IUBAT- International University of Business Agriculture and Technology

Final Assignment

Course Name: Compiler Design

Course Number: CSC 437

Submitted To

Md. Alomgir Hossain

Assistant Professor

Department Of Computer Science and Engineering

Submitted By

Ahana Nandi Tultul

ID: 18203032

Program: BCSE

Section: D

Date of Submission: 05 – 01 – 2020

Syntax Analysis

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i-1$) {
4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \delta_k$ are all the current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions;
7) }

Conceptual Technique Summary (AGAIN)

- Put some order in the nonterminals.
- Start by making first nonterminal productions left-recursion-free.
- Put the first nonterminal left-recursion-free productions into those of the second one.
- Now make the productions of second nonterminal left-recursion-free.
- Thus keep on growing the set of left-recursion-free productions.

Left-Recursive Grammar

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- All $A_i = A_2 = A$-productions together,

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- Eliminating the immediate left recursion among the A-productions yields the following,

$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

- Put together we get the following non-left-recursive grammar,

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - needs a special form of grammars (LL(1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
    - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

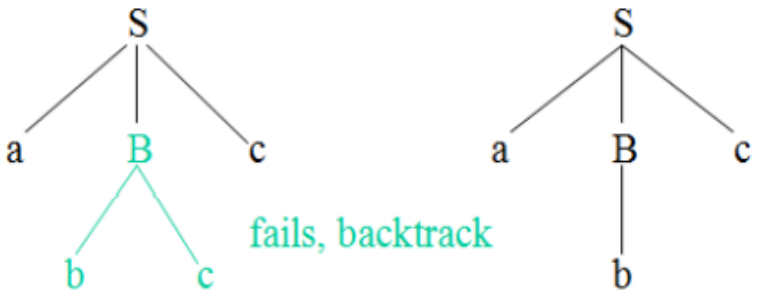
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

input: abc



Predictive Parsing

- The goal of predictive parsing is to construct a top-down parser that never backtracks. To do so, we must transform a grammar in two ways:
- eliminate left recursion, and
- perform left factoring.

left factoring helps predict which rule to use without backtracking.

Predictive Parser

a grammar	→	→	a grammar suitable for predictive
	eliminate left recursion	left factor	parsing (a LL(1) grammar) no %100 guarantee.

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \rightarrow \alpha_1 \mid ... \mid \alpha_n$ input: ... a .....

current token

Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
- The basic idea is that sometimes it is not clear which of two alternative productions to use to expand a nonterminal *A*.
- We may be able to rewrite the *A*-productions to defer the decision until we have seen enough of the input to make the right choice.

- We have the two productions,

$$\begin{aligned} \textit{stmt} \rightarrow\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\ \mid\ & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \end{aligned}$$

- On seeing the input token **if**, we cannot immediately tell which production to choose to expand *stmt*.

Left Factoring — *continued*

- $A \to \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions.
- The input begins with a nonempty string derived from α.
- We do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$.
- However, we may defer the decision by expanding A to $\alpha A'$.
- Then, after seeing the input derived from α we expand A' to β_1 or β_2.
- Left-factored, the original productions become,

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

Left Factoring Algorithm — *continued*

Method.

- For each nonterminal A find the longest prefix α common to two or more of its alternatives.
- If $\alpha \neq \epsilon$ (there is a nontrivial common prefix), replace all the A productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ where γ represents all alternatives that do not begin with α by

  $$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{aligned}$$

  where A' is a new nonterminal.
- Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example

- The following grammar abstracts the dangling-else problem:

  $S \rightarrow iEtS \mid iEtSeS \mid a$

  $E \rightarrow b$

- Here i, t, and e stand for **if**, **then** and **else**, E and S for "expression" and "statement."
- Left-factored, this grammar becomes:

  $S \rightarrow iEtSS' \mid a$

  $S' \rightarrow eS \mid \epsilon$

  $E \rightarrow b$

- Thus, we may expand S to $iEtSS'$ on input i, and wait until $iEtS$ has been seen to decide whether to expand S' to eS or to ϵ.

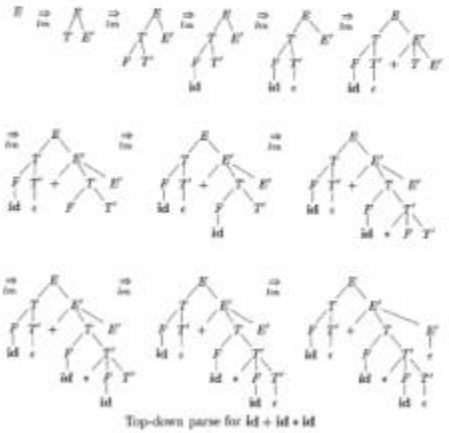
Top-Down Parsing

- Top-down parsing can be viewed as the problem of
  - constructing a parse tree for the input string,
  - starting from the root and
  - creating the nodes of the parse tree in preorder (depth-first).
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Example

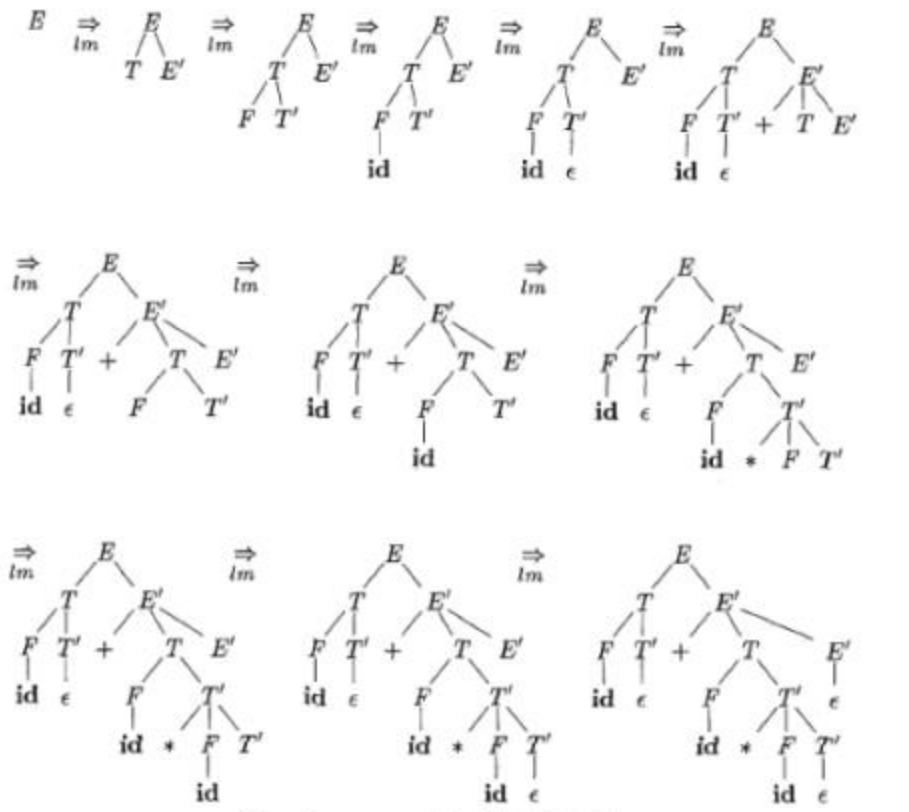
- The sequence of parse trees for the input **id** + **id** ∗ **id** is a top-down parse according to grammar.

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$



Top-down parse for id + id * id

- This sequence of trees corresponds to a leftmost derivation of the input.



Top-down parse for id + id * id

Top-Down Parsing — *continued*

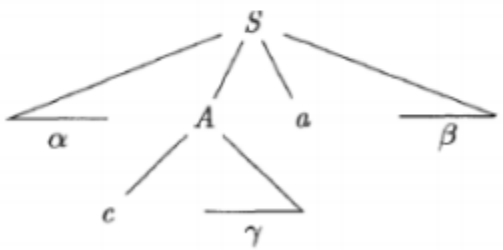
- The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input is sometimes called the LL(k) class.

FIRST and FOLLOW

- The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar *G*.
- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.

FIRST and FOLLOW — *continued*

- Define FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.
- If $\alpha \Rightarrow \epsilon$, then ϵ is also in FIRST(α).
- For example, in figure $A \Rightarrow c\gamma$, so c is in FIRST(A).



Terminal c is in FIRST(A) and a is in FOLLOW(A)

FIRST and FOLLOW — *continued*

- Let us see how FIRST can be used during predictive parsing.
- Consider two A-productions $A \to \alpha \mid \beta$, where FIRST(α) and FIRST(β) are disjoint sets.
- We can then choose between these A-productions by looking at the next input symbol a, since a can be in at most one of FIRST(α) and FIRST(β), not both.
- For instance, if a is in FIRST(β) choose the production $A \to \beta$.

FIRST and FOLLOW — *continued*

- Define FOLLOW(A), nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form.

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Example

1. If X is terminal, then FIRST(X) is $\{X\}$.
2. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).
3. If X is nonterminal and $X \to Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is, $Y_1, \ldots, Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$ then add ϵ to FIRST(X).

Grammar,

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \mathbf{id}
\end{aligned}
$$

Then,

$$
\begin{aligned}
&\text{FIRST}(E) = \text{FIRST}(T) = \\
&\quad \text{FIRST}(F) = \{(, \mathbf{id}\} \\
&\text{FIRST}(E') = \{+, \epsilon\} \\
&\text{FIRST}(T') = \{*, \epsilon\}
\end{aligned}
$$

Example

1. FIRST(F) = FIRST(T) = FIRST(E) = $\{(, \mathbf{id}\}$.

- To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis.
- T has only one production, and its body starts with F.
- Since F does not derive ϵ, FIRST(T) must be the same as FIRST(F).
- The same argument covers FIRST(E).

Grammar,

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid \mathbf{id}
\end{aligned}
$$

Then,

$$
\begin{aligned}
&\text{FIRST}(E) = \text{FIRST}(T) = \\
&\quad \text{FIRST}(F) = \{(, \mathbf{id}\} \\
&\text{FIRST}(E') = \{+, \epsilon\} \\
&\text{FIRST}(T') = \{*, \epsilon\}
\end{aligned}
$$

Example

3. $\text{FIRST}(T') = \{*, \epsilon\}$.
   - The reasoning is analogous to that for $\text{FIRST}(E')$.

