Context-Free Grammar

- Is a specification for the syntax of a programming language
- Is a set of **rewriting rules** or **productions** of the form:

  $A \rightarrow X_1\ X_2\ \dots\ X_n$

  - A production has exactly one symbol A on the left-hand side (LHS)
  - Can have zero, one, or more symbols X_i on the right-hand side (RHS)
- For example, a while statement is syntactically defined as:

  while-stmt → **while** *expr* **do** *stmt-list* **end ;**
- Two kinds of symbols may appear in a context-free grammar:
  - Nonterminals appear in *italic*
  - Terminals appear in **bold**
- A **nonterminal** is a place holder
  - Is rewritten by the RHS of a production where it appears on the LHS
- **Terminals** represent the tokens of a language

Language of a Context-Free Grammar

- A sequence of tokens is syntactically legal if
  - It can be **derived** by applying the productions of the CFG
- A context-free grammar defines a language
  - This language is a set of strings (sequences) of tokens (terminals)
  - Each string of tokens is **derivable** from the production rules of the CFG
- Consider the following simplified grammar for expressions

$$expr \rightarrow expr\ \ op\ \ expr \mid \mathbf{(}\ expr\ \mathbf{)} \mid \mathbf{id} \mid \mathbf{num}$$

$$op \rightarrow \texttt{+} \mid \texttt{–} \mid \texttt{*} \mid \texttt{/}$$

  - The string: **`(id+num)*id`** is syntactically legal and part of the language
  - Similarly: **`id*(num-id)`** is also part of the language
  - However: **`(id+num`** is NOT part of the language
  - Similarly: **`id*-num+`** is NOT part of the language

Derivations

- To check whether a sequence of tokens is legal or not:
    - We start with a nonterminal called the **start** symbol
    - We apply productions, rewriting nonterminals, until only terminals remain
    - A **derivation** replaces a nonterminal on LHS of a production with RHS
    - The ⇒ symbol denotes a derivation step
- For example, a derivation for `(id + num)*id` is given below:

$$\begin{aligned}\mathit{expr} &\Rightarrow \mathit{expr\ op\ expr} \Rightarrow \mathbf{(}\ \mathit{expr}\ \mathbf{)}\ \mathit{op\ expr} \Rightarrow \mathbf{(}\ \mathit{expr\ op\ expr}\ \mathbf{)}\ \mathit{op\ expr} \\ &\Rightarrow \mathbf{(}\ \mathit{expr} + \mathit{expr}\ \mathbf{)}\ \mathit{op\ expr} \Rightarrow \mathbf{(}\ \mathit{expr} + \mathit{expr}\ \mathbf{)}\ \mathbf{*}\ \mathit{expr} \\ &\Rightarrow \mathbf{(\ id} + \mathit{expr}\ \mathbf{)\ *}\ \mathit{expr} \Rightarrow \mathbf{(\ id + num\ )\ *}\ \mathit{expr} \Rightarrow \mathbf{(\ id + num\ )\ *\ id}\end{aligned}$$

- Similarly, a derivation for `id*(num - id)` is given below:

$$\begin{aligned}\mathit{expr} &\Rightarrow \mathit{expr\ op\ expr} \Rightarrow \mathit{expr\ op}\ \mathbf{(}\ \mathit{expr}\ \mathbf{)} \Rightarrow \mathit{expr\ op}\ \mathbf{(}\ \mathit{expr\ op\ expr}\ \mathbf{)} \\ &\Rightarrow \mathit{expr}\ \mathbf{*\ (}\ \mathit{expr\ op\ expr}\ \mathbf{)} \Rightarrow \mathit{expr}\ \mathbf{*\ (}\ \mathit{expr} - \mathit{expr}\ \mathbf{)} \\ &\Rightarrow \mathbf{id\ *\ (}\ \mathit{expr} - \mathit{expr}\ \mathbf{)} \Rightarrow \mathbf{id\ *\ (\ num} - \mathit{expr}\ \mathbf{)} \Rightarrow \mathbf{id\ *\ (\ num - id\ )}\end{aligned}$$

