Beyond syntax analysis

- An identifier named x has been recognized.
  - Is x a scalar, array or function?
  - How big is x?
  - If x is a function, how many and what type of arguments does it take?
  - Is x declared before being used?
  - Where can x be stored?
  - Is the expression x+y type-consistent?

- Semantic analysis is the phase where we collect information about the types of expressions and check for type related errors.

- The more information we can collect at compile time, the less overhead we have at run time.

Semantic Analysis

- Ultimate goal: generate machine code.
- Before we generate code, we must collect information about the program:
- Front end
  - scanning (recognizing words) CHECK
  - parsing (recognizing syntax) CHECK
  - semantic analysis (recognizing meaning)
    - There are issues deeper than structure. Consider:

```
int func (int x, int y);
int main () {
        int list[5], i, j;
        char *str;
        j = 10 + 'b';
        str = 8;
        m  = func("aa", j, list[12]);
        return 0;
}
```

This code is syntactically correct, but will not work. What problems are there?

Semantic analysis

- Collecting type information may involve "computations"
  - What is the type of x+y given the types of x and y?
- Tool: attribute grammars
  - CFG
  - Each grammar symbol has associated attributes
  - The grammar is augmented by rules (semantic actions) that specify how the values of attributes are computed from other attributes.
  - The process of using semantic actions to evaluate attributes is called syntax-directed translation.
  - Examples:
    - Grammar of declarations.
    - Grammar of signed binary numbers.

Attribute grammars

Example 1: Grammar of declarations

Production	Semantic rule
$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ char	T.type = character
$L \rightarrow L_1$, id	L_1.in = L.in
	addtype (id.index, L.in)
$L \rightarrow$ id	addtype (id.index, L.in)

Attribute grammars

Example 2: Grammar of signed binary numbers

Production	Semantic rule

$N \rightarrow S\ L$	if (S.neg)
	        print('-');
	else print('+');
	print(L.val);

$S \rightarrow +$	S.neg = 0
$S \rightarrow -$	S.neg = 1
$L \rightarrow L_1, B$	$L.val = 2 * L_1.val + B.val$
$L \rightarrow B$	L.val = B.val
$B \rightarrow 0$	$B.val = 0 * 2^0$
$B \rightarrow 1$	$B.val = 1 * 2^0$

Attribute grammars

Example 3: Grammar of expressions
Creating an AST

The attribute for each non-terminal is a node of the tree.

Production	Semantic rule
$E \rightarrow E_1 + E_2$	$E.node = new\ PlusNode(E_1.node, E_2.node)$
$E \rightarrow num$	$E.node = num.yylval$
$E \rightarrow (E_1)$	$E.node = E_1.node$

- Notes:
  - yylval is assumed to be a node (leaf) created during scanning.
  - The production $E \rightarrow (E_1)$ does not create a new node as it is not needed.

Syntax-Directed Definitions and Translation Schemes

- When we associate semantic rules with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes**
- **Syntax-Directed Definitions:**
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- **Translation Schemes:**
  - indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, translation schemes give a little bit information about implementation details.

Syntax-Directed Definitions and Translation Schemes

- With each production in a grammar, we give semantic rules or *actions, which describe* how to compute the attribute values associated with each grammar symbol in a production. The attribute value for a parse node may depend on information from its children nodes below or its siblings and parent node above.
- Evaluation of these semantic rules (using SDT one can perform following with parser):
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform consistency check like type checking, parameter checking etc…
  - may issue error messages
  - may build syntax tree
  - in fact, they may perform almost any activities.
- Procedure :

1) Input – Grammer
2) Output – Attached semantic rules

Syntax-Directed Definitions

- A syntax-directed definition is a generalization of a context-free grammar in which:
  - Each grammar symbol is associated with a set of attributes.
  - This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol.
  - Each production rule is associated with a set of semantic rules.
- *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.
- This *dependency graph* determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.
- The order of these computations depends on the dependency graph induced by the semantic rules.

- An attribute is said to be **synthesized** if its value at a parse tree node is determined by the attribute values at the child nodes.
- An attribute is said to be **inherited** if its value at a parse tree node is determined by the attribute values of the parent and/or siblings of that node.

