## Syntax Directed Semantic Analysis - Introduction

- Use the syntactic representation of language to drive semantic analysis.
- Approach:
  - Define a set of **attributes** of nonterminals of program
  - Define a set of semantic **equations** that determine how attributes can be evaluated
  - Define **order** in which equations should be evaluated
  - Construct a parse tree that captures the syntactic structure of a program.
  - Traverse the tree in the order of evaluation of attributes; use equations to compute attributes.

  Note that parse tree need not be constructed explicitly.
- What are attributes?
- What are semantic equations?
- What is order of evaluation? Is it specified implicitly or explicitly?
- Are we going to do semantic analysis and code generation using this technique?
  Yes
- Does YACC have any such notion?
  Yes.

# Organization of Lectures

- Symbol tables:
  - What do symbol tables store?
  - How can they be organized?
  - How do we gather this information from context free grammar?
- Semantic analysis:
  - What kinds of properties do nonterminals have?
  - How do we evaluate them from cfg?
  - Evaluation of values, types, etc.
- Project part 3:
  - How to organize your class definitions?
  - How to build symbol table?
  - How to build parse tree?
  - How to perform checkings?

## Symbol Table

- A mechanism for associating values (or attributes) with names of programs.
  Example: associate variable name with its type attribute.
- How is it used?
  - Use declarations to add name/attribute pair in the table.
  - Every time a name is used, access its attributes through symbol table.
- A symbol table is an integral part of semantic processing phase.
- A symbol table data structure will provide support for storing, removing, and searching names in a table.
- Implementation mechanisms for symbol tables:
  - Unordered list
  - Ordered list
  - Binary search trees
  - Hash tables (most common means of implementing symbol tables).

  You could use `Map` or `Multimap` facility of STL for implementing it.

## Block Structured Symbol Tables

- Name scope: Program text enclosed by a program unit such as procedure, block, subprogram, class or package.
- Block-structured languages: Languages that allow nested name scopes. (That is, program units can be defined within other program units.)
- A line of text in a program may therefore occur within a set of name scopes. For this text:
  - *Current scope*: The name scope defined by *innermost* such unit
  - *Open scope*: Name scopes defined by current scope and by enclosing program units.
  - *Closed scope*: Name scopes that do not enclose a given text.
- *Visibility rules*: define what names mean when there are multiple scopes.
  1. Only names declared in current scope and open scopes containing current scope are accessible.
     All visible names can be determined statically.
  2. If a name is declared in more than one open scope, *innermost* declaration (the one nearest the reference), is used to interpret a reference to that name.
  3. New declarations can be made only in current scope.

  $\Rightarrow$ All names within a scope become inaccessible once the scope is closed.

## Implementation of symbol tables for block-structured languages

- Note that a name can be defined many times in different scopes. A single symbol table that stores (name, attribute) pair will not work.
- Approach: Construct a symbol table for each scope.
  Scope stack: since name scopes are opened and closed in a last-in, first-out manner, a stack is an appropriate mechanism for organizing symbol tables as one moves from one name scope to another.
- At a given location in text, scope stack contains one entry for each open name scope.
- Manipulation of stack:
  1. Every time one enters a scope, create a symbol table for the scope and push it on the stack.
  2. Every time one exits a scope, pop off top symbol table stack. For one pass compilers this symbol table can be destroyed. However, for multiple pass compiler, it must be stored somewhere so that it can be used to find names in later passes.
- Searching a name: Search top symbol table first, then second from top, and so on until name is found.

## Symbol Table

- Disadvantage of using a scope stack:
  - May need to search for a name in a number of different symbol tables before it is found. What about global variables?
  - Overhead of having many symbol tables, especially when hash tables are used. For hash tables, one must allocate blocks of space. If blocks are large, much storage space is wasted since many scopes include definitions of few names. However, if tables are small, search for names be slow due to long hash chains.
- **Single symbol table:** All names for all scopes appear in a single table. Each name scope is given a unique *scope number*.
  - A name may appear more than once, except with different scope numbers.
  - Provides slightly faster searching. Also tends to use space more efficiently. However, need to add extra information (scope number).
  - Useful mostly for one-pass compilers. Remove an entry for a name once a scope is closed.
  - For multi-pass compilers, which build the table first and then access it, entries cannot be removed when a scope is closed . The implementation of a single global table gets quite complex as one needs to keep track of all different name scopes and assign them different numbers.

## Extensions to name scoping rules

- Alter the visibility of individual names or set of names.
  Standard rule: definition to reference of a name is the innermost definition found.
  Names that do not obey this rule: names of fields in records, also control scopes of names through export and import kind of constructs.
- Alter search rules: constructs like `with` and `use` bring new search rules
- How to take care of these rules?
- Fields and Records: Fields in different records may have identical names. Most languages require one to specify `Record.FieldName` to clearly identify a name. Two ways to handle:
  1. Allocate a symbol table for each name. This symbol table does not go on scope stack. It is an attribute of record.
     Two levels of search: first find entry for record, and then search field symbol table.
  2. Treat field names like ordinary identifiers and put them into the regular symbol table.

## Extensions to name scoping rules - cont'd.

- Export Rules: Allow a programmer to specify that some names local to a scope are to be made visible outside that scope.
  Used mostly in languages that define modules (such as Ada's package and Modula's module)
  To handle export rules, make sure that when a scope is ended, exported names remain visible, as if they were declared in enclosed scope.
  In Ada, exported objects are not automatically exported. They must be explicitly accessed through package name or `use`. Similar to defining fields in records.
- How to deal with separate compilation?
  When a package specification or definition module is compiled, the compiler saves information about the exported declaration in a *library*. When compiler compiles another package, it consults this library for exported symbols.
  Library therefore makes it possible to do complete compile-time static semantic checking even if separate compilation is used.
  Approach in C: Use header files to make definitions available $\Rightarrow$ build symbol table every time a header file is included. Two problems: i) slower, and ii) definition in header file may not match that of compiled unit.

