Prelude

- **What is triskaidekaphobia?**
  - Fear of the number 13
  - Examples:
    - No aisle 13 in airplanes, no 13th floor in buildings
    - Fear of Friday the 13th
    - Paraskevidekatriaphobia or friggatriskaidekaphobia
- **Why unlucky 13?**
  - Parity: 12 considered perfect number
  - Christianity: 13th guest at last supper was Judas
  - Norse myth: 13th god Loki arranged murder of Baldr
  - Code of Hammurabi: no number 13
  - Lucky 13
    - Christianity: 13 guests at last supper, 13 attributes of mercy
    - Judaism: 13 principles of Maimonides, age of Bar Mitzvah
    - Sikhism: lucky number

Summary of parsing

- **Parsing**
  - A solid foundation: context-free grammars
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - LALR(1) parser generators

More power?

- **So far:**
  - LALR and SLR: reduce size of tables
  - Also reduce space of languages
  - What if I want to expand the space of languages?
- **What could I do at a reduce/reduce conflict?**
  - Try both reductions!
  - GLR parsing
    - At a choice: split the stack, explore both possibilities
    - If one doesn’t work out, kill it
    - Run-time proportional to “amount of ambiguity”
  - Must design the stack data structure very carefully
General algorithms

- Parsers for full class of context-free grammars
  - Mostly used in linguistics – constructive proof of decidability
- CYK (1965)
  - Bottom-up dynamic programming algorithm
  - \( O(n^3) \)
- Earley’s algorithm (1970)
  - Top-down dynamic programming algorithm
  - Developed the "•" notation for partial production (LR item)
  - Worst-case \( O(n^3) \) running time
  - But, \( O(n^2) \) even for unambiguous grammars
- GLR
  - Worse-case \( O(n^3) \), but \( O(n) \) for unambiguous grammars

LR parsing

- Input: \( a_1, a_2, \ldots, a_n \)
- Stack: \( X_m X_{m-1} \ldots X_1 \)
- LR Parsing Engine
- Action Goto
- Grammar
- LR tables
- Compiler construction

Real world parsers

- Real generated code
  - lex, flex, yacc, bison
- Interaction between lexer and parser
  - C typedef problem
  - Merging two languages
- Debugging
  - Diagnosing reduce/reduce conflicts
  - How to step through an LR parser

Parser generators

- Example: JavaCUP
  - LALR(1) parser generator
  - Input: grammar specification
  - Output: Java classes
    - Generic engine
    - Action/goto tables
  - Separate scanner specification
- Similar tools:
  - SableCC
  - yacc and bison generate C/C++ parsers
  - JavaCC: similar, but generates LL(1) parser

JavaCUP example

- Simple expression grammar
  - Operations over numbers only
  ```java
  // Import generic engine code
  import java_cup.runtime.*;
  /* Preliminaries to set up and use the scanner. */
  init with { scanner.init(); }
  scan with { return scanner.next_token(); ; }
  Note: interface to scanner
  One issue: how to agree on names of the tokens
  ```

Example

- Define terminals and non-terminals
  - Indicate operator precedence
  ```java
  /* Terminals (tokens returned by the scanner). */
  terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
  terminal UNMINUS, LPAREN, RPAREN;
  terminal Integer NUMBER;
  /* Non terminals */
  expr_list, expr_part;
  non terminal Integer expr, term, factor;
  /* Precedences */
  precedence left PLUS, MINUS;
  precedence left TIMES, DIVIDE, MOD;
  ```
Example

- Grammar rules

```
expr_list ::= expr_list expr_part
  |  expr_part;
expr_part ::= expr SEMI
expr ::= expr PLUS expr
  |  expr MINUS expr
  |  expr TIMES expr
  |  expr DIVIDE expr
  |  LPAREN expr RPAREN
  |  NUMBER;
```

Roadmap

- Parsing
  - Tells us if input is syntactically correct
  - Gives us derivation or parse tree
  - But we want to do more:
    - Build some data structure – the IR
    - Perform other checks and computations

Syntax-directed translation

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

- General strategy
  - Associate values with grammar symbols
  - Associate computations with productions

- Implementation approaches
  - Formal: attribute grammars
  - Informal: ad-hoc translation schemes

- Syntax-directed translation

Example

- Desk calculator
  - Expression grammar
  - Build parse tree
  - Evaluate the resulting tree

```
Production rule
#      | G   | E   | T   | F
-----   |-----|-----|-----|-----
1        | G   | E   | T   | F
2        | E   | E, T| T   | T
3        | T   | T   | F   |
4        | T   | T   | T   |
5        | T   | T   |
6        | F   | F   |
7        | F   | F   |
```

Example

- Can we evaluate the expression without building the tree first?
  - “Piggyback” on parsing
Example

- Codify:
  - Store intermediate values with non-terminals
  - Perform computations in each production

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G → E</td>
<td><code>print(E.val)</code></td>
</tr>
<tr>
<td>2</td>
<td>E → E₁ + T</td>
<td>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>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 → (E₁)</td>
<td>F.val ↔ E₁.val</td>
</tr>
<tr>
<td>7</td>
<td>F → num</td>
<td>F.val ↔ valueof(num)</td>
</tr>
</tbody>
</table>

