Prelude

- What is a qubit?
  - Quantum bit
- Why quantum computing?
  - "Superposition" can search solutions to a problem simultaneously
  - 3 bits: 1 of 8 possible values
  - 3 qubits = all 8 values, with probabilities
- Is it fundamentally more powerful?
  - No. Just massively parallel.

Practical?

Today

- Issues with LR parsers
- Syntax-directed translation
- New homework assignment (on parsing)

Issues with LR parsers

- What happens if a state contains:
  \[ [X \rightarrow \alpha \cdot a \beta, b] \text{ and } [Y \rightarrow \gamma \cdot a] \]
- Then on input "a" we could either
  - Shift into state \([X \rightarrow \alpha a \cdot \beta, b]\), or
  - Reduce with \(Y \rightarrow \gamma\)
- This is called a shift-reduce conflict
  - Typically due to ambiguity

Shift/Reduce conflicts

- Classic example: the dangling else
  \[ S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER} \]
- Will have DFA state containing
  \[ [\text{S \rightarrow if E then S \cdot else}] \]
  \[ [\text{S \rightarrow if E then S \cdot else S \cdot x}] \]
- Practical solutions:
  - Painful: modify grammar to reflect the precedence of else
  - Many LR parsers default to "shift"
  - Often have a precedence declaration
Another example

- Consider the ambiguous grammar
  \[ E \rightarrow E + E \mid E * E \mid \text{int} \]
- Part of the DFA:
  \[
  \begin{align*}
  & [E \rightarrow E \cdot E, +] & E \\
  & [E \rightarrow E + E, +] & E \\
  & [E \rightarrow E \cdot E, +] & E \\
  & \ldots & \ldots
  \end{align*}
  \]
- We have a shift/reduce on input +
- What do we want to happen?
  - Consider: \( x \cdot y + z \)
  - We need to reduce (\( \cdot \) binds more tightly than +)
  - Default action is shift

Precedence

- Declare relative precedence
  - Explicitly resolve conflict
  - Tell parser: we prefer the action involving \( \cdot \) over +
  \[
  \begin{align*}
  & [E \rightarrow E \cdot E, +] & E \\
  & [E \rightarrow E + E, +] & E \\
  & [E \rightarrow E \cdot E, +] & E \\
  & \ldots & \ldots
  \end{align*}
  \]
- In practice:
  - Parser generators support a precedence declaration for operators

More…

- Still a problem?
  \[
  \begin{align*}
  & [E \rightarrow E \cdot E, +] & E \\
  & [E \rightarrow E * E, +] & E \\
  & [E \rightarrow E + E, +] & E \\
  & \ldots & \ldots
  \end{align*}
  \]
- Shift/reduce conflict on +
  - Do we care?
  - Maybe: we want left associativity
    - parse: “a+b+c” as “((a+b)+c)”
  - Which rule should we choose?
  - Also handled by a declaration “+ is left-associative”

Other problems

- If a DFA state contains both
  \[ X \rightarrow \alpha \cdot a \] and \[ Y \rightarrow \beta \cdot a \]
  - What’s the problem here?
  - Two reductions to choose from when next token is \( a \)
  - This is called a reduce/reduce conflict
  - Usually a serious ambiguity in the grammar
  - Must be fixed in order to generate parser

Reduce/Reduce conflicts

- Example: a sequence of identifiers
  \[ S \rightarrow c \mid id \mid id S \]
- There are two parse trees for the string \( id \)
  \[ S \rightarrow id \]
  \[ S \rightarrow id \mid id S \rightarrow id \]
- How does this confuse the parser?

Reduce/Reduce conflicts

- Consider the DFA states:
  \[
  \begin{align*}
  & [S \rightarrow \text{id} \mid \text{id} S, \$] \\
  & [S \rightarrow \text{id} \mid \text{id} S, \$] \\
  & [S \rightarrow \text{id}, \$] \\
  & [S \rightarrow \text{id}, \$] \\
  & [S \rightarrow \text{id} S, \$] \\
  & [S \rightarrow \text{id} S, \$]
  \end{align*}
  \]
- Reduce/reduce conflict on input \$:
  \[ G \rightarrow S \rightarrow id \]
  \[ G \rightarrow S \rightarrow id S \rightarrow id \]
- Better rewrite the grammar: \[ S \rightarrow c \mid id S \]
Practical issues

We use an LR parser generator…

- Question: how many DFA states are there?
  - Does it matter?
  - What does that affect?
    - Parsing time is the same
    - Table size: occupies memory

- Even simple languages have 1000s of states
  - Most LR parser generators don’t construct the DFA as described

---

LR(1) Parsing tables

- But many states are similar, e.g.
  - \( E \rightarrow \text{int} \cdot , \) and \( E \rightarrow \text{int} \cdot ,/+ \)
  - How can we exploit this?
    - Same reduction, different lookahead tokens
    - Idea: merge the states

---

The core of a set of LR Items

- When can states be merged?
  - **Def:** the core of a set of LR items is:
    - Just the production parts of the items
    - Without the lookahead terminals

- Example: the core of
  \[ \{ [X \rightarrow \alpha \cdot , b], [Y \rightarrow \gamma \cdot , d] \} \]
  is
  \[ \{ X \rightarrow \alpha \cdot , Y \rightarrow \gamma \cdot \} \]

