

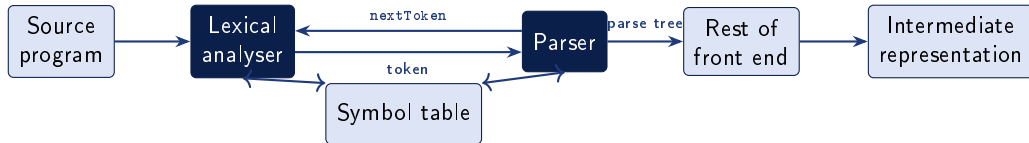
Syntax Analysis

Compiler Construction, Lecture 6

K.R. Chowdhary, Professor of CS

August 5–7, 2019 | Fall 2019

Why Grammars? The Role of the Parser



Benefits of a context-free grammar

- Precise, easy-to-read language specification
- Parsers can be built automatically; this reveals ambiguities
- Structure aids translation and error detection
- Languages can evolve by adding constructs

What the parser does

- Checks that the tokens are generated by the grammar
- Builds a parse tree for the rest of the compiler
- Reports syntax errors clearly and recovers to continue

Three general types

- **Universal:** works for any grammar but is inefficient, so not used in compilers
- **Top-down:** builds the parse tree from the root to the leaves
- **Bottom-up:** builds it from the leaves to the root

Both scan the input left to right, one symbol at a time, and efficient ones work only for grammar subclasses.

LL and LR

- **LL:** Left-to-right scan, Leftmost derivation; often hand-written (predictive parsing)
- **LR(k):** Left-to-right scan, Rightmost derivation in reverse, k look-ahead symbols; usually built with tools

Left recursion

A production is left recursive if the first symbol of its body equals its head, e.g. $E \rightarrow E + T$.

Derivations, Parse Trees and Ambiguity

Grammar:

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

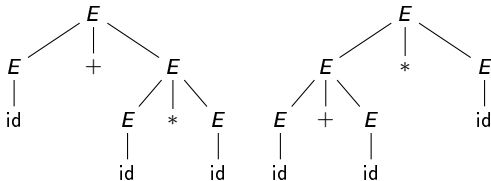
Derivation 1

$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow id + E * E$
 $\Rightarrow id + id * E \Rightarrow id + id * id$

Derivation 2

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow id + E * E$
 $\Rightarrow id + id * E \Rightarrow id + id * id$

A grammar with more than one
parse tree for a string is
ambiguous. Parentheses remove it:
 $id + (id * id)$ has one tree.



Tree 1: $id + (id * id)$

Tree 2: $(id + id) * id$

Grammars for Expressions

Expressions are harder than keyword-led constructs because of operator **precedence** and **associativity**. Use expressions E , terms T and factors F .

LR grammar (bottom-up)

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

Left recursive, so **not** usable for top-down parsing.

Left recursion removed (top-down)

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

The grammar $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$ treats $+$ and $*$ alike. It is ambiguous: $a + b * c$ has more than one parse tree.

Kinds of errors

- **Lexical:** misspelt keywords or operators
- **Syntax:** missing brackets or braces, misplaced operators
- **Semantic:** operator and operand mismatch
- **Logical:** wrong reasoning, e.g. + instead of *, or = for ==; hardest to detect

Error handler goals

Report errors clearly and accurately; recover quickly to find more; add minimal overhead for correct programs.

Recovery strategies

- **Panic mode:** discard input until a synchronizing token (e.g. ; or }) is found. Simple and cannot loop forever, but may skip much code
- **Phrase level:** local correction, e.g. replace a comma by a semicolon, or insert or delete one. Must avoid infinite loops
- **Error productions:** augment the grammar with productions for anticipated mistakes, so diagnostics are specific

Review questions

1. What do LL and LR stand for?
2. Why are expressions harder to parse than `while` or `if`?
3. Panic-mode vs. phrase-level recovery?
4. Main advantage of panic mode?
5. Which kind of error is a missing brace? A misspelt keyword?
6. Which recovery method may loop forever?

Exercises

1. Why can't left recursion be used in top-down parsers? Give examples
2. Why is a general-purpose parser uncommon?
3. Why can't $E \rightarrow E + E \mid E * E \mid \text{id}$ be parsed top-down?
4. Give left- and right-recursive grammars
5. Which language features help error detection and recovery?
6. Is exception handling for detection or recovery?
7. What are the primary functions of an error handler?