Context-free Grammars

Adapted from material by:

Prof. Alex Aiken and Prof. George Necula (UCB)

Outline

- Regular languages revisited
- Parser overview
- Context-free grammars (CFGs)
- Derivations
- Ambiguity

Regularity

- Languages requiring counting modulo a fixed number are *regular*.
- Finite automaton cannot count without limit or remember number of times it has visited a particular state.
- Many languages are not regular. E.g., Language of balanced parentheses is *not* regular.

$$\{(^i)^i \mid i \geq 0\}$$

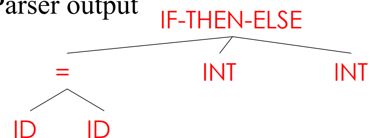
Parsing : *Example*

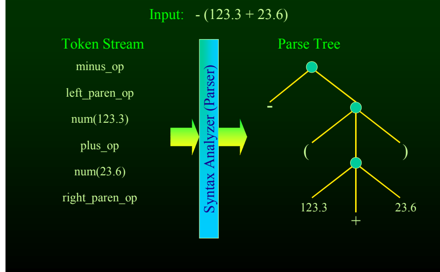
- Cool

  if x = y then 1 else 2 fi
- Parser input

  IF ID = ID THEN INT ELSE INT FI
- Parser output

IF-THEN-ELSE





Context-Free Grammars

- Programming language constructs have recursive structure.
  - An EXPR is
    if EXPR then EXPR else EXPR fi
    while EXPR loop EXPR pool
    …
- Context-free grammars are a natural notation for this recursive structure.
  - Iteration : Regular Expression
  - Tail Recursion : Regular Grammar
  - General Recursion : Context-free Grammar

CFG = (N, T, P, S)

- N : Finite set of variables/non-terminals
- T : Alphabet/Finite set of terminals
- P : Finite set of rules/productions
- S : Start symbol

$$S \in N$$

$$N \cap T = \phi$$

$$\text{Rule: } A \to \omega$$

$$A \in N \quad \omega \in (N \cup T)^*$$

- a^* represents a context-free language because we can write a CFG for it.

$$(\{S\}, \{a\}, \{S \to \varepsilon, S \to aS\}, S)$$

- *Context-freeness:* An A-rule can be applied whenever A occurs in a string, irrespective of the context (that is, non-terminals and terminals around A).
  - *Cf.* context-sensitive grammar (“declare-use”)

Examples of CFGs

- A fragment of Cool:

  EXPR → if EXPR then EXPR else EXPR fi
  | while EXPR loop EXPR pool
  | id

  -Non-terminals are written in upper-case.

  -Terminals are in lower-case.

  -The start symbol is the left-hand side of the first production.

- Balanced Parenthesis Grammar

  S → (S) S | ε

- Simple arithmetic expressions:

  E → E ∗ E
  | E + E
  | (E)
  | id

Example: Hierarchical Scope

```
Procedure foo(integer m, integer n, integer j) {
      for i = 1 to n do {
            if (i == j) {
                  j = j + 1;
                  m = i*j;
            }
            for k = i to n {
                  m = m + k;
            }
      }
}
```

- Balanced Parentheses Problem
  - Example: {{} {{{} {{}}}}}

From CFG to Language

- One-step Derivation

$$uAv \underset{A \to \omega}{\Rightarrow} u\omega v$$

- w is derivable from v in CFG, if there is a finite sequence of rule applications such that:

$$v \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n = w$$

Let G=(N, T, P, S) be a CFG.

