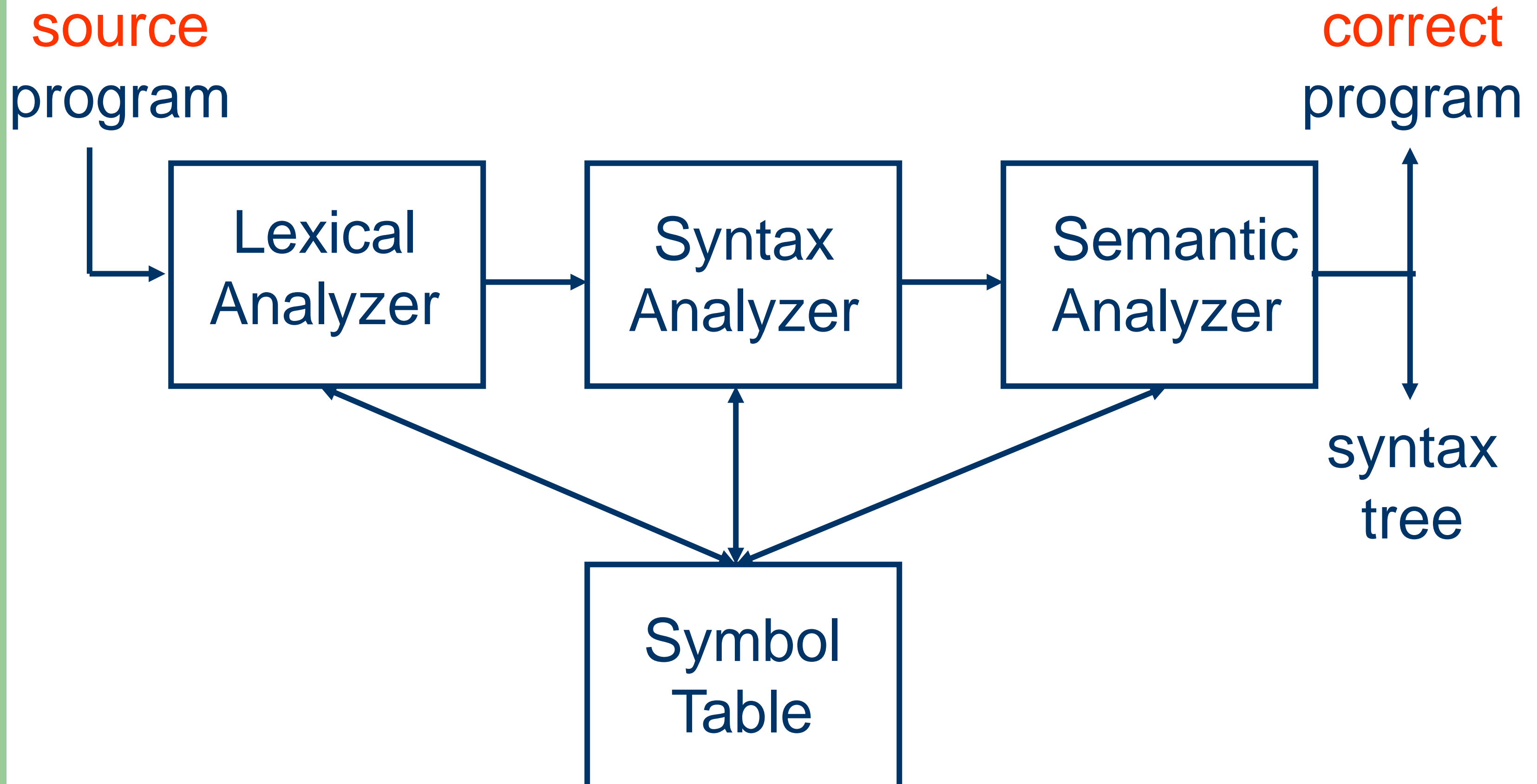
Semantic Analysis

Semantic Analysis

- Semantic Analyzer
- Attribute Grammars
- Syntax Tree Construction
- Top-Down Translators
- Bottom-Up Translators
- Recursive Evaluators
- Type Checking

Semantic Analyzer



Semantic Analysis

- Type-checking of programs
- Translation of programs
- Interpretation of programs