## Extensions to name scoping rules - cont'd.

- Import Rules Two kinds of scopes: *Importing* and *non-importing*
  Importing scope: Automatically receive access to definitions in containing scope. (As in Pascal or Algol).
  Non-importing scope: Access to some or all nonlocal names must be explicitly requested via an import declaration. (E.g., modules in Modula-2).
  Implementation:
  1. Alter standard search mechanism so that all names or names of certain kinds of objects (variables, typically) cannot be automatically referenced across the boundaries of non-importing scope.
     Just mark which scope is importing and which is non-importing.
- Altered search rules Pascal's `with` statement alters the order in which scopes are examined in order to interpret the meaning of an identifier:

  `with` *R* `do <statement>`

  Here all identifiers are interpreted, if possible as references to a field of record *R*. So scopes are examined in the following order: symbol table associated with *R*, then rest on symbol table stack.

## Extensions to name scoping rules - cont'd.

- Implicit declarations first usage of variable defines it as well. For instance, label on a statement in Algol60, or loop index in Ada.
  Two cases:
  1. name being implicitly declared obeys normal scoping rules. Easy to handle. Just put the name in the symbol table as before.
  2. Name opens a new scope (as in Ada). Construct a new symbol table that contains only a single entry.
     Handling is more complex in Pascal labels because declaration and access rules don't mesh well: there can be only one declaration of a given label in a name scope, but it is not legal to jump to that label from all points in scope of declaration.
- Forward references Refer to an entity that is defined later in scope. (That is, use before declare).
  One example: pointer definitions.
  Handle in the following manner:
  1. Multi-pass: First pass gathers all definitions. Succeeding passes resolve names.
  2. In certain cases, single pass will suffice in that information can be filled in as they are found. However, in cases where immediate steps must be taken, single pass may not be sufficient. For instance, in order to generate code for `A := B + C`, we must know the types of the variables before code can be generated.

## Declaration processing

- Following kinds of declarations:
  - Types: primitives and complex (Class definition)
  - Variables
  - Constants;
  - Function/Procedures;
  - Labels
- Each declaration specifies an identifier and its attributes.
  Note: different kinds of entities have different kinds of attributes:
  - Types: type descriptor for every predefined and newly defined types.
  - Variables: type
  - Procedures: parameters, return types, body

  Your design of the symbol table must take different attribute types into account.
- How can types be represented?
  - Fixed representations for primitive types.
  - Represent each complex type in terms of its components. For instance, each class type defined by a pointer to a symbol table.
- Each variable is thus a tuple (identifier, pointer to type entry).
- Each function by an entry: (identifier, return type, set of parameters, body)
- Arrays: (identifier, element type, index type)

## What kind of information stored in a symbol table?

- Information gleaned during processing of declarations
- Names used for variables, procedures, types, and constants. Each name may have different set of attributes.
- Type names:
  1. Size of type
  2. Scalar type: what is the scalar type?
  3. Subrange: range base type, min and max values
  4. Pointer: base type
  5. Array Type: Index type, Element type
  6. Records: Set of Fields (name, type) pair.
- Variables and constants
  1. Data type
  2. Value
  3. Storage information
- Procedure declarations:
  1. Symbol table
  2. Set of parameters:
  3. body (represented as a parse tree
  4. return type
  5. return value

## Processing Declarations

- Identify scope:

