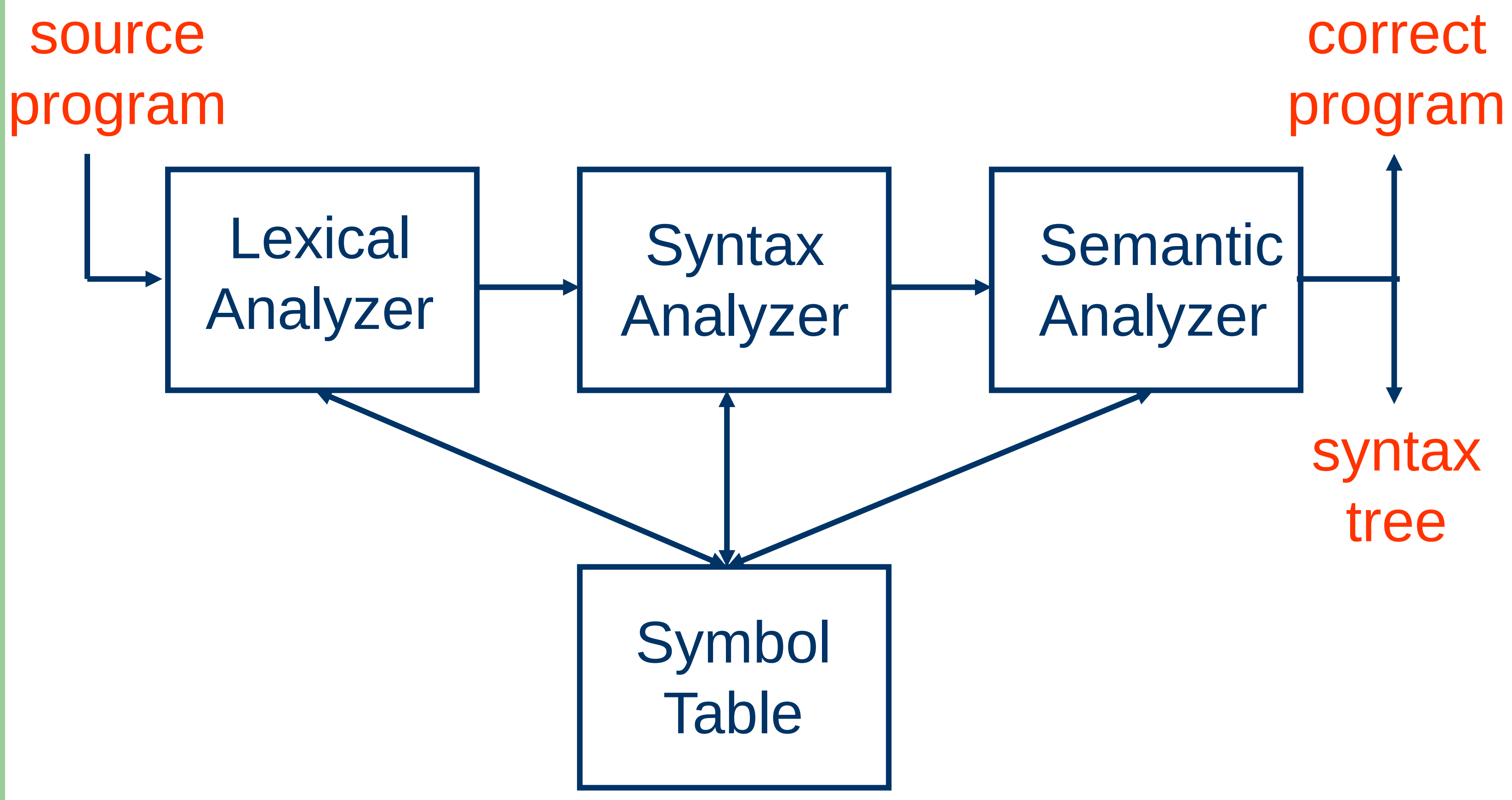
Semantic Analysis

Semantic Analysis

- Semantic Analyzer
- Attribute Grammars
- Syntax Tree Construction
- Top-Down Translators
- 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