- $w \in (N \cup T)^*$ is a *sentential form*, if $S \Rightarrow_G^* w$.
- $w \in T^*$ is a *sentence*, if $S \Rightarrow_G^* w$.
- The *language of G*,

  $L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$

$$G = \{\{S\}, \{a, b\}, \{S \to \varepsilon, S \to aSb\}, S\}$$
$$L(G) = \{a^n b^n \mid n \geq 0\}$$

Cool Example

- Some "terminal strings" in the Cool language.

id
if id then id else id fi
while id loop id pool
if while id loop id pool then id else id
if if id then id else id fi then id else id fi

Notes on Parser

- Parser checks the membership in a language + constructs a parse tree for the input.
- Parser must handle errors gracefully.
- Parser generators implement CFG's (e.g., `bison`).
- Form of the grammar is important.
  - Many grammars generate the same language.
  - Tools are sensitive to the grammar.

Derivations and Parse Trees

- A *derivation* is a sequence of production applications.
- A derivation can be drawn as a *tree*
  - Start symbol is the tree's root
  - For a production $X \to Y_1 \cdots Y_n$ add children $Y_1 \cdots Y_n$ to (parent) node X

Derivation Example

- Grammar

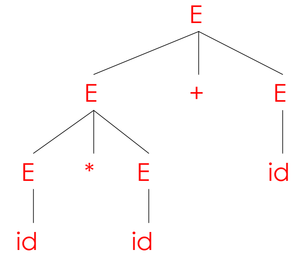
$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation

$$\begin{aligned} & E \\ \rightarrow\ & E+E \\ \rightarrow\ & E * E+E \\ \rightarrow\ & id * E + E \\ \rightarrow\ & id * id + E \\ \rightarrow\ & id * id + id \end{aligned}$$

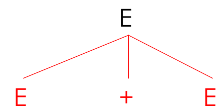


Derivation in Detail (1)

E

$$E$$

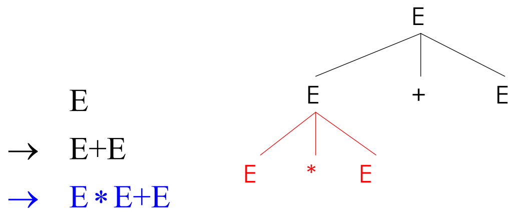
Derivation in Detail (2)



$$\begin{aligned} & E \\ \rightarrow\ & E+E \end{aligned}$$

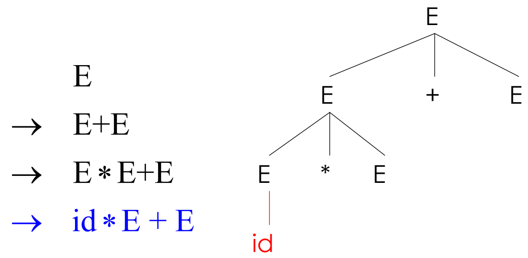
Derivation in Detail (3)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E
\end{aligned}
$$



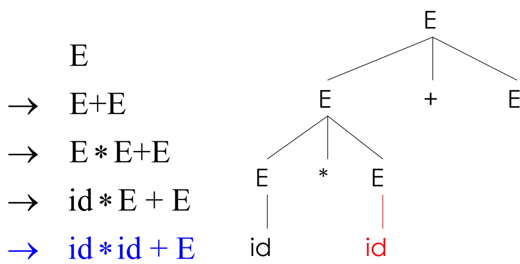
Derivation in Detail (4)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E
\end{aligned}
$$



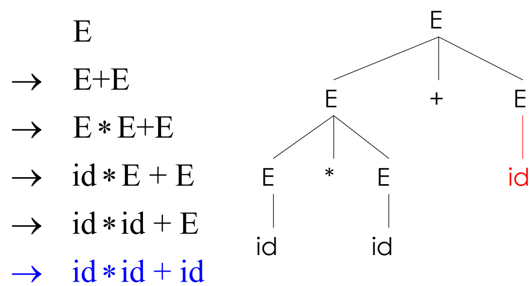
Derivation in Detail (5)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E
\end{aligned}
$$



Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Notes on Derivations

- A parse tree has
  - Terminals at the leaves.
  - Non-terminals at the interior nodes.
- An *in-order* traversal of the leaves is the original input.
- The parse tree shows the association of operations, the input string does not.