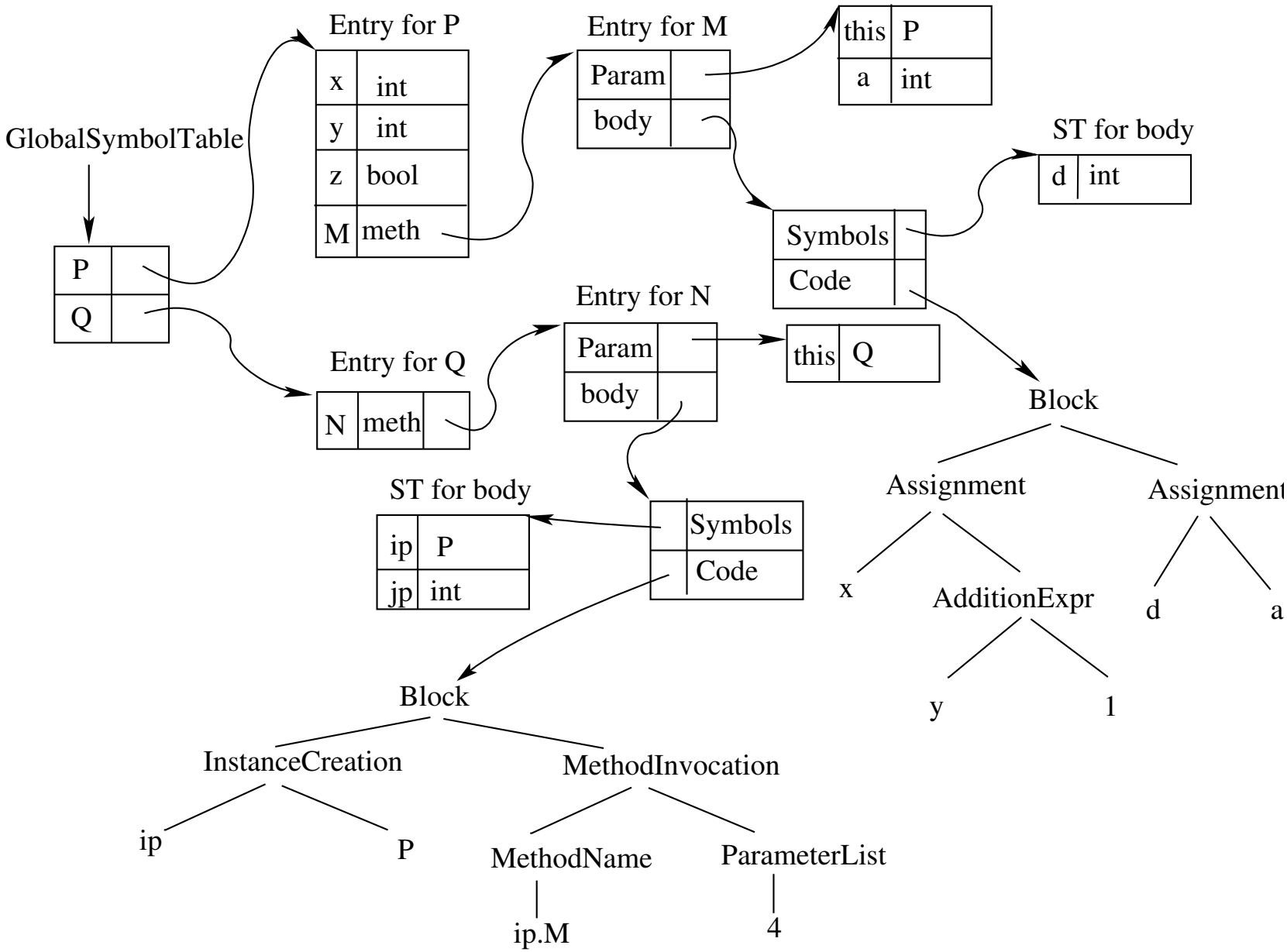
  `Class ::= class ID { ClassBody }`
- Create symbol table when enter a scope + Add symbol table on scope stack
- Any symbols created should be inserted in current symbol table
- When you exit, pop off the symbol table
- Example program:

```
class P {
  int x;
  int y;
  boolean z;
  void M(int a) {
     int d;

     x = y+1;
     d = a;
  }
}
class Q {
  void N() {
    P ip;
    int jp;

    ip = new P;
    ip.M(4);
  }
}
```

# A compact parse tree



## Attributes

- Attribute: a semantic property of an entity of language.
  Attributes give meanings to the strings to languages
- Process of computing an attribute and associating its computed value with the language construct is referred to as **binding** of the attribute.
- Binding of attributes:
  - **Static:** Evaluated during compilation. e.g., type.
  - **Dynamic:** Bound during program execution, e.g., value. So need to generate code for their evaluation.
- Examples of attributes and binding times:
  - **Data type:** type checker will compute data type attributes of all entities of a language. (Can be done statically in statically typed languages such as C, Pascal, Java).
  - **Value:** Usually dynamic and compiler needs to generate code for compute value during execution.
    Constant expressions: can be evaluated at compile time.
  - **Binding of location to variables:**
    - **Static:** relative locations for global variables, local variables etc. determined at compile time
    - **Dynamic:** make arrangements for allocation of dynamic variables in heap
  - **Target code:** Can be generated during compilation process.

## Attribute Grammars

- Definition and evaluation of attributes:
  - Associate attributes with each grammar symbol.
  - For each rule, determine how an attribute can be evaluated from other attributes.
- Attribute equation or semantic rule: given attributes $a_1, \cdots, a_n$, and production $X_0 \rightarrow X_1 X_2 \ldots X_n$:
  $X_i.a_i = f_{ij}(X_0.a_1, \ldots X_0.a_k, X_1.a_1 \ldots X_1.a_k, ....)$
- Attribute grammar: collection of such equations for the attributes.
- Observations: typically, equations quite simple + dependency among only a few attributes.
- Notation: Write attribute grammar in tabular form:

| Production | Attribute Equation |
|---|---|
| Rule 1 | Equation |
| Rule 2 | Equation |
| ⋮ | ⋮ |
| Rule n | Equation |

- Notation: If a NT appears more than once in a rule, associate subscript to distingish NT in semantic rules.
  Example:

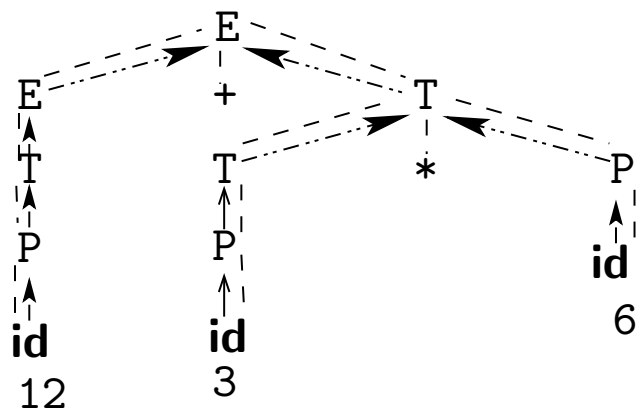
| `Production` | `Attribute Equation` |
|---|---|
| $S ::= aS$ | $S_1$`.count = `$S_2$`.count + 1` |