Grammar,

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \mid \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \mid \epsilon \\
F & \rightarrow & (E) \mid \textbf{id}
\end{array}
$$

Then,

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\}$$
$$\text{FIRST}(E') = \{+, \epsilon\}$$
$$\text{FIRST}(T') = \{*, \epsilon\}$$

Example — *continued*

- Grammar:

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \mid \epsilon
\end{array}
\qquad
\begin{array}{lcl}
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \mid \epsilon
\end{array}
\qquad
\begin{array}{lcl}
F & \rightarrow & (E) \mid \textbf{id}
\end{array}
$$

- Computation of FOLLOW:

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)

Initially all sets are empty

Put \$ in FOLLOW(E) by rule (1) *(Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker)*

\$				

Example — *continued*

$$
\begin{array}{lcllcllcl}
E & \to & TE' & & T & \to & FT' & & F & \to & (E) \mid \mathbf{id} \\
E' & \to & +TE' \mid \epsilon & & T' & \to & *FT' \mid \epsilon & & & &
\end{array}
$$

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \mathbf{id}\}$, $\text{FIRST}(E') = \{+, \epsilon\}$, $\text{FIRST}(T') = \{*, \epsilon\}$

By rule (2) *(If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B))* *applied to,*

- $E \to TE'$: *FIRST(E') except ϵ i.e. $\{+\}$ are in FOLLOW(T)*
- $E' \to +TE'$: *FIRST(E') except ϵ i.e. $\{+\}$ are in FOLLOW(T)*
- $T \to FT'$: *FIRST(T') except ϵ i.e. $\{*\}$ are in FOLLOW(F)*
- $T \to *FT'$: *FIRST(T') except ϵ i.e. $\{*\}$ are in FOLLOW(F)*
- $F \to (E)$: *FIRST()) i.e. $\{)\}$ are in FOLLOW(E)*

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
$,)		+		*

Rule (2) is not applicable any more since it depends only on FIRST, which are now stable sets.

Example — *continued*

Application of rule (3) *If there is a production $A \to \alpha B$, or a production $A \to \alpha B\beta$ where FIRST(β) contains ϵ (i.e., $\beta \Rightarrow \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

$E \to TE'$ $\quad T \to FT'$ $\quad F \to (E) \mid \mathbf{id}$

$E' \to +TE' \mid \epsilon$ $\quad T' \to *FT' \mid \epsilon$

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
$,)		+		*

$E \to TE'$: Everything in FOLLOW(E) are in FOLLOW(E')

$,)	$,)	+		*

$E' \to +TE'$ (also $\epsilon \in$ FIRST(E')): Everything in FOLLOW(E') are in FOLLOW(T)

$,)	$,)	+, $,)		*

$T \to FT'$: Everything in FOLLOW(T) are in FOLLOW(T')

$,)	$,)	+, $,)	+, $,)	*

Example — *continued*

Application of rule (3) — continued *(If there is a production $A \to \alpha B$, or a production $A \to \alpha B\beta$ where FIRST(β) contains ϵ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B))*

$E \to TE'$ $\quad$ $T \to FT'$ $\quad$ $F \to (E) \mid \mathbf{id}$

$E' \to +TE' \mid \epsilon$ $\quad$ $T' \to *FT' \mid \epsilon$

FOLLOW(E)	FOLLOW(E')	FOLLOW(T)	FOLLOW(T')	FOLLOW(F)
$,)	$,)	+, $,)	+, $,)	*

$T' \to *FT'$ (also $\epsilon \in$ FIRST(T')): Everything in FOLLOW(T') are in FOLLOW(F)

$,)	$,)	+, $,)	+, $,)	*, +, $,)

We can try applying Rule (3) again, but will find that the sets have stabilized (nothing can be added to any FOLLOW set).

Context Free Grammar

Definition :

That is used to specify the syntax of a language. A grammar naturally describes the hierarchical structure of most programming language constructs.

For example, an if-else statement in Java can have the form

if (expression) statement else statement

Using the variable expr to denote an expression and

the variable stmt to denote a statement, this structuring rule can be expressed as

Stmt → **if expr stmt else stmt**

- in which the arrow may be read as "\can have the form."
- Such a rule is called a **production In a production**,
- lexical elements like the keyword if and the parentheses are called **terminals**
- Variables like expr and stmt represent sequences of terminals and are called n**onterminals.**

Context Free Grammar

A **context-free grammar** has four components:

1. A set of **termina**l symbols, sometimes referred to as \tokens." The terminals are the elementary symbols of the language defined by the grammar.
2. A set of **nonterminals**, sometimes called \syntactic variables." Each nonterminal represents a set of strings of terminals.
3. A set of **productions** where each production consists of a nonterminal, called the **head or left side** of the production, an arrow, and a sequence of terminals and/or nonterminals, called the **body or right side** of the production.
4. A designation of one of the nonterminals as **the start symbol**.

Context Free Grammar

The grammar here defines simple arithmetic expressions. In this grammar,

- the terminal symbols are id + - * / ( ) .
- The nonterminal symbols are expression, term and factor,
- and expression is the start symbol

expression → expression + term
expression → expression - term
Expression → term
Term → term *factor
term→ term / factor
term →factor
factor →(expression)
factor→ id

Figure : Grammar for simple arithmetic expressions

Context Free Grammar

expression → expression + term
express → expression - term
Expression term
Term → term *factor
term→ term / factor
term →factor
factor →(expression)
factor→ id

Using these conventions, we can write down,

E→ E+ T | E- T | T
T →T* F | T / F | F
F→(E) | id

The notational conventions tell us that E T, and F are nonterminals, with E the start symbol. The remaining symbols are terminals

Derivation

For example, consider the following grammar, with a single nonterminal E which adds a production :

E→ E +E | E *E | -E |(E)| id

2 types derivation can happen : Left Most & Right Most Derivation

Example : (Left Most)

Input: **-(id+id)**

E → -E →-(E) →-(E+ E) →- (id +E) →- (id + id)

TRY RIGHT MOST!!

Each nonterminal is replaced by the same body in the two derivations, but the order of replacements is different.

To understand how parsers work, we shall consider derivations in which the nonterminal to be replaced at each step is chosen as follows:

1. In *leftmost* derivations, the leftmost nonterminal in each sentential is always chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$.

2. In *rightmost* derivations, the rightmost nonterminal is always chosen; we write $\alpha \underset{rm}{\Rightarrow} \beta$ in this case.

Derivation (4.8) is leftmost, so it can be rewritten as

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

Note that (4.9) is a rightmost derivation.

Using our notational conventions, every leftmost step can be written as $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$, where w consists of terminals only, $A \rightarrow \delta$ is the production applied, and γ is a string of grammar symbols. To emphasize that α derives β by a leftmost derivation, we write $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$. If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, then we say that α is a *left-sentential form* of the grammar at hand.

Analogous definitions hold for rightmost derivations. Rightmost derivations are sometimes called *canonical* derivations.

4.2.4 Parse Trees and Derivations

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

For example, the parse tree for $-(\mathbf{id} + \mathbf{id})$ in Fig. 4.3, results from the derivation (4.8) as well as derivation (4.9).

The leaves of a parse tree are labeled by nonterminals or terminals and, read from left to right, constitute a sentential form, called the *yield* or *frontier* of the tree.

To see the relationship between derivations and parse trees, consider any derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$, where α_1 is a single nonterminal A. For each sentential form α_i in the derivation, we can construct a parse tree whose yield is α_i. The process is an induction on i.

BASIS: The tree for $\alpha_1 = A$ is a single node labeled A.

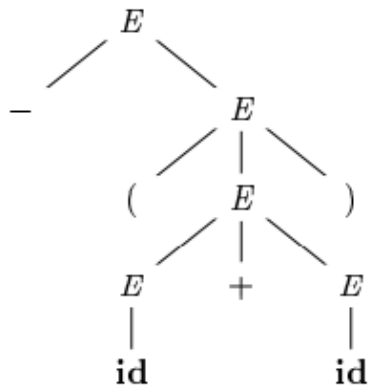


Figure 4.3: Parse tree for $-(\mathbf{id} + \mathbf{id})$

Ambiguity

- a grammar that produces more than one parse tree for some sentence is said to be ambiguous .Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence
- : The arithmetic expression grammar permits two distinct leftmost derivations for the sentence id+id*id
- E→ E *E
  →E+E*E ,
  →Id+E*E
  → id+id*E
  → Id+id*id

  E →E +E
  → id+E
  → id+E*E
  → id+id*E
  → id+id*id

Ambiguity

a

b

4.3.2 Eliminating Ambiguity

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, we shall eliminate the ambiguity from the following "dangling-else" grammar:

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ stmt \\ & | & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ & | & \textbf{other} \end{array} \tag{4.14}$$

Here "**other**" stands for any other statement. According to this grammar, the compound conditional statement

$$\textbf{if}\ E_1\ \textbf{then}\ S_1\ \textbf{else if}\ E_2\ \textbf{then}\ S_2\ \textbf{else}\ S_3$$

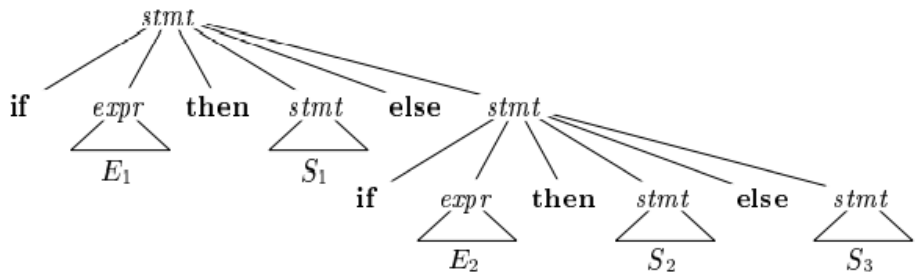


Figure 4.8: Parse tree for a conditional statement

;

has the parse tree shown in Fig. 4.8.[1] Grammar (4.14) is ambiguous since the string

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2 \tag{4.15}$$

has the two parse trees shown in Fig. 4.9.

