Attribute Grammars
Chapter 4

Programming Languages

Objective

Today's primary objective:

- Know that action routines are similar to attribute grammars, but are more flexible
- Know that action routines are often used in production compilers
- Understand an action routine example

Action Routines

Action routine – a semantic function that a grammar writer can associate with a grammar for the compiler to execute at particular points during parsing

Action routines are more flexible than attribute grammars.

Action Routines

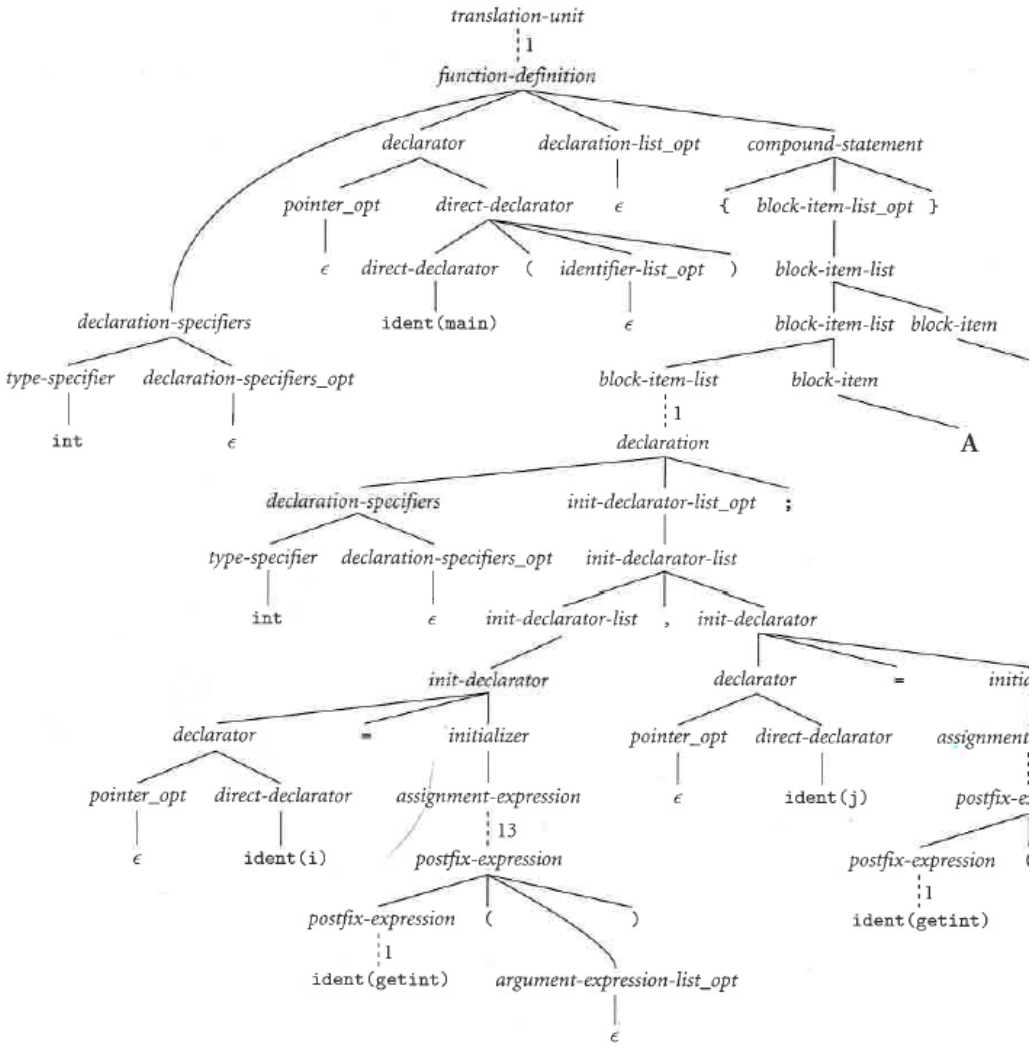
Action routines – routines attached to a grammar which is executed by the compiler as it parses code

