Announcements

- HW2 due Thursday, 2/18 in class and in Homework Server
  - Note on Problem 3: Show means prove
  - Note on 4&7: Will cover material today
- Download SWI Prolog
- Prolog Project will be next Thursday, 2/18

Last Class

- Bottom-up (LR) parsing
  - Model of the LR parser
  - LR Items
  - Characteristic Finite State Machine (CFSM)

Today’s Lecture Outline

- Bottom-up (LR) parsing
  - Characteristic Finite State Machine (CFSM)
  - From CFSM to SLR(1) parsing table
  - Conflicts in SLR(1)
  - LR parsing variants
- Logic Programming and Prolog
  - Logic programming
  - Logic programming concepts
  - Prolog constructs: facts, rules, queries

Programming Language Syntax

Bottom-up Parsing

Logic Programming and Prolog

Read: finish Scott, Chapter 2.3.3, start Scott, Chapter 11.1, 11.2 (beginning)

Model of the LR parser

Characteristic Finite State Machine (CFSM)

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

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, in state 2 we have:

$expr \rightarrow term*id$

reduce by $expr \rightarrow term$ on $$, +.

From CFSM to SLR(1) Parsing Table

1. expr -> expr + term
2. expr -> term
3. term -> term * id
4. term -> id

<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>1</td>
<td></td>
<td></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>6</td>
<td></td>
<td></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>

SLR(1) Parsing Table

Input: An augmented grammar $G$ (G with starting production start -> ...)
Output: Functions action and goto for $G$

Construct $C = (I, P, I_0)$ the collection of sets of LR items with transitions
State $i$ is constructed from $i$. The parsing actions for state $i$ are
a) If item $A \rightarrow \alpha$ is in $i$, and there is a transition from $i$ to $j$ on $\alpha$, then set action[$a$] to "shift".
b) If item $A \rightarrow \alpha$ is in $i$, then set action[$a$] to "reduce by $A \rightarrow \alpha$" for all terminals $a$ in FOLLOW(A).
c) If start -> $\ldots \cdot i$ is in $i$, then set action[0] to "accept".

The goto transition for state $i$ are constructed for all nonterminals $A$ using the rule: If there is a transition from $i$ to $j$ on $A$, set goto[1][A] = j

If the table contains no multiply-defined entries, the grammar is said to be SLR(1)
Conflicts in SLR(1): Shift-reduce

- **Shift-reduce conflict** in state $k$ on $a$:
  - State $k$ contains item $A \rightarrow \beta$ and $a$ is in FOLLOW($A$)
  - State $k$ contains item $A' \rightarrow \alpha \beta'$
  - 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 $A' \rightarrow \alpha \beta'$ 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 $A \rightarrow \beta$ and $a$ is in FOLLOW($A$)
  - State $k$ contains item $A' \rightarrow \beta'$ 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 $A \rightarrow \beta$, 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

Exercise

- **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

Question

- Recall the **id**-list grammars we saw earlier
  - “Top-down” grammar:
    - **list $\rightarrow$ id list_tail**
    - **list_tail $\rightarrow$ , id list_tail | ;**
  - “Bottom-up” grammar:
    - **list $\rightarrow$ list_prefix ;**
    - **list_prefix $\rightarrow$ list_prefix , id**
    - **list_prefix $\rightarrow$ id**

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

Lecture Outline

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

- Logic Programming and Prolog
  - Logic programming
  - Logic programming concepts
  - Prolog constructs: facts, rules, queries
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 → term
term → term•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 → 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
term → term•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)
- 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?

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 +

Lecture Outline

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

- Logic Programming and Prolog
  - Logic programming
  - 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

Prolog Program:

```prolog
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X):-rainy(X),cold(X).
```

Java Program:

```java
HashSet rainy = new HashSet();
rainy.add("seattle");
rainy.add("rochester");
cold.add("rochester");
```

```java
for (Iterator it=rainy.iterator(); it.hasNext();) {
    String city = (String) it.next();
    if (cold.contains(city)) snowy.add(city);
}
```

```java
for (Iterator it=snowy.iterator(); it.hasNext();) {
    System.out.println((String) it.next());
}
```

Logic Programming Concepts

- A Horn Clause is: \( H \leftarrow B_1, B_2, ..., B_n \)
  - Antecedents \( B \)'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 \( B_1, B_2, ..., B_n \) are all true
  - When the antecedents \( B \) 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).