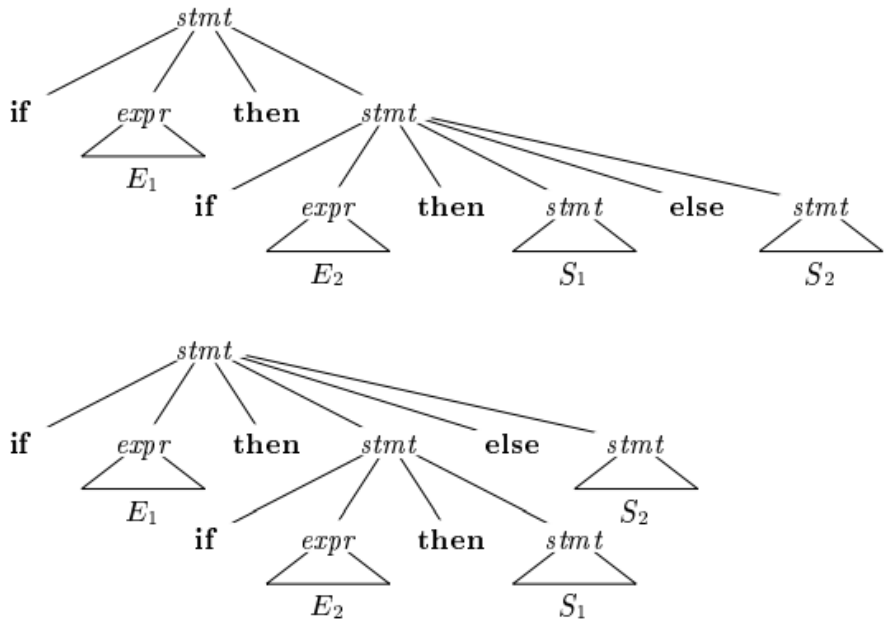


Figure 4.9: Two parse trees for an ambiguous sentence

LL(1) Grammer

- A *predictive parser* is a recursive descent parser that does not require backtracking.
- Predictive parsing is possible only for the class of LL(*k*) grammars, which are the context-free grammars for which there exists some positive integer *k* that allows a recursive descent parser to decide which production to use by examining only the next *k* tokens of input.
- The LL(*k*) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion.

- The first "L" in LL(1) stands for scanning the input from left to right.
- The second "L" for producing a leftmost derivation.
- And the "1" for using one input symbol of lookahead at each step to make parsing action decisions.

LL(1) Grammars — *continued*

A grammar G is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A).
   Likewise, $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

- The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets.
- The third condition is equivalent to stating that if ϵ is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint sets, and likewise if ϵ is in FIRST(α).

LL(1) Grammars — *continued*

- The next algorithm collects the information from FIRST and FOLLOW sets into a predictive parsing table $M[A, a]$, a two dimensional array, where A is a nonterminal, and a is a terminal or the symbol \$, the input endmarker.
- The idea behind the algorithm is the following.
- Suppose $A \to \alpha$ is a production with a in FIRST(α).
- Then, the parser will expand A by α when the current input symbol is a.
- The only complication occurs when $\alpha = \epsilon$ or $\alpha \stackrel{*}{\Rightarrow} \epsilon$.
- In this case, we should again expand A by α if the current input symbol is in FOLLOW(A), or if the \$ on the input has been reached and \$ is in FOLLOW(A).

Algorithm for Construction of a Predictive Parsing Table

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$.
   If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table).

Example

- For the expression grammar below,

$E \rightarrow TE'$ $\quad T \rightarrow FT'$ $\quad F \rightarrow (E) \mid \mathbf{id}$

$E' \rightarrow +TE' \mid \epsilon$ $\quad T' \rightarrow *FT' \mid \epsilon$

the algorithm produces the parsing table in figure.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

- Blanks are error entries.
- Nonblanks indicate a production with which to expand a nonterminal.

For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id			$F \to (E)$		

- Consider production $E \to TE'$.
- Since

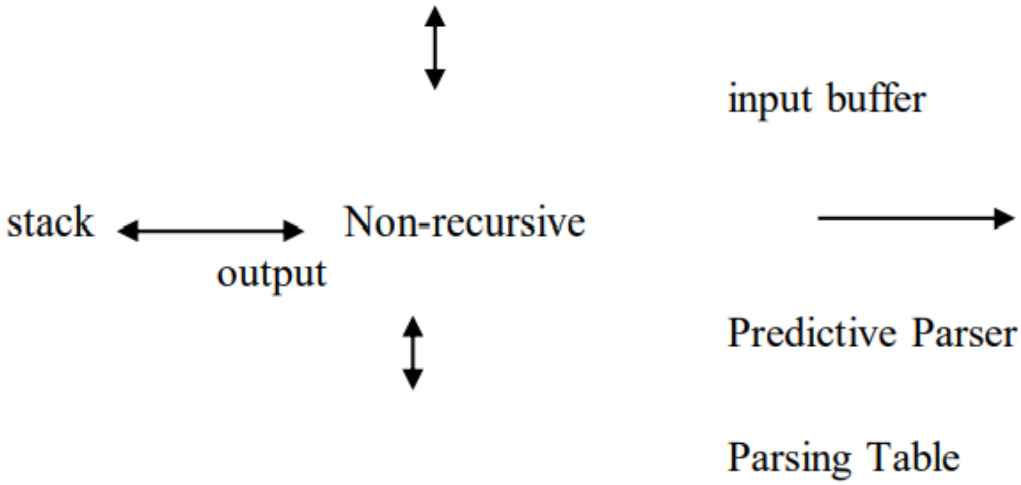
$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \textbf{id}\}$$

this production is added to $M[E, (]$ and $M[E, \textbf{id}]$.

- Production $E' \to +TE'$ is added to $M[E', +]$ since FIRST($+TE'$) $= \{+\}$.
- Since FOLLOW(E') $= \{), \$\}$, production $E' \to \epsilon$ is added to $M[E',)]$ and $M[E', \$]$.

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.



Non-Recursive Predictive Parsing -- LL(1) Parser

Nonrecursive Predictive Parsing

- A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls.
- The parser mimics a leftmost derivation.
- If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols a such that

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol $.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol $.
- initially the stack contains only the symbol $ and the starting symbol S. $S ← initial stack
- when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array M[A,a]
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol $
- each entry holds a production rule.

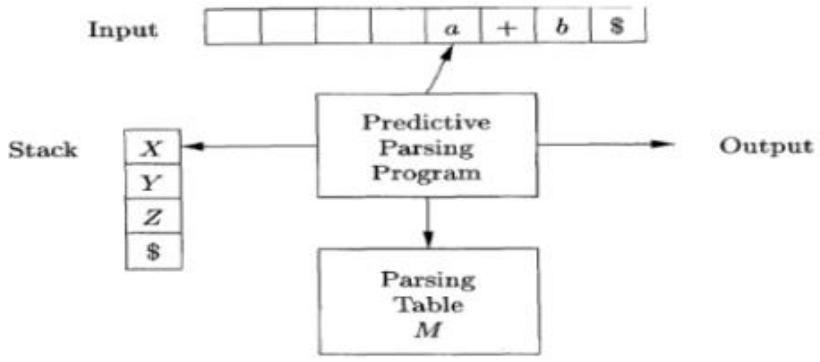
LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

1. If X and a are \$ ➔ parser halts (successful completion)
2. If X and a are the same terminal symbol (different from \$)

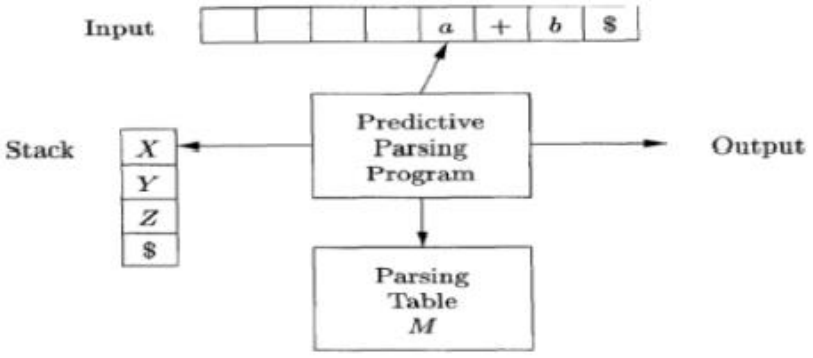
   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.
3. If X is a non-terminal

   ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \rightarrow Y_1 Y_2 ... Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, ..., Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 ... Y_k$ to represent a step of the derivation.
4. none of the above ➔ error
   - all empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.



Model of a table-driven predictive parser

- The table-driven parser in figure has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by algorithm, and an output stream.
- The input buffer contains the string to be parsed, followed by the endmarker $.
- We reuse the symbol $ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of $.



Model of a table-driven predictive parser

- The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current input symbol.
  - If X is a nonterminal, the parser chooses an X-production by consulting entry $M[X, a]$ of the parsing table M.
  - Additional code could be executed here, for example, code to construct a node in a parse tree.
  - Otherwise, it checks for a match between the terminal X and current input symbol a.

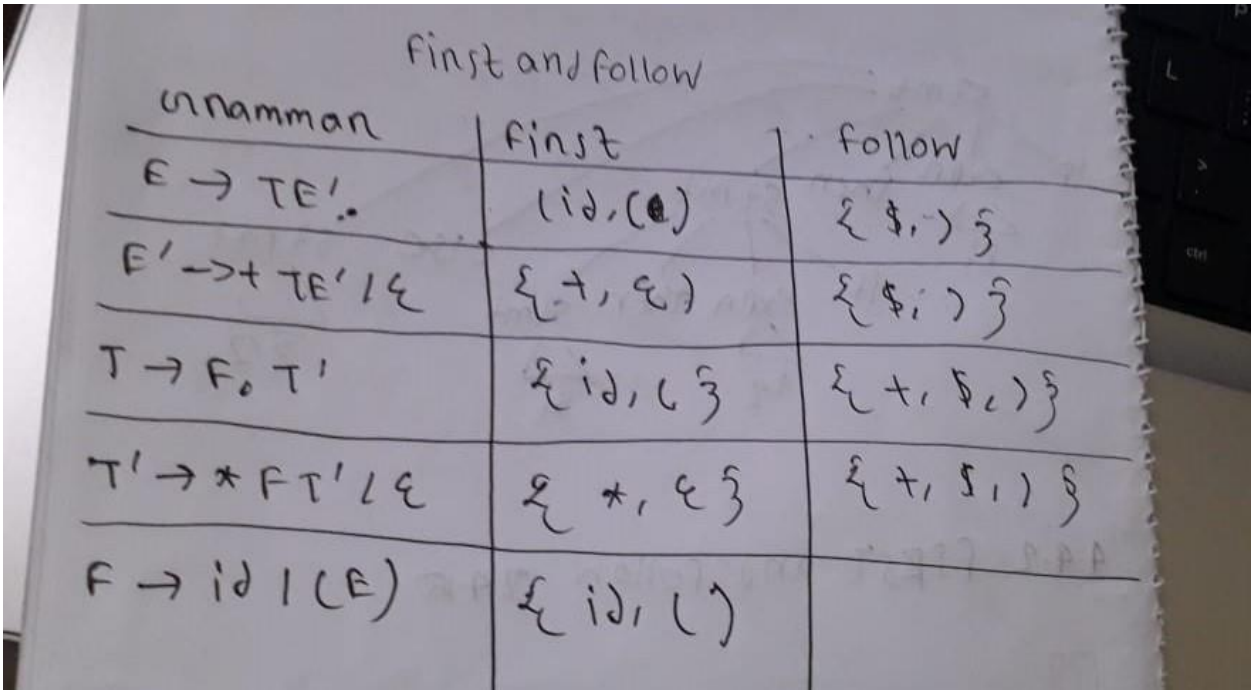
Nonrecursive Predictive Parsing — *continued*

set *ip* to point to the first symbol of w;
set X to the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if (X is a) pop the stack and advance *ip*;
    else if (X is a terminal) *error*();
    else if ($M[X, a]$ is an error entry) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
    }
    set X to the top stack symbol;
}

