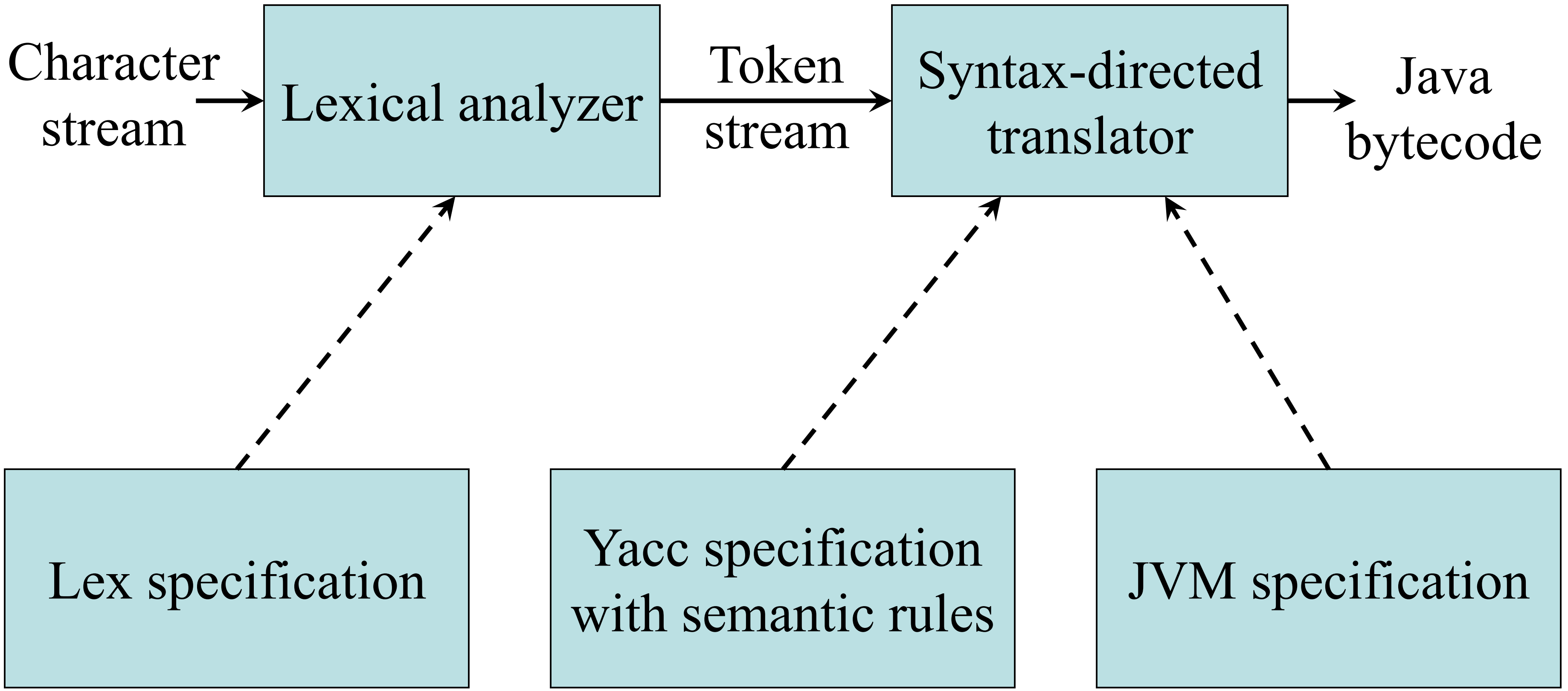
Syntax-Directed Translation

The Structure of our Compiler Revisited



Syntax-Directed Definitions

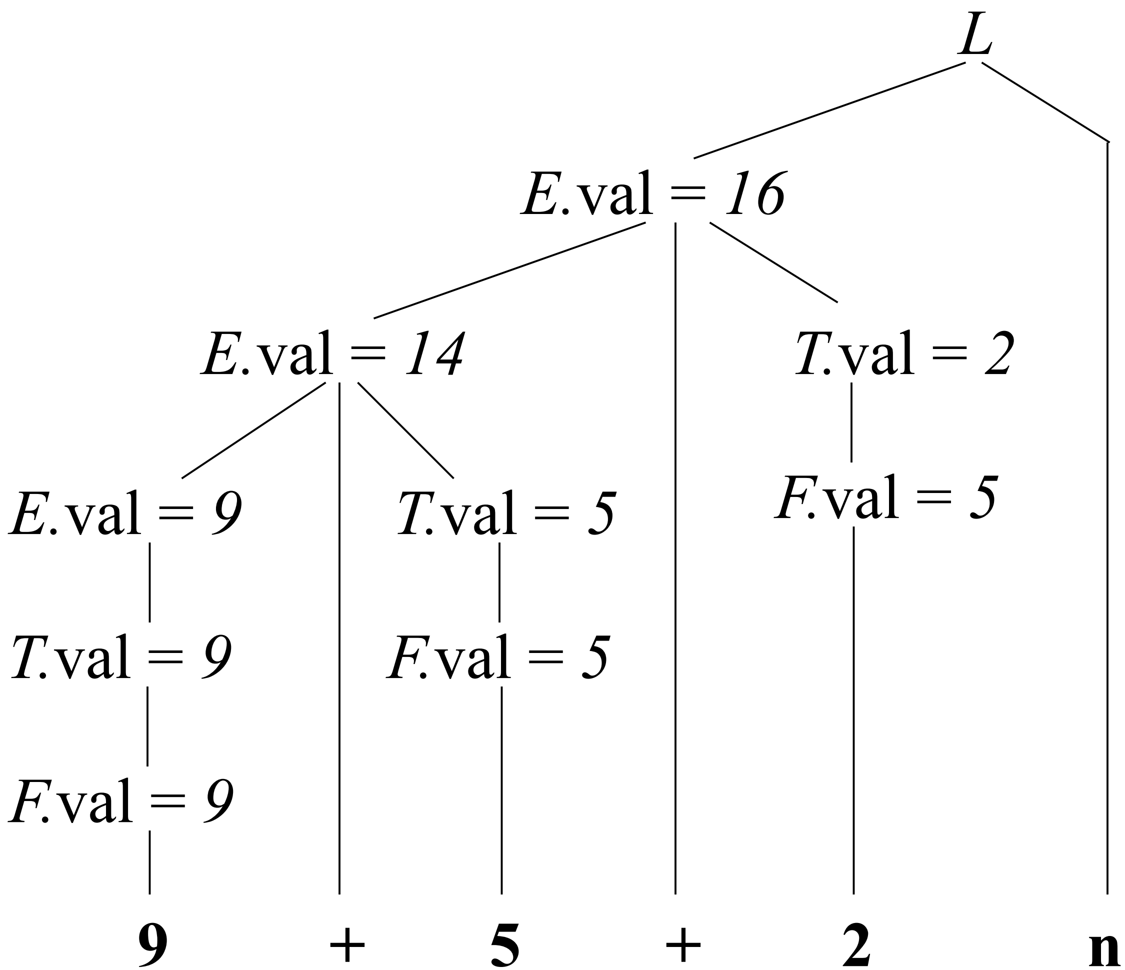
- A *syntax-directed definition* (or *attribute grammar*) binds a set of *semantic rules* to productions
- Terminals and nonterminals have *attributes* holding values set by the semantic rules
- A *depth-first traversal* algorithm traverses the parse tree thereby executing semantic rules to assign attribute values
- After the traversal is complete the attributes contain the translated form of the input