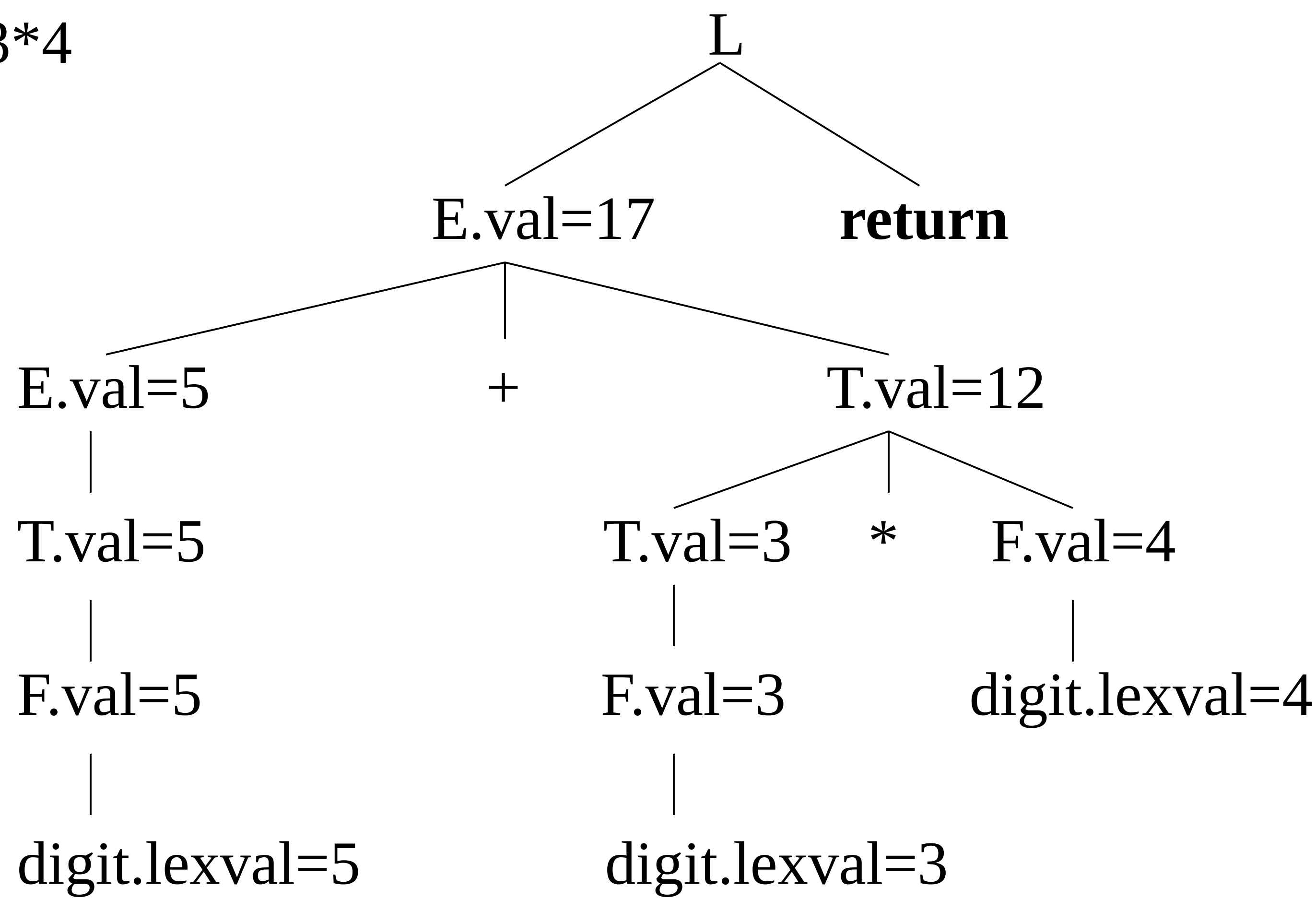
Example - Synthesized Attributes

Production	**Semantic Rules**
$L \rightarrow E$ **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **digit**	F.val = **digit**.lexval

- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
- Terminals attributes calculated at the time of lexical analysis phase

