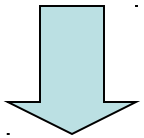
Syntax-Directed Translation Part II

Chapter 5

Translation Schemes using Marker Nonterminals

Need a stack!
(for nested if-then)

$S \rightarrow$ **if** E { emit(**`iconst_0`**); push(pc); emit(**`if_icmpeq`**, 0) }
then S { backpatch(top(), pc-top()); pop() }



Insert marker nonterminal

Synthesized attribute
(automatically stacked)

$S \rightarrow$ **if** E M **then** S { backpatch(M.loc, pc-M.loc) }

$M \rightarrow \varepsilon$ { emit(**`iconst_0`**); M.loc := pc; emit(**`if_icmpeq`**, 0) }

Translation Schemes using Marker Nonterminals in Yacc

```
S : IF E M THEN S { backpatch($3, pc-$3); }
  ;
M : /* empty */   { emit(iconst_0);
                    $$ = pc;
                    emit3(if_icmpeq, 0);
                  }
  ;
…
```

Replacing Inherited Attributes with Synthesized Lists

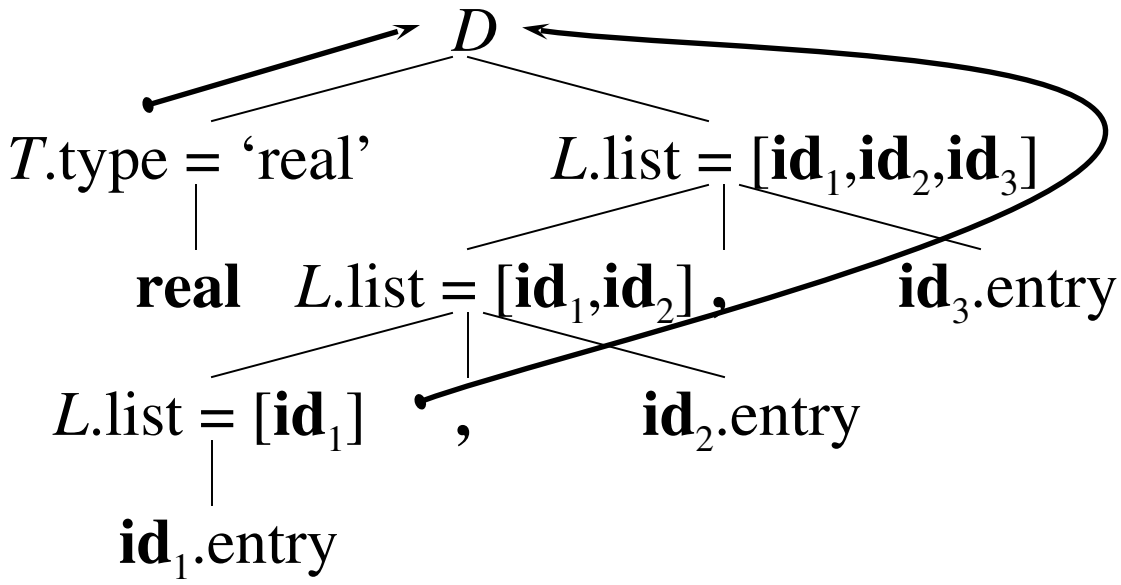
$D \rightarrow T\ L$ { for all **id** $\in$ L.list : *addtype*(**id**.entry, T.type) }

$T \rightarrow$ **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**] }



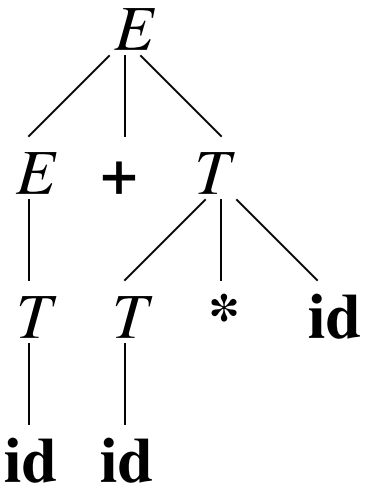
Replacing Inherited Attributes with Synthesized Lists in Yacc

```
%{
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

%%
```