Example Attribute Grammar

Production	Semantic Rule
$L \to E\ \mathbf{n}$	$\mathit{print}(E.\mathrm{val})$
$E \to E_1 + T$	$E.\mathrm{val} := E_1.\mathrm{val} + T.\mathrm{val}$
$E \to T$	$E.\mathrm{val} := T.\mathrm{val}$
$T \to T_1 * F$	$T.\mathrm{val} := T_1.\mathrm{val} * F.\mathrm{val}$
$T \to F$	$T.\mathrm{val} := F.\mathrm{val}$
$F \to \mathbf{(}\ E\ \mathbf{)}$	$F.\mathrm{val} := E.\mathrm{val}$
$F \to \mathbf{digit}$	$F.\mathrm{val} := \mathbf{digit}.\mathrm{lexval}$

Note: all attributes in this example are of the synthesized type

Example Annotated Parse Tree

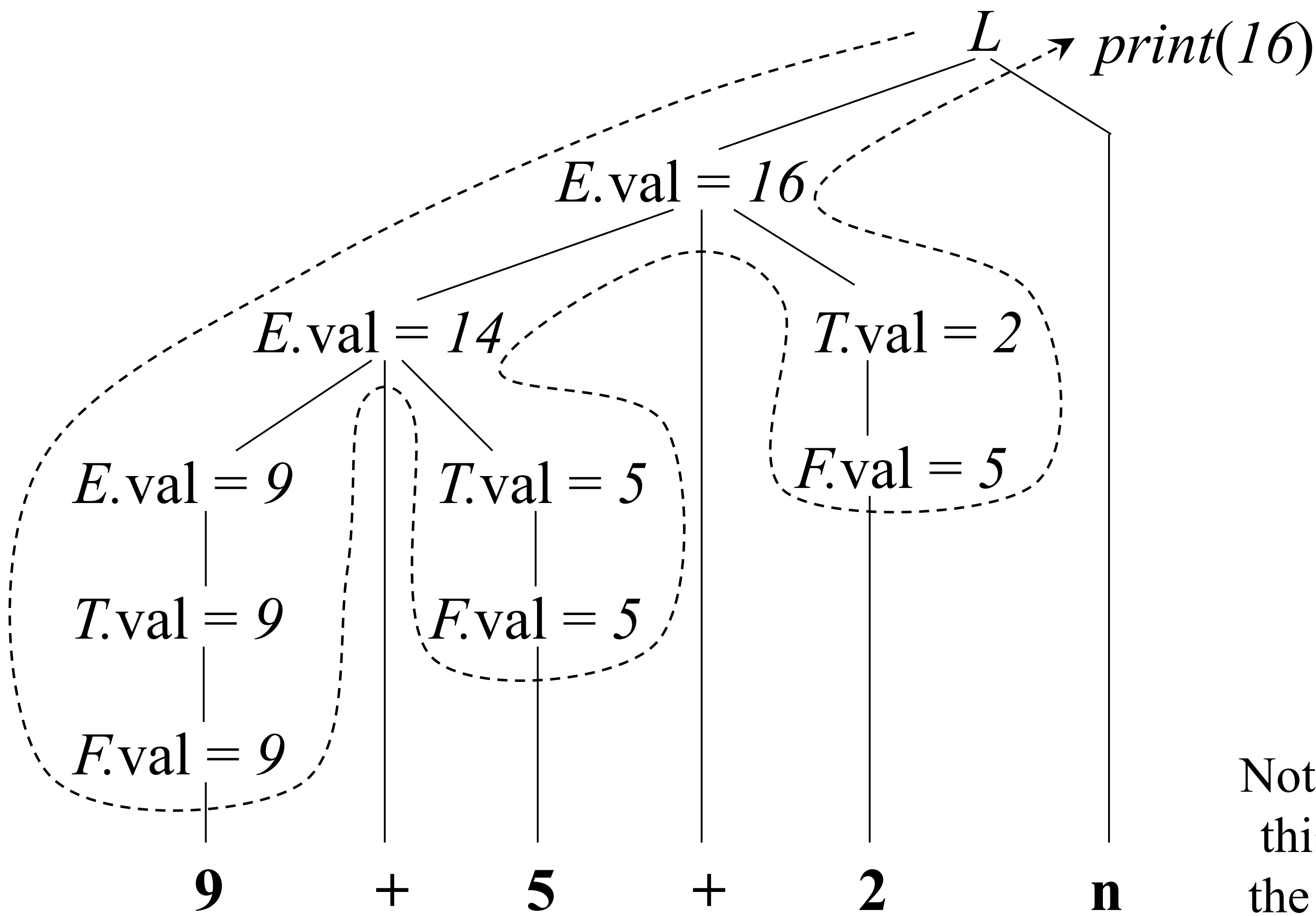


Note: all attributes in this example are of the synthesized type

Annotating a Parse Tree With Depth-First Traversals

procedure *visit*(*n* : *node*);
begin
    for each child *m* of *n*, from left to right **do**
        visit(*m*);
    evaluate semantic rules at node *n*
end

