CS432F/CSL 728:
Compiler Design
July 2004

S. Arun-Kumar
`sak@cse.iitd.ernet.in`
Department of Computer Science and Engineering
I. I. T. Delhi, Hauz Khas, New Delhi 110 016.

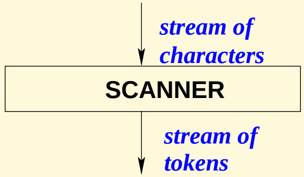
July 30, 2004

Contents

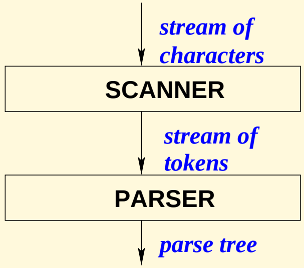
Contd . . .

Contents

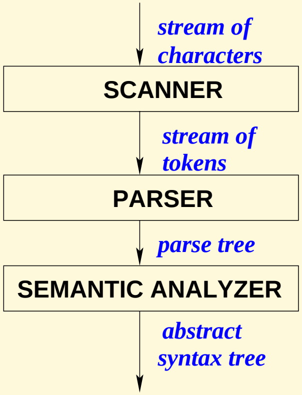
The Big Picture: 1



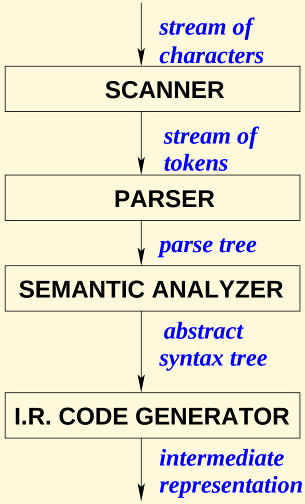
The Big Picture: 2



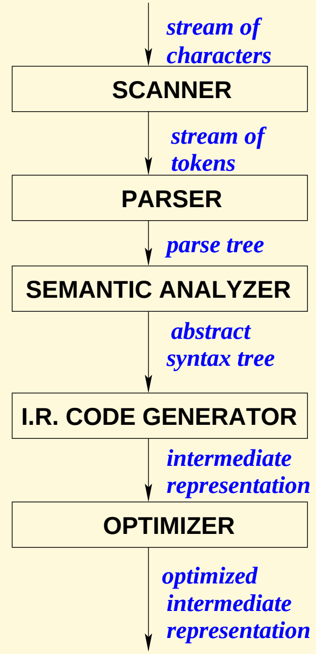
The Big Picture: 3



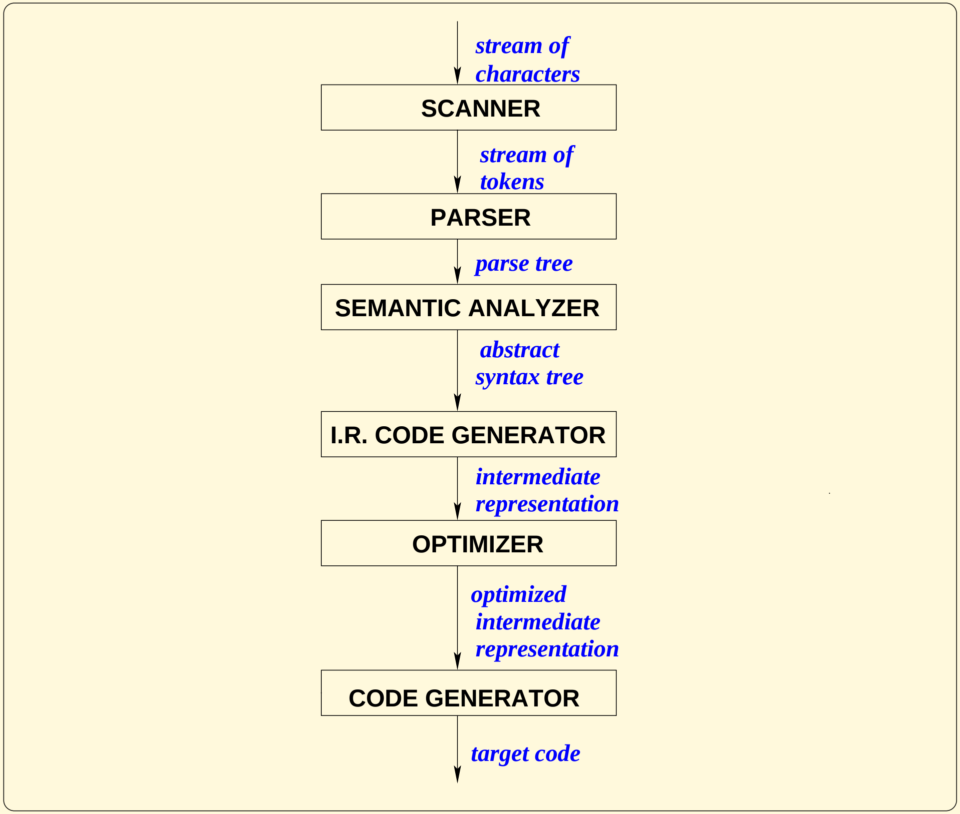
The Big Picture: 4



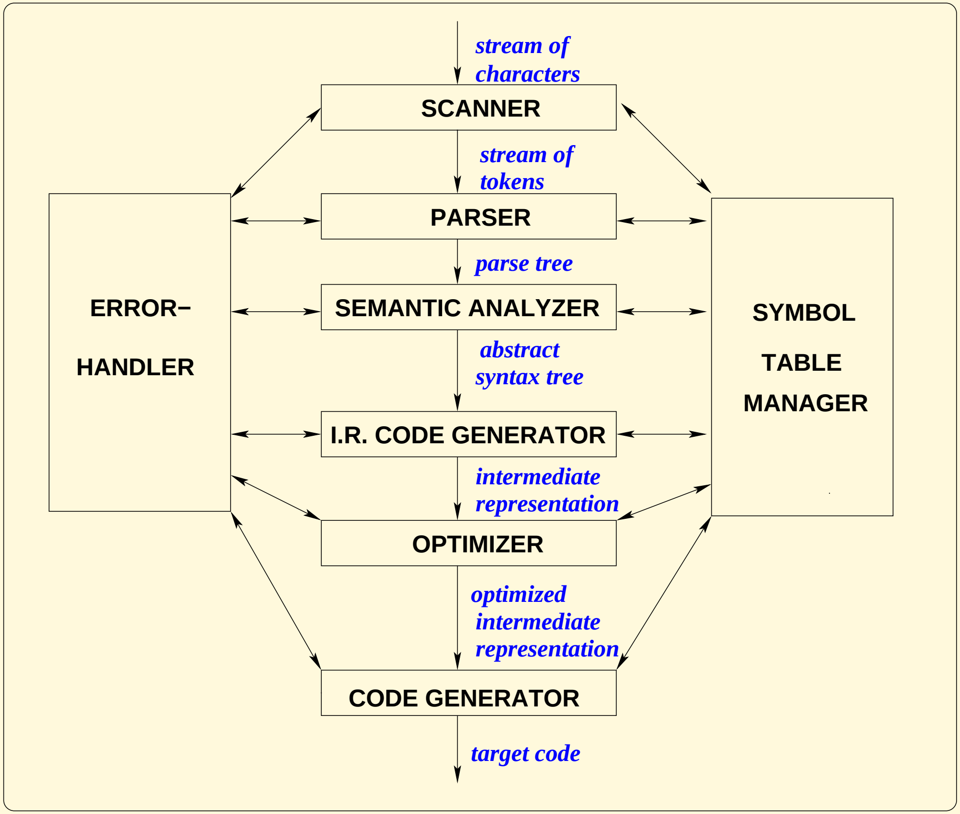
The Big Picture: 5



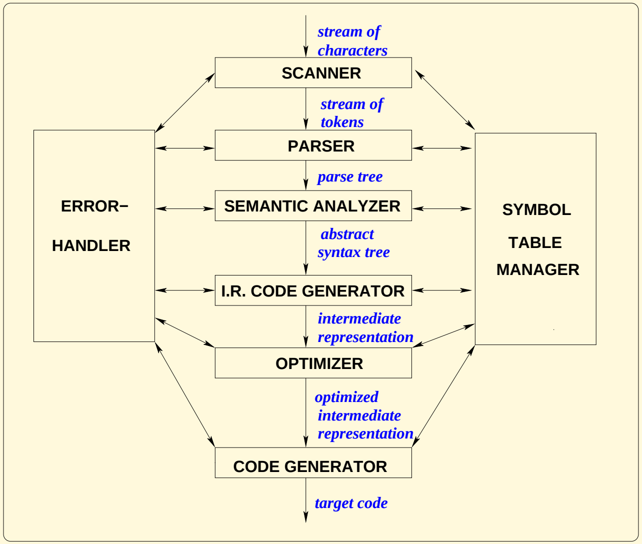
The Big Picture: 6



The Big Picture: 7



The Big Picture: 7



Scanner	Parser	Semantic Analysis	Symbol Table
IR	Optimization		Contents

Scanning: 1

- Takes a stream of characters and identifies tokens from the lexemes.

- Eliminates comments and redundant whitespace.

- Keeps track of line numbers and column numbers and passes them as parameters to the other phases to enable error-reporting to the user.

Scanning: 2

- Whitespace: A sequence of space, tab, newline, carriage-return, form-feed characters etc.
- Lexeme: A sequence of non-whitespace characters delimited by whitespace or special characters (e.g. operators like +, -, *).
- Examples of lexemes.
  - reserved words, keywords, identifiers etc.
  - Each comment is usually a single lexeme
  - preprocessor directives

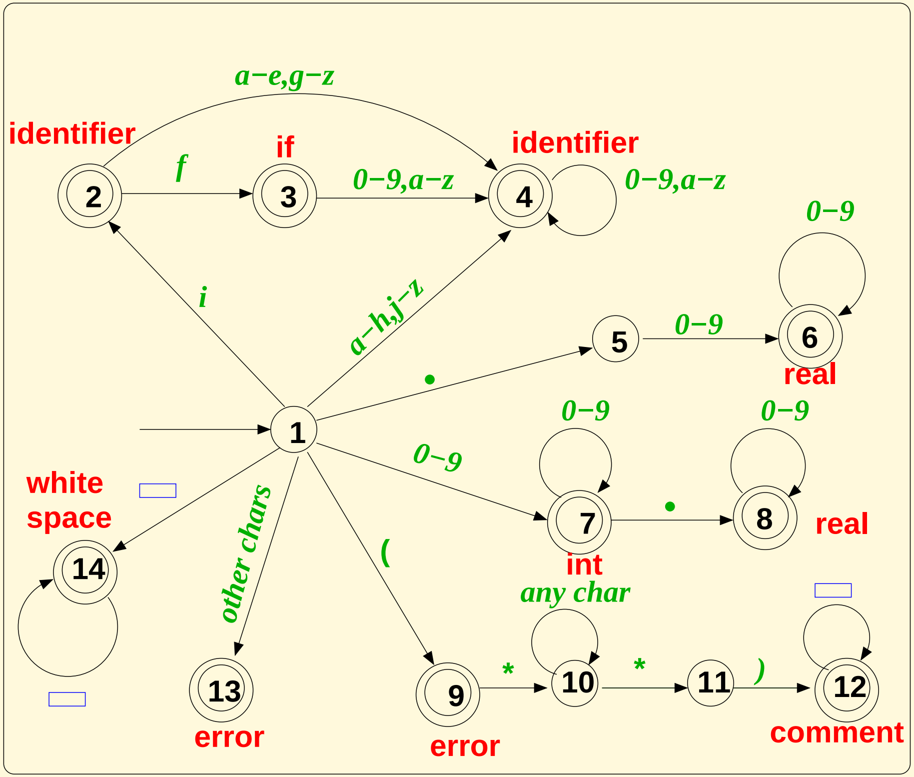
Scanning: 3

- Token: A sequence of characters to be treated as a single unit.
- Examples of tokens.
  - Reserved words (e.g. **begin**, **end**, **struct**, **if** etc.)
  - Keywords (*integer*, *true* etc.)
  - Operators (+, &&, ++ etc)
  - Identifiers (variable names, procedure names, parameter names)
  - Literal constants (numeric, string, character constants etc.)
  - Punctuation marks (:, , etc.)

Scanning: 4

- Identification of tokens is usually done by a Deterministic Finite-state automaton (DFA).
- The set of tokens of a language is represented by a large regular expression.
- This regular expression is fed to a lexical-analyser generator such as Lex, Flex or ML-Lex.
- A giant DFA is created by the Lexical analyser generator.

Scanning: 5



The Big Picture

Home Page
Title Page
Page 18 of 100
Go Back
Full Screen
Close
Quit

Syntax Analysis

Consider the following two languages over an alphabet $A = \{a, b\}$.

$$\begin{aligned} R &= \{a^n b^n | n < 100\} \\ P &= \{a^n b^n | n > 0\} \end{aligned}$$

- R may be finitely represented by a regular expression (even though the actual expression is very long).
- However, P cannot actually be represented by a regular expression
- A regular expression is not powerful enough to represent languages which require parenthesis matching to arbitrary depths.
- All high level programming languages require an underlying language of expressions which require parentheses to be nested and matched to arbitrary depth.

CF-Grammars: Definition

A context-free grammar (CFG) $G = \langle N, T, P, S \rangle$ consists of

- a set N of nonterminal symbols,
- a set T of terminal symbols or the alphabet,
- a set P of productions or rewrite rules,
- each production is of the form $X \longrightarrow \alpha$, where
  - $X \in N$ is a nonterminal and
  - $\alpha \in (N \cup T)^*$ is a string of terminals and nonterminals
- a start symbol $S \in N$.

CFG: Example

$G = \langle \{S\}, \{a, b\}, P, S \rangle$, where $S \longrightarrow ab$ and $S \longrightarrow aSb$ are the only productions in P.

Derivations look like this:

$$S \Rightarrow ab$$

$$S \Rightarrow aSb \Rightarrow aabb$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$

$\mathcal{L}(G)$, the **language** generated by G is $\{a^n b^n | n > 0\}$.

Actually can be proved by induction on the length and structure of derivations.

CFG: Empty word

$G = \langle \{S\}, \{a, b\}, P, S \rangle$, where $S \longrightarrow SS \mid aSb \mid \varepsilon$ generates all sequences of matching nested parentheses, including the empty word ε.

A leftmost derivation might look like this:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \ldots$$

A rightmost derivation might look like this:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow SaSb \Rightarrow Sab \Rightarrow aSbab \ldots$$

Other derivations might look like *God alone knows what!*

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow \ldots$$

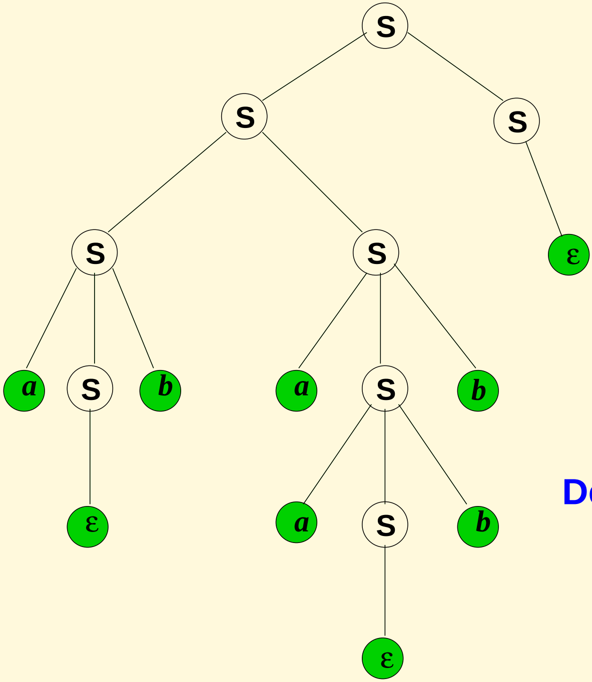
Could be quite confusing!

CFG: Derivation trees 1

Derivation sequences

- put an artificial order in which productions are fired.
- instead look at **trees** of derivations in which we may think of productions as being fired in **parallel**.
- There is then no highlighting in red to determine which copy of a nonterminal was used to get the next member of the sequence.
- Of course, generation of the empty word ε must be shown explicitly in the tree.

