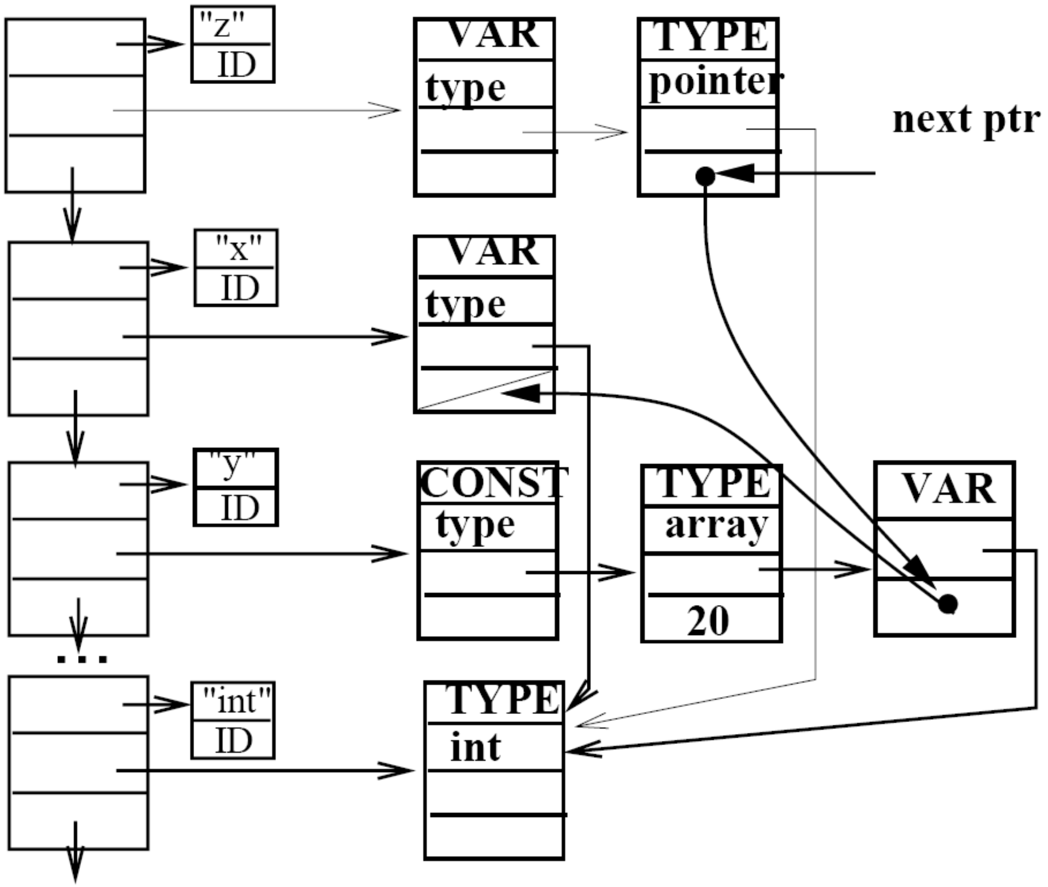
- For array type decls, we made the elementvar ptr to point a VAR decl instead of TYPE decl, to make sure an element of the array in LHS of an assignment statement is a variable when we do the type checking

Example: List of Variable Declarations

- Assume **`struct decl`** has one more field: **`next`** which links decls whose type are not yet defined

```
def         : type var_list “;” {add_type_to_var($1, $2);}
            ;
var_list    : var_list “,” var   {$3→next = $1; $$ = $3}
            | var                {$$ = $1;} /* $1→next is assumed to be NULL */
            ;
var         : ID                           {declare($1, $$ = makevardecl(NULL));}
            | ID “[” const_expr “]” {declare($1,
                  makeconstdecl(makearraydecl($3, $$=makevardecl(NULL)));}
            | “*” ID              {declare($2, makevardecl($$=makeptrdecl(NULL)));}
            ;
```


