Parsing IV
Bottom-up Parsing

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved. Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Parsing Techniques

Top-down parsers \((LL(1), \text{recursive descent})\)

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad “pick” \(\Rightarrow\) may need to backtrack
- Some grammars are backtrack-free \((\text{predictive parsing})\)

Bottom-up parsers \((LR(1), \text{operator precedence})\)

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars
Bottom-up Parsing (definitions)

The point of parsing is to construct a derivation

A derivation consists of a series of rewrite steps

\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow ... \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence} \]

• Each \( \gamma_i \) is a sentential form
  → If \( \gamma \) contains only terminal symbols, \( \gamma \) is a sentence in \( L(G) \)
  → If \( \gamma \) contains \( \geq 1 \) non-terminals, \( \gamma \) is a sentential form

• To get \( \gamma_i \) from \( \gamma_{i-1} \), expand some NT \( A \in \gamma_{i-1} \) by using \( A \rightarrow \beta \)
  → Replace the occurrence of \( A \in \gamma_{i-1} \) with \( \beta \) to get \( \gamma_i \)
  → In a leftmost derivation, it would be the first NT \( A \in \gamma_{i-1} \)

A left-sentential form occurs in a leftmost derivation
A right-sentential form occurs in a rightmost derivation
Bottom-up Parsing

A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol $S$

$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence}$

To reduce $\gamma_i$ to $\gamma_{i-1}$ match some rhs $\beta$ against $\gamma_i$ then replace $\beta$ with its corresponding lhs, $A$.  (assuming the production $A \rightarrow \beta$)

In terms of the parse tree, this is working from leaves to root

- Nodes with no parent in a partial tree form its upper fringe
- Since each replacement of $\beta$ with $A$ shrinks the upper fringe, we call it a reduction.

The parse tree need not be built, it can be simulated

$|\text{parse tree nodes}| = |\text{words}| + |\text{reductions}|$
Finding Reductions

Consider the simple grammar

$1$  \hspace{1cm} Goal $\rightarrow$ a A B e

$2$  \hspace{1cm} A $\rightarrow$ A b c

$3$  \hspace{1cm} | b

$4$  \hspace{1cm} B $\rightarrow$ d

And the input string abbcde

The trick is scanning the input and finding the next reduction.

The mechanism for doing this must be efficient.
Finding Reductions

(Handles)

The parser must find a substring $\beta$ of the tree’s frontier that matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation ($\Rightarrow \beta \rightarrow A$ is in RRD).

Informally, we call this substring $\beta$ a **handle**

Formally,

A **handle** of a right-sentential form $\gamma$ is a pair $<A \rightarrow \beta, k>$ where $A \rightarrow \beta \in P$ and $k$ is the position in $\gamma$ of $\beta$’s rightmost symbol.

If $<A \rightarrow \beta, k>$ is a handle, then replacing $\beta$ at $k$ with $A$ produces the right sentential form from which $\gamma$ is derived in the rightmost derivation.

Because $\gamma$ is a right-sentential form, the substring to the right of a handle contains only terminal symbols

$\Rightarrow$ the parser doesn’t need to scan past the handle (very far)
Finding Reductions (Handles)

Critical Insight (Theorem?)

If $G$ is unambiguous, then every right-sentential form has a unique handle.

If we can find those handles, we can build a derivation!

Sketch of Proof:

1. $G$ is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \rightarrow \beta$ applied to derive $\gamma_i$ from $\gamma_{i-1}$
3. $\Rightarrow$ a unique position $k$ at which $A \rightarrow \beta$ is applied
4. $\Rightarrow$ a unique handle $<A \rightarrow \beta, k>$

This all follows from the definitions
THE EXAMPLE

(a very busy slide)

The expression grammar Handles for rightmost derivation of \( x = 2 \ast y \)

This is the inverse of Figure 3.9 in EaC
Handle-pruning, Bottom-up Parsers

The process of discovering a handle & reducing it to the appropriate left-hand side is called handle pruning.

Handle pruning forms the basis for a bottom-up parsing method.

