Syntax-Directed Translation

Part V

Eliminating Left Recursion from Translation Scheme

- A translation scheme with a left recursive grammar.

$E \rightarrow E_1 + T$ { $E.val = E_1.val + T.val$ }
$E \rightarrow E_1 - T$ { $E.val = E_1.val - T.val$ }
$E \rightarrow T$ { $E.val = T.val$ }
$T \rightarrow T_1 * F$ { $T.val = T_1.val * F.val$ }
$T \rightarrow F$ { $T.val = F.val$ }
$F \rightarrow (E)$ { $F.val = E.val$ }
$F \rightarrow$ **digit** { $F.val =$ **digit**.lexval }

- When we eliminate the left recursion from the grammar (to get a suitable grammar for the top-down parsing) we also have to change semantic actions

Eliminating Left Recursion (cont.)

inherited attribute   synthesized attribute

$E \rightarrow T$ { A.in=T.val } A { E.val=A.syn }

$A \rightarrow + T$ { A_1.in=A.in+T.val } A_1 { A.syn = A_1.syn}

$A \rightarrow - T$ { A_1.in=A.in-T.val } A_1 { A.syn = A_1.syn}

$A \rightarrow \varepsilon$ { A.syn = A.in }

$T \rightarrow F$ { B.in=F.val } B { T.val=B.syn }

$B \rightarrow * F$ { B_1.in=B.in*F.val } B_1 { B.syn = B_1.syn}

$B \rightarrow \varepsilon$ { B.syn = B.in }

$F \rightarrow (E)$ { F.val = E.val }

$F \rightarrow$ **digit** { F.val = **digit**.lexval }

Eliminating Left Recursion (in general)

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

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

a left recursive grammar with synthesized attributes (a,y,x).

