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!

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \\
\mid \text{begin } S \ L \\
\mid \text{print } E
\]

\[
\text{digit} ::= \text{0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
\]

\[
\text{expr} ::= \text{digit} \\
\mid \text{expr} - \text{expr} \\
\mid ( \text{expr} - \text{expr} )
\]

\*

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

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

\[ E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + 2 + E \Rightarrow 3 + 2 + 1 \]
**Production Sequences vs. Parse Trees**

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

\[
E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + 2 + E \Rightarrow 3 + 2 + 1
\]

\[
E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1
\]
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

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

- \(E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + 2 + E \Rightarrow 3 + 2 + 1\)
- \(E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1\)
- \(E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1\)
- \(E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow E + 2 + 1 \Rightarrow 3 + 2 + 1\)
Production Sequences vs. Parse Trees

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

\[
E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + 2 + E \Rightarrow 3 + 2 + 1
\]
\[
E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E + E \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1
\]
\[
E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow 3 + E + 1 \Rightarrow 3 + 2 + 1
\]
\[
E \Rightarrow E + E \Rightarrow E + 1 \Rightarrow E + E + 1 \Rightarrow E + 2 + 1 \Rightarrow 3 + 2 + 1
\]
**Production Sequences vs. Parse Trees**

\[
E ::= \text{E} + \text{E} \mid 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid 9
\]

\[
\begin{align*}
E & \Rightarrow \text{E} + \text{E} \Rightarrow 3 + \text{E} \\
& \Rightarrow 3 + \text{E} + \text{E} \Rightarrow 3 + 2 + \text{E} \Rightarrow 3 + 2 + 1
\end{align*}
\]

\[
\begin{align*}
E & \Rightarrow \text{E} + \text{E} \Rightarrow 3 + \text{E} \Rightarrow 3 + \text{E} + \text{E} \Rightarrow 3 + \text{E} + 1 \Rightarrow 3 + 2 + 1
\end{align*}
\]

\[
\begin{align*}
E & \Rightarrow \text{E} + \text{E} \Rightarrow \text{E} + 1 \Rightarrow \text{E} + \text{E} + 1 \Rightarrow 3 + \text{E} + 1 \Rightarrow 3 + 2 + 1
\end{align*}
\]

\[
\begin{align*}
E & \Rightarrow \text{E} + \text{E} \Rightarrow \text{E} + 1 \Rightarrow \text{E} + \text{E} + 1 \Rightarrow \text{E} + 2 + 1 \Rightarrow 3 + 2 + 1
\end{align*}
\]

![Parse Tree Diagram](image-url)
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.
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**

\[
\begin{align*}
S & \rightarrow Aa \mid Ba \\
A & \rightarrow a \mid c \mid ac \\
B & \rightarrow Bb \mid b
\end{align*}
\]

**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

$$S \rightarrow Aa \mid Ba$$

$$A \rightarrow a \mid c \mid ac$$

$$B \rightarrow Bb \mid b$$

**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**

\[
S \rightarrow Aa \mid Ba
\]

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**

\[ S \rightarrow Aa \mid Ba \]

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 $k$?

$S ::= E \, \$ \\
E ::= n \\
    | \text{plus} \, E \, E \\
    | \text{times} \, E \, E$

$S ::= A \\
A ::= a \\
    | x \, A$

$B ::= b \\
    | y \, B$

$S ::= A \\
    | B$

$S ::= B \\
B ::= b \\
    | x \, B$

$S ::= E \, \$ \\
$E ::= n$ \\
$E ::= E + n$

$S ::= E \, \$ \\
$E ::= n + E$ \\
$E ::= E + n$
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

\[ \text{FIRST}(\alpha) := \left\{ t \in \Sigma \mid \alpha \rightarrow^* t\beta \right\} \]
Making Predictions (1)

Define

\[ \text{FIRST}(\alpha) := \{ \, t \in \Sigma \mid \alpha \rightarrow^* t\beta \, \} \]

Consider a grammar

\[
A ::= \alpha_1 \\
| \quad \alpha_2
\]

When should our \texttt{consumeA} function predict \(\alpha_1\)? \(\alpha_2\)?
Making Predictions (2)

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

\[
\begin{align*}
  S & ::= E \, \$ \\
  E & ::= T \, E' \\
  E' & ::= \epsilon \\
       & \mid + \, T \, E \\
  T & ::= n
\end{align*}
\]

Which prediction should consume E' do if the next token is

+ 
$ 
n
**Recursive Descent Schema for LL(1) Grammars**

Given $A \rightarrow a_1 \mid a_2 \mid \ldots \mid a_n$, the corresponding code is

```plaintext
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

```plaintext
NULLABLE(\alpha) := \alpha \rightarrow^* \epsilon
FIRST(\alpha) := \{ t \in \Sigma \mid \alpha \rightarrow^* t\beta \}
FOLLOW(X) := \{ t \in \Sigma \mid S \rightarrow^* \beta_1 X t \beta_2 \}
```
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 $LL(1)$ 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 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.
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
↓
Massaging Grammars

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

\(\downarrow\)
Eliminate “left recursion”
\(\downarrow\)
Massaging Grammars

\[
\begin{align*}
  S & \rightarrow E \ \$ \\
  E & \rightarrow n \\
    & \quad \mid E - n \\
  \downarrow \\
  \text{Eliminate “left recursion”} \\
  \downarrow \\
  S & \rightarrow E \ \$ \\
  E & \rightarrow n \\
    & \quad \mid n - E
\end{align*}
\]
Massaging Grammars

\[ S \rightarrow E \, \$ \]
\[ E \rightarrow n \]
\[ \text{| } n \rightarrow E \]

\[ \downarrow \]
Massaging Grammars

\[
\begin{align*}
S & \rightarrow E \ \$ \\
E & \rightarrow n \\
& \quad | \ n - E \\
\Downarrow & \\
& \quad \text{Left-factor} \\
\Downarrow & \\
S & \rightarrow E \ \$ \\
E & \rightarrow n \ E' \\
E' & \rightarrow \\
& \quad | \ n - E
\end{align*}
\]
Working out the Gory Details

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

\[
S \rightarrow Z \$
\]
\[
Z \rightarrow d \mid X Y W Z
\]
\[
Y \rightarrow \epsilon \mid c
\]
\[
X \rightarrow W W \mid a
\]
\[
W \rightarrow Y \mid w
\]