



Unit 10

Semantic Analysis



Summary

- **Semantics of programming languages**
- **Symbol tables**
 - Scope Rules and Symbol Tables
 - Translation Schemes for building Symbol Tables
- **Type checking**
 - Type system
 - Specifying a Type Checker
 - Type Conversion



Semantic analysis

- Extract types and other information from the program
- Check language rules that go beyond the grammar
- Assign storage locations
- Construct symbol tables – For each identifier in the program, record its attributes (kind, type, etc.)



Semantic analysis

■ Find errors after parsing

- The type of the right-side expression of an assignment statement should match the type of the left-side, and the left-side needs to be a properly declared and assignable identifier (i.e. not some sort of constant).
- The parameters of a function should match the arguments of a function call in both number and type.
- The language may require that identifiers are unique, disallowing a global variable and function of the same name.
- The operands to multiplication operation will need to be of numeric type, perhaps even the exact same type depending on the strictness of the language

■ Parse trees are used in semantic analysis



Type checking

- A **compiler** must check that the program follows the *Type Rules* of the language.
- Information about *Data Types* is maintained and computed by the **compiler**.
- The *Type Checker* is a module of a **compiler** devoted to type checking tasks.



Examples of Tasks

- The operator modulus (%) is defined only if the operands are integers;
- Indexing is allowed only on an array and the index must be an integer;
- A function must have a precise number of arguments and the parameters must have a correct type;



Type checking

- Type Checking may be either *static* or *dynamic*.
- The one done at compile time is static.
- In languages like Pascal and C type checking is primarily static and is used to check the correctness of a program before its execution.
- Static type checking is also useful to determine the amount of memory needed to store variables.
- The design of a Type Checker depends on the syntactic structure of language constructs, the *Type Expressions* of the language, and the rules for assigning types to constructs.



Type Expressions

- A **Type Expression** denotes the type of a language construct.
- A type expression is either a *Basic Type* or is built applying *Type Constructors* to other types.
- A *Basic Type* is a type expression (int, real, boolean, char). The basic type void represents the empty set and allows statements to be checked;
- Type expressions can be associated with a name: *Type Names* are type expressions;
- A *Type Constructor* applied to type expressions is a type expression.



Type expressions (con'd)

- *Array*. If T is a type expression, then $array(I, T)$ is a type expression denoting an array with elements of type T and index range in I (e.g., $array[1..10]$ of $int == array(1..10, int)$)
- *Cartesian Product* If T_1 and T_2 are type expressions, then their Cartesian Product $T_1 \times T_2$ is a type expression;
- *Record*. Similar to Product but with names for different fields (used to access the components of a record).

Example of a C record type

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



Type expressions (con'd)

- *Pointer*. If T is a type expression, then $\text{pointer}(T)$ is the type expression “pointer to an object of type T ”
- *Function*. If D is the domain and R the range of the function then we denote its type by the type expression: $D : R$.
- **Example**
The Pascal function:
function $f(a, b: \text{char}): \text{integer}$
has type: $\text{char} \times \text{char} : \text{int}$.



Type System

- **Type System:** Collection of rules for assigning type expressions to the various part of a program.
- Type Systems are specified using syntax directed definitions.
- A *type checker* implements a type system.
- A language is *strongly typed* if its **compiler** can guarantee that the program it accepts will execute without type errors.



Specification of a Type Checker

- We specify a type checker for a simple language where identifiers have an associated type.
- Attribute Grammar for Declarations and Expressions:

$P \rightarrow D;E$

$D \rightarrow D;D \mid \text{id} : T$

$T \rightarrow \text{char} \mid \text{int} \mid \text{array}[\text{num}] \text{ of } T \mid \uparrow T$

$E \rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E[E] \mid E\uparrow$



Syntax-Directed Translation

- Technique used to build semantic information for large structures, based on its syntax.
- In a compiler, Syntax-Directed Translation is used for
 - Constructing Abstract Syntax Tree
 - Type checking
 - Intermediate code generation
- The semantics (i.e., meaning) of the various constructs in the language is viewed as attributes of the corresponding grammar symbols.
- Attributes are associated with Terminal as well as Nonterminal symbols



Attributes

- **Synthesized attribute:** An **attribute** that gets its values from the attributes attached to the children of its nonterminal.
 - Value flows from child to parent in the parse tree.
 - Example: val in Expression grammar
- **Inherited attribute:** An **attribute** that gets its values from the attributes attached to the parent (or siblings) of its nonterminal.
 - Value flows into a node in the parse tree from parents and/or siblings.
 - Example: type of VarList in declaration grammar



Syntax Directed Definition

- **Syntax Directed Definitions** are a generalization of context-free grammars in which:
 1. Grammar symbols have an associated set of **Attributes**;
 2. Productions are associated with **Semantic Rules** for computing the values of attributes.
- Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).



