Parsing

September 20, 2011
CS 42: Principles and Practice of Computer Science

Can you parse these "garden path" sentences?

The horse raced past the barn fell.

The young horse around.

The woman that whistles tunes pianos.

The man who hunts ducks out on weekends.

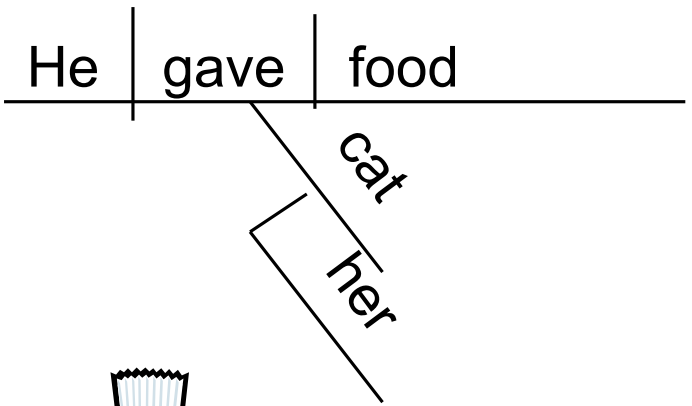
Those who admire a man that paints like Monet.

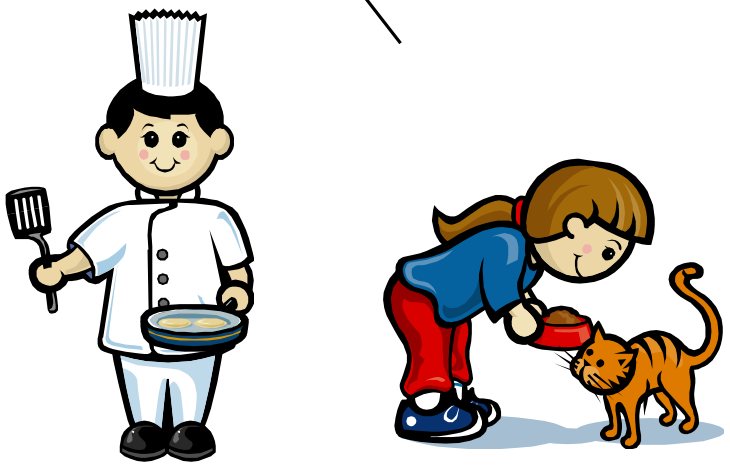
Using Structure to Clarify Meaning

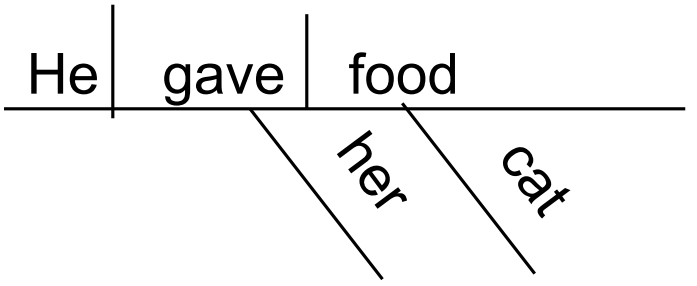
He gave her cat food.

Using Structure to Clarify Meaning

He *gave* her cat food.







Specifying Syntax via CFGs

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

Specifying Syntax via CFGs

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

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

Ingredients:

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

Show how to produce 4 starting from S.
Show how to produce 4 + 5 starting from S.
What other strings can we produce?

Last Week's "Quiz"

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

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?

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 are more practical

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.

Major Parsing Algorithms

- ✓ General Algorithms (CYK, Earley's Algorithm)
  - Handle *any* context-free grammar.
  - Slow: $O(n^3)$, where n is the length of the string being parsed.
  - Mostly used for parsing natural language, jazz standards, ...
- ✓ Bottom-Up/Shift-Reduce Parsing
  - Handles most of the grammars we care about, but not all.
  - Fast enough to be used by many compilers: $O(n)$
  - Implementation is too complex for people to write.
- ✓ Top-Down/Predictive Parsing
  - Handles many grammars we care about, after adjusting.
  - Very easy to code via "Recursive Descent"
  - Efficient: $O(n)$.

Recursive Descent

One parsing function per nonterminal.

```
;; parse-X
;;   Finds a prefix of the input that can be produced
;;   by the grammar starting with nonterminal X.
;;
;; Input : a list of tokens (symbols)
;; Output: a pair containing
;;           * the parse tree for X,
;;           * all leftover tokens.
;;         calls (error ...) if no prefix matches X
```

Recursive Descent

One parsing function per nonterminal.

```
;; parse-X
;;   Finds a prefix of the input that can be produced
;;   by the grammar starting with nonterminal X.
;;
;; Input : a list of tokens (symbols)
;; Output: a pair containing
;;           * the parse tree for X,
;;           * all leftover tokens.
;;         calls (error ...) if no prefix matches X
```

```
(parse-p '(3 * 4 + 2 * 6))   ===>  '( (* 3 4) (+ 2 * 6) )
```

A more "functional" approach would return a flag signalling success or error, rather than throwing an exception

Example

```
;; S -> P + S | P
(define (parse-s tokens)
```

Example

```
;; S -> P + S | P
(define (parse-s tokens)

  (let* ((pair1   (parse-p tokens))
         (p       (result pair1))    ;; first
         (after-p (residual pair1))) ;; second
        (cond
          ((starts-with? '+ after-p)
                (let* ((pair2 (parse-s (rest after-p)))
                       (p2       (result pair2))
                       (after-p2 (residual pair2)))
                      (list (list '+ p p2) after-p2)))
          (else  (list p after-p)))))
```

Example

```
;; V -> any number
(define (parse-v tokens)
```

Example

```
;; V -> any number
(define (parse-v tokens)

 (cond
    ((null?   tokens)
          (error "Input ended too soon"))
    ((number? (first tokens))
          (list (first tokens) (rest tokens)))
    (else (error "Unexpected token"))))
```

Name:

"Quiz:" Grammars

$$
\begin{array}{lcl}
S & \rightarrow & P + S \mid P \\
P & \rightarrow & G * P \mid G \\
G & \rightarrow & V \mid (\ S\ ) \\
V & \rightarrow & \textit{any number}
\end{array}
$$

```
(define (parse-g tokens)
```