Predictive parsing algorithm

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\,\mathbf{id} * \mathbf{id}\$$	match **id**
$\mathbf{id}$	$E'\$$	$+\,\mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\,TE'\$$	$+\,\mathbf{id} * \mathbf{id}\$$	output $E' \to +\,TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$*\,\mathbf{id}\$$	match **id**
$\mathbf{id} + \mathbf{id}$	$*\,FT'E'\$$	$*\,\mathbf{id}\$$	output $T' \to *\,FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match **id**
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$



First and follow

Grammar	First	Follow
$E \rightarrow TE'$	{id, (}	{$,)}
$E' \rightarrow +TE' \mid \varepsilon$	{+, ε}	{$,)}
$T \rightarrow F T'$	{id, (}	{+, $,)}
$T' \rightarrow *FT' \mid \varepsilon$	{*, ε}	{+, $,)}
$F \rightarrow id \mid (E)$	{id, (}	

$A \rightarrow a\beta$

First(A) $\rightarrow \{a\}$

$A \rightarrow a \mid \varepsilon$

$A \rightarrow a'$
$A \rightarrow \varepsilon'$

$\therefore$ First(A) $= \{a, \varepsilon\}$

$A \rightarrow (a, B) \mid \varepsilon$

First(A) $= \{(, \varepsilon\}$

$A \rightarrow (a\ B) \mid \varepsilon$

First(A) $= \{(, \varepsilon)$

Example-2

Gnamman	first	follow
S → Dd/cb	{a, b d, c, b}	{ $ }
B → aB / ε	{a ε}	{d}
C → cC / ε	{c, ε}	{b}

Example-3

Gnamman	first	follow
S → a AB bc	{a}	{$}
A → c / ε	{c, ε}	{d, b}
B → d / ε	{d, ε}	{a}

Example:-4

Gnommon	FIRST	Follow
$S \to iEtSS' / a$	$\{i, a\}$	$\{\$, e\}$
$S' \to eS / \varepsilon$	$\{e, \varepsilon\}$	$\{\$, e\}$
$E \to b$	$\{b\}$	$\{t\}$

Example-5 : How to find out first and follow in LL(1)

Gnamman	First	Follow
$S \to ABCDE$	$\{a, b, c\}$	$\{\$\}$
$A \to a / \varepsilon$	$\{a, \varepsilon\}$	$\{b, c\}$
$B \to b / \varepsilon$	$\{b, \varepsilon\}$	$\{c\}$
$C \to c$	$\{c\}$	$\{d, e, \$\}$
$D \to d / \varepsilon$	$\{d, \varepsilon\}$	$\{e, \$\}$
$E \to e / \varepsilon$	$\{e, \varepsilon\}$	$\{\$\}$

Construction of LL(1) Parsing Table or LL(1) Parsen table or Predictive Parse Table.

① If you want create Parsing table using grammar remainven that must ~~the~~ be do at first. : first and follow then Parse table.

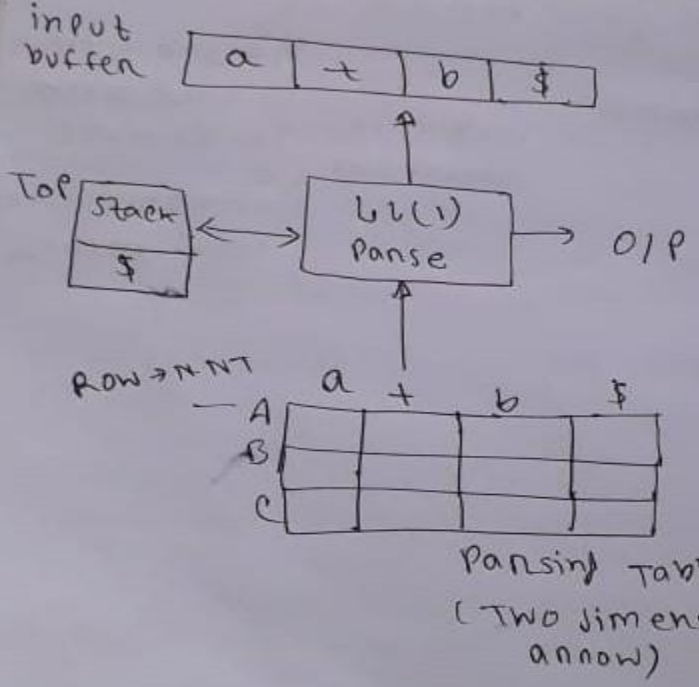
	FIRST	~~Follow~~ Follow
$E \to TE'$	$\{id, (\}$	$\{\$,)\}$
$E' \to TE'/\varepsilon$	$\{+, \varepsilon\}$	$\{\$,)\}$
$T \to FT'$	$\{id, (\}$	$\{+, \$,)\}$
$T' \to *FT'/\varepsilon$	$\{*, \varepsilon\}$	$\{+, \$,)\}$
$F \to id \mid (E)$	$\{id, (\}$	$\{*, +, \$,)\}$

LL(1) Parsing Table

Variable / non-terminal	Terminal					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

Data structure used by (LL(1)) Parser

→ i/p buffer
→ stack
→ ~~Parsing~~ Table Parsing Table.



Parsing Table
(Two dimensional array)

stack	input	action
$E	id+id+id $	E → TE'
$E'T	id+id*id $	T → FT'
$E'T'F	id+id*id $	F → id
$E'T'id	id+id*id $	
$E'T'	+id*id $	T' → ε
$E'	+id*id $	E' → +TE'
$E'T+	+id*id $	
$E'T	id*id $	T → FT'
$E'T'F	id*id $	F → id
$E'T'id	id*id $	
$E'T'	*id $	T' → *FT'
$E'T'F*	*id $	
$E'T'F	id $	F → id

$E'T'id	id$	
$E'T'	$	T' → ε
$	$	Accepted

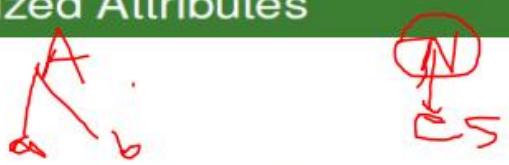
Chapter-5

Syntax Directed Translation

Syntax-Directed Definitions

- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules.
- Attributes are associated with grammar symbols and rules are associated with productions.
- If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse-tree node labeled X.
- Attributes may be of any kind: numbers, types, table references, or strings, for instance.

Inherited and Synthesized Attributes



We shall deal with two kinds of attributes for nonterminals:

1. A synthesized attribute for a nonterminal *A* at a parse-tree node *N* is defined by a semantic rule associated with the production at *N*.

   Note that the production must have *A* as its head.

   A synthesized attribute at node *N* is defined only in terms of attribute values at the children of *N* and at *N* itself.

2. An inherited attribute for a nonterminal *B* at a parse-tree node *N* is defined by a semantic rule associated with the production at the parent of *N*.

   Note that the production must have *B* as a symbol in its body.

   An inherited attribute at node *N* is defined only in terms of attribute values at *N*'s parent, *N* itself, and *N*'s siblings.

Inherited and Synthesized Attributes — *continued*

- While we do not allow an inherited attribute at node N to be defined in terms of attribute values at the children of node N, we do allow a synthesized attribute at node N to be defined in terms of inherited attribute values at node N itself.
- Terminals can have synthesized attributes, but not inherited attributes.
- Attributes for terminals have lexical values that are supplied by the lexical analyzer.
- There are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

Example

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

Syntax-directed definition of a simple desk calculator

- The SDD is based on our familiar grammar for arithmetic expressions with operators $+$ and $*$.
- An SDD that involves only synthesized attributes is called *S*-attributed.
- The SDD in figure has this property.
  - In an *S*-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

Example — *continued*

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
2)	$E \rightarrow E_1\ +\ T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1\ *\ F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (\ E\ )$	$F.val = E.val$
7)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}$.lexval

Syntax-directed definition of a simple desk calculator

- The rule for production 1, $L \rightarrow E\mathbf{n}$, sets $L.val$ to $E.val$, which we shall see is the numerical value of the entire expression.
- Production 2, $E \rightarrow E_1 + T$, also has one rule, which computes the *val* attribute for the head E as the sum of the values at E_1 and T.
- At any parse-tree node N labeled E, the value of *val* for E is the sum of the values of *val* at the children of node N labeled E and T.
- Production 3, $E \rightarrow T$, has a single rule that defines the value of *val* for E to be the same as the value of *val* at the child for T.
- Production 4 is similar to the second production.
- Its rule multiplies the values at the children instead of adding them.
- The rules for productions 5 and 6 copy values at a child, like that for the third production.
- Production 7 gives $F.val$ the value of a digit, that is, the numerical value of the token **digit** that the lexical analyzer returned.

Evaluating an SDD at the Nodes of a Parse Tree

- To visualize the translation specified by an SDD, it helps to work with parse trees.
- Even though a translator need not actually build a parse tree.
- Imagine therefore that the rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree.
- A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.

