Announcements

- HW2 due on Tuesday

Last Class

- Top-down (LL) parsing
  - LL(1) parsing tables, FIRST, FOLLOW and PREDICT sets
  - Writing an LL(1) grammar
- Bottom-up (LR) parsing
  - Intro with example

Today's Lecture Outline

- Bottom-up (LR) parsing
  - Handles
  - LR Items
  - Characteristic Finite State Machine (CFSM)
  - SLR(1) parsing tables
  - Conflicts in SLR(1)
  - LR parsing variants

Programming Language Syntax

Bottom-up Parsing

Read: Scott, Chapter 2.3.3

Bottom-up Parsing

- Also called LR parsing
- LR parsers work with LR(k) grammars
  - L stands for “left-to-right” scan of input
  - R stands for “rightmost” derivation
  - k stands for “need k tokens of lookahead”
- We are interested in LR(0) and LR(1) and variants in between
- LR parsing is better than LL parsing!
  - Accepts larger class of languages
  - Just as efficient!

Model of the LR Parser

- Stack ← Input
  - Stack: holds the part of the input seen so far
    - A string of both terminals and nonterminals
  - Input: holds the remaining part of the input
    - A string of terminals
- Parser performs two actions
  - Reduce: parser pops a “suitable” production right-hand-side off top of stack, and pushes production's left-hand-side on the stack
  - Shift: parser pushes next terminal from the input on top of the stack
id + id*id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>id+id*id</td>
<td>shift id</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>+id*id</td>
<td>reduce by term→id</td>
</tr>
<tr>
<td>term</td>
<td>+id*id</td>
<td>reduce by expr→term</td>
</tr>
<tr>
<td>expr</td>
<td>+id*id</td>
<td>shift +</td>
</tr>
<tr>
<td>expr+</td>
<td>id*id</td>
<td>shift id</td>
</tr>
<tr>
<td>expr+id</td>
<td>*id</td>
<td>reduce by term→id</td>
</tr>
</tbody>
</table>

Sequence of reductions performed by parser

- A rightmost derivation in reverse
- The stack (e.g., expr) concatenated with remaining input (e.g., +id*id) gives a sentential form (expr+id*id) in the rightmost derivation.
- Right sentential forms.

Question

- Consider id*id*id
- Stack | Input
- term  | *id*id Is expr→term at position ε a handle of term*id*id?
- Answer: No! It brings sentential form term*id*id into expr*id*id which is NOT derivable from expr!

Handle

- A handle
- Consider a rightmost derivation $S \Rightarrow \ldots \Rightarrow \alpha A \beta \Rightarrow \alpha \beta w$. We say that $A \rightarrow \beta$ at position $\alpha$ is a handle of $\alpha \beta w$
- Recall our example id+id*id
- Stack | Input
- expr→expr+term | term term→term*id|id
- expr→expr+term  | *id Is expr→expr+term at position $\varepsilon$ a handle of expr*term*id?
- expr→expr+term*id  Is term→id at position expr*term* a handle of expr*term*id?

Question

- How about
- Stack | Input
- term*id  | *id Is term→term*id at position $\varepsilon$ a handle of term*id*id?
- Answer: Yes! It brings sentential form term*id*id into term*id which is clearly derivable: expr→term→term*id
id + id*id

Stack    Input    Action
0        id + id*id  On state 0 and id, action[0, id] = shift 3
0 id 3   +id*id     On 3 and +, action[3, +] = reduce by term → id
0 term   On 0 and term, goto[0, term] = 2
0 term 2  +id*id    On 2 and +, action[2, +] = ...