Left-most and Right-most Derivations

- The previous example is a *left-most* derivation.
  - At each step, replace the left-most non-terminal.
- There is an equivalent notion of a *right-most* derivation.

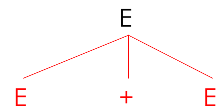
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$

Right-most Derivation in Detail (1)

E

E

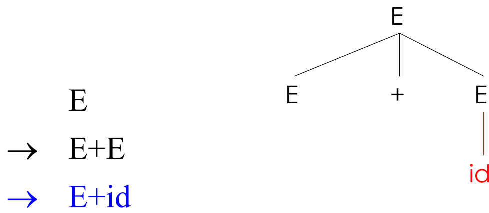
Right-most Derivation in Detail (2)



$$
\begin{aligned}
& E \\
\rightarrow\ & E+E
\end{aligned}
$$

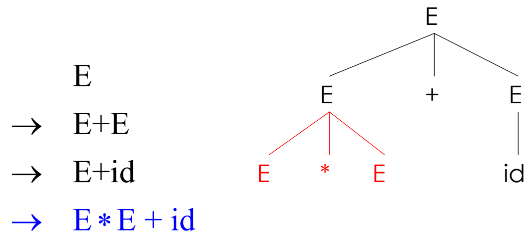
Right-most Derivation in Detail (3)

E
$\rightarrow E+E$
$\rightarrow E+id$



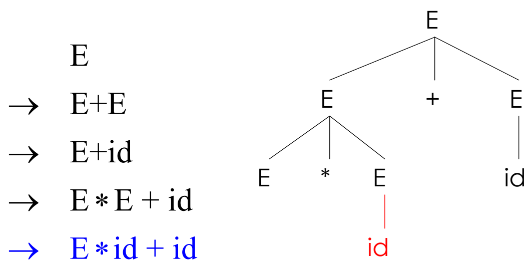
Right-most Derivation in Detail (4)

E
$\rightarrow E+E$
$\rightarrow E+id$
$\rightarrow E * E + id$



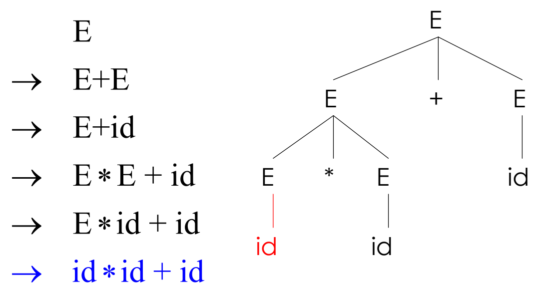
Right-most Derivation in Detail (5)

E
$\rightarrow E+E$
$\rightarrow E+id$
$\rightarrow E * E + id$
$\rightarrow E * id + id$



Right-most Derivation in Detail (6)

E
$\rightarrow E+E$
$\rightarrow E+id$
$\rightarrow E * E + id$
$\rightarrow E * id + id$
$\rightarrow id * id + id$

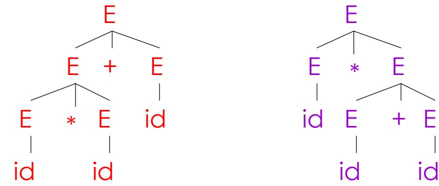


Summary of Derivations

- Note that right-most and left-most derivations have the *same* parse tree; the difference is the order in which the branches are added.
- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for *s*.
- A derivation defines a parse tree, but one parse tree may have many associated derivations.
- Left-most and right-most derivations are important in parser implementation.

Ambiguity

This string has two parse trees.



- A grammar is *ambiguous* if it has more than one parse tree for a string.
- Ambiguity is **BAD** because it leaves meaning of some programs ill-defined.
- Ambiguity can be avoided by rewriting the grammar.

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

  - Interprets non-fully parenthesized expression, giving precedence to * over +.