Forms of a syntax directed definition

- A grammar production $Y \rightarrow X_1 \dots X_n$ may have zero or more associated semantic rules. Each semantic rule has the form, $b = f(c_1, \dots, c_k)$ are attributes of the grammar symbol in the production such that:
 1. b is a **synthesized attribute** and c_1, \dots, c_k are the attributes of the grammar symbols on the rhs or
 2. b is an **inherited attribute** of one of the RHS grammar symbols and c_1, \dots, c_k are any other attributes in the production.
- If the function f does not have side effects, syntax directed definitions is also called as attribute grammars.



Example

Production	Semantic rules
$L \rightarrow E \text{ return}$	Print (E.val)
$E \rightarrow E_1 + T$	$E.\text{val} = E1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit.Lexval}$

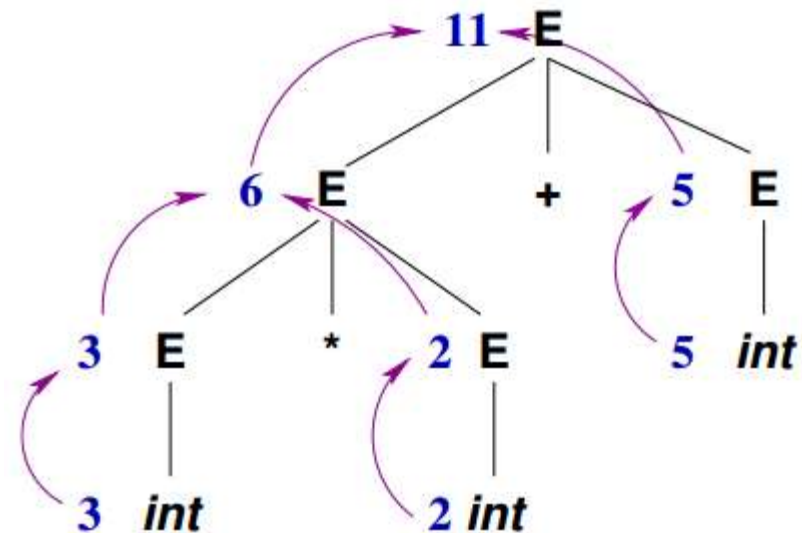
Consider the Grammar for arithmetic expressions above. The **Syntax Directed Definition** associates to each non terminal a synthesized attribute called **val**.

Synthesized Attribute Example

E	\longrightarrow	$E * E$
E	\longrightarrow	$E + E$
E	\longrightarrow	int

$E \longrightarrow E_1 * E_2 \quad \{E.val := E_1.val * E_2.val\}$
 $E \longrightarrow E_1 + E_2 \quad \{E.val := E_1.val + E_2.val\}$
 $E \longrightarrow \text{int} \quad \{E.val := \text{int}.val\}$

