Announcements

Reminder: Office hours

- Uli Kremer, CoRE 318, Wednesdays 10:00am - 11:00am
- Liu Liu, CoRE 334, Mondays 3:00pm - 4:00pm, and Fridays noon - 1:00pm

First homework has been posted. Please submit by Wednesday, February 10 (extensino), if you want some of the problems to be reviewed (no grades!). Submission is done through sakai as a pdf, or you can a hardcopy in before class.

Second homework will be posted by Monday, February 8.
Homework Problem 4.6: Recursive Descent Parser

ILOC – intermediate language (quadrupels)

Compiler part

- load-store architecture
- register-transfer language
- three-address code
- explicit loads and stores

<table>
<thead>
<tr>
<th>instr. format</th>
<th>description</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>memory instructions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loadI ( c \Rightarrow R_x )</td>
<td>load constant value ( c ) into register ( R_x )</td>
<td>( R_x \leftarrow c )</td>
</tr>
<tr>
<td>loadAI ( R_x, c \Rightarrow R_y )</td>
<td>load value of ( \text{MEM}(R_x + c) ) into ( R_y )</td>
<td>( R_y \leftarrow \text{MEM}(R_x + c) )</td>
</tr>
<tr>
<td>storeAI ( R_x \Rightarrow R_y, c )</td>
<td>store value in ( R_x ) into ( \text{MEM}(R_y + c) )</td>
<td>( \text{MEM}(R_y + c) \leftarrow R_x )</td>
</tr>
<tr>
<td><strong>arithmetic instructions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add ( R_x, R_y \Rightarrow R_z )</td>
<td>add contents of registers ( R_x ) and ( R_y ), and store result into register ( R_z )</td>
<td>( R_z \leftarrow R_x + R_y )</td>
</tr>
<tr>
<td>sub ( R_x, R_y \Rightarrow R_z )</td>
<td>subtract contents of register ( R_x ) from register ( R_y ), and store result into register ( R_z )</td>
<td>( R_z \leftarrow R_x - R_y )</td>
</tr>
<tr>
<td>mult ( R_x, R_y \Rightarrow R_z )</td>
<td>multiply contents of registers ( R_x ) and ( R_y ), and store result into register ( R_z )</td>
<td>( R_z \leftarrow R_x * R_y )</td>
</tr>
<tr>
<td>div ( R_x, R_y \Rightarrow R_z )</td>
<td>divide contents of registers ( R_x ) and ( R_y ), and store result into register ( R_z )</td>
<td>( R_z \leftarrow R_x / R_y )</td>
</tr>
<tr>
<td><strong>I/O instruction</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>output ( c )</td>
<td>write value of ( \text{MEM}(c) ) to standard output</td>
<td>( \text{print}(\ \text{MEM}(c)) )</td>
</tr>
</tbody>
</table>
Homework Problem 4.6 (cont.)

- **Code shape**: register-register model ⇒ freshly created values have their own virtual register; no virtual register is ever redefined!

- **Use of base register**: the register $r_0$ is assumed to hold the base address into our memory space. All local variables are referenced relative to this base address. This corresponds to a code shape model that uses stack frames to store local variables. Here, $r_0$ would contain the frame pointer. For the homework, $r_0$ is assumed to be 1024.

The homework does not use memory objects. However, I/O needs to be done through memory (output instruction).

ILOC will be used for the first programming project.
Homework Problem 4.6 (cont.)

You can find **executables** of the sample solutions of the compiler (**compile-sol**) and interpreter (**interpret-sol**) on the ilab cluster in directory

```
~/uli/cs515/homeworks/hw1/students
```

This directory also contains a program that can “execute” ILOC code (**sim**). You can use this simulator to check for correctness of your compiler generated ILOC code. **compiler.c** and **interpreter.c** are C versions of the compiler and interpreter that you can expand for your own solution. However, you may also use C++ or Java. This is up to you.
Review: Bottom-up parsers

- start at the leaves and fill in
- construct rightmost derivation in reverse
- find the next right-hand side of a production (handle) such that its replacement by left-hand side nonterminal will yield previous right-sentential form
- as input is consumed, change state to encode possibilities (recognize valid prefixes); if handle is found, REDUCE, otherwise SHIFT (or ERROR)

\[ S \Rightarrow_{*_{rm}} \alpha By \Rightarrow_{rm} \alpha \gamma y \Rightarrow_{*_{rm}} xy \]

- LR parsing: Reads input from left to right and constructs rightmost derivation in reverse


Review: Handles

We trying to find a substring $\alpha$ of the current right-sentential form where:

- $\alpha$ matches some production $A ::= \alpha$
- reducing $\alpha$ to $A$ is one step in the reverse of a rightmost derivation.