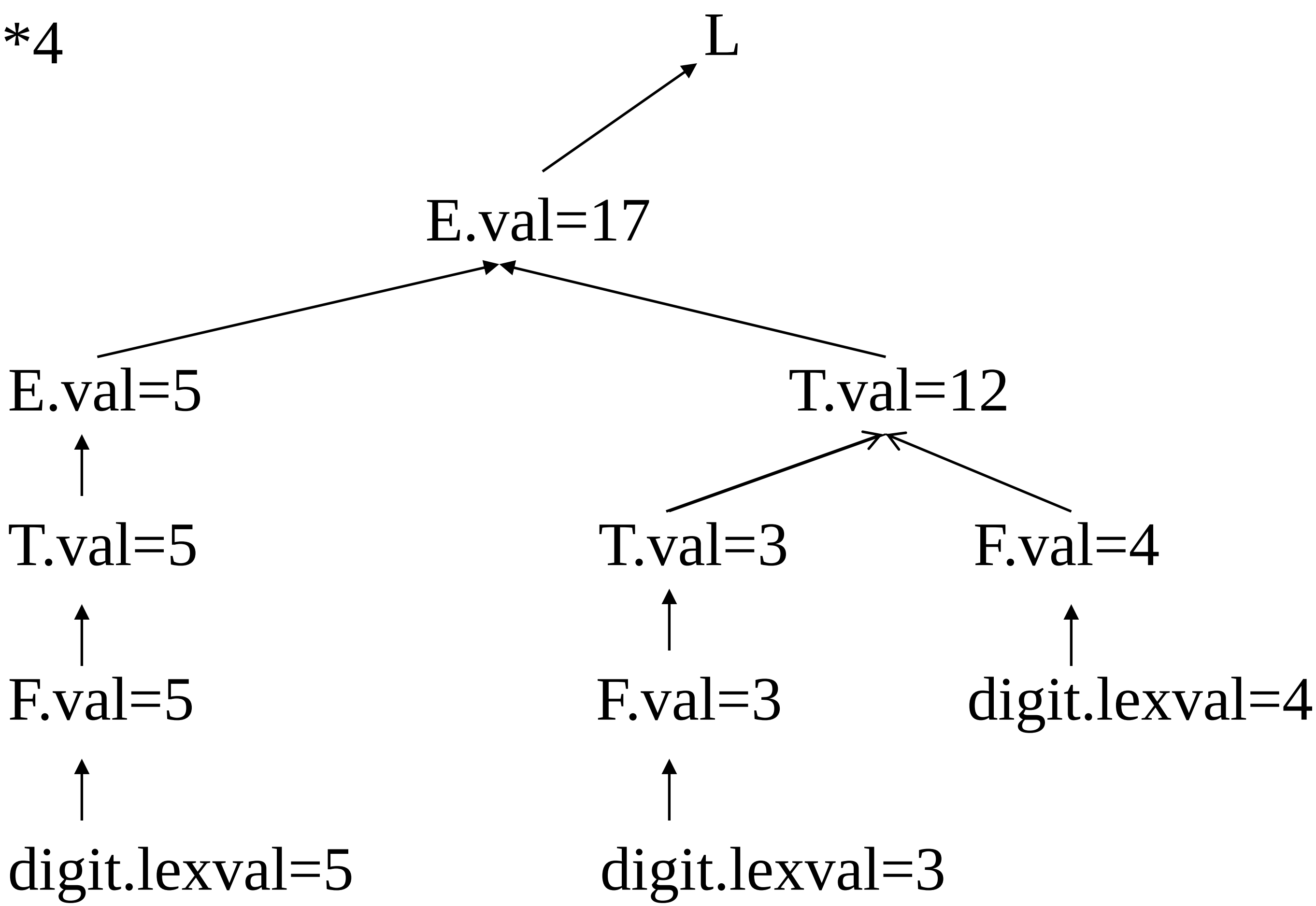
Annotated Parse Tree -- Example

Input: 5+3*4



Dependency Graph

Input: 5+3*4



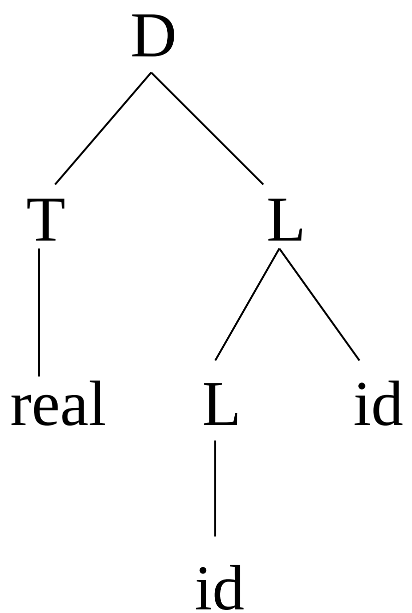
Example - Inherited Attributes

<u>**Production**</u>	<u>**Semantic Rules**</u>
$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	T.type = real
$L \rightarrow L_1$ **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
$L \rightarrow$ **id**	addtype(**id**.entry,L.in)