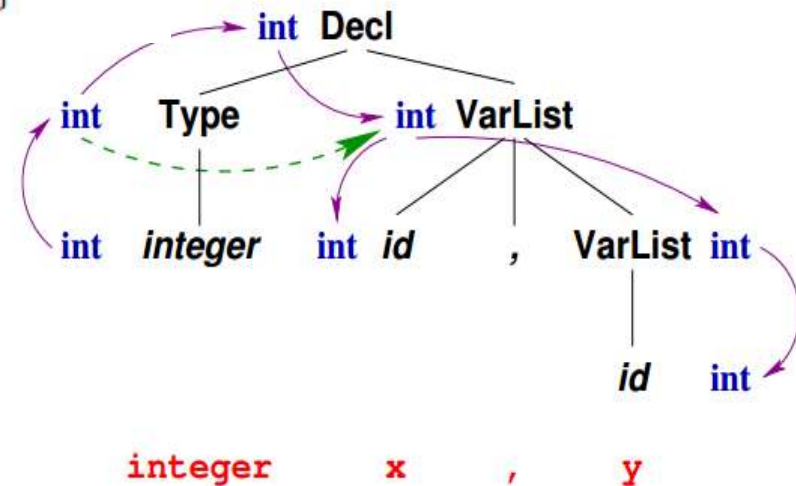
Information Flow for $3*2+5$



Inherited Attribute Example

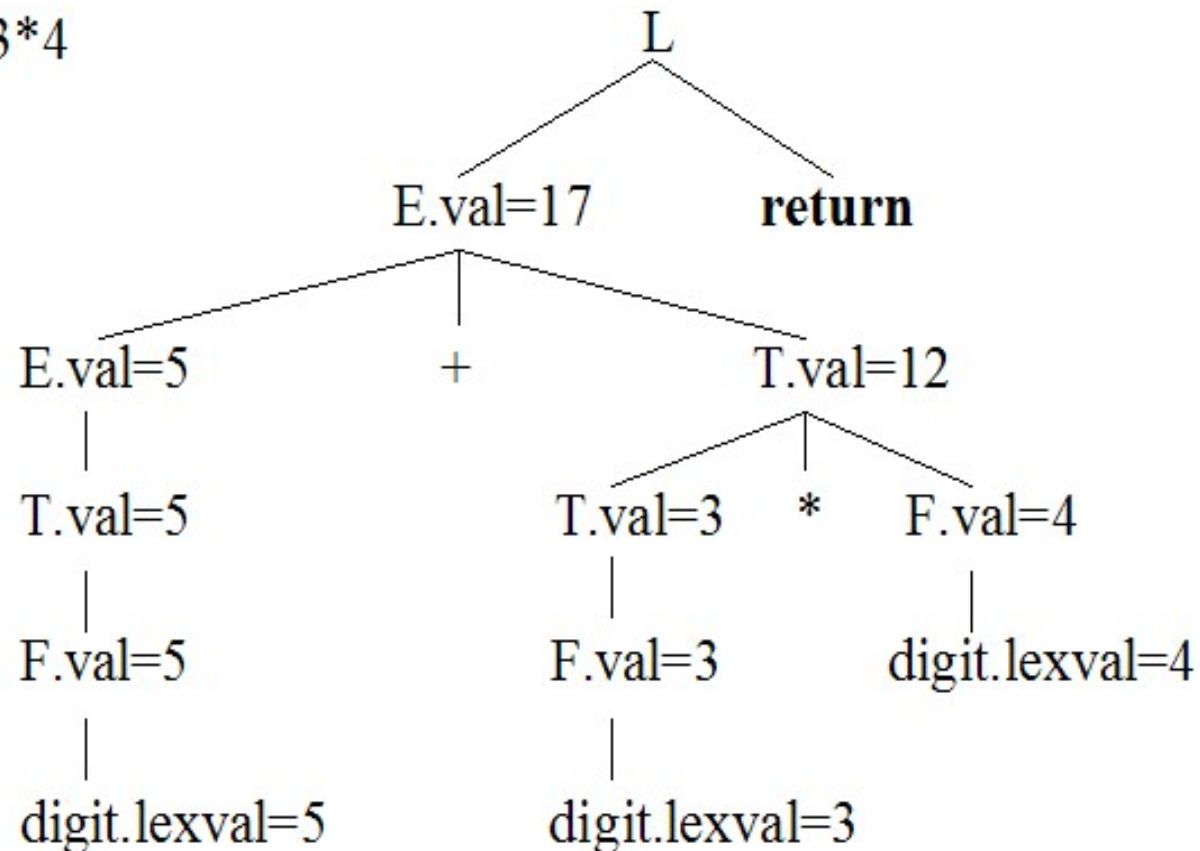
<i>Decl</i>	→	<i>Type</i> <i>VarList</i>
<i>Type</i>	→	<i>integer</i>
<i>Type</i>	→	<i>float</i>
<i>VarList</i>	→	<i>id</i> , <i>VarList</i>
<i>VarList</i>	→	<i>id</i>

