**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

--

**Bottom-up parsing**
- Start with input stream
  - “Leaves” of parse tree
- Build up towards goal symbol
  - Construct the reverse derivation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>aA</td>
</tr>
<tr>
<td>3</td>
<td>ab</td>
</tr>
<tr>
<td>4</td>
<td>abcde</td>
</tr>
<tr>
<td>5</td>
<td>abbcde</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G → a A B e</td>
</tr>
<tr>
<td>2</td>
<td>A → A b c</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>B → d</td>
</tr>
</tbody>
</table>

**Bottom-up parsing**
- Problem:
  - Not good enough to simply find production right-hand sides and reduce
  - Example:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G → a A B e</td>
</tr>
<tr>
<td>2</td>
<td>A → A b c</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>B → d</td>
</tr>
</tbody>
</table>

```
   | a    | b    | c    | d    | e    |
---|------|------|------|------|------|
G  | A    | B    |
A  | a    | b    | c    |
B  | d    |
```

- “aAbcde” is not part of any sentential form

**LR parsing**
- State of the parser:
  - $\alpha \mid \gamma$
  - $\alpha$ is a stack of terminals and non-terminals
  - $\gamma$ is string of unexamined terminals

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E → E + ( E )</td>
</tr>
<tr>
<td>2</td>
<td>E → int</td>
</tr>
</tbody>
</table>

- Two operations:
  - **Shift** – read next terminal, push on stack
    - $E \ast (\text{int}) \rightarrow E \ast (\text{int})$
  - **Reduce** – pop RHS symbols off stack, push LHS
    - $E \ast (E + (E)) \rightarrow E \ast (E)$

**LR parsing**
- repeat
  - if top symbols on stack $\beta$ for some $A \rightarrow \beta$
    - **Reduce**: “found an A”
    - **Pop** those symbols off
    - **Push A on stack**
  - else Get next token from scanner
    - if token is useful
      - **Shift**: “still working on something”
      - **Push on stack**
    - else error
  - until stack contains goal and no more input
Example

1. `int + ( int ) + ( int )` Nothing on stack, get next token

Stack

Example

1. `int + ( int ) + ( int )` Nothing on stack, get next token
2. `int | + ( int ) + ( int )` Shift: push int

Top of stack matches

E → int

Stack

Example

1. `int + ( int ) + ( int )` Nothing on stack, get next token
2. `int | + ( int ) + ( int )` Shift: push int
3. `int | + ( int ) + ( int )` Reduce: pop int

Top of stack matches

E → int

Stack

Example

1. `int + ( int ) + ( int )` Nothing on stack, get next token
2. `int | + ( int ) + ( int )` Shift: push int
3. `int | + ( int ) + ( int )` Reduce: pop int, push E

Top of stack matches

E → int

Stack

Example

1. `int + ( int ) + ( int )` Nothing on stack, get next token
2. `int | + ( int ) + ( int )` Shift: push int
3. `int | + ( int ) + ( int )` Reduce: pop int, push E
4. `int | + ( int ) + ( int )` Shift: push +
5. `int + ( | int | ) + ( int )` Shift: push +
6. `int + ( int | ) + ( int )` Shift: push int
7. `int + ( int | ) + ( int )` Reduce: pop int, push E

Top of stack matches

E → int

Stack
Example

1. int + ( int ) + ( int )  Nothing on stack, get next token
2. int | + ( int ) + ( int )  Shift: push int
3. int | + ( int ) + ( int )  Reduce: pop int, push E
4. int | + ( int ) + ( int )  Shift: push +
5. int | + ( int ) + ( int )  Shift: push ( int ) + ( int )
6. int | + ( int ) + ( int )  Shift: push int
7. int | + ( int ) + ( int )  Reduce: pop int, push E
8. int | + ( int ) + ( int )  Shift: push )

Stack  

Top of stack matches E → E + ( int )

Stack  

Example

....

9. int + ( int ) | + ( int )  Reduce: pop x 5, push E
10. int + ( int ) | + ( int )  Shift: push +
11. int + ( int ) | + ( int )  Shift: push ( int ) + ( int )
12. int + ( int ) | + ( int )  Shift: push int

Stack  

Example

....