Attribute Grammars

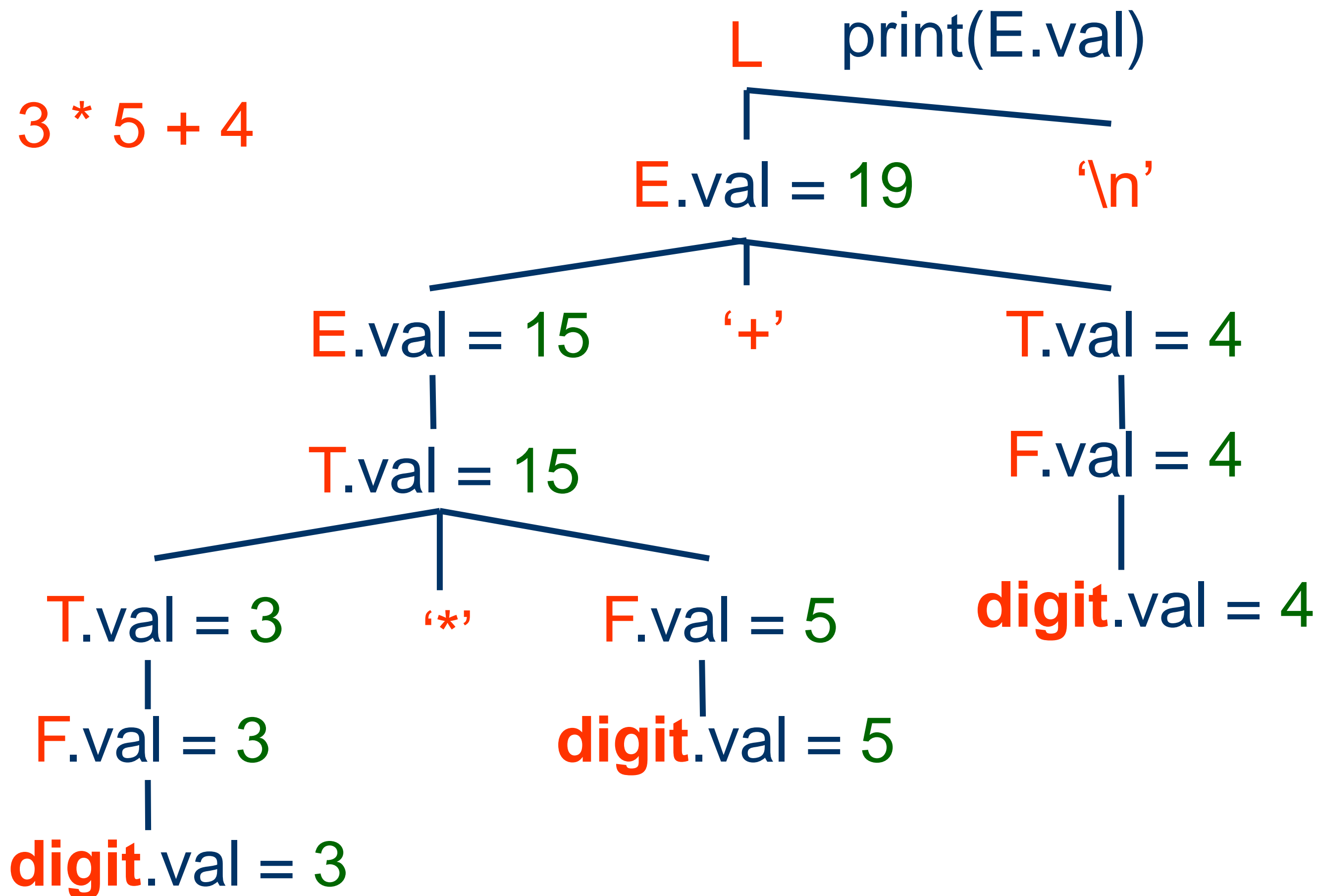
- An attribute grammar is a context free grammar with associated semantic attributes and semantic rules
- Each grammar symbol is associated with a set of semantic attributes
- Each production is associated with a set of semantic rules for computing semantic attributes

An Example - Interpretation

$L \rightarrow E$ '\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 := **digit**.val;}