## Attributes

- Two kinds of attributes: **Synthesized** and **Inherited**.
- **Synthesized:** Value of a synthesized attribute of a node is computed from values of attributes of children of that node.
  - Used extensively in practice.
  - **S-attributed definition**: Annotated grammar that uses synthesized attributes only.
  - Information flows from bottom-up in a parse tree.
  - Terminals may have only synthesized attributes. Supplied by scanner.
  - Example:

| Productions | Rules |
|---|---|
| $E_1 \rightarrow E_2 + T$ | $E_1$`.val := `$E_2$`.val + `$T$`.val` |
| $E \rightarrow T$ | $E$`.val := `$T$`.val` |
| $T_1 \rightarrow T_2 * P$ | $T_1$`.val := `$T_2$`.val * `$P$`.val` |
| $T \rightarrow P$ | $T$`.val := `$P$`.val` |
| $P \rightarrow$ **id** | $P$`.val := `**id**` .lexeme` |
| $P \rightarrow (E)$ | $P$`.val := `$E$`.val` |

  - Information flow in parse tree for input `12 + 3 * 6`:



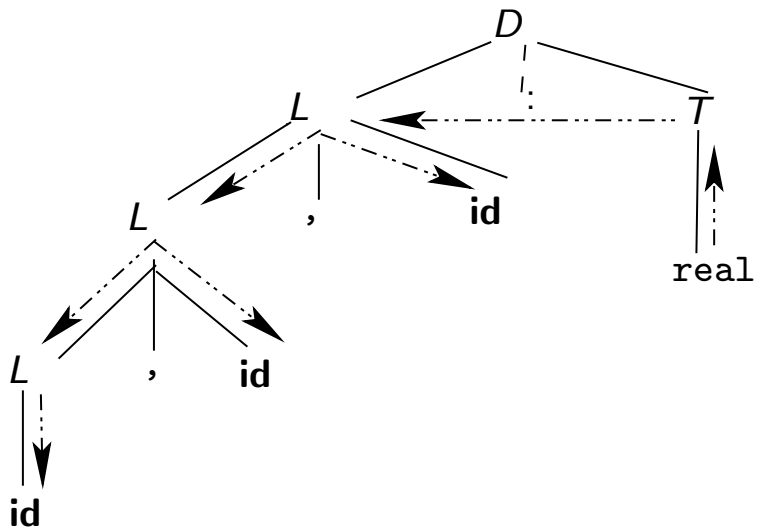
## Inherited Attributes

- Value at a node computed from attributes of parent and siblings.
- Convenient for expressing dependence of a programming language construct on the context in which it appears.
- Example:

| | | | |
|---|---|---|---|
| $D$ | $\rightarrow$ | $L : T$ | `L.in := T.type` |
| $T$ | $\rightarrow$ | *integer* | `T.type := integer` |
| $T$ | $\rightarrow$ | *real* | `T.type := real` |
| $L_1$ | $\rightarrow$ | $L_2$, **id** | $L_2$`.in =` $L_1$`.in` |
| | | | `addtype(`**id** `.entry,` $L$`.in)` |
| $L$ | $\rightarrow$ | **id** | `addtype(`**id** `.entry,` $L$`.in)` |

- Information flow in parse tree for string

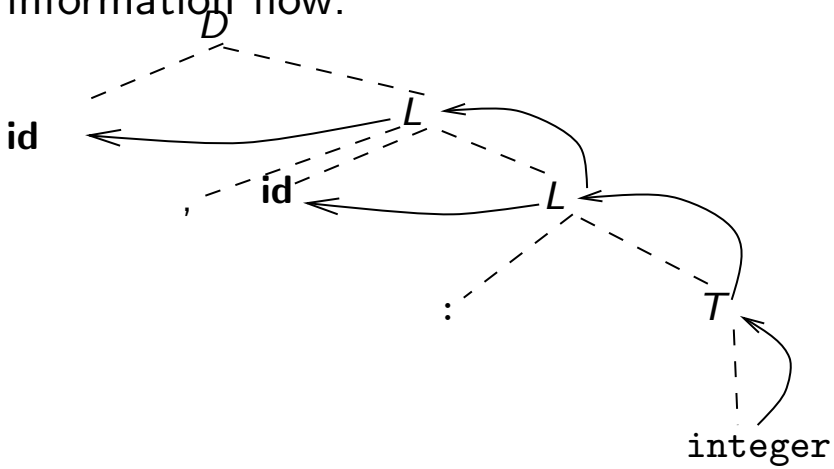
`id1, id2, id3:real`



## How to replace Inherited Attributes by Synthesized Attributes?

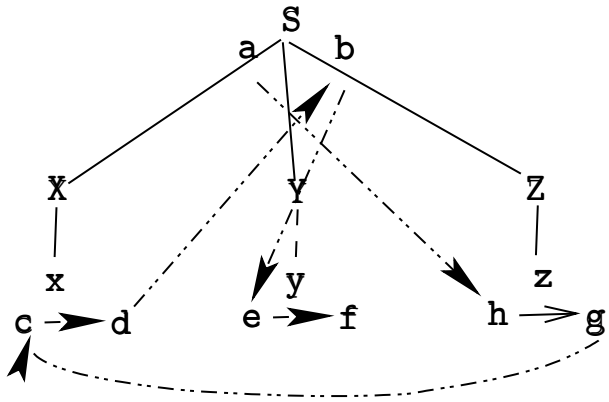
- Always possible to rewrite an annotated grammar to use only synthesized attributes.
- Can replace inherited attributes by changing underlying grammar.
- Modified grammar:
  $D \rightarrow \mathbf{id}\ L$
  $L \rightarrow ,\mathbf{id}\ L \mid : T$
  $T \rightarrow \mathit{integer} \mid \mathit{real}$

- information flow:



## How to represent dependencies among attributes?

- Language semantics specifies how an attribute can be evaluated in terms of other attributes; the attribute is said to be *dependent* on other attributes.
- Example: Nonterminals and their attributes:




```
S: a {inh}, b {syn}
X: c {inh}, d {syn}
Y: e {inh}, f {syn}
Z: h {inh}, g {syn}
```

| | |
|---|---|
| $S \rightarrow XYZ$ | `Z.h := S.a` |
| | `X.c := Z.g` |
| | `S.b := X.d -2` |
| | `Y.e := S.b` |
| $X \rightarrow x$ | `X.d = 2*X.c` |
| $Y \rightarrow y$ | `Y.f = Y.e+3` |
| $Z \rightarrow z$ | `X.d = 2*X.c` |

- Dependencies among attributes represented as a graph.