Depth-First Traversals (Example)



Note: all attributes in this example are of the synthesized type

Attributes

- Attribute values typically represent
  - Numbers (literal constants)
  - Strings (literal constants)
  - Memory locations, such as a frame index of a local variable or function argument
  - A data type for type checking of expressions
  - Scoping information for local declarations
  - Intermediate program representations

Synthesized Versus Inherited Attributes

- Given a production

  $$A \to \alpha$$

  then each semantic rule is of the form

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

  where f is a function and c_i are attributes of A and α, and either
  - b is a *synthesized* attribute of A
  - b is an *inherited* attribute of one of the grammar symbols in α

Synthesized Versus Inherited Attributes (cont’d)

inherited

Production	Semantic Rule
$D \to T\,L$	L.in := T.type
$T \to$ **int**	T.type := ‘integer’
...	...
$L \to$ **id**	… := L.in

synthesized

int id

S-Attributed Definitions

- A syntax-directed definition that uses synthesized attributes exclusively is called an *S-attributed definition* (or *S-attributed grammar*)
- A parse tree of an S-attributed definition is annotated with a single bottom-up traversal
- Yacc/Bison only support S-attributed definitions

Example Attribute Grammar in Yacc

```
%token DIGIT
%%
L : E ‘\n’              { printf(“%d\n”, $1); }
  ;
E : E ‘+’ T             { $$ = $1 + $3; }
  | T                   { $$ = $1; }
  ;
T : T ‘*’ F             { $$ = $1 * $3; }
  | F                   { $$ = $1; }
  ;
F : ‘(’ E ‘)’           { $$ = $2; }
  | DIGIT               { $$ = $1; }
  ;
%%
```

Synthesized attribute of parent node **F**

Bottom-up Evaluation of S-Attributed Definitions in Yacc

Stack	val	Input	Action	Semantic Rule
$	_	**3*5+4n$**	shift	
$ 3	*3*	***5+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *F*	*3*	***5+4n$**	reduce $T \rightarrow F$	$$ = $1
$ *T*	*3*	***5+4n$**	shift	
$ *T* *	*3 _*	**5+4n$**	shift	
$ *T* * **5**	*3 _ 5*	**+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *T* * *F*	*3 _ 5*	**+4n$**	reduce $T \rightarrow T * F$	$$ = $1 * $3
$ *T*	*15*	**+4n$**	reduce $E \rightarrow T$	$$ = $1
$ *E*	*15*	**+4n$**	shift	
$ *E* +	*15 _*	**4n$**	shift	
$ *E* + **4**	*15 _ 4*	**n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *E* + *F*	*15 _ 4*	**n$**	reduce $T \rightarrow F$	$$ = $1
$ *E* + *T*	*15 _ 4*	**n$**	reduce $E \rightarrow E + T$	$$ = $1 + $3
$ *E*	*19*	**n$**	shift	
$ *E* **n**	*19 _*	**$**	reduce $L \rightarrow E$ **n**	*print* $1
$ *L*	*19*	**$**	accept	

Example Attribute Grammar with Synthesized+Inherited Attributes

Production	Semantic Rule
$D \rightarrow T\ L$	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)

Synthesized: T.type, **id**.entry
Inherited: L.in

E→E+T {printf("+");}

| T {}

T→T*F {printf("*");}

| F {}

F→id {printf(id.lexval)}

2+3*4 234*+

E→E + T

T T * F

F F id

id id 4

2 3

$S \rightarrow xxW$ {print(1);}

$\quad | \; y$ {print(2);}

$W \rightarrow Sz$ {print(3);}

I/P:xxxxyzz

O/P: **Find out**

S→x x W

S z

x x W

S z

y

E→E*T {E.val=E.val*T.val}

| T {E.val=T.val}

T→F-T {T.val=F.val-T.val}

| F {T.val=F.val}

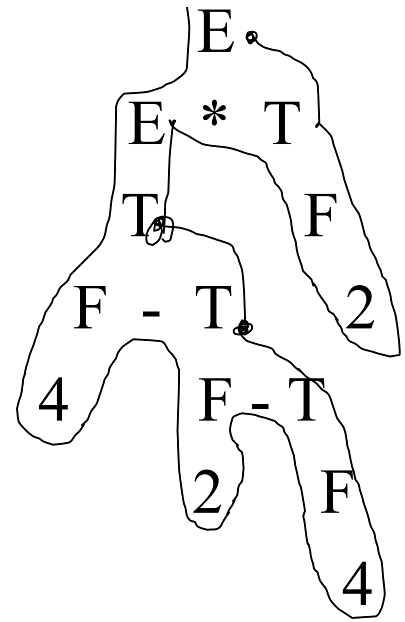
F→2 {F.val=2}

|4 {F.val=4}

(4 – (2 – 4)) * 2

12

reductions: 10



$E \rightarrow E\#T$ {E.val=E.val*T.val}

$\quad | \; T$ {E.val=T.val}

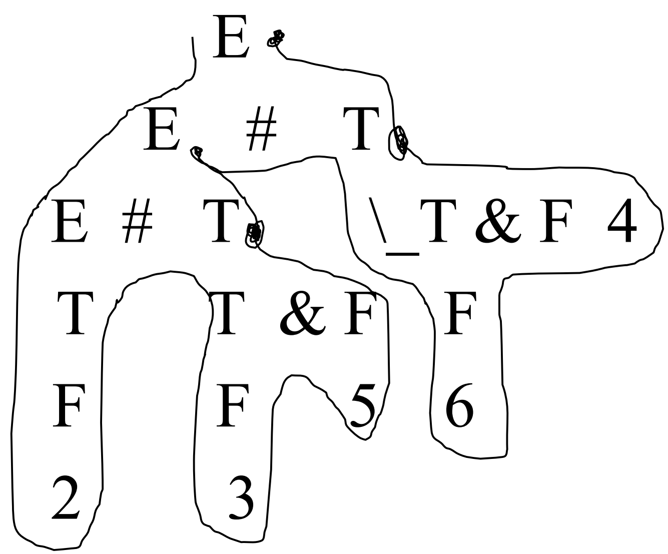
$T \rightarrow T\&F$ {T.val=T.val+F.val}

