

Syntax-Directed Translation

Part II

Chapter 5

Translation Schemes using Marker Nonterminals

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

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

Insert marker nonterminal

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

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

Translation Schemes using Marker Nonterminals in Yacc

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

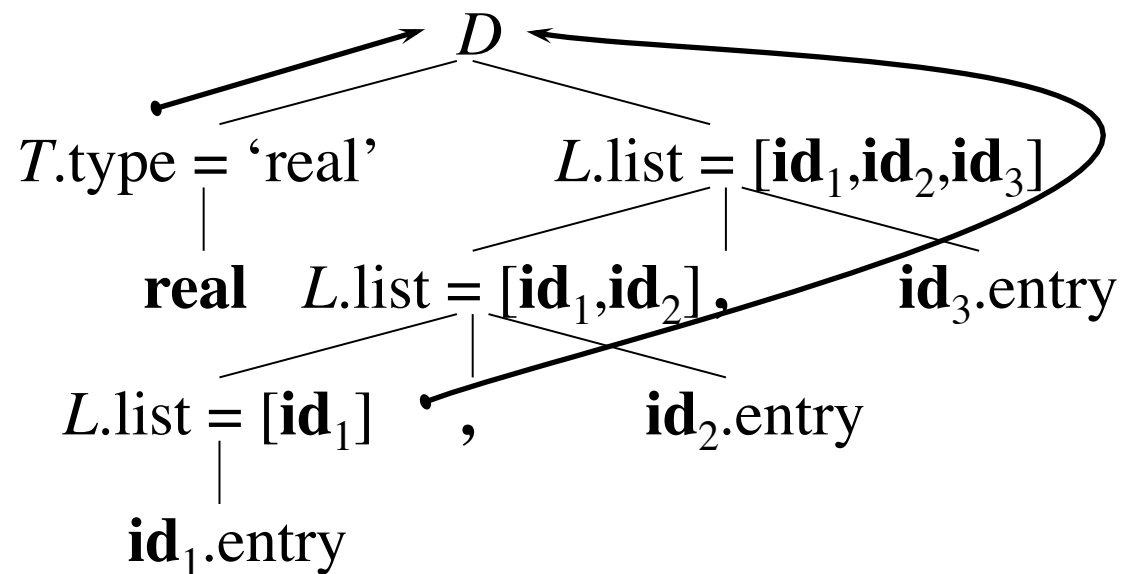
Replacing Inherited Attributes with Synthesized Lists

$$D \rightarrow T L \{ \text{for all } \mathbf{id} \in L.\text{list} : \text{addtype}(\mathbf{id}.\text{entry}, T.\text{type}) \}$$

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

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

$$L \rightarrow L_1 , \mathbf{id} \{ L.\text{list} := L_1.\text{list} + [\mathbf{id}] \}$$

$$L \rightarrow \mathbf{id} \{ L.\text{list} := [\mathbf{id}] \}$$


Replacing Inherited Attributes with Synthesized Lists in Yacc

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

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

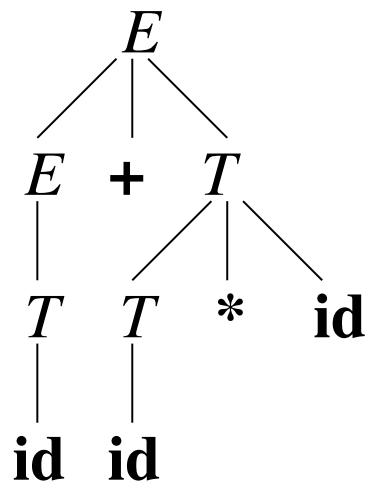
%token <sym> ID
%type <list> L
%type <type> T

%%
```

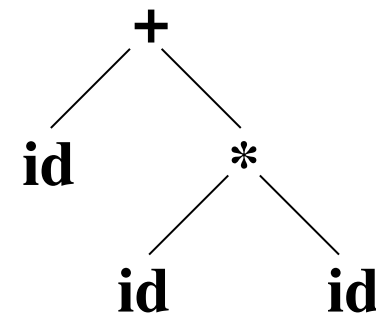
```
D : T L { List *p;
        for (p = $2; p; p = p->next)
            addtype(p->entry, $1);
        }
;
T : INT { $$ = TYPE_INT; }
  | REAL { $$ = TYPE_REAL; }
;
L : L ',' ID
    { $$ = malloc(sizeof(List));
      $$->entry = $3;
      $$->next = $1;
    }
  | ID { $$ = malloc(sizeof(List));
    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