## How to evaluate attributes?

- Construct an evaluation order from attribute dependencies. That is, order attributes according to dependencies.
- Topological sorting: Construct an order $m_1, m_2 \cdots m_n$ such that $i < j \Rightarrow$ there is a directed path from $m_i$ to $m_j$. On other words, if $m_j$ depends on $m_i$, it can only be evaluated after $m_i$ has been evaluated.
  Example: code generated for previous example:

  ```
  Z.h := S.a; Z.g := Z.h; X.c := Z.g
  X.d := X.c; S.b := X.d; Y.e := S.b;
  Y.f := Y.e
  ```

  In topological sorting, dependencies must be specified somehow.
- In certain cases, evaluation order is chosen without considering semantic rules. For instance, order of evaluation may be forced by parsing methods.
  Such evaluation orders restrict class of syntax-directed definitions.
- Note that synthesized attributes can be evaluated by a bottom-up parser as input is parsed.
  Reminder: S-attributed definitions contain grammar symbols annotated with only synthesized attributes.

## Bottom-up Evaluation of S-Attributed Definitions

- Approach:
  1. Semantic stack: Store attributes. (May be separate from main stack).
  2. For every symbol shifted, store its corresponding attribute on stack.
  3. For every reduction $A \rightarrow \alpha\beta$, compute attribute of $A$ by popping off attributes for $\alpha$ and $\beta$ from semantic stack.
- Example:

| Productions | Rules |
|---|---|
| $E_1 \rightarrow E_2 + T$ | $E_1$.val := $E_2$.val + $T$.val |
| $E \rightarrow T$ | $E$.val := $T$.val |
| $T_1 \rightarrow T_2 * P$ | $T_1$.val := $T_2$.val * $P$.val |
| $T \rightarrow P$ | $T$.val := $P$.val |
| $P \rightarrow$ **id** | $P$.val := **id** .lexeme |
| $P \rightarrow (E)$ | $P$.val := $E$.val |

- Stack for 2 + 3:

| Stack | Val stack | Input | Reduction | Action |
|---|---|---|---|---|
| | | 2 + 3 | | |
| 2 | | + 3 | | |
| P | P.v | + 3 | $P \rightarrow$ **id** | P.v=2 |
| T | T.v | + 3 | $T \rightarrow P$ | T.v=P.v |
| E1 | E1.v | + 3 | $E \rightarrow T$ | E1.v=T.v |
| E1+ | E1.v | 3 | | |
| E1+2 | E1.v | 3 | | |
| E1+P | E1.v P.v | | $P \rightarrow$ **id** | P.v=3 |
| E1+T | E1.v T.v | | $T \rightarrow P$ | T.v=P.v |
| E | E.v | | $E \rightarrow E + T$ | E.v=E1.v+T.v |

## L-attributed definitions

- **L-attributed Definition**: A definition is L-attributed if each inherited attributed of a NT, say $X_j$, in a rule $A \rightarrow X_1 X_2 \cdots X_n$ depends only on
  1. attributes of NT that occur to left of $X_j$ ($X_1, X_2, \cdots X_{j-1}$, and
  2. inherited attributes of $A$.
- Attributes of L-attributed definitions can be evaluated using depth-first order evaluation:

```
DepthFirstVisit(N:node)
   foreach child m of n (from left to right)
      evaluate inherited attributes of m
      DepthFirstVisit(m);
   evaluate synthesized attributes of n.
end
```

- Evaluate inherited attributes while going down the tree, whereas evaluate synthesized attributes while coming up.
- Given an L-attributed grammar in which no inherited attributes depend on synthesized attributes, a recursive-descent parser can evaluate all attributes by turning inherited attributes into parameters and synthesized attributed into return values.

## Translation Scheme (TS)

- Semantic Action: Program fragments that will be added to CFG.
- TS: A CFG in which attributes are associated with grammar symbols and semantic actions enclosed between braces $\{\}$ are inserted within right side of productions.

  Like syntax-directed definitions *except* order of evaluation is explicitly shown.
- Example of TS:

  $E \rightarrow TR$

  $R \rightarrow$ **addop** $T$ $\{print(\textbf{addop}.lexeme)\}R_1|\epsilon$

  $T \rightarrow$ **num** $\{print(\textbf{num}.val)\}$

  Note semantic action in middle.

