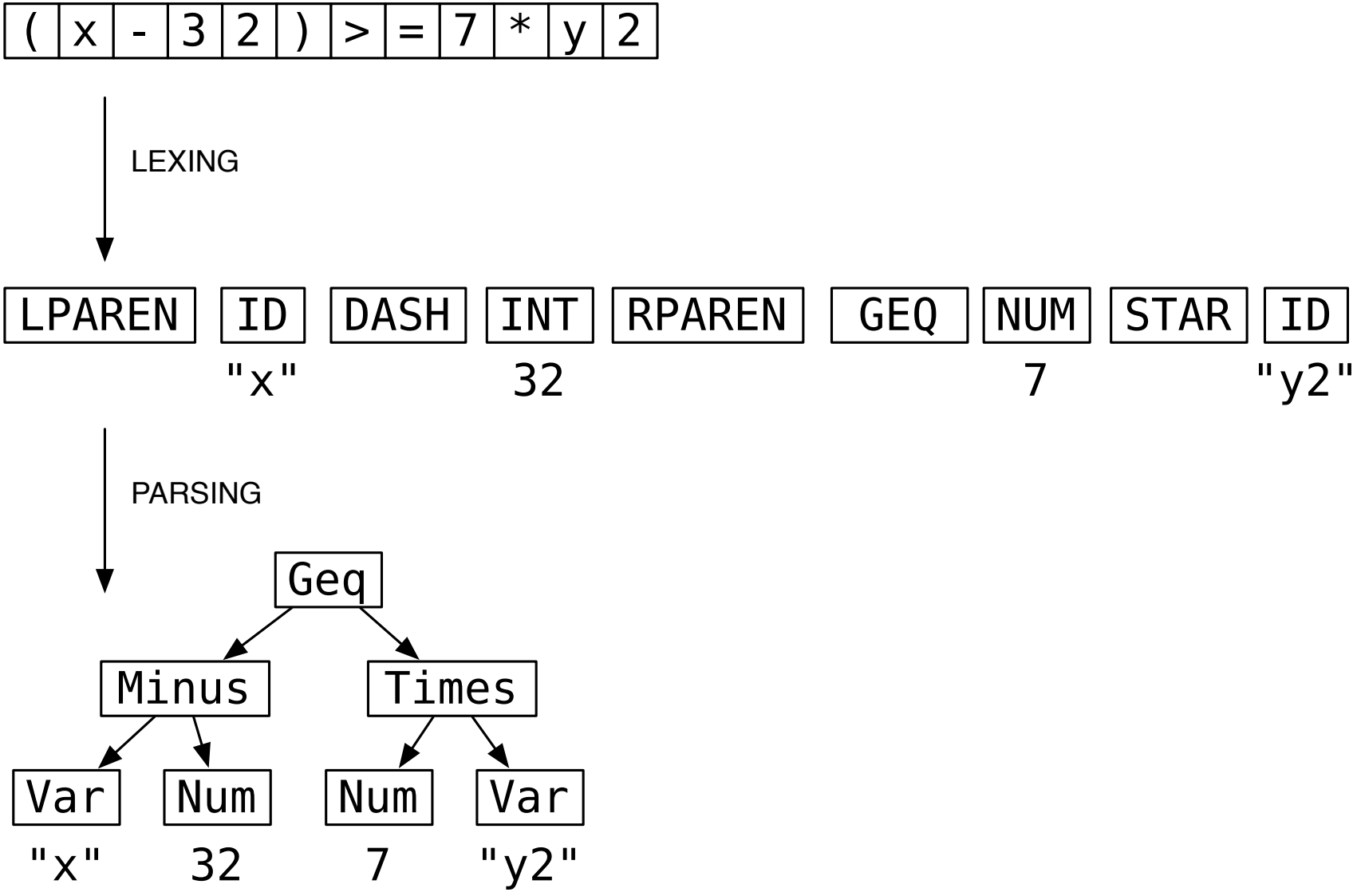
Grammars and Recursive Descent Parsing

March 28–29, 2012

CS 60: Principles of Computer Science

Assignment 8 (Full Unicalc) due Monday, April 2.