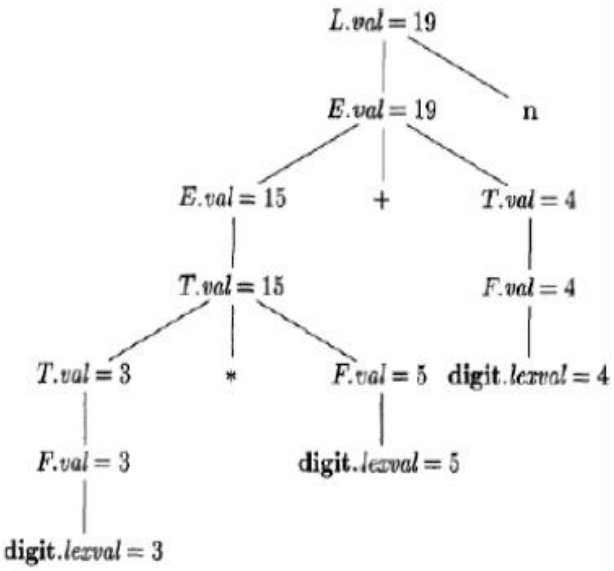
Evaluating an SDD at the Nodes of a Parse Tree — *continued*

- How do we construct an annotated parse tree?
- In what order do we evaluate attributes?
- Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.
- If all attributes are synthesized, then we must evaluate the attributes at all of the children of a node before we can evaluate the attribute at the node itself.
- With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

Example

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

Syntax-directed definition of a simple desk calculator



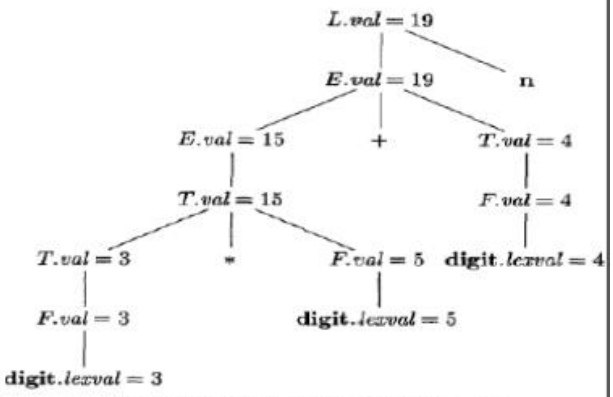
Annotated parse tree for 3 * 5 + 4 **n**

- Figure shows an annotated parse tree for the input string 3 * 5 + 4 **n**, constructed using the grammar and rules.

Example

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

Syntax-directed definition of a simple desk calculator



Annotated parse tree for 3 * 5 + 4 **n**

- The values of *lexval* are presumed supplied by the lexical analyzer.
- Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order, and we see the resulting values associated with each node.
- For instance, at the node with a child labeled $*$, after computing $T.val = 3$ and $F.val = 5$ at its first and third children, we apply the rule that says $T.val$ is the product of these two values, or 15.

Parse Tree Using Inherited Attributes

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

An SDD based on a grammar suitable for top-down parsing

- The SDD in figure computes terms like $3 * 5$ and $3 * 5 * 7$.
- The top-down parse of input $3 * 5$ begins with the production $T \rightarrow FT'$.
- Here, F generates the digit 3, but the operator $*$ is generated by T'.

Example

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * F\, T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

An SDD based on a grammar suitable for top-down parsing

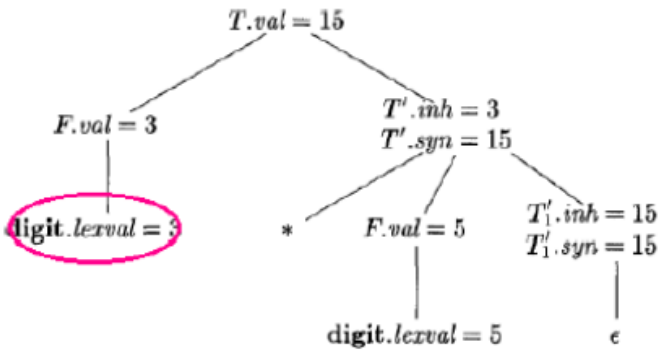
- Thus, the left operand 3 appears in a different subtree of the parse tree from $*$.
- An inherited attribute will therefore be used to pass the operand to the operator.
  - Each of the nonterminals T and F has a synthesized attribute *val*.
  - The terminal **digit** has a synthesized attribute *lexval*.
  - The nonterminal T' has two attributes: an inherited attribute *inh* and a synthesized attribute *syn*.

Example

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * F\, T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

An SDD based on a grammar suitable for top-down parsing

- The semantic rules are based on the idea that the left operand of the operator $*$ is inherited.
- More precisely, the head T' of the production $T' \rightarrow *FT_1'$ inherits the left operand of $*$ in the production body.
- Given a term $x * y * z$, the root of the subtree for $*y * z$ inherits x.
- Then, the root of the subtree for $*z$ inherits the value of $x * y$, and so on, if there are more factors in the term.
- Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.

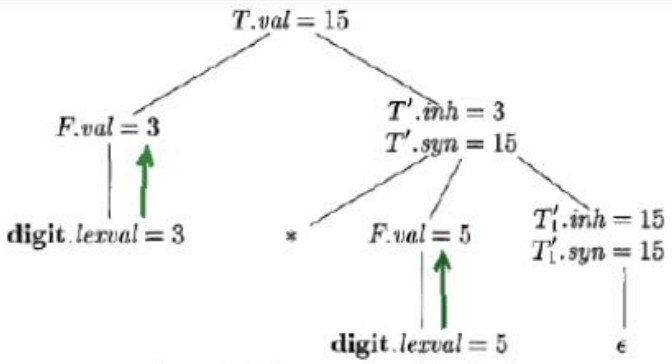


Annotated parse tree for 3 * 5

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

An SDD based on a grammar suitable for top-down parsing

- To see how the semantic rules are used, consider the annotated parse tree for 3 * 5 in figure.
- The leftmost leaf in the parse tree, labeled **digit**, has attribute value *lexval* = 3, where the 3 is supplied by the lexical analyzer.

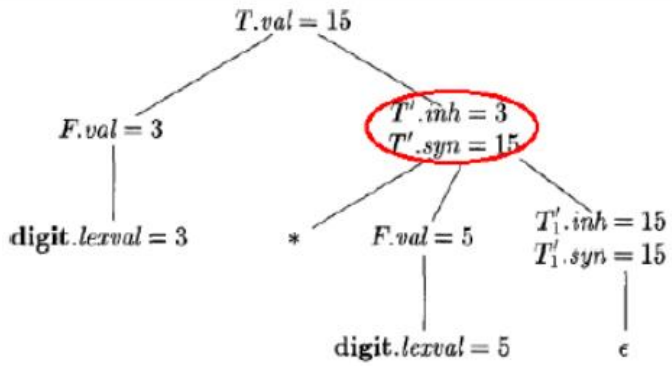


Annotated parse tree for 3 * 5

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**.*lexval*

An SDD based on a grammar suitable for top-down parsing

- Its parent is for production 4, $F \rightarrow$ **digit**.
- The only semantic rule associated with this production defines $F.val =$ **digit**.*lexval*, which equals 3.

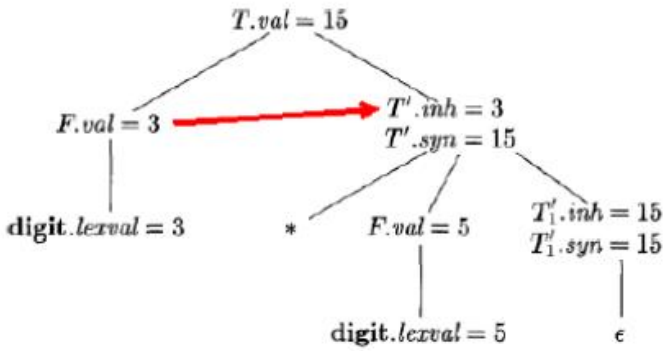


Annotated parse tree for 3 * 5

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

An SDD based on a grammar suitable for top-down parsing

- At the second child of the root, the inherited attribute $T'.inh$ is defined by the semantic rule $T'.inh = F.val$ associated with production 1.

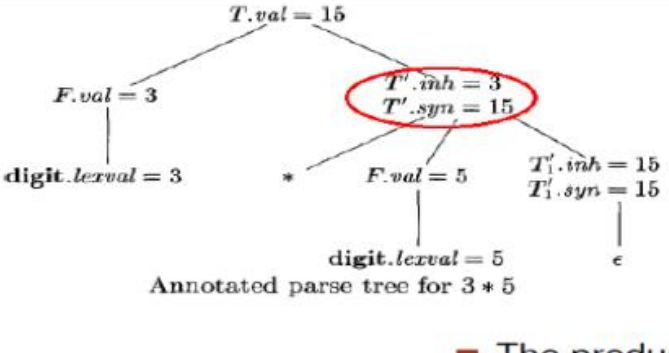


Annotated parse tree for 3 * 5

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

An SDD based on a grammar suitable for top-down parsing

- Thus, the left operand, 3, for the * operator is passed from left to right across the children of the root.
- The inherited attribute $T'.inh$ is defined by the semantic rule $T'_1.inh = T'.inh \times F.val$ associated with production 2.

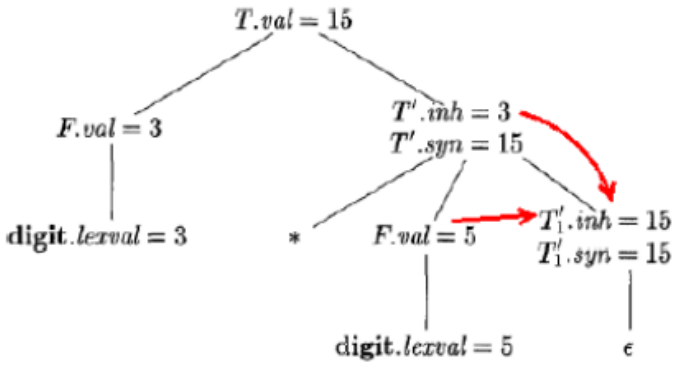


Annotated parse tree for 3 * 5

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

An SDD based on a grammar suitable for top-down parsing

- The production at the node for T' is $T' \rightarrow *FT'_1$.
- We retain the subscript 1 in the annotated parse tree to distinguish between the two nodes for T'.