$\Downarrow$ eliminate left recursion

inherited attribute of the new non-terminal

synthesized attribute of the new non-terminal

$A \rightarrow X\ \{\ R.in = f(X.x)\ \}\ R\ \{\ A.a = R.syn\ \}$

$R \rightarrow Y\ \{\ R_1.in = g(R.in, Y.y)\ \}\ R_1\ \{\ R.syn = R_1.syn \}$

$R \rightarrow \varepsilon\ \ \{\ R.syn = R.in\ \}$

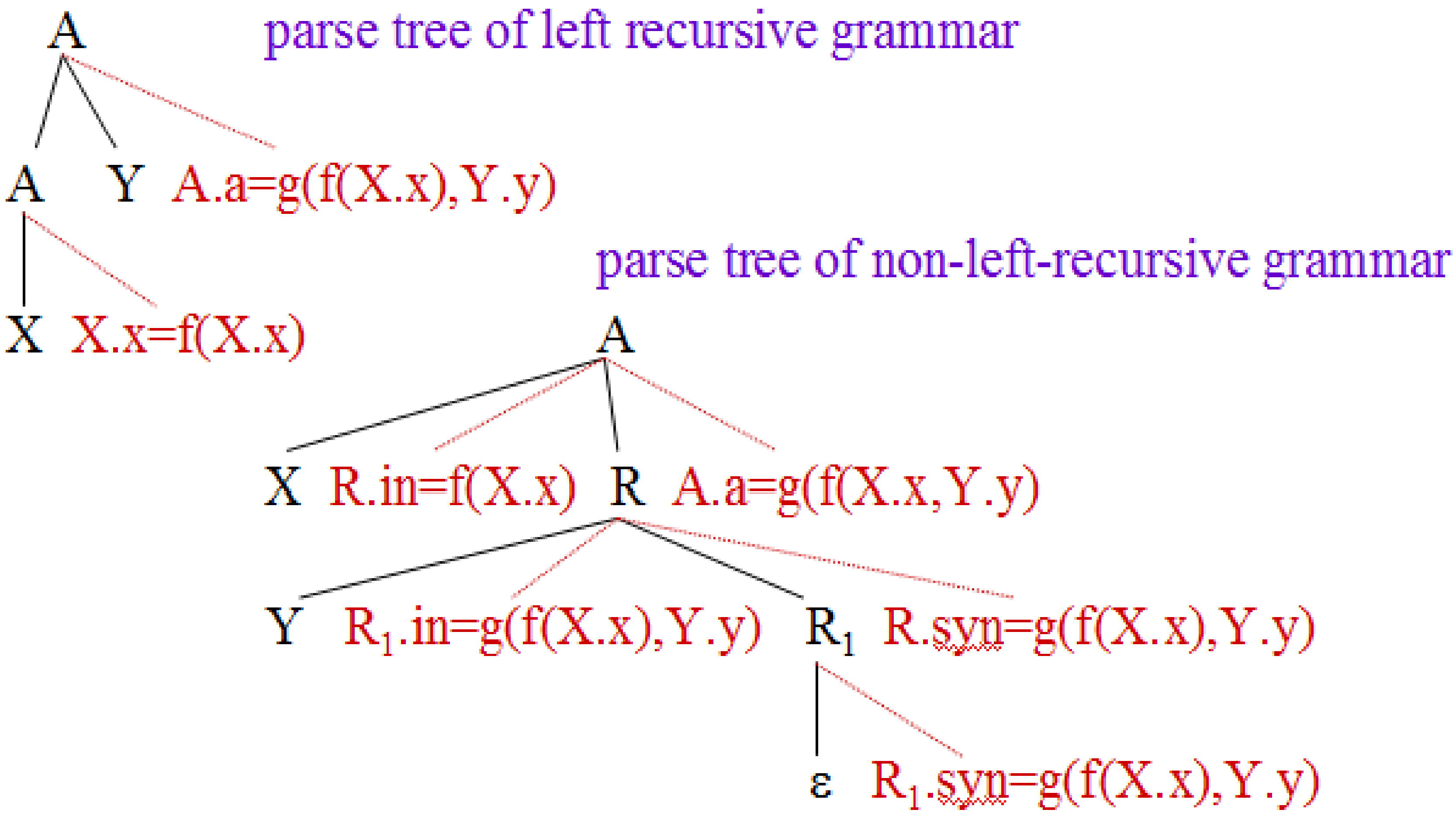
Eliminating Left Recursion (Example)

1) $T \rightarrow T * F$ $\{ T.val = T_1.val * F.val\}$
2) $T \rightarrow F$ $\{T.val = F.val\}$
3) $F \rightarrow$ **digit** $\{F.val=digit.lexval\}$

$T \rightarrow FT'$
$T' \rightarrow *F\ T'$
$T' \rightarrow \varepsilon$
$F \rightarrow$ **digit**

Translation Scheme
1) $T \rightarrow F\ \{T'.inh = F.val\}\ T'\ \{T.Val = T'.syn\}$
2) $T' \rightarrow *\ F\ \{T'_1.inh = T'.inh * F.val\}\ T'_1\ \{T'.syn=T'_1.syn\}$
3) $T' \rightarrow \varepsilon\ \{T'.syn=T'.inh\}$
4) $F \rightarrow$ **digit** {F.val=**digit**.*lexval*}

Evaluating attributes



Eliminating Left Recursion (Example)

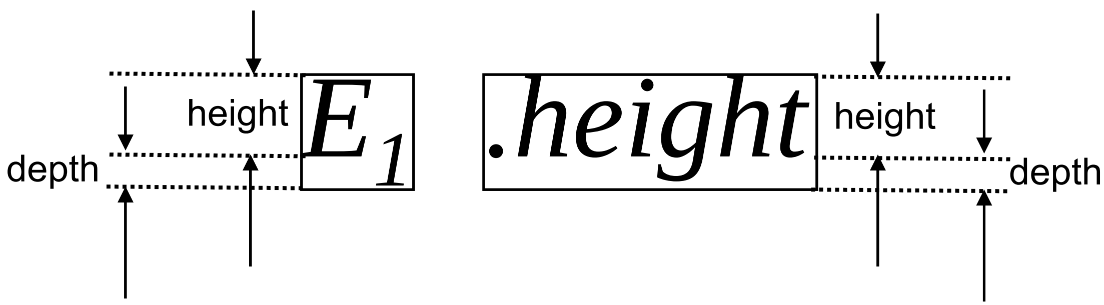
- Other examples from book
  - Figure 5.8: Page 315
  - Figure 5.13: Page 321
  - Figure 5.16: Page 322

SDTs for L-Attributed Definitions

- Rules for turning L-attributed SDD into an SDT

- Embed action that computes the inherited attribtes for a non-terminal A immediately before that occurences of A in the body of the production.
  - If several inherited attributes for A depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.
- Place the actions that computed a synthesized attribute for the head of a production at the end of the body of that production.