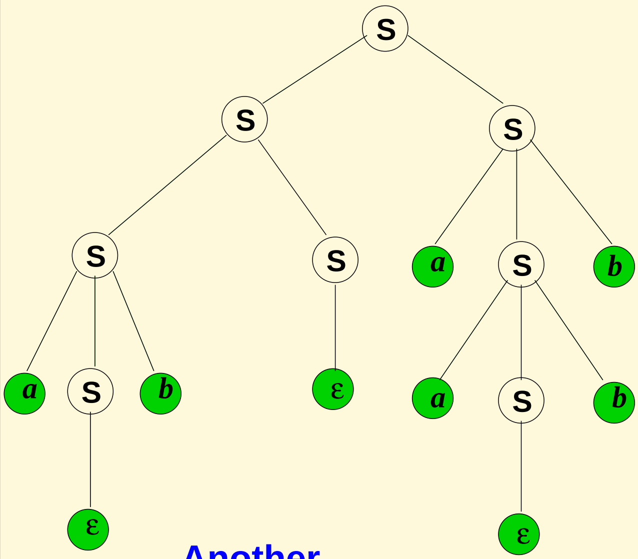
CFG: Derivation trees 2



Derivation tree of

abaabb

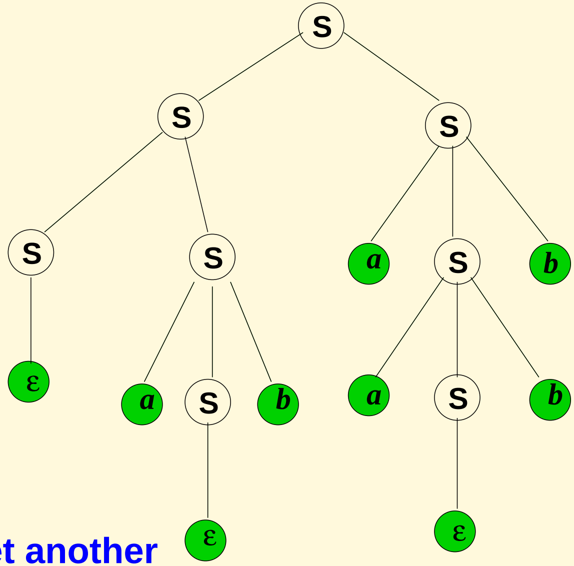
CFG: Derivation trees 3



Another Derivation tree of

abaabb

CFG: Derivation trees 4

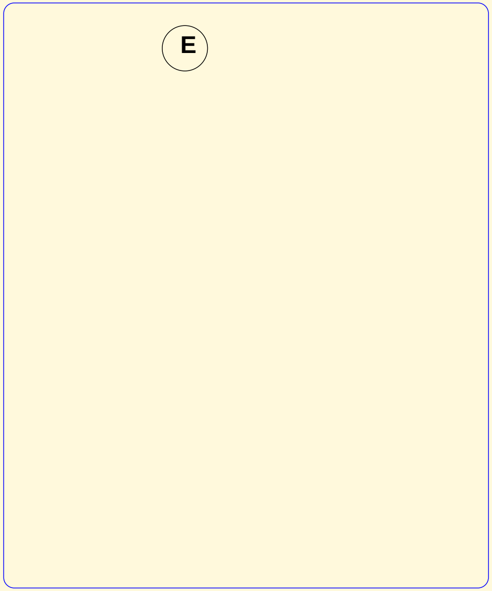


Yet another Derivation tree of *abaabb*

Ambiguity: 1

$$
\begin{array}{lcl}
E & \rightarrow & I \mid C \mid E\mathbf{+}E \mid E\mathbf{*}E \\
I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\
C & \rightarrow & \mathbf{4}
\end{array}
$$

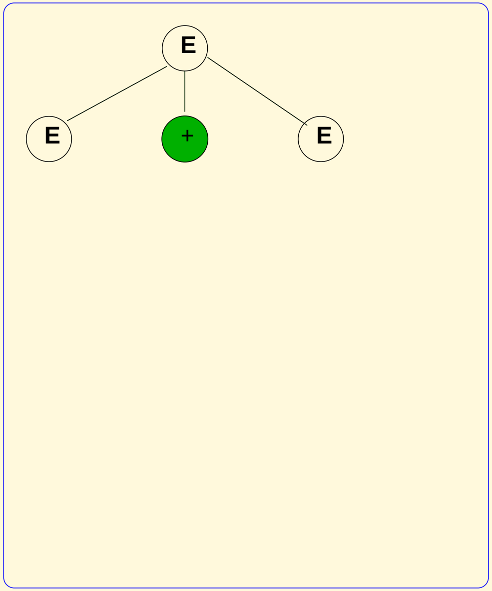
Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.

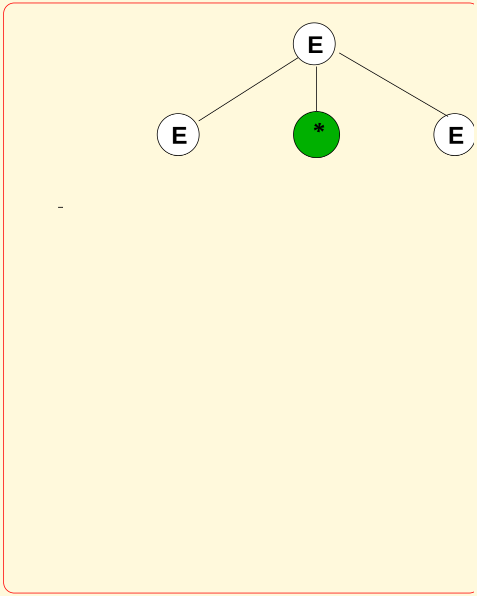


Ambiguity: 2

$$
\begin{array}{lcl}
E & \longrightarrow & I \mid C \mid E{+}E \mid E{*}E \\
I & \longrightarrow & \mathbf{y} \mid \mathbf{z} \\
C & \longrightarrow & \mathbf{4}
\end{array}
$$

Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.



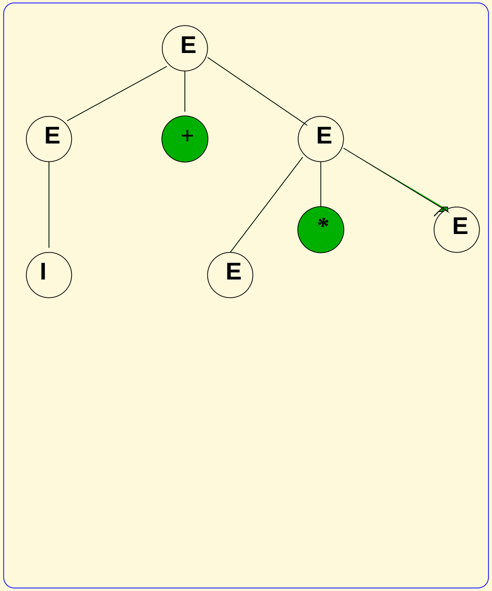


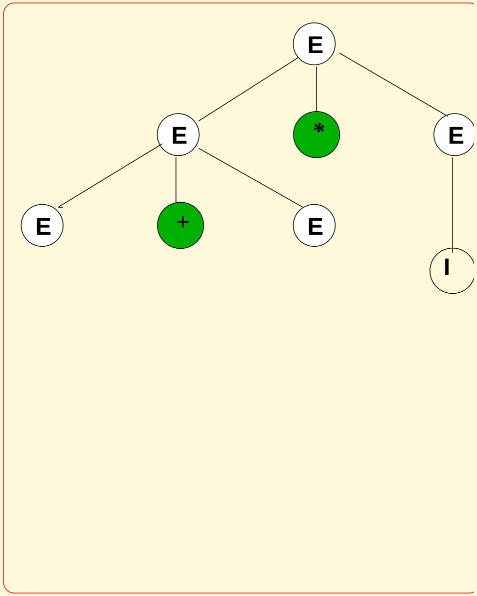
Home Page
Title Page
Page 29 of 100
Go Back
Full Screen
Close
Quit

Ambiguity: 3

$$
\begin{array}{lcl}
E & \rightarrow & I \mid C \mid E{+}E \mid E{*}E \\
I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\
C & \rightarrow & \mathbf{4}
\end{array}
$$

Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.

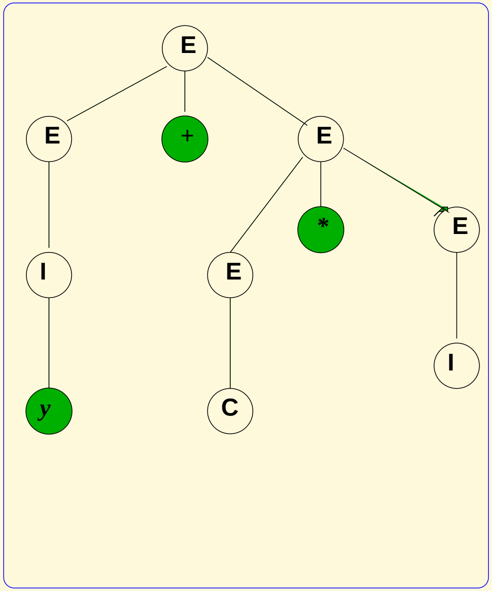


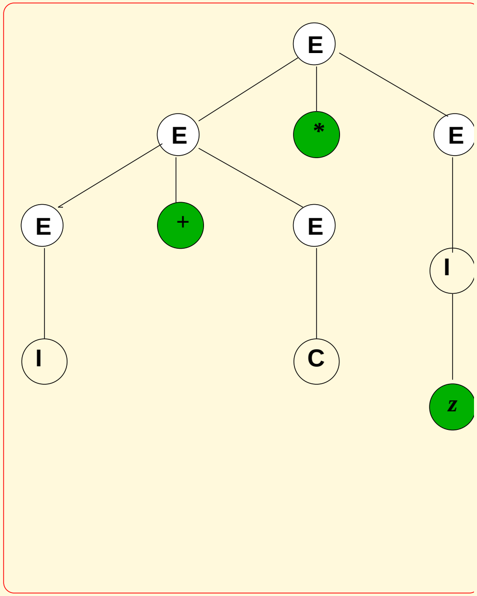


Ambiguity: 4

$$\begin{array}{rcl} E & \rightarrow & I \mid C \mid E{+}E \mid E{*}E \\ I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\ C & \rightarrow & \mathbf{4} \end{array}$$

Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.

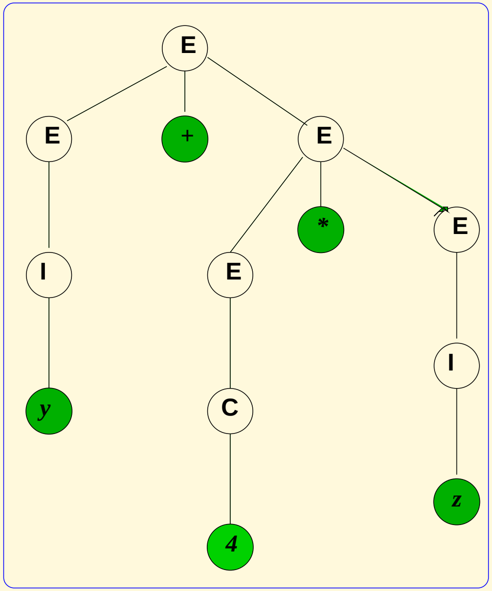


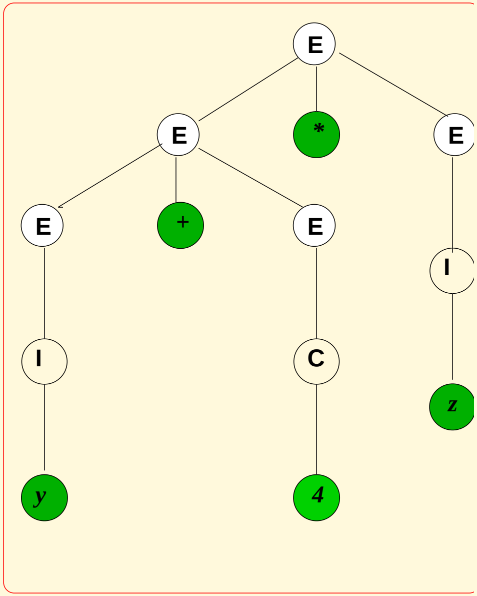


Ambiguity: 5

$$
\begin{array}{lcl}
E & \rightarrow & I \mid C \mid E+E \mid E*E \\
I & \rightarrow & \mathbf{y} \mid \mathbf{z} \\
C & \rightarrow & \mathbf{4}
\end{array}
$$

Consider the sentence $\mathbf{y} + \mathbf{4} * \mathbf{z}$.





Parsing: 0

r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

a − a / b

Parsing: 1

r1. E ⟶ E - T

r2 E ⟶ T

r3 T ⟶ T / D

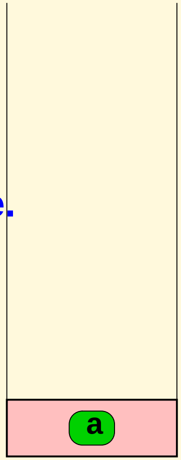
r4 T ⟶ D

r5 D ⟶ a | b | (E)

- a / b

Principle:

Reduce whenever possible. Shift only when reduce is impossible



Shift

Parsing: 2

r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

− a / b

D

Reduce by r5

Home Page
Title Page
Page 36 of 100
Go Back
Full Screen
Close
Quit

Parsing: 3

r1. E → E - T

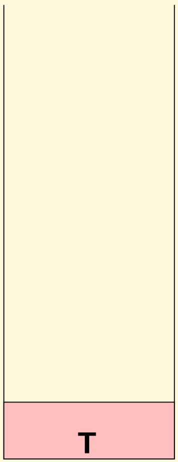
r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)

- a / b



Reduce by r4

Parsing: 4

r1. $E \longrightarrow E \; - \; T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \; / \; D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$

$-\;a\;/\;b$

E

Reduce by r2

Parsing: 5

r1. E $\longrightarrow$ E - T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

a / b

-

E

Shift

Parsing: 6

r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

/ b

a

−

E

Shift

Parsing: 7

r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \,/\, D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$

/ b

D
−
E

Reduce by r5

Parsing: 8

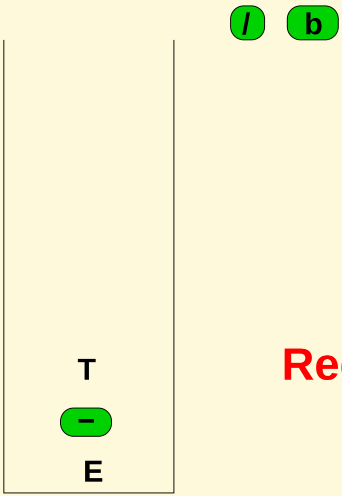
r1. E → E − T

r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)



Reduce by r4

Parsing: 8a

r1. E $\longrightarrow$ E **-** T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T **/** D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ **a** | **b** | **(** E **)**

/ **b**

T
-
E

Reduce by r4

Parsing: 9a

r1. E → E - T

r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)

/ b

E

Reduce by r1

Parsing: 10a

r1. E → E - T

r2 E → T

r3 T → T / D

r4 T → D

r5 D → a | b | (E)

b

/

E

Shift

Parsing: 11a

r1. $E \longrightarrow E \; - \; T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \; / \; D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\; E \;)$

b
/
E

Shift

Parsing: 12a

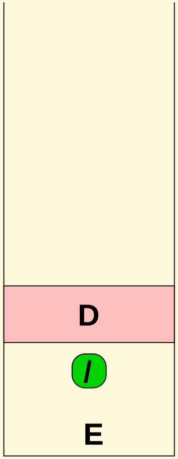
r1. E $\longrightarrow$ E – T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Reduce by r5

Parsing: 13a

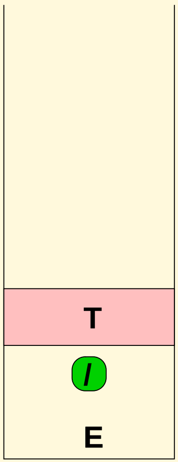
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Reduce by r4

Home Page
Title Page
Page 48 of 100
Go Back
Full Screen
Close
Quit

Parsing: 14a

r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

Stuck!

E

/

E

Reduce by r2

Get back!

Parsing: 14b

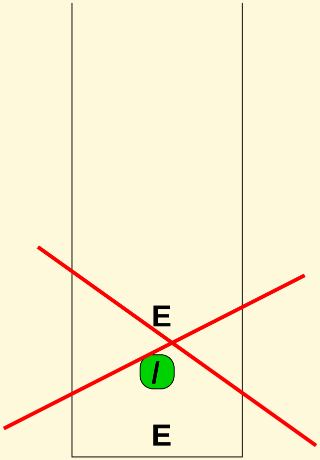
r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \,/\, D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$



Reduce by r2

Get back!

Parsing: 13b

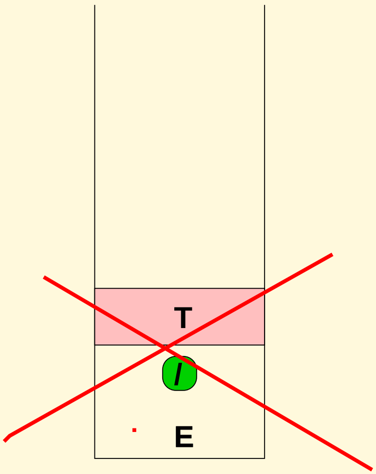
$$r1.\quad E \longrightarrow E - T$$

$$r2\quad E \longrightarrow T$$

$$r3\quad T \longrightarrow T \, / \, D$$

$$r4\quad T \longrightarrow D$$

$$r5\quad D \longrightarrow a \mid b \mid (E)$$



Reduce by r4

Get back!