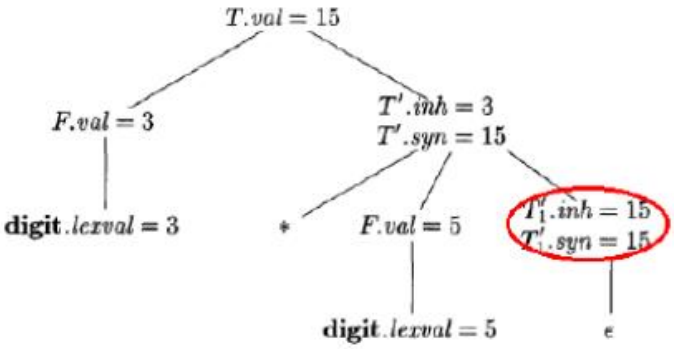


Annotated parse tree for $3 * 5$

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**.*lexval*

An SDD based on a grammar suitable for top-down parsing

- With $T'.inh = 3$ and $F.val = 5$, we get $T_1'.inh = 15$.

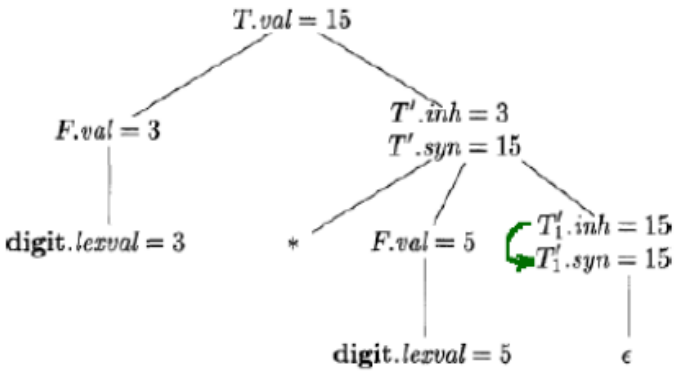


Annotated parse tree for 3 * 5

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

An SDD based on a grammar suitable for top-down parsing

- At the lower node for T'_1, the production is $T' \rightarrow \epsilon$.

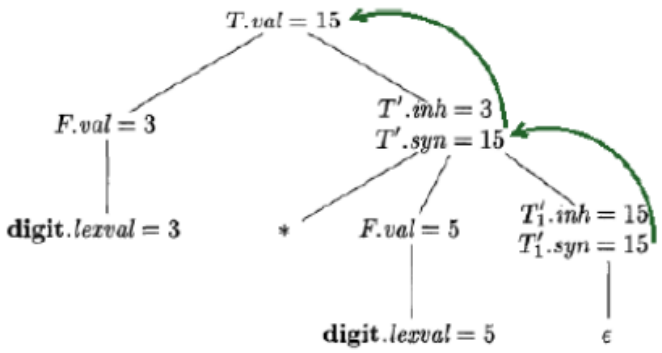


Annotated parse tree for $3 * 5$

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * F\, T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

An SDD based on a grammar suitable for top-down parsing

- The semantic rule $T'.syn = T'.inh$ defines $T'_1.syn = 15$.



Annotated parse tree for $3 * 5$

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow * F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

An SDD based on a grammar suitable for top-down parsing

- The *syn* attributes at the nodes for T' pass the value 15 up the tree to the node for T, where $T.val = 15$.

Dependency Graphs

- A dependency graph depicts the flow of information among the attribute instances in a particular parse tree.
- An edge from one attribute instance to another means that the value of the first is needed to compute the second.
- Edges express constraints implied by the semantic rules.

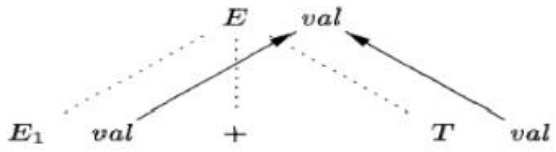
- For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.

Example

- Consider the following production and rule:

PRODUCTION	Semantic Rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

- At every node N labeled E, with children corresponding to the body of this production, the synthesized attribute val at N is computed using the values of val at the two children, labeled E and T.

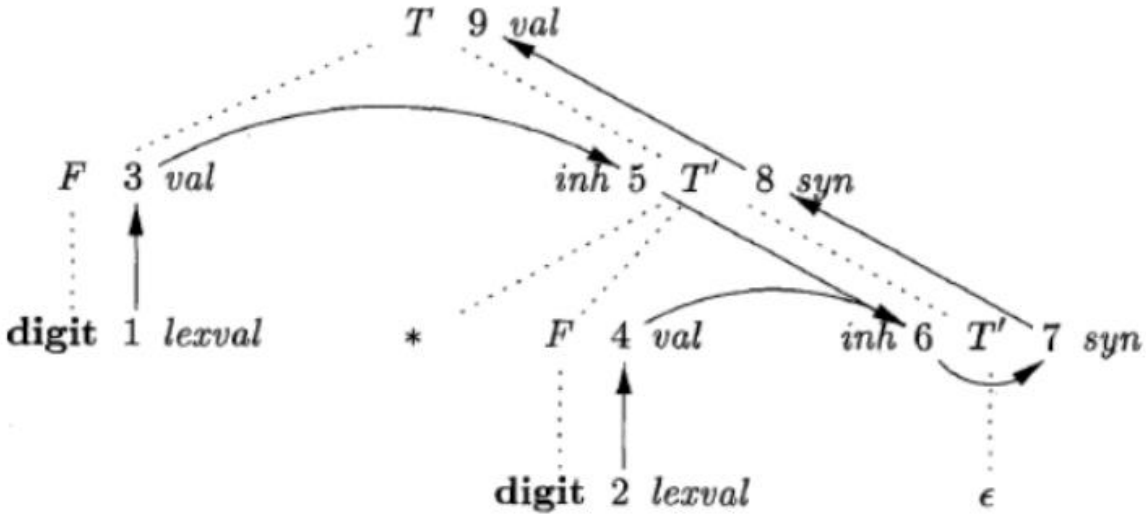


$E.val$ is synthesized from $E_1.val$ and $E_2.val$

- As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

Example

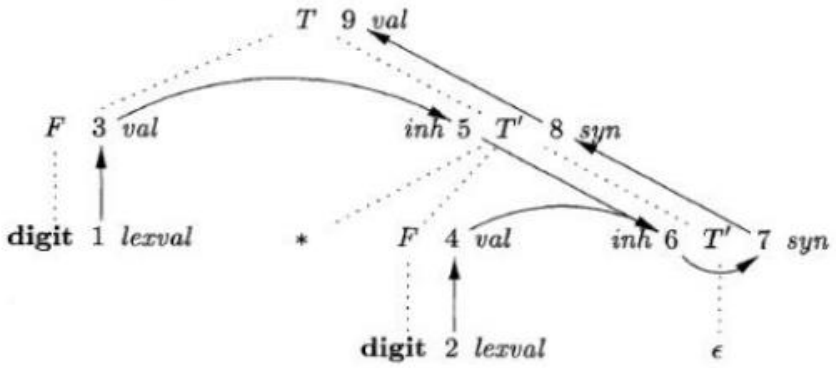
- An example of a complete dependency graph appears in figure.



Dependency graph for the annotated parse tree

Annotated parse tree for $3 * 5$

- Nodes 1 and 2 represent the attribute *lexval* associated with the two leaves labeled digit.
- Nodes 3 and 4 represent the attribute *val* associated with the two nodes labeled F.
- The edges to node 3 from 1 and to node 4 from 2 result from the semantic rule that defines $F.val$ in terms of **digit**.*lexval*.
- In fact, $F.val$ equals **digit**.*lexval*, but the edge represents dependence, not equality.
- Nodes 5 and 6 represent the inherited attribute $T'.inh$ associated with each of the occurrences of nonterminal T'.
- The edge to 5 from 3 is due to the rule $T'.inh = F.val$, which defines $T'.inh$ at the right child of the root from $F.val$ at the left child.
- We see edges to 6 from node 5 for $T'.inh$ and from node 4 for $F.val$, because these values are multiplied to evaluate the attribute *inh* at node 6.
- Nodes 7 and 8 represent the synthesized attribute *syn* associated with the occurrences of T'.
- The edge to node 7 from 6 is due to the semantic rule $T'.syn = T'.inh$ associated with production 3.
- The edge to node 8 from 7 is due to a semantic rule associated with production 2.
- The edge to node 7 from 6 is due to the semantic rule $T'.syn = T'.inh$ associated with production 3.



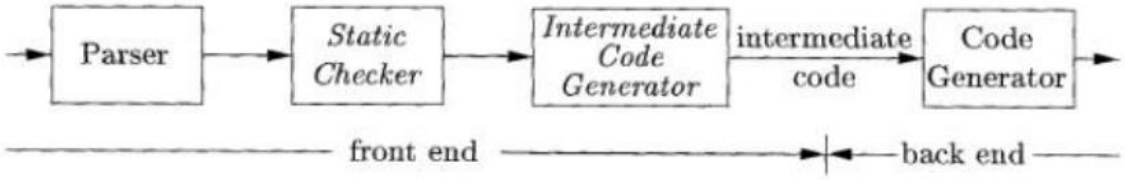
Dependency graph for the annotated parse tree

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

- Finally, node 9 represents the attribute $T.val$.
- The edge to 9 from 8 is due to the semantic rule, $T.val = T'.syn$, associated with production 1.

Chapter-6

Intermediate-Code Generation



Logical structure of a compiler front end

Variants of Syntax Trees

- Nodes in a syntax tree represent constructs in the source program.
- The children of a node represent the meaningful components of a construct.
- A directed acyclic graph (hereafter called a DAG) for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression.
- As we shall see, DAG's can be constructed by using the same techniques that construct syntax trees.