Example: type setting boxes



- ps: point size. Non subscript character is 10. Subscript point size = 0.7p, p is the box point size
- baseline: vertical position that corresponds to the bottom of the lines of text.
- ht: height. Distance from the top of the box to baseline.
- dp: depth. Distance from the baseline to the bottom of the box.

Example: type setting boxes

Production	Semantic Rules
$S \rightarrow B$	B.ps=10
$B \rightarrow B_1\ B_2$	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)
$B \rightarrow B_1$ sub B_2	B_1.ps=B.ps B_2.ps=0.7 * B.ps B.ht=max(B_1.ht,B_2.ht -0.25XB.ps) B.dp=max(B_1.dp,B_2.dp+0.25*B.ps)
$B \rightarrow (B_1)$	B_1.ps = B.ps B.ht = B_1.ht B.dp = B_1.dp
$B \rightarrow$ text	B.ht = getHt(B.ps,text.lexval) B.dp=getDp(B.ps,text.lextval)

SDD for type setting box

Production	Semantic Actions
$S \rightarrow$ **B**	{B.ps=10}
$B \rightarrow$ **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)}
$B \rightarrow$ **B_1 sub** **B_2**	{B_1.ps=B.ps} {B_2.ps=0.7 * B.ps} {B.ht=max(B_1.ht,B_2.ht -0.25XB.ps) B.dp=max(B_1.dp,B_2.dp+0.25*B.ps)}
$B \rightarrow$ (**B_1)**	{B_1.ps = B.ps} {B.ht = B_1.ht B.dp = B_1.dp}
$B \rightarrow$ text	{B.ht = getHt(B.ps,text.lexval) B.dp=getDp(B.ps,text.lextval)}

SDT for type setting box

Example: intermediate code for while-statement

- S → **while** (C) S_1

Production	Semantic Rules
S → while (C) S_1	L1=new(); L2=new(); S_1.next = L1; C.false=S.next; C.true=L2; S.code=label || L1 || C.code || label || L2 || S_1.code

Production	Semantic Actions
S → while (	{L1=new(); L2=new(); C.false=S.next; C.true=L2;}
C)	{S1.next = L1;}
S_1	S.code=label || L1 || C.code || label || L2 || S1.code

Translation during Recursive-Descent Parsing

- In recursive-descent parser a function A is used for each non-terminal A
  - The arguments of function A are the inherited attributes of non-terminal A.
  - The return-value of function A are the collection of synthesized attributes of non-terminal A.

Translation during Recursive-Descent Parsing

- In the body of function A
  - Decide upon the production used to expand A
  - Check that each terminal appears on the input when it is required
  - Preserve in local variables the values of all attributes needed to compute inherited attributes for nonterminals in the body or synthesized attributes for the head nonterminal.
  - Call functions corresponding to nonterminals in the body of the selected production with proper arguments.

Example

Production	Semantic Actions
$S \rightarrow$ while (C) S_1	L1=new(); L2=new(); S1.next = L1; C.false=S.next; C.true=L2; S.code=label || L1 || C.code || label || L2 || S1.code

```
String S(label next){
        string Scode, Ccode; /* local variables holding code fragments */
        label L1, L2; /* the local labels */
        if (current input == token while){
                advance input;
                check '(' is next on the input, and advance;
                L1=new();
                L2=new();
                Ccode=C(next,L2);
                check ')' is next on the input, and advance;
                Scode= S(L1);
                return("label" || L1 || Ccode || "label" || L2 || Scode);
        }
        else / * other statement types */
}
```