Parsing: 12b

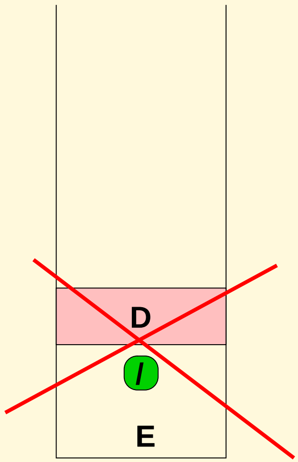
r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \, / \, D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$



Reduce by r5

Get back!

Parsing: 11b

r1. $E \longrightarrow E \; - \; T$

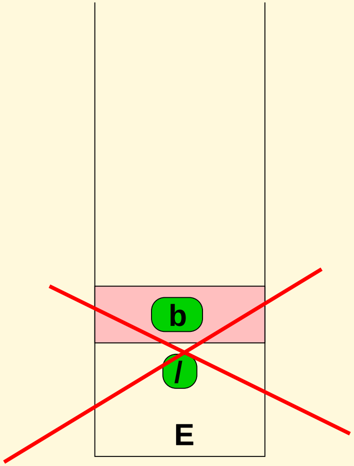
r2 $E \longrightarrow T$

r3 $T \longrightarrow T \; / \; D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\; E \;)$

Get back!



Shift

Parsing: 10b

r1. E → E - T

r2 E → T

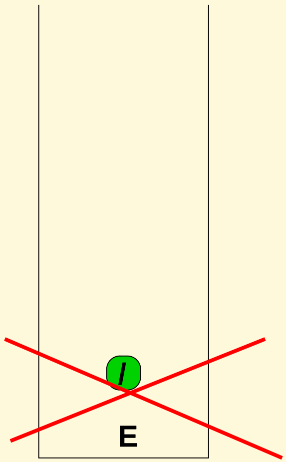
r3 T → T / D

r4 T → D

r5 D → a | b | (E)

b

Get back!



Shift

Parsing: 9b

r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \; / \; D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\; E \;)$

/ b

~~E~~

Get back to where you once belonged!

Reduce by r1

Parsing: 8b

r1. E ⟶ E − T

r2 E ⟶ T

r3 T ⟶ T / D

r4 T ⟶ D

r5 D ⟶ a | b | (E)

modified

Principle:

Reduce whenever possible, but

but depending upon

lookahead

/ b

Shift instead of reduce here!

Shift−reduce conflict

T

−

E

Reduce by r4

Parsing: 8

r1. $E \longrightarrow E$ **–** T

r2 $E \longrightarrow T$

r3 $T \longrightarrow T$ **/** D

r4 $T \longrightarrow D$

r5 $D \longrightarrow$ **a** | **b** | **(** E **)**

/ **b**

T
–
E

Reduce by r4

Parsing: 9

r1. $E \longrightarrow E\ -\ T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T\ /\ D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$

b

/

T

−

E

Shift

Parsing: 10

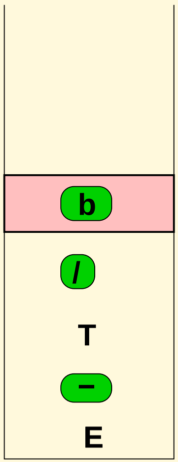
r1. $E \longrightarrow E \; - \; T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \; / \; D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\; E \;)$



Shift

Parsing: 11

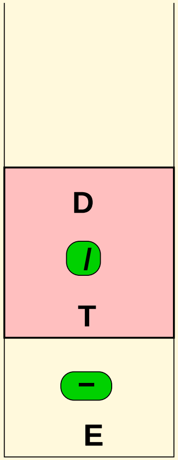
r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \; / \; D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$



Reduce by r5

Parsing: 12

r1. $E \longrightarrow E\ -\ T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T\ /\ D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\ E\ )$

T

−

E

Reduce by r3

Parsing: 13

r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

E

Reduce by r1

Parse Trees: 0

r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)

a − a / b

Parse Trees: 1

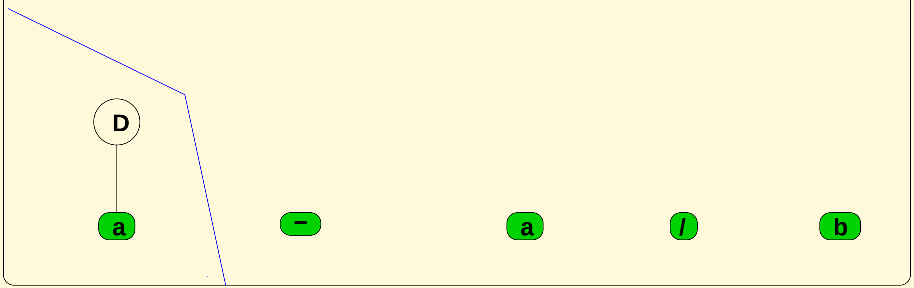
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Parse Trees: 2

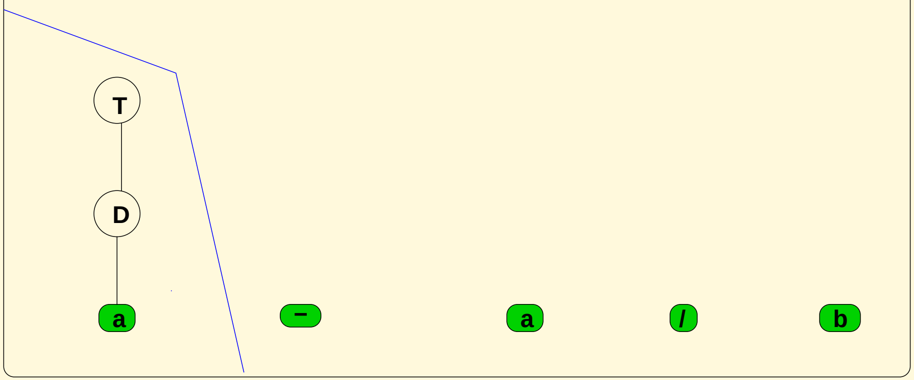
r1. $E \longrightarrow E - T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \,/\, D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (E)$



Parse Trees: 3

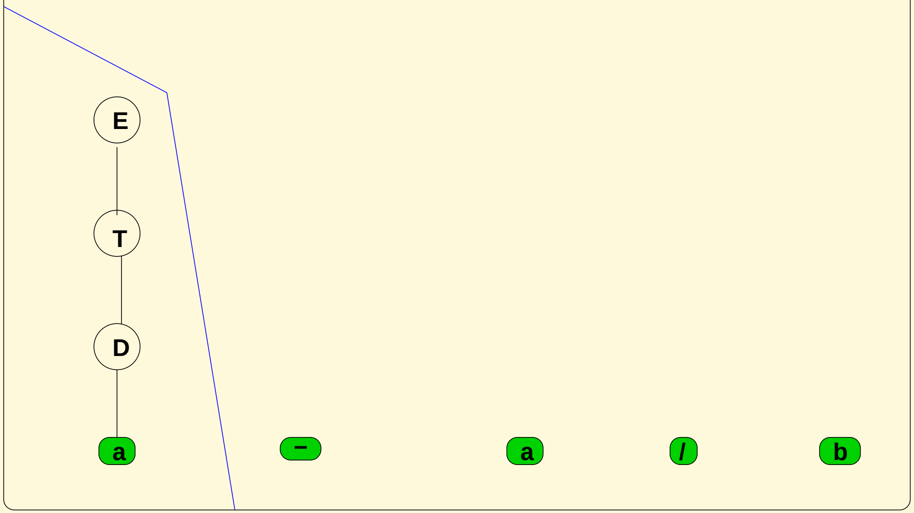
r1. $E \longrightarrow E$ **-** T

r2 $E \longrightarrow T$

r3 $T \longrightarrow T$ **/** D

r4 $T \longrightarrow D$

r5 $D \longrightarrow$ **a** | **b** | **(** E **)**



Home Page
Title Page
Page 66 of 100
Go Back
Full Screen
Close
Quit

Parse Trees: 3a

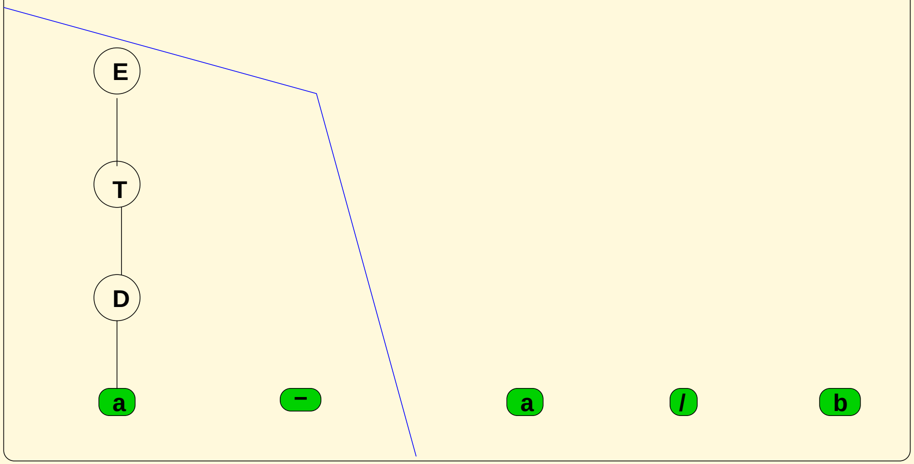
r1. E → E **−** T

r2 E → T

r3 T → T **/** D

r4 T → D

r5 D → **a** | **b** | **(** E **)**



Parse Trees: 3b

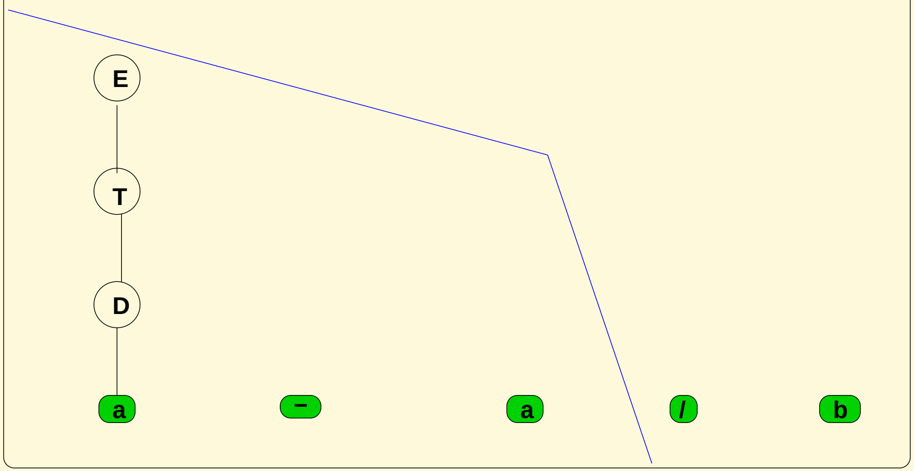
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Parse Trees: 4

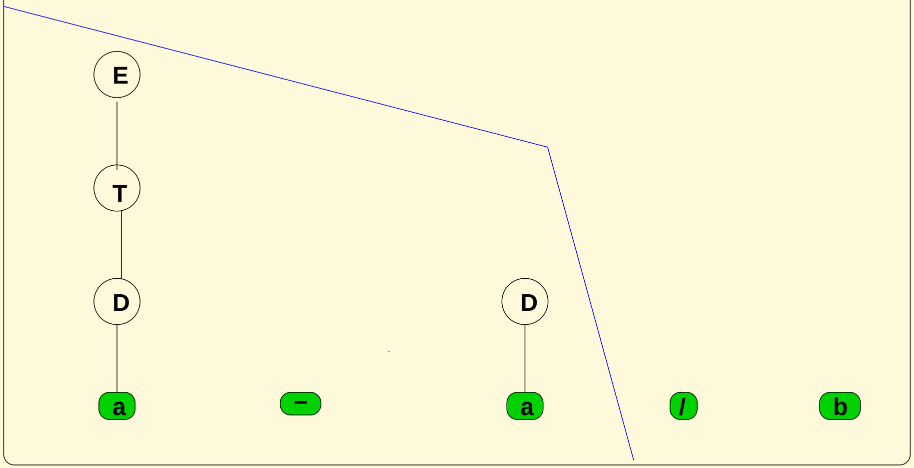
r1. $E \longrightarrow E \; - \; T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T \; / \; D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\; E \;)$



Parse Trees: 5

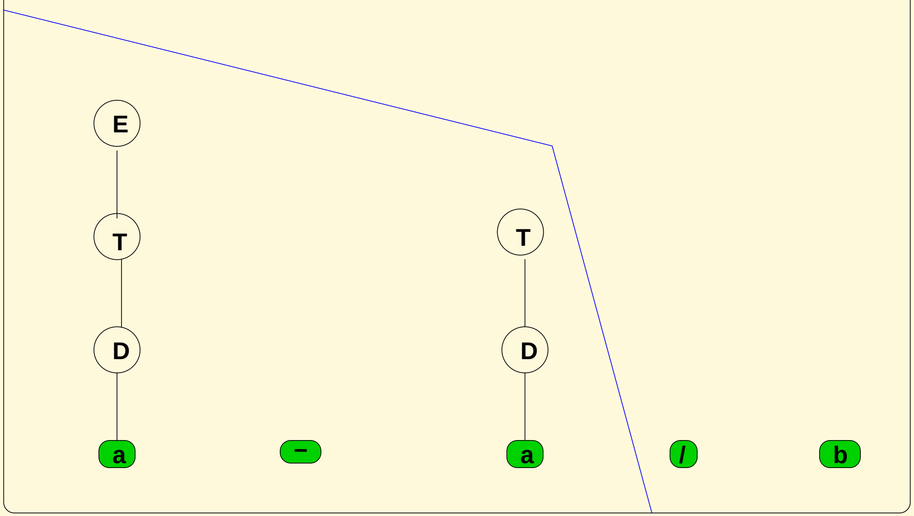
r1. $E \longrightarrow E\ -\ T$

r2 $E \longrightarrow T$

r3 $T \longrightarrow T\ /\ D$

r4 $T \longrightarrow D$

r5 $D \longrightarrow a \mid b \mid (\ E\ )$



Parse Trees: 5a

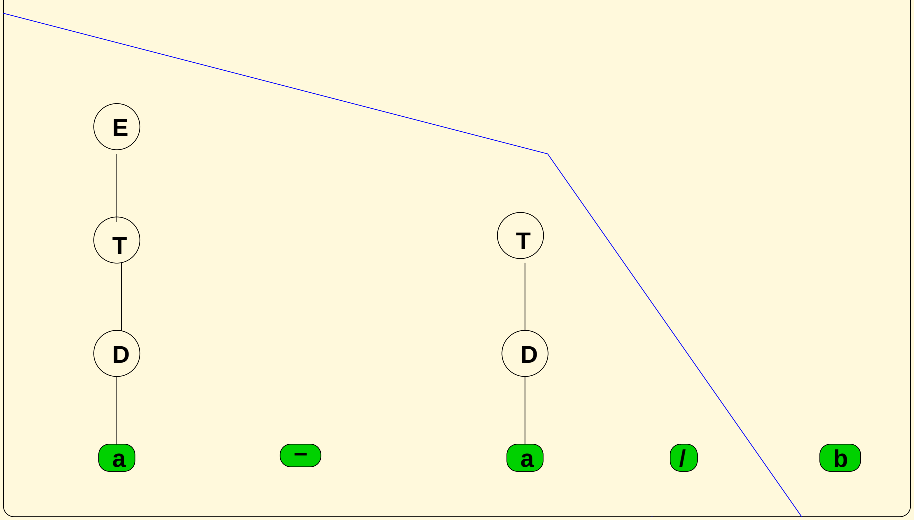
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Parse Trees: 5b

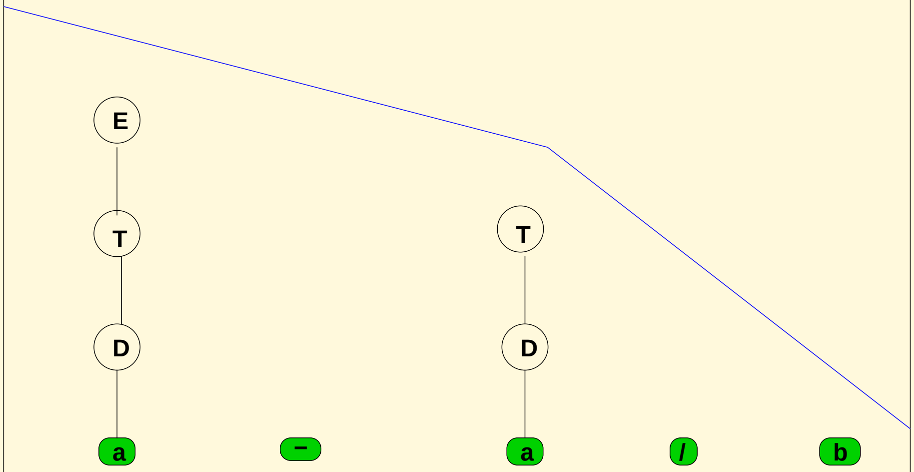
r1. E $\longrightarrow$ E − T

r2 E $\longrightarrow$ T

r3 T $\longrightarrow$ T / D

r4 T $\longrightarrow$ D

r5 D $\longrightarrow$ a | b | (E)