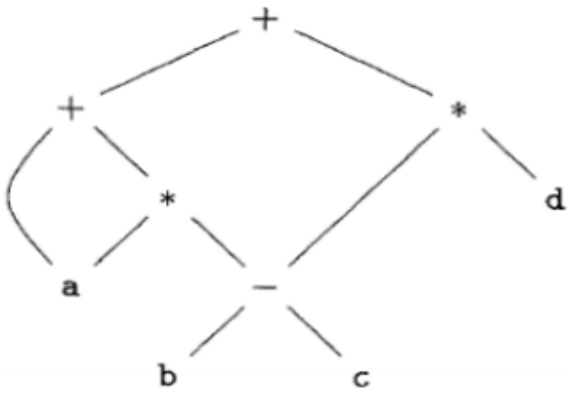
Directed Acyclic Graphs for Expressions

- Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators.
- The difference is that a node *N* in a DAG has more than one parent if *N* represents a common subexpression.
- In a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.
- Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example

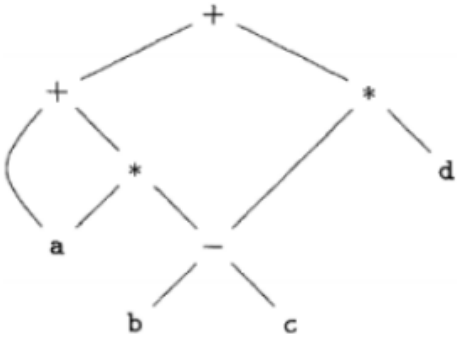
- Figure shows the DAG for the expression,

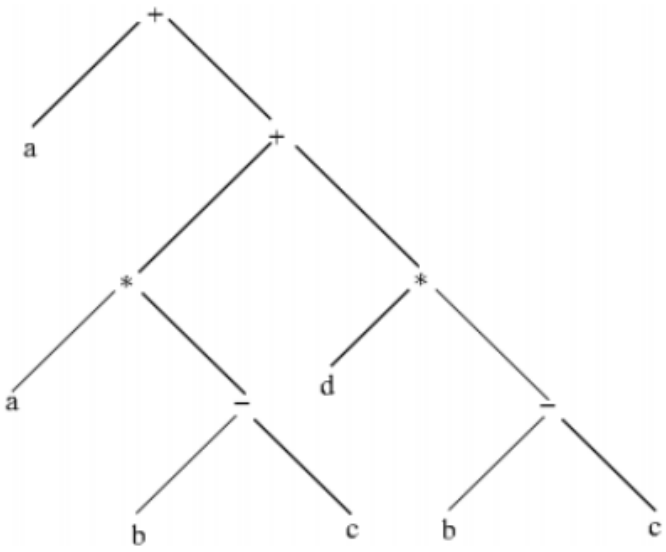
$$a + a * (b - c) + (b - c) * d$$



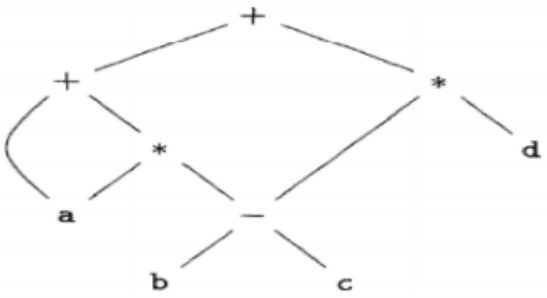
Dag for the expression a + a * (b - c) + (b - c) * d

$$a + a * (b - c) + (b - c) * d$$





$a + a * (b - c) + (b - c) * d$



- The leaf for a has two parents, because a appears twice in the expression.
- More interestingly, the two occurrences of the common subexpression $b - c$ are represented by one node, the node labeled −.

- That node has two parents, representing its two uses in the subexpressions $a * (b - c)$ and $(b - c) * d$.
- Even though b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression $b - c$.

- The SDD of figure can construct either syntax trees or DAG's.

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf(\textbf{num}, \textbf{num}.val)$

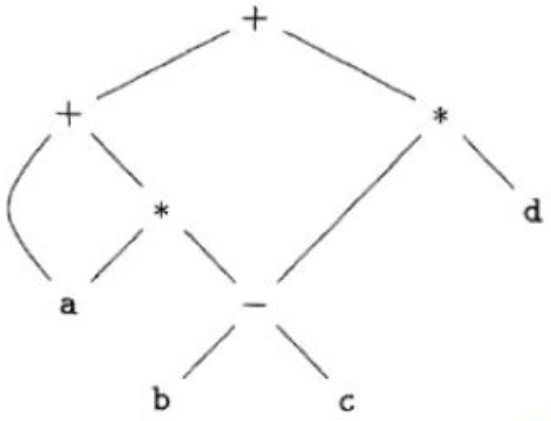
Syntax-directed definition to produce syntax trees or DAG's

- It will construct a DAG if, before creating a new node, these functions first check whether an identical node already
  - If a previously created identical node exists, the existing node is returned.
  - For instance, before constructing a new node, *Node(op, left, right)* we check whether there is already a node with label *op*, and children *left* and *right*, in that order.
  - If so, *Node* returns the existing node; otherwise, it creates a new node.

Example

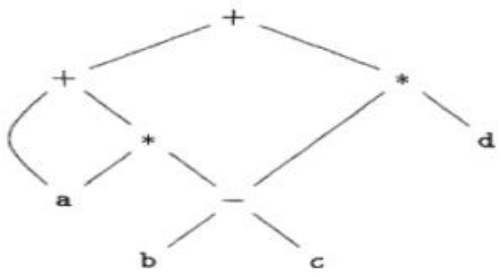
- The sequence of steps shown in figure constructs the DAG.

1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$



a + a * (b − c) + (b − c) * d

1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$



- The sequence provided *Node* and *Leaf* return an existing node, if possible, as discussed above.
- We assume that *entry-a* points to the symbol-table entry for a, and similarly for the other identifiers.
- When the call to *Leaf*(**id**, *entry-a*) is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$.
- Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_9 = p_4$).
- Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_5$.

Three Address Code

Addresses and Instructions — *continued*

Here is a list of the common three-address instruction forms:

1. Assignment instructions of the form $x = y\ op\ z$, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.
2. Assignments of the form $x = op\ y$, where *op* is a unary operation.
   - Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number.
3. Copy instructions of the form $x = y$, where x is assigned the value of y.
4. An unconditional jump `goto` *L*.
   - The three-address instruction with label *L* is the next to be executed.
5. Conditional jumps of the form `if` *x* `goto` *L* and `ifFalse` *x* `goto` *L*.

6. Conditional jumps such as `if` *x* *relop* *y* `goto` *L*, which apply a relational operator (<, ==, >=, etc.) to *x* and *y*, and execute the instruction with label *L* next if *x* stands in relation *relop* to *y*.
   - If not, the three-address instruction following `if` *x relop y* `goto` *L* is executed next, in sequence.

Here is a list of the common three-address instruction forms:

7. Procedure calls and returns are implemented using the following instructions:
   - `param` *x* for parameters
   - `call` *p*, *n* for procedure calls
   - *y* = `call` *p*, *n* for function calls
   - `return` *y*, where *y*, representing a returned value, is optional.
8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$.
   - The instruction $x = y[i]$ sets x to the value in the location i memory units beyond location y.
   - The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value of y.
9. Address and pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$.

Example — *continued*

```
do i = i+1; while (a[i] < v);
```

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

(a) Symbolic labels.

```
100: t1 = i + 1
101: i = t1
102: t2 = i * 8
103: t3 = a [ t2 ]
104: if t3 < v goto 100
```

(b) Position numbers.

- The translation in (a) uses a symbolic label *L*, attached to the first instruction.
- The translation in (b) shows position numbers for the instructions, starting arbitrarily at position 100.

- In both translations, the last instruction is a conditional jump to the first instruction.
- The multiplication `i * 8` is appropriate for an array of elements that each take 8 units of space.

Representation of Three Address Code

- 3 Types :

1. Quadruples
2. Triples
3. Indirect Triples

Quadruples — *continued*

- A quadruple (or just "quad") has four fields, which we call *op*, arg_1, arg_2, and *result*.
- The op field contains an internal code for the operator.
- For instance, the three-address instruction $x = y + z$ is represented by placing + in *op*, y in arg_1, z in arg_2, and x in *result*.

- The following are some exceptions to this rule:
  1. Instructions with unary operators like $x =$ `minus` y or $x = y$ do not use arg_2.
     - Note that for a copy statement like $x = y$, *op* is =, while for most other operations, the assignment operator is implied.
  2. Operators like `param` use neither arg_2 nor *result*.
  3. Conditional and unconditional jumps put the target label in *result*.

Example

- Three-address code for the assignment `a = b * -c + b * -c;` appears in figure.

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Three-address code and its quadruple representation

Triples

- A triple has only three fields, which we call *op*, arg_1, arg_2.
- Note that the *result* field in (b) is used primarily for temporary names.

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

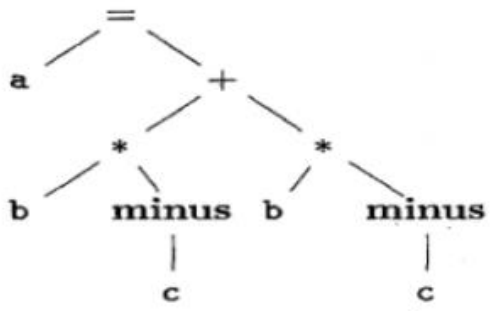
	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

- Parenthesized numbers represent pointers into the triple structure itself.
- Previously, positions or pointers to positions were called value numbers.

Example

- The syntax tree and triples in figure correspond to the three-address code and quadruples.



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

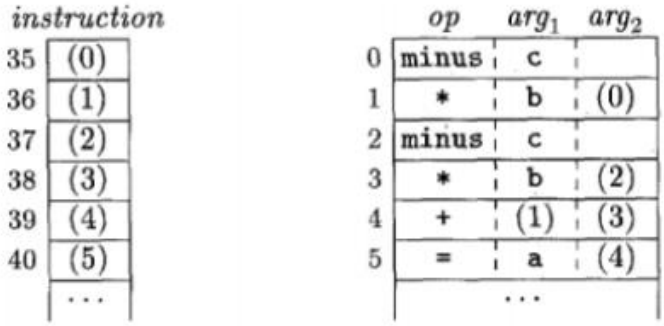
Representations of `a = b * - c + b * - c ;`

- In the triple representation in (b), the copy statement $a = t_5$ is encoded in the triple representation by placing `a` in the arg_1 field and (4) in the arg_2 field.

Indirect Triples

- *Indirect triples* consist of a listing of pointers to triples, rather than a listing of triples themselves.
- For example, let us use an array instruction to list pointers to triples in the desired order.