To construct a rightmost derivation:

\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w \]

Apply the following simple algorithm:

\begin{verbatim}
for i ← n to 1 by -1
    Find the handle \(<A_i \rightarrow \beta_i, k_i>\) in \(\gamma_i\)
    Replace \(\beta_i\) with \(A_i\) to generate \(\gamma_{i-1}\)
\end{verbatim}

This takes \(2n\) steps.
Handle-pruning, Bottom-up Parsers

One implementation technique is the *shift-reduce parser*

```plaintext
push INVALID
token ← next_token()
repeat until (top of stack = Goal and token = EOF)
  if the top of the stack is a handle A→β
    then // reduce β to A
      pop |β| symbols off the stack
      push A onto the stack
  else if (token ≠ EOF)
    then // shift
      push token
      token ← next_token()
  else // need to shift, but out of input
    report an error
```

How do errors show up?
- failure to find a handle
- hitting EOF & needing to shift (final else clause)

Either generates an error

---

Figure 3.7 in EAC
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce
Back to $x - 2 \ast y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id = num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>id</td>
<td>= num * id</td>
<td>9,1</td>
<td>red. 9</td>
</tr>
<tr>
<td>$Factor$</td>
<td>= num * id</td>
<td>7,1</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Term$</td>
<td>= num * id</td>
<td>4,1</td>
<td>red. 4</td>
</tr>
<tr>
<td>$Expr$</td>
<td>= num * id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce
### Back to $x = 2 \times y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$$</td>
<td>id = num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$id$</td>
<td>= num * id</td>
<td>9,1</td>
<td>red. 9</td>
</tr>
<tr>
<td>$Factor$</td>
<td>= num * id</td>
<td>7,1</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Term$</td>
<td>= num * id</td>
<td>4,1</td>
<td>red. 4</td>
</tr>
<tr>
<td>$Expr$</td>
<td>= num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr =$</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr =$</td>
<td>* id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce
Back to $x \times 2 * y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>$id = num * id$</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$id$</td>
<td>$- num * id$</td>
<td>9,1</td>
<td>red. 9</td>
</tr>
<tr>
<td>$Factor$</td>
<td>$- num * id$</td>
<td>7,1</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Term$</td>
<td>$- num * id$</td>
<td>4,1</td>
<td>red. 4</td>
</tr>
<tr>
<td>$Expr$</td>
<td>$- num * id$</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr -$</td>
<td>$num * id$</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr - num$</td>
<td>$* id$</td>
<td>8,3</td>
<td>red. 8</td>
</tr>
<tr>
<td>$Expr - Factor$</td>
<td>$* id$</td>
<td>7,3</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Expr - Term$</td>
<td>$* id$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce
### Back to $x = 2 * y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id = num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$id$</td>
<td>num * id</td>
<td>9,1</td>
<td>red. 9</td>
</tr>
<tr>
<td>$Factor$</td>
<td>num * id</td>
<td>7,1</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Term$</td>
<td>num * id</td>
<td>4,1</td>
<td>red. 4</td>
</tr>
<tr>
<td>$Expr$</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr =$</td>
<td>id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Expr =$ num</td>
<td>* id</td>
<td>8,3</td>
<td>red. 8</td>
</tr>
<tr>
<td>$Expr =$ Factor</td>
<td>* id</td>
<td>7,3</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Expr =$ Term</td>
<td>* id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr =$ Term $</td>
<td>$ id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr =$ Term $*$ id</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce
1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id = num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>= num * id</td>
<td>9,1</td>
<td>red. 9</td>
</tr>
<tr>
<td>$Factor</td>
<td>= num * id</td>
<td>7,1</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Term</td>
<td>= num * id</td>
<td>4,1</td>
<td>red. 4</td>
</tr>
<tr>
<td>$Expr</td>
<td>= num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr =</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr = num</td>
<td>* id</td>
<td>8,3</td>
<td>red. 8</td>
</tr>
<tr>
<td>$Expr = Factor</td>
<td>* id</td>
<td>7,3</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Expr = Term</td>
<td>* id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr = Term *</td>
<td>id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr = Term * id</td>
<td></td>
<td>9,5</td>
<td>red. 9</td>
</tr>
<tr>
<td>$Expr = Term * Factor</td>
<td></td>
<td>5,5</td>
<td>red. 5</td>
</tr>
<tr>
<td>$Expr = Term</td>
<td></td>
<td>3,3</td>
<td>red. 3</td>
</tr>
<tr>
<td>$Expr</td>
<td></td>
<td>1,1</td>
<td>red. 1</td>
</tr>
<tr>
<td>$Goal</td>
<td>none</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