We will call such a string a *handle*.

Formally,

- a *handle* of a right-sentential form $\gamma$ is a production $A ::= \beta$ and a position in $\gamma$ where $\beta$ may be found. Convention: position specifies the right end of handle.
- If $(A ::= \beta, k)$ is a handle, then replacing the $\beta$ in $\gamma$ at position $k$ with $A$ produces the previous right-sentential form in a rightmost derivation of $\gamma$. 
Handles

Provable fact:

The substring to the right of a handle contains only terminal symbols.

Proof: Follows from the fact that all $\gamma_i$ are right-sentential forms.

Corollary

The right end of a handle is to the right of the previously reduced variable.
Shift-reduce parsing

One scheme to implement a handle-pruning, bottom-up parser is called a *shift-reduce* parser.

Shift-reduce parsers use a *stack* and an *input buffer*

1. initialize stack with $\$

2. Repeat until the top of the stack is the goal symbol and the input token is *eof*
   
   a) *find the handle*
      
      if we don’t have a handle on top of the stack, *shift* an input symbol onto the stack
   
   b) *prune the handle*
      
      if we have a handle $(A ::= \beta, k)$ on top of the stack, *reduce*
      
      i) pop $|\beta|$ symbols off the stack
      
      ii) push $A$ onto the stack
Example

Left-recursive expression grammar

(*original form*, before left recursion removal and factoring
→ our “larger” example LL(1) grammar - lecture 2, page 17)

1  ⟨goal⟩ ::= ⟨expr⟩
2  ⟨expr⟩ ::= ⟨expr⟩ + ⟨term⟩
3          | ⟨expr⟩ - ⟨term⟩
4          | ⟨term⟩
5  ⟨term⟩ ::= ⟨term⟩ * ⟨factor⟩
6          | ⟨term⟩ / ⟨factor⟩
7          | ⟨factor⟩
8  ⟨factor⟩ ::= num
9          | id
“$x - 2 * y$”

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle expr\rangle$</td>
<td>$id - num * id$</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$- num * id$</td>
<td>9,1</td>
<td>reduce 9</td>
</tr>
<tr>
<td>$\langle term\rangle$</td>
<td>$- num * id$</td>
<td>7,1</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$\langle term\rangle$</td>
<td>$- num * id$</td>
<td>4,1</td>
<td>reduce 4</td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$- num * id$</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$num * id$</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$* id$</td>
<td>8,3</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$* id$</td>
<td>7,3</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$* id$</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$id$</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$9,5$</td>
<td>reduce 9</td>
<td></td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$5,5$</td>
<td>reduce 5</td>
<td></td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$3,3$</td>
<td>reduce 3</td>
<td></td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$1,1$</td>
<td>reduce 1</td>
<td></td>
</tr>
<tr>
<td>$\langle expr\rangle$</td>
<td>$none$</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until top of stack is the right end of a handle

2. Find the left end of the handle and reduce

5 shifts + 9 reduces + 1 accept
Viable prefix

A viable prefix is

1. a prefix of a right-sentential form that does not continue past the right end of the handle of that sentential form†, or

2. a prefix of a right-sentential form that can appear on the stack of a shift-reduce parser.

It is always possible to add terminals onto the end of a viable prefix to obtain a right-sentential form.

As long as the prefix represented by the stack is viable, the parser has not seen a detectable error.

† If the grammar is unambiguous, there is a unique rightmost handle. *LR(1)* grammars are unambiguous.
Shift-reduce parsing

Grammars that are often used to construct shift-reduce parsers:

- operator grammars (will not discuss here → Aho, Sethi, Ullman p.203)
- LR(1) grammars
  - canonical LR(1) grammars
  - simple LR(1) grammars ($SLR(1)$)
  - lookahead LR(1) grammars ($LALR(1)$)

Grammars use different methods or levels of ”context” information to detect handle.