Formal Definition of a Context-Free Grammar

- A Context-Free Grammar consists of
  - A finite **set of terminals** T
  - A finite **set of nonterminals** N (disjoint from T)
  - A **start symbol** $S \in N$
  - A **set of productions** or **grammar rules** P
    - A production rule is of the form: $A \rightarrow X_1 X_2 \ldots X_n$, where $A \in N$ and $X_i \in N \cup T$
    - A production with zero symbols on the RHS ($n = 0$) is of the form: $A \rightarrow \varepsilon$
    - A production is also written as: $A \rightarrow \alpha$, where $\alpha \in (N \cup T)^*$
- The following notation is used

$a, b, c, \ldots$	denote terminal symbols in T
$A, B, C, \ldots$	denote nonterminal symbols in N
X	denotes a grammar symbol in $N \cup T$
$\alpha, \beta, \gamma, \ldots$	denote strings of grammar symbols in $(N \cup T)^*$ including ε
x	denotes a string of terminals in T^* including ε

More Formal Definitions

- If $A \rightarrow \alpha$ is a production then $\beta A\gamma \Rightarrow \beta\alpha\gamma$ is a derivation step
  - The nonterminal A is replaced with α using the production $A \rightarrow \alpha$
- The derivation symbol $\Rightarrow$ can be extended to

  $\Rightarrow^+$ derived in one or more steps

  $\Rightarrow^*$ derived in zero or more steps
- If the start symbol $S \Rightarrow^* \beta$ then β is called a **sentential form**
- The **language of a grammar** G is $L(G) = \{x \in T^* \mid \mathrm{S} \Rightarrow^+ x\}$
- Often more than one production share the same LHS

  $A \rightarrow \alpha \mid \beta \mid \ldots \mid \zeta$ is an abbreviation for

  $A \rightarrow \alpha \quad A \rightarrow \beta \quad \ldots \quad A \rightarrow \zeta$

Leftmost and Rightmost Derivations

- When deriving a sequence of tokens …
  - More than one nonterminal may be present and can be expanded
  - A **leftmost derivation** chooses the leftmost nonterminal to expand
  - A leftmost derivation is denoted by $\Rightarrow_{\mathrm{lm}}$
  - A **rightmost derivation** chooses the rightmost nonterminal to expand
  - A rightmost derivation is denoted by $\Rightarrow_{\mathrm{rm}}$
- A leftmost derivation for `(id + num)*id`

$$
\begin{aligned}
\mathit{expr} &\Rightarrow_{\mathrm{lm}} \mathit{expr\ op\ expr} \Rightarrow_{\mathrm{lm}} \mathbf{(}\ \mathit{expr}\ \mathbf{)}\ \mathit{op\ expr} \Rightarrow_{\mathrm{lm}} \mathbf{(}\ \mathit{expr\ op\ expr}\ \mathbf{)}\ \mathit{op\ expr} \\
&\Rightarrow_{\mathrm{lm}} \mathbf{(\ id}\ \mathit{op\ expr}\ \mathbf{)}\ \mathit{op\ expr} \Rightarrow_{\mathrm{lm}} \mathbf{(\ id} + \mathit{expr}\ \mathbf{)}\ \mathit{op\ expr} \\
&\Rightarrow_{\mathrm{lm}} \mathbf{(\ id + num\ )}\ \mathit{op\ expr} \Rightarrow_{\mathrm{lm}} \mathbf{(\ id + num\ ) *}\ \mathit{expr} \Rightarrow_{\mathrm{lm}} \mathbf{(\ id + num\ ) * id}
\end{aligned}
$$

