Unit 10
Semantic Analysis

Summary

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

Semantics

- Find errors after parsing
  - Type checking
  - Check the correspondence between the use of variables, functions . . . and their declarations
  - Scope of variables
- 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 mod 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 expresions (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 T1× T2 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 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 wedenote its type by the type expression: D : R.
- Example

  The Pascal function:

  function f(a, b): integer

  has type: *char* × *char : int*.

Type System

- **Type System:** Collection of rules for assigning type expressions to thevarious 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 → D;E

  D → D;D | id : T

  T → char | int | array[num] of T | ↑T

  E → literal | num | id | E mod E | E[E] | E↑

Attributes

- **Attribute grammar** is a formal way to define attributes for the productions of a formal grammar, associating these attributes to values.

- **Synthesized attribute**: An **attribute** that gets its values from the attributes attached to the children of its nonterminal
- **Inherited attribute**: An **attribute** that gets its values from the attributes attached to the parent (or siblings) of its nonterminal.

Syntax Directed Definition

- **Syntax Directed Definitions** are a generalization of context-free grammarsin 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

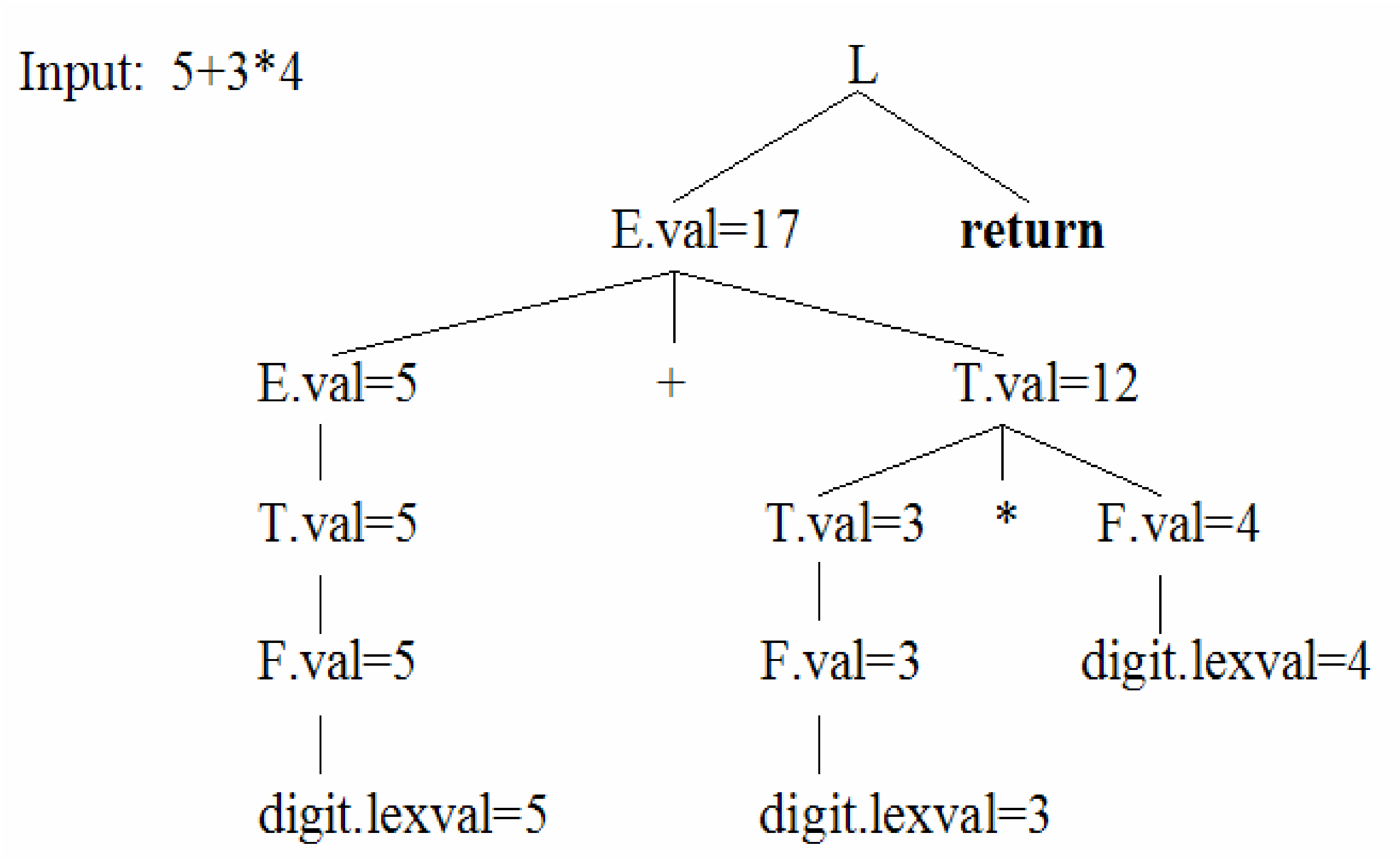
- **A grammar production Y --> X_1 … X_n may have zero or more associated semantic rules. Each semantic rules has the form, b=f(c_1,…,c_k) are attributes of the grammar symbol in the production such that:**
  - **1. b is a synthesized attribute and c_1,…,c_k are the attributes of the grammar symbols on the rhs or**
  - **2. b is an inherited attribute of one of the RHS grammar symbols and c_1,…,c_k are any other attributes in the production.**
-