$LR(1)$, $SLR(1)$ and $LALR(1)$ grammars use finite automata ($NFA$s or $DFA$s) together with a PDA to recognize viable prefixes and store ”context” information.
**LR**\((k)\) **grammars**

Informally, we say that a grammar \(G\) is LR\((k)\) if,

given a rightmost derivation

\[
S' = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \cdots \Rightarrow_{rm} \gamma_n = w,
\]

we can, for each right-sentential form in the derivation,

1. isolate the handle of each right-sentential form,

   and

2. determine the production by which to reduce

by scanning \(\gamma_i\) from left to right, going at most \(k\) symbols beyond the right end of the handle of \(\gamma_i\).
Table-driven $LR$ parsing

A table-driven $LR(k)$ parser looks like

Stack two items per state: \textit{state} and \textit{symbol}
Why study LR(1) grammars?

- All context-free, deterministic languages have an LR(1) grammar. Therefore LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers.
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- Efficient shift-reduce parsers can be implemented for LR(1) grammars — time proportional to $tokens + reductions$
- Easy to build since table construction can be automated
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- Everyone’s favorite parser ($EFP$) — tools widely available (example: $yacc$).
LR(1) parsing

The skeleton parser:

```python
token = next_token()
repeat forever
  s = state on top of stack
  if action[s, token] = "shift $s_i$" then
    push token
    push state $s_i$
    token = next_token()
  else if action[s, token] = "reduce $A := \beta$" then
    pop 2 * $|\beta|$ symbols
    s = state on top of stack
    push $A$
    push goto[s, $A$]
  else if action[s, token] = "accept" then
    return
  else error()
```

This takes $k$ shifts, $l$ reduces, and 1 accept, where $k$ is the length of the input string and $l$ is the length of the reverse rightmost derivation.

Note: See Figure 4.36, Aho, Lam, Sethi, and Ullman
**LR(0) parsing: our example**

<table>
<thead>
<tr>
<th>Stack (without states)</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>- num * id</td>
<td>9,1</td>
<td>reduce 9</td>
</tr>
<tr>
<td>$\langle \text{factor} \rangle</td>
<td>- num * id</td>
<td>7,1</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$\langle \text{term} \rangle</td>
<td>- num * id</td>
<td>4,1</td>
<td>reduce 4</td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle</td>
<td>- num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle$ -</td>
<td>num * id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle$ - num</td>
<td>* id</td>
<td>8,3</td>
<td>reduce 8</td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle$ - $\langle \text{factor} \rangle$</td>
<td>* id</td>
<td>7,3</td>
<td>reduce 7</td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle$ - $\langle \text{term} \rangle$</td>
<td>* id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle$ - $\langle \text{term} \rangle$ *</td>
<td>id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle$ - $\langle \text{term} \rangle$ * id</td>
<td>9,5</td>
<td>reduce 9</td>
<td></td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle$ - $\langle \text{term} \rangle$ * $\langle \text{factor} \rangle$</td>
<td>5,5</td>
<td>reduce 5</td>
<td></td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle$ - $\langle \text{term} \rangle$</td>
<td>3,3</td>
<td>reduce 3</td>
<td></td>
</tr>
<tr>
<td>$\langle \text{expr} \rangle$</td>
<td>1,1</td>
<td>reduce 1</td>
<td></td>
</tr>
<tr>
<td>$\langle \text{goal} \rangle$</td>
<td>none</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

The corresponding grammar and language is not LR(0).

**Theorem:** A language L has an LR(0) grammar iff

- L is deterministic
- no proper prefix of a word in L is in L (prefix property)
**LR parsing**

There are three commonly used algorithms to build tables for an “LR” parser:

1. **SLR(1)** = $LR(0) + \text{FOLLOW}$
   - smallest class of grammars
   - smallest tables (number of states)
   - simple, fast construction

2. **LR(1)**
   - full set of LR(1) grammars
   - largest tables (number of states)
   - slow, large construction

3. **LALR(1)**
   - intermediate sized set of grammars
   - same number of states as SLR(1)
   - canonical construction is slow and large
   - better construction techniques exist

An $LR(1)$ parser for either ALGOL or PASCAL has several thousand states, while an $SLR(1)$ or $LALR(1)$ parser for the same language may have several hundred states.
**$SLR(1)$ parsing**

**Viable prefix** of a right-sentential form:

- contains both terminals and nonterminals
- can be recognized using a DFA

Building a $SLR$ parser

- construct DFA for recognizing handles
- augment with FOLLOW to disambiguate actions

States in the DFA are sets of $LR(0)$ items (subset construction)

**Note:** An “augmented/extended grammar” (ECFG) is one where the start symbol appears only on the *lhs* of productions. For the rest of LR parsing, we will assume the grammar is augmented with a production $S' ::= S'$
**LR(0) items**

An LR(0) item is a string $[\alpha]$, where

$\alpha$ is a production from $G$ with a $\bullet$ at some position in the rhs

The $\bullet$ indicates how much of an item we have seen at a given state in the parsing process.