  Show A parse tree for string $9 - 5 + 2$

  Treat semantic actions as though terminal symbols. When performed in depth-first order, actions print out $95 - 2+$.
- In a TS, an attribute value should be available when an action in $\{\}$ refers to it.
- For S-attributed grammar, the following transformation from syntax-directed definition to TS:

  `Production  Semantic Rule`

  $T_1 \rightarrow T_2 * F \quad T_1.val := T_2.val \times F.val$

  `Translation Scheme:`

  $T_1 \rightarrow T_2 * F \quad \{T_1.val := T_2.val \times F.val\}$

## Translation Scheme - cont'd.

- In a translation scheme,
  1. Inherited attribute for a symbol on right side of production must be computed in action before that symbol.
  2. An action must not refer to a synthesized attribute of a symbol to right of action
  3. Synthesized attribute can be computed only after all attribute it references has been computed.
- In grammar

  $S \rightarrow A_1 A_2 \quad \{A_1.in := 1; A_2.in := 2\}$
  $A \rightarrow a \quad \{print(A.in)\}$

  inherited attribute `in` of $A$ is not evaluated when $a$ is reduced to $A$.
  Does not satisfy first constraint.
- Example L-attributed grammar: syntax directed translation of grammar that defines placement of boxes on page (or screen for that matter).
  For instance E sub 1.val puts boxes E, . and val next to each other, whereas box 1 is shrunk and put into a subscript position.
  Represented boxes by NT $B$.
  Attributes: $ps$: point size, $ht$: height of text

## Example - cont'd.

- Grammar:

| Production | | | Semantic Rule |
|---|---|---|---|
| $S$ | $\rightarrow$ | $B$ | $B.ps := 10$ |
| | | | $S.ht := B.ht$ |
| $B$ | $\rightarrow$ | $B_1 B_2$ | $B_1.ps := B.ps$ |
| | | | $B_2.ps := B.ps$ |
| | | | $B.ht := max(B_1.ht, B_2.ht)$ |
| $B$ | $\rightarrow$ | $B_1 sub B_2$ | $B_1.ps := B.ps$ |
| | | | $B_2.ps := shrink(B.ps)$ |
| | | | $B.ht := disp(B_1.ht, B_2.ht)$ |
| $B$ | $\rightarrow$ | $text$ | $B.ht := text.h \times B.ps$ |

- Translation Scheme:

| | | | |
|---|---|---|---|
| $S$ | $\rightarrow$ | | $\{B.ps := 10\}$ |
| | | $B$ | $\{S.ht := B.ht\}$ |
| $B$ | $\rightarrow$ | | $\{B_1.ps := B.ps\}$ |
| | | $B_1$ | $\{B_2.ps := B.ps\}$ |
| | | $B_2$ | $\{B.ht := max(B_1.ht, B_2.ht)\}$ |
| $B$ | $\rightarrow$ | | $\{B_1.ps := B.ps\}$ |
| | | $B_1$ | |
| | | sub | $\{B_2.ps := shrink(B.ps)\}$ |
| | | $B_2$ | $\{B.ht := disp(B_1.ht, B_2.ht)\}$ |
| $B$ | $\rightarrow$ | $text$ | $\{B.ht := text.h \times B.ps\}$ |

## Bottom up evaluation of Inherited Attributes

- Problem: How to use the semantic stack to locate inherited value of siblings?
- Case 1: Grammar allows us to predict where inherited value will be.

$$
\begin{array}{lcll}
D & \rightarrow & T & \{L.in := T.type\} \\
 & & L & \\
T & \rightarrow & integer & \{T.type := integer\} \\
T & \rightarrow & real & \{T.type := real\} \\
L_1 & \rightarrow & & \{L_2.in = L_1.in\} \\
 & & L_2, \textbf{id} & \{addtype(\textbf{id}.entry, L_1.in)\} \\
L & \rightarrow & \textbf{id} & \{addtype(\textbf{id}.entry, L.in)\}
\end{array}
$$

- parser's move on input `p, q: real`:

| Input | State | Rule |
|---|---|---|
| real p, q | - | |
| p, q | real | |
| p, q | T | $T \rightarrow real$ |
| , q | T p | |
| ,q | T L | $L \rightarrow \textbf{id}$ |
| q | T L, | |
| | T L,q | |
| | T L | $L \rightarrow L, \textbf{id}$ |
| | D | $D \rightarrow TL$ |

- Note: every time the right side of a production for $L$ is reduced, $T$ is in stack just below right side. Since $L.in$ is same as $T.type$, use $T.Type$ instead of $L.in$.

$$
\begin{array}{lcll}
D & \rightarrow & TL & \\
T & \rightarrow & integer & \texttt{val[ntop] = integer;} \\
T & \rightarrow & real & \texttt{val[ntop] = real;} \\
L_1 & \rightarrow & L_2, \textbf{id} & \texttt{addtype(val[top], val[top-3]);} \\
L & \rightarrow & \textbf{id} & \texttt{addtype(val[top], val[top-1]);}
\end{array}
$$

## Bottom up evaluation of Inherited Attributes - cont'd.

- Case 2: What if position of attribute value cannot be predicted in stack?

| Production | Semantic Rule |
|---|---|
| $S \rightarrow aAC$ | $C.i := A.s$ |
| $S \rightarrow bABC$ | $C.i := A.s$ |
| $C \rightarrow c$ | $C.s := g(C.i)$ |

- Resolve by systematically adding marker non-terminals:

| Production | Semantic Rule |
|---|---|
| $S \rightarrow aAC$ | $C.i := A.s$ |
| $S \rightarrow bABMC$ | $M.i := A.s; C.i := M.s$ |
| $C \rightarrow c$ | $C.s := g(C.i)$ |
| $M \rightarrow \epsilon$ | $M.s := M.i$ |

- The above transformation ensures that $C.i$ can be found in $val[top - 1]$.
- Show the attribute evaluation figure.
- Approach also used to simulate semantic rules that are not simple copies:

| Production | Semantic Rule |
|---|---|
| $S \rightarrow aAC$ | $C.i := f(A.s)$ |

- Solution: Again, add marker:

| Production | Semantic Rule |
|---|---|
| $S \rightarrow aANC$ | $N.i := A.s; C.i := N.s$ |
| $N \rightarrow \epsilon$ | $N.s := f(N.i)$ |

## Semantic Checking/Contextual Analysis

- Are type rules being used properly?
  - Type equivalence rules
  - Type compatibility
  - Type coercion
- Uniqueness checks
- Name related checks
- Flow of control checks
- For our sample language, additional rules
  - Enforce this super constraints
  - No loops in inheritance structure
- Focus here: specify a type checker using attributes:
  - Verify if the type rules of a language are being enforced.