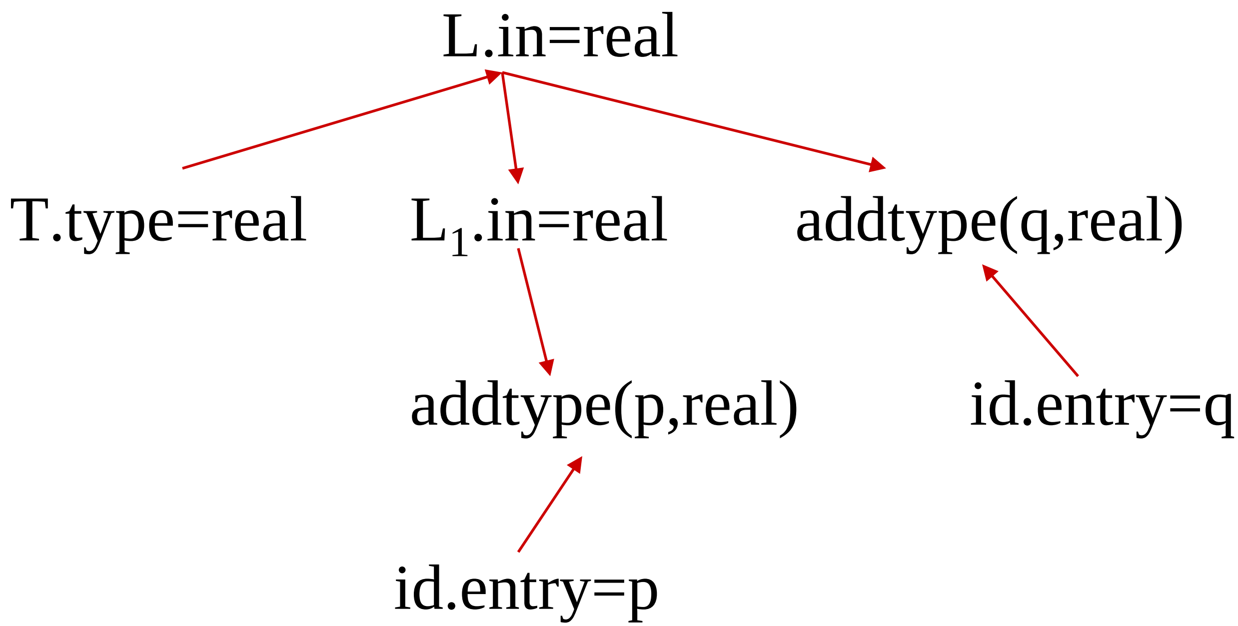
- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in*.

A Dependency Graph

Input: `real p q`



parse tree



dependency graph

S-Attributed Definitions

- There are two sub-classes of the syntax-directed definitions:
  - **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
  - **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.
- Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

L-Attributed Definitions

- A syntax-directed definition is **L-attributed** if each inherited attribute of X_j, where $1 \leq j \leq n$, on the right side of $A \rightarrow X_1X_2...X_n$ depends only on:
  1. The attributes of the symbols $X_1,...,X_{j-1}$ to the left of X_j in the production and
  2. the inherited attribute of A

- L-Attributed Definitions can always be evaluated by the depth first visit of the parse tree.
- This means that they can also be evaluated during the parsing.
- Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes).

A Definition which is NOT L-Attributed

Productions	**Semantic Rules**
$A \rightarrow L\ M$	L.in=l(A.i), M.in=m(L.s), A.s=f(M.s)
$A \rightarrow Q\ R$	R.in=r(A.in), Q.in=q(R.s), A.s=f(Q.s)

- .s = synthesized attributes , .in = inherited attributes
- This syntax-directed definition is not L-attributed because the semantic rule Q.in=q(R.s) violates the restrictions of L-attributed definitions.
- When Q.in must be evaluated before we enter to Q because it is an inherited attribute.
- But the value of Q.in depends on R.s which will be available after we return from R. So, we are not be able to evaluate the value of Q.in before we enter to Q.