$[A := \bullet X Y Z]$ indicates that the parser is looking for a string that can be derived from $X Y Z$

$[A := X Y \bullet Z]$ indicates that the parser has seen a string derived from $X Y$ and is looking for one derivable from $Z$

**LR(0) Items** (no lookahead)

$A ::= X Y Z$ generates 4 LR(0) items.

1. $[A ::= \bullet X Y Z]$
2. $[A ::= X \bullet Y Z]$
3. $[A ::= X Y \bullet Z]$
4. $[A ::= X Y Z \bullet]$

198:515, Spring 2016
Canonical *LR(0)* items

The *SLR(1)* table construction algorithm uses a specific set of sets of *LR(0)* items.

These sets are called the *canonical collection of sets of LR(0) items* for a grammar *G*.

The canonical collection corresponds to the set of states of the DFA that recognizes viable prefixes. Each state is the set of valid LR(0) items at a particular point in the parse.

The LR(0) item \[ A ::= \beta_1 \bullet \beta_2 \] is *valid* for a viable prefix \( \alpha \beta_1 \) if there is a derivation 
\[
S' \Rightarrow_{rm}^* \alpha Aw \Rightarrow_{rm} \alpha_1 \beta_2 w.
\]
In general, an item will be valid for many viable prefixes.
SLR(1) parser example

The Grammar

1 \[ E ::= T + E \]
2 \[ E :: T \]
3 \[ T ::= \text{id} \]

The Augmented Grammar

0 \[ S' ::= E \]
1 \[ E ::= T + E \]
2 \[ E :: T \]
3 \[ T ::= \text{id} \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ S' ]</td>
<td>{ \text{id} }</td>
<td>{ \text{eof} }</td>
</tr>
<tr>
<td>[ E ]</td>
<td>{ \text{id} }</td>
<td>{ \text{eof} }</td>
</tr>
<tr>
<td>[ T ]</td>
<td>{ \text{id} }</td>
<td>{ +, \text{eof} }</td>
</tr>
</tbody>
</table>
Canonical Collection of LR(0) items

To construct the canonical collection we need two functions:

- closure($I$)
  
  if \([A ::= \alpha \bullet B\beta] \in I_j\), then, in state \(j\), the parser might next see a string derivable from \(B\beta\)  
  
  \(\Rightarrow\) to form its closure, add all items of the form  
  
  \([B ::= \bullet \gamma] \in G\)

- GOTO($I$, $X$)
  
  If $I$ is the set of items that are valid for some viable prefix $\gamma$, then GOTO($I$, $X$) is the set of items that are valid for the viable prefix $\gamma X$. 
Closure($I$)

Given an item $[A ::= \alpha \cdot B\beta]$, its closure contains the item and any other items that can generate legal substrings to follow $\alpha$.

Thus, if the parser has viable prefix $\alpha$ on its stack, the input should reduce to $B\beta$ (or $\gamma$ for some other item $[B ::= \bullet \gamma]$ in the closure).

To compute closure($I$)

```
function closure(I)
    repeat
        new_item ← false
        for each item $[A ::= \alpha \cdot B\beta] \in I$,
            each production $B ::= \gamma \in G'$
            if $[B ::= \bullet \gamma] \notin I$ then
                add $[B ::= \bullet \gamma]$ to I
                new_item ← true
            endif
        until (new_item = false)
    return I
```
Let $I$ be a set of $LR(0)$ items and $X$ be a grammar symbol.

Then, $\text{GOTO}(I, X)$ is the closure of the set of all items $[A ::= \alpha X \bullet \beta]$ such that $[A ::= \alpha \bullet X \beta] \in I$.

If $I$ is the set of valid items for some viable prefix $\gamma$, then $\text{goto}(I, X)$ is the set of valid items for the viable prefix $\gamma X$.

$\text{goto}(I, X)$ represents state after recognizing $X$ in state $I$.

