Readings

- ✓ Why did Wilkes' final code example use self-modifying code?

Readings

- ✓ Why did Wilkes' final code example use self-modifying code?
- ✓ Any final observations (not necessarily in 42 words)?

(Top-Down) Parsing

CS 132: Compiler Design

January 26, 2011

Context Free Grammars

Grammars are used to describe concrete syntax. And several different concrete syntaxes are used for grammars!

$$
\begin{array}{rl}
S \rightarrow & \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \\
\mid & \textbf{begin}\ S\ L \\
\mid & \textbf{print}\ E
\end{array}
$$

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<expr>  ::= <digit>
          | <expr> - <expr>
          | ( <expr> - <expr> )
```

pointer:

    * *type-qualifier-list*$_{opt}$

    * *type-qualifier-list*$_{opt}$ *pointer*

Production Sequences vs. Parse Trees

```
E ::= E + E | 0 | 1 | 2 | 3 | ... | 9
```

Production Sequences vs. Parse Trees

```
E ::= E + E | 0 | 1 | 2 | 3 | ... | 9
```

E $\Rightarrow$ E + E $\Rightarrow$ 3 + E $\Rightarrow$ 3 + E + E $\Rightarrow$ 3 + 2 + E $\Rightarrow$ 3 + 2 + 1

Production Sequences vs. Parse Trees

```
E ::= E + E | 0 | 1 | 2 | 3 | ... | 9
```

$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + 2 + E \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1$

Production Sequences vs. Parse Trees

$$E ::= E + E \mid 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid 9$$

$$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + 2 + E \Rightarrow 3 + 2 + 1$$

$$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1$$

$$E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1$$

Production Sequences vs. Parse Trees

```
E ::= E + E | 0 | 1 | 2 | 3 | ... | 9
```

$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + 2 + E \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow E + 2 + 1 \Rightarrow 3 + 2 + 1$

Production Sequences vs. Parse Trees

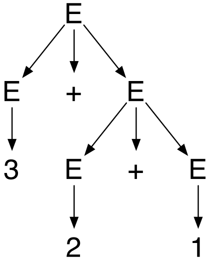
```
E ::= E + E | 0 | 1 | 2 | 3 | ... | 9
```

$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + 2 + E \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow E + 2 + 1 \Rightarrow 3 + 2 + 1$



Production Sequences vs. Parse Trees

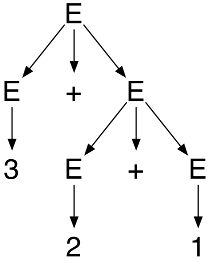
```
E ::= E + E | 0 | 1 | 2 | 3 | ... | 9
```

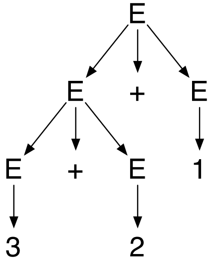
$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + 2 + E \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1$

$E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow E + 2 + 1 \Rightarrow 3 + 2 + 1$





Ambiguity

What is an *ambiguous* grammar? Which of the following grammars are ambiguous? Why do we care?

```
<exp> ::= <digit>
        | <exp> - <exp>

<exp> ::= <digit>
        | ( <exp> - <exp> )

<exp> ::= <digit>
        | <exp> - <digit>

<exp> ::= <digit>
        | <digit> - <exp>

<exp> ::= <digit>
        | <exp> <exp> -
```

Encoding Precedence

```
<exp> ::= <term>
        | <exp> + <term>
        | <exp> - <term>

<term> ::= <factor>
         | <factor> * <term>
         | <factor> / <term>

<factor> ::= <digit>
           | ( <expr> )
```

Practical Issues

- ✓ What might we return from the parser?
  - Parse Tree
  - Abstract Syntax Tree (AST)
  - Value (of a parsed expression)
  - Code (for a parsed program)
  - ...
- ✓ Typical Performance Constraints:
  - $O(n)$ time for n tokens. (Compare with $O(n^3)$.)
  - Single (left-to-right) pass through the tokens.

Most Popular Approaches

Without loss of generality, assume we are building the parse tree.

We can build this tree:

- ✓ Top-Down: Start at the root, and work down (depth-first)
- ✓ Bottom-Up: Start with the leaves, and repeatedly join small trees together to make bigger trees.

Naïve Top-Down: Backtracking search

Try all possible ways of parsing.

Naïve Top-Down: Backtracking search

Try all possible ways of parsing.

This could often be made to work. But

1. it's often inefficient
2. it's trickier to implement than one might think

Bogus Backtracking

```
S -> Aa | Ba
A -> a | c | ac
B -> Bb | b
```

```
Consume_S():
    try Consume_A(), then consume a
    if fails, try Consume_B(), then consume a
```

Bogus Backtracking

```
S -> Aa | Ba
A -> a | c | ac
B -> Bb | b
```

```
Consume_S():
    try Consume_A(), then consume a
    if fails, try Consume_B(), then consume a

Consume_A():
    try consume a
    if fails, try consume c
    if fails, try consume a then consume c
```

[What's wrong?]

Bogus Backtracking

```
S -> Aa | Ba
A -> a | c | ac
B -> Bb | b
```

```
Consume_S():
    try Consume_A(), then consume a
    if fails, try Consume_B(), then consume a

Consume_A():
    try consume a
    if fails, try consume c
    if fails, try consume a then consume c
```

[What's wrong?]