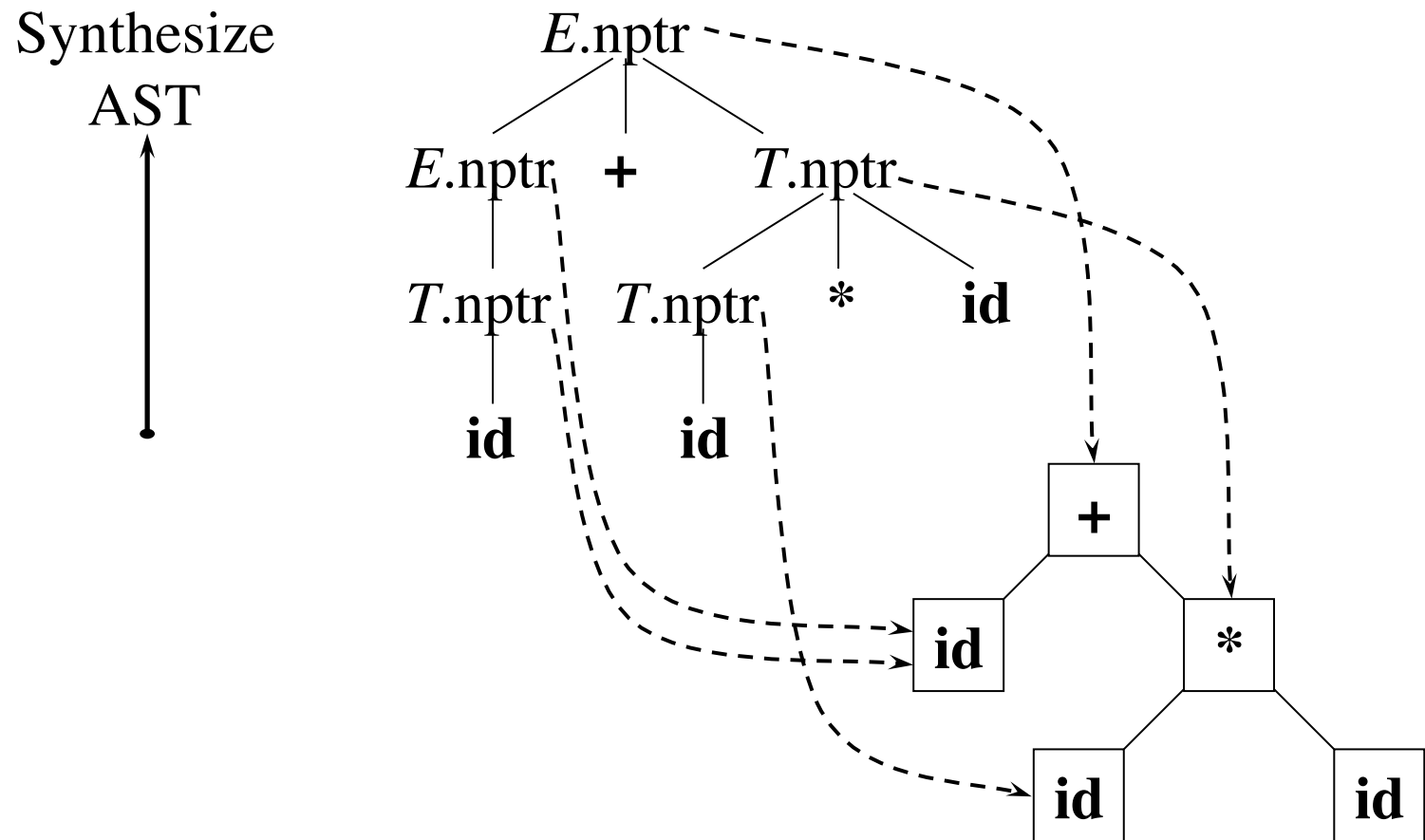


Concrete syntax tree



Abstract syntax tree

Generating Abstract Syntax Trees



S-Attributed Definitions for Generating Abstract Syntax Trees

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

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

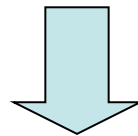
%%
E : E '+' T    { $$ = mknode('+', $1, $3); }
  | E '-' T    { $$ = mknode('-', $1, $3); }
  | T          { $$ = $1; }
  ;
T : T '*' F    { $$ = mknode('*', $1, $3); }
  | T '/' F    { $$ = mknode('/', $1, $3); }
  | F          { $$ = $1; }
  ;
F : '(' E ')'  { $$ = $2; }
  | ID         { $$ = mkleaf($1);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