- Often used in production compilers
- Can be used to build parse trees

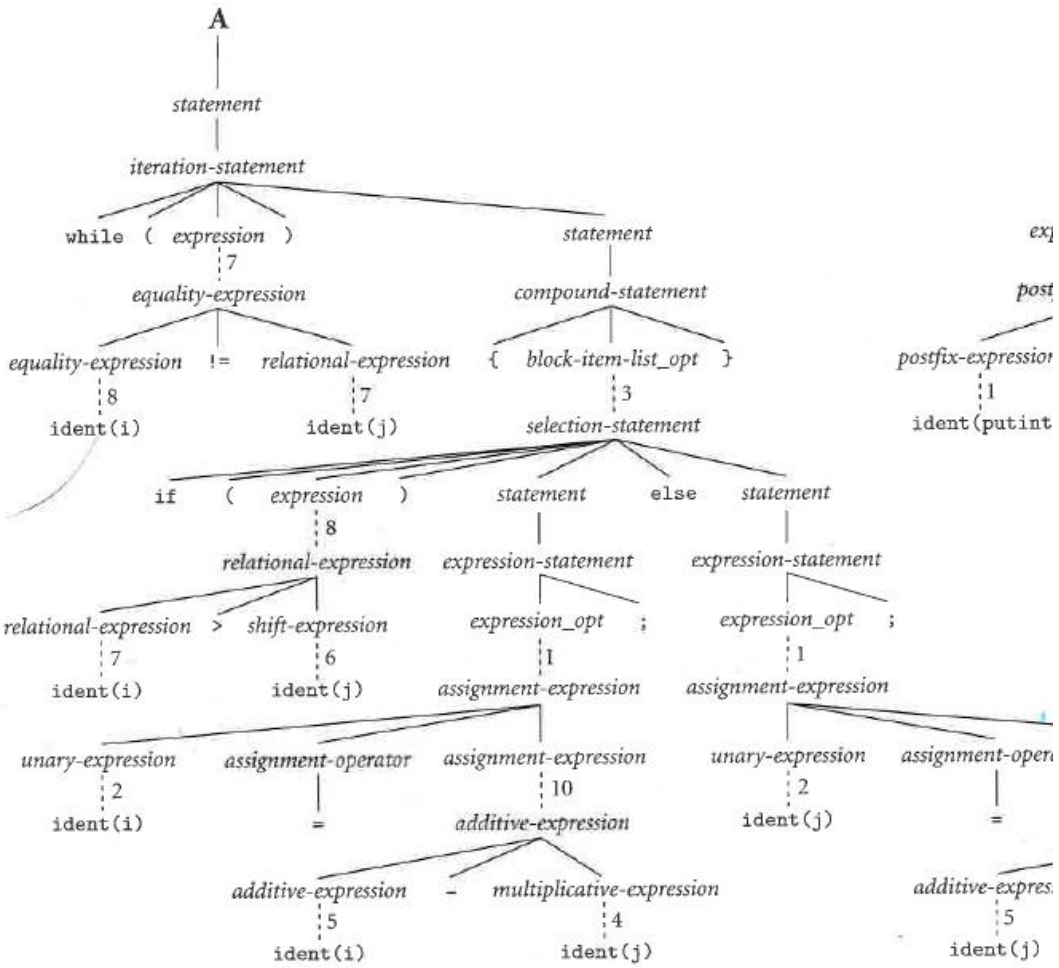
Greatest Common Divisor Program

```
int main() {
    int i = getint(), j = getint()
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

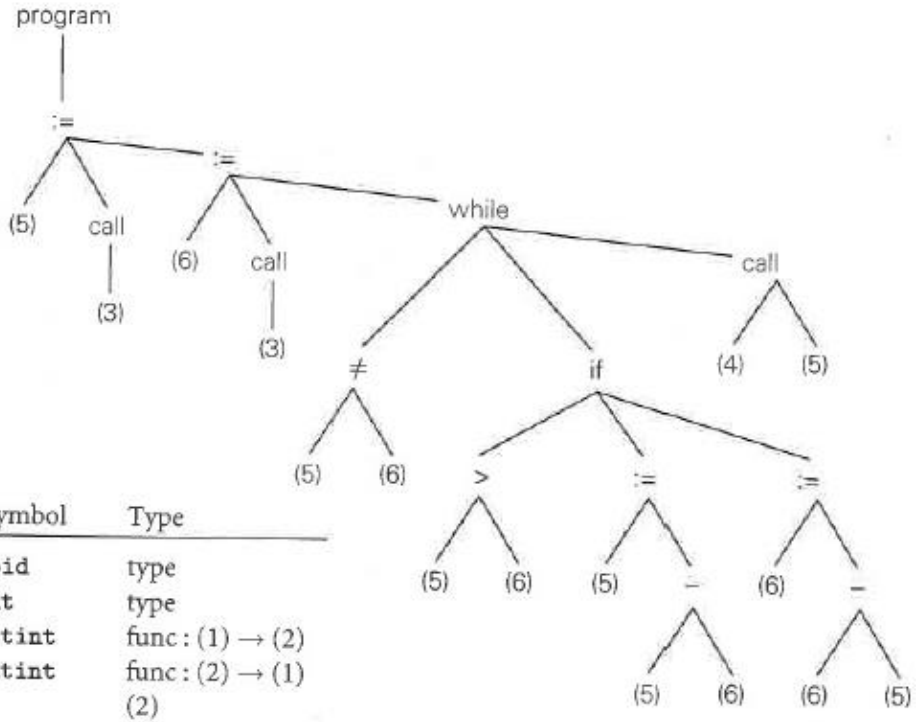
Greatest Common Divisor Syntax Tree



Greatest Common Divisor Syntax Tree - Continued



Greatest Common Divisor Abstract Syntax Tree



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Example for LL Parsing

LL parsing – action routines can appear anywhere within a right-had side. Pointer to the action routine is pushed onto the stack and is executed when it is at the top of the stack.

Thus:

- Action routine at the beginning is executed when the parser predicts the production
- Routine embedded in the middle of the right-hand side is executed when the parser "matches" the symbol to its left

Action Routine Example

Action routines can be used to build a tree.

$E_1 \longrightarrow E_2 + T$