Parse Trees: 6

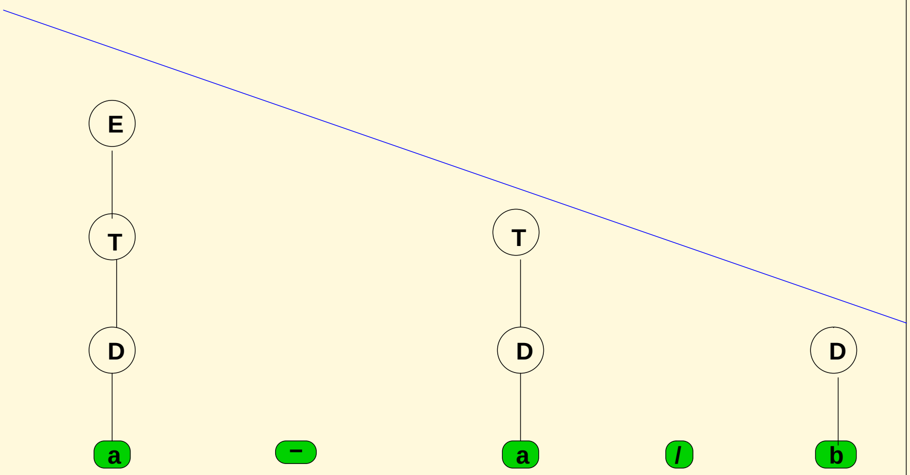
r1. $E \longrightarrow E$ − T

r2 $E \longrightarrow T$

r3 $T \longrightarrow T$ / D

r4 $T \longrightarrow D$

r5 $D \longrightarrow$ a | b | (E)



Parse Trees: 7

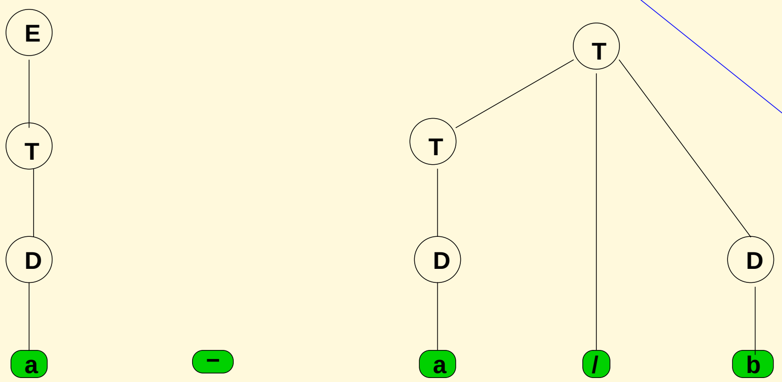
r1. $E \longrightarrow E$ − T

r2 $E \longrightarrow T$

r3 $T \longrightarrow T$ / D

r4 $T \longrightarrow D$

r5 $D \longrightarrow$ a | b | (E)



Parse Trees: 8

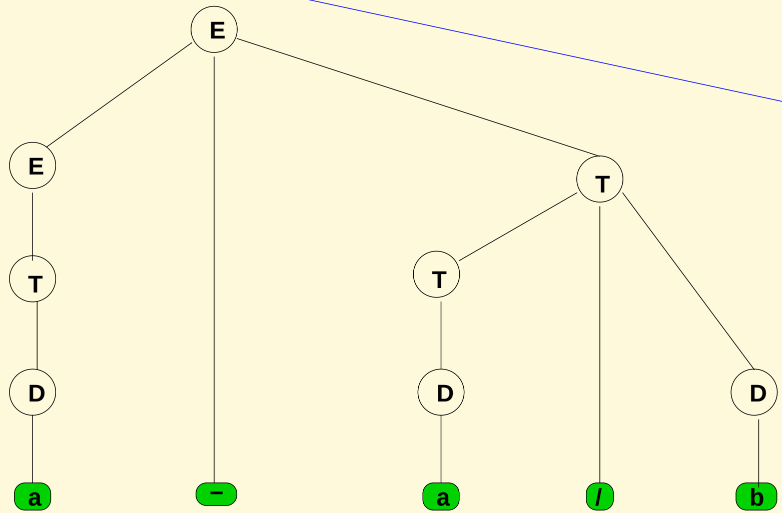
r1. $E \longrightarrow E$ − T

r2 $E \longrightarrow T$

r3 $T \longrightarrow T$ / D

r4 $T \longrightarrow D$

r5 $D \longrightarrow$ a | b | (E)



Parsing: Summary: 1

- All high-level languages are designed so that they may be parsed in this fashion with only a single token look-ahead.
- Parsers for a language can be automatically constructed by parger-generators such as Yacc, Bison, ML-Yacc.
- Shift-reduce conflicts if any, are automatically detected and reported by the parser-generator.
- Shift-reduce conflicts may be avoided by suitably redesigning the context-free grammar.

Parsing: Summary: 2

- Very often shift-reduce conflicts may occur because of the prefix problem. In such cases many parser-generators resolve the conflict in favour of shifting.
- There is also a possiblility of reduce-reduce conflicts. This usually happens when there is more than one non-terminal symbol to which the contents of the stack may reduce.
- A minor reworking of the grammar to avoid redundant non-terminal symbols will get rid of reduce-reduce conflicts.

The Big Picture

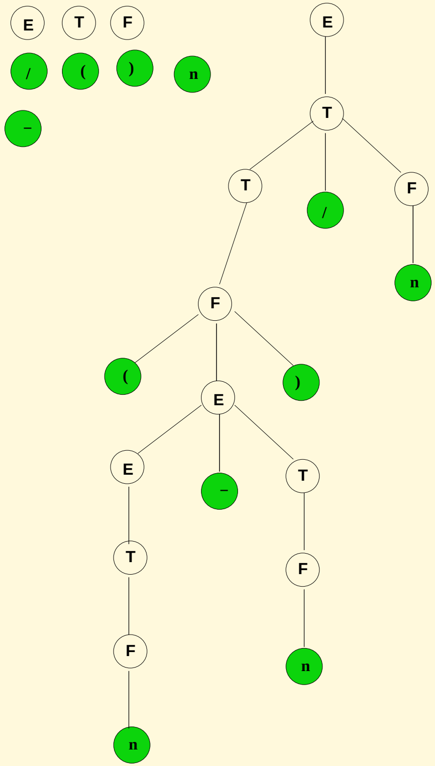
Semantic Analysis: 1

- Every Programming langauge can be used to program any computable function, assuming of course, it has
  - unbounded memory, and
  - unbounded time
- The parser of a programming language provides the *framework* within which the target code is to be generated.
- The parser also provides a *structuring* mechanism that divides the task of code generation into bits and pieces determined by the individual nonterminals and production rules.
- However, contex-free grammars are not powerful enough to represent all computable functions. Example, the language $\{a^nb^nc^n | n > 0\}$.

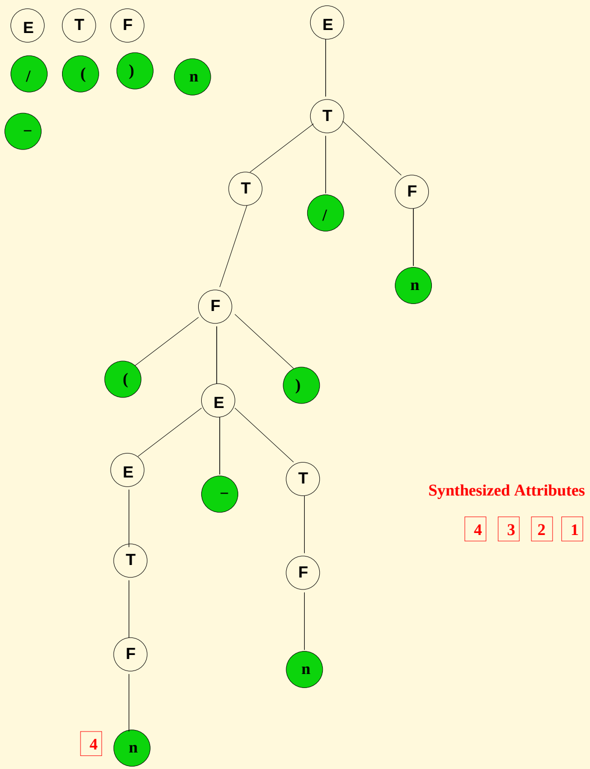
Semantic Analysis: 2

- There are context-sensitive aspects of a program that cannot be represented/enforced by a context-free grammar definition. Examples include
  - correspondence between formal and actual parameters
  - type consistency between declaration and use.
  - scope and visibility issues with respect to identifiers in a program.