---

Merging states

- Consider for example the LR(1) states
  \[ \{ [X \rightarrow \alpha \cdot , b], [Y \rightarrow \beta \cdot , d] \} \]
  \[ \{ [X \rightarrow \alpha \cdot , b], [Y \rightarrow \beta \cdot , d] \} \]
  - They have the same core and can be merged
  - Resulting state is:
    \[ \{ X \rightarrow \alpha \cdot , b/d, [Y \rightarrow \beta \cdot , d/d] \} \]
  - These are called **LALR(1) states**
    - Stands for LookAhead LR
    - Typically 10X fewer LALR(1) states than LR(1)

---

The LALR(1) DFA

- **Algorithm:**
  - **repeat**
    - Choose two states with same core
    - Merge the states by combining the items
    - Point edges from predecessors to new state
    - New state points to all the previous successors
    - until all states have distinct core

---

Conversion LR(1) to LALR(1).

Does this state do the same thing?
LALR states
- Consider the LR(1) states:
  \([\{X \to \alpha \cdot a], [Y \to \beta \cdot b]\}\)
  \([\{X \to \alpha \cdot b], [Y \to \beta \cdot a]\}\)
- And the merged LALR(1) state
  \([\{X \to \alpha \cdot a/b], [Y \to \beta \cdot a/b]\}\)
- What’s wrong with this?
  - Introduced a new reduce-reduce conflict
  - In practice such cases are rare

LALR vs. LR Parsing
- LALR is an efficiency hack on LR languages
- Any “reasonable” programming language has a LALR(1) grammar
  Languages that are not LALR(1) are weird, unnatural languages
- LALR(1) has become a standard for programming languages and for parser generators
- Variant: SLR
  LR(0), with special rule: reduce A -> b only if next token is in FOLLOW of A

LR parsing
- Input: \(a_1, a_2, \ldots, a_i, \ldots, a_n, \$$
- Stack: \(s_0, X_m, s_{m-1}, X_{m-1}, \ldots, s_1, \) 
- LR Parsing Engine
- Scanner
- Compiler construction
- LR tables
- Action/Goto

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
  /* Terminals (tokens returned by the scanner). */
  terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
  terminal UMINUS, LPAREN, RPAREN;
  terminal Integer NUMBER;
  /* Non terminals */
  non terminal expr_list, expr_part;
  non terminal expr, term, factor;
  /* Precedences */
  precedence left PLUS, MINUS;
  precedence left TIMES, DIVIDE, MOD;
  /* Terminals (tokens returned by the scanner). */
  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
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 |
       expr MOD expr |
       LPAREN expr RPAREN |
       NUMBER ;
```

Notes on 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

A Hierarchy of Grammar Classes

From Andrew Appel, "Modern Compiler Implementation in Java"

Overview

- 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
    - Syntax-directed translation
  - Parser generators
    - Add action code to do something
    - Typically build the IR
  - How much can we do during parsing?
Example

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

G → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → num

Production rule

Example

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

G → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → num

Production rule

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>print(E.val)</td>
</tr>
<tr>
<td>2</td>
<td>E → E1 + T</td>
<td>E1.val ← E.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>

Production rule

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:
  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>Bit → 0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Bit → 1</td>
<td></td>
</tr>
</tbody>
</table>

Production rule

Example derivations
Attribute grammar

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

### Attributes

**Rules**

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
<th>How to compute Number from Sign and List?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number → Sign List</td>
<td>if Sign.neg then Number.val ← -1 * List.val else Number.val ← List.val</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>List₀ → List₁ Bit</td>
<td>List₀.val ← List₁.val + Bit.val</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Bit → 0</td>
<td>Bit.val ← 0</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Bit.val ← 2(Bit.pos)</td>
</tr>
</tbody>
</table>

**Attributes**

- Top-down values are **inherited attributes**
- Bottom-up values are **synthesized attributes**
- Values with no dependence are called **independent attributes**
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?

Dependence graph

Annotate parse tree with attributes

Inherited attributes flow down in the tree

At leaves, add dependences between inherited and synthesized attributes

Collect the synthesized attributes
**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 grammars
  - 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

**Adding actions**

- **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
    - Table driven: store intermediate values on stack

- **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 ::= expr :e1 PLUS expr :e2
{ RESULT = new Integer(e1.intValue() + e2.intValue()); :}
| expr:Minus expr :e2
{ RESULT = new Integer(e1.intValue() - e2.intValue()); :}
| LPAREN expr :e RPAREN
{ RESULT = e; :}
| NUMBER :n
{ RESULT = n; :} ;

Name for the value associated with this production
Arbitrary code between {: and :}
RESULT refers to the attribute of the LHS non-terminal

Another example
Build an abstract syntax tree

expr_part ::= expr SEMI ;
expr ::= expr :e1 PLUS expr :e2
{ RESULT = new AddNode(e1, e2); :}
| expr:Minus expr :e2
{ RESULT = new SubNode(e1, e2); :}
| LPAREN expr :e RPAREN
{ RESULT = e; :}
| NUMBER :n
{ RESULT = new NumberNode(n); :}

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 → β
- Pop 3 * |β| symbols – 1 symbol, 1 state, 1 value
- Map values to names:
  - e2 = top of stack, then PLUS, then e1
- Invoke action code on values – store in RESULT
- Push RESULT back on stack with new symbol, state

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