Translation Schemes

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated?).

- A **translation scheme** is a context-free grammar in which:
  - attributes are associated with the grammar symbols and
  - semantic actions enclosed between braces {} are inserted within the right sides of productions.

- *Ex:* $A \rightarrow \{ \ldots \}\ X\ \{ \ldots \}\ Y\ \{ \ldots \}$

Semantic Actions

Translation Schemes

- When designing a translation scheme, some restrictions should be observed to ensure that an attribute value is available when a semantic action refers to that attribute.
- These restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet computed.
- In translation schemes, we use *semantic action* terminology instead of *semantic rule* terminology used in syntax-directed definitions.
- The position of the semantic action on the right side indicates when that semantic action will be evaluated.

Translation Schemes for S-attributed Definitions

- If our syntax-directed definition is S-attributed, the construction of the corresponding translation scheme will be simple.
- Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.

Production	Semantic Rule	
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$	➔ a production of a syntax directed definition

$$\Downarrow$$

$E \rightarrow E_1 + T \; \{ \; E.val = E_1.val + T.val \; \}$ ➔ the production of the corresponding translation scheme

To sum up,

- S-attributed definition (also called post fix definition):
- Uses only synthesized attributes
- Semantic actions are placed at the end of production rules
- Attributes may be evaluated during bottom up parsing

- L-attributed definition:
- Allows both synthesized as well as inherited attributes
- Semantic actions can be anywhere at the right end side
- Attributes are evaluated by traversing the parse tree depth first , left to right

Example: convert infix to postfix

E->E+T {print('+')} #1
|T
T->T*F {print('*')} #2
|F
F->num {print(num.val)} #3

→ S-attributed definition

By Removing left recursion….luckily it becomes L-attributed definitions(not always possible)

E->TE'
E'->+T#1E'|ε
T->FT'
T'->*F#2T'| ε
F->num#3

→ L-attributed definition

Convert L-AD to S-AD

If we have L-attributed attribute like

S->B{}C

Then we can convert it to S-attributed in following manner:

S->BMC

M-> ε {}

- Convert following L-attributed definitions to S-attributed:

E->TE’

E’->+T{print(‘+’);}E’| ε

T->num{print(num.val);}

SDT to store type info into Symbol Table

```
D->TL          {L.in=T.type;}
T-> int        {T.type=int;}
   | char      {T.type=char;}
L->L1,id       {L1.in=L.in; add-type(L1.in, id.num);}
   | id        {add-type(L.in, id.name);}
```

- Evaluation of synthesized attributes in S-attributed definition follows bottom up parsing
- Evaluation of synthesized as well as inherited attributes in L-attributed definition :

- Traverse the tree in depth first , left to right
- Evaluate inherited attribute when a node is visited for 1st time
- Evaluate synthesized attribute when a node is visited for last time

A Translation Scheme with Inherited Attributes

$D \rightarrow T$ **id** { addtype(**id**.entry,T.type), L.in = T.type } L

$T \rightarrow$ **int** { T.type = integer }

$T \rightarrow$ **real** { T.type = real }

$L \rightarrow$ **id** { addtype(**id**.entry,L.in), L_1.in = L.in } L_1

$L \rightarrow \varepsilon$

- This is a translation scheme for an L-attributed definitions.

SDT for type checking

E-> E1+E2 {if(E1.type==E2.type)&&(E1==int) than E.type=int else error;}

| E1==E2 {if(E1.type==E2.type)&&(E1.type==int/bool) than E.type=bool else error;}

|(E1) {E.type==E1.type;}

|num {E.type=int;}

|true {E.type=bool;}

|false {E.type=bool;}

- Here, .type is an attribute

SDT to build syntax tree

```
E->E+T          {E.nptr=mknode(E1.nptr, '+' , T.nptr);}
    |T          {e.nptr=T.nptr;}
T->T*F          {T.nptr=mknode(T.nptr, '*' , F.nptr);}
    |F          {t.nptr=F.nptr;}
F->id           {f.nptr=mknode(NULL, id.name , NULL);}
```

- mknode(lptr,data,rptr) – with three arguments – returns the pointer to newly created node.