%%

```

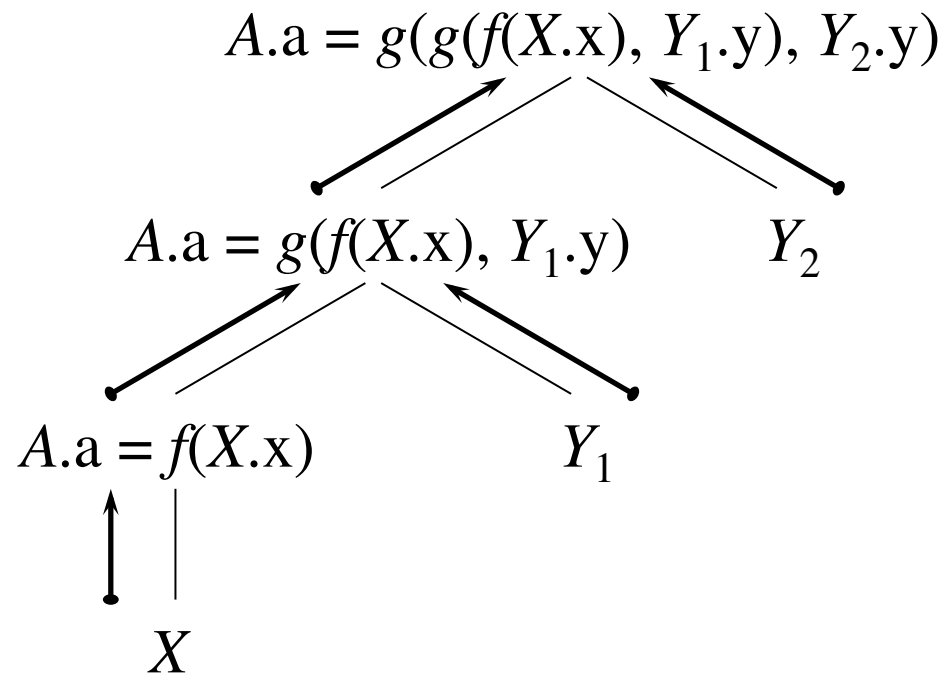
Eliminating Left Recursion from a Translation Scheme

$$\begin{array}{ll}
 A \rightarrow A_1 Y & \{ A.a := g(A_1.a, Y.y) \} \\
 A \rightarrow X & \{ A.a := f(X.x) \}
 \end{array}$$

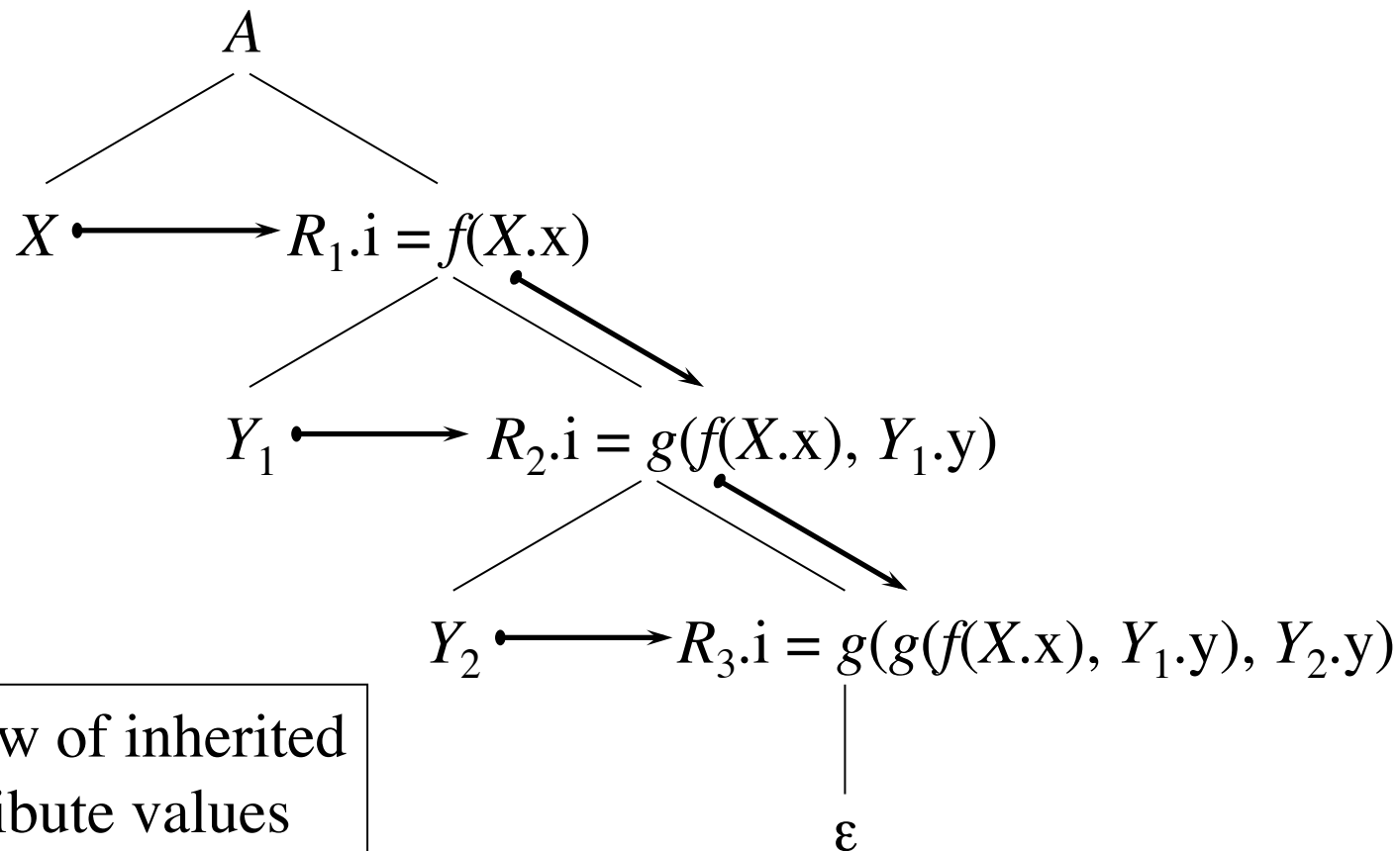


$$\begin{array}{ll}
 A \rightarrow X & \{ R.i := f(X.x) \} \quad R \quad \{ A.a := R.s \} \\
 R \rightarrow Y & \{ R_1.i := g(R.i, Y.y) \} \quad R_1 \quad \{ R.s := R_1.s \} \\
 R \rightarrow \varepsilon & \{ R.s := R.i \}
 \end{array}$$

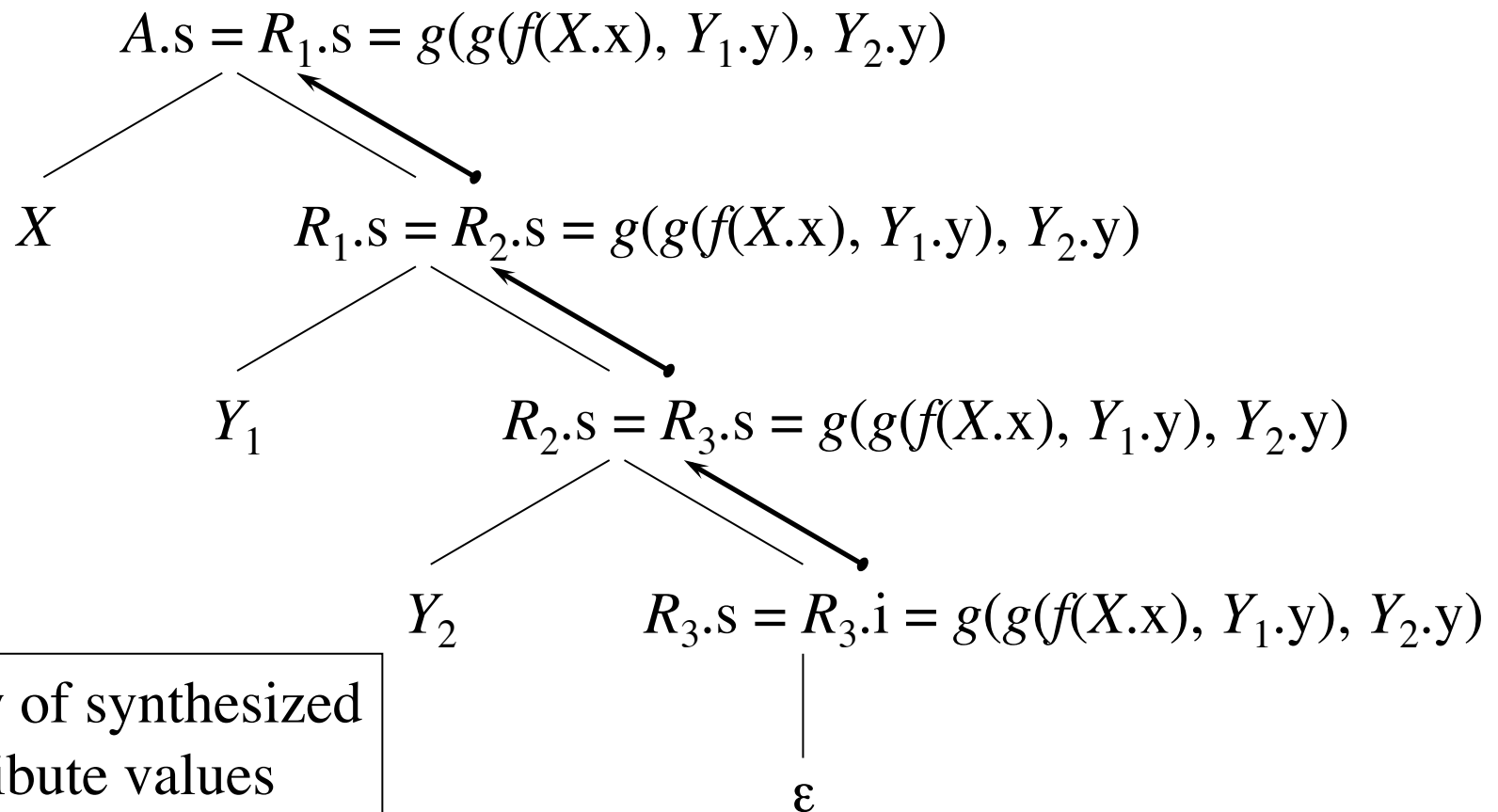
Eliminating Left Recursion from a Translation Scheme (cont'd)