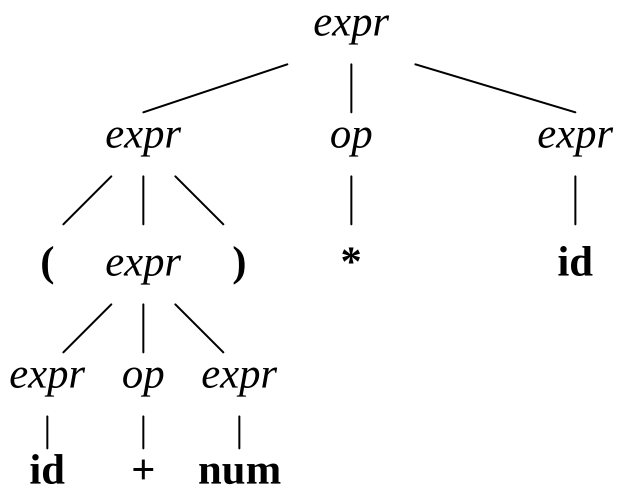
- A rightmost derivation for `(id + num)*id`

$$
\begin{aligned}
\mathit{expr} &\Rightarrow_{\mathrm{rm}} \mathit{expr\ op\ expr} \Rightarrow_{\mathrm{rm}} \mathit{expr\ op}\ \mathbf{id} \Rightarrow_{\mathrm{rm}} \mathit{expr}\ \mathbf{*\ id} \Rightarrow_{\mathrm{rm}} \mathbf{(}\ \mathit{expr}\ \mathbf{)\ *\ id} \\
&\Rightarrow_{\mathrm{rm}} \mathbf{(}\ \mathit{expr\ op\ expr}\ \mathbf{)\ *\ id} \Rightarrow_{\mathrm{rm}} \mathbf{(}\ \mathit{expr\ op}\ \mathbf{num\ )\ *\ id} \\
&\Rightarrow_{\mathrm{rm}} \mathbf{(}\ \mathit{expr} + \mathbf{num\ )\ *\ id} \Rightarrow_{\mathrm{rm}} \mathbf{(\ id + num\ )\ *\ id}
\end{aligned}
$$

Parse Tree

- Is a graphical representation for a derivation
  - Filters out choice regarding replacement order
  - Rooted by the start symbol S
  - Interior nodes represent nonterminals in N
  - Leaf nodes are terminals in T or ε
  - Node A can have children $X_1\, X_2 \ldots X_n$ if a rule $A \rightarrow X_1\, X_2 \ldots X_n$ exists
- The following is a parse tree for `( id + num ) * id`

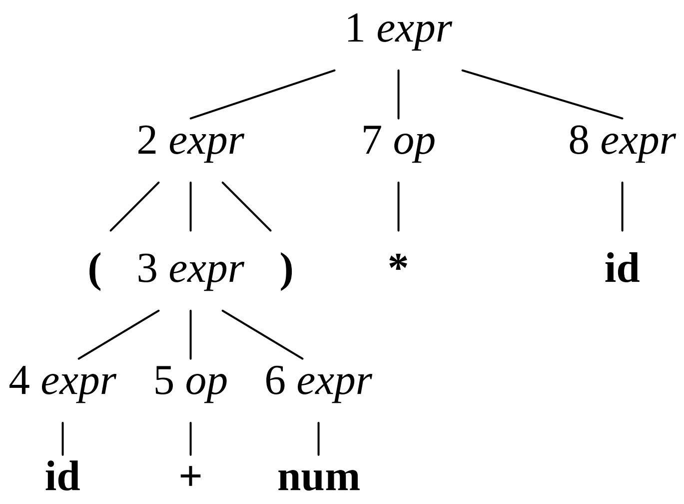
expr → *expr* *op* *expr*
expr → **(** *expr* **)**
expr → **id**
expr → **num**
op → + | – | * | /