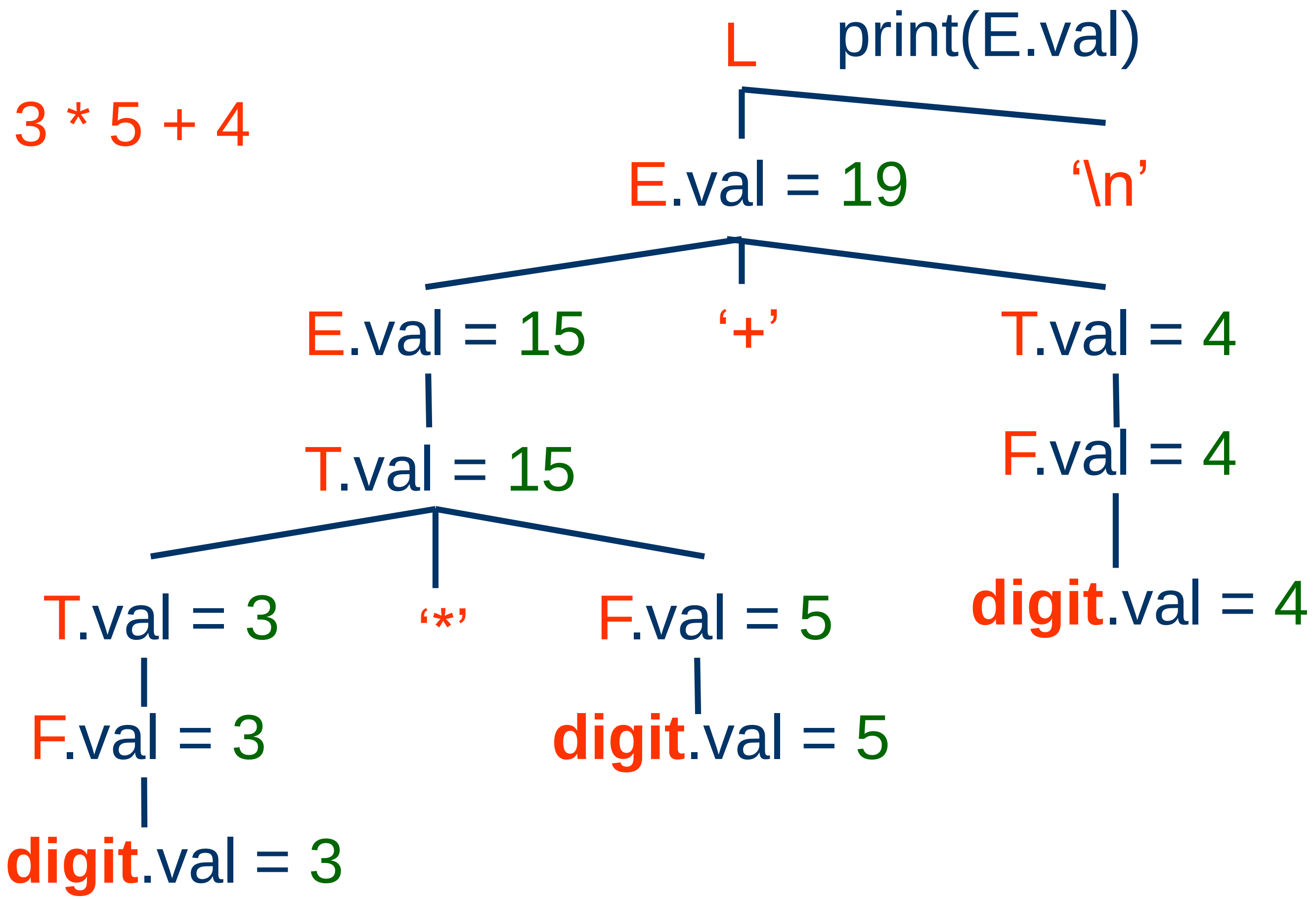
An Example - Interpretation

$L \rightarrow E$ ‘\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

3 * 5 + 4



Semantic Attributes

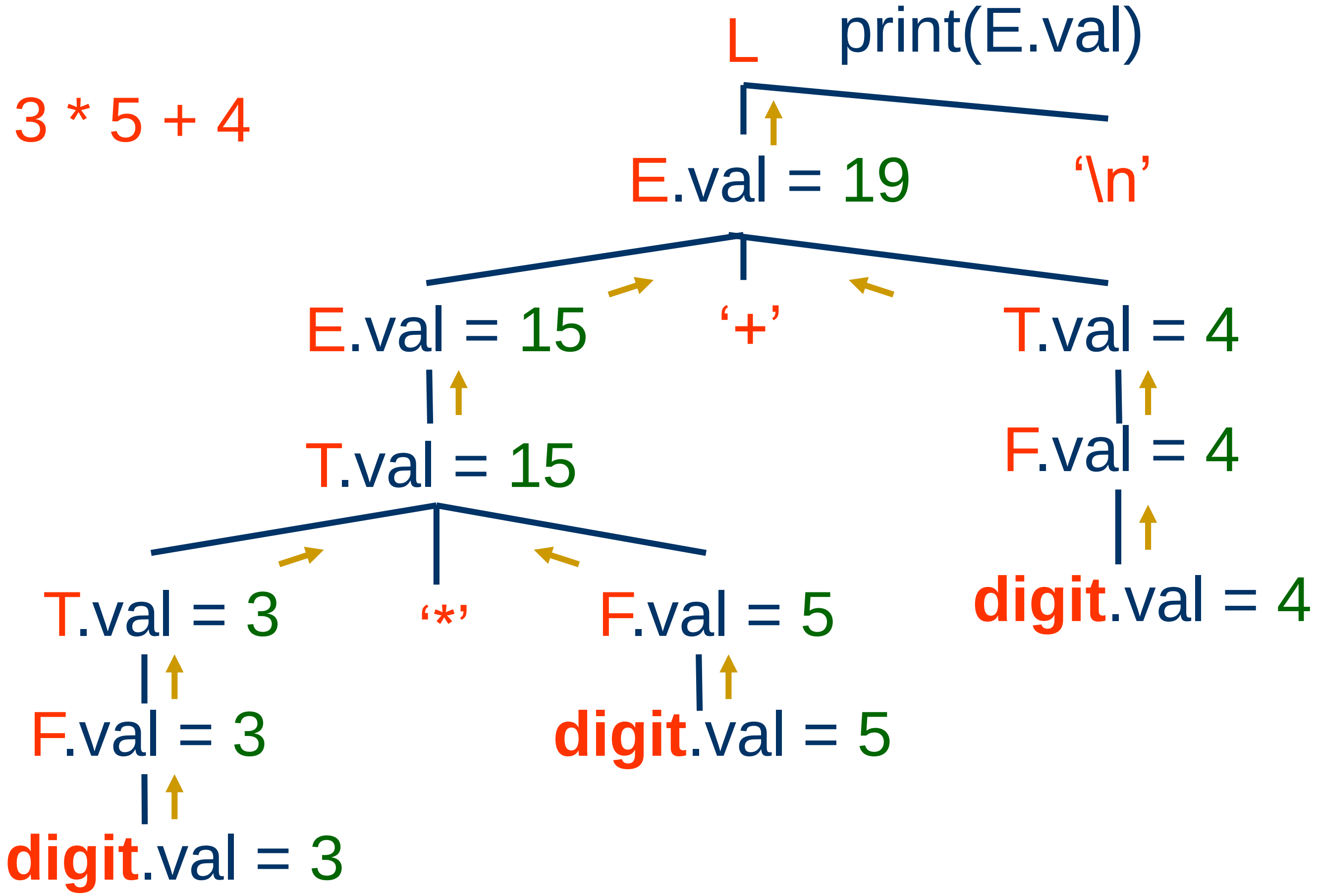
- Each node (grammar symbol) of the parse tree can have an associated set of semantic attributes representing semantics of the node
- An attribute of a node in the parse tree is synthesized if its value is computed from that of its children
- An attribute of a node in the parse tree is inherited if its 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