Resolution Principle

If two Horn clauses

\[
A \leftarrow B_1, B_2, B_3, ..., B_m
\]
\[
C \leftarrow D_1, D_2, D_3, ..., D_n
\]

are such that \( A \) matches \( D_1 \), then we can replace \( D \) with \( B_1, B_2, B_3, ..., B_m \)

\[
C \leftarrow B_1, B_2, B_3, ..., B_m, D_2, D_3, ..., D_n
\]

For example:

\[
C \leftarrow A, B
\]

\[
D \leftarrow C
\]
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. $\text{tiger(hobbes)}$. Depends on no other conditions
- Horn Clause with a tail is a rule

\[ \text{likes(calvin, X)} : - \text{tiger(calvin)}, \text{child(X)}. \]

Horn Clauses in Prolog

- Clause is composed of terms
  - Constants
    - Number, e.g., 123, etc.
    - Atoms e.g., \text{calvin}, \text{hobbes}, \text{foo}
    - In Prolog, atoms begin with a lower-case letter!
  - Variables
    - \text{X}, \text{Foo}, \text{My_var}
    - In Prolog, variables must begin with upper-case letter!
  - Structures
    - Consist of an atom, called a functor and a list of arguments: e.g., $\text{tiger(hobbes)}, \text{likes(calvin, X)}$

Prolog

- Program is a database of clauses i.e., facts and rules; the rules help derive more facts
- We add simple queries with constants and variables ("binding"). conjunctions and disjunctions

\begin{align*}
\text{rainy(seattle)}.
\text{rainy(rochester)}.
\text{cold(rochester)}.
\text{snowy(X)} : - \text{rainy(X)}, \text{cold(X)}.
\end{align*}

? - rainy(C).
? - snowy(C).

Facts

\begin{align*}
\text{likes(eve, pie).} & \quad \text{food(pie).}
\text{likes(al, eve).} & \quad \text{food(apple).}
\text{likes(eve, tom).} & \quad \text{person(tom).}
\text{likes(eve, eve).} & \quad \text{and}
\end{align*}

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

Queries

\begin{align*}
\text{likes(eve, pie).} & \quad \text{food(pie).}
\text{likes(al, eve).} & \quad \text{food(apple).}
\text{likes(eve, tom).} & \quad \text{person(tom).}
\text{likes(eve, eve).} & \quad \text{and}
\end{align*}

\begin{align*}
?-\text{likes(al, eve)}. & \quad \text{true.} \quad \text{answer} \\
?-\text{likes(al, pie)}. & \quad \text{false.} \\
?-\text{likes(al, eve)}. & \quad \text{false.} \\
?-\text{likes(eve, pie)}. & \quad \text{true.} \quad \text{answer with variable binding} \\
?-\text{likes(eve, tom)}. & \quad \text{false.} \\
?-\text{likes(eve, eve)}. & \quad \text{false.} \\
\end{align*}

Harder Queries

\begin{align*}
\text{likes(eve, pie).} & \quad \text{food(pie).}
\text{likes(al, eve).} & \quad \text{food(apple).}
\text{likes(eve, tom).} & \quad \text{person(tom).}
\text{likes(eve, eve).} & \quad \text{and}
\end{align*}

\begin{align*}
?-\text{likes(al, V)} , \text{likes(eve, V)}. \\
?-\text{likes(eve, W)}, \text{person(W)}. \\
?-\text{likes(A, B)}. \\
?-\text{likes(D, E)}. \\
\end{align*}

}\end{align*}
Harder Queries

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

?-likes(eve, W), likes(W, V).

?-likes(eve, W), person(W), food(V).

?-likes(eve, V), (person(V); food(V)).

Rules

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

Add a rule to the database:

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

?-rule1.
true

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.

?- female(X).
X = alice ;
; asks for more answers
X = victoria.
Variable X has been unified to all possible values
that make female(X) true.

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.
Exercises

- Download and install SWI Prolog!
- Execute “Queen Victoria Example”
  - Load all facts and rules in a file family.pl
  - Execute a few queries

Next Class

- Read Chapter 11.2.1-4
- Bring your laptops. We will be doing in-class exercises in Prolog.