Attribute val represents the value of a construct

Annotated Parse Trees



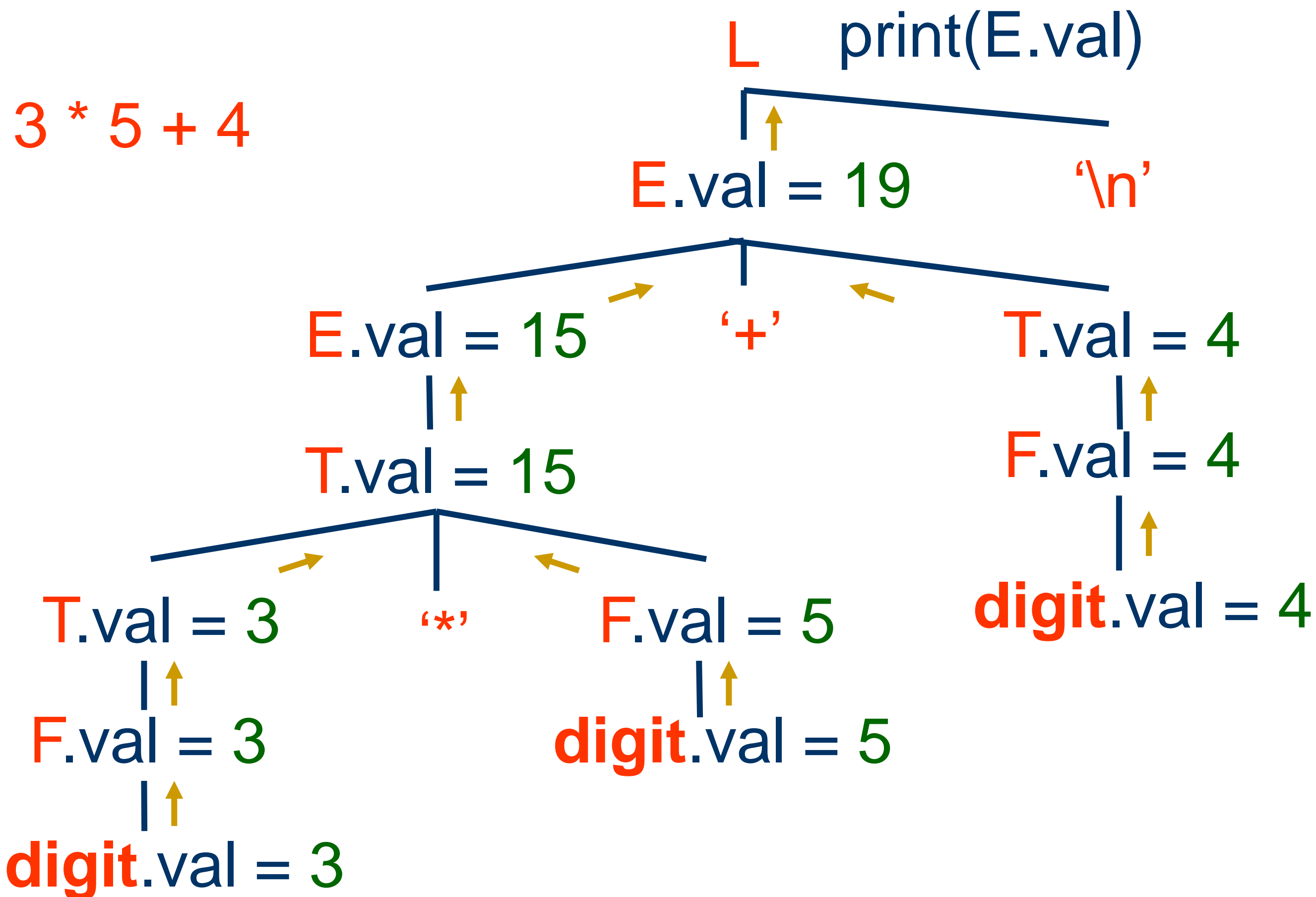
Semantic Attributes

- A (semantic) attribute of a node (grammar symbol) in the parse tree is synthesized if its value is computed from that of its children
- An attribute of a node in the parse tree is inherited if its value is computed from that of its parent and siblings

Synthesized Attributes

$L \rightarrow E$ '\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 := **digit**.val;}

Synthesized Attributes



Inherited Attributes

$D \rightarrow T$ {L.in := T.type;} L

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

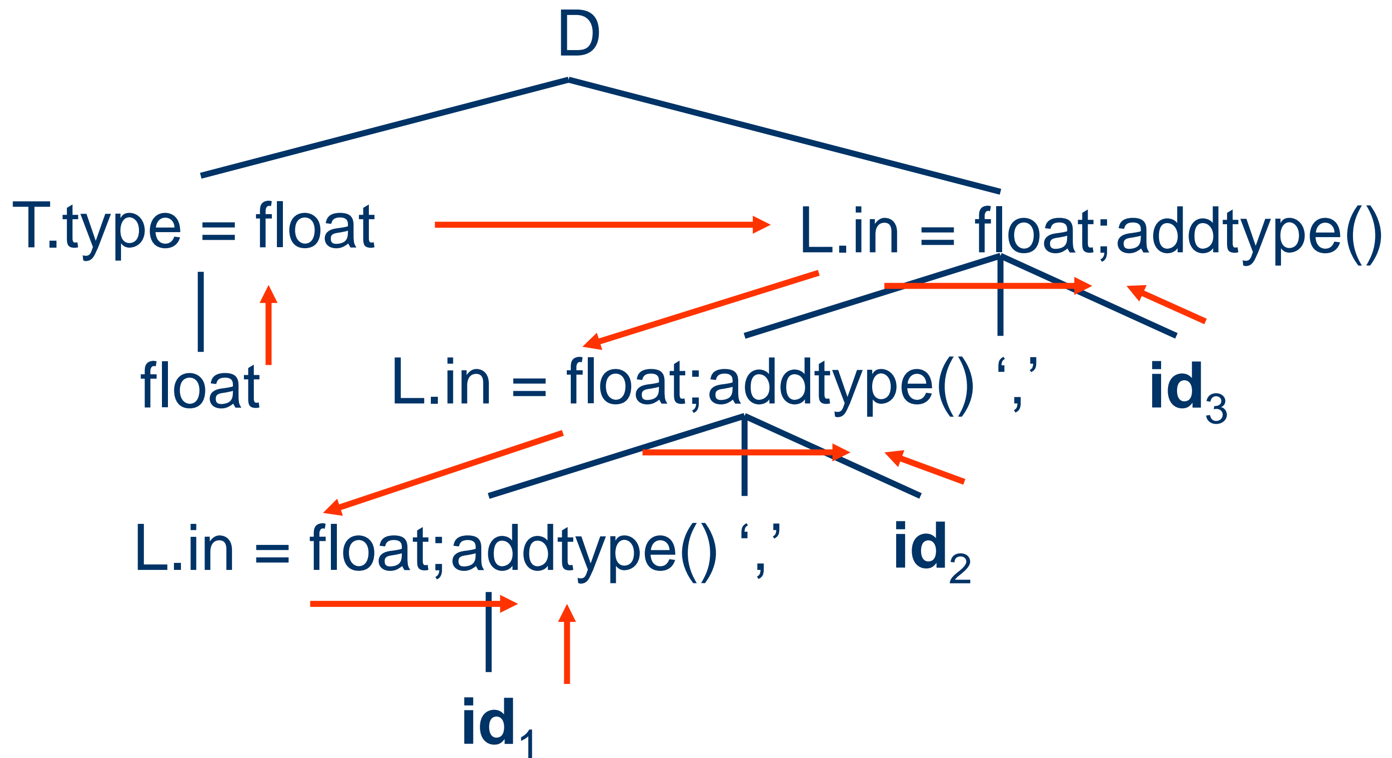
$T \rightarrow$ **float** {T.type := float;}

$L \rightarrow$ {L_1.in := L.in;}

L_1 ‘,’ **id** {addtype(**id**.entry, L.in);}

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

Inherited Attributes



Semantic Rules

- Each grammar production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form