9. int + ( int ) | + ( int )  Reduce: pop x 5, push E
10. int + ( int ) | + ( int )  Shift: push +
11. int + ( int ) | + ( int )  Shift: push ( int ) + ( int )
12. int + ( int ) | + ( int )  Shift: push int
13. int + ( int ) | + ( int )  Reduce: pop int, push E
14. int + ( int ) | + ( int )  Shift: push )
15. int + ( int ) | + ( int )  Reduce: pop x 5, push E

Stack  

DONE!

Key problems

- (1) Will this work?
- How do we know that shifting and reducing using a stack is sufficient to compute the reverse derivation?

- (2) How do we know when to shift and reduce?
  - Can we efficiently match top symbols on the stack against productions?
    - Right-hand sides of productions may have parts in common
  - Will shifting a token move us closer to a reduction?
  - Are we making progress?
  - How do we know when an error occurs?
Key problems

1. Will this always work?
   - Yes, for unambiguous grammars

Why?

- Unambiguous:
  - Unique right-most derivation for every string
  - At each parsing step, one possible reduction

\[ G \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \rightarrow \gamma_5 \rightarrow \text{input} \]

Shift-reduce parsing

- Consider last step:
  - \[ G \rightarrow \cdots \rightarrow a b c B x y z \rightarrow \cdots \rightarrow \text{input} \]
  - Production:
    \[ B \rightarrow g r s \]

To reverse this step:

- Shift until \( g r s \) on top of stack
- Reduce: pop \( g r s \), push \( B \)

Parsing state:

\[ \text{Input: } a b c q r s | x y z \]
\[ \text{Stack: } 2 b c B \]

Right-most derivation

- Could there be an alternative reduction?
  - \[ G \rightarrow \cdots \rightarrow a b c B x y z \rightarrow \cdots \rightarrow \text{input} \]
  - and \[ G \rightarrow \cdots \rightarrow a D r s x y z \rightarrow \cdots \rightarrow \text{input} \]
  - No
    - Two right-most derivations for the same string
    - I.e., the grammar would be ambiguous

Implications

- Cases:
  - \( \gamma_5 \rightarrow a f y z \rightarrow a b c B x y z \)
  - \( \gamma_4 \rightarrow a c B H z \)
- Parsing state:
  - Input: \( a b c q r s | x y z \)
  - Stack: \( 2 b c B \)
- Key: next reduction must consume top of stack
  - Possibly after shifting more terminal symbols

Key problems

2. How do we know when to shift or reduce?

- Reductions
  - Good news:
    - At any given step, reduction is unique
    - Matching production occurs at top of stack
  - Problem:
    - How to efficiently find the right production

- Shifts
  - Default behavior: shift when there's no reduction
  - Still need to handle errors
When to shift or reduce

- What is on the stack?

  **Either:**
  - A right sentential form
  - A prefix of a right sentential form (missing some terminals)

  Called a **viable prefix**

- **Idea:** a DFA that recognizes viable prefixes
  - Input: stack contents (a mix of terminals, non-terminals)
  - Each state represents either
    - A right sentential form – labeled with the reduction to apply
    - A viable prefix – labeled with tokens to expect next

---

Shift/reduce DFA

- **Using the DFA**
  - At each parsing step run DFA on stack contents
  - Examine the resulting state X and the token t immediately following | in the input stream
    - If X has an outgoing edge labeled t, then shift
    - If X is labeled “A → β on t”, then reduce

- **Example:**

  - First, we’ll look at how to use such a DFA…
Improvements

- Each DFA state represents stack contents
  - At each step, we rerun the DFA to compute the new state
  - Can we avoid this?
- Two actions:
  - Shift: Push a new token
  - Reduce: Pop some symbols off, push a new symbol
- Idea:
  - For each symbol on the stack, remember the DFA state that represents the contents up to that point
  - Push a new token = go forward in DFA
  - Pop a sequence of symbols = “unwind” DFA to previous state

Algorithm components

- Stack
  - String of the form: (sym1, state1)…(symn, staten)
  - symi: grammar symbol (left part of string)
  - statei: DFA state
    - Intuitively: represents what we’ve seen so far
    - state, is the final state of the DFA on sym1 … symk
    - And, captures what we’re looking for next
- Represent as two tables:
  - action – whether to shift, reduce, accept, error
  - goto – next state