<i>Decl</i>	→	<i>Type</i> <i>VarList</i>	{ <i>VarList.type</i> := <i>Type.type</i> }
<i>Type</i>	→	<i>integer</i>	{ <i>Type.type</i> := <i>int</i> }
<i>Type</i>	→	<i>float</i>	{ <i>Type.type</i> := <i>float</i> }
<i>VarList</i>	→	<i>id</i> , <i>VarList</i> ₁	{ <i>VarList</i> ₁ . <i>type</i> := <i>VarList.type</i> ; <i>id.type</i> := <i>VarList.type</i> }
<i>VarList</i>	→	<i>id</i>	{ <i>id.type</i> := <i>VarList.type</i> }



Example of annotated syntax tree

Input: $5+3*4$





Type checker of identifiers

PRODUCTIONS	SEMANTIC RULES
$D \rightarrow \text{id} : T$	$\text{addtype}(\text{id.entry}, T.\text{type})$
$T \rightarrow \text{char}$	$T.\text{type} := \text{char}$
$T \rightarrow \text{int}$	$T.\text{type} := \text{int}$
$T \rightarrow \uparrow T_1$	$T.\text{type} := \text{pointer}(T_1.\text{type})$
$T \rightarrow \text{array}[\text{num}] \text{ of } T_1$	$T.\text{type} := \text{array}(1..\text{num.val}, T_1.\text{type})$



Type checker of expressions

PRODUCTIONS	SEMANTIC RULES
$E \rightarrow \text{literal}$	$E.type := \text{char}$
$E \rightarrow \text{num}$	$E.type := \text{int}$
$E \rightarrow \text{id}$	$E.type := \text{lookup}(\text{id.entry})$
$E \rightarrow E_1 \text{ mod } E_2$	$E.type := \text{if } E_1.type = \text{int} \text{ and } E_2.type = \text{int}$ then int else type_error
$E \rightarrow E_1[E_2]$	$E.type := \text{if } E_2.type = \text{int} \text{ and } E_1.type = \text{array}(s,t)$ then t else type_error
$E \rightarrow E_1 \uparrow$	$E.type := \text{if } E_1.type = \text{pointer}(t) \text{ then } t$ else type_error



Type checker of statements

PRODUCTIONS	SEMANTIC RULES
$S \rightarrow \text{id} := E$	$S.type := \text{if } \text{id}.type = E.type \text{ then } void$ $\text{else } type_error$
$S \rightarrow \text{if } E \text{ then } S_1$	$S.type := \text{if } E.type = boolean \text{ then } S_1.type$ $\text{else } type_error$
$S \rightarrow \text{while } E \text{ do } S_1$	$S.type := \text{if } E.type = boolean \text{ then } S_1.type$ $\text{else } type_error$
$S \rightarrow S_1; S_2$	$S.type := \text{if } S_1.type = void \text{ and } S_2.type = void$ $\text{then } void$ $\text{else } type_error$



Type checker of functions

PRODUCTIONS	SEMANTIC RULES
$D \rightarrow \text{id} : T$	$\text{addtype}(\text{id.entry}, T.\text{type}); D.\text{type} := T.\text{type}$
$D \rightarrow D_1; D_2$	$D.\text{type} := D_1.\text{type} \times D_2.\text{type}$
$\text{Fun} \rightarrow \text{fun id}(D) : T; B$	$\text{addtype}(\text{id.entry}, D.\text{type} : T.\text{type})$
$B \rightarrow \{S\}$	
$S \rightarrow \text{id}(E\text{List})$	$E.\text{type} := \text{if } \text{lookup}(\text{id.entry}) = t_1 : t_2 \text{ and } E\text{List.type} = t_1$ then t_2 else type_error
$E\text{List} \rightarrow E$	$E\text{List.type} := E.\text{type}$
$E\text{List} \rightarrow E\text{List}, E$	$E\text{List.type} := E\text{List}_1.\text{type} \times E.\text{type}$

Function for checking equivalence of types

```
function sequiv(s, t): boolean;
begin
    if s and t have the same basic type then
        return true;
    else if s = array(s1, s2) and t = array(t1, t2) then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = s1 x s2 and t = t1 x t2 then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = pointer(s1) and t = pointer(t1) then
        return sequiv(s1, t1)
    else if s = s1 → s2 and t = t1 → t2 then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else
        return false;
end;
```



Type Conversion

- What's the type of “ $x + i$ ” if:
 1. x is of type real;
 2. i is of type int;
 3. Different machine instructions are used for operations on reals and integers.
- Depending on the language, specific conversion rules must be adopted by the **compiler** to convert the type of one of the operand of $+$.
- The type checker in a **compiler** can insert these conversion operators into the intermediate code.
- Such an implicit type conversion is called *Coercion*.