▷ E_1.ptr := make_bin_op("+", E_2.ptr, T.ptr)

$E_1 \longrightarrow E_2 - T$
▷ E_1.ptr := make_bin_op("−", E_2.ptr, T.ptr)

$E \longrightarrow T$
▷ E.ptr := T.ptr

$T_1 \longrightarrow T_2 * F$
▷ T_1.ptr := make_bin_op("×", T_2.ptr, F.ptr)

$T_1 \longrightarrow T_2 / F$
▷ T_1.ptr := make_bin_op("÷", T_2.ptr, F.ptr)

$T \longrightarrow F$
▷ T.ptr := F.ptr

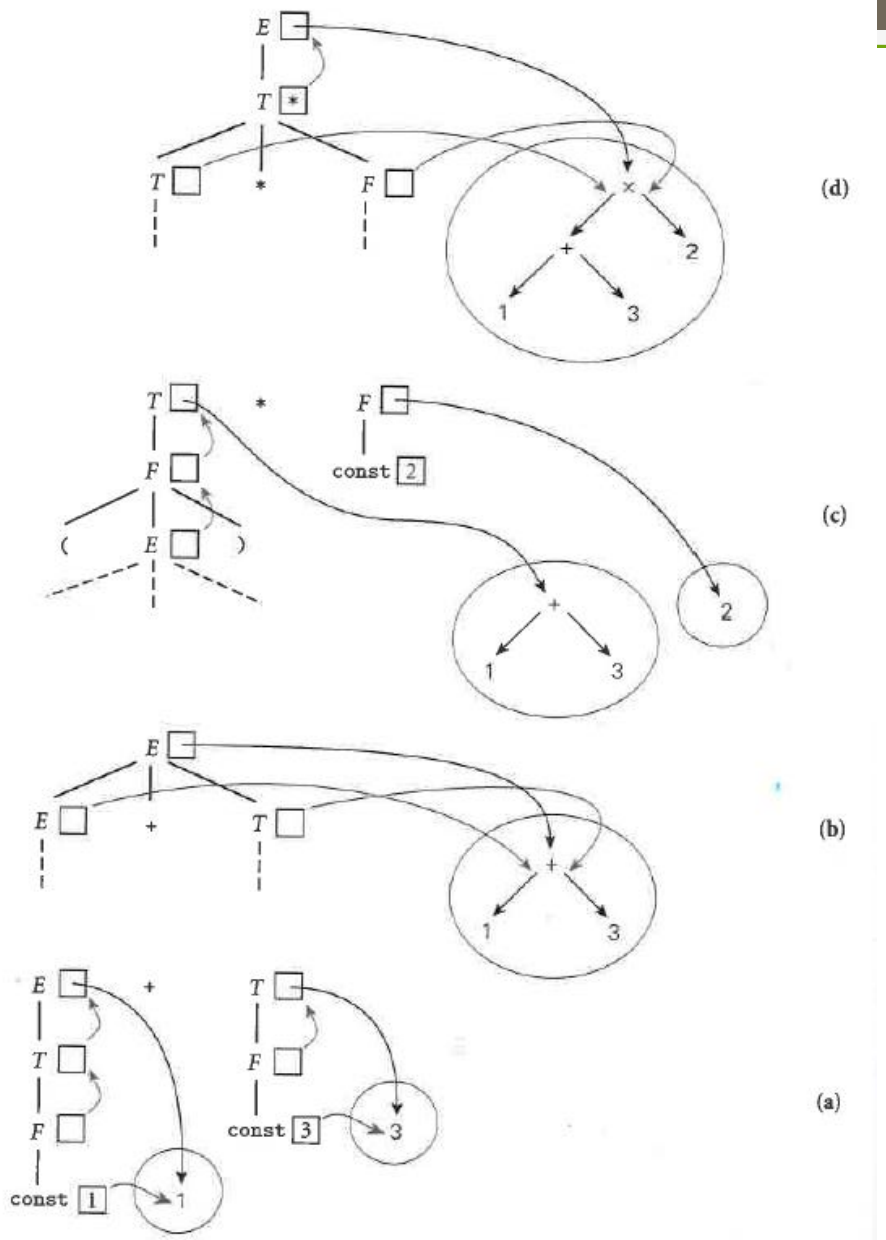
$F_1 \longrightarrow - F_2$
▷ F_1.ptr := make_un_op("+/−", F_2.ptr)

$F \longrightarrow (E)$
▷ F.ptr := E.ptr

$F \longrightarrow$ const
▷ F.ptr := make_leaf(const.val)

Figure 4.5 **Bottom-up (S-attributed) attribute grammar to construct a syntax tree.** The symbol +/− is used (as it is on calculators) to indicate change of sign.

Action Routine Example



Action Routine Example

$E \longrightarrow T\ TT$
- ▷ TT.st := T.ptr
- ▷ E.ptr := TT.ptr

$TT_1 \longrightarrow +\ T\ TT_2$
- ▷ TT_2.st := make_bin_op("+", TT_1.st, T.ptr)
- ▷ TT_1.ptr := TT_2.ptr

$TT_1 \longrightarrow -\ T\ TT_2$
