

# Syntax-Directed Translation

## Part II

### Chapter 5

COP5621 Compiler Construction  
Copyright Robert van Engelen, Florida State University, 2005

## Translation Schemes using Marker Nonterminals

Need a stack!  
(for nested if-then)

$$S \rightarrow \text{if } E \{ \text{emit}(\text{iconst\_0}); \text{push}(\text{pc}); \text{emit}(\text{if\_icmpeq}, 0) \} \\ \text{then } S \{ \text{backpatch}(\text{top}(), \text{pc-top}()); \text{pop}() \}$$

Insert marker nonterminal

Synthesized attribute  
(automatically stacked)

$$S \rightarrow \text{if } E M \text{ then } S \{ \text{backpatch}(M.\text{loc}, \text{pc-}M.\text{loc}) \} \\ M \rightarrow \epsilon \{ \text{emit}(\text{iconst\_0}); M.\text{loc} := \text{pc}; \text{emit}(\text{if\_icmpeq}, 0) \}$$

## 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);
                   }
  ;
...

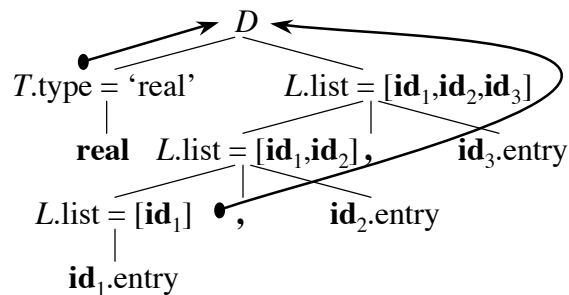
```

## Replacing Inherited Attributes with Synthesized Lists

```

D → TL { for all id ∈ L.list : addtype(id.entry, T.type) }
T → int { T.type := 'integer' }
T → real { T.type := 'real' }
L → L1, id { L.list := L1.list + [id] }
L → 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);
        }
;

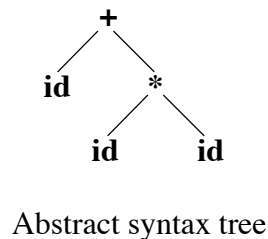
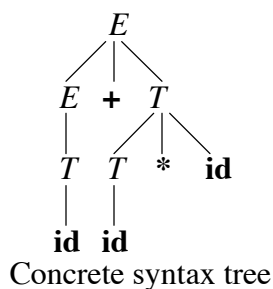
T : INT { $$ = TYPE_INT; }
  | REAL { $$ = TYPE_REAL; }
;

L : L ',' ID
  { $$ = malloc(sizeof(List));
    $$->entry = $3;
    $$->next = $1;
  }
  | ID { $$ = malloc(sizeof(List));
        $$->entry = $1;
        $$->next = NULL;
      }
;

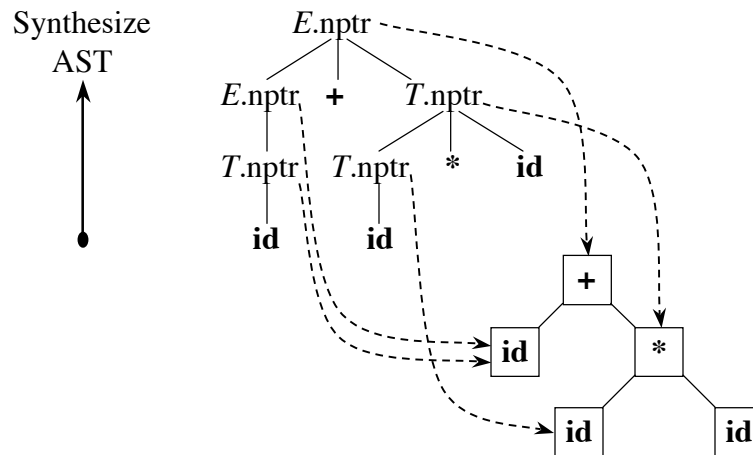
```

## 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



## Generating Abstract Syntax Trees



## S-Attributed Definitions for Generating Abstract Syntax Trees

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

## 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); }
  ;

%%

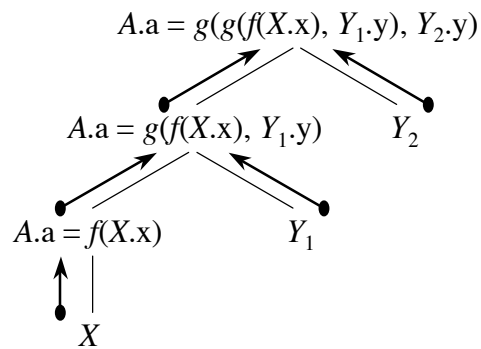
```