$\quad | \; F$ {T.val=F.val}

$F \rightarrow num$ {F.val=num.lexval}

2#3&5#6&4

(2*(3+5))*(6+4)



```
E→E+T  {E.ptr=mknode(E.ptr,’+’,T.ptr)}
  | T  {E.ptr=T.ptr}
T→T*F  {T.ptr=mknode(T.ptr,’*’,F.ptr}
  | F  {T.ptr=F.ptr}
F→id   {F.ptr=mknode(null,id.name,null)}
```

id1+id2*id3

$N \rightarrow L \quad \{N.val=L.val\}$

$L \rightarrow L_1B \quad \{L.val=L_1.val*2+B.val\}$

$\quad |B \quad \{L.val=B.val\}$

$B \rightarrow 0 \quad \{B.val=0\}$

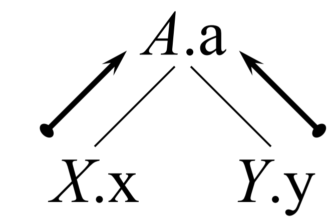
$\quad |1 \quad \{B.val=1\}$

010100 = 20

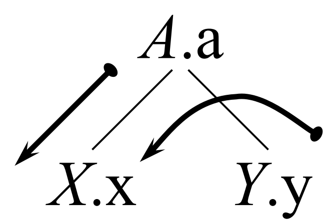
```
N20  -    L20
        L10  B0
      L5  B0
     L2  B1
    L1  B0
   L0  B1
   B0
```

Acyclic Dependency Graphs for Attributed Parse Trees

$A \to X\ Y$



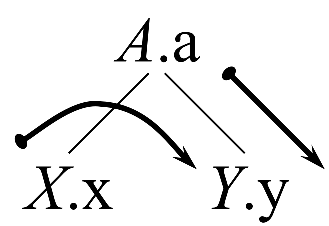
$A.\mathrm{a} := f(X.\mathrm{x},\ Y.\mathrm{y})$



$X.\mathrm{x} := f(A.\mathrm{a},\ Y.\mathrm{y})$

Direction of

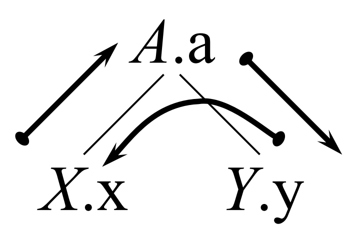
value dependence



$Y.\mathrm{y} := f(A.\mathrm{a},\ X.\mathrm{x})$

Dependency Graphs with Cycles?

- Edges in the dependency graph determine the evaluation order for attribute values
- Dependency graphs cannot be cyclic



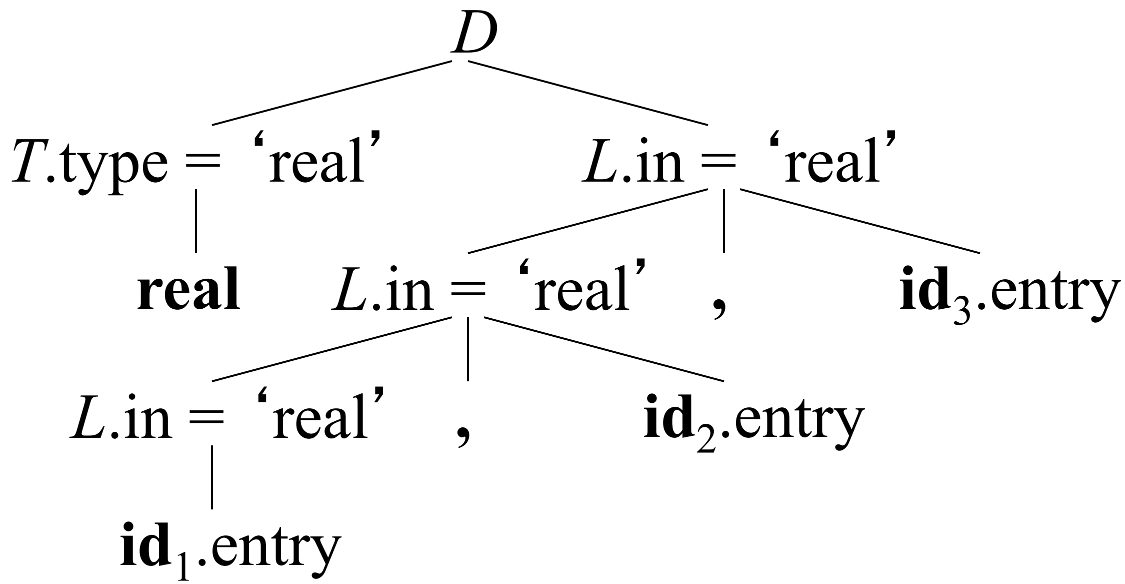
$$A.\mathrm{a} := f(X.\mathrm{x})$$
$$X.\mathrm{x} := f(Y.\mathrm{y})$$
$$Y.\mathrm{y} := f(A.\mathrm{a})$$

Error: cyclic dependence

- $A \rightarrow X1\ X2\ X3\ \ldots\ Xn$
- $X_i.in=f(A.in,X_1.x,X_2.y,\ldots X_{i-1}.z)$