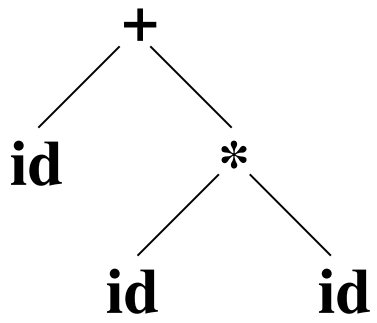
```
D : T L  { List *p;
           for (p = $2; p; p = p->next)
             addtype(p->entry, $1);
         }
  ;
T : INT  { $$ = TYPE_INT; }
  | REAL { $$ = TYPE_REAL; }
  ;
L : L ',' ID
         { $$ = malloc(sizeof(List));
           $$->entry = $3;
           $$->next = $1;
         }
  | ID   {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

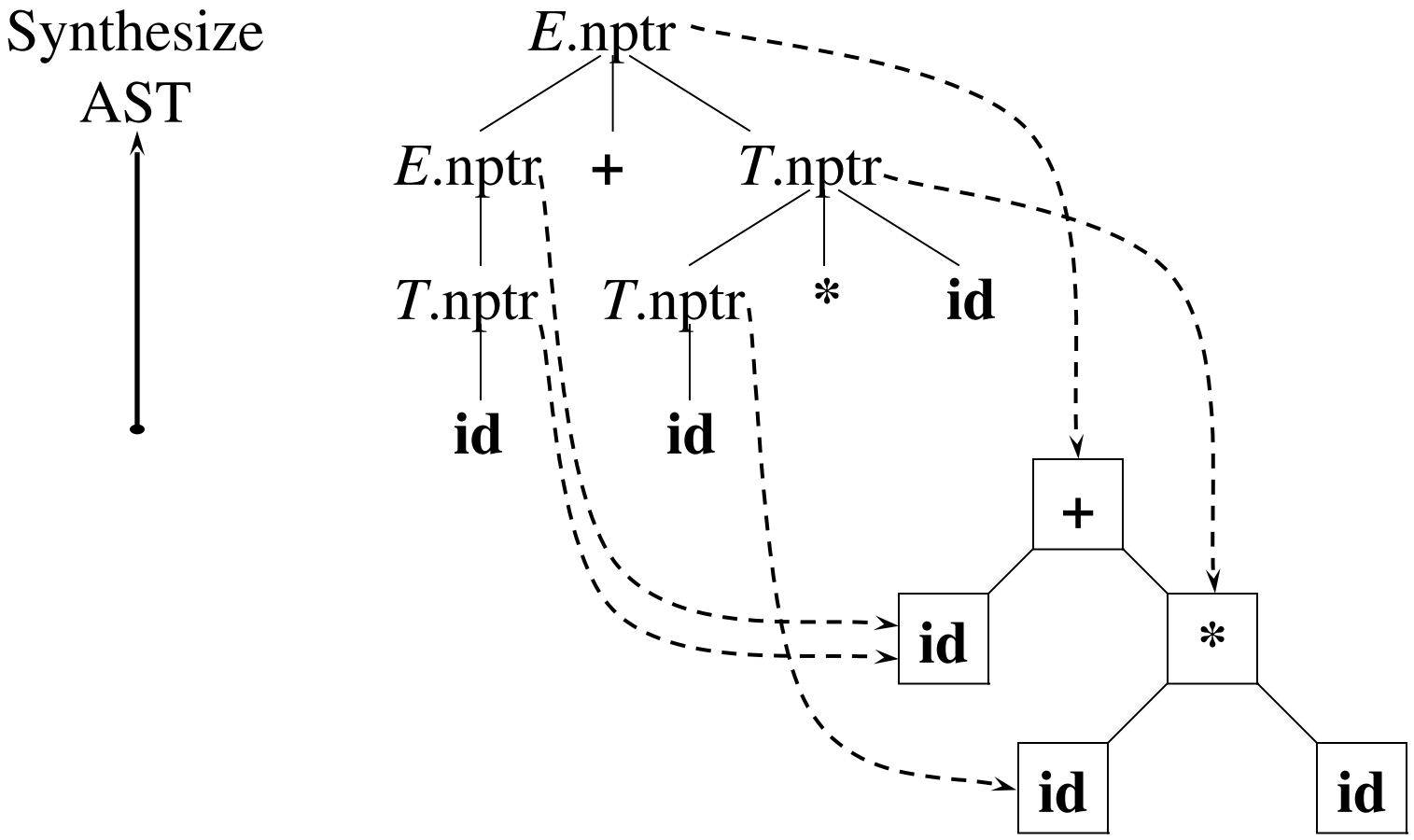


Concrete syntax tree



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

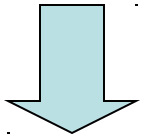
```
%{
typedef struct Node
{ int op;      /* node op */
  Symbol *entry; /* leaf */
  struct Node *left, *right;
} Node;
%}