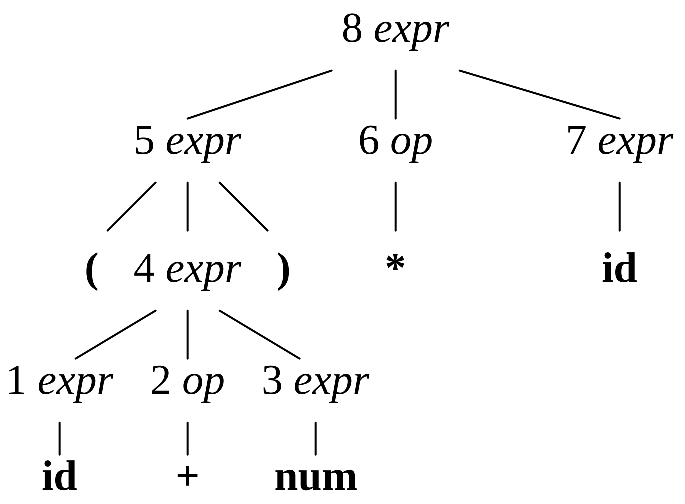
Preorder and Postorder Traversal

- A parse tree has a unique leftmost and rightmost derivation
- Leftmost derivation is a **Preorder traversal** of a parse tree
- The **reverse** of a rightmost derivation is **Postorder traversal**
- Preorder traversal corresponds to **top-down parsing**
- Postorder traversal corresponds to **bottom-up parsing**

Preorder Traversal of Internal Nodes


1 *expr*
2 *expr* 7 *op* 8 *expr*
(3 *expr*) ***** **id**
4 *expr* 5 *op* 6 *expr*
id + **num**

Postorder Traversal


8 *expr*
5 *expr* 6 *op* 7 *expr*
(4 *expr*) ***** **id**
1 *expr* 2 *op* 3 *expr*
id + **num**

Ambiguous Grammars

- A grammar is called **ambiguous** if
  - It permits a terminal string to have more than one parse tree
  - This means also more than one leftmost derivation for a given string
  - Also, more than one rightmost derivation for same string
- The grammar for expressions used so far is ambiguous
  - Two parse trees for **id + id * id**

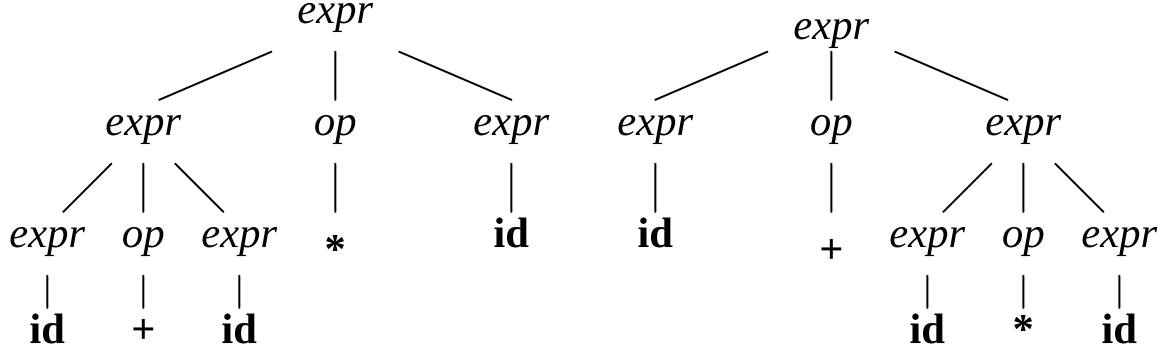
$expr \rightarrow expr\ \ op\ \ expr$

$expr \rightarrow (\ \ expr\ \ )$

$expr \rightarrow \textbf{id}$

$expr \rightarrow \textbf{num}$

$op \rightarrow +\ |\ -\ |\ *\ |\ /$



- Ambiguous grammars should be avoided
  - Do not guarantee unique parsing and translation
  - Expression evaluation is not clearly defined in the above grammar