Eliminating Left Recursion from a Translation Scheme (cont'd)



Eliminating Left Recursion from a Translation Scheme (cont'd)

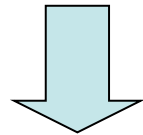


Generating Abstract Syntax Trees with Predictive Parsers

$$E \rightarrow E_1 + T \{ E.\text{nptr} := \text{mknode}('+', E_1.\text{nptr}, T.\text{nptr}) \}$$

$$E \rightarrow E_1 - T \{ E.\text{nptr} := \text{mknode}('-', E_1.\text{nptr}, T.\text{nptr}) \}$$

$$E \rightarrow T \{ E.\text{nptr} := T.\text{nptr} \}$$

$$T \rightarrow \mathbf{id} \{ T.\text{nptr} := \text{mkleaf}(\mathbf{id}, \mathbf{id}.\text{entry}) \}$$


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

$$R \rightarrow + T \{ R_1.i := \text{mknode}('+', R.i, T.\text{nptr}) \} R_1 \{ R.s := R_1.s \}$$

$$R \rightarrow - T \{ R_1.i := \text{mknode}('-', R.i, T.\text{nptr}) \} R_1 \{ R.s := R_1.s \}$$

$$R \rightarrow \varepsilon \{ R.s := R.i \}$$

$$T \rightarrow \mathbf{id} \{ T.\text{nptr} := \text{mkleaf}(\mathbf{id}, \mathbf{id}.\text{entry}) \}$$

Generating Abstract Syntax Trees with Predictive Parsers (cont'd)

```
Node *R(Node *i)
{ Node *s, *i1;
  if (lookahead == '+')
  { match('+');
    s = T();
    i1 = mknode('+', i, s);
    s = R(i1);
  } else if (lookahead == '-')
  { ...
  } else
    s = i;
  return s;
}
```