Example

Production	Semantic rules
L → E **return**	Print (E.val)
E → E**1**+T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T**1** * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = digit.Lexval

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

Examlpe of annotated syntax tree



Type checker of identifiers

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

Type checker of expressions

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

Type checker of statements

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

Type checker of functions

PRODUCTIONS	SEMANTIC RULES
$D \rightarrow \mathsf{id} : T$	$addtype(\mathsf{id}.entry, T.type);\ D.type := T.type$
$D \rightarrow D_1 ; D_2$	$D.type := D_1.type \times D_2.type$
$Fun \rightarrow \mathsf{fun}\ \ \mathsf{id}(D) : T ; B$	$addtype(\mathsf{id}.entry, D.type{:}T.type)$
$B \rightarrow \{S\}$	
$S \rightarrow \mathsf{id}(EList)$	$E.type :=$ if $lookup(\mathsf{id}.entry){=}t_1 : t_2$ and $EList.type{=}\ t_1$ then t_2 else $type\_error$
$EList \rightarrow E$	$EList.type := E.type$
$EList \rightarrow EList, E$	$EList.type := EList_1.type \times E.type$

Function for checking equivalence of types

```
function sequiv(s, t): boolean;
begin
   if s và t là cùng kiểu dữ liệu chuẩn 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*.

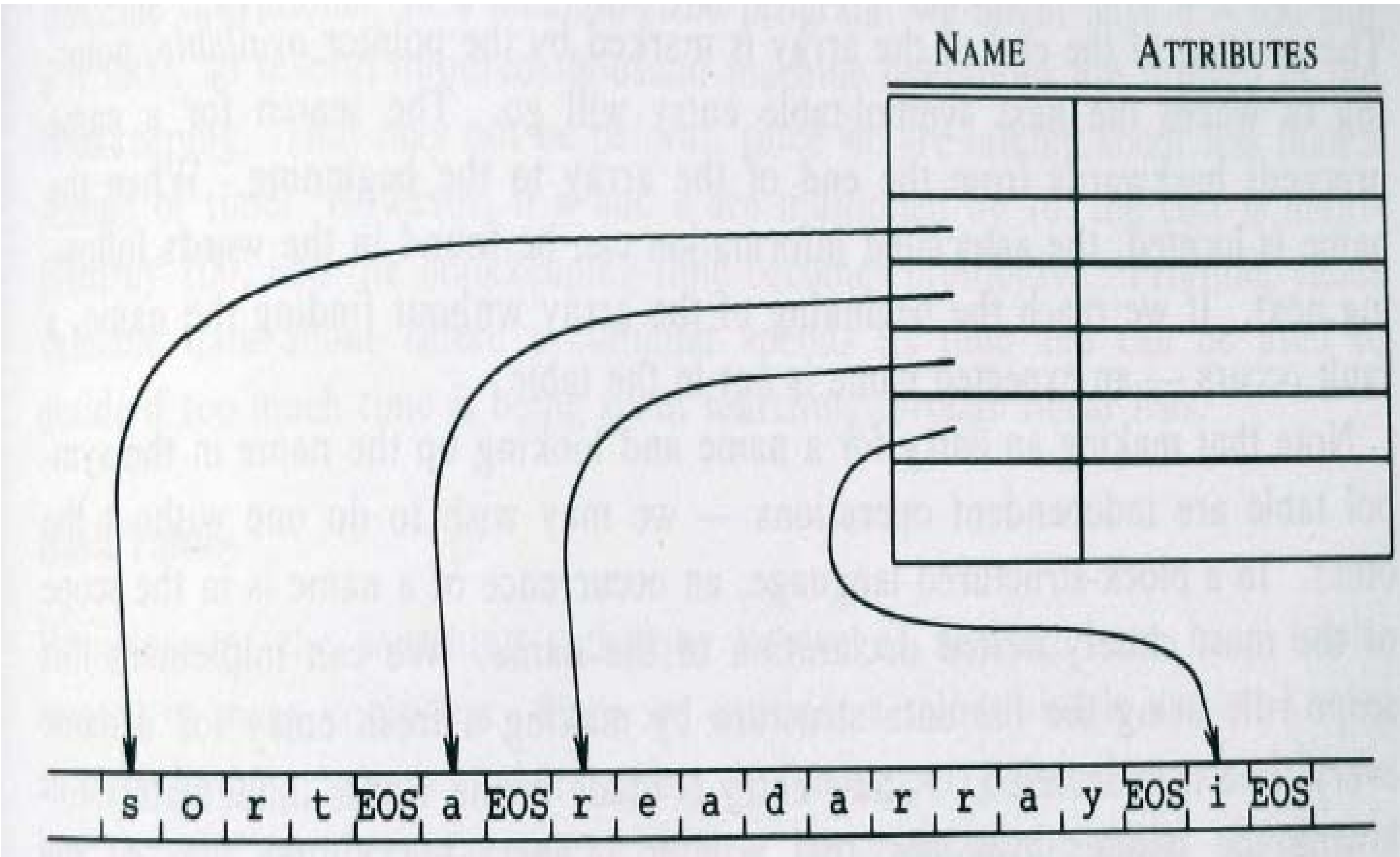
Type coercion rules

PRODUCTIONS	SEMANTIC RULES
$E \to \text{num}$	$E.type := int$
$E \to \text{num.num}$	$E.type := real$