## Eliminating Left Recursion from a Translation Scheme

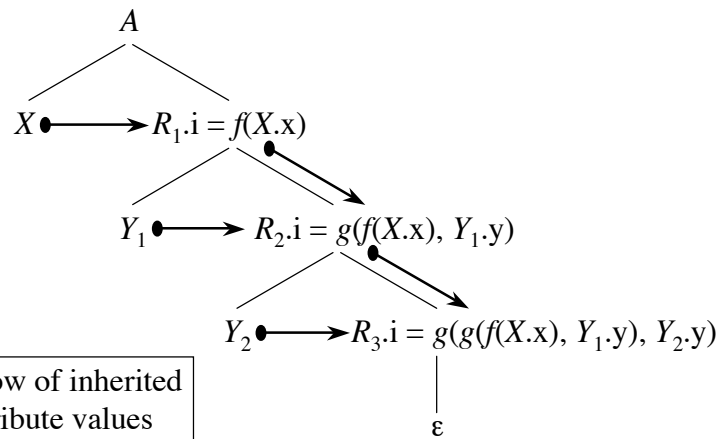
$$\begin{array}{l}
 A \rightarrow A_1 Y \quad \{ A.a := g(A_1.a, Y.y) \} \\
 A \rightarrow X \quad \quad \{ A.a := f(X.x) \}
 \end{array}$$


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

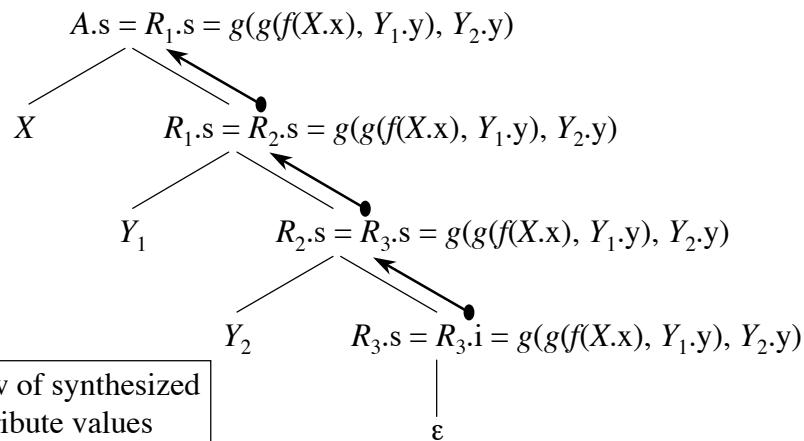
## Eliminating Left Recursion from a Translation Scheme (cont'd)



## 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

$$\begin{aligned}
 E &\rightarrow E_1 + T \{ E.nptr := \text{mknode}('+', E_1.nptr, T.nptr) \} \\
 E &\rightarrow E_1 - T \{ E.nptr := \text{mknode}('-', E_1.nptr, T.nptr) \} \\
 E &\rightarrow T \{ E.nptr := T.nptr \} \\
 T &\rightarrow \mathbf{id} \{ T.nptr := \text{mkleaf}(\mathbf{id}, \mathbf{id}.entry) \}
 \end{aligned}$$


$$\begin{aligned}
 E &\rightarrow T \{ R.i := T.nptr \} R \{ E.nptr := R.s \} \\
 R &\rightarrow + T \{ R_1.i := \text{mknode}('+', R.i, T.nptr) \} R_1 \{ R.s := R_1.s \} \\
 R &\rightarrow - T \{ R_1.i := \text{mknode}('-', R.i, T.nptr) \} R_1 \{ R.s := R_1.s \} \\
 R &\rightarrow \epsilon \{ R.s := R.i \} \\
 T &\rightarrow \mathbf{id} \{ T.nptr := \text{mkleaf}(\mathbf{id}, \mathbf{id}.entry) \}
 \end{aligned}$$

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

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