3 * 5 + 4



Inherited Attributes

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

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

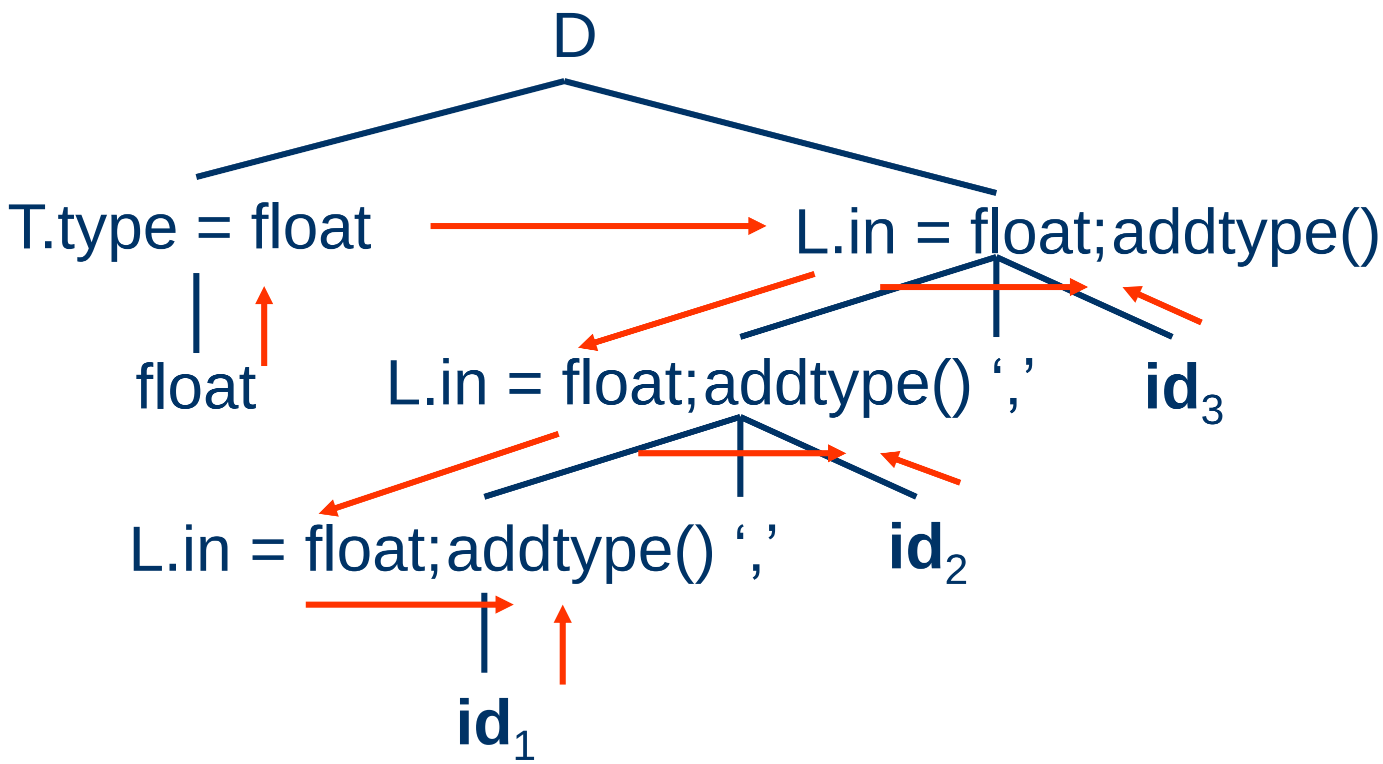
$T \rightarrow$ **float** {T.type := float;}

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

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

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

Inherited Attributes



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 $c_1, c_2, \ldots, c_k$

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

  $$A \rightarrow X_1\ X_2\ \dots\ X_n$$

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

  1. the attributes of X_1, X_2, …, $X_{j\text{-}1}$
  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);}

