Prelude

- What is this?
  - Micrograph of influenza A

- Where does influenza A come from?
  - Birds – aka “Avian Flu”
  - **BUT**, it’s extremely rare for humans to be infected

- So, how do people get influenza A
  - Pigs can get both avian and human flu strains
  - Virus recombines in pigs

Summary

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-down recursive descent</td>
<td></td>
</tr>
<tr>
<td>Fast</td>
<td>Hand-coded</td>
</tr>
<tr>
<td>Good locality</td>
<td>Good error</td>
</tr>
<tr>
<td>Simplicity</td>
<td>detection</td>
</tr>
<tr>
<td>Good error detection</td>
<td></td>
</tr>
<tr>
<td>Left associativity</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LR(1)</td>
<td></td>
</tr>
<tr>
<td>Fast</td>
<td>Large working sets</td>
</tr>
<tr>
<td>Deterministic</td>
<td>Poor error</td>
</tr>
<tr>
<td>langs.</td>
<td>messages</td>
</tr>
<tr>
<td>Automatable</td>
<td></td>
</tr>
<tr>
<td>Left associativity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Large table sizes</td>
</tr>
</tbody>
</table>

Overview

- Parsing
  - Tells us if input is syntactically correct
  - Gives us derivation or parse tree
  - From that, perform other computations
    - Analyze parse tree, check for errors
    - Transform into other representations

Syntax-directed translation

- In practice:
  - Fold some computations into parsing
  - Computations are triggered by parsing steps
  - Syntax-directed translation

- Project
  - Action code in `{ ; }` delimiters
  - Builds abstract syntax tree

- How much can we do during parsing?
Example

- Desk calculator
  - Expression grammar
  - Evaluate the resulting tree

```
G → E
E → E1 + T
E → T
T → T1 * F
T → F
F → (E)
F → num
```

Example

- Can we evaluate the expression without building the tree first?
  - "Piggyback" on parsing

```
G → E
E → E1 + T
E → T
T → T1 * F
T → F
F → (E)
F → num
```

Example

- Codify as a set of rules

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G → E</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>2</td>
<td>E → E1 + T</td>
<td>E.val = E1.val + T.val</td>
</tr>
<tr>
<td>3</td>
<td>E → T</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>4</td>
<td>T → T1 * F</td>
<td>T.val = T1.val * F.val</td>
</tr>
<tr>
<td>5</td>
<td>T → F</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>6</td>
<td>F → (E)</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>7</td>
<td>F → num</td>
<td>F.val = valueof(num)</td>
</tr>
</tbody>
</table>

Example derivations

For "-5"
```
Number → Sign List
    → - Bit
    → 1
```

For "-101"
```
Number → Sign List Bit
    → Sign List Bit 1
    → Sign Bit 0 1
    → 101
```

Attribute grammars

- A context-free grammar with a set of rules
  - Each symbol has a set of values, or attributes
  - Semantic rules: how to compute each attribute

**Example grammar**

- This grammar describes signed binary numbers
- We would like to augment it with rules that compute the decimal value of each valid input string

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number → Sign List</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Sign → +</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>List → List Bit</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Bit → 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Bit → 1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Bit → 0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Bit → 1</td>
<td></td>
</tr>
</tbody>
</table>

Attribute grammars

- The bad news: Attribute grammars never widely adopted
- Why study them?
  - The attribute grammar formalism is important
    - Succinctly makes many points clear
    - Sets the stage for actual, ad-hoc practice
  - The problems motivate practice
    - Non-local computation
    - Need for centralized information
Attribute grammar

- **Goal:**
  - Compute the value of the binary number
- **Information we need**
  - Position of each 1 bit – to compute value
  - Sum of bit values
- **Computation**
  - Propagate position information
  - Accumulate the sums

Attributes

Rules

### Attribution rules

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
<th>Attributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number → Sign List</td>
<td>List.pos = 0</td>
</tr>
<tr>
<td>2</td>
<td>Sign → +</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>List → List1 Bit</td>
<td>List.pos → List.pos + 1</td>
</tr>
<tr>
<td>5</td>
<td>Bit</td>
<td>Bit.val ← List.val</td>
</tr>
<tr>
<td>6</td>
<td>Bit → 0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How to compute List value?
- List.val ← List.val + Bit.val or Bit.val ← 2(Bit.pos)

How to compute List1 value?
- List1.val ← List.val + Bit.val

### Example

This is the complete attribute dependence graph for “–101”.
It shows the flow of all attribute values in the example.
A rule may use attributes in the parent, children, or siblings of a node.