%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); }
  ;

%%
```

Eliminating Left Recursion from a Translation Scheme

$A \rightarrow A_1\ Y \qquad \{\ A.\mathrm{a} := g(A_1.\mathrm{a},\ Y.\mathrm{y})\ \}$

$A \rightarrow X \qquad \{\ A.\mathrm{a} := f(X.\mathrm{x})\ \}$

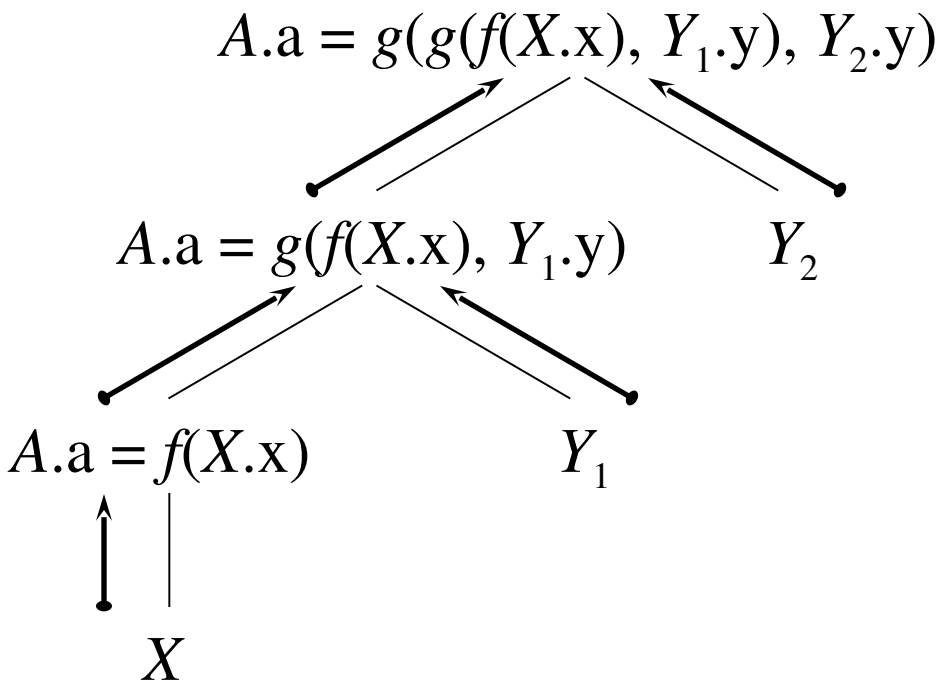


$A \rightarrow X\ \{\ R.\mathrm{i} := f(X.\mathrm{x})\ \}\ R\ \{\ A.\mathrm{a} := R.\mathrm{s}\ \}$

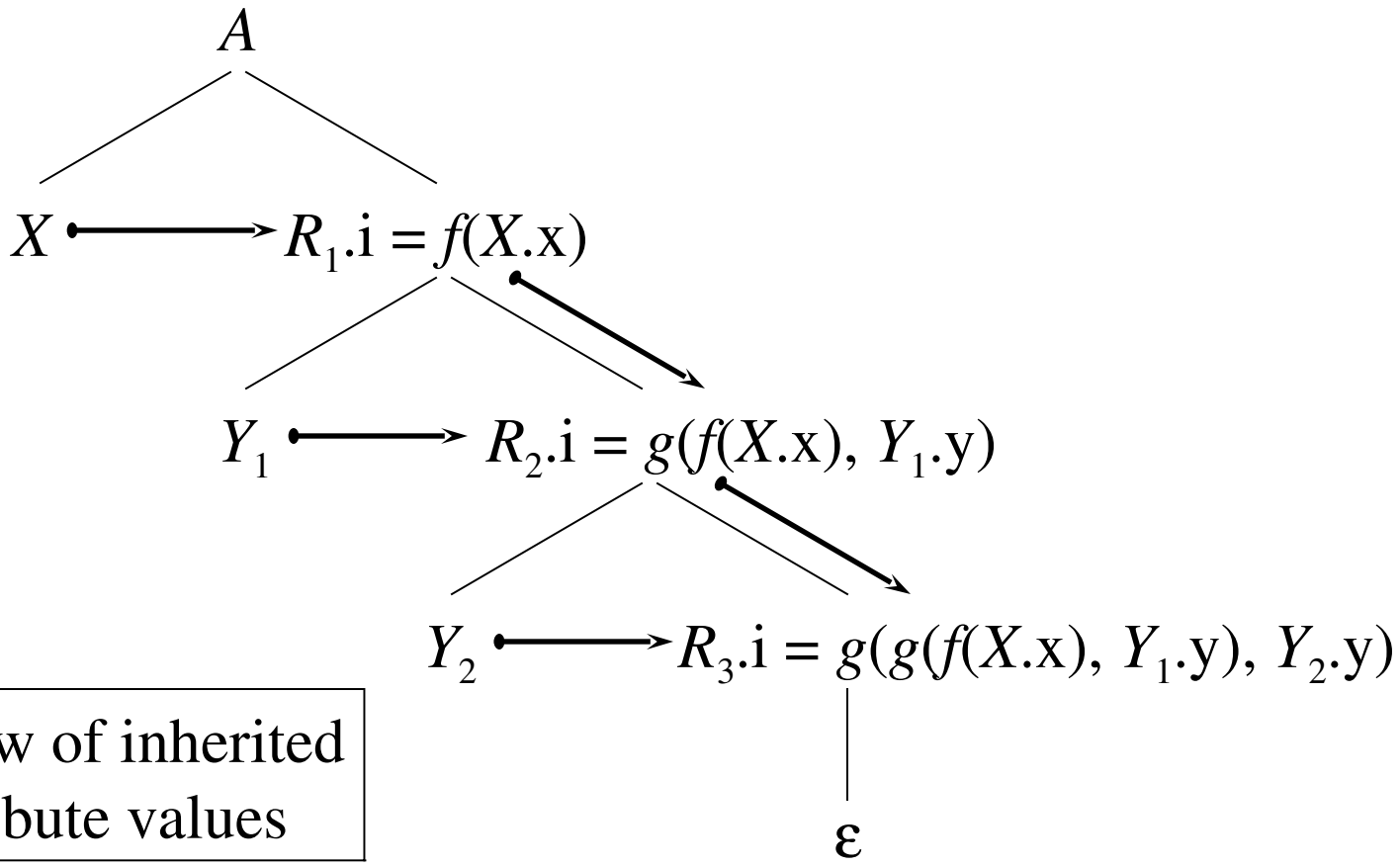