Lecture Outline

- Bottom-up (LR) parsing
  - Handles
  - LR Items
  - Characteristic Finite State Machine (CFSM)
  - SLR(1) parsing table
  - Conflicts in SLR(1)
  - LR Parsing variants

LR Items

- An LR item is a production with a dot at some position on the right-hand side
  - E.g., A → αβ
  - We are trying to find an A
  - We already have seen α (it is on top of the stack)
  - We are looking for β

Closure of an LR Item

- The closure of an LR item A → αβ is the set of LR items formed as follows:
  - A → αβ is in the closure of A → αβ
  - If the dot is in front of a nonterminal B for some item in the closure, then all of B → γ1, B → γ2, ...
    B → γn are in the closure (B → γ1, B → γ2, ... B → γn are all productions for B)

Example

- Compute closure of start → • expr

  Answer:
  start → • expr
  expr → • expr + term
  expr → • term
  term → • term * id
  term → • id
Question

Compute closure of $expr \rightarrow expr + \cdot term$

- Answer:
  
  $expr \rightarrow expr + \cdot term$
  
  $term \rightarrow \cdot term \cdot id$
  
  $term \rightarrow \cdot id$

Example

Construct the collection of sets of LR items with transitions for the above grammar

Lecture Outline

- Bottom-up (LR) parsing
  - Handles (brief review)
  - LR Items
  - Characteristic Finite State Machine (CFSM)
  - SLR(1) parsing table
  - Conflicts in SLR(1)
  - LR Parsing variants

Sets of LR Items with Transitions

The collection of sets of items with transitions is a DFA. This DFA is one part of the CFSM (we’ll see the other part shortly). CFSM states are parsing states. Transitions on terminals represent shifts. Transitions on nonterminals represent gotos.
Question

Sets of LR Items with Transitions

• 3,7 contain only items of kind $A \rightarrow \alpha$, i.e., reduce items
• 0,4,5 contain items of kind $A \rightarrow \alpha \beta$, i.e., shift items
• 1,2,6 contains both reduce and shift items

Question

Sets of LR Items with Transitions

• After parser pops right-hand side term{id} off the stack (as it reduces in state 7), what state(s) could end up on top of the stack?

“Reduce by” Labels

For every state that contains a reduce item $A \rightarrow \alpha$, add label “reduce by $A \rightarrow \alpha$ on FOLLOW($A$)”

For example, we add label on state 2:

}$expr \rightarrow term{id}$, reduce by $expr \rightarrow term$ on $\$$, +.$

Characteristic Finite State Machine (CFSM)
CFSM consists of 2 parts
- Collection of sets of LR items with transitions
- “Reduce by” labels
To construct the CFSM for a grammar \( G \)
- First, construct the collection of sets of LR items with transitions
- Second, add the “reduce by” labels

### From CFSM to SLR(1) Parsing Table

<table>
<thead>
<tr>
<th>State in CFSM</th>
<th>Action</th>
<th>$+$</th>
<th>*</th>
<th>$$$</th>
<th>expr</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>shift 3</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>shift 4</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>reduce 2</td>
<td>shift 5</td>
<td>reduce 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>reduce 4</td>
<td>reduce 4</td>
<td>reduce 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>shift 3</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>shift 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>reduce 1</td>
<td>shift 5</td>
<td>reduce 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>reduce 3</td>
<td>reduce 3</td>
<td>reduce 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

White – action table
Blue – goto table

### Exercise

Construct the CFSM for above grammar
- First, construct collection of sets of LR items with transitions
- Second, add “reduce by” labels

### Lecture Outline

- Bottom-up (LR) parsing
  - Handles (brief review)
  - LR Items
  - Characteristic Finite State Machine (CFSM)
  - SLR(1) parsing table
  - Conflicts in SLR(1)
  - LR Parsing variants

### Conflicts in SLR(1): Shift-reduce

- **Shift-reduce conflict** in state \( k \) on \( a \):
  - State \( k \) contains item \( A \rightarrow \alpha \beta \) and \( a \) is in \( \text{FOLLOW}(A) \)
  - **and**
  - State \( k \) contains item \( A' \rightarrow \alpha a \beta' \)
  - The parser does not know whether it is at the end of production \( A \rightarrow \beta \) and thus should reduce by \( A \rightarrow \beta \), or it is in the middle of production \( A' \rightarrow \alpha a \beta' \) and thus should shift \( a \) and continue looking for \( \beta' \)
Conflicts in SLR(1): Reduce-reduce

- **Reduce-reduce conflict** in state \( k \) on \( a \):
  - State \( k \) contains item \( A \rightarrow \beta \cdot \) and \( a \) is in FOLLOW(\( A \))
  - State \( k \) contains item \( A' \rightarrow \beta' \cdot \) and \( a \) is in FOLLOW(\( A' \))