def
type
var_list
";"
type_id
var_list
var
int
var_list
var
* z
var
y[20]
x

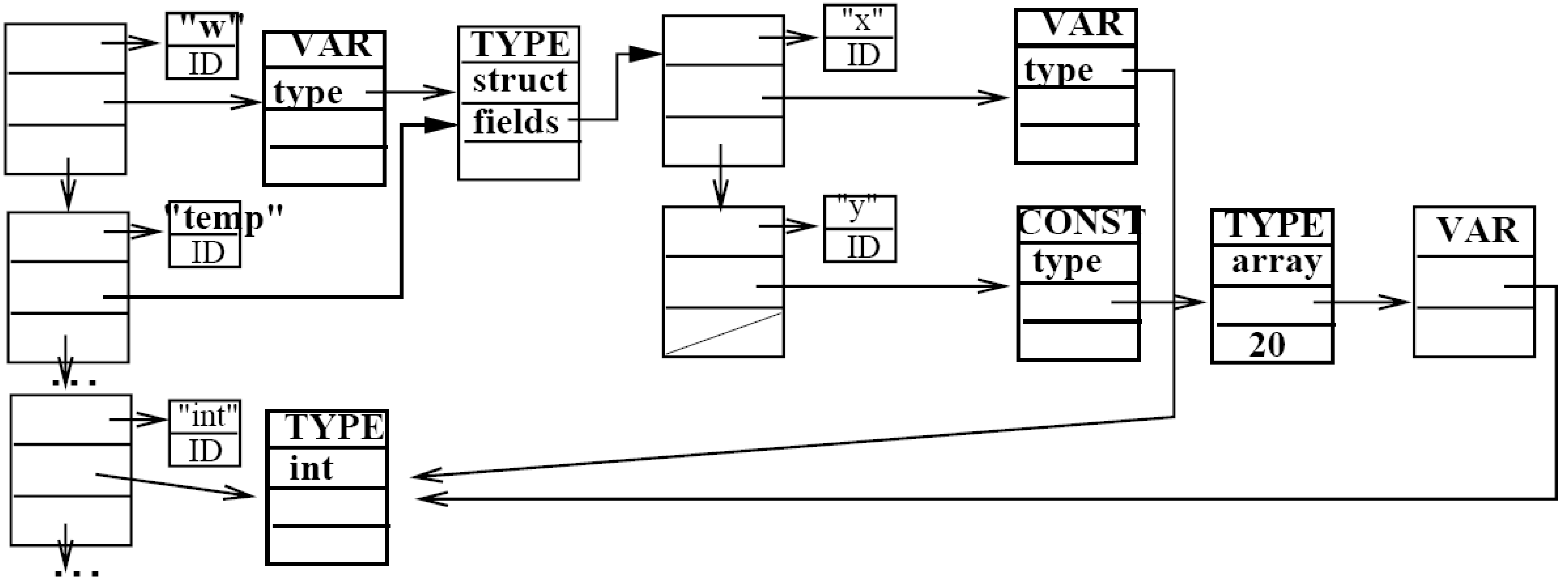

"z"
ID
VAR
type
TYPE
pointer
next ptr
"x"
ID
VAR
type
"y"
ID
CONST
type
TYPE
array
20
VAR
...
"int"
ID
TYPE
int

Example: Struct Declaration

- Structures: how to collect fields?

```
struct_specifier : STRUCT tag “{”
                   { push_scope(); }
                     def_list /* popscope reverses stes */
                   { struct ste *fields = popscope();
                      declare($2, ($$=makestructdecl(fields))); }
                  “}”
                 | STRUCT tag
                   { struct decl *declptr = findcurrentdecl($2);
                      check_is_struct_type(declptr);
                      $$ = declptr;
                   }
                  ;
```

struct temp { int x; int y[20]; } w;