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

The bad news:
Formal 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

Example

- 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>List → Bit</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>
</tbody>
</table>

Example derivations

For "-1"

Number → Sign List → Sign List Bit → Sign List 1 → Sign List Bit 1 → Sign List 1 1 → Sign Bit 0 1 → Sign 1 0 1 → -101

For "-101"

Number → Sign List → Sign List Bit → Sign List 1 → Sign List Bit 1 → Sign List 1 1 → Sign Bit 0 1 → Sign 1 0 1 → -101

Attribute grammar

- Goal:
  Compute the value of the binary number

- Information we need:
  - Position of each 1 bit – to compute place value
    101 = 1*2² + 0*2¹ + 1*2⁰
  - Sum of bit values

- Computation:
  - Propagate position information
  - Accumulate the sums

<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>List → Bit</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>
</tbody>
</table>

Attribute rules
**Attribution rules**

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
<th>Attribution rules</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>( \text{List}.\text{val} \rightarrow \text{List}.\text{val} + \text{Bit}.\text{val} )</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>List.pos ← List.pos + 1</td>
</tr>
<tr>
<td>4</td>
<td>List₀ → List₁ Bit</td>
<td>Bit.pos ← List₀.pos</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>or Bit.pos ← List₀.pos</td>
</tr>
<tr>
<td>6</td>
<td>Bit → 0</td>
<td>Now we can compute Bit value</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Bit.val = 0 or Bit.val = 2^(Bit.pos)</td>
</tr>
</tbody>
</table>

**Notice**: Information can flow top-down or bottom-up (Also: Left-to-right or Right-to-left)

**Attribute grammars**

- **Specification:**
  - Attributes
    - Associated with nodes in parse tree
    - Distinguish multiple non-terminals with index
  - 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 \textit{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 dependences between attributes
    - Let’s look at a general technique for evaluation

**Dependence graph**

*For ~101*
Dependence graph

Annotate parse tree with attributes

Dependence graph

Inherited attributes flow down in the tree

Dependence graph

At leaves, add dependences between inherited and synthesized attributes

Dependence graph

Collect the synthesized attributes

Dependence graph

Complete graph

Now, throw away the parse tree...

Dependence graph

Need a method to solve the set of constraints described by this dependence graph


### Evaluation

- Dynamic, dependence-based methods
  - Build the parse tree, dependence graph
  - Topologically sort the graph
  - Gives us an order of evaluation
- Rule-based methods
  - Analyze rules at compiler-generation time
  - Determine a fixed (static) ordering
  - Evaluate nodes in that order
- Oblivious methods
  - Ignore rules & parse tree
  - Pick a convenient order (at design time) & use it

### Syntax-directed translation

- Attribute-directed methods
  - Clean, declarative
  - Handle a wide variety of problems
  - BUT, have limitations and evaluation issues
  - Never widely adopted
- Reality
  - In practice:
    - Apply arbitrary code actions on attributes
    - Order of evaluation dictated by parsing algorithm
      - Only works for limited classes of attribute grammars

### Special class: L-attributed

- **L-attributed** definition
  - Use values from parent and 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 $A$
- Suited to LL parsing
  - Evaluate in a single top-down pass (left to right)
  - Pass values down through recursive descent
  - Emit code to compute attributes in appropriate procedures

### Special class: S-attributed

- **S-attributed** definition
  - All attributes are synthesized
  - For production $A \rightarrow X_1 X_2 \ldots X_n$
  - Value of $A$ is computed as a function of the attributes already computed for $X_1 X_2 \ldots X_n$
- Suited to LR parsing
  - Can be computed in a single bottom-up pass
  - Associate pieces of code with each production
  - At each reduction, the code is executed

### Example

```
expr_part ::= expr SEMI

expr ::= expres1 PLUS expres2
      | expres1 MINUS expres2
      | LPAREN expres RPAREN
      | NUMBER

RESULT = new Integer(e1.intValue() + e2.intValue());
```

### Another example

- Build an abstract syntax tree

```
expr_part ::= expr SEMI

expr ::= expres1 PLUS expres2
      | expres1 MINUS expres2
      | LPAREN expres RPAREN
      | NUMBER

RESULT = new AddNode(e1, e2);
```

---

**Name for the value associated with this production**

**Arbitrary code between [: and :]**

**RESULT refers to the attribute of the LHS non-terminal**
Implementation

- How does this work?
  - Where are the attributes stored?
  - What do e1, e2, n, RESULT refer to?
- Key: store attributes on stack
  At a reduction of $A \rightarrow \beta$
  - Pop $3 \times |\beta|$ symbols – 1 symbol, 1 state, 1 attribute value
  - Map values to names:
    - expr\(1\) PLUS expr\(2\)
    - e\(2\) = top of stack, then PLUS, then e\(1\)
  - Invoke action code on values – store in RESULT
  - Push RESULT back on stack with new symbol, state

Next...

- Static checking