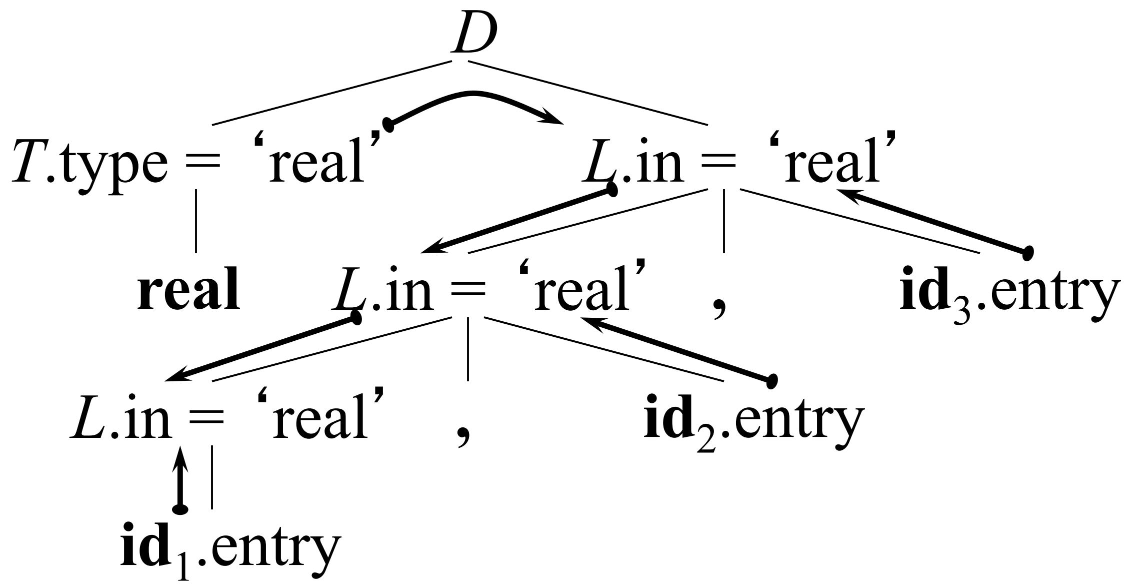
Example Annotated Parse Tree

$D \rightarrow T\ L$
$T \rightarrow$ **int**
$T \rightarrow$ **real**
$L \rightarrow L_1$ **, id**
$L \rightarrow$ **id**



float id1,id2,id3
float a,b,c (a,real) (b,real) …

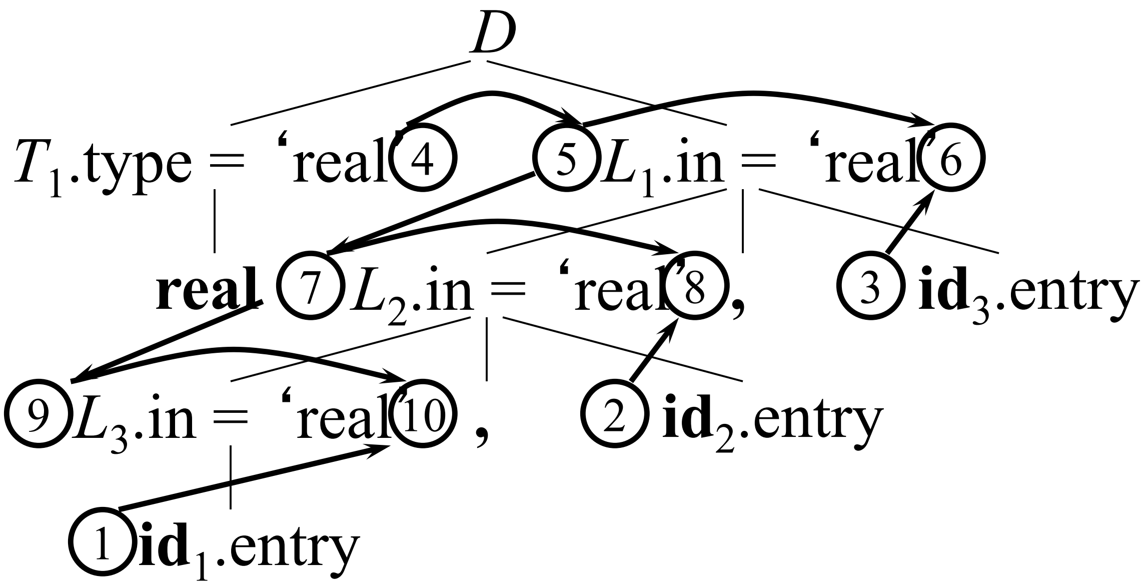
Example Annotated Parse Tree with Dependency Graph



Evaluation Order

- A *topological sort* of a directed acyclic graph (DAG) is any ordering $m_1, m_2, \ldots, m_n$ of the nodes of the graph, such that if $m_i \rightarrow m_j$ is an edge, then m_i appears before m_j
- Any topological sort of a dependency graph gives a valid evaluation order of the semantic rules

Example Parse Tree with Topologically Sorted Actions



Topological sort:

1. Get **id**$_1$.entry
2. Get **id**$_2$.entry
3. Get **id**$_3$.entry
4. T_1.type= 'real'
5. L_1.in=T_1.type
6. *addtype*(**id**$_3$.entry, L_1.in)
7. L_2.in=L_1.in
8. *addtype*(**id**$_2$.entry, L_2.in)
9. L_3.in=L_2.in
10. *addtype*(**id**$_1$.entry, L_3.in)

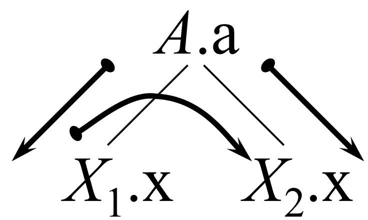
Evaluation Methods

- *Parse-tree methods* determine an evaluation order from a topological sort of the dependence graph constructed from the parse tree for each input
- *Rule-base methods* the evaluation order is pre-determined from the semantic rules
- *Oblivious methods* the evaluation order is fixed and semantic rules must be (re)written to support the evaluation order (for example S-attributed definitions)

L-Attributed Definitions

- The example parse tree on slide 18 is traversed "in order", because the direction of the edges of inherited attributes in the dependency graph point top-down and from left to right
- More precisely, a syntax-directed definition is *L-attributed* if each <u>inherited</u> attribute of X_j on the right side of $A \to X_1\ X_2\ \ldots\ X_n$ depends only on
  1. the attributes of the symbols $X_1, X_2, \ldots, X_{j\text{-}1}$
  2. the inherited attributes of A