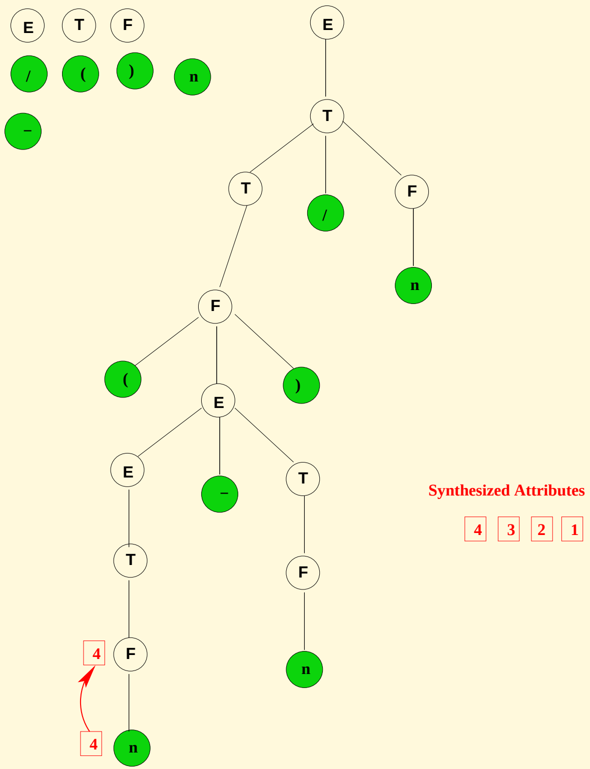
Synthesized Attributes: 0



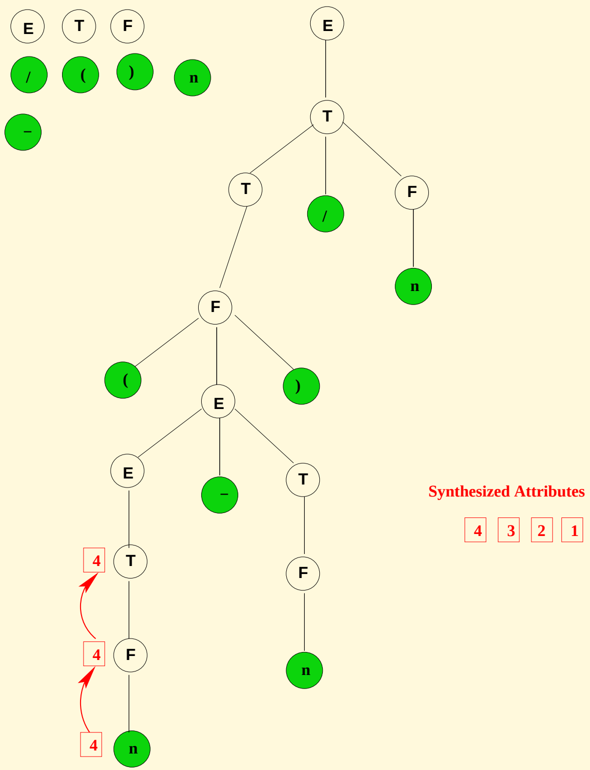
Synthesized Attributes: 1



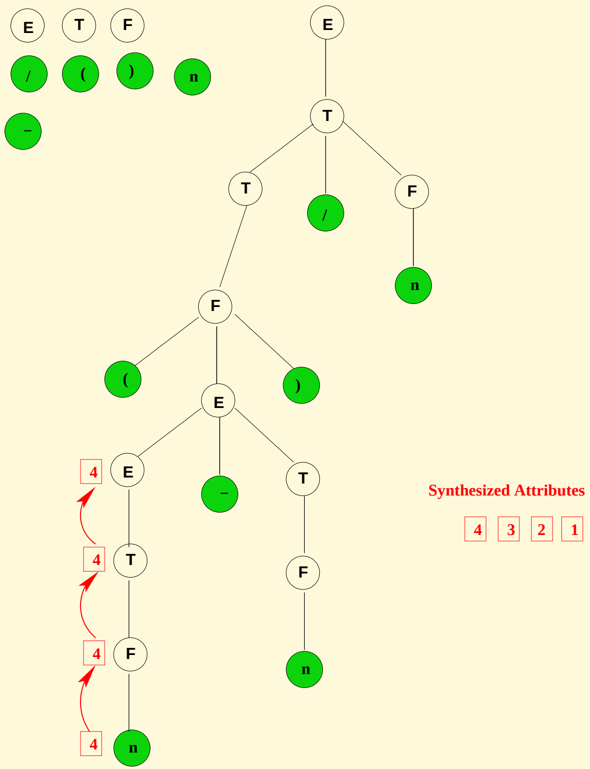
Synthesized Attributes: 2



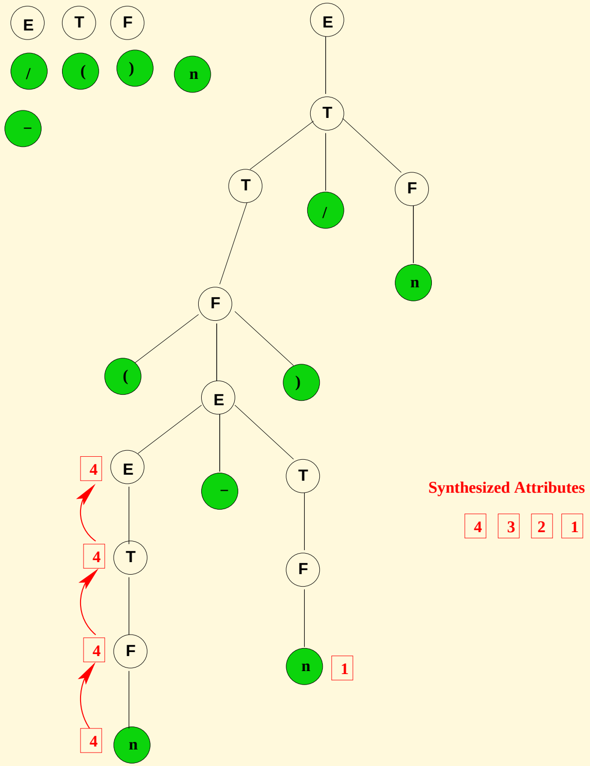
Synthesized Attributes: 3



Synthesized Attributes: 4

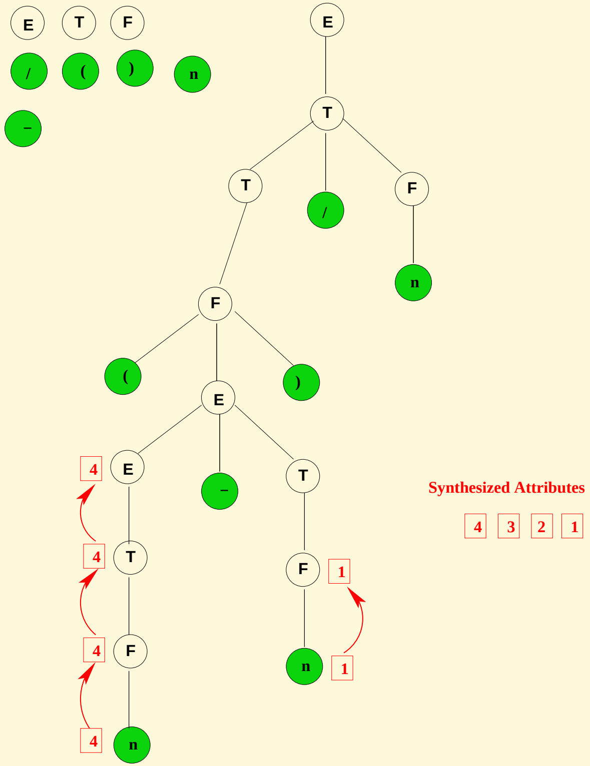


Synthesized Attributes: 5

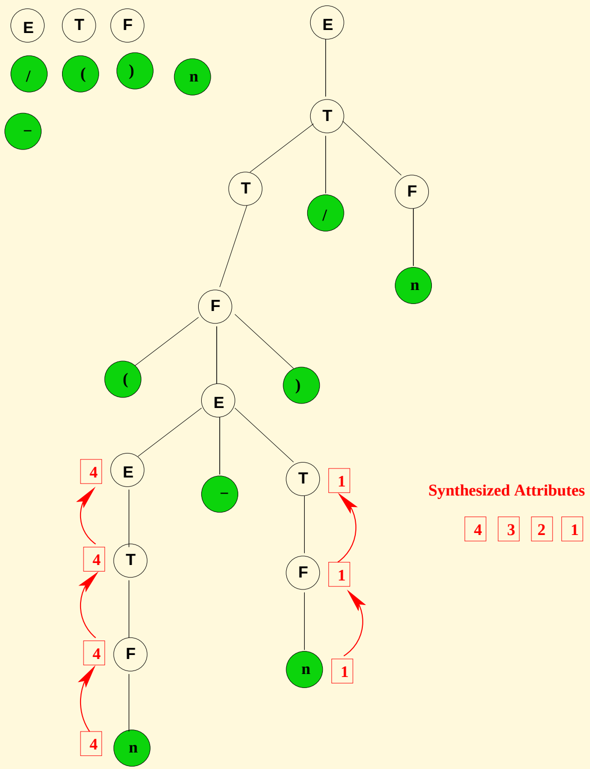


Home Page
Title Page
Page 86 of 100
Go Back
Full Screen
Close
Quit

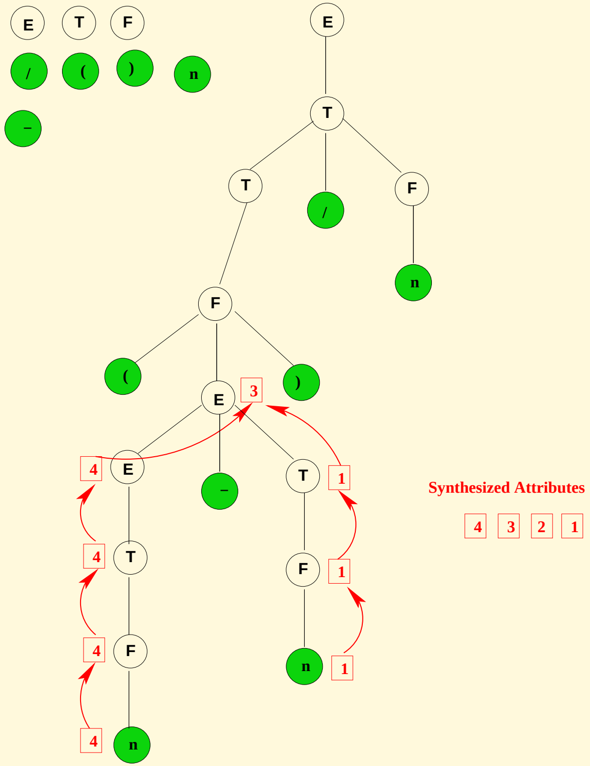
Synthesized Attributes: 6



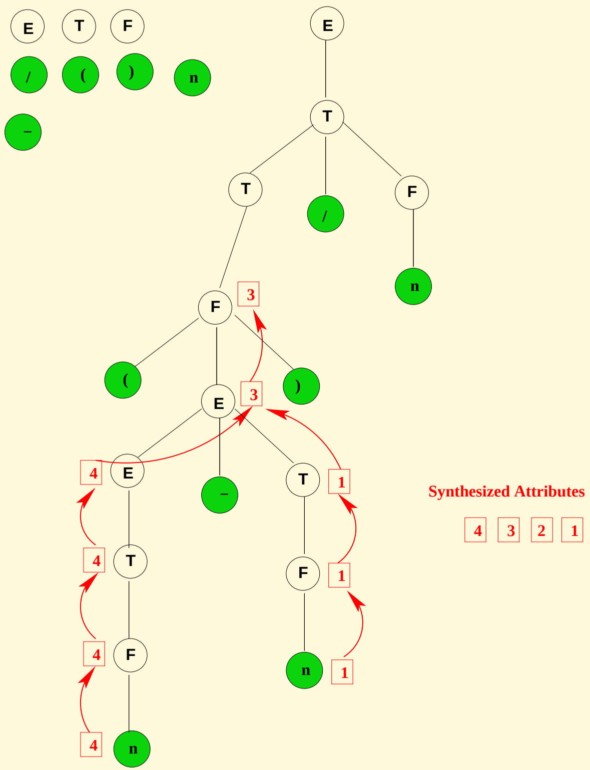
Synthesized Attributes: 7



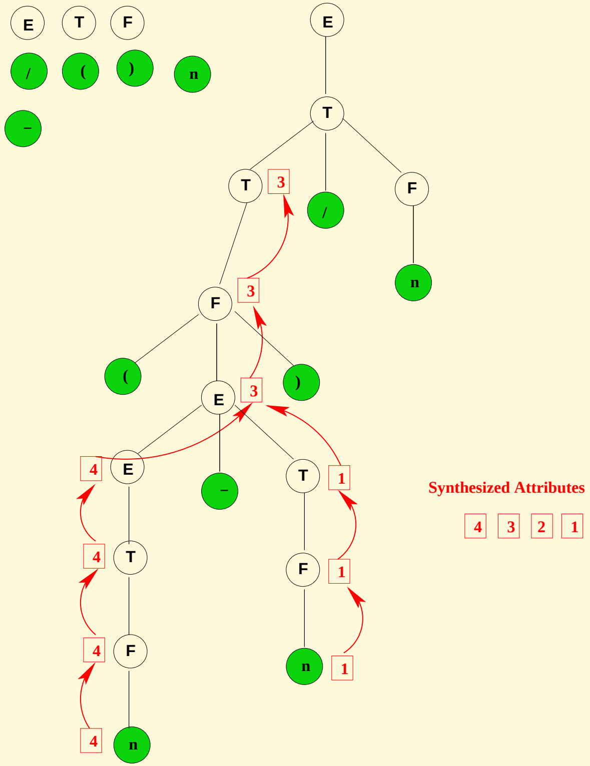
Synthesized Attributes: 8



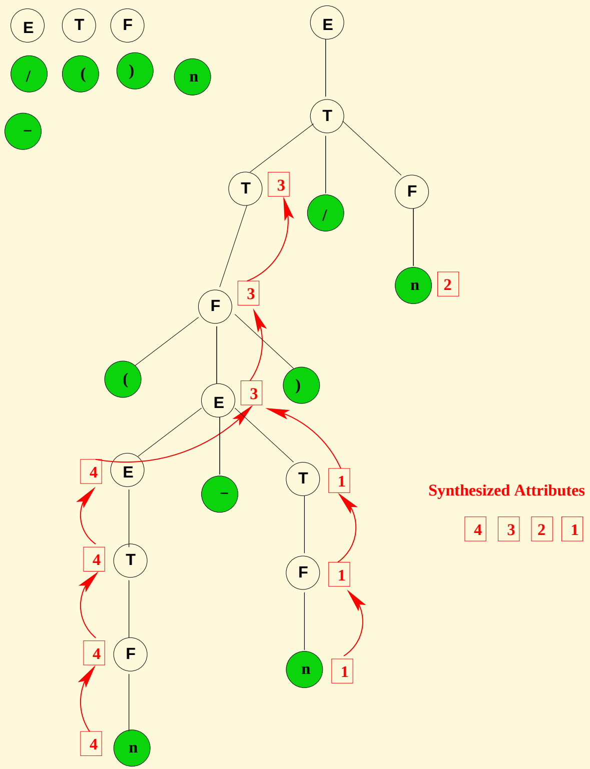
Synthesized Attributes: 9



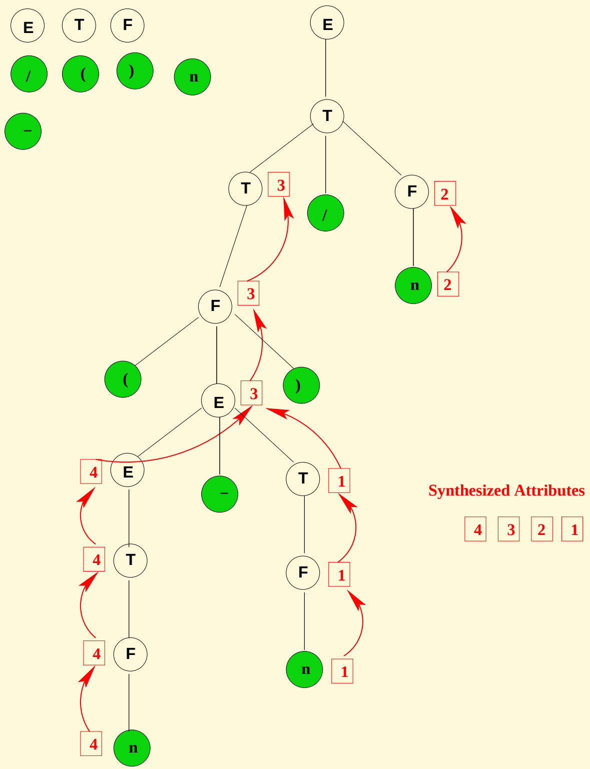
Synthesized Attributes: 10



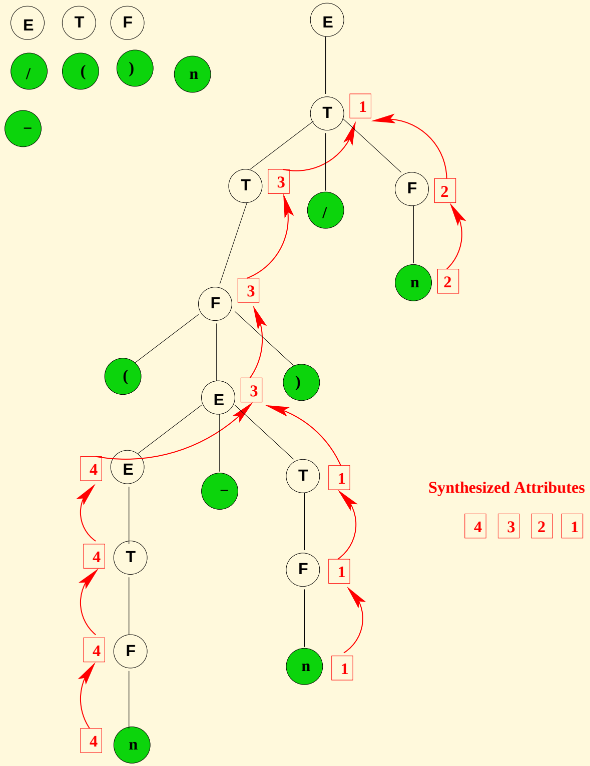
Synthesized Attributes: 11



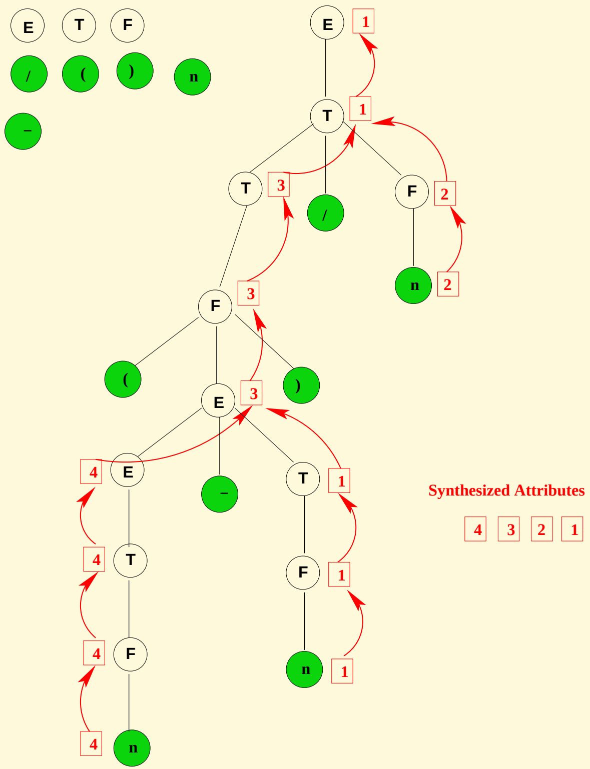
Synthesized Attributes: 12



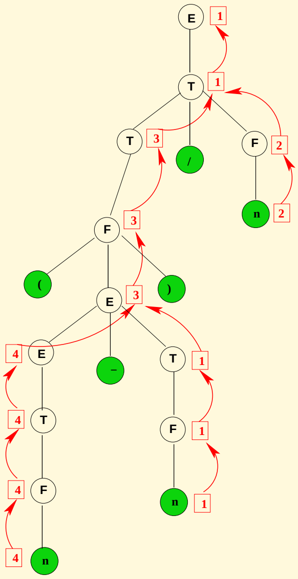
Synthesized Attributes: 13



Synthesized Attributes: 14



An Attribute Grammar

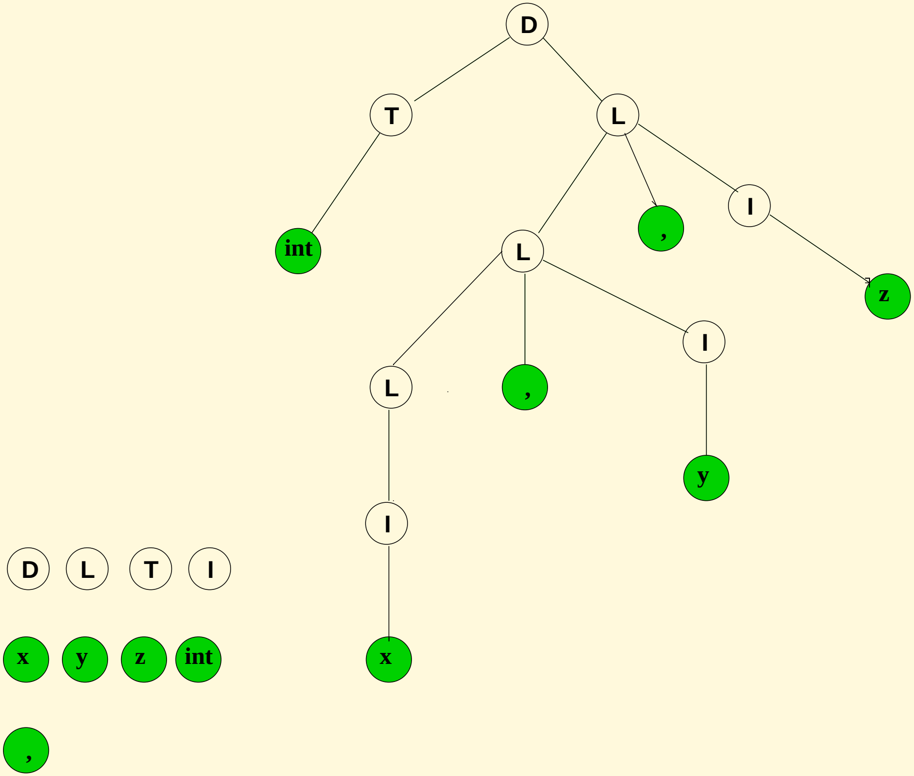


$$
\begin{array}{llll}
E_0 & \rightarrow & E_1 - T & \triangleright\ E_0.val := \underline{sub}(E_1.val, T.val) \\
E & \rightarrow & T & \triangleright\ E.val := T.val \\
T_0 & \rightarrow & T_1 / F & \triangleright\ T_0.val := \underline{div}(T_1.val, F.val) \\
T & \rightarrow & F & \triangleright\ T.val := F.val \\
F & \rightarrow & (E) & \triangleright\ F.val := E.val \\
F & \rightarrow & \mathbf{n} & \triangleright\ F.val := \mathbf{n}.val
\end{array}
$$

Inherited Attributes: 0

C-style declarations generating **int x, y, z**.

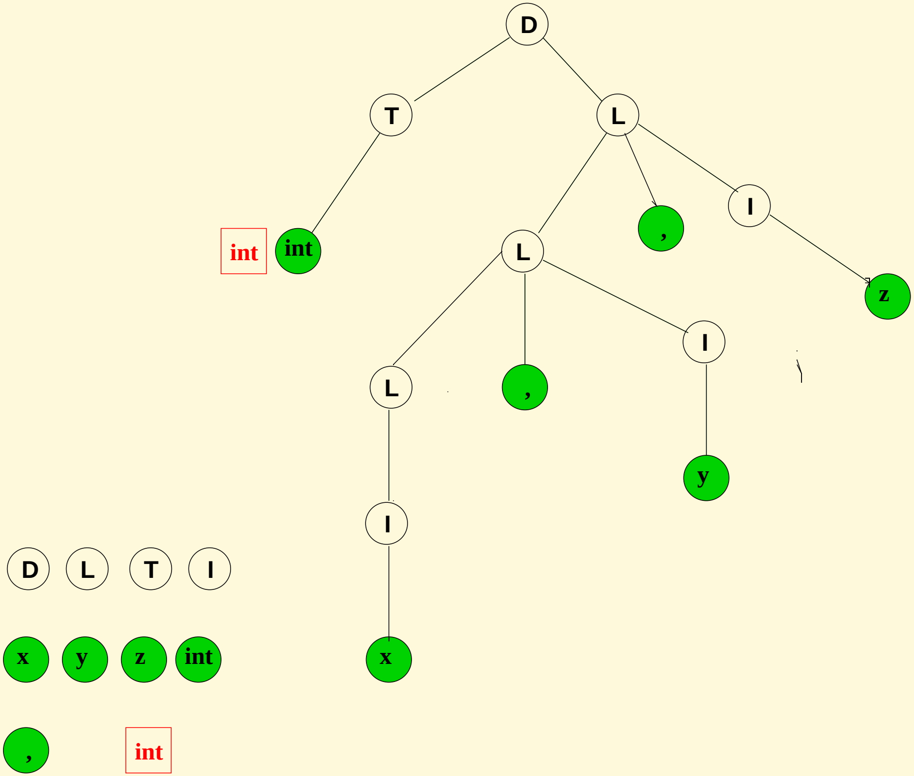
$$
\begin{array}{lcl}
D & \rightarrow & T\ L \\
L & \rightarrow & L\,\mathbf{,}\,I \mid I
\end{array}
\qquad
\begin{array}{lcl}
T & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
I & \rightarrow & \mathbf{x} \mid \mathbf{y} \mid \mathbf{z}
\end{array}
$$



Inherited Attributes: 1

C-style declarations generating $\mathbf{int}\ \mathbf{x}, \mathbf{y}, \mathbf{z}$.