- ▷ TT_2.st := make_bin_op("−", TT_1.st, T.ptr)
- ▷ TT_1.ptr := TT_2.ptr

$TT \longrightarrow \epsilon$
- ▷ TT.ptr := TT.st

$T \longrightarrow F\ FT$
- ▷ FT.st := F.ptr
- ▷ T.ptr := FT.ptr

$FT_1 \longrightarrow *\ F\ FT_2$
- ▷ FT_2.st := make_bin_op("×", FT_1.st, F.ptr)
- ▷ FT_1.ptr := FT_2.ptr

$FT_1 \longrightarrow /\ F\ FT_2$
- ▷ FT_2.st := make_bin_op("÷", FT_1.st, F.ptr)
- ▷ FT_1.ptr := FT_2.ptr

$FT \longrightarrow \epsilon$
- ▷ FT.ptr := FT.st

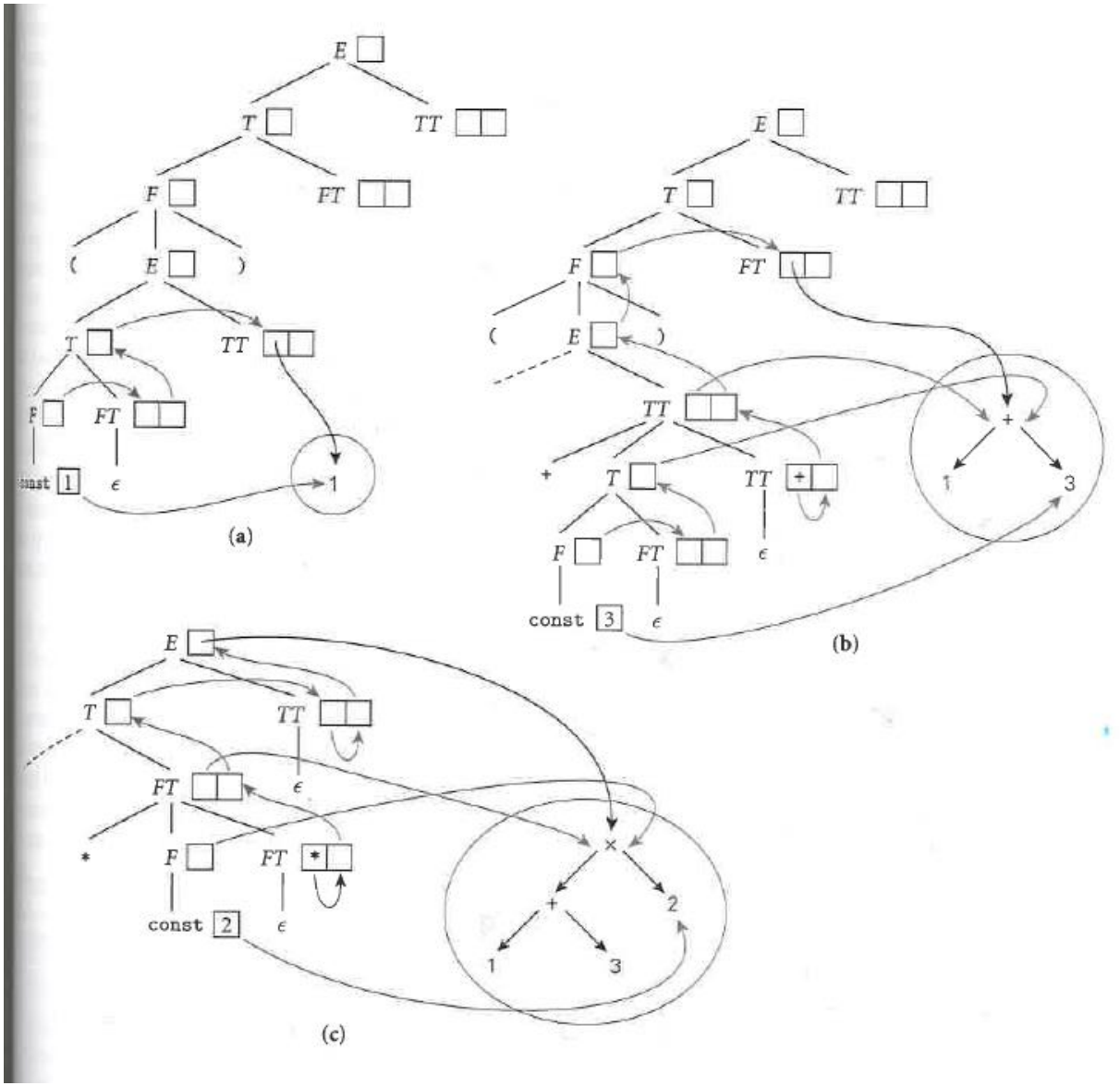
$F_1 \longrightarrow -\ F_2$
- ▷ F_1.ptr := make_un_op("+/−", F_2.ptr)

$F \longrightarrow (\ E\ )$
- ▷ F.ptr := E.ptr

$F \longrightarrow$ const
- ▷ F.ptr := make_leaf(const.val)

Figure 4.6 Top-down (L-attributed) attribute grammar to construct a syntax tree. Here the st attribute, like the ptr attribute (and unlike the st attribute of Figure 4.3), is a pointer to a syntax tree node.

Action Routine Example



Action Routine Example

Consider the grammar:

E → T TT