Type coercion rules

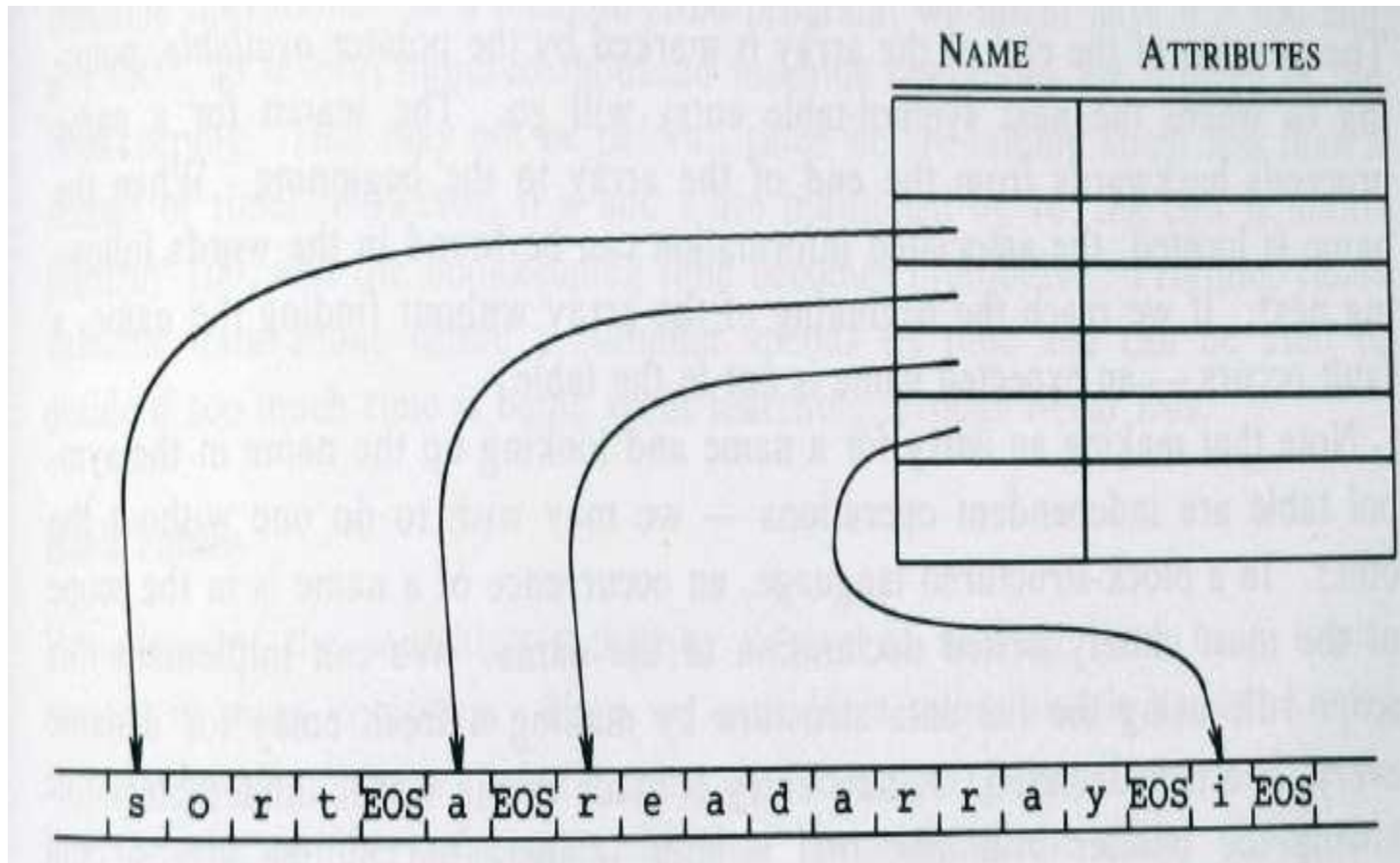
PRODUCTIONS	SEMANTIC RULES
$E \rightarrow \text{num}$	$E.type := int$
$E \rightarrow \text{num.num}$	$E.type := real$
$E \rightarrow \text{id}$	$E.type := \text{lookup}(\text{id.entry})$
$E \rightarrow E_1 \text{ op } E_2$	$E.type :=$ if $E_1.type = int$ and $E_2.type = int$ then int else if $E_1.type = int$ and $E_2.type = real$ then $real$ else if $E_1.type = real$ and $E_2.type = int$ then $real$ else if $E_1.type = real$ and $E_2.type = real$ then $real$ else $type_error$



