Announcements

- Quiz 2
- Hold on to paper, bring over at the end
- HW2 due tomorrow (extended 1 day)
  - We have office hours 4-7pm today
- Download SWI Prolog!
- HW3 will be posted tonight or tomorrow
- Prolog
- Due Friday, Sep 28 at 2pm
- Individual assignment

Last Class

- Bottom-up (LR) parsing
  - Handle
  - LR item
- Characteristic Finite State Machine (CFSM)
- SLR(1) parsing tables
- Conflicts in SLR(1)

Today's Lecture Outline

- The conclusion of bottom-up (LR) parsing
  - Conflicts in SLR(1)
  - LR parsing variants
- Logic Programming Concepts
- Prolog
  - Language constructs: facts, rules, queries

Programming Language Syntax

Bottom-up Parsing

Logic Programming and Prolog

CFSM: Sets of LR items with Transitions + "Reduce by" Labels
From CFSM to SLR(1) Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>id</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>1</td>
<td>2</td>
<td></td>
<td>expr</td>
<td>term</td>
</tr>
<tr>
<td>1</td>
<td>shift 4</td>
<td>accept</td>
<td></td>
<td></td>
<td></td>
<td></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>6</td>
<td></td>
<td></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>

Conflicts in SLR(1): Shift-reduce

- **Shift-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 $A' \rightarrow \alpha \cdot a \beta'$
  - 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 $A' \rightarrow \alpha \cdot a \beta'$ 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 $A \rightarrow \beta$, or whether it is at the end of production $A' \rightarrow \beta'$ and thus should reduce by $A' \rightarrow \beta'$
  - 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
  - Popular technique with parsing tools: use of directives to choose shift or reduce

Exercise

- **CFSM for Grammar**
  - start $\rightarrow$ expr
  - expr $\rightarrow$ expr $+$ expr $|$ id

- 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

- **CFSM for Grammar**
  - start $\rightarrow$ expr
  - expr $\rightarrow$ expr $+$ expr $|$ id

- There is a shift-reduce conflict on $+$ in state 4!
From CFSM to SLR(1) Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Expr</td>
<td>Shift 2</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>Shift 3</td>
</tr>
<tr>
<td>2</td>
<td>Expr</td>
<td>Reduce 2</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>Shift 2</td>
</tr>
<tr>
<td>4</td>
<td>Expr</td>
<td>Shift 3</td>
</tr>
</tbody>
</table>

White – action table
Blue – goto table

Shift-reduce conflict!

To force left associativity of + resolve conflict in favor of reduce!