TT → + T TT | - T TT | ε

T → F FT

FT → * F FT | / F FT | ε

F → - F

F → (E)

F → const

Figure 4.9 in text, page 195

Action Routine Example

Action routines added:

$E \rightarrow T \ \{TT.st:=T.ptr\} \ TT \ \{E.ptr:=TT.ptr\}$

$TT_1 \rightarrow T \ \{TT_2.st:=make\_bin\_op(\text{“+”},TT_1.st,T.ptr)\}$
$TT_2 \ \{TT_1.ptr:=TT_2.ptr\}$

$TT_1 \rightarrow T \ \{TT_2.st:=make\_bin\_op(\text{“-”},TT_1.st,T.ptr)\}$
$TT_2 \ \{TT_1.ptr:=TT_2.ptr\}$

$TT \rightarrow \varepsilon \ \{TT.ptr:=TT.st\}$

$T \rightarrow F \ \{FT.st:=F.ptr\} \ FT \ \{T.ptr:=FT.ptr\}$

Action Routine Example - continued

$FT_1 \rightarrow F$ {FT_2.st:=make_bin_op("x",FT_1.st,F.ptr)}
FT_2 {FT_1.ptr:=FT_2.ptr}

$FT_1 \rightarrow F$ {FT_2.st:=make_bin_op("/",FT_1.st,F.ptr)}
FT_2 {FT_1.ptr:=FT_2.ptr}

$FT \rightarrow \varepsilon$ {FT.ptr:=FT.st}

$F_1 \rightarrow F_2$ {F_1.ptr:=make_un_op("+/-",FT_2.ptr)}

$F \rightarrow (E)$ {F.ptr:=E.ptr}

$F \rightarrow$ const {F.ptr:=make_leaf(const.ptr)}

Build Tree