- Ex – convert following SDT to an SDT that is a post fix(??) and has no left recursion:

A-> A {a} B | B {b}

B-> C {c}

Eliminate left recursion…

A-> B {b} A'

A'-> {a} B A' | ε

B -> C {c}

S-attributed…

A->A"A'

A"->B {b}

A'->MBA' | ε

M-> ε {A}

B-> C {c}

Ex – What is the final count value after calculating input string w=i+i*i

E-> E+T	{count = count + 5;}
| T	
T-> T*F	{count = count * 5;}
| F	
F = i	{count = 10;}

Ans: 55

Attributes

- Attributed parse tree = parse tree annotated with attribute rules
- Each rule implicitly defines a set of dependences
  - Each attribute's value depends on the values of other attributes.
- These dependences form an attribute-dependence graph.
- Note:
  - Some dependences flow upward
    - The attributes of a node depend on those of its children
    - We call those synthesized attributes.
  - Some dependences flow downward
    - The attributes of a node depend on those of its parent or siblings.
    - We call those inherited attributes.
- How do we handle non-local information?
  - Use copy rules to "transfer" information to other parts of the tree.

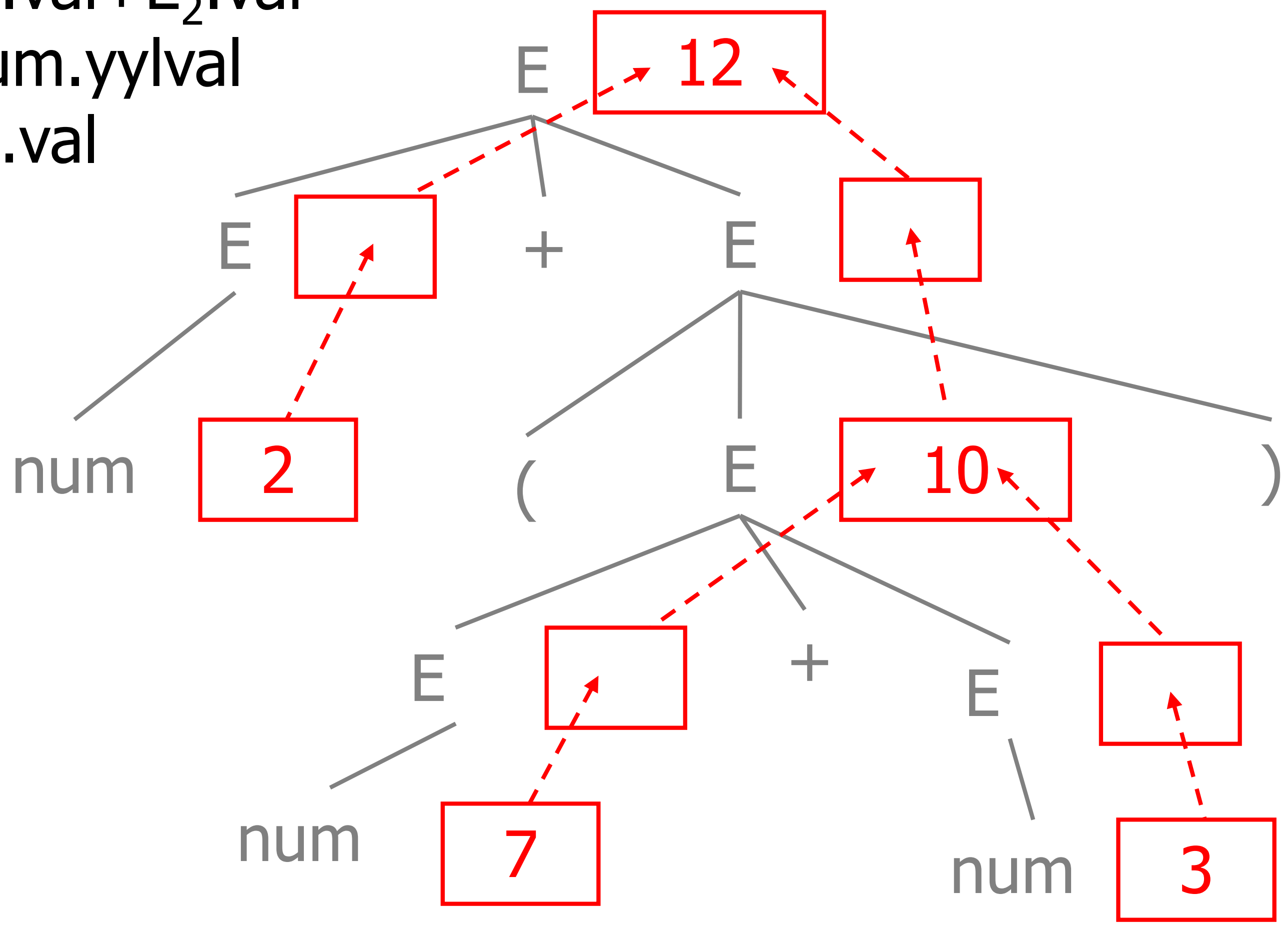
Attribute grammars

- Attribute grammars have several problems
  - Non-local information needs to be explicitly passed down with copy rules, which makes the process more complex
  - In practice there are large numbers of attributes and often the attributes themselves are large. Storage management becomes an important issue then.
  - The compiler must traverse the attribute tree whenever it needs information (e.g. during a later pass)
- However, our discussion of rule evaluation gives us an idea for a simplified approach:
  - Have actions organized around the structure of the grammar
  - Constrain attribute flow to one direction.
  - Allow only one attribute per grammar symbol.
- Practical application: BISON

Attribute grammars

Production	Semantic rule
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E \rightarrow num$	$E.val = num.yylval$
$E \rightarrow (E_1)$	$E.val = E_1.val$

attribute-dependence graph



Syntax-Directed Translation

- *Syntax-directed translation* refers to a method of compiler implementation where the source language translation is completely driven by the parser.
- In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program. This can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called *attribute grammars*.\
- We augment a grammar by associating *attributes with each grammar symbol* that describes its properties. An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register—whatever information we need.