Eliminating Ambiguity in Expressions

- To guarantee unique translation, ambiguity should be eliminated
  - Unfortunately, there is NO algorithm that detects ambiguity in any CFG
  - However, some classes of grammars can be shown to be unambiguous
- To handle ambiguity in expressions …
  - The **precedence** and **associativity** of operators specify order of evaluation
  - Higher precedence operators are evaluated first
  - Equal precedence operators are evaluated according to associativity
    - Left-to-right or Right-to-left
- To handle precedence of operators …
  - We divide operators into groups of equal precedence
  - For each precedence level, we introduce a nonterminal and grammar rules
- To handle associativity of operators …
  - We design grammar rules to be either left or right recursive

Eliminating Ambiguity in Expressions – cont'd

- Ambiguous Grammar for Expressions:

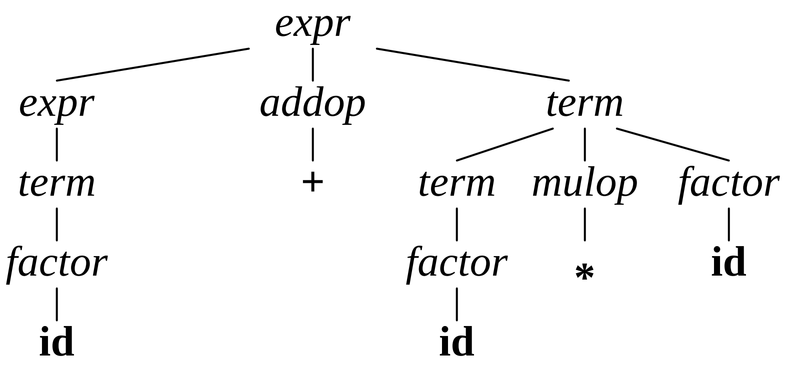
expr → *expr op expr*
expr → (*expr*)
expr → **id**
expr → **num**
op → + | – | * | /

Left and Right Recursive => Ambiguous

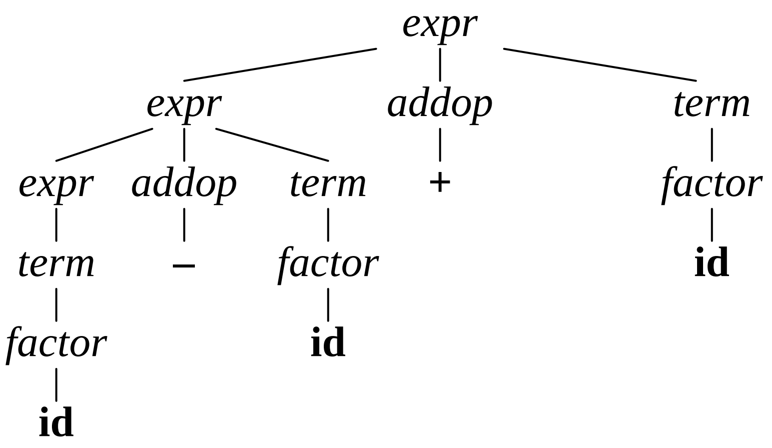
- Unambiguous Grammar:

expr → *expr addop term*
expr → *term*
term → *term mulop factor*
term → *factor*
factor → (*expr*)
factor → **id**
factor → **num**
addop → + | –
mulop → * | /