Shown: dependences of inherited attributes



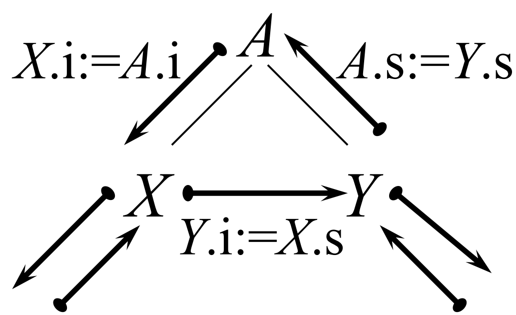
$A \rightarrow X1, X2, X3, \ldots, X_n$

$X_i.in = f(X_1.x,\ X_2.y, \ldots X_{i-1}.z, A.p)$

L-Attributed Definitions (cont'd)

- L-attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right

$A \to X\ Y$

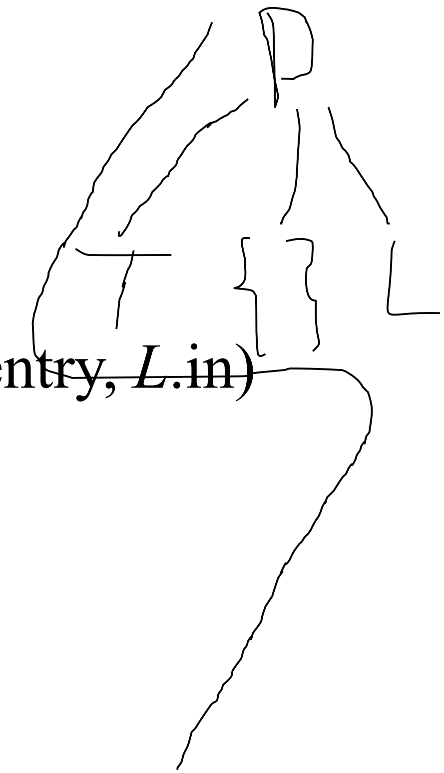


X.i := A.i
Y.i := X.s
A.s := Y.s

- Note: every S-attributed syntax-directed definition is also L-attributed

Using Translation Schemes for L-Attributed Definitions

Production	Semantic Rule
$D \rightarrow T\ L$	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)



Translation Scheme

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

$T \rightarrow$ **int** { T.type := 'integer' }

$T \rightarrow$ **real** { T.type := 'real' }

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

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

$X \rightarrow \{1\}A\ \{2\}B\ \{3\}C\ \{4\}$

i i i s

A=X X.s=f

Implementing L-Attributed Definitions in Top-Down Parsers

Attributes in L-attributed definitions implemented in translation schemes are passed as arguments to procedures (synthesized) or returned (inherited)

$D \rightarrow T \; \{ \; L.\text{in} := T.\text{type} \; \} \; L$
$T \rightarrow \textbf{int} \; \{ \; T.\text{type} := \text{`integer'} \; \}$
$T \rightarrow \textbf{real} \; \{ \; T.\text{type} := \text{`real'} \; \}$

```
void D()
{ Type Ttype = T();
  Type Lin = Ttype;
  L(Lin);
}
Type T()
{ Type Ttype;
  if (lookahead == INT)
  { Ttype = TYPE_INT;
    match(INT);
  } else if (lookahead == REAL)
  { Ttype = TYPE_REAL;
    match(REAL);
  } else error();
  return Ttype;
}
void L(Type Lin)
{ … }
```

Output: synthesized attribute

Input: inherited attribute

Implementing Translation Schemes in Bottom-Up Parsers

- Insert marker nonterminals to remove embedded actions from translation schemes, that is

  $A \to X$ { actions } Y

  is rewritten with marker nonterminal N into

  $A \to X\, N\, Y$

  $N \to \varepsilon$ { actions }
- Problem: inserting a marker nonterminal may introduce a conflict in the parse table
- Problem: how to propagate inherited attributes?

- A→M1 X1 M2 X2

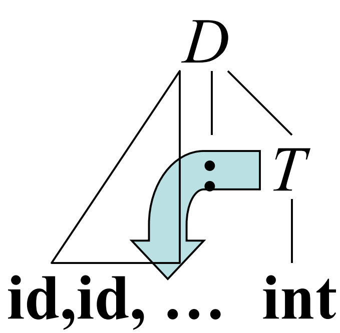
Emulating the Evaluation of L-Attributed Definitions in Yacc

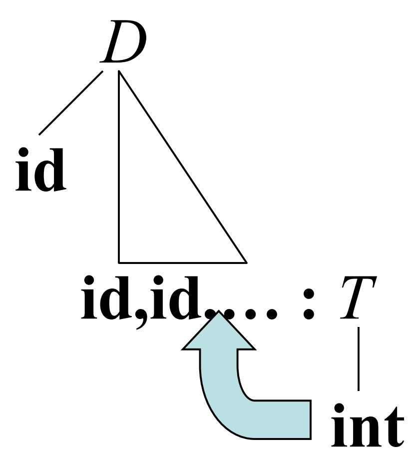
$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ **int** { T.type := 'integer' }
$T \rightarrow$ **real** { T.type := 'real' }
$L \rightarrow$ { L_1.in := L.in } L_1 **, id**
{ *addtype*(**id**.entry, L.in) }
$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

```
%{
Type Lin; /* global variable */
%}
%%
D  : Ts L
   ;
Ts : T              { Lin = $1; }
   ;
T  : INT            { $$ = TYPE_INT; }
   | REAL           { $$ = TYPE_REAL; }
   ;
L  : L ',' ID { addtype($3, Lin);}
   | ID             { addtype($1, Lin);}
   ;
%%
```

Rewriting a Grammar to Avoid Inherited Attributes

Production	Production	Semantic Rule
$D \rightarrow L$ **:** T	$D \rightarrow$ **id** L	*addtype*(**id**.entry, L.type)
$T \rightarrow$ **int**	$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$ **, id**	$L \rightarrow$ **, id** L_1	*addtype*(**id**.entry, L.type)
$L \rightarrow$ **id**	$L \rightarrow$ **:** T	L.type := T.type





Translation Schemes using Marker Nonterminals

Need a stack to keep track of gotos to backpatch!
(to handle nested if-then)

$S \rightarrow$ **if** E {push(pc); emit(**`ifeq`**, 0) }
then S { backpatch(top(), pc-top()); pop() }

Insert marker nonterminal

Synthesized attribute
(automatically stacked in shift-reduce parser!)