Translation Scheme - Intermediate Code Generation

$E \rightarrow T$ { A.in=T.loc } A { E.loc=A.loc }

$A \rightarrow + T$ { A_1.in=newtemp(); emit(add,A.in,T.loc,A_1.in) }

A_1 { A.loc = A_1.loc}

$A \rightarrow \varepsilon$ { A.loc = A.in }

$T \rightarrow F$ { B.in=F.loc } B { T.loc=B.loc }

$B \rightarrow * F$ { B_1.in=newtemp(); emit(mult,B.in,F.loc,B_1.in) }

B_1 { B.loc = B_1.loc}

$B \rightarrow \varepsilon$ { B.loc = B.in }

$F \rightarrow (E)$ { F.loc = E.loc }

$F \rightarrow$ **id** { F.loc = **id**.name }

Predictive Parsing – Intermediate Code Generation

```
procedure E(char **Eloc) {
    char *Ain, *Tloc, *Aloc;
    call T(&Tloc);  Ain=Tloc;
    call A(Ain,&Aloc); *Eloc=Aloc;
}
procedure A(char *Ain, char **Aloc) {
    if (currtok is +) {
        char *A1in, *Tloc, *A1loc;
       consume(+);  call T(&Tloc);  A1in=newtemp(); emit("add",Ain,Tloc,A1in);
       call A(A1in,&A1loc);  *Aloc=A1loc;
    }
    else { *Aloc = Ain }
}
```

Predictive Parsing (cont.)

```
procedure T(char **Tloc) {
    char *Bin, *Floc, *Bloc;
    call F(&Floc);  Bin=Floc;
    call B(Bin,&Bloc); *Tloc=Bloc;
}
procedure B(char *Bin, char **Bloc) {
    if (currtok is *) {
       char *B1in, *Floc, *B1loc;
      consume(+);  call F(&Floc);  B1in=newtemp(); emit("mult",Bin,Floc,B1in);
      call B(B1in,&B1loc);  Bloc=B1loc;
    }
    else { *Bloc = Bin }
}
procedure F(char **Floc) {
    if (currtok is "(") { char *Eloc; consume("("); call E(&Eloc); consume(")");  *Floc=Eloc
    }
    else { char *idname; consume(id,&idname); *Floc=idname }
}
```

Removing Embedding Semantic Actions

- In bottom-up evaluation scheme, the semantic actions are evaluated during the reductions.
- During the bottom-up evaluation of S-attributed definitions, we have a parallel stack to hold synthesized attributes.
- *Problem*: where are we going to hold inherited attributes?
- *A Solution*:
  - We will convert our grammar to an equivalent grammar to guarantee to the followings.
  - All embedding semantic actions in our translation scheme will be moved into the end of the production rules.
  - All inherited attributes will be copied into the synthesized attributes (most of the time synthesized attributes of new non-terminals).
  - Thus we will be evaluate all semantic actions during reductions, and we find a place to store an inherited attribute.

Removing Embedding Semantic Actions

- To transform our translation scheme into an equivalent translation scheme:

1. Remove an embedding semantic action S_i, put new a non-terminal M_i instead of that semantic action.
2. Put that semantic action S_i into the end of a new production rule $M_i \rightarrow \varepsilon$ for that non-terminal M_i.
3. That semantic action S_i will be evaluated when this new production rule is reduced.
4. The evaluation order of the semantic rules are not changed by this transformation.

Removing Embedding Semantic Actions

$A \rightarrow \{S_1\}\ X_1\ \{S_2\}\ X_2\ \ldots\ \{S_n\}\ X_n$

$\Downarrow$ remove embedding semantic actions

$A \rightarrow M_1\ X_1\ M_2\ X_2\ \ldots\ M_n\ X_n$

$M_1 \rightarrow \varepsilon\ \{S_1\}$

$M_2 \rightarrow \varepsilon\ \{S_2\}$

.

.

$M_n \rightarrow \varepsilon\ \{S_n\}$

Removing Embedding Semantic Actions

$E \rightarrow T\ R$

$R \rightarrow +\ T$ { print("+") } R_1

$R \rightarrow \varepsilon$

$T \rightarrow$ **id** { print(**id**.name) }

$\Downarrow$ remove embedding semantic actions

$E \rightarrow T\ R$

$R \rightarrow +\ T\ M\ R_1$

$R \rightarrow \varepsilon$

$T \rightarrow$ **id** { print(**id**.name) }

$M \rightarrow \varepsilon$ { print("+") }