  $b := f(c_1, c_2, \ldots, c_k)$

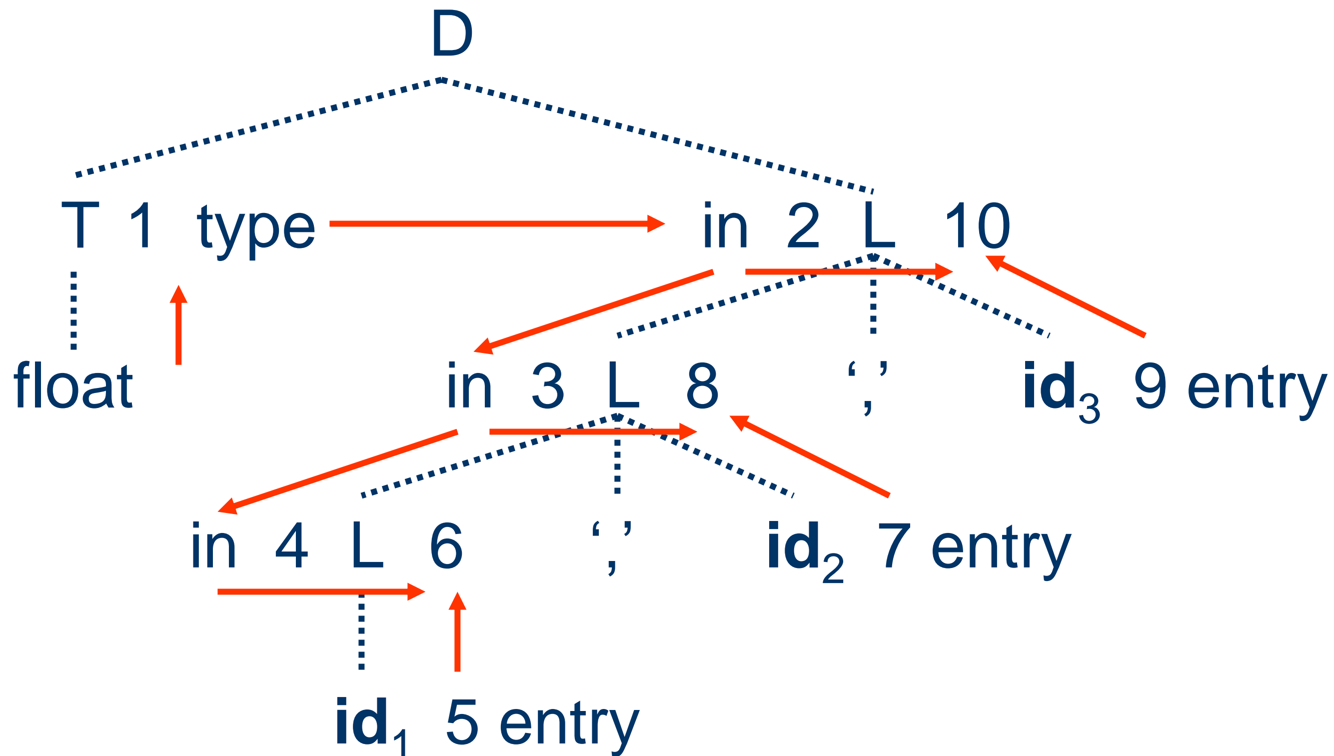
  where f is a function and

  1. b is a synthesized attribute of A and $c_1, c_2, \ldots, c_k$ are attributes of A or grammar symbols in α, or
  2. b is an inherited attribute of one of the grammar symbols in α and $c_1, c_2, \ldots, c_k$ are attributes of A or grammar symbols in α

Dependencies of Attributes

- In the semantic rule
  $b := f(c_1, c_2, \ldots, c_k)$
  we say b depends on $c_1, c_2, \ldots, c_k$
- The semantic rule for b must be evaluated after the semantic rules for $c_1, c_2, \ldots, c_k$
- The dependencies of attributes can be represented by a directed graph called dependency graph

Dependency Graphs



S-Attributed Attribute Grammars

- An attribute grammar is S-attributed if it uses synthesized attributes exclusively

An Example

$L \rightarrow E$ '\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 := **digit**.val;}

L-Attributed Attribute Grammars

- An attribute grammar is L-attributed if each attribute computed in each semantic rule for each production

  $A \to X_1\ X_2 \dots X_n$

  is a synthesized attribute, or an inherited attribute of X_j, $1 \le j \le n$, depending only on

  1. the attributes of $X_1, X_2, \dots, X_{j-1}$
  2. the inherited attributes of A

An Example

$D \rightarrow T\ L$	{L.in := T.type;}
$T \rightarrow$ **int**	{T.type := integer;}
$T \rightarrow$ **float**	{T.type := float;}
$L \rightarrow L_1$ ',' **id**	{L_1.in := L.in; addtype(**id**.entry, L.in);}
$L \rightarrow$ **id**	{addtype(**id**.entry, L.in);}