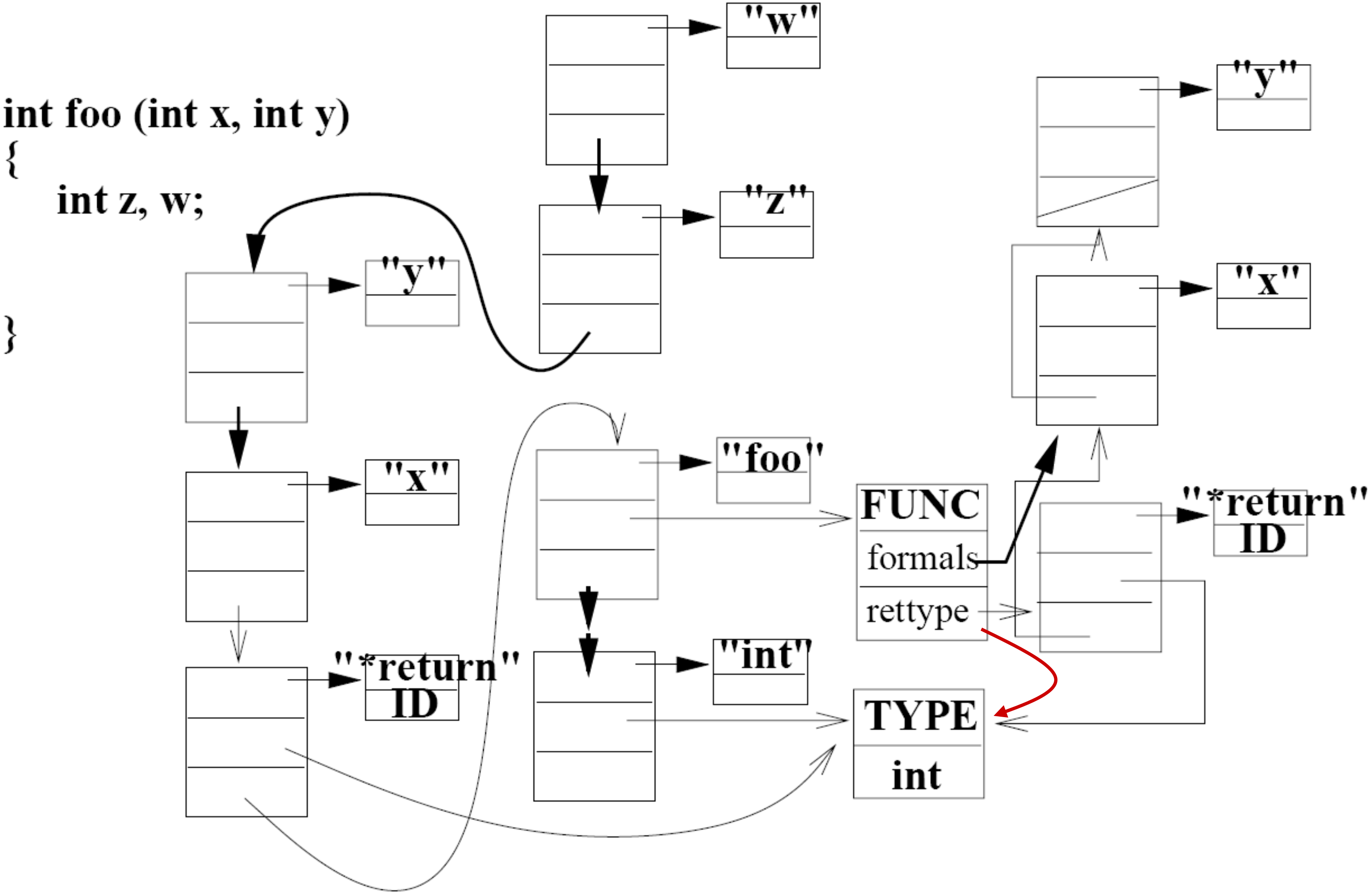


Examples: Function Declarations

```
func_decl: opt_type ID “(”
          {
               struct decl *procdecl = makeprocdecl();
               declare($2, procdecl);
               pushscope(); /* for collecting formals */
               declare(returnid, $1);
               $<declptr>$ = procdecl;
          }
          var_list “)”
          {
               struct ste *formals;
               struct decl *procdecl = $<declptr>4;
               formals = popscope();
               /* popscope reverses stes (first one is the returnid) */
               procdecl→returntype = formals→decl;
```

```
          procdecl→formals = formals→prev;
          pushscope() /* for installing formals & locals in this scope */
          pushtelist(formals);
        }
        compound_stmts
        {
          popscope();
        }
opt_type: type_id             {$$ = $1;}
        | /* empty */         {$$ = voidtype; }
        ;
```