5 shifts + 9 reduces + 1 accept
## Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id = num * id</td>
<td>shift</td>
</tr>
<tr>
<td>id</td>
<td>= num * id</td>
<td>red. 9</td>
</tr>
<tr>
<td>$ Factor</td>
<td>= num * id</td>
<td>red. 7</td>
</tr>
<tr>
<td>$ Term</td>
<td>= num * id</td>
<td>red. 4</td>
</tr>
<tr>
<td>$ Expr</td>
<td>= num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr –</td>
<td>num * id</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr – num</td>
<td>num * id</td>
<td>red. 8</td>
</tr>
<tr>
<td>$ Expr – Factor</td>
<td>* id</td>
<td>red. 7</td>
</tr>
<tr>
<td>$Expr – Term</td>
<td>* id</td>
<td>shift</td>
</tr>
<tr>
<td>$Expr – Term *</td>
<td>id</td>
<td>shift</td>
</tr>
<tr>
<td>$ Expr – Term * id</td>
<td></td>
<td>red. 9</td>
</tr>
<tr>
<td>$ Expr – Term * Factor</td>
<td></td>
<td>red. 5</td>
</tr>
<tr>
<td>$ Expr – Term</td>
<td></td>
<td>red. 3</td>
</tr>
<tr>
<td>$ Expr</td>
<td></td>
<td>accept</td>
</tr>
<tr>
<td>$ Goal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diagram

```
 Goal
  /\   \
 Expr  Term
   /\   /\  \
  Expr  Term  Fact.
   /   /   /  \
  Expr  Term Fact.  <id,y>  
   /   /   \
  Term  Fact.
   /\   \
 Expr  Term
   /  \
 Fact.  Fact.
   /  \
 <id,x>  <num,2>
```

### Notes
- Stack: Symbols pushed onto the stack.
- Input: Tokens read from the input.
- Action: Type of action taken based on the grammar rules.
- Grammar Tree: Visualization of the parsing process.

### Grammar
- **Goal:**
  - Expr
  - Term

- **Expr:**
  - Factor
  - Term

- **Term:**
  - Num
  - Id

- **Factor:**
  - Num
  - Id
Shift-reduce Parsing

Shift reduce parsers are easily built and easily understood

A shift-reduce parser has just four actions

- **Shift** — next word is shifted onto the stack
- **Reduce** — right end of handle is at top of stack
  - Locate left end of handle within the stack
  - Pop handle off stack & push appropriate *lhs*
- **Accept** — stop parsing & report success
- **Error** — call an error reporting/recovery routine

Accept & Error are simple

- **Shift** is just a push and a call to the scanner
- **Reduce** takes |rhs| pops & 1 push

If handle-finding requires state, put it in the stack ⇒ 2x work
An Important Lesson about Handles

To be a handle, a substring of a sentential form $\gamma$ must have two properties:

- It must match the right hand side $\beta$ of some rule $A \rightarrow \beta$
- There must be some rightmost derivation from the goal symbol that produces the sentential form $\gamma$ with $A \rightarrow \beta$ as the last production applied

- Simply looking for right hand sides that match strings is not good enough

- **Critical Question:** How can we know when we have found a handle without generating lots of different derivations?
  
  - **Answer:** we use look ahead in the grammar along with tables produced as the result of analyzing the grammar.
  - $LR(1)$ parsers build a DFA that runs over the stack & finds them
LR(1) Parsers

• LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
• LR(1) parsers recognize languages that have an LR(1) grammar

Informal definition:
A grammar is LR(1) if, given a rightmost derivation
\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence} \]

We can
1. isolate the handle of each right-sentential form \( \gamma_i \), and
2. determine the production by which to reduce,

by scanning \( \gamma_i \) from left-to-right, going at most 1 symbol beyond the right end of the handle of \( \gamma_i \).
LR(1) Parsers

A table-driven LR(1) parser looks like

source code \rightarrow \text{Scanner} \rightarrow \text{Table-driven Parser} \rightarrow IR

grammar \rightarrow \text{Parser Generator} \rightarrow \text{ACTION & GOTO Tables}