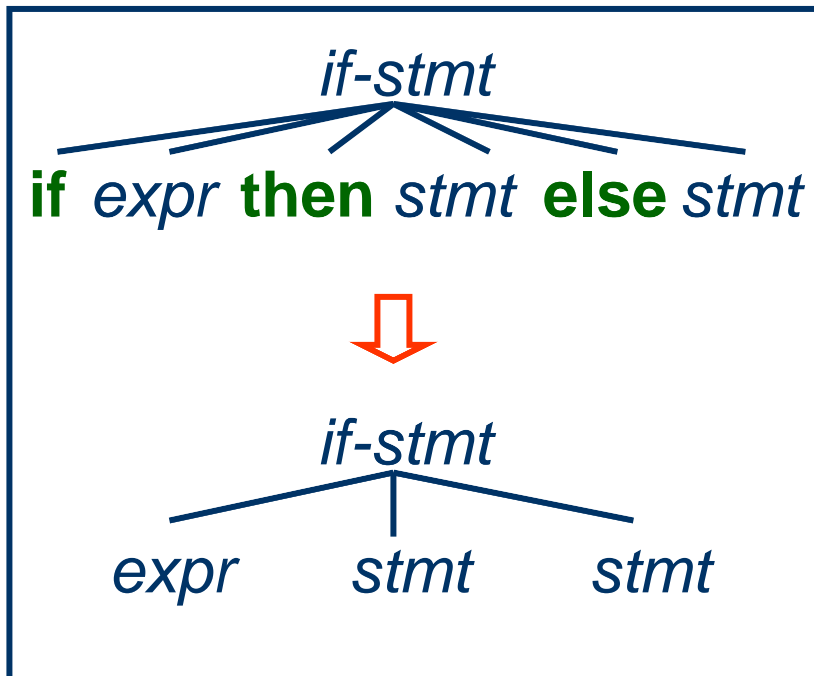
Evaluating L-attributes

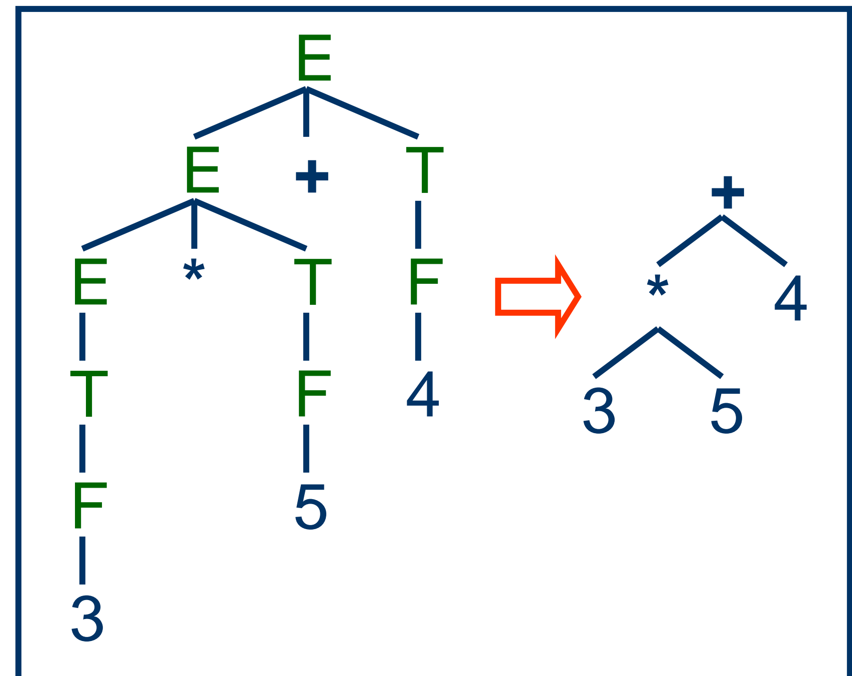
- The evaluation of L-attributes can be made by a depth-first traversal

```
function dfvisit (node)
    foreach child m of node from left to right do
        evaluate inherited attributes of m
        dfvisit (m)
    evaluate synthesized attributes of node
end
```

Construction of Syntax Trees

- An abstract syntax tree is a condensed form of parse tree



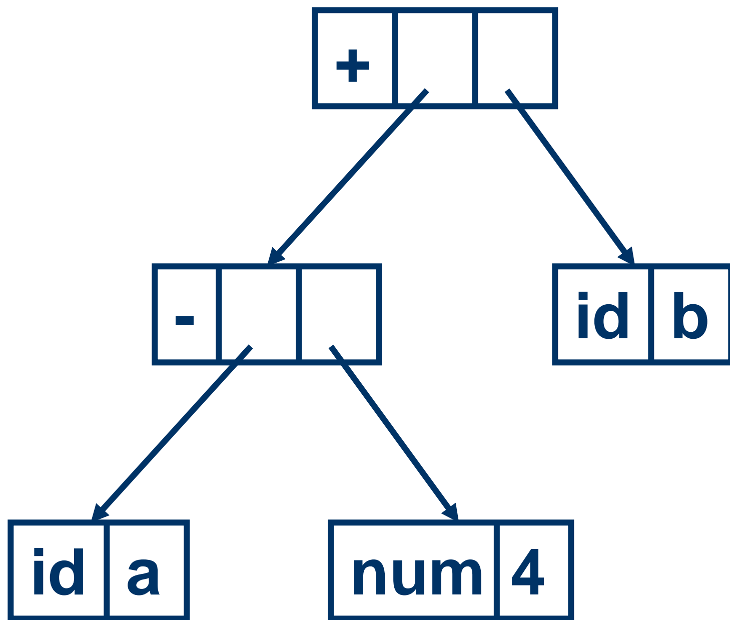


Syntax Trees for Expressions

- Interior nodes are operators
- Leaves are identifiers or numbers
- Functions for constructing nodes
  - mknode(op, left, right)
  - mkleaf(id, entry)
  - mkleaf(num, value)

An Example

a - 4 + b



p1 := mkleaf(id, $entry_a$);
p2 := mkleaf(num, 4);
p3 := mknode('-', p1, p2);
p4 := mkleaf(id, $entry_b$);
p5 := mknode('+', p3, p4);

An Example

$E \rightarrow E_1$ '+' T	{E.ptr := mknode('+', E_1.ptr, T.ptr);}
$E \rightarrow E_1$ '-' T	{E.ptr := mknode('-', E_1.ptr, T.ptr);}
$E \rightarrow T$	{E.ptr := T.ptr;}
$T \rightarrow$ '(' E ')'	{T.ptr := E.ptr;}
$T \rightarrow$ **id**	{T.ptr := mkleaf(id, **id**.entry);}
$T \rightarrow$ **num**	{T.ptr := mkleaf(num, **num**.value);}

Translation Schemes

- A translation scheme is a context-free grammar where attributes are associated with each symbol and actions are specified within the productions, usually with braces ({})
- This is not the same as a syntax-directed definition where semantics have simply been associated with the grammar rules.