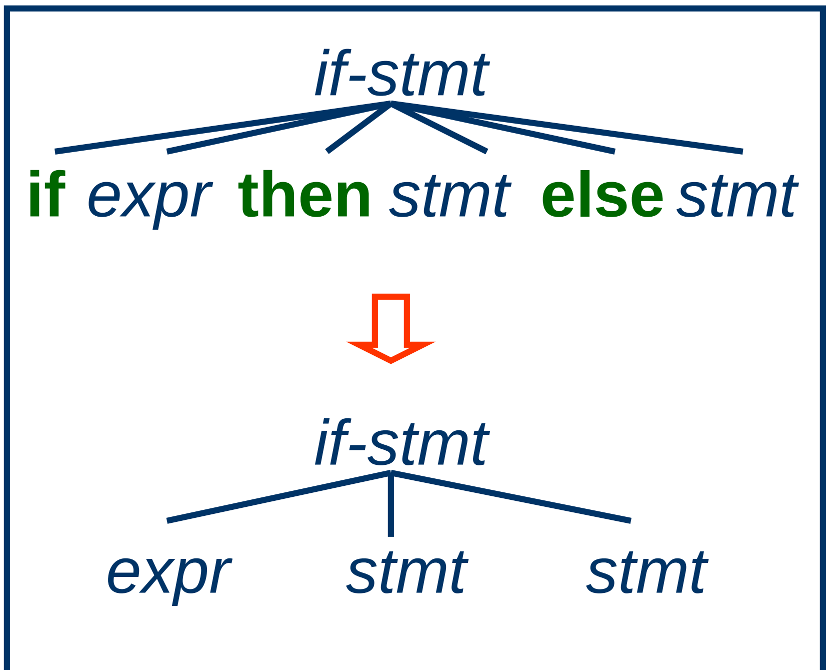
A Counter Example

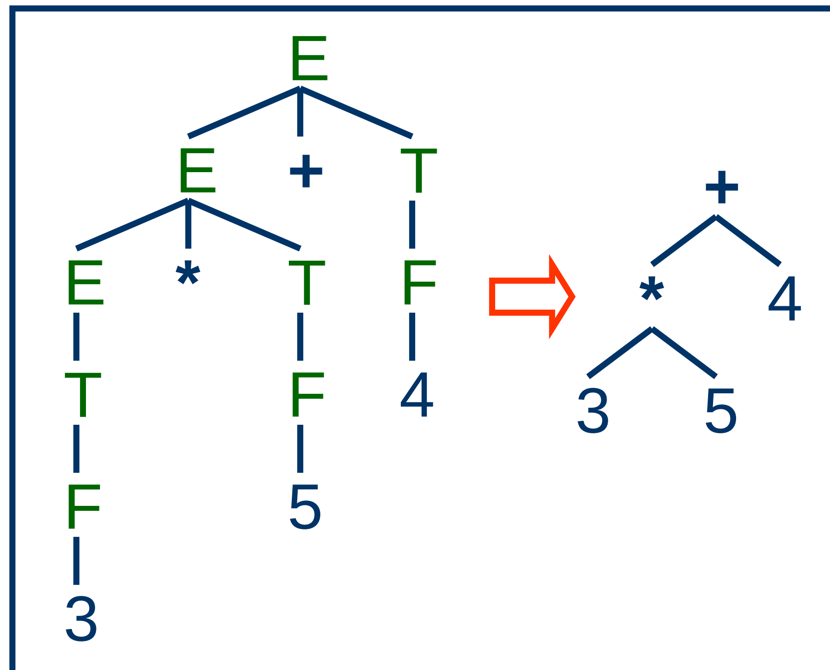
$A \rightarrow \{L.i := l(A.i);\}\ L$
$\{M.i := m(L.s);\}\ M$
$\{A.s := f(M.s);\}$

$A \rightarrow \{Q.i := q(R.s);\}\ Q$
$\{R.i := r(A.i);\}\ R$
$\{A.s := f(Q.s);\}$