Tables can be built by hand
However, this is a perfect task to automate
LR(1) Skeleton Parser

```java
stack.push(INVALID); stack.push(s₀);
not_found = true;
token = scanner.next_token();
do while (not_found) {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);
        stack.push(GOTO[s, A]);
    }
    else if ( ACTION[s,token] == "shift s;" ) then {
        stack.push(token); stack.push(s);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
        & token == EOF )
        then not_found = false;
        else report a syntax error and recover;
}
report success;
```

The skeleton parser

- uses ACTION & GOTO tables
- does |words| shifts
- does |derivation| reductions
- does 1 accept
- detects errors by failure of 3 other cases
LR(1) Parsers (parse tables)

To make a parser for $L(G)$, need a set of tables

The grammar

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shift 2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>shift 3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>reduce 3</td>
<td>0</td>
</tr>
</tbody>
</table>

The tables

Remember, this is the left-recursive SheepNoise; EaC shows the right-recursive version.
LR(1) Parsers

How does this LR(1) stuff work?

- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - All active handles include top of stack (TOS)
  - Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
  - Build a handle-recognizing DFA
  - ACTION & GOTO tables encode the DFA
- To match subterm, invoke subterm DFA
  & leave old DFA’s state on stack
- Final state in DFA ⇒ a reduce action
  - New state is GOTO[state at TOS (after pop), lhs]
  - For SN, this takes the DFA to $s_1$

Control DFA for SN
Building LR(1) Parsers

How do we generate the ACTION and GOTO tables?

• Use the grammar to build a model of the DFA
• Use the model to build ACTION & GOTO tables
• If construction succeeds, the grammar is LR(1)

The Big Picture

• Model the state of the parser
• Use two functions $goto(s, X)$ and $closure(s)$
  $\rightarrow goto()$ is analogous to $move()$ in the subset construction
  $\rightarrow closure()$ adds information to round out a state
• Build up the states and transition functions of the DFA
• Use this information to fill in the ACTION and GOTO tables
What can go wrong?

What if set $s$ contains $[A \to \beta \cdot ay, b]$ and $[B \to \beta \cdot a]$?

- First item generates “shift”, second generates “reduce”
- Both define $\text{ACTION}[s, a]$ — cannot do both actions
- This is a fundamental ambiguity, called a shift/reduce error
- Modify the grammar to eliminate it (if-then-else)
- Shifting will often resolve it correctly

What is set $s$ contains $[A \to \gamma \cdot a]$ and $[B \to \gamma \cdot a]$?

- Each generates “reduce”, but with a different production
- Both define $\text{ACTION}[s, a]$ — cannot do both reductions
- This fundamental ambiguity is called a reduce/reduce error
- Modify the grammar to eliminate it (PL/I’s overloading of (...))

In either case, the grammar is not $LR(1)$
Left Recursion versus Right Recursion

- **Right recursion**
  - Required for termination in top-down parsers
  - Uses (on average) more stack space
  - Produces right-associative operators
- **Left recursion**
  - Works fine in bottom-up parsers
  - Limits required stack space
  - Produces left-associative operators
- **Rule of thumb**
  - Left recursion for bottom-up parsers
  - Right recursion for top-down parsers

```
*   *
|   |
|   |
w  x  y  z
```

```
*   *
|   |
|   |
w  x  y  z
```

```
*   *
|   |
|   |
( (w * x) * y ) * z
```
Associativity

• What difference does it make?
• Can change answers in floating-point arithmetic
• Exposes a different set of common subexpressions

• Consider \( x+y+z \)

\[
\text{Ideal operator} \\
\begin{array}{c}
  + \\
  \downarrow \\
  x \quad y \quad z \\
\end{array}
\]

\[
\begin{array}{c}
  + \\
  \downarrow \\
  x \quad + \\
  \quad \downarrow \\
  y \quad z \\
\end{array}
\]

\[
\begin{array}{c}
  + \\
  \downarrow \\
  + \\
  \quad \downarrow \\
  x \quad y \\
\end{array}
\]