One parse tree for id + id * id



One parse tree for id – id + id



Ambiguity of Else in If Statements

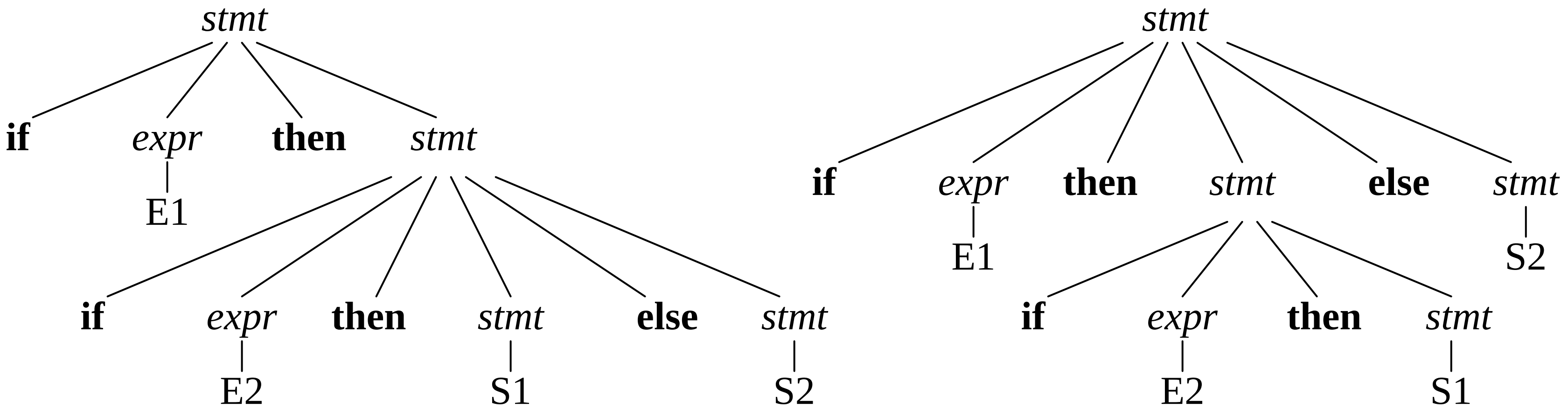
- Consider the following grammar for **if** statements:

  stmt → **if** *expr* **then** *stmt*

  stmt → **if** *expr* **then** *stmt* **else** *stmt*

  stmt → *other-stmt*

- There are two parse trees for: **if** E1 **then** **if** E2 **then** S1 **else** S2
  - The two parse trees translate differently the **else** part
  - The else part can be attached to inner if (should be the case) or to outer if



Eliminating Ambiguity of Else in If Statements

- To eliminate ambiguity of **else** in **if** statements ...
  - We distinguish **matched if** statements from **unmatched** ones
  - We insist on having a **matched statement between then and else**
- Unambiguous grammar for **if** statements is given below

stmt → *matched* | *unmatched*

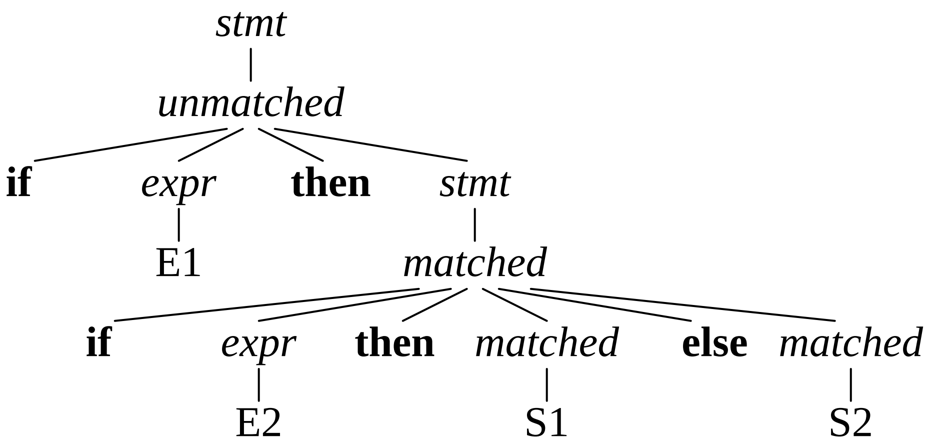
matched → **if** *expr* **then** *matched* **else** *matched* | *other-stmt*