Evaluating attributes

- The time when an attribute is evaluated often depends on the language.
  - Static attributes are those that are evaluated during compilation.
    - E.g. code
  - Dynamic attributes are those that are evaluated at runtime
    - E.g. a variable's type in a dynamically typed language.
- We are interested in static attributes.
- An attribute equation can be seen as a function:

$L \rightarrow L_1, B \qquad L.val = 2*L_1.val+B.val$

$L.val = f(L_1.val, B.val)$

To compute the function, its arguments must already have been evaluated.

Evaluating attributes

- Semantic analysis can be performed during parsing.
- Alternatively, we may wait until the parse is complete (and a parse tree has been created) and then traverse the tree and perform semantic analysis
  - This is potentially easier
  - It requires a second pass.

Evaluating attributes

- Method 1: Using a dependence graph
- Each rule implicitly defines a set of dependences
  - Each attribute's value depends on the values of other attributes.
- These dependences form an attribute-dependence graph.
- Attributed parse tree = parse tree annotated with attribute rules (i.e. the dependence graph is superimposed on the parse tree)
  - Some dependences flow upward
    - The attributes of a node depend on those of its children
    - We call those synthesized attributes.
  - Some dependences flow downward
    - The attributes of a node depend on those of its parent or siblings.
    - We call those inherited attributes.

Evaluating attributes

- Given a dependence graph, the attributes can be evaluated in topological order.
- This can only work when the dependence graph is acyclic.
  - Circular dependencies may appear due to features such as goto
  - It is possible to test for circularity.

Evaluating attributes

- Method 2: Rule-based
  - At compiler construction time
  - Analyze rules
  - Determine ordering based on grammatical structure (parse tree)

Attribute grammars

- We are interested in two kinds of attribute grammars:
  - S-attributed grammars
    - All attributes are synthesized
  - L-attributed grammars
    - Attributes may be synthesized or inherited, AND
    - Inherited attributes of a non-terminal only depend on the parent or the siblings to the left of that non-terminal.
      - This way it is easy to evaluate the attributes by doing a depth-first traversal of the parse tree.
- Idea (useful for rule-based evaluation)
  - Embed the semantic actions within the productions to impose an evaluation order.