Translation Schemes

- An infix-to-postfix translator, as a translation scheme:

  $E \rightarrow T\ R$

  $R \rightarrow$ addop T {print (addop. val)} R_1

  $R \rightarrow \varepsilon$

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

- With S-attributed definitions, translation schemes are essentially the same as the definition (attributes are computed at the end).

Translation Schemes

- With L-attributed definitions, one must follow some rules:
  1. An inherited attribute for a symbol must be computed in an action before that symbol.
  2. No action can refer to a synthesized attribute of a symbol to the right.
  3. A synthesized attribute is computed at the end of the rule.

Top-Down Translators

- For each nonterminal,
  - inherited attributes → formal parameters
  - synthesized attributes → returned values
- For each production,
  - for each terminal X with synthesized attribute x, save X.x; match(X);
  - for nonterminal B, c := B(b_1, b_2, …, b_k);
  - for each semantic rule, copy the rule to the parser

An Example - Interpretation

$E \rightarrow T$ { R.i := T.nptr } R { E.nptr := R.s }

$R \rightarrow$ **addop** T
{ R_1.i := mknode(**addop**.lexeme, R.i, T.nptr) }
R_1 { R.s := R_1.s }
$R \rightarrow \varepsilon$ { R.s := R.i }

$T \rightarrow$ '(' E ')' { T.nptr := E.nptr }
$T \rightarrow$ **num** { T.nptr := mkleaf(num, **num**.value) }

An Example

```
syntax_tree_node *E( );
syntax_tree_node *R( syntax_tree_node * );
syntax_tree_node *T( );
```

An Example

```
syntax_tree_node *E( ) {
    syntax_tree_node *enptr, *tnptr, *ri, *rs;
    switch (token) {
        case '(': case num:
            tnptr = T( );  ri = tnptr;        /* R.i := T.nptr */
            rs = R(ri);  enptr = rs;          /* E.nptr := R.s */
            break;
        default: error();
    }
    return enptr;
}
```

An Example

```
syntax_tree_node *R(syntax_tree_node * i) {
    syntax_tree_node *nptr, *i1, *s1, *s;   char add;
    switch (token) {
        case addop:
            add = yylval; match(addop);
            nptr = T(); i1 = mknode(add, i, nptr);
              /* R1.i := mknode(addop.lexeme, R.i, T.nptr) */
            s1 = R(i1);  s = s1; break;         /* R.s := R1.s */
        case EOF: s = i; break;                 /* R.s := R.i */
        default: error();  }
    return s;
}
```

An Example

```
syntax_tree_node *T( ) {
    syntax_tree_node *tnptr, *enptr;   int numvalue;
    switch  (token) {
        case ‘(’: match(‘(’);  enptr = E( );  match(‘)’);
            tnptr = enptr; break;          /* T.nptr := E.nptr */
        case num: numvalue = yylval;  match(num);
            tnptr = mkleaf(num, numvalue); break;
                /* T.nptr := mkleaf(num, num.value) */
        default: error( );
    }
    return tnptr;
}
```

Bottom-Up Translators

- Keep the values of synthesized attributes on the parser stack

$A \rightarrow X\ Y\ Z$ {A.a := f(X.x, Y.y, Z.z);}

	symbol	val	
	...	...	
	X	X.x	val[top-2]
	Y	Y.y	val[top-1]
top →	Z	Z.z	val[top]

$A \rightarrow X\ Y\ Z$ {val[ntop] := f(val[top-2], val[top-1], val[top]);}

Evaluation of Synthesized Attributes

- When a token is shifted onto the stack, its attribute value is placed in val[top]
- Code for semantic rules are executed just before a reduction takes place
- If the left-hand side symbol has a synthesized attribute, code for semantic rules will place the value of the attribute in val[ntop]

An Example

$L \rightarrow E$ '\n'	{print(val[top-1]);}
$E \rightarrow E_1$ '+' T	{val[ntop] := val[top-2] + val[top];}
$E \rightarrow T$	{val[ntop] := val[top];}
$T \rightarrow T_1$ '*' F	{val[ntop] := val[top-2] * val[top];}
$T \rightarrow F$	{val[ntop] := val[top];}
$F \rightarrow$ '(' E ')'	{val[ntop] := val[top-1];}
$F \rightarrow$ **digit**	{val[ntop] := digit.value;}

An Example

Input	symbol	val	production used
3*5+4n			
*5+4n	**digit**	3	
*5+4n	F	3	F → **digit**
*5+4n	T	3	T → F
5+4n	T *	3 _	
+4n	T * **digit**	3 _ 5	
+4n	T * F	3 _ 5	F → **digit**
+4n	T	15	T → T * F
+4n	E	15	E → T

An Example

Input	symbol	val	production used
+4n	E	15	E → T
4n	E +	15 _	
n	E + **digit**	15 _ 4	
n	E + F	15 _ 4	F → **digit**
n	E + T	15 _ 4	T → F
n	E	19	E → E + T
	E n	19 _	
	L	_	L → E n