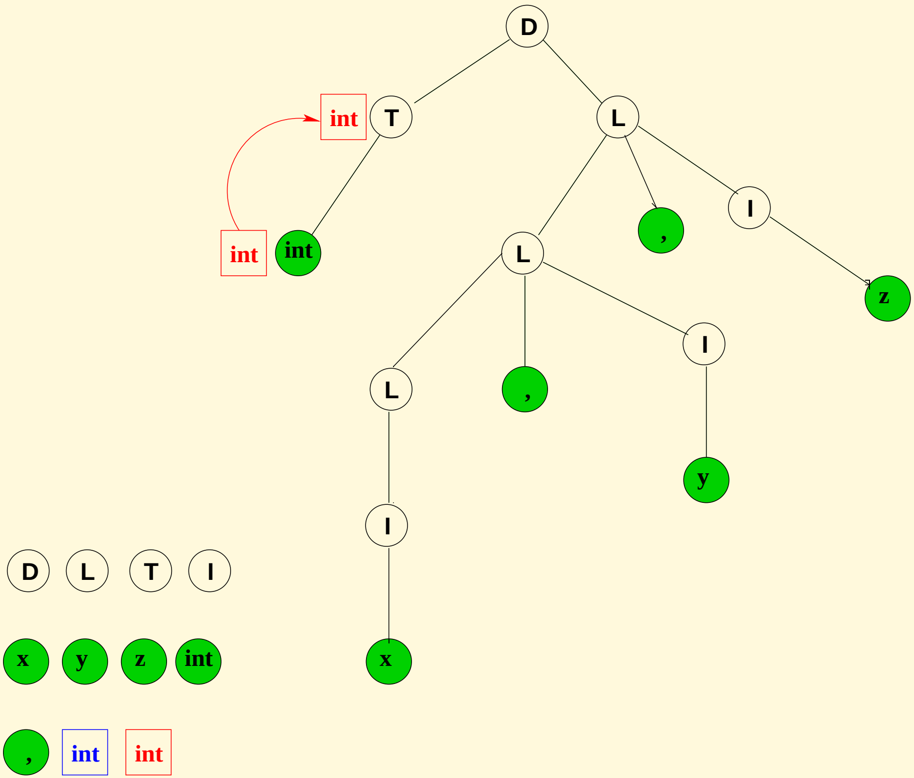
$$
\begin{array}{lll}
D & \rightarrow & T\ L \\
L & \rightarrow & L\mathbf{,}I \mid I
\end{array}
\qquad
\begin{array}{lll}
T & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
I & \rightarrow & \mathbf{x} \mid \mathbf{y} \mid \mathbf{z}
\end{array}
$$



Inherited Attributes: 2

C-style declarations generating $\mathbf{int\ x, y, z}$.

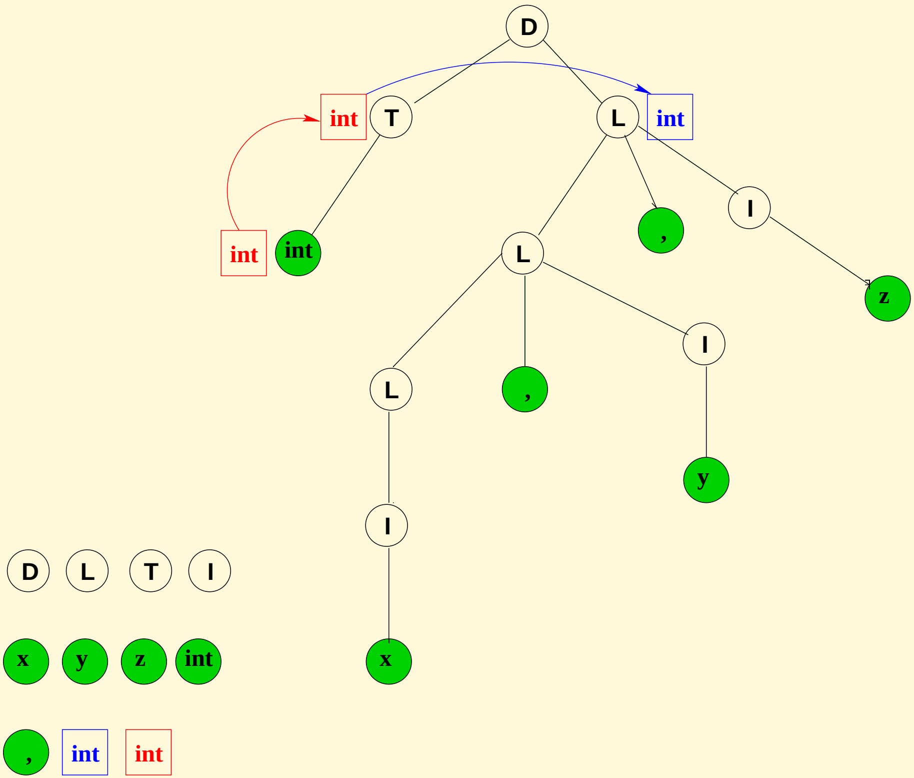
$$\begin{array}{lll} D & \rightarrow & T\ L \\ L & \rightarrow & L\mathbf{,}I \mid I \end{array} \qquad \begin{array}{lll} T & \rightarrow & \mathbf{int} \mid \mathbf{float} \\ I & \rightarrow & \mathbf{x} \mid \mathbf{y} \mid \mathbf{z} \end{array}$$



Inherited Attributes: 3

C-style declarations generating $\mathbf{int}\ \mathbf{x}, \mathbf{y}, \mathbf{z}$.

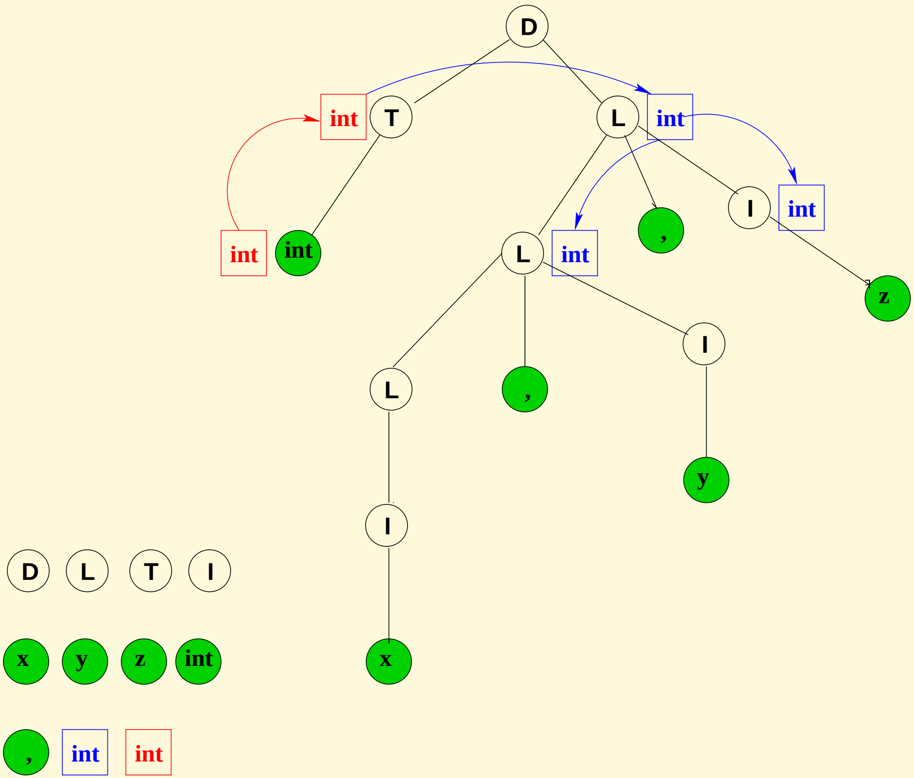
$$\begin{array}{lcl} D & \rightarrow & T\ L \\ L & \rightarrow & L\mathbf{,}I \mid I \end{array} \qquad \begin{array}{lcl} T & \rightarrow & \mathbf{int} \mid \mathbf{float} \\ I & \rightarrow & \mathbf{x} \mid \mathbf{y} \mid \mathbf{z} \end{array}$$



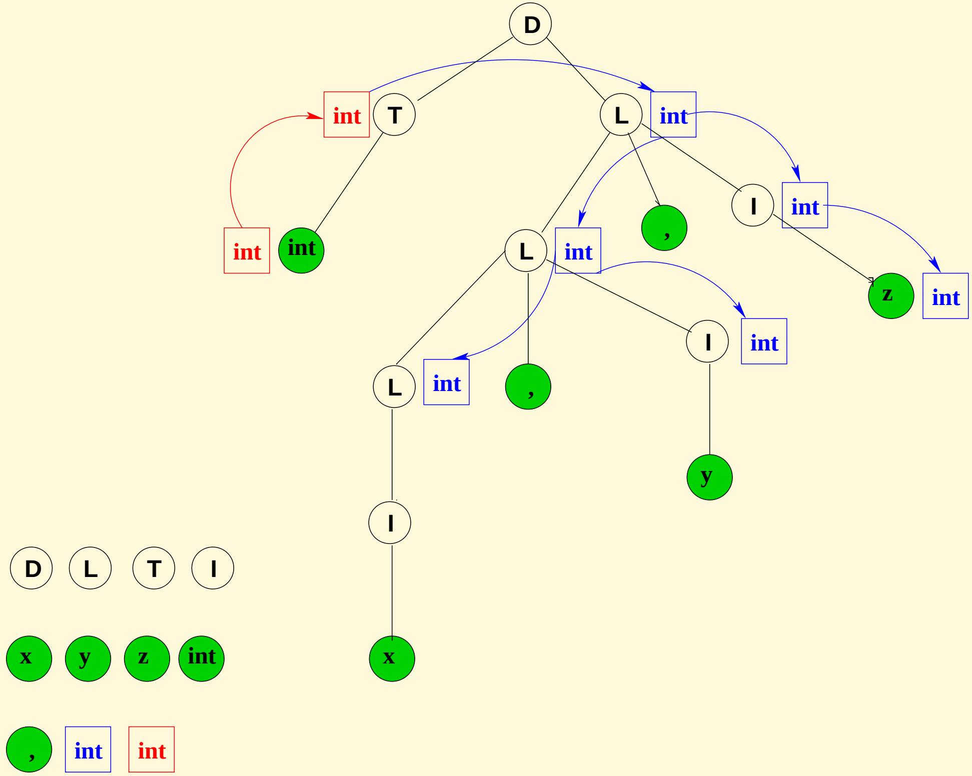
Inherited Attributes: 4

C-style declarations generating **int x, y, z**.

$$
\begin{array}{lcl}
D & \rightarrow & T\ L \\
L & \rightarrow & L\mathbf{,}I \mid I
\end{array}
\qquad\qquad
\begin{array}{lcl}
T & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
I & \rightarrow & \mathbf{x} \mid \mathbf{y} \mid \mathbf{z}
\end{array}
$$



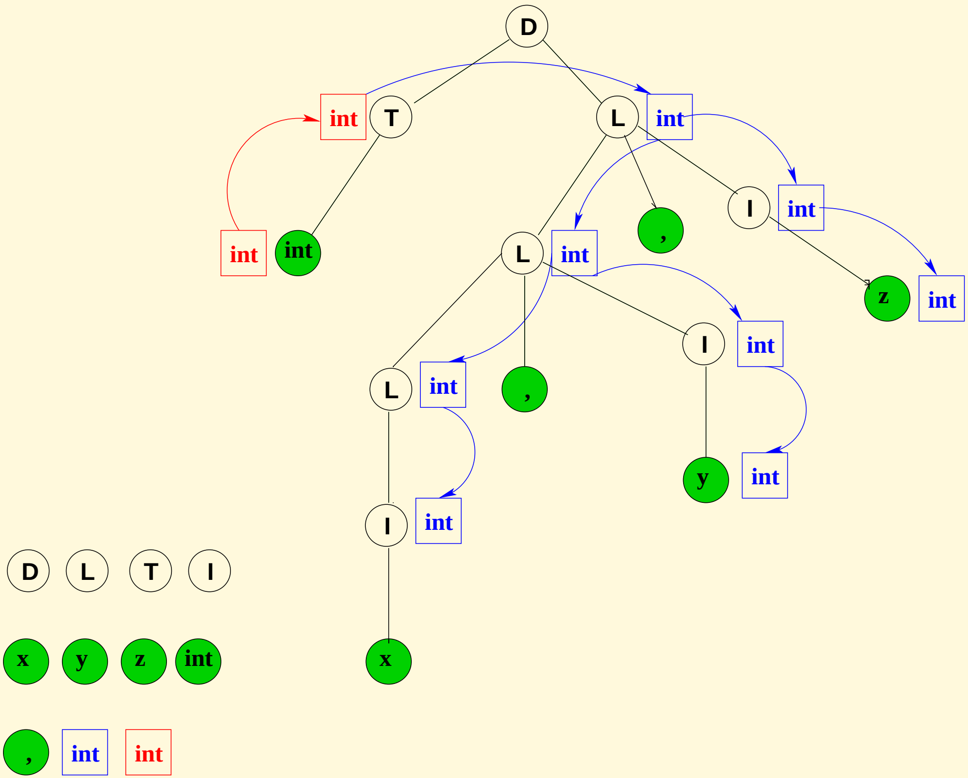
Inherited Attributes: 5



Inherited Attributes: 6

C-style declarations generating $\mathbf{int}\ \mathbf{x}, \mathbf{y}, \mathbf{z}$.

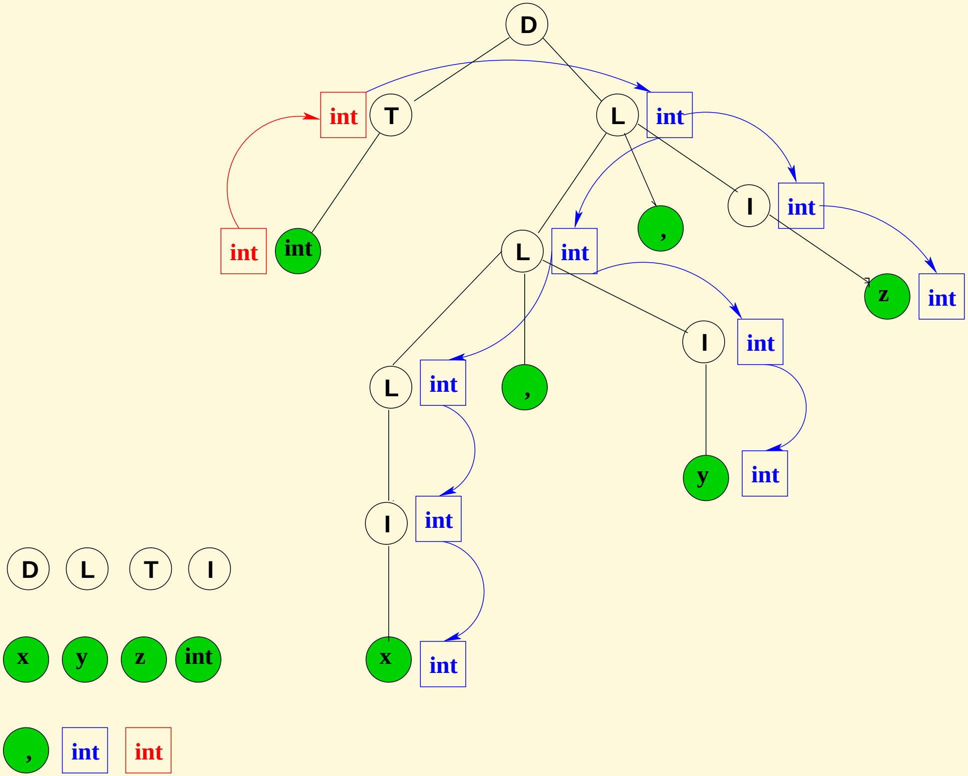
$$
\begin{array}{lcl}
D & \rightarrow & T\ L \\
L & \rightarrow & L \mathbf{,} I \mid I
\end{array}
\qquad\qquad
\begin{array}{lcl}
T & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
I & \rightarrow & \mathbf{x} \mid \mathbf{y} \mid \mathbf{z}
\end{array}
$$



Inherited Attributes: 7

C-style declarations generating **int x, y, z**.