Embedding rules in productions

- Synthesized attributes depend on the children of a non-terminal, so they should be evaluated after the children have been parsed.
- Inherited attributes that depend on the left siblings of a non-terminal should be evaluated right after the siblings have been parsed.
- Inherited attributes that depend on the parent of a non-terminal are typically passed along through copy rules (more later).

L.in is inherited and evaluated after parsing T but before L

T.type is synthesized and evaluated after parsing int

D $\rightarrow$ T {L.in = T.type} L
T $\rightarrow$ int {T.type = integer}
T $\rightarrow$ char {T.type = character}
L $\rightarrow$ {L_1.in = L.in} L_1, id {L.action = addtype (id.index, L.in)}
L $\rightarrow$ id {L.action = addtype (id.index, L.in)}

Rule evaluation in top-down parsing

- Recall that a predictive parser is implemented as follows:
  - There is a routine to recognize each lhs. This contains calls to routines that recognize the non-terminals or match the terminals on the rhs of a production.
  - We can pass the attributes as parameters (for inherited) or return values (for synthesized).
  - Example:
    $D \rightarrow T$ {L.in = T.type} L
    $T \rightarrow$ int {T.type = integer}
    - The routine for T will return the value T.type
    - The routine for L, will have a parameter L.in
    - The routine for D will call T(), get its value and pass it into L()

Rule evaluation in bottom-up parsing

- S-attributed grammars
  - All attributes are synthesized
  - Rules can be evaluated bottom-up
    - Keep the values in the stack
    - Whenever a reduction is made, pop corresponding attributes, compute new ones, push them onto the stack
- Example: Implement a desk calculator using an LR parser
  - Grammar:

Production	Semantic rule
$L \rightarrow E$ \n	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = yylval$

Rule evaluation in bottom-up parsing

Production	Semantic rule	Stack operation
$L \rightarrow E \backslash n$	print(E.val)	
$E \rightarrow E_1 + T$	$E.val = E_1.val+T.val$	val[newtop]=val[top-2]+val[top]
$E \rightarrow T$	$E.val = T.val$	
$T \rightarrow T_1 * F$	$T.val = T_1.val+F.val$	val[newtop]=val[top-2]*val[top]
$T \rightarrow F$	$T.val = F.val$	
$F \rightarrow (E)$	$F.val = E.val$	val[ntop]=val[top-1]
$F \rightarrow$ digit	F.val = yylval	

Rule evaluation in bottom-up parsing

- How can we inherit attributes on the stack? (L-attributed only)
- Use copy rules
  - Consider A→XY where X has a synthesized attribute s.
  - Parse X. X.s will be on the stack before we go on to parse Y.
  - Y can "inherit" X.s using copy rule Y.i = X.s where i is an inherited attribute of Y.
  - Actually, we can just use X.s wherever we need Y.i, since X.s is already on the stack.
- Example: back to the type declaration grammar:

Production	Semantic rule	Stack operation
D → T L	L.in = T.type	
T → int	T.type = integer	val[ntop]=integer
T → char	T.type = character	val[ntop]=character
L → L_1, id	L_1.in = L.in	
	addtype (id.index, L.in)	addtype(val[top], val[top-3])
L → id	addtype (id.index, L.in)	addtype(val[top], val[top-1])

Rule evaluation in bottom-up parsing

- Problem w/ inherited attributes: What if we cannot predict the position of an attribute on the stack?
- For example:

Production	Semantic rule
$S \rightarrow aAC$	C.i = A.s
$S \rightarrow bABC$	C.i = A.s
$C \rightarrow c$	C.s = f(C.i)

case 1: $S \Rightarrow aAC$

- After we parse A, we have A.s at the top of the stack.
- Then, we parse C. Since C.i=A.s, we could just use the top of the stack when we need C.i

case 2: $S \Rightarrow aABC$

- After we parse AB, we have B's attribute at the top of the stack and A.s below that.
- Then, we parse C. But now, A.s is not at the top of the stack.
- A.s is not always at the same place!

Rule evaluation in bottom-up parsing

- Solution: Modify the grammar.
  – We want C.i to be found at the same place every time
  – Insert a new non-terminal and copy C.i again:

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 = f(C.i)
$M \rightarrow \varepsilon$	M.s = M.i

  – Now, by the time we parse C, A.s will always be two slots down in the stack. So we can compute C.s by using stack[top-1]