$S \rightarrow$ **if** E M **then** S { backpatch(M.loc, pc-M.loc) }
$M \rightarrow \varepsilon$ { M.loc := pc; emit(**`ifeq`**, 0) }

xy XY X_1

	Production	Semantic Rules
1)	$S \to B$	$B.ps = 10$
2)	$B \to B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
3)	$B \to B_1$ **sub** B_2	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4)	$B \to (\ B_1\ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5)	$B \to$ **text**	$B.ht = getHt(B.ps, \textbf{text}.lexval)$ $B.dp = getDp(B.ps, \textbf{text}.lexval)$

$X \rightarrow \{A.in\}A\ \{B.in\}B\ \{C.in\}C\ \ \{\text{synthesized Attr}\}$

$A \rightarrow X_1\, X_2\, X_3 \ldots X_n$

$X_i.in = f(A.in, X_1.x, X_2.y, \ldots X_{i-1}.k)$

	Production			Actions
1)	S	$\to$		$\{\ B.ps = \ 10;\ \}$
			B	
2)	B	$\to$		$\{\ B_1.ps = \ B.ps;\ \}$
			B_1	$\{\ B_2.ps = \ B.ps;\ \}$
			B_2	$\{\ B.ht = \ \max(B_1.ht, B_2.ht);$
				$B.dp = \ \max(B_1.dp, B_2.dp);\ \}$
3)	B	$\to$		$\{\ B_1.ps = \ B.ps;\ \}$
			B_1 **sub**	$\{\ B_2.ps = \ 0.7 \times B.ps;\ \}$
			B_2	$\{\ B.ht = \ \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$
				$B.dp = \ \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	B	$\to$	(	$\{\ B_1.ps = \ B.ps;\ \}$
			B_1)	$\{\ B.ht = \ B_1.ht;$
				$B.dp = \ B_1.dp;\ \}$
5)	B	$\to$	**text**	$\{\ B.ht = \ getHt(B.ps, \textbf{text}.lexval);$
				$B.dp = \ getDp(B.ps, \textbf{text}.lexval);\ \}$

S→{B.ps=10}B

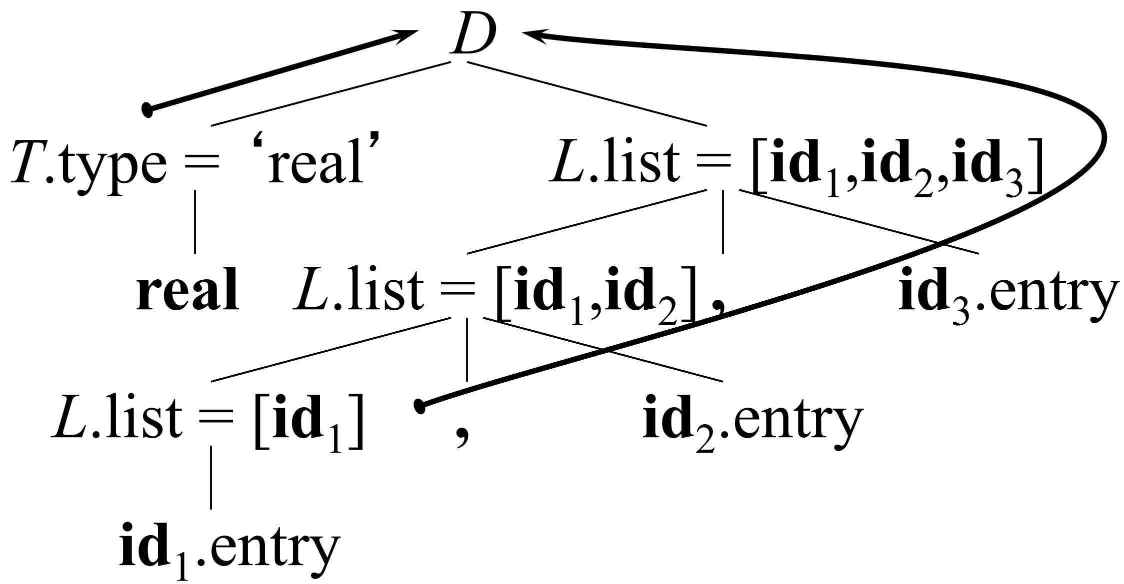
B→{B1.ps=B.ps}B1

Translation Schemes using Marker Nonterminals in Yacc

```
S : IF E M THEN S { backpatch($3, pc-$3); }
  ;
M : /* empty */   { $$ = pc;
                    emit3(ifeq, 0);
                  }
  ;
…
```

Replacing Inherited Attributes with Synthesized Lists

$D \rightarrow T\, L$ { for all **id** $\in L$.list : *addtype*(**id**.entry, T.type) }
$T \rightarrow$ **int** { T.type := 'integer' }
$T \rightarrow$ **real** { T.type := 'real' }
$L \rightarrow L_1$ **, id** { L.list := L_1.list + [**id**] }
$L \rightarrow$ **id** { L.list := [**id**] }



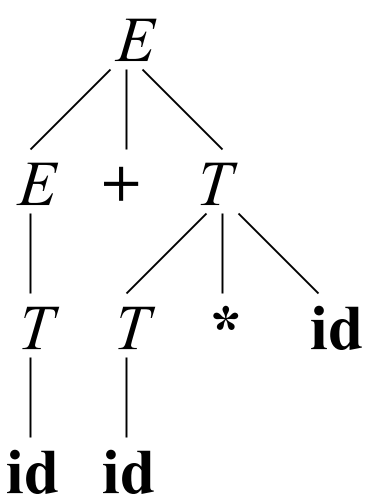
Replacing Inherited Attributes with Synthesized Lists in Yacc

```
%{
typedef struct List
{ Symbol *entry;
  struct List *next;
} List;
%}

%union
{ int type;
  List *list;
  Symbol *sym;
}

%token <sym> ID
%type <list> L
%type <type> T

%%
```