$$\begin{array}{lcl} D & \rightarrow & T\ L \\ L & \rightarrow & L , I \mid I \end{array} \qquad \begin{array}{lcl} T & \rightarrow & \mathbf{int} \mid \mathbf{float} \\ I & \rightarrow & \mathbf{x} \mid \mathbf{y} \mid \mathbf{z} \end{array}$$



Home Page

Title Page

Page 104 of 100

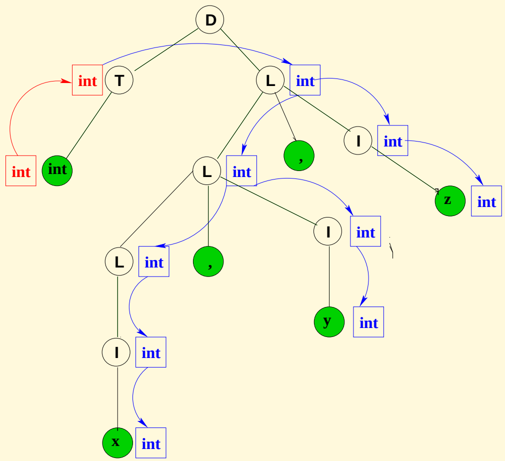
Go Back

Full Screen

Close

Quit

An Attribute Grammar

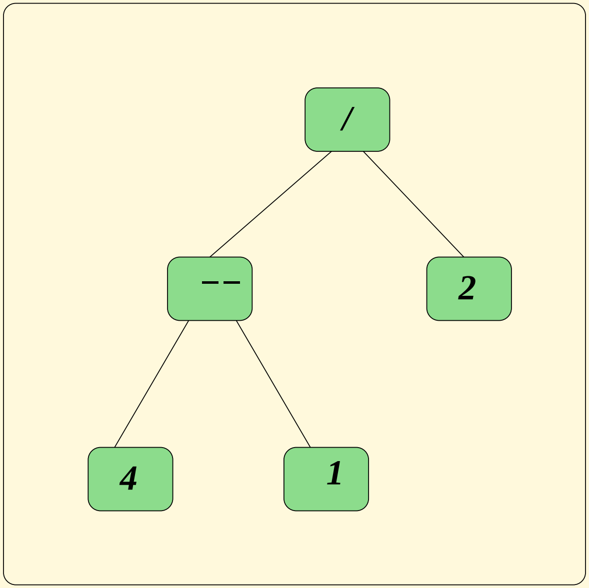


$$
\begin{array}{llll}
D & \rightarrow & T L & \triangleright \; L.in := T.type \\
T & \rightarrow & \mathbf{int} & \triangleright \; T.type := \mathbf{int}.int \\
T & \rightarrow & \mathbf{float} & \triangleright \; T.type := \mathbf{float}.float \\
L_0 & \rightarrow & L_1 , I & \triangleright \; L_1 := L_0.in \\
L & \rightarrow & I & \triangleright \; I.in := L.in \\
I & \rightarrow & \mathbf{id} & \triangleright \; \mathbf{id}.type := I.in
\end{array}
$$

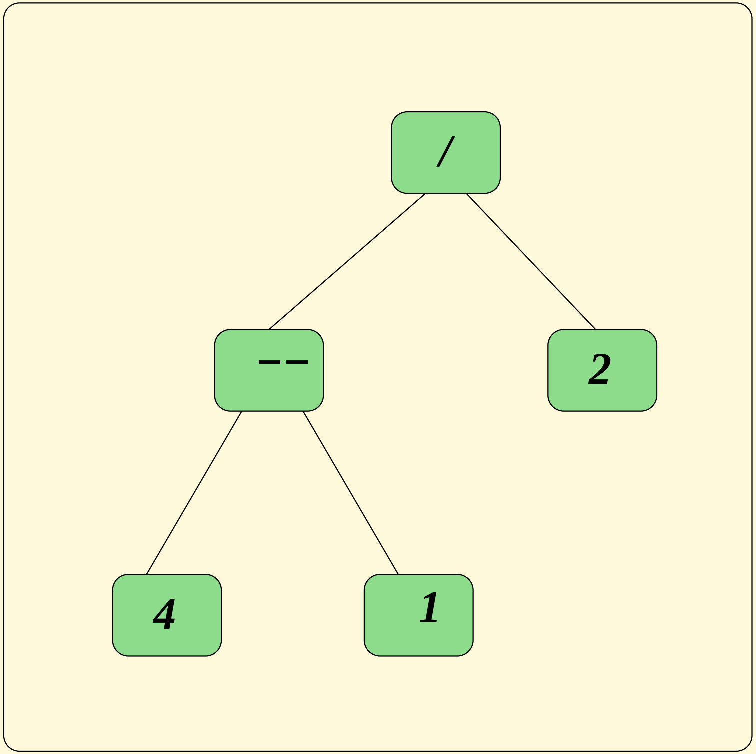
Abstract Syntax: 0

$$
\begin{array}{lcl}
E & \longrightarrow & E - T \mid T \\
T & \longrightarrow & T / F \mid F \\
F & \longrightarrow & \mathbf{n} \mid (E)
\end{array}
$$

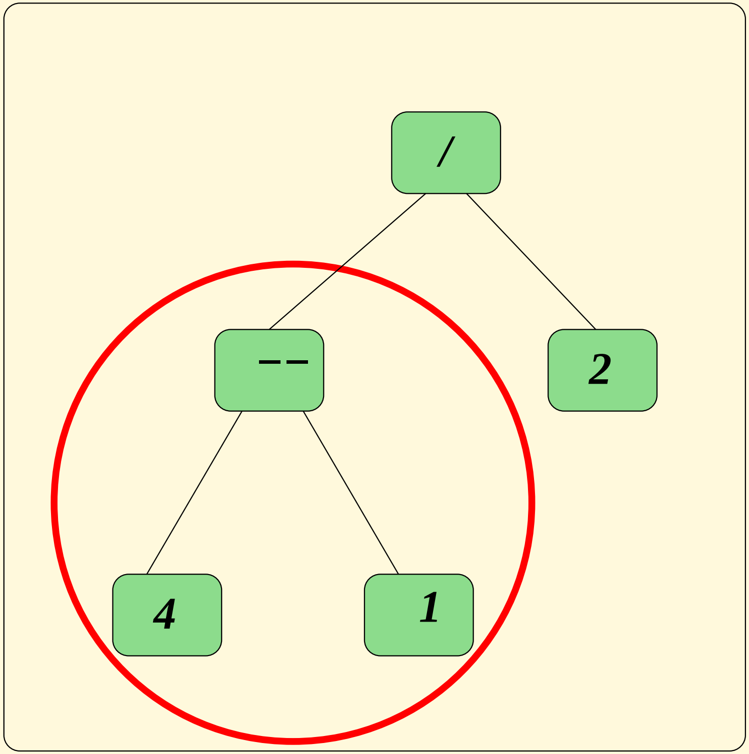
Suppose we want to evaluate an expression $(4-1)/2$. What we actually want is a tree that looks like this:



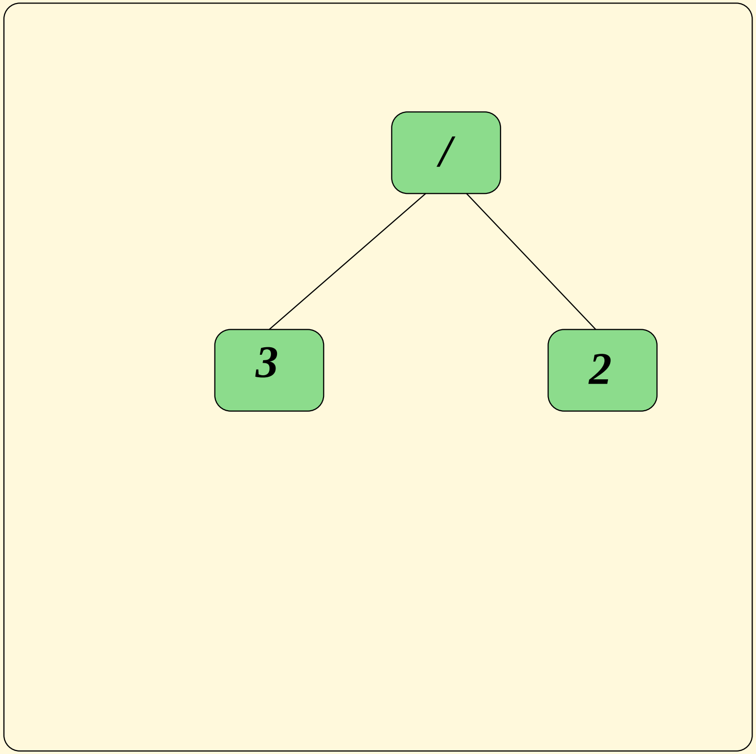
Evaluation: 0



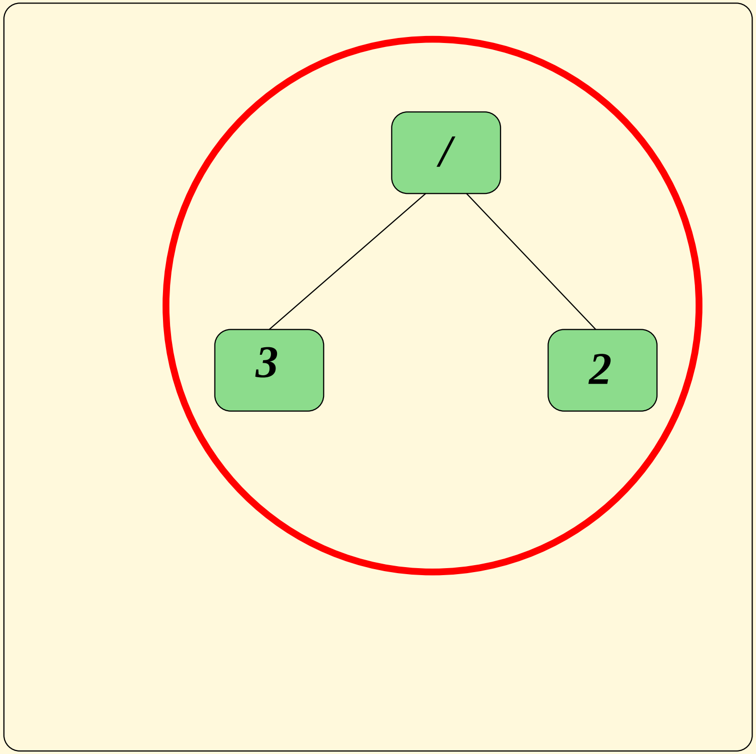
Evaluation: 1



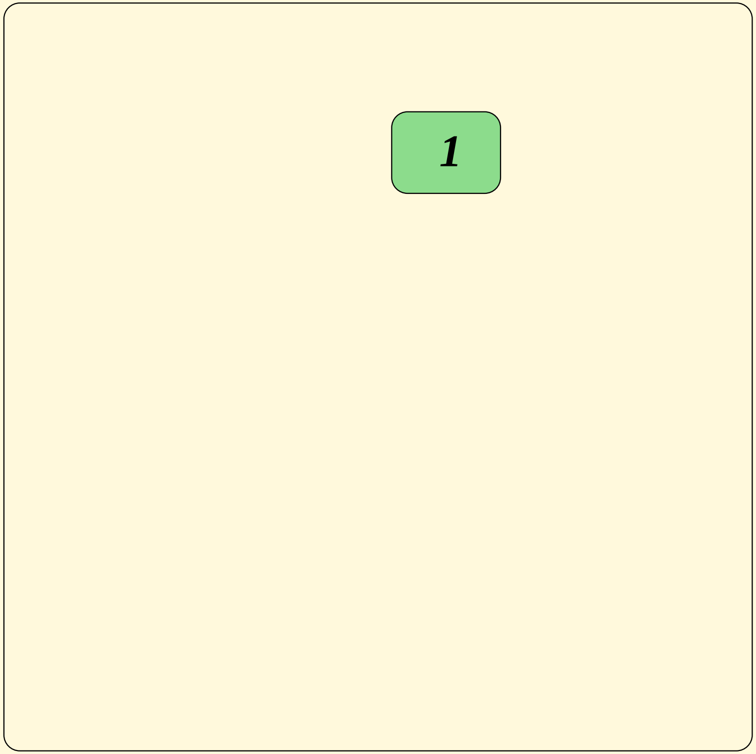
Evaluation: 2



Evaluation: 3



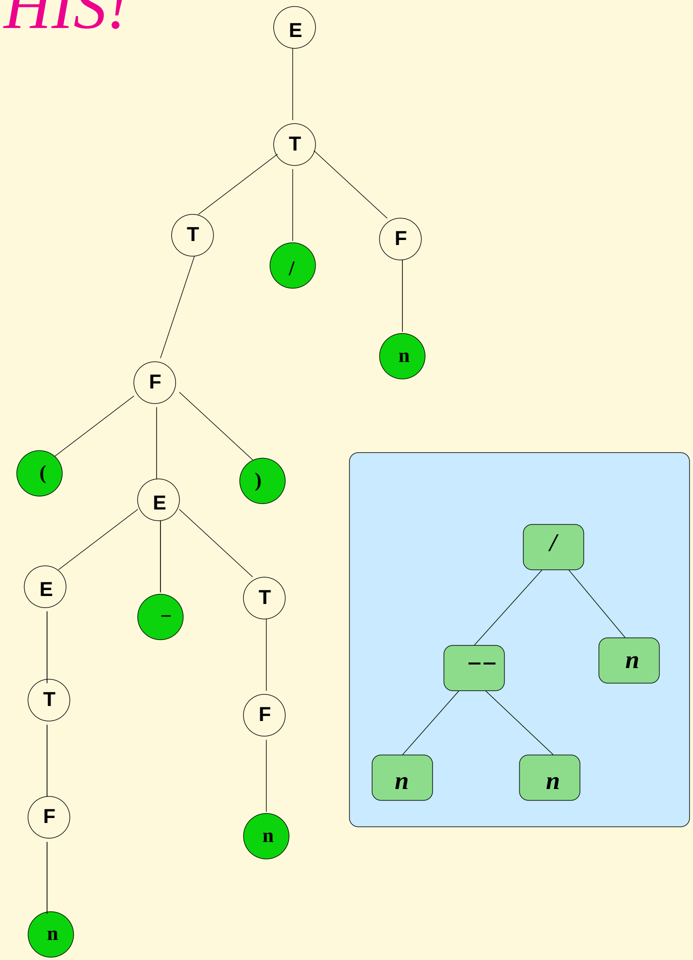
Evaluation: 4



But what we actually get during parsing is a tree that looks like . . .

Abstract Syntax: 1

. . . THIS!



Abstract Syntax: 2

We use attribute grammar rules to construct the abstract syntax tree (AST)!.
But in order to do that we first require two procedures for tree construction.

makeLeaf(literal) : Creates a node with label **literal** and returns a pointer to it.

makeBinaryNode(opr, opd1, opd2) : Creates a node with label **opr** (with fields which point to opd1 and opd2) and returns a pointer to the newly created node.

Now we may associate a synthesized attribute called ptr with each terminal and nonterminal symbol which points to the root of the subtree created for it.

Abstract Syntax: 3

$$
\begin{array}{llll}
E_0 & \rightarrow & E_1 - T & \triangleright \; E_0.ptr := makeBinaryNode(-, E_1.ptr, T.ptr) \\
E & \rightarrow & T & \triangleright \; E.ptr := T.ptr \\
T_0 & \rightarrow & T_1 / F & \triangleright \; T_0.ptr := makeBinaryNode(/, T_1.ptr, F.ptr) \\
T & \rightarrow & F & \triangleright \; T.ptr := F.ptr \\
F & \rightarrow & (E) & \triangleright \; F.ptr := E.ptr \\
F & \rightarrow & \mathbf{n} & \triangleright \; F.ptr := makeLeaf(\mathbf{n}.val)
\end{array}
$$

The Big Picture

Symbol Table:1

- The store house of context-sensitive and run-time information about every identifier in the source program.
- All accesses relating to an identifier require to first find the attributes of the identifier from the symbol table
- Usually organized as a hash table – provides fast access.
- Compiler-generated temporaries may also be stored in the symbol table

Symbol Table:2

Attributes stored in a symbol table for each identifier:

- type
- size
- scope/visibility information
- base address
- addresses to location of auxiliary symbol tables (in case of records, procedures, classes)
- address of the location containing the string which actually names the identifier and its length in the string pool

Symbol Table:3

- A symbol table exists through out the compilation and run-time.
- Major operations required of a symbol table:
  - insertion
  - search
  - deletions are purely logical (depending on scope and visibility) and not physical
- Keywords are often stored in the symbol table before the compilation process begins.

Symbol Table:4

Accesses to the symbol table at every stage of the compilation process,

Scanning: Insertion of new identifiers.

Parsing: Access to the symbol table to ensure that an operand exists (declaration before use).

Semantic analysis:

- Determination of types of identifiers from declarations
- type checking to ensure that operands are used in type-valid contexts.
- Checking scope, visibility violations.

Symbol Table:5

IR generation: . Memory allocation and relative[a] address calculation.

Optimization: All memory accesses through symbol table

Target code: Translation of relative addresses to absolute addresses in terms of word length, word boundary etc.