Evaluation of Inherited Attributes

- Removing embedding actions from attribute grammar by introducing marker nonterminals

E → T R
R → "+" T {print('+')} R | "-" T {print('-')} R | ε
T → **num** {print(**num**.val)}

E → T R
R → "+" T M R | "-" T N R | ε
T → **num** {print(**num**.val)}
M → ε {print('+')}
N → ε {print('-')}

Evaluation of Inherited Attributes

- Inheriting synthesized attributes on the stack

$$A \rightarrow X \ \{Y.i := X.s\} \ Y$$

symbol	val
...	...
X	X.s
Y	

top → (Y)

An Example

$D \rightarrow T$ {L.in := T.type;} L
$T \rightarrow$ **int** {T.type := integer;}
$T \rightarrow$ **float** {T.type := float;}
$L \rightarrow$ {L_1.in := L.in} L_1 ',' **id** {addtype(**id**.entry, L.in);}
$L \rightarrow$ **id** {addtype(**id**.entry, L.in);}

$D \rightarrow T\ L$
$T \rightarrow$ **int** {val[ntop] := integer;}
$T \rightarrow$ **float** {val[ntop] := float;}
$L \rightarrow L_1$ ',' **id** {addtype(val[top], val[top-3]); }
$L \rightarrow$ **id** {addtype(val[top], val[top-1]); }

An Example

Input	symbol	val	production used
int p,q,r			
p,q,r	**int**	_	
p,q,r	T	*i*	T → **int**
,q,r	T **id**	*i p*	
,q,r	T L	*i* _	L → **id**
q,r	T L ,	*i* _ _	
,r	T L , **id**	*i* _ _ *q*	
,r	T L	*i* _	L → L "," **id**
r	T L ,	*i* _ _	
	T L , **id**	*i* _ _ *r*	
	T L	*i* _	L → L "," **id**
	D		

Evaluation of Inherited Attributes

- Simulating the evaluation of inherited attributes

Inheriting the value of a synthesized attribute works only if the grammar allows the position of the attribute value to be predicted

An Example

$S \rightarrow a\ A\ C$ {C.i := A.s}
$S \rightarrow b\ A\ B\ C$ {C.i := A.s}
$C \rightarrow c$ {C.s := g(C.i)}

$S \rightarrow a\ A\ C$ {C.i := A.s}
$S \rightarrow b\ A\ B\ M\ C$ {M.i := A.s; C.i := M.s}
$C \rightarrow c$ {C.s := g(C.i)}
$M \rightarrow \varepsilon$ {M.s := M.i}

Another Example

$S \rightarrow a\ A\ C$ {C.i := f(A.s)}

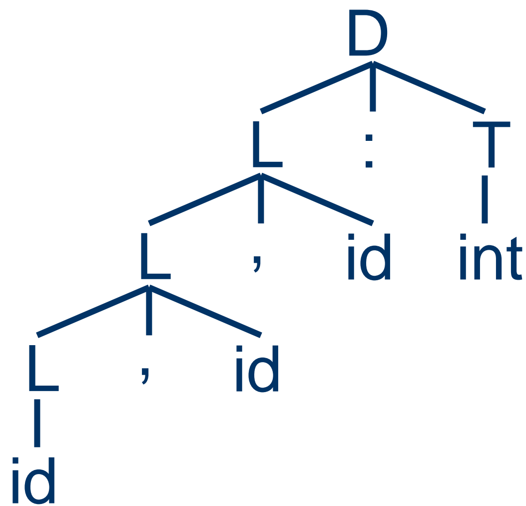
$S \rightarrow a\ A\ N\ C$ {N.i := A.s; C.i := N.s}

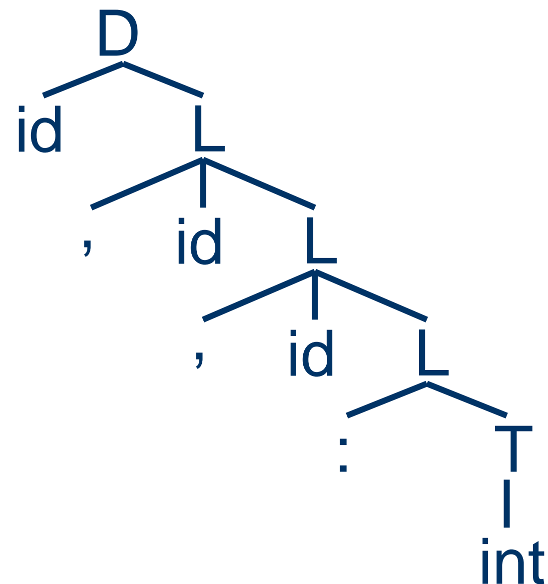
$N \rightarrow \varepsilon$ {N.s := f(N.i)}

From Inherited to Synthesized

$D \rightarrow L$ ":" T
$L \rightarrow L$ "," **id** | **id**
$T \rightarrow$ **integer** | **char**

$D \rightarrow$ **id** L
$L \rightarrow$ "," **id** L | ":" T
$T \rightarrow$ **integer** | **char**





Bison

```
%token DIGIT
%%
line :  expr  '\n'  {printf("%d\n", $1);}
     ;
expr:  expr  '+'  term  {$$ = $1 + $3;}
     |  term
     ;
term:  term  '*' factor  {$$ = $1 * $3;}
     |  factor
     ;
factor:  '('  expr  ')'  {$$ = $2;}
     |  DIGIT
     ;
```