unmatched → **if** *expr* **then** *stmt*

unmatched → **if** *expr* **then** *matched* **else** *unmatched*

One parse tree for:

if E1 **then if** E2 **then** S1 **else** S2



Extended BNF Notation

- A context free grammar is also called a BNF notation
  - BNF is the Backus-Naur Form (named after its inventors)
- Repetitive and optional sequences are common in grammars
  - An optional sequence is enclosed in brackets [and]
  - An optional and repetitive sequence is enclosed in braces { and }
- For example, a statement sequence can be defined in many ways

stmt-seq → *stmt-seq* **;** *stmt* | *stmt* **BNF Left Recursive**

stmt-seq → *stmt* **;** *stmt-seq* | *stmt* **BNF Right Recursive**

stmt-seq → *stmt* { **;** *stmt* } **EBNF Notation**

- An optional **else** part in an **if** statement terminated with **end**

if-stmt → **if** *expr* **then** *stmt-seq* [**else** *stmt-seq*] **end**

- EBNF has the same definitional capability of ordinary BNF
  - Advantage of EBNF grammars is that they are more compact and readable

The Chomsky Hierarchy

- The form of productions has a profound effect on grammar
- In **unrestricted grammars**, a production is of the form $\alpha \rightarrow \beta$
  - There is no restriction on α except that it should be different from ε
- In **context-sensitive grammars**, a production is $\beta A\gamma \rightarrow \beta\alpha\gamma$
  - Context-sensitive grammars are more powerful than context-free grammars
  - However, context-sensitive grammars are more difficult to parse
- In **context-free grammars**, a production is of the form $A \rightarrow \alpha$
  - There is no context for A; A may be replaced by α anywhere we like
- In **regular grammars**, a production is $A \rightarrow \mathbf{c}\ B$, $A \rightarrow \mathbf{c}$, or $A \rightarrow \varepsilon$
  - The language generated by a regular grammar is a **regular language**
  - Regular grammars are equivalent to regular expressions
  - For example, the regular expression **`ab*c`** can be described as:
    - $A \rightarrow \mathbf{a}\ B \qquad B \rightarrow \mathbf{b}\ B \mid \mathbf{c}$

Syntax of TINY Language

- The syntax of TINY language is given below:

program → *stmt-seq*

stmt-seq → *stmt-seq* *stmt* **;** | *stmt* **;** **semicolons terminate statements**

stmt → **if** *expr* **then** *stmt-seq* **end**

stmt → **if** *expr* **then** *stmt-seq* **else** *stmt-seq* **end**

stmt → **while** *expr* **do** *stmt-seq* **end**

stmt → **id** **:=** *expr*

stmt → **read** *id-seq*

stmt → **write** *expr-seq*

id-seq → *id-seq* **,** **id** | **id**

expr-seq → *expr-seq* **,** *expr* | *expr*

expr → *expr* **relop** *addexpr* | *addexpr*

addexpr → *addexpr* **addop** *mulexpr* | *mulexpr*

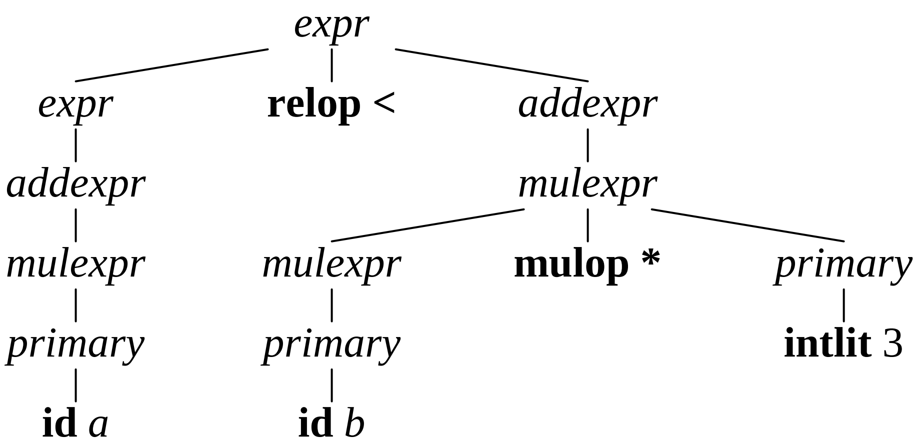
mulexpr → *mulexpr* **mulop** *primary* | *primary*