Symbol Table

- The **Symbol Table** is the major inherited attribute and the major data structure as well.
- Symbol Tables store information about the name, type, scope and allocation size.
- Symbol Table must maintain efficiency against insertion and lookup.
- Dynamic data structures must be used to implement a symbol table: Linear Lists and Hash Tables are the most used.
- Each entry has the form of a record with a field for each piece of information.

Storing names





Symbol Tables and Scope Rules

- A **Block** in a programming language is any set of language constructs that can contain declarations.
- A language is **Block Structured** if
 1. Blocks can be *nested* inside other blocks, and
 2. The *Scope* of declarations in a block is limited to that block and the blocks contained in that block.
- **Most Closely Nested Rule.** Given several different declarations for the same identifier, the declaration that applies is the one in the most closely nested block.



Symbol Tables and Scope Rules (con'd)

- To implement symbol tables complying with nested scopes
 1. The *insert* operation into the symbol table must not overwrite previous declarations;
 2. The *lookup* operation into the symbol table must always refer to the most close block rule;
 3. The *delete* operation must delete only the most recent declarations.
- The symbol table behaves in a stack-like manner.

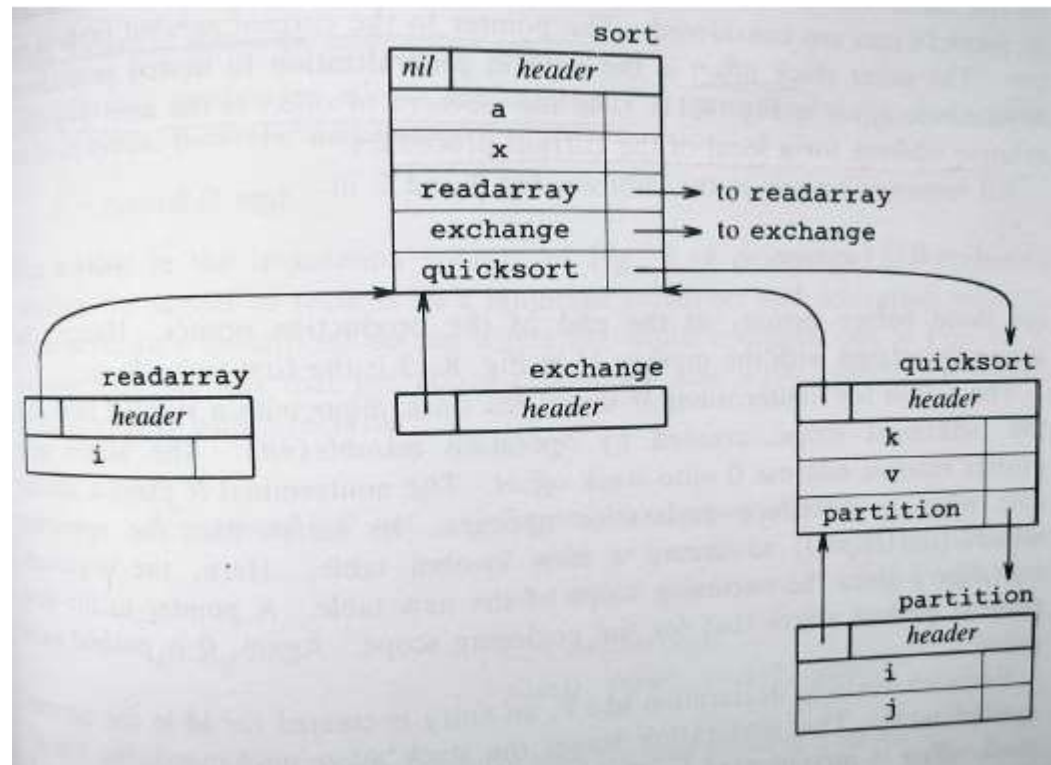


Procedure Activations: Example

```
program sort;  
  var a : array[0..10] of integer;  
      x : integer;  
  procedure readarray;  
    var i : integer;  
    begin ... a ... end { readarray };  
  procedure exchange( i:integer; j : integer );  
    begin  
      x := a[i]; a[i] := a[j]; a[j] := x  
    end { exchange };  
  procedure quicksort( m: integer; n: integer );  
    var k, v : integer;  
  function partition( y:integer; z: integer ): integer;  
    var i, j : integer;  
    begin... a ...  
      ... v ...  
      ... exchange( i, j ); ...  
    end { partition };  
  begin ... end { quicksort };  
begin ... end { sort };
```