$E \to \text{id}$	$E.type := lookup(\text{id}.entry)$
$E \to E_1 \text{ op } E_2$	$E.type :=$ 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 peace of information.

Storing names



Symbol Tables and Scope Rules

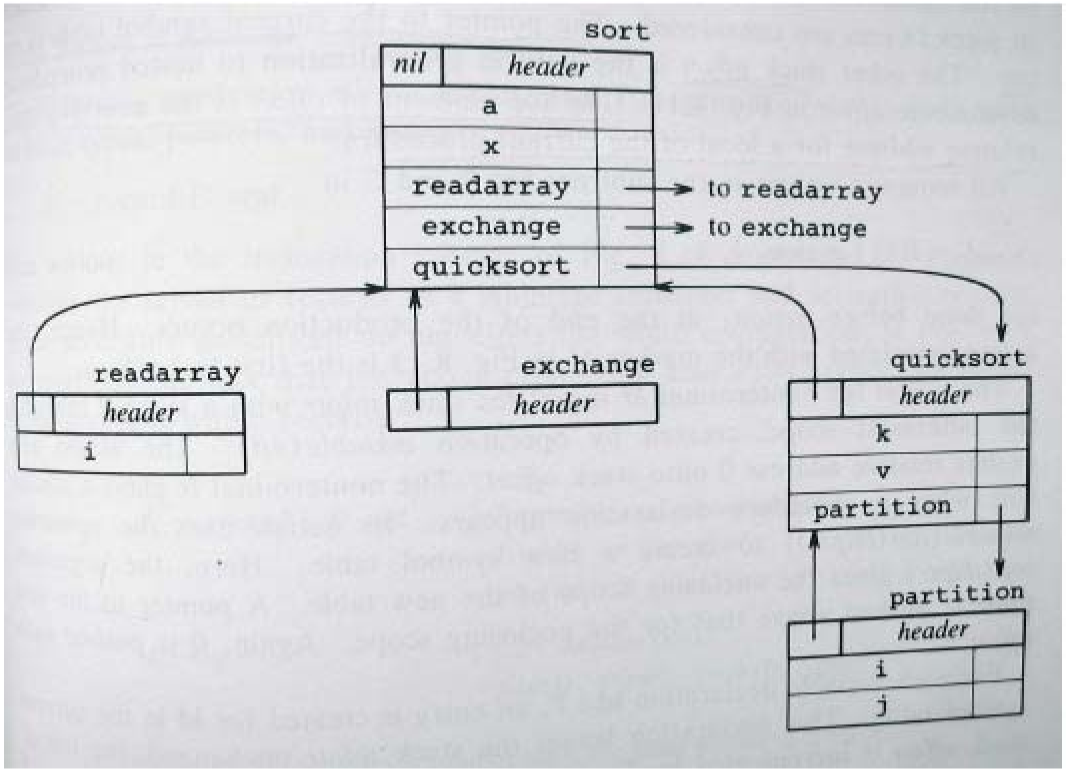
- A **Block** in a programming language is any set of language constructs that can contain declarations.
- A language is **Block Structured** if
  1. Blocks can be *nested* inside other blocks, and
  2. The *Scope* of declarations in a block is limited to that block and the blocks contained in that block.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

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*.

$P \rightarrow \{\text{offset} := 0\}\ D$

$D \rightarrow D; D$

$D \rightarrow \text{id} : T \qquad \{enter(\text{id}.name,\ T.type,\ offset);$
$\quad \textbf{offset := offset + T.width}\}$

$T \rightarrow \text{int} \qquad \{T.type := int;\ T.width := 4\}$

$T \rightarrow \text{real} \qquad \{T.type := real;\ T.width := 8\}$

$T \rightarrow \text{array}[\text{num}] \text{ of } T_1 \qquad \{T.type := array(\text{num}.val, T_1.type);$
$\quad T.width := \text{num}.val * T_1.width\}$

$T \rightarrow \uparrow T_1 \qquad \{T.type := pointer(T_1.type);\ T.width := 4\}$

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

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

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* inthe symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.