Traditional Lexing and Parsing



Specifying Syntax via CFGs

A *context-free grammar* is a set of rules for producing a set of strings (a *language*).

$$\begin{aligned} S &\to V + S \mid V \\ V &\to 0 \mid 1 \mid 2 \mid \cdots \mid 9 \end{aligned}$$

Ingredients:

- ✓ Nonterminals: S, V
- ✓ Terminals: $+$, 0, 1, 2, …, 9
- ✓ Production rules: (*see above*)
- ✓ Where to start: S

Parse Trees

The *parse tree* of a string makes explicit how a string was produced:

- ✓ Root is the start symbol
- ✓ When we apply a rule, items on the right-hand-side become children

Parse trees for **4** and 4+5?

$$
\begin{aligned}
S &\rightarrow V + S \mid V \\
V &\rightarrow 0 \mid 1 \mid 2 \mid \cdots \mid 9
\end{aligned}
$$

Exercise

$$
\begin{aligned}
P &\rightarrow S * P \mid S / P \mid V \\
S &\rightarrow V + S \mid V - S \mid V \\
V &\rightarrow 0 \mid 1 \mid 2 \mid \cdots \mid 9
\end{aligned}
$$

1. Draw the parse tree for

   `7 * 3 + 9 / 2`

2. How could we change the grammar to get a parse tree with the *usual* mathematical meaning of this expression?
3. What could we add to permit parentheses (and) to group sub-expressions?

Name: "Quiz"

Here is the *grammar* that you will use for the Homework 8.

```
S -> def V L | L
L -> # E | E
E -> P + E | P - E | P
P -> K * P | K / P | K
K -> - K | Q
Q -> R | R Q
R -> V | V ^ J
V -> D | D ~ D | W | ( L )
J -> I | - I

I -> (any nonnegative integer)
D -> (any nonnegative number)
W -> (any word that is a unit name)
```

Draw the *parse* tree for the input

```
def accel 2 meter / second^2
```

Ambiguous Grammars

A *grammar* is *ambiguous* if there are strings that have more than one possible parse tree.

$$
\begin{aligned}
E &\to E + E \mid E - E \\
&\mid E * E \mid E / E \\
&\mid (E) \mid 0 \mid \cdots \mid 9
\end{aligned}
$$

$$
\begin{aligned}
E &\to E + T \mid E - T \mid T \\
T &\to T * F \mid T / F \mid F \\
F &\to (E) \mid 0 \mid \cdots \mid 9
\end{aligned}
$$

Abstract Syntax Trees

- ✓ Ambiguous grammar have very nice parse trees
- ✓ Unambiguous grammars work better

Compromise:

- ✓ Parse input with an unambiguous grammar
- ✓ Produce "simplified" parse trees

Abstract Syntax Trees (ASTs) preserve the essential structure of parse trees, without extra crud.

Recursive Descent Parsing

The simplest algorithm for parsing (which works on some, but not all grammars) is called "recursive descent."

- ✓ We write one function for each variable/nonterminal in the grammar.
- ✓ Each function `A()` will "consumes" tokens that match the right-hand-side of some rule $A \rightarrow \cdots$

Trivial Example of Recursive Descent

```
S -> read V
   | write V
V -> x | y | z
```

```
S():
  # consume tokens matching
  # S in the grammar
  peek at the next token
  if (it's "read"):
     consume it
     V()
  else if (it's "write"):
     consume it
     V()
  else:
    report a parsing error
```

```
V():
  # consume tokens matching
  # V in the grammar
  peek at the next token
  if (it's "x"):
     consume it
  else if (it's "y"):
     consume it
  else if (it's "z"):
     consume it
  else:
    report a parsing error
```

Simple Example of Recursive Descent

```
L -> V
   | V , L
V -> x | y | z
```

```
L():
  # consume tokens matching
  # L in the grammar
  V()

  peek at the next token
  if (it's ","):
     # there's more to this L
     consume the comma
     L()
  else
     # there was only one V.
     return
```

```
V():
  # consume tokens matching
  # V in the grammar
  peek at the next token
  if (it's "x"):
     consume it
  else if (it's "y"):
     consume it
  else if (it's "z"):
     consume it
  else:
    report a parsing error
```

Exercise

```
J -> I | - I
V -> D | D ~ D | W | ( L )

I -> (any nonnegative integer)
D -> (any nonnegative number)
W -> (any word that is a unit name)
```

Sketch *pseudocode* for `J()` and `V()`.
(You may *assume* that `I()`, `D()`, and `W()` already exist.)

Minilog.java

```
// Minilog
// Author: Chris Stone
// Simple logical calculator, for inputs like
//    true or false
//    ( true and false) or false
//  ...etc...
```