```
Consume_B():
    try Consume_B(), then consume b
    if fails, try consume b
```

[What's wrong?]

$LL(k)$ Grammars

```
S -> Aa | Ba
```

If each `Consume` function always "knew" which right-hand-side was correct, **we would never need to backtrack, or get tangled in infinite loops**.

$LL(k)$ Grammars

```
S -> Aa | Ba
```

If each `Consume` function always "knew" which right-hand-side was correct, **we would never need to backtrack, or get tangled in infinite loops**.

Then

- ✓ It would be easy to write correct `Consume` functions
- ✓ Our parser would run in linear time.

$LL(k)$ Grammars

```
S -> Aa | Ba
```

If each `Consume` function always "knew" which right-hand-side was correct, **we would never need to backtrack, or get tangled in infinite loops**.

Then

- ✓ It would be easy to write correct `Consume` functions
- ✓ Our parser would run in linear time.

We say that a grammar is $LL(k)$ if, by "peeking ahead" no more than k tokens, we can guarantee a decision that is

1. correct
2. unique

When will Top-Down Parsing Work?

Are these grammars $LL(k)$ for some k?

```
S ::= E $
E ::= n
    | plus E E
    | times E E
```

```
S ::= A
    | B
A ::= a
    | x A
B ::= b
    | y B
```

```
S ::= A
    | B
A ::= a
    | x A
B ::= b
    | x B
```

```
S ::= E $
E ::= n
    | n + E
```

```
S ::= E $
E ::= n
    | E + n
```

```
S ::= E $
E ::= E + E
    | E * E
    | n
```

Recursive Descent: Grammar as a Recursive Program

Grammar

```
S  ->  B  $
B  ->  d  |  c  B  B
```

with input `c  c  d  d  d  $`.

Making Predictions (1)

Define

$$\texttt{FIRST}(\alpha) := \{\, t \in \Sigma \mid \alpha \to^* t\beta \,\}$$

Making Predictions (1)

Define

$$\texttt{FIRST}(\alpha) := \{\, t \in \Sigma \mid \alpha \to^* t\beta \,\}$$

Consider a grammar

$$\begin{array}{lll} \texttt{A} & \texttt{::=} & \alpha_1 \\ & \texttt{|} & \alpha_2 \end{array}$$

When should our `consumeA` function predict α_1? α_2?

Making Predictions (2)

Suppose we've already predicted the next input will be an `E'`.

```
S  ::= E $
E  ::= T E'
E' ::= ϵ
     | + T E
T  ::= n
```

Which prediction should `consumeE'` do if the next token is

```
+
$
n
```

Recursive Descent Schema for $LL(1)$ Grammars

Given `A -> a1 | a2 | ... | an`*, the corresponding code is*

```
consumeA() =
   if (first_token is in FIRST(a1)) or
      (NULLABLE(a1) and (first_token is in FOLLOW(A)) then
          ...match a1 against input stream...
   else if (first_token is in FIRST(a2)) or
      (NULLABLE(a2) and (first_token is in FOLLOW(A)) then
          ...match a2 against input stream...
   ...
   else if (first token is in FIRST(a_n)) or
      (NULLABLE(an) and (first_token is in FOLLOW(A)) then
          ...match an against input stream...
   else error()
```

where

$$
\begin{array}{lcl}
\texttt{NULLABLE}(\alpha) & := & \alpha \rightarrow^* \epsilon \\
\texttt{FIRST}(\alpha) & := & \{\, t \in \Sigma \,|\, \alpha \rightarrow^* t\beta \,\} \\
\texttt{FOLLOW}(X) & := & \{\, t \in \Sigma \,|\, \texttt{S} \rightarrow^* \beta_1 X t \beta_2 \,\}
\end{array}
$$

$LL(k)$

A grammar is said to be $LL(1)$ if that generic recursive descent parser will work for all inputs.

$LL(k)$

A grammar is said to be $LL(1)$ if that generic recursive descent parser will work for all inputs.

A grammar is said to be $LL(k)$ if it works given the chance to peek at the next k tokens of the input.

$LL(k)$

A grammar is said to be $LL(1)$ if that generic recursive descent parser will work for all inputs.

A grammar is said to be $LL(k)$ if it works given the chance to peek at the next k tokens of the input.

A language is said to be $LL(k)$ if it has an $LL(k)$ grammar

$LL(k)$

A grammar is said to be $LL(1)$ if that generic recursive descent parser will work for all inputs.

A grammar is said to be $LL(k)$ if it works given the chance to peek at the next k tokens of the input.

A language is said to be $LL(k)$ if it has an $LL(k)$ grammar
(which generates the right strings, but not necessarily the right parse trees)

Massaging Grammars

```
S -> E $
E -> n
   | E - n
```

$\Downarrow$

Massaging Grammars

```
S -> E $
E -> n
   | E - n
```

$\Downarrow$

Eliminate "left recursion"

$\Downarrow$

Massaging Grammars

```
S -> E $
E -> n
   | E - n
```

⇓

Eliminate "left recursion"

⇓

```
S -> E $
E -> n
   | n - E
```

Massaging Grammars

```
S -> E $
E -> n
   | n - E
```

$\Downarrow$

Massaging Grammars

```
S -> E $
E -> n
   | n - E
```

$\Downarrow$

Left-factor

$\Downarrow$

```
S  -> E $
E  -> n E'
E' ->
    | n - E
```

Working out the Gory Details

Compute **NULLABLE**, **FIRST**, and **FOLLOW** for each nonterminal in the following grammar:

```
S -> Z $
Z -> d | X Y W Z
Y -> ϵ | c
X -> W W | a
W -> Y | w
```