## Type System

- Type checking: Check to ensure that every operator/function is applied on parameters of valid types.
  Example:

  ```
  X := Y + 1;
  ```

  Two possible violations (called **type-clash**)
  - Application of "+" operator over operands of wrong type (X, 1)
  - Application of ":=" operator over operands of wrong type
- Widely know language design principle: every expression must have a type that is known and fixed at compile time.
- Type inference rules: languages define a set of rules for inferring types from expressions.

  ```
  X := Y * (Z + W);
  ```

  Given that Y, Z and W are integer, what should be the type of expression?
- Type system: set of rules for associating a type with expressions in a language. A type system *rejects* an expression if it does not associate a type with the expression.
  - Defines types +
  - Rules for
    1. Type equivalence: Are two types equal?
    2. Type compatibility
    3. Type inference

## Type checking

- Static: Types of expressions and variables determined from the text of the program. Performed by the translator before execution.
- Dynamic: If type information is maintained and checked at runtime, the checking is dynamic. Dynamic checking required when types of objects can be determined only at runtime
  Examples: LISP, Scheme, SmallTalk.
  Usually languages with dynamic scoping require dynamic typing.
- A type system is strong if it accepts only those expressions that are guaranteed to evaluate without a type error.
  FORTRAN is not strongly typed: COMMON can be used to set up two variables of different types to alias to same location.
  Ada, Algol68, Modula-2, Pascal: strongly typed.
  C has many loopholes:
  - Union
  - Subroutines with variable numbers of parameters
  - Interoperability of integers, pointers, and arrays
- A type system is weak if it accepts expressions that may contain type errors.
- A strong type system accepts some subset of safe programs; however, we have no information about the size of this subset.

# Type Equivalence

## Structural equivalence

- Two types are same if they consist of the same component put together in the same way
  - Set-based view: two sets are same if they contain the same values.

    $$A \times B \neq B \times A$$
  - A type name is structurally equivalent to itself

    `char` $\equiv$ `char`
  - Two type expressions are *structurally equivalent* if they are formed by applying the same type constructor to structurally equivalent types

    `array[0..99] of char` $\equiv$ `array[0..99] of char`
  - After a type declaration, `type n = T`, the type name n is structurally equivalent to T.
- Structure equivalence relatively easier to implement.
- Used in Algol60, Algol68, FORTRAN, and COBOL
- How much information is included in a type constructor?

  ```
  T1 = array[-1..9] of integer;
  T2 = array[0..10] of integer;
  ```

  Algol68: dimension and component type of an array are part of its type but not the index set

  Ada: Base type of index set is also part of an array type but not the bounds.

  If size in included, problems with writing functions:

  ```
  Sort(array[1..10] of integer):  integer;
  ```
- Are two records with different field names but same types equivalent? (in Algol68, they are not..)

# Type equivalence - cont'd.

## Name equivalence

- A type name is equivalent to itself, but no constructed type is equal to any other constructed type.

```
type  y:  array[1..10] of integer;
      z:  array[1..10] of integer;
var   a, b:  y;
      c:  z;
```

- Stronger than structural equivalence because a name can refer to only a single type equivalence.
- Name equivalence available in Ada, Java, and Standard Pascal.
- Ambiguities in name equivalence: what if no names are given?

```
var   x:  array[1..10] of integer;
      y:  array[1..10] of integer;
```

  Not name equivalent as internal names for types will be different

## Declaration equivalence

- Type names that lead back to same original structural declaration by a series of re-declaration

```
type  t1 = array[1..10] of integer;
      t2 = t1;
      t3 = t2;
```

- Used in Pascal and Modula-2
- C: mixture of structural and declaration equivalence
  Declaration equivalence for structures and unions
  structural equivalence for pointers and arrays

## Type compatibility + Type Conversion

- Given expression `i := j + 2.3;`
- In C, translator performs implicit type-conversions
- Modula-2: error because integer and real types cannot be intermixed.

```
i := TRUNC (FLOAT(j) + 2.3)
```

- In many languages (e.g.,C) type conversion is built into the type system so that they can be applied automatically. Also called *type coercion*
  Makes for simpler programming but weakens the type system: some errors may escape.
  Also, automatic conversions may not match user's expectations. For instance, PL/I's conversion: value of
  `(1/3 + 15)` is 5.3333333333333..... The leading 1 is lost by overflow because the precision of fractional values must be maintained.
- cast: A value or variable is preceded by a type name.

```
int (3.14)
```

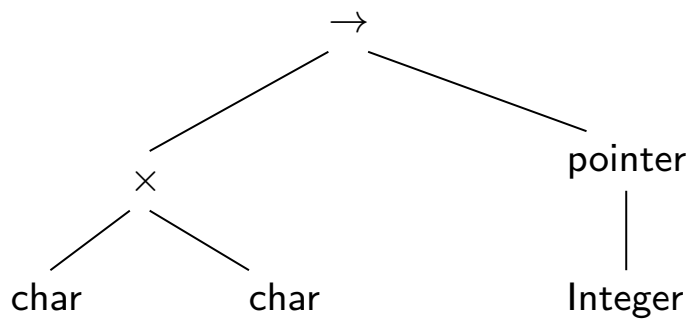
  In C, casts are recommended for use in allocating storage pointer:

```
char *str;
str = (char *)malloc(strlen(string)+1);
```

- Type conversion through union: Undiscriminated unions can hold values of different types, and without a tag, a translator cannot distinguish values of one type from another.