$R \rightarrow Y\ \{\ R_1.\mathrm{i} := g(R.\mathrm{i},\ Y.\mathrm{y})\ \}\ R_1\ \{\ R.\mathrm{s} := R_1.\mathrm{s}\ \}$

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

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

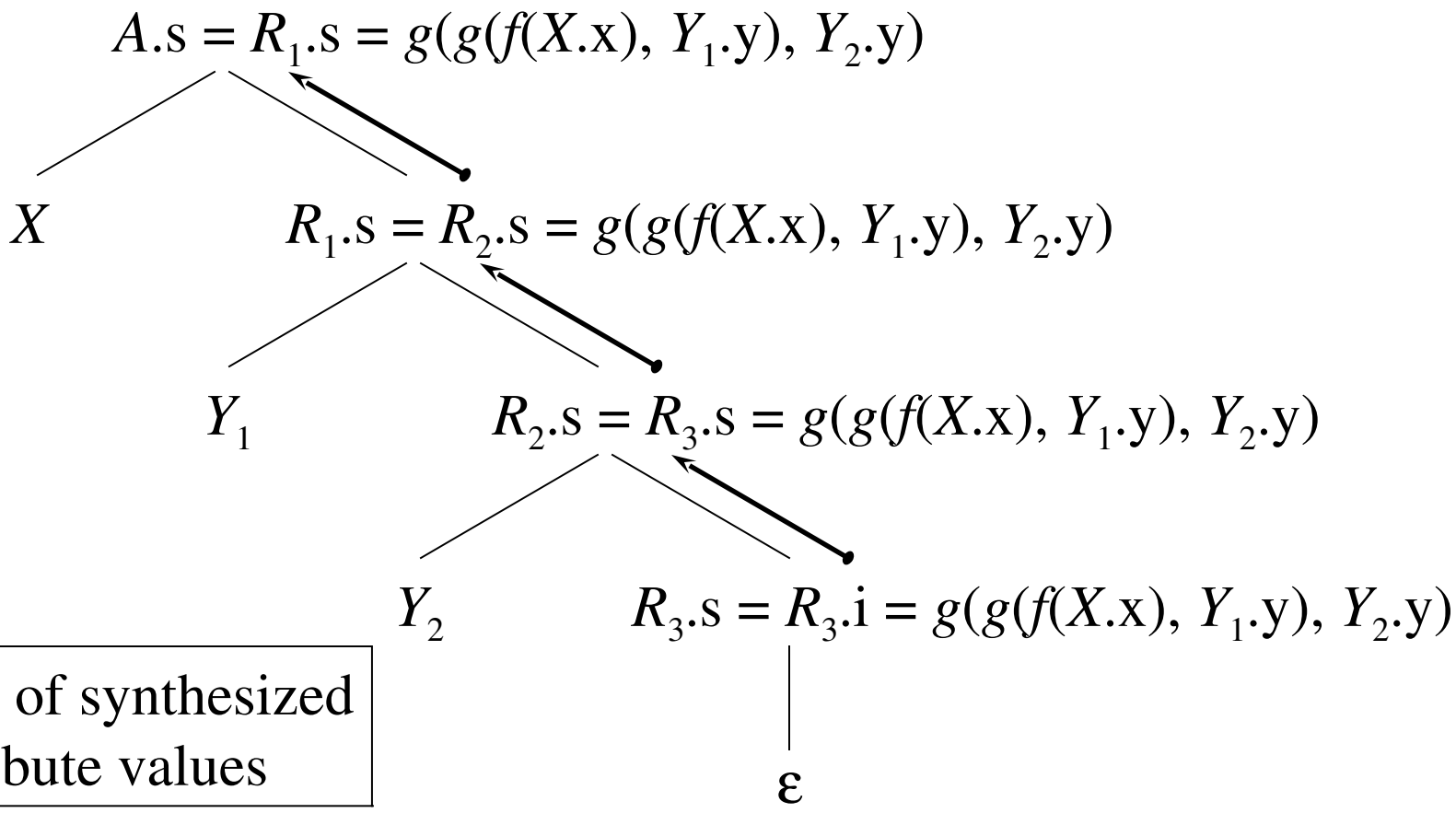


Eliminating Left Recursion from a Translation Scheme (cont'd)



1. Flow of inherited attribute values

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



2. Flow of synthesized attribute values

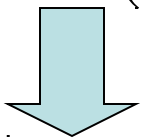
Generating Abstract Syntax Trees with Predictive Parsers

$E \rightarrow E_1$ **+** 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 }

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



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

$R \rightarrow$ **+** T {R_1.i := mknode(‘+’, R.i, T.nptr) } R_1 { R.s := R_1.s }

$R \rightarrow$ **-** T {R_1.i := mknode(‘-’, R.i, T.nptr) } R_1 { R.s := R_1.s }

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

$T \rightarrow$ **id** { T.nptr := mkleaf(**id**, **id**.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;
}
```