Bison

```
%union {
        char op_type;
        int value;
  }
%token <value> DIGIT
%type  <op_type>  op
%type  <value>  expr  factor
%%
```

Bison

```
expr: expr op factor
        {$$ = $2 == '+' ? $1 + $3 : $1 - $3;}
      | factor
      ;
op: +   {$$ = '+';}
  | -   {$$ = '-';}
  ;
factor: DIGIT ;
```

Bison

```
program:
        { init_symbol_table(); }
        PROGRAM  ID
        { declare_program_name($3); }
        IS declarations BODY statements END
        { build_program( $3, $6, $8 ); }
    ;
```

Recursive Evaluators

- The parser constructs a syntax tree
- A recursive evaluator is a function that traverses the syntax tree and evaluates attributes
- A recursive evaluator can traverse the syntax tree in any order
- A recursive evaluator can traverse the syntax tree multiple times

An Example

$E_1.val$

$E_1.val$

An Example

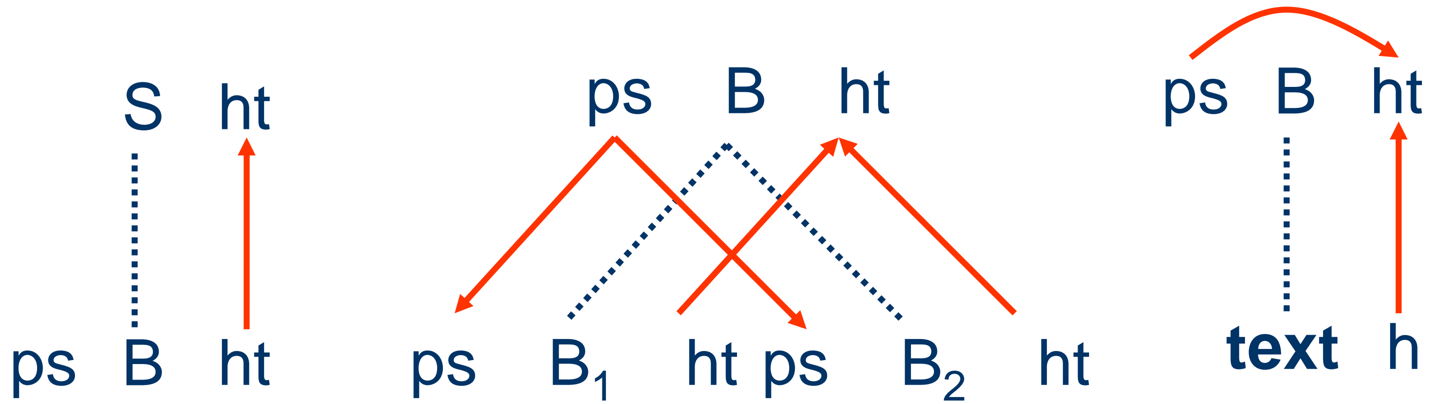
$S \rightarrow$ { B.ps := 10 } B { S.ht := B.ht }

$B \rightarrow$ { 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 }

An Example



An Example

```
function B(n, ps);
   var  ps1, ps2, ht1, ht2;
begin
   case  production at node n  of
   'B → B1 B2':
      ps1 :=  ps;  ht1 :=  B(child(n, 1), ps1);  ps2 :=  ps;
      ht2 :=  B(child(n, 2), ps2);  return max(ht1, ht2);
   'B → B1 sub B2':
      ps1 :=  ps;  ht1 :=  B(child(n, 1), ps1);  ps2 :=  shrink(ps);
      ht2 :=  B(child(n, 3), ps2);  return disp(ht1, ht2);
   'B → text':  return ps  ×  text.h;
   default:  error  end
end;
```

Type Systems

- A type system is a collection of rules for assigning types to the various parts of a program
- A type checker implements a type system
- Types are represented by type expressions

Type Expressions

- A basic type is a type expression
  - boolean, char, integer, real, void, type_error
- A type constructor applied to type expressions is a type expression
  - array: array(I, T)
  - product: $T_1 \times T_2$
  - record: $record((N_1 \times T_1) \times (N_2 \times T_2))$
  - pointer: pointer(T)
  - function: $D \rightarrow R$

Type Declarations

P → D ";" E

D → D ";" D

| **id** ":" T { addtype(**id**.entry, T.type) }

T → **char** { T.type := char }

T → **integer** { T.type := int }

T → "*" T_1 {T.type := pointer(T_1.type) }

T → **array** "[" **num** "]" **of** T_1

{ T.type := array(**num**.value, T_1.type) }

Type Checking of Expressions

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

$E \rightarrow$ **num** {E.type := int}

$E \rightarrow$ **id** {E.type := lookup(**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_1.type = array(s, t) and E_2.type = int
then t else type_error}

$E \rightarrow$ "*" E_1

{E.type := if E_1.type = pointer(t)
then t else type_error}

Type Checking of Statements

P → D ";" S

S → **id** ":=" E

{S.type := if lookup(**id**.entry) = E.type
then void else type_error}

S → **if** E **then** S_1

{S.type := if E.type = boolean then S_1.type else type_error}

S → **while** E **do** S_1

{S.type := if E.type = boolean then S_1.type else type_error}

S → S_1 ";" S_2

{S.type := if S_1.type = void and S_2.type = void
then void else type_error}

Type Checking of Functions

$T \rightarrow T_1$ "$\rightarrow$" T_2

{$T.type := T_1.type \rightarrow T_2.type$}

$E \rightarrow E_1$ "(" E_2 ")"

{$E.type :=$ if $E_1.type = s \rightarrow t$ and $E_2.type = s$

then t else type_error}