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 → β on FOLLOW(A)” to states containing items A → [β]. 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 → term
 reduce by expr → term on $, +
```

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
    - Minor challenge: grammar in Problem 5 is LALR(1). Think about why.
  - 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)
- 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 what production to apply before it has seen the string. LR determines what production to reduce by, after it has seen the entire string!

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 +
Today’s Lecture Outline

- The conclusion of bottom-up (LR) parsing
  - Conflicts in SLR(1)
  - LR parsing variants

- Logic Programming Concepts
  - Prolog
    - Language constructs: facts, rules, queries

Moving on: Logic Programming

- Download and install SWI Prolog on laptop!

Logic Programming

- Logic programming
  - Logic programming is declarative programming
  - Logic program states what (logic), not how (control)

- Programmer declares axioms
  - In Prolog, facts and rules

- Programmer states a theorem, or a goal
  - In Prolog, a query

- Language implementation determines how to use the axioms to prove the goal

Logic Programming Concepts

- A Horn Clause is: $H \leftarrow B_1, B_2, \ldots, B_n$
  - Antecedents ($B_i$'s): conjunction of zero or more terms in predicate calculus; this is the body of the horn clause
  - Consequent ($H$): a term in predicate calculus

- Meaning of a Horn clause:
  - $H$ is true if $B_1, B_2, \ldots, B_n$ are all true
  - When the antecedents $B_i$ are all true, we deduce that consequent $H$ is true as well

- For example:
  - tiger(hobbes).
  - child(calvin).
  - likes(calvin,hobbes) $\leftarrow$ tiger(hobbes), child(calvin).
Logic Programming Concepts

Resolution Principle
If two Horn clauses
\[ A \leftarrow B_1, B_2, B_3, \ldots, B_m \]
\[ C \leftarrow D_1, D_2, D_3, \ldots, D_n \]
are such that \( A \) matches \( D_1 \),
then we can replace \( D_1 \) with \( B_1, B_2, B_3, \ldots, B_m \)
\[ C \leftarrow B_1, B_2, B_3, \ldots, B_m, D_2, D_3, \ldots, D_n \]
For example:
\[ C \leftarrow A, B \]
\[ D \leftarrow C \]
\[ D \leftarrow A, B, C \]

Horn Clauses in Prolog

- In Prolog, a Horn clause is written
  \[ h :- b_1, \ldots, b_n. \]
- Horn Clause is called clause
- Consequent is called goal or head
- Antecedents are called subgoals or tail

- Horn Clause with no tail is a fact
  - E.g., \texttt{rainy(seattle)}. Depends on no other conditions
- Horn Clause with a tail is a rule
  \[ \texttt{snowy(X) :- rainy(X),cold(X)}. \]

Horn Clauses in Prolog

- Clause is composed of terms
  - Constants
    - Number, e.g., 123, etc.
    - Atoms e.g., \texttt{seattle, rochester, rainy, foo}
    - In Prolog, atoms begin with a lower-case letter!
  - Variables
    - \( X, \text{Foo, My\_var} \)
    - In Prolog, variables must begin with upper-case letter!
  - Structures consists of an atom, called a functor
    and a list of arguments
    - \texttt{rainy(seattle), snowy(X)}

Prolog

- Program has a database of clauses i.e., facts and rules; the rules help derive more facts
- We add simple queries with constants, variables, conjunctions or disjunctions
  \[ \texttt{rainy(seattle)}. \]
  \[ \texttt{rainy(rochester)}. \]
  \[ \texttt{cold(rochester)}. \]
  \[ \texttt{snowy(X) :- rainy(X),cold(X)}. \]
  \[ ? – \texttt{rainy(C)}. \]
  \[ ? – \texttt{snowy(C)}. \]

Facts

| \texttt{likes(eve, pie)}. | \texttt{food(pie)}. |
| \texttt{likes(al, eve)}. | \texttt{food(apple)}. |
| \texttt{likes(eve, tom)}. | \texttt{person(tom)}. |
| \texttt{likes(eve, eve)}. |

The combination of the functor and its arity (i.e., its number of arguments) is called a predicate.
Queries

likes(eve, pie).  food(pie).
likes(al, eve).  food(apple).
likes(eve, tom).  person(tom).
likes(eve, eve).  variable

?-likes(al, eve).  true.  answer
?-likes(al, pie).  false.
?-likes(eve, al).  false.

?-likes(al, Who).  \( \text{Who} = \text{eve} \).

?-likes(al, pie), answer with variable binding

?-likes(eve, W), variable binding

force search for more answers

Harder Queries

likes(eve, pie).  food(pie).
likes(al, eve).  food(apple).
likes(eve, tom).  person(tom).
likes(eve, eve).

?-likes(al, V), likes(eve, V).  \(\text{V} = \text{pie} ; \text{V} = \text{tom} ; \text{V} = \text{eve} \).

?-likes(eve, W), person(W).  \(\text{W} = \text{tom} ; \text{W} = \text{eve} \).

?-likes(eve, W), food(V).  \(\text{V} = \text{pie} ; \text{V} = \text{apple} \).

Prolog gives us the answer precisely in this order: first \(\text{W} = \text{pie} \) then \(\text{W} = \text{tom} \) and finally \(\text{W} = \text{eve} \). Can you guess why?

Rules

Add a rule to the database:

```
rule1 :- likes(eve, V), person(V).
```

?-rule1.  true

?-rule2(H).  \(\text{H} = \text{tom} \).

?-rule2(pie).  false.

rule1 and rule2 are just like any other predicate!
Queen Victoria Example

- male(albert).
- male(edward).
- female(alice).
- female(victoria).
- parents(edward,victoria,albert).
- parents(alice,victoria,albert).

?- (family). Loads file family.pl
true.
?- male(albert). A query
true.
?- male(alice).
false.
?- parents(edward,victoria,albert).
true.
?- parents(bullwinkle,victoria,albert).
false.

?- female(X). a query
X = alice ; asks for more answers
X = victoria.
- Variable X has been unified to all possible values that make female(X) true.
- Variables are upper-case, functors (predicates and constants) are lower-case!

Queen Victoria Example

- sister_of(X,Y) :- female(X),parents(X,M,F),
  parents(Y,M,F).

?- sister_of(alice, Y).
Y = edward <enter>: not asking for more answers
?- sister_of(alice, victoria).
false.

Next class

- We’ll continue with Prolog!
- Download SWI Prolog and run through the lecture examples!!!