Grammar

Dr. Gerald Baumgartner *

1 Grammar

$X \rightarrow a\ Y\ b\ Z\ c$

2 Parser

```
Node* parseX(void) {
   ch = getNextToken();
   if(ch != 'a') error("...");
      y = parseY();
   ch = getNextToken();
   if(ch != 'b') error("...");
      z = parseZ();
   ch = getNextToken();
   if(ch != 'c') error("...");
      return new X(y,z);
}
```

*Typeset by Michael Miceli

3 Context Free Grammars

Definition:

- terminal symbols (tokens)
  id, num, -, (,), +, *, /
- non-terminal symbols
  exp, op
- start symbol
  exp
- rules of the form
  N $\rightarrow$ X...X where
  - N non-terminal
  - X (non-)terminal

4 Context Free Grammars

Example:

exp → id
| num
| -exp
| (exp)
| exp op exp

op → +
| -
| *
| /

5 Derivations

5.1 Input

x * y + z

5.2 Derivations

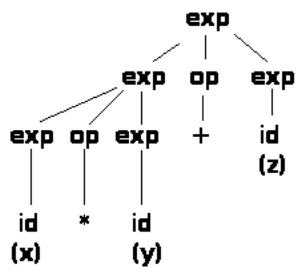
$$
\begin{aligned}
\text{exp} &\rightarrow \text{exp op } \underline{\text{exp}} \\
&\rightarrow \text{exp } \underline{\text{op}} \text{ id} \\
&\rightarrow \underline{\text{exp}} + \text{id} \\
&\rightarrow \text{exp op } \underline{\text{exp}} + \text{id} \\
&\rightarrow \text{exp } \underline{\text{op}} \text{ id} + \text{id} \\
&\rightarrow \underline{\text{exp}} * \text{id} + \text{id} \\
&\rightarrow \text{id} * \text{id} + \text{id}
\end{aligned}
$$

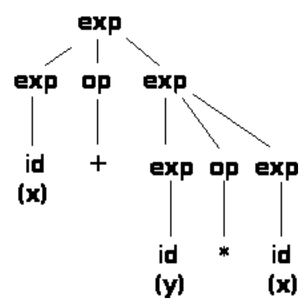
left-most derivations
= top-down parsing
(recursive-descent, LL(1))

right-most derivations
= bottom-up parsing
(yacc, bison, etc, LR(1), LALR(1))

6 Parse Trees

x * y + z





7 Grammar with Precedence and Associativity

exp → term
| exp add-op term

term → factor
| term mult-op factor

factor → id
| num
| -factor
| (exp)

add-op → + | -
add-op → * | /

8 Styles of Grammars

8.1 Ambiguous:

exp $\rightarrow$ exp op exp

- allows multiple parses (trees)

8.2 Left-recursive:

exp $\rightarrow$ exp op id

- does not work with recursive descent parsers

8.3 Right-recursive:

exp $\rightarrow$ id op exp

9 Right-recursive Grammar

exp → term erest

erest →
| add-op exp

term → factor trest

trest →
| mult-op term

factor → id
| num
| -factor
| (exp)

add-op → + | -
add-op → * | /

10 Lookahead

Must read ahead for making parse decisions

Typical lookahead:

- 0 or 1 token

For our parser:

- 0 tokens

11 Recursive-descent Parsing with Lookahead

Every parse function assumes first token was read already

```
Node* parseFactor(void) {
   if(tok->getType() == ID) {
      tok = getNextToken();
      return new Ident();
   }
   else if(tok->getType() == LPAREN) {
      tok = getNextToken();
      t = parseExp();
      tok = getNextToken();
      if(tok->getType() != RPAREN)
         error("...");
      tok = getNextToken();
      return t;
   }
}
```

12 Recursive-descent Parsing without Lookahead

```
Node* parseFactor(void) {
   tok = getNextToken();
   if(...) {
      .
      .
      .
   }
   ...
}
```

- Sometimes we need lookahead
- Might require putting token back before calling parse function