- Then, the triples in (b) might be represented as in this figure.



instruction	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Indirect triples representation of three-address code

6.1.1 Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example 6.1: Figure 6.3 shows the DAG for the expression

```
a + a * (b - c) + (b - c) * d
```

The leaf for `a` has two parents, because `a` appears twice in the expression. More interestingly, the two occurrences of the common subexpression `b-c` are represented by one node, the node labeled $-$. That node has two parents, representing its two uses in the subexpressions `a*(b-c)` and `(b-c)*d`. Even though `b` and `c` appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression `b-c`. □

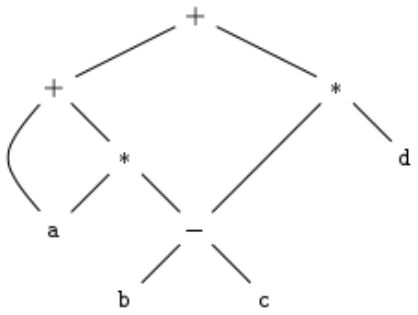


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

The SDD of Fig. 6.4 can construct either syntax trees or DAG's. It was used to construct syntax trees in Example 5.11, where functions *Leaf* and *Node* created a fresh node each time they were called. It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned. For instance, before constructing a new node, *Node*(*op*, *left*, *right*), we check whether there is already a node with label *op*, and children *left* and *right*, in that order. If so, *Node* returns the existing node; otherwise, it creates a new node.

Example 6.2: The sequence of steps shown in Fig. 6.5 constructs the DAG in Fig. 6.3, provided *Node* and *Leaf* return an existing node, if possible, as

	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

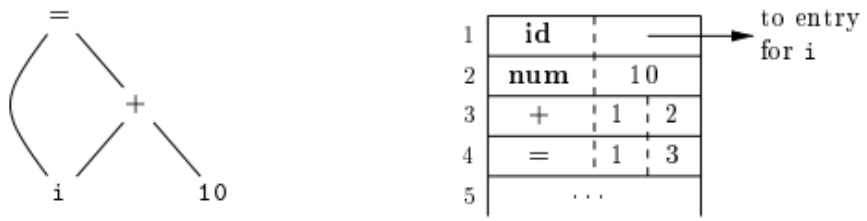
Figure 6.5: Steps for constructing the DAG of Fig. 6.3

discussed above. We assume that *entry-a* points to the symbol-table entry for a, and similarly for the other identifiers.

When the call to $Leaf(\textbf{id}, entry\text{-}a)$ is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_9 = p_4$). Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_5$. □

6.1.2 The Value-Number Method for Constructing DAG's

Often, the nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 6.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 6.6(b), leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and interior nodes have two additional fields indicating the left and right children.



(a) DAG (b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer historically has been called the *value number* for the node or for the expression represented by the node. For instance, in Fig. 6.6, the node labeled + has value number 3, and its left and right children have value numbers 1 and 2, respectively. In practice, we could use pointers to records or references to objects instead of integer indexes, but we shall still refer to the reference to a node as its "value number." If stored in an appropriate data structure, value numbers help us construct expression DAG's efficiently; the next algorithm shows how.

Suppose that nodes are stored in an array, as in Fig. 6.6, and each node is referred to by its value number. Let the *signature* of an interior node be the triple $\langle op, l, r \rangle$, where op is the label, l its left child's value number, and r its right child's value number. A unary operator may be assumed to have $r = 0$.

6.2.1 Addresses and Instructions

Three-address code is built from two concepts: addresses and instructions. In object-oriented terms, these concepts correspond to classes, and the various kinds of addresses and instructions correspond to appropriate subclasses. Alternatively, three-address code can be implemented using records with fields for the addresses; records called quadruples and triples are discussed briefly in Section 6.2.2.

An address can be one of the following:

- *A name.* For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol table entry, where all information about the name is kept.

- *A constant.* In practice, a compiler must deal with many different types of constants and variables. Type conversions within expressions are considered in Section 6.5.2.

- *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

We now consider the common three-address instructions used in the rest of this book. Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a three-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by "backpatching," discussed in Section 6.7. Here is a list of the common three-address instruction forms:

1. Assignment instructions of the form x = y *op* z, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form x = *op* y, where *op* is a unary operation. Essential unary operations include unary minus, logical negation, and conversion operators that, for example, convert an integer to a floating-point number.

3. *Copy instructions* of the form x = y, where x is assigned the value of y.

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y. If not, the three-address instruction following `if` x *relop* y `goto` L is executed next, in sequence.

7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y = `call` p, n for procedure and function calls, respectively; and `return` y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

```
param x1
param x2
...
param xn
call p, n
```

   generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$. The integer n, indicating the number of actual parameters in "`call` p, n," is not redundant because calls can be nested. That is, some of the first `param` statements could be parameters of a call that comes after p returns its value; that value becomes another parameter of the later call. The implementation of procedure calls is outlined in Section 6.9.

8. Indexed copy instructions of the form x = y`[`i`]` and x`[`i`]` = y. The instruction x = y`[`i`]` sets x to the value in the location i memory units beyond location y. The instruction x`[`i`]` = y sets the contents of the location i units beyond x to the value of y.

9. Address and pointer assignments of the form x = `&` y, x = `*` y, and `*` x = y. The instruction x = `&` y sets the r-value of x to be the location (l-value) of y.[2] Presumably y is a name, perhaps a temporary, that denotes an expression with an l-value such as `A[i][j]`, and x is a pointer name or temporary. In the instruction x = `*` y, presumably y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location. Finally, `*` x = y sets the r-value of the object pointed to by x to the r-value of y.

Example 6.5: Consider the statement

```
do i = i+1; while (a[i] < v);
```

Two possible translations of this statement are shown in Fig. 6.9. The translation in Fig. 6.9(a) uses a symbolic label L, attached to the first instruction.

The translation in (b) shows position numbers for the instructions, starting arbitrarily at position 100. In both translations, the last instruction is a conditional jump to the first instruction. The multiplication `i * 8` is appropriate for an array of elements that each take 8 units of space. □

```
L:  t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a [ t2 ]
    if t3 < v goto L
```

(a) Symbolic labels.

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set clearly must be rich enough to implement the operations in the source language. Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine. However, if the front end must generate long sequences of instructions for some source-language operations, then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations.

Chapter 7

Run - Time Environment

7.2.2 Activation Records

Procedure calls and returns are usually managed by a run-time stack called the *control stack*. Each live activation has an *activation record* (sometimes called a *frame*) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides. The latter activation has its record at the top of the stack.

Example 7.3: If control is currently in the activation $q(2, 3)$ of the tree of Fig. 7.4, then the activation record for $q(2, 3)$ is at the top of the control stack. Just below is the activation record for $q(1, 3)$, the parent of $q(2, 3)$ in the tree. Below that is the activation record $q(1, 9)$, and at the bottom is the activation record for m, the main function and root of the activation tree. □

We shall conventionally draw control stacks with the bottom of the stack higher than the top, so the elements in an activation record that appear lowest on the page are actually closest to the top of the stack.

The contents of activation records vary with the language being implemented. Here is a list of the kinds of data that might appear in an activation record (see Fig. 7.5 for a summary and possible order for these elements):

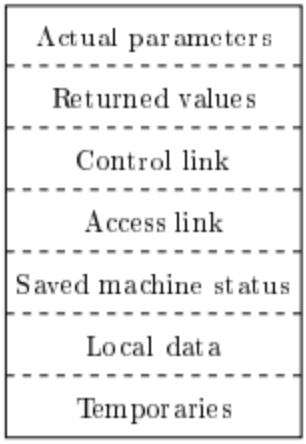


Figure 7.5: A general activation record

1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
2. Local data belonging to the procedure whose activation record this is.
3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

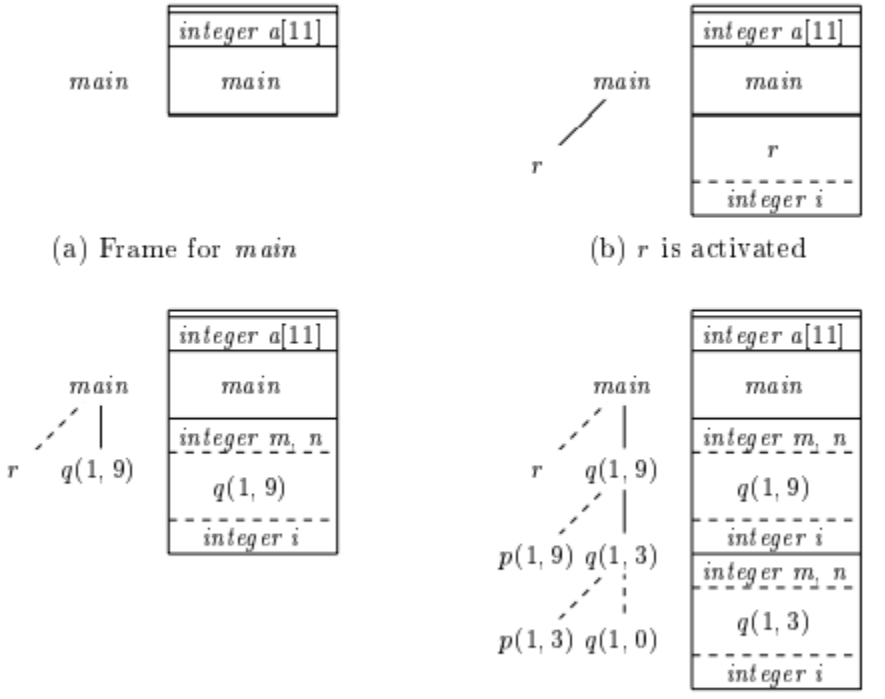
4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record. Access links are discussed in Section 7.3.5.

5. A *control link*, pointing to the activation record of the caller.

6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

Example 7.4: Figure 7.6 shows snapshots of the run-time stack as control flows through the activation tree of Fig. 7.4. Dashed lines in the partial trees go to activations that have ended. Since array a is global, space is allocated for it before execution begins with an activation of procedure *main*, as shown in Fig. 7.6(a).



(a) Frame for *main*

(b) r is activated

(c) r has been popped and $q(1, 9)$ pushed

(d) Control returns to $q(1, 3)$

Figure 7.6: Downward-growing stack of activation records

When control reaches the first call in the body of *main*, procedure r is activated, and its activation record is pushed onto the stack (Fig. 7.6(b)). The activation record for r contains space for local variable i. Recall that the top of stack is at the bottom of diagrams. When control returns from this activation, its record is popped, leaving just the record for *main* on the stack.

Control then reaches the call to q (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack, as in Fig. 7.6(c). The activation record for q contains space for the parameters m and n and the local variable i, following the general layout in Fig. 7.5. Notice that space once used by the call of r is reused on the stack. No trace of data local to r will be available to $q(1, 9)$. When $q(1, 9)$ returns, the stack again has only the activation record for *main*.