Use the previous grammar with action routines to build a tree for the expression:

2 * 3 + 5

Build Tree

Stack:

F → const {F.ptr:=make_leaf(const.ptr)}

Build Tree

Stack:

$FT_1 \rightarrow$ * F {FT_2.st:= make_bin_op("x",FT_1.st, F.ptr)}

FT_2 {FT_1.ptr:= FT_2.ptr}

*

F

FT

TT

Simplified Action Routine Example

Demonstrating action routines using a more simple (LR) grammar.

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow - F$

$F \rightarrow (E)$

$F \rightarrow \text{const}$

Simplified Action Routine Example

More simplified example:

E → E + T

E_1.ptr:= make_bin_op("+", E_2.ptr, T.ptr)

E → E - T

E_1.ptr:= make_bin_op("-", E_2.ptr, T.ptr)

E → T

E.ptr:= T.ptr

T → T * F

T_1.ptr:= make_bin_op("x", T_2.ptr, F.ptr)

T → T / F

T_1.ptr:= make_bin_op("/", T_2.ptr, F.ptr)

T → F

T.ptr:= F.ptr

F → - F

F_1.ptr:= make_un_op("+/-", F_2.ptr) (+/- - indicates a change of sign, similar to on calculators)

F → (E)

F.ptr:=E.ptr

F → const

F.ptr:=make_leaf(const.val)