- For the type checking of return types within the function, we declare a fake ID *return in the symbol table and when we parse **return** *expr* ; we compare the current declaration of *expr* to the return type which can be get via findcurrentdecl(returnid)



```
int foo (int x, int y)
{
    int z, w;


}
```

- stmt: RETURN expr; {checksametype(findcurrentdecl(returnid), $2);}

Some Type Checking Examples

```
unary           : INTCONST    {$$ = makenumconstdecl(inttype, $1);}
                | ID          {$$ = findcurrentdecl($1);}
                | unary “.” ID {$$ = structaccess($1, $3);}
                | unary “[” expr “]” {$$ = arrayaccess($1, $3);}
                ;
binary          | unary             {$$ = $1→type;}
                | binary ‘+’ binary {$$ = plustype($1, $3);}
                ;
expr            : binary
                ;
assignment      | unary “=“ expr {check_isvar($1);
                                  check_compatible($1, $3);
                                  $$ = $1 →type;}
                ;
```

- When **unary** becomes **binary**, we take type information and propagate it

Array and Structure Accesses

```
struct decl *arrayaccess (struct decl *arrayptr, struct decl *indexptr) {
    struct decl *arraytype = arrayptr→type;
    check_isarray(arraytype);
    check_sametype(inttype, indexptr);
    return (arraytype→elementvar);
};

struct decl *structaccess (struct decl *structptr, struct id *fieldid) {
    struct decl *typeptr = structptr →type;
    check_isstruct(typeptr);
    return (finddecl(fieldid, typeptr →fields));
}
```

Example: Function Calls

```
unary       : unary “(” args “)”
              { checkisproc($1);
                $$ = checkfunctioncall($1, $3); }

args        : expr “,” args { $1→next = $3; $$ = $1; }
            | expr            { $$ = $1; }
            ;

struct decl *plustype(struct decl type1, struct decl type2)
{
    struct decl *type_after;
    type_after = check_compatible_type(type1, type2);
    return (type_after);
}
```

```
struct decl * checkfunctioncall(struct decl *procptr, struct decl *actuals)
{
    struct ste *formals = procptr→formals;
    /* 1. compare number of formals and actuals */

    /* 2. check for type match                   */
    while(formals != NULL && actuals != NULL) {
        checkisvar(formals →decl);
        check_compatible(formals →decl, actuals);
        formals = formals →prev;
        actuals = actuals →next;
    }
    return (procptr →returntype); /* for decl of the call */
}
```

- Above method of argument checking does not work for actuals

Type Theory: Type Equivalence

- Two Type Equivalence: Structural & Name Equivalence
    - Structural Equivalence: Same Type Expression
    - Name Equivalence: Same Type Name
    - Ex: `struct s1 { int a;};` and `struct s2 { int a; };` : structurally-equivalent but not name-equivalent
- In C, with exceptions of `structs` and `unions`, structural equivalence holds. So, comparing pointers to `struct decls` is not enough to decide type equivalence but helps to determine it quickly if they are equal

Type Compatibility

- Operand Compatibility
  - What combinations of operators and operands are allowed by the language
- Assignment Compatibility
  - Check the correctness of assignment
  - Function calls: the formals must be assignment compatible with actuals

Type Determination

- Simple Model: Type of an expression depends on its operands
  Ex: int + int → int
  - Literals (numbers or strings):
    - Lexical type determines its type
  - ID: type depends on its declaration
  - Compound expression: function of operator and operands
  - Type conversion
- Type coercion: Implicit Type Conversion that takes place during assignment or when evaluating an expression