The Big picture

[a] i.e. relative to a base address that is known only at run-time

Home Page

Title Page

Page 120 of 100

Go Back

Full Screen

Close

Quit

Intermediate Representation

Intermediate representations are important for reasons of portability.

- *(more or less)* independent of specific features of the high-level language.

  Example. Java byte-code for any high-level language.

- *(more or less)* independent of specific features of any particular target architecture (e.g. number of registers, memory size)

  - number of registers
  - memory size
  - word length

IR Properties: 1

1. It is fairly low-level containing instructions common to all target architectures and assembly languages.

   How low can you stoop? . . .

2. It contains some fairly high-level instructions that are common to most high-level programming languages.

   How high can you rise?

3. To ensure portability
   - an unbounded number of variables and memory locations
   - no commitment to Representational Issues

4. To ensure type-safety
   - memory locations are also typed according to the data they may contain,
   - no commitment is made regarding word boundaries, and the structure of individual data items.

Next

IR: Representation?

- No commitment to word boundaries or byte boundaries
- No commitment to representation of
  - int vs. float,
  - float vs. double,
  - packed vs. unpacked,
  - strings – where and how?.

Back to IR Properties:1

IR: How low can you stoop?

- most arithmetic and logical operations, load and store instructions etc.
- so as to be interpreted easily,
- the interpreter is fairly small,
- execution speeds are high,
- to have fixed length instructions (where each operand position has a specific meaning).

Back to IR Properties:1

IR: How high can you rise?

- typed variables,
- temporary variables instead of registers.
- array-indexing,
- random access to record fields,
- parameter-passing,
- pointers and pointer management
- no limits on memory addresses

Back to IR Properties:1

A typical instruction set: 1

Three address code: A suite of instructions. Each instruction has at most 3 operands.

- an opcode representing an operation with at most 2 operands
- two operands on which the binary operation is performed
- a target operand, which accumulates the result of the (binary) operation.

If an operation requires less than 3 operands then one or more of the operands is made null.

A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

A typical instruction set: 2

- Assignments (LOAD-STORE)
  - `x := y bop z`, where `bop` is a binary operation
  - `x := uop y`, where `uop` is a unary operation
  - `x := y`, load, store, copy or register transfer
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
  - `goto L` – Unconditional jump,
  - `x relop y goto L` – Conditional jump, where `relop` is a relational operator
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
  - `call p n`, where `n` is the number of parameters
  - `return y`, return value from a procedures call
  - `param x`, parameter declaration
- Arrays and array-indexing
- Pointer Referencing and Dereferencing

A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
  - `x := a[i]` – array indexing for *r-value*
  - `a[j] := y` – array indexing for *l-value*

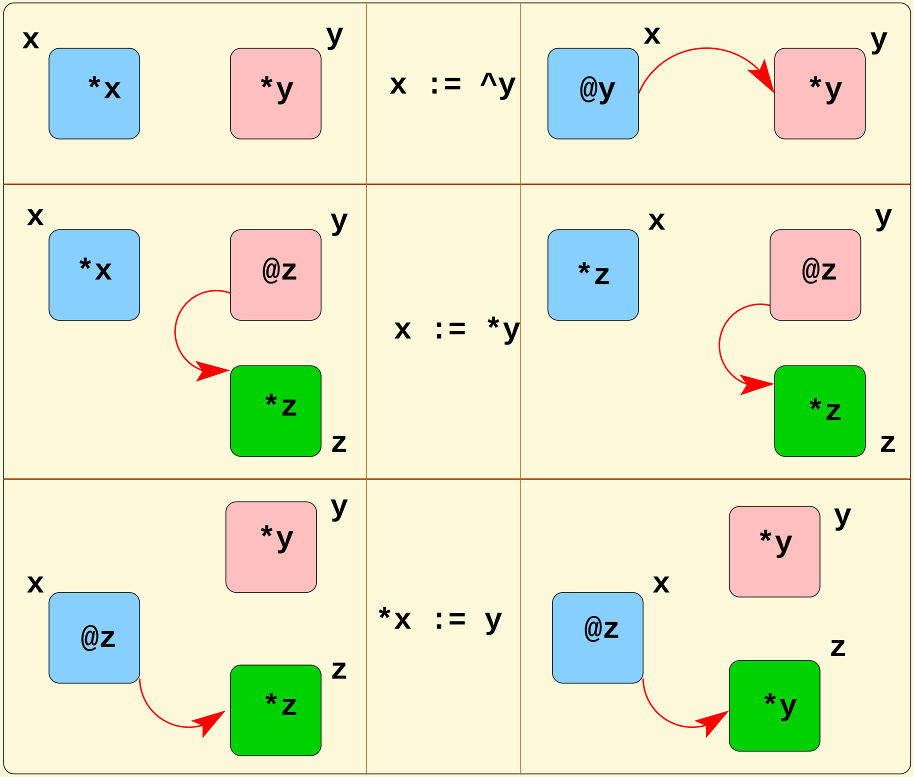
  Note: The two opcodes are different depending on whether *l-value* or *r-value* is desired. `x` and `y` are *always* simple variables
- Pointer Referencing and Dereferencing

A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing
  - `x := ^y` – referencing: set x to point to y
  - `x := *y` – dereferencing: copy contents of location pointed to by y into x
  - `*x := y` – dereferencing: copy *r-value* of y into the location pointed to by x

Picture

Pointers



Home Page
Title Page
Page 133 of 100
Go Back
Full Screen
Close
Quit

IR: Generation

- Can be generated by recursive traversal of the abstract syntax tree.
- Can be generated by syntax-directed translation as follows:

  For every non-terminal symbol N in the grammar of the source language there exist two attributes

  N.place , which denotes the address of a temporary variable where the result of the execution of the generated code is stored

  N.code , which is the actual code segment generated.
- In addition a global counter for the instructions generated is maintained as part of the generation process.
- It is independent of the source language but can express target machine operations without committing to too much detail.

IR: Infrastructure

Given an abstract syntax tree T, with T also denoting its root node.

T.place address of temporary variable where result of execution of the T is stored.

newtemp returns a *fresh* variable name and also installs it in the symbol table along with relevant information

T.code the actual sequence of instructions generated for the tree T.

newlabel returns a *label* to mark an instruction in the generated code which may be the target of a jump.

emit emits an instructions (regarded as a string).

IR: Infrastructure

Colour and font coding of IR code generation

- *Green*: Nodes of the Abstract Syntax Tree
- `Brown`: Characters and strings of the `Intermediate Representation`
- *Red*: Variables and data structures of the *language* in which the IR code generator is written
- *blue*: Names of relevant *procedures* used in IR code generation.
- *Black*: All other stuff.

IR: Example

$$E \to id \quad \triangleright$$

$$\begin{aligned} E.place &:= id.place; \\ E.code &:= emit() \end{aligned}$$

$$E_0 \to E_1 - E_2 \quad \triangleright$$

$$\begin{aligned} E_0.place &:= newtemp; \\ E_0.code &:= E_1.code \\ &\;\;\,|| \;\; E_2.code \\ &\;\;\,|| \;\; emit(E_0.place \;:= E_1.place - E_2.place) \end{aligned}$$

IR: Example

$$S \rightarrow id := E \qquad \triangleright$$

$$\begin{array}{lcl} S.code & := & E.code \\ & \| & emit(id.place{:=}E.place) \end{array}$$

$$S_0 \rightarrow while\ E\ do\ S_1 \qquad \triangleright$$

$$\begin{array}{lcl} S_0.begin & := & newlabel; \\ S_0.after & := & newlabel; \\ S_0.code & := & emit(S_0.begin:) \\ & \| & E.code \\ & \| & emit(\texttt{if}\ E.place{=}\ \texttt{0 goto}\ S_0.after) \\ & \| & S_1.code \\ & \| & emit(\texttt{goto} S_0.begin) \\ & \| & emit(S_0.after:) \end{array}$$

Home Page

Title Page

◀◀ ▶▶

◀ ▶

Go Back

Full Screen

Close

Quit

IR: Example

$$S \rightarrow id := E \qquad \triangleright$$

$$\begin{aligned} S.code &:= E.code \\ &\;\;\|\;\; emit(id.place{:=}E.place) \end{aligned}$$

$$S_0 \rightarrow while\ E\ do\ S_1 \qquad \triangleright$$

$$\begin{aligned} S_0.begin &:= newlabel; \\ S_0.after &:= newlabel; \\ S_0.code &:= emit(S_0.begin:) \\ &\;\;\|\;\; E.code \\ &\;\;\|\;\; emit(\texttt{if}\ E.place{=}\ \texttt{0 goto}\ S_0.after) \\ &\;\;\|\;\; S_1.code \\ &\;\;\|\;\; emit(\texttt{goto} S_0.begin) \\ &\;\;\|\;\; emit(S_0.after:) \end{aligned}$$

IR: Generation

While generating the intermediate representation, it is sometimes necessary to generate jumps into code that has not been generated as yet (hence the address of the label is unknown). This usually happens while processing

- forward jumps
- short-circuit evaluation of boolean expressions

It is usual in such circumstances to either fill up the empty label entries in a second pass over the the code or through a process of backpatching (which is the maintenance of lists of jumps to the same instruction number), wherein the blank entries are filled in once the sequence number of the target instruction becomes known.

A Calling Chain

Main program

Globals

Procedure P2
Locals of P2

Procedure P21

Locals of P21

Body of P21

Call P21

Body of P2

Call P21

Procedure P1

Locals of P1

Body of P1

Call P2

Main body

Call P1

Run-time Structure: 1

Main program

Globals

Procedure P2

Locals of P2

Procedure P21

Locals of P21

Body of P21

Body of P2

Procedure P1

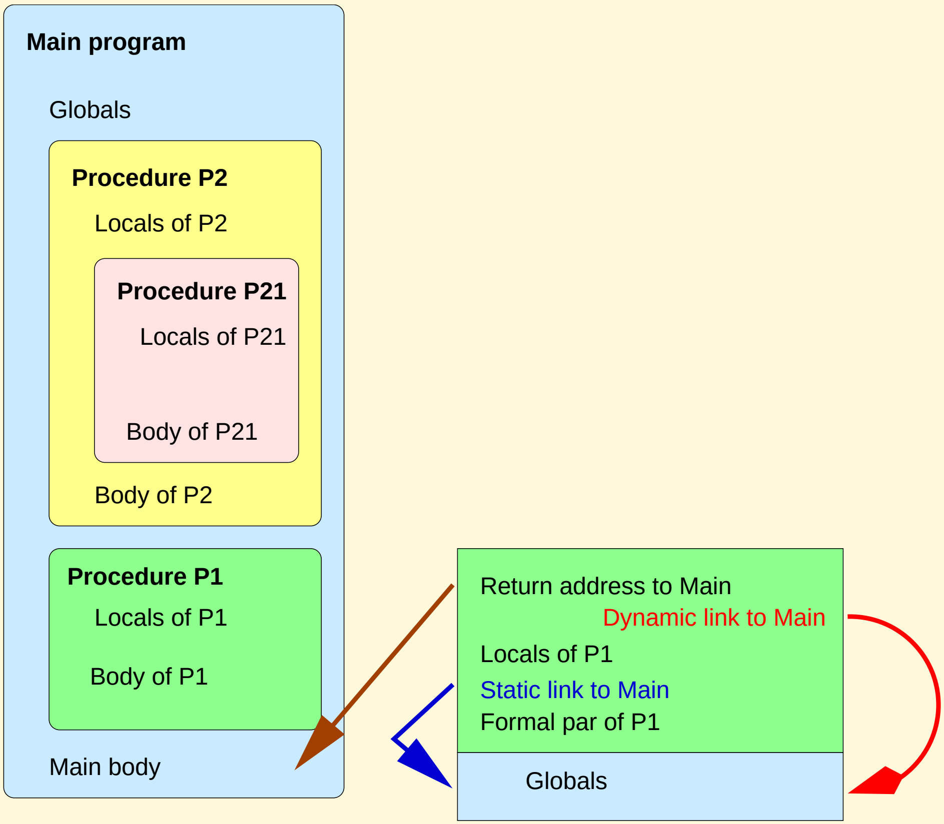
Locals of P1

Body of P1

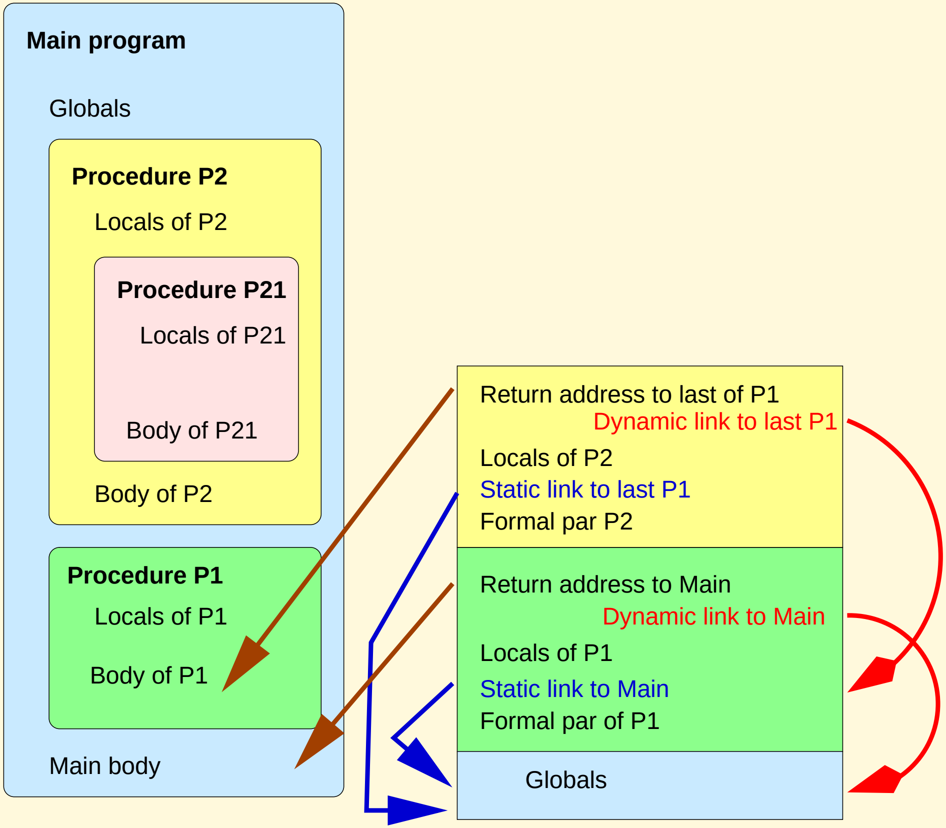
Main body

Globals

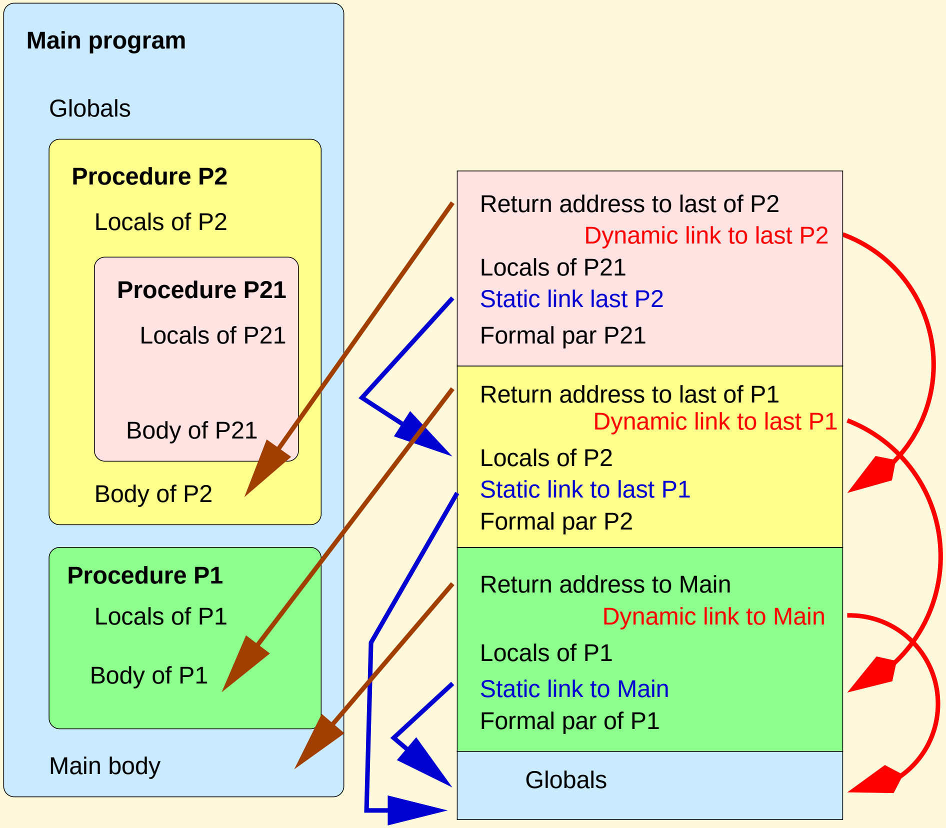
Run-time Structure: 2



Run-time Structure: 3



Run-time Structure: 4



Run-time Structure: 5