primary → **(** *expr* **)** | **id** | **intliteral** | **strliteral**

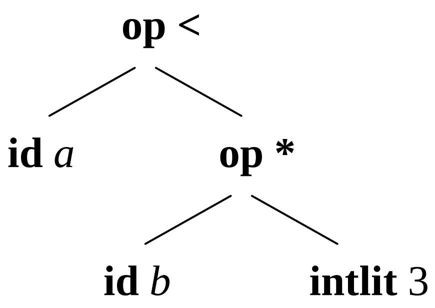
Syntax Trees

- A parse tree captures the derivation steps of a parser
- However, a parse tree is NOT useful to represent computation
  - Contains a lot more information than needed for translation
- A **syntax tree** is a more compact representation of a computation
  - Useful to generate, by a parser, as a first step in the translation process
  - Nonterminals and some tokens are unnecessary and hence removed
- Consider the expression: $a < b * 3$

Parse Tree

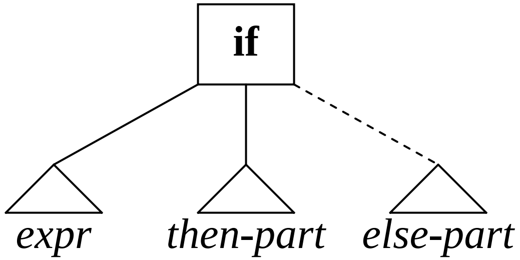


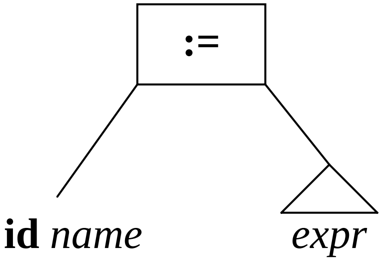
Syntax Tree

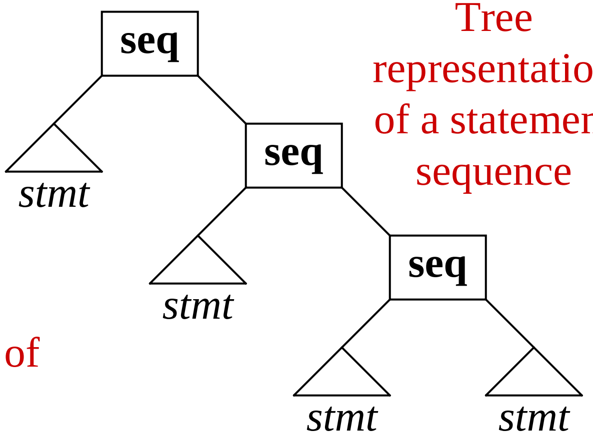


Syntax Trees – cont'd

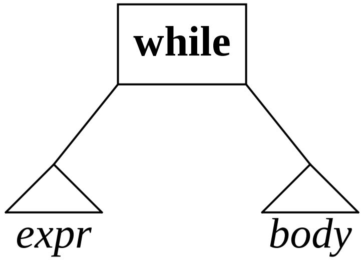
- Syntax trees are also appropriate to represent statements
  - A statement sequence can be represented either as a tree or as a linked list
- Syntax trees are also called **abstract syntax trees**
  - They represent the **abstract structure** of programs
  - Parse trees, however, represent the **concrete structure** of programs







Tree representation of a statement sequence



Linked list representation of a statement sequence