## Type Expressions

- Define a representation for types of languages.
- Primitive types: real, integer, char, boolean. Also, add an `ERRORTYPE` to denote errors in types.
- Arrays: Represent as $array(I, T)$ where $T$ is type of element, and $I$ represents index set $I$.
- Records: Represent record as $record((name_1, type_1), (name_2, type_2), \cdots)$. Here $name_i$ denotes the ith field name.
- Pointer: $pointer(T)$ denotes type "pointer to an object of type $T$."
- Functions: Use expression $T_1 \times T_2 \times \cdots T_n \rightarrow R$ to denote $R\ f(T_1, T_2, \ldots, T_n)$
- Type expressions can be represented as a graph as well.
  Example: graphical representation of $char \times char \rightarrow pointer(integer)$



## Specification of Type Checker

- Specify a translation scheme that synthesizes type of each expression from type of its subexpressions
- Language:
  $P \rightarrow D\ ;\ E$
  $D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T$
  $T \rightarrow \textbf{char} \mid \textbf{integer} \mid \textbf{Array}\ [\textbf{num}]\ \textbf{of}\ T \mid \uparrow T$
  $E \rightarrow \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E\ \textbf{mod}\ E \mid E\ [E] \mid E \uparrow$

  $P$: Program
  $D$: declaration
  $E$: expression
- Translation scheme for saving type of an identifier:
  $T.type$: synthesized attribute of $T$.
  $addtype$: add an attribute for an entry in symbol table.

$$
\begin{array}{lll}
P \rightarrow D; E & & \\
D \rightarrow D; D & & \\
D \rightarrow \textbf{id}\ :\ T & \{addtype(\textbf{id}\ .entry, T.type)\} & \\
T \rightarrow \textbf{char} & \{T.type := char\} & \\
T \rightarrow \textbf{integer} & \{T.type := integer\} & \\
T \rightarrow \textbf{Array}[\textbf{num}]\textbf{of}\ T_1 & \{T.type := array(1..\textbf{num}.val, T_1.type)\} & \\
T \rightarrow \uparrow T_1 & \{T.type := pointer(T_1.type)\} &
\end{array}
$$

## Type checking - cont'd.

- *type*: synthesized attribute of an expression $E$.
- Semantic rules for tokens.

  $E \rightarrow$ **literal** $\{E.type := char\}$

  $E \rightarrow$ **num** $\{E.type := integer\}$
- For identifiers:

  $E \rightarrow$ **id** $\{E.type := lookup(\textbf{id}.entry);\}$

  *lookup* searches symbol table for an identifier.

  What if identifier is not in table? Give an error indicating that identifier is not declared. Also set $E.type$ to `ERRORTYPE`.
- Mod operator:

  $E \rightarrow E_1$ **mod** $E_2$ { $E.type$ `:= if` $E_1.type$ `= integer and` $E_2.type$ `= integer then integer else ERRORTYPE;` }
- Array reference:

  $E \rightarrow E_1[E_2]$ { $E.type$ `:= if` $E_2.type$ `= integer and` $E_1.type$ `= array(s, t) then` $t$ `else ERRORTYPE;` }

  $t$: Array element type.
- Pointer reference

  $E \rightarrow E_1 \uparrow$ { $E.type$ `:= if` $E_1.type$ `= pointer(t) then` $t$ `else ERRORTYPE;` }

## Type checking of statements and functions

- Since statements do not have any values, they can be assigned a type `void` or `ERRORTYPE` to indicate a type error.

  $S \rightarrow$ **id** $:= E$ { $S.type$ := if **id** $.type = E.type$)
  then *void* else ERRORTYPE; }

  $S \rightarrow$ **if** $E$ **then** $S_1$ { $S.type$ := if $E.type$ = boolean)
  then $S_1.type$ else ERRORTYPE; }

  $S \rightarrow$ **while** $E$ **do** $S_1$ { $S.type$ := if $E.type$ = boolean)
  then $S_1.type$ else ERRORTYPE; }

  $S \rightarrow S_1 \; ; \; S_2$ { $S.type$ := if $S_1.type$ = void and
  $S_2.type$ = void then void
  else ERRORTYPE; }

- Type specification for function definitions:
  $T \rightarrow T_1 \; ' \rightarrow ' \; T_2$ $\{T.type := T_1.type \rightarrow T_2.type\}$
- Check application of functions/procedures

  $E \rightarrow E_1(E_2)$ { $E.type$ := if $E_2.type$ = $s$ and
  $E_1.type = s \rightarrow t$ then $t$
  else ERRORTYPE}

  Can be generalized to function of more than one argument.

## Type coercion

```
E ::= num
 {E.type = integer}
E ::= num . num
 {E.type = real}
E ::= if
 {E.type = lookup (id.entry)}
E ::= E1 op E2   { E.type
 {E.type = if E1.type = integer and
              E2.type = integer
            then integer
          else if E1.type = integer and
                  E2.type = real
            then real
          else if E1.type = real and
                  E2.type = integer
            then real
          else if E1.type = real and
                  E2.type = real
            then real
  }
```

## Additional type checking

- Overloading: allow users to overload the usage of names of functions and operators.
  - Augment symbol table look up mechanism so that one can use both name and type of parameters to look for a function.
  - Maintain set of possible types for each name, and return the set. Use the set to disambiguate
- Type conversion and coercion: During type checking, not just check for equivalence but also check if they can be converted..
  You will have to remember type-subtype rule.
- Polymorphic typing: Need a more sophisticated pattern matching algorithm for ensuring type safety.