Construction of Syntax Trees

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



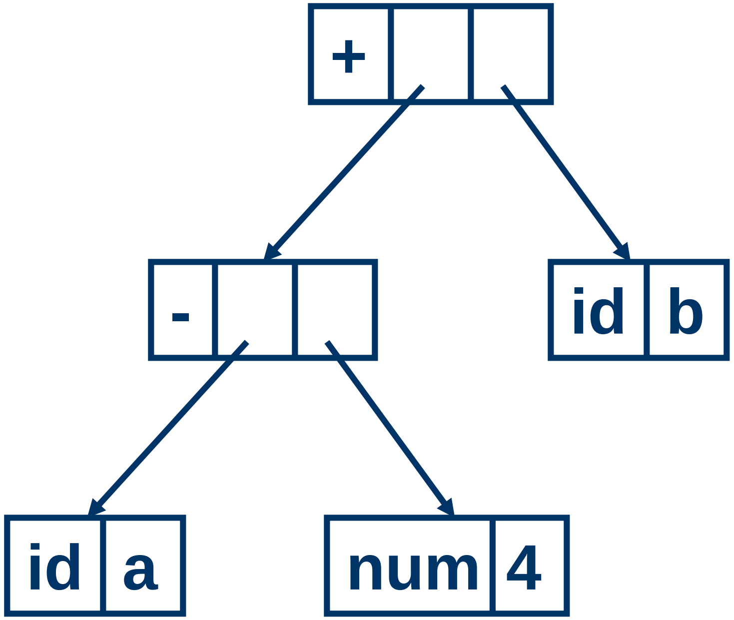


Syntax Trees for Expressions

- 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);}

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 - Translation

$E \rightarrow T\ R$

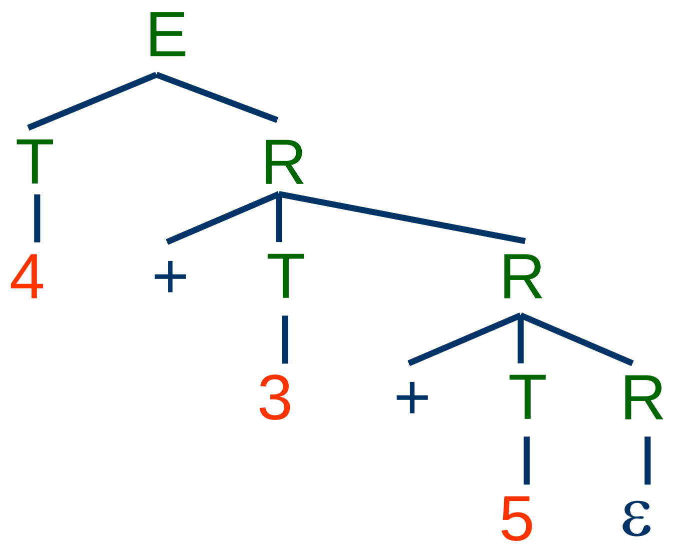
$T \rightarrow$ '(' E ')'

$T \rightarrow$ **num**

$R \rightarrow$ '+' $T\ R_1$

$R \rightarrow \varepsilon$

4 + 3 + 5



An Example - Translation

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

$R \rightarrow$ '+' T
{ R_1.i := mknode(**add**, 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 '+':
            add = yylval; match('+');
            nptr = T(); i1 = mknode(add, i, nptr);
              /* R1.i := mknode(add, 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;
}
```

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

$P \rightarrow D$ ";" S

$S \rightarrow$ **id** ":=" E

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

$S \rightarrow$ **if** E **then** S_1

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

$S \rightarrow$ **while** E **do** S_1

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

$S \rightarrow S_1$ ";" 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}

ANTLR Semantic Rules

- ANTLR semantic rules can be embedded in the parser rules as actions.
- Each action is a code block in the target language.
- Actions are executed immediately after the preceding rule element and immediately before the following rule element.

An Example

decl : INT ID {addtype($ID.text, “int”);} ‘;’
;

Arguments and Return Values

- Inherited attributes can be specified in the ANTLR parser rules as arguments
- Synthesized attributes can be specified in the ANTLR parser rules as return values
- Return values can have multiple values

An Example

```
r [int a, String b] returns [int c, String d] :
  …  { $c = $a; $d = $b; }
;

s : …
   v = r [3, “test”]
   { System.out.println($v.d); }
```

An Example

```
decl : a = type vars[$a.t] ‘;’
         ;

type returns [String t] :
           INT {$t = “int”;}
         | FLOAT {$t = “float”;}
         ;

vars [String t] : ID {addtype($ID.text, $t);} vars1[$t]
vars1 [String t] : ‘,’ ID {addtype($ID.text, $t);} vars1[$t]
         |
         ;
```