Example graph showing the flow of attributes in the example.
Attribute grammars

- Components
  - Attributes
    - Associated with nodes in parse tree
  - Rules
    - Value assignments associated with productions
    - Information is entirely local: it can only refer to values in the given production
- Given a parse tree
  - Rules form a dependence graph between attributes
  - Result: a high-level, functional specification

Evaluation

- Tricky part
  - Values flowing both up and down in tree
  - How do we order the computation?
  - And, how does that relate to parsing order? (i.e., the order in which parse tree nodes are created)
- Key
  - Must obey the dependence graph
  - What other constraints?

Evaluation

- Dynamic, dependence-based methods
  - Build the parse tree
  - Build the dependence graph
  - Topological sort the dependence graph
- Rule-based methods (treewalk)
  - Analyze rules at compiler-generation time
  - Determine a fixed (static) ordering
  - Evaluate nodes in that order
- Oblivious methods (passes, dataflow)
  - Ignore rules & parse tree
  - Pick a convenient order (at design time) & use it

Evaluation

For "–101"
- Knuh: a data-flow model for evaluation
  - Independent attributes first
  - Others in order as input values become available

Evaluation

For "–101"
- One possible evaluation order:
  1. List.pos
  2. Sign.neg
  3. Bit.pos
  4. Bit.val
  5. List.val
  6. Number.val
- Other ordering are possible

Dependence graph

For "–101"
- Annotate parse tree with attributes
For “–101”:

Dependence graph

Inherited attributes flow down in the tree

At leaves, add dependences between inherited and synthesized attributes

Add synthesized attributes

Complete graph
Now, throw away the parse tree...

Evaluation

- What could prevent evaluation?
  - Cyclic dependences
- We can only evaluate acyclic instances
  - We can prove that some grammars can only generate instances with acyclic dependence graphs
  - Largest class is “strongly non-circular” grammars (SNC)
  - SNC grammars can be tested in polynomial time
- Some methods discover circularity dynamically
  - Bad property for a compiler to have
    - (SNC grammars were first defined by Kennedy & Warren)
Syntax-directed translation
- Attribute grammars
  - Clean, declarative
  - Handle a wide variety of problems
  - BUT, have limitations
    - Never widely adopted
- Reality
  - LR parsers
  - Associate attributes with parser symbols
  - Apply arbitrary code actions on attributes
    - This is what you’re doing in the project

Realist’s alternative
- Ad-hoc syntax-directed translation
  - Associate pieces of code with each production
  - At each reduction, the code is executed
  - Arbitrary code provides complete flexibility
    - Includes ability to do tasteless & bad things
- To make this work:
  - Need names for attributes on lhs & rhs
  - Yacc introduced $\$, $1, $2, … $n, left to right

Syntax-directed translation and parsing
- Classes of attributes that fit into parsing
  - S-attributed definition
    - All attributes are synthesized
    - Use values from children & from constants
    - Can evaluate bottom-up in one pass
    - Fits cleanly into LR parsing

Example
- Building the AST

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Production</th>
<th>Action Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>Expr + Term</td>
<td>$$ = MakeAddNode($1,$3);</td>
</tr>
<tr>
<td></td>
<td>Expr - Term</td>
<td>$$ = MakeSubNode($1,$3);</td>
</tr>
<tr>
<td></td>
<td>Term</td>
<td>$$ = $1;</td>
</tr>
<tr>
<td>Term</td>
<td>Term * Factor</td>
<td>$$ = MakeMulNode($1,$3);</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
<td>$$ = $1;</td>
</tr>
<tr>
<td>Factor</td>
<td>[Expr]</td>
<td>$$ = $2;</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td>$$ = MakeNumNode(token);</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>$$ = MakeIdNode(token);</td>
</tr>
</tbody>
</table>