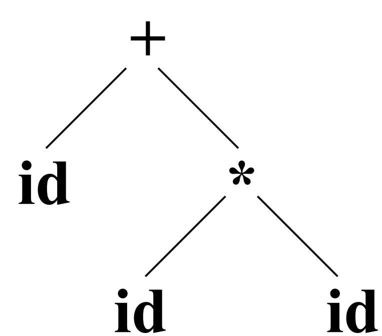
```
D : T L  { List *p;
           for (p = $2; p; p = p->next)
             addtype(p->entry, $1);
         }
  ;
T : INT  { $$ = TYPE_INT; }
  | REAL { $$ = TYPE_REAL; }
  ;
L : L ',' ID
         { $$ = malloc(sizeof(List));
           $$->entry = $3;
           $$->next = $1;
         }
  | ID   { $$ = malloc(sizeof(List));
           $$->entry = $1;
           $$->next = NULL;
         }
  ;
```

Concrete and Abstract Syntax Trees

- A parse tree is called a *concrete syntax tree*
- An *abstract syntax tree* (AST) is defined by the compiler writer as a more convenient intermediate representation

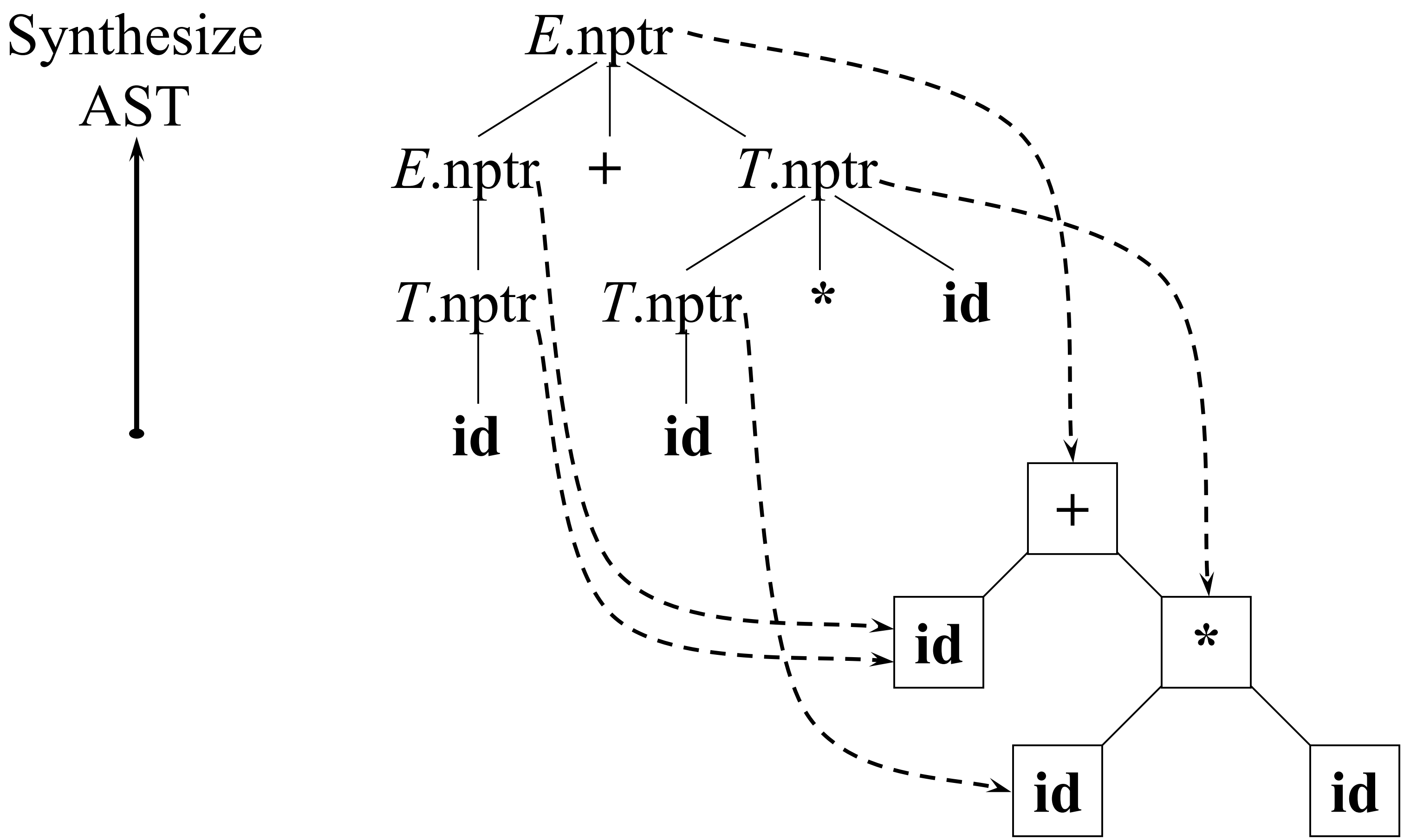


Concrete syntax tree



Abstract syntax tree

Generating Abstract Syntax Trees



S-Attributed Definitions for Generating Abstract Syntax Trees

Production	Semantic Rule
$E \rightarrow E_1$ **+** T	E.nptr := mknode('+' , E_1.nptr, T.nptr)
$E \rightarrow E_1$ **-** T	E.nptr := mknode('-' , E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow T_1$ *** id**	T.nptr := mknode('*' , T_1.nptr, mkleaf(**id**, **id**.entry))
$T \rightarrow T_1$ **/ id**	T.nptr := mknode('/' , T_1.nptr, mkleaf(**id**, **id**.entry))
$T \rightarrow$ **id**	T.nptr := mkleaf(**id**, **id**.entry)

Generating Abstract Syntax Trees with Yacc

```
%{
typedef struct Node
{ int op;     /* node op */
  Symbol *entry; /* leaf */
  struct Node *left, *right;
} Node;
%}

%union
{ Node *node;
  Symbol *sym;
}

%token <sym> ID
%type <node> E T F

%%
```

```
E : E '+' T   { $$ = mknode('+', $1, $3); }
  | E '-' T   { $$ = mknode('-', $1, $3); }
  | T         { $$ = $1; }
  ;
T : T '*' F   { $$ = mknode('*', $1, $3); }
  | T '/' F   { $$ = mknode('/', $1, $3); }
  | F         { $$ = $1; }
  ;
F : '(' E ')' { $$ = $2; }
  | ID        { $$ = mkleaf($1); }
  ;

%%
```