• What if \( y+z \) occurs elsewhere? Or \( x+y \)? or \( x+z \)?
• What if \( x = 2 \) & \( z = 17 \)? Neither left nor right exposes 19.
• Best choice is function of surrounding context
Hierarchy of Context-Free Languages

Context-free languages

Deterministic languages (LR(k))

LR(k) \equiv LR(1)

LL(k) languages

Simple precedence languages

LL(1) languages

Operator precedence languages
Hierarchy of Context-Free Grammars

- Operator precedence includes some ambiguous grammars
- LL(1) is a subset of SLR(1)

The inclusion hierarchy for context-free grammars
Shrinking the Tables

Three options:

• **Combine terminals such as number & identifier, + & -, *, /**
  → Directly removes a column, may remove a row
  → For expression grammar, 198 (vs. 384) table entries

• **Combine rows or columns**
  → Implement identical rows once & remap states
  → Requires extra indirection on each lookup
  → Use separate mapping for ACTION & for GOTO

• **Use another construction algorithm**
  → Both LALR(1) and SLR(1) produce smaller tables
  → Implementations are readily available
LR(k) versus LL(k) (Top-down Recursive Descent)

Finding Reductions

LR(k) ⇒ Each reduction in the parse is detectable with
  1. the complete left context,
  2. the reducible phrase, itself, and
  3. the k terminal symbols to its right

LL(k) ⇒ Parser must select the reduction based on
  1. The complete left context
  2. The next k terminals

Thus, LR(k) examines more context

“... in practice, programming languages do not actually seem to fall in the gap between LL(1) languages and deterministic languages” J.J. Horning, “LR Grammars and Analysers”, in Compiler Construction, An Advanced Course, Springer-Verlag, 1976
<table>
<thead>
<tr>
<th>Top-down recursive descent</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fast</td>
<td>Hand-coded</td>
</tr>
<tr>
<td></td>
<td>Good locality</td>
<td>High maintenance</td>
</tr>
<tr>
<td></td>
<td>Simplicity</td>
<td>Right associativity</td>
</tr>
<tr>
<td></td>
<td>Good error detection</td>
<td></td>
</tr>
<tr>
<td>LR(1)</td>
<td>Fast</td>
<td>Large working sets</td>
</tr>
<tr>
<td></td>
<td>Deterministic langs.</td>
<td>Poor error messages</td>
</tr>
<tr>
<td></td>
<td>Automatable</td>
<td>Large table sizes</td>
</tr>
<tr>
<td></td>
<td>Left associativity</td>
<td></td>
</tr>
</tbody>
</table>
There is a level of correctness that is deeper than grammar.

```c
fie(a,b,c,d)
    int a, b, c, d;
{ ... }
fee() {
    int f[3], g[0],
        h, i, j, k;
    char *p;
    fie(h, i, "ab", j, k);
    k = f * i + j;
    h = g[17];
    printf("<%s,%s>.
", p, q);
    p = 10;
}
```

What is wrong with this program?

*(let me count the ways ...)*
There is a level of correctness that is deeper than grammar.

What is wrong with this program?
(let me count the ways …)

• declared g[0], used g[17]
• wrong number of args to fie()
• “ab” is not an int
• wrong dimension on use of f
• undeclared variable q
• 10 is not a character string

All of these are “deeper than syntax”
Beyond Syntax

To generate code, the compiler needs to answer many questions:

- Is “x” a scalar, an array, or a function? Is “x” declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of “x” does each use reference?
- Is the expression “x * y + z” type-consistent?
- In “a[i,j,k]”, does a have three dimensions?
- Where can “z” be stored? *(register, local, global, heap, static)*
- In “f ← 15”, how should 15 be represented?
- How many arguments does “fie()” take? What about “printf ()”?
- Does “*p” reference the result of a “malloc()”?
- Do “p” & “q” refer to the same memory location?
- Is “x” defined before it is used?

*These cannot be expressed in a CFG*
Beyond Syntax

These questions are part of context-sensitive analysis

• Answers depend on values, not parts of speech
• Questions & answers involve non-local information
• Answers may involve computation

How can we answer these questions?

• Use formal methods
  → Context-sensitive grammars?
  → Attribute grammars? (attributed grammars?)

• Use ad-hoc techniques
  → Symbol tables
  → Ad-hoc code (action routines)

In scanning & parsing, formalism won; different story here.