Syntax-directed translation and parsing
- L-attributed definition
  - Use values from parent, constants, & siblings
  - For production $A \rightarrow X_1 X_2 \ldots X_n$
  - Each attribute of $X_i$ depends on
    - Attributes of $X_1 X_2 \ldots X_{i-1}$, and
    - Inherited attributes of $A$
  - Evaluate in a single top-down pass (left to right)
  - Good match for predictive parsing, but can also be adapted for LR parsing

SDT and LR parsers
- Two issues
  - Where to store the attribute values
  - Integrating the action code
- Key: store attributes on stack
  - At a reduction of $A \rightarrow \beta$
    - Pop 3 × $|\beta|$ symbols – 1 symbol, 1 state, 1 value
    - Map values to $\$1 \ldots $n
    - Invoke action code on values, collect result in $\$
    - Push $\$ back on stack with new symbol, state
SDT at work

- Building a symbol table
  - Enter declaration information as processed
  - At end of declaration syntax, do some post processing
  - Use table to check errors as parsing progresses
- Simple error checking/type checking
  - Define before use → lookup on reference
  - Dimension, type, ... → check as encountered
  - Type conformance of expression → bottom-up walk
- Procedure interfaces are harder
  - Build a representation for parameter list & types
  - Check actual vs. formal parameter list

Theory and practice

Relationship between practice and attribute grammars

- Similarities
  - Both rules & actions associated with productions
  - Application order determined by tools
  - (Somewhat) abstract names for symbols
- Differences
  - Actions applied as a unit; not true for AG rules
  - Anything goes in ad-hoc actions; AG rules are (purely) functional
  - AG rules are higher level than ad-hoc actions

Next time...

- We’ve built our abstract syntax tree
- Now what?
  - Type checking
  - Symbol tables
  - Semantic checking

Notes

- Lecture too short
- Too repetitious
- Expand discussion of how yacc/cup implemented

Parsing wrap-up

- 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

Last time

- LR parsers
  - Production rules
    - $G \rightarrow E^+$
    - $E \rightarrow E + T^+$
    - $E \rightarrow T$

Error Recovery in LR Parsers

The problem: parser encounters an invalid token
Goal: Want to parse the rest of the file

Basic idea (panic mode):
- Assume something went wrong while trying to find handle for nonterminal A
- Pretend handle for A has been found; pop "handle", skip over input to find terminal that can follow A

Restarting the parser (panic mode):
- Find a restartable state on the stack (has transition for nonterminal A)
- Move to a consistent place in the input (token that can follow A)
- Perform (error) reduction (for nonterminal A)
- Print an informative message

Error Recovery

Yacc's (bison's) error mechanism (note: version dependent):
- Designated token \texttt{error}
- Used in error productions of the form
  \[ A \rightarrow \beta \text{ error} \alpha \]
- \( \alpha \) specifies synchronization points

When error is discovered:
- Pops stack until it finds state where it can shift the \texttt{error} token
- Resumes parsing to match \( \alpha \)

Special cases:
- \( \alpha = w \), where \( w \) is string of terminals; skip input until \( w \) has been read
- \( \alpha = \varepsilon \); skip input until state transition on input token is defined
- Error productions can have actions

Error Recovery

cmpdstmt: BEG stmt_list END
stmt_list : stmt
| stmt_list ; ; stmt
| error { yyerror("***Error: illegal statement
");

This should
- Throw out the erroneous statement
- Synchronize at ";" or "end" (implicit: \( \alpha = \varepsilon \))
- Writes message "***Error: illegal statement" to stderr

Example: begin a & 5 | hello ; a := 3 end
↑↑ resume parsing
**Error: illegal statement