Algorithm

- Work
  - Shift each token
  - Pop each token
- Errors
  - Input exhausted
  - Error entry in table

Tables

- Action
  - Given state and the next token, action[s, a] =
    - Shift s’, where s’ is a state
    - Reduce by a grammar production A → β
    - Accept
    - Error
- Goto
  - Given a state and a grammar symbol, goto[s, X] =
    - Transition to next state (after recognizing an X)

Representing the DFA

- Combined table:
LR Items

- An LR(1) item is a pair:
  \[ [A \rightarrow \alpha \cdot \beta, a] \]
- \( A \rightarrow \alpha \beta \) is a production
- \( a \) is a terminal (the lookahead terminal)
- LR(1) means 1 lookahead terminal

- \([A \rightarrow \alpha \cdot \beta, a]\) describes a context of the parser
  - We are trying to find an \( A \) followed by an \( a \)
  - We have \( \alpha \) on top of the stack
  - We need to see a prefix derived from \( \beta \)

LR Items

- Consider the item
  \[ E \rightarrow E + ( \cdot E ), + \]
  - What could we see next?
  - We expect a string derived from \( E \) +
  - There are two productions for \( E \)
    \( E \rightarrow \text{int} \) \( and \) \( E \rightarrow E + ( E ) \)
  - We extend the context with two more items:
    \( E \rightarrow \cdot \text{int} , ) \)
    \( E \rightarrow E + ( E ), ) \)

LR Items

- In context containing
  \[ [E \rightarrow E + ( \cdot E ), +] \]
  - If \( + \) next then we can a shift to context containing
    \[ [E \rightarrow E + ( \cdot E ), +] \]
  - In context containing
    \[ [E \rightarrow E + ( \cdot E ), +] \]
    - We can reduce with \( E \rightarrow E + ( E ) \)
    - But only if \( a + \) follows

Closure operation

- The operation of extending the context with items is called the closure operation
- Closure function:
  - Given a set of items
  - Compute all other items that could represent the current parsing state
- Observation:
  - At \( A \rightarrow \alpha B \beta \) we expect to see \( B \beta \) next
  - If \( B \rightarrow \gamma \) is a production, then we could also see a \( \gamma \)
Closure operation

- Algorithm:
  
  closure(Items) =
  repeat
    for each [A → α • Bβ, a] in Items
      for each production B → γ
        add [B → • γ, ?] to Items
  until Items is unchanged

Building the DFA – part 1

- Starting context = closure(S → • E, $)
  - Abbreviated:
    - S → • E, $
    - E → E(E),$
    - E → int, $
    - E → E(E),+

DFA transitions

- Idea:
  - If the parser was in state [A → αXβ] and then recognized an instance of X, then the new state is [A → αX•β]
  - Note: X could be a terminal or non-terminal

- Algorithm:
  transition(l, X) =
  J = {};
  for each [A → αXβ, b] ∈ L
    add [A → αX•b, b] to J
  return closure(J)

Building the DFA – part 2

- DFA states
  - Each DFA state is a closed set of LR(1) items
  - Start state: closure(S → • E, $)

- Reductions
  - Label each item [A → αX•, x] with “Reduce with A → αX• on x”

- What about transitions?

DFA construction

- Data structure:
  - T – set of states (each state is a set of items)
  - E – edges of the form I → J
    - where I, J ∈ T and X is a terminal or non-terminal

- Algorithm:
  T = closure(S → • E, $), E = {};
  repeat
    for each state I in T
      for each item [A → αXβ, b] ∈ L
        let J = transition(I, X)
        T = T + J
        E = E + {I → J}
    until E and T no longer change
To form into tables

- Two tables
  - action(I, token)
  - goto(I, symbol)

- Layout:
  - One row for each states – each I in T
  - One columns for each symbol

- Entries:
  - For each edge I → J
  - If X is a terminal, add shift J at position (I, X) in action
  - If X is a non-terminal, add goto J at position (I, X) goto
  - For each state [ A → aβ • x ] in I
    - Add reduce n at position (I, x) in action (where n is |rhs|)

Next time...

- Some issues with bottom-up parsing
- Parser-generator tools
- New programming assignment