Syntax and Parsing

Textbook, Sections 2.1, 2.2.1-2.2.2, 2.3.1

1

Grammar vs. Parsing Code

Grammar:

X → a Y b Z c

Parser:

```
Node* parseX() {
  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);
}
```

2

Context-Free Grammars

Definition:

- **Terminal symbols (tokens)**
  `id, num, -, (, ), +, *, /`
- **Non-terminal symbols**
  `exp, op`
- **Start symbol**
  `exp`
- **Rules**
  `N -> X ...`
  where `N` is a non-terminal and `X`'s are (non-) terminals

3

Context-Free Grammar Example

```
exp -> id
     |  num
     |  - exp
     |  ( exp )
     |  exp op exp
op  -> +
     |  -
     |  *
     |  /
```

4

Derivations

- **Input**
  ```
  x * y + z
  ```
- **Derivation**
  ```
  exp  => exp op exp
       => exp op id
       => exp + id
       => exp op exp + id
       => exp op id + id
       => exp * id + id
       => id * id + id
  ```

5

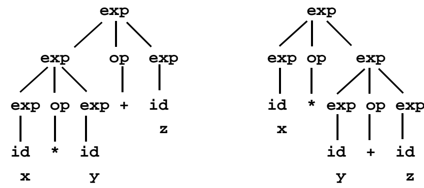
Derivations (cont'd)

- **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

Precedence and Associativity

```
exp     -> term
        |  exp add-op term

term    -> factor
        |  term mul-op factor

factor -> id
        |  num
        |  - factor
        |  ( exp )

add-op -> + | -
mul-op -> * | /
```

8

Styles of Grammar

- **Ambiguous**
  `exp -> exp op exp`
  Allows multiple parses (trees)
- **Left-recursive**
  `exp -> exp op id`
  Does not work with top-down parsers
- **Right-recursive**
  `exp -> id op exp`

9

Right-Recursive Grammar

```
exp    -> term erest
erest  ->
        |  add-op exp
term   -> factor trest
trest  ->
        |  mul-op term
factor -> id
        |  num
        |  - factor
        |  ( exp )
add-op -> + | -
mul-op -> * | /
```

10

Lookahead

- **Reading ahead for making parse decisions**
- **Typical lookahead**
  0 or 1 tokens
- **For our parser**
  0 tokens

11

Recursive-Descent Parsing with Lookahead

- **Every parse function assumes first token was already read**

```
Node* parseFactor() {
  if (tok->getType() == ID) {
    tok = getNextToken();
    return new Ident();
  }
  else if (tok->getType() == LPAREN) {
    tok = getNextToken();
    t = parseExp();
    // getNextToken() not needed here
    if (tok->getType() != RPAREN)
      error("...");
    tok = getNextToken();
    return t;
  }
}
```

12

Recursive-Descent Parsing without Lookahead

- Every parse function needs to read token first

```
Node* parseFactor() {
  tok = getNextToken();
  if (tok->getType() == ID)
    ...
}
```

- Since Scheme is interactive, we need zero lookahead for top-level parse function
- Lookahead is needed inside expressions
- Requires selective use of lookahead tokens in parser

13