Maintain separate symbol tables for each scope

- Tables must be linked both from inner to outer scope, and from outer to inner





Lexical- Vs. Syntactic-Time Construction

- Information is first entered into a symbol table by the lexical analyzer only if the programming language does not allow for different declarations for the same identifier (scope).
- If scoping is allowed, the lexical analyzer will only return the name of the identifier together with the token
- The identifier is inserted into the symbol table when the syntactic role played by the identifier is discovered.



Relative Address

- **Relative Address.** Is a storage allocation information consisting of an offset from a base (usually zero) address: The Loader will be responsible for the run-time storage.

Computes relative address

Use a global variable called *offset*.

$$P \rightarrow \{\text{offset} := 0\} D$$
$$D \rightarrow D; D$$
$$D \rightarrow \text{id} : T \quad \{ \text{enter}(\text{id.name}, T.\text{type}, \text{offset}); \\ \text{offset} := \text{offset} + T.\text{width} \}$$
$$T \rightarrow \text{int} \quad \{ T.\text{type} := \text{int}; T.\text{width} := 4 \}$$
$$T \rightarrow \text{real} \quad \{ T.\text{type} := \text{real}; T.\text{width} := 8 \}$$
$$T \rightarrow \text{array}[\text{num}] \text{ of } T_1 \quad \{ T.\text{type} := \text{array}(\text{num.val}, T_1.\text{type}); \\ T.\text{width} := \text{num.val} * T_1.\text{width} \}$$
$$T \rightarrow \uparrow T_1 \quad \{ T.\text{type} := \text{pointer}(T_1.\text{type}); T.\text{width} := 4 \}$$



Computes relative address

- The global variable *offset* keeps track of the next available address.
- Before the first declaration, *offset* is set to 0
- As each new identifier is seen it is entered in the symbol table and *offset* is incremented.
- *type* and *width* are synthesized attributes for non-terminal T.



Keeping Track of Scope Information

- Consider *Nested Procedures*: When a nested procedure is seen processing of declarations in the enclosing procedure is suspended.
- To keep track of nesting a stack is maintained.
- We associate a new symbol table for each procedure:
- When we need to *enter* a new identifier into a symbol table we need to specify which symbol table to use.

Processing declarations in nested procedures

$P \rightarrow \mathbf{M} D$	$\{addwidth(top(tblptr), top(offset));$ $pop(tblptr); pop(offset)\}$
$\mathbf{M} \rightarrow \epsilon$	$\{t := mktable(nil);$ $push(t, tblptr); push(0, offset)\}$
$D \rightarrow D; D$	
$D \rightarrow \text{proc } id; \mathbf{N} D_1; S$	$\{t := top(tblptr); addwidth(t, top(offset));$ $pop(tblptr); pop(offset);$ $enterproc(top(tblptr), id.name, t)\}$
$D \rightarrow id : T$	$\{enter(top(tblptr), id.name, T.type, top(offset));$ $top(offset) := top(offset) + T.width\}$
$\mathbf{N} \rightarrow \epsilon$	$\{t := mktable(top(tblptr));$ $push(t, tblptr); push(0, offset)\}$



Keeping Track of Scope Information

- The semantic rules make use of the following procedures and stack variables:
 1. *mktable(previous)* creates a new symbol table and returns its pointer. The argument *previous* is the pointer to the enclosing procedure.
 2. The stack *tblptr* holds pointers to symbol tables of the enclosing procedures.
 3. The stack *offset* keeps track of the relative address w.r.t. a given nesting level.
 4. *enter(table,name,type,offset)* creates a new entry for the identifier *name* in the symbol table pointed to by *table*, specifying its *type* and *offset*.
 5. *addwidth(table,width)* records the cumulative *width* of all the entries in *table* in the header of the symbol table.
 6. *enterproc(table,name,newtable)* creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.