  - The parser does not know whether it is at the end of production \( A \rightarrow \beta \) and thus should reduce by \( A \rightarrow \beta \cdot \), or
  - whether it is at the end of production \( A' \rightarrow \beta' \) and thus should reduce by \( A' \rightarrow \beta' \cdot \)

  - Usually, a reduce-reduce conflict indicates a serious problem with the grammar

Resolving Conflicts in SLR(1)

- In some cases, it makes sense to use a non-SLR(1) grammar (e.g., an ambiguous grammar)

  - Interestingly, we can still ensure desired behavior by choosing one of the conflicting actions

  - E.g., we can resolve a shift-reduce conflict by deterministically choosing the shift or the reduce

Exercise

- Construct the CFSM and SLR(1) parsing table for above grammar

  - This grammar is ambiguous and as expected we have shift-reduce conflict(s)

  - Resolve the conflict(s) so that + is left-associative

Question

- Recall the id-list grammars we saw earlier

  - “Top-down” grammar:
    - list \( \rightarrow \text{id} \ list\_tail \)
    - list\_tail \( \rightarrow \), \text{id} \ list\_tail | ;

  - “Bottom-up” grammar:
    - list \( \rightarrow \text{list\_prefix} \)
    - list\_prefix \( \rightarrow \text{list\_prefix} \), id
    - list\_prefix \( \rightarrow \text{id} \)

  - We saw that the top-down grammar is LL(1), but the bottom-up one is not LL(1)

  - How about SLR(1)?

    - Both grammars are SLR(1). However, the top-down one is not ideal for LR parsing. Why?

LR Parsing Variants

- LR(0), SLR(1), LALR(1), LR(1), … LR(k)

  - LR(0), SLR(1), LALR(1) are most practical

    - Use the same set of parsing states

    - Differ in the handling of “shift-reduce” states (states with both a shift item and a reduce item)

  - LR(0) uses 0 tokens of lookahead

    - Disallows shift-reduce states

    - Our running example is not LR(0)

  - E.g., state 2:
    - expr \( \rightarrow \text{term}^{*} \text{id} \)
    - term \( \rightarrow \text{term}^{*} \text{id} \)

LR Parsing Variants

- SLR(1) – uses 1 token of lookahead

  - Resolves (some) shift-reduce states by peeking at one token ahead

  - Adds labels “reduce by \( A \rightarrow \beta \) on FOLLOW(\( A \))” to states containing items \( A \rightarrow \beta \cdot \). The FOLLOW sets serve as filters

  - If after filtering by FOLLOW, there are no shift-reduce and no reduce-reduce conflicts, then grammar is SLR(1)

  - Is our running example SLR(1)?

    - Yes. Filtering by FOLLOW resolves the shift-reduce issue in state 2:
      - expr \( \rightarrow \text{term}^{*} \text{id} \)
      - term \( \rightarrow \text{term}^{*} \text{id} \)
Parsing Variants

- LALR(1)
  - Uses the same set of states as SLR(1)
  - Constructs local, context-sensitive FOLLOW sets and is able to avoid more conflicts
  - An efficiency hack
  - Most common parsers in practice
- LR(1)
  - Uses a different set of states
  - More states, in order to keep paths disjoint

Hierarchy of Grammar Classes

- LL(0) < LL(1) < LL(k), where k>1
- LR(0) < SLR(1) < LALR(1) < LR(1) < LR(k)
- Also, LL(k) < LR(k)

Question: SLR(1) parsers are more powerful than LL(1) ones? Why, what is the intuition?

Answer: LL(1) predicts production before it has seen the string derived from this production. SLR applies production (reduce by), after it has seen the entire string!

Group Exercise

Consider the grammar:

```
start-> expr
expr -> expr + expr | expr * expr | id
```

Construct the CFSM for this grammar

- First, construct the sets of LR items with transitions
- Second, add “reduce by” labels

Resolve the conflicts in such a way that the operators will behave “normally”:

- + and * are left-associative
- * has higher precedence than +

Next class

- We’ve concluded with parsing and programming language syntax!
- Next class: logic programming and Prolog. Read Chapter 12.