To compute $\text{goto}(I, X)$

```
function goto(I, X)
    J ← set of items $[A ::= \alpha X \bullet \beta]$
    such that $[A ::= \alpha \bullet X \beta] \in I$
    $J' ← \text{closure}(J)$
    return $J'$
```

Collection of sets of $LR(0)$ items

We start the construction of the collection of sets of $LR(0)$ items with the item $[S' ::= \bullet S]$, where

$S'$ is the start symbol of the augmented grammar $G'$
$S$ is the start symbol of $G$

To compute the collection of sets of LR(0) items

```
procedure items($G''$)
    $S_0 \leftarrow \text{closure}([S' ::= \bullet S])$
    Items $\leftarrow \{ S_0 \}$
    ToDo $\leftarrow \{ S_0 \}$
    while ToDo not empty do
        remove $S_i$ from ToDo
        for each grammar symbol $X$ do
            $S_{new} \leftarrow \text{goto}(S_i, X)$
            if $S_{new}$ is a new state then
                Items $\leftarrow$ Items $\cup \{ S_{new} \}$
                ToDo $\leftarrow$ ToDo $\cup \{ S_{new} \}$
            endif
        endfor
    endwhile
    return Items
```
**LR(0) machines**

**LR(0) DFA**

- states – canonical sets of LR(0) items
- edges – goto transitions
- recognizes handles and viable prefixes using stack
- no lookahead

Reducing a handle (rhs of production) to a nonterminal can be viewed as:

- returning to state at beginning of handle
- making transition on nonterminal for this state

To return to state at beginning of the handle, we must use the stack to store the state!
**SLR(1) tables**

**SLR(1) parser**

- augment *LR(0)* machine
- add FOLLOW information using one token of lookahead
- encoded as **ACTION, GOTO** tables

**ACTION table**

- for each [state, lookahead] pair
- have we reached end of handle?
- if not, shift
- if at end of handle, reduce
- may also accept or error
- use lookahead to guide decision

**GOTO table**

- for each [state, nonterminal] pair
- pick state to go to after reduction
SLR(1) table construction

The Algorithm

1. construct the collection of sets of LR(0) items for $G'$.
2. State $i$ of the parser is constructed from $I_i$.
   
   (a) if $[A ::= \alpha \bullet a\beta] \in I_i$ and $\text{goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to “shift $j$”. ($a$ must be a terminal)
   
   (b) if $[A ::= \alpha\bullet] \in I_i$, then set $\text{ACTION}[i, a]$ to “reduce $A ::= \alpha$” for all $a$ in $\text{FOLLOW}(A)$.
   
   (c) if $[S' ::= S\bullet] \in I_i$, then set $\text{ACTION}[i, \text{eof}]$ to “accept”.

3. If $\text{goto}(I_i, A) = I_j$, then set $\text{GOTO}[i, A]$ to $j$.

4. All other entries in $\text{ACTION}$ and $\text{GOTO}$ are set to “error”

5. The initial state of the parser is the state constructed from the set containing the item $[S' ::= \bullet S]$. 
SLR(1) parser example

DFA for handles / viable prefixes based on LR(0) canonical collection
Example LR(0) states

$S_0$:  
\[
\begin{align*}
S' & ::= \bullet E, \\
E & ::= \bullet T + E, \\
E & ::= \bullet T, \\
T & ::= \bullet \text{id}
\end{align*}
\]

$S_1$:  
\[
S' ::= E \bullet
\]

$S_2$:  
\[
\begin{align*}
E & ::= T \bullet + E, \\
E & ::= T \bullet
\end{align*}
\]

$S_3$:  
\[
T ::= \text{id} \bullet
\]

$S_4$:  
\[
\begin{align*}
E & ::= T + \bullet E, \\
E & ::= \bullet T + E, \\
E & ::= \bullet T, \\
T & ::= \bullet \text{id}
\end{align*}
\]

$S_5$:  
\[
E ::= T + E \bullet
\]
Example GOTO function

Start

\[ S_0 \leftarrow \text{closure} ( \{ [ S ::= \bullet E ] \} ) \]

Iteration 1

\[ \text{goto}(S_0, E) = S_1 \]
\[ \text{goto}(S_0, T) = S_2 \]
\[ \text{goto}(S_0, \text{id}) = S_3 \]

Iteration 2

\[ \text{goto}(S_2, +) = S_4 \]

Iteration 3

\[ \text{goto}(S_4, \text{id}) = S_3 \]
\[ \text{goto}(S_4, E) = S_5 \]
\[ \text{goto}(S_4, T) = S_2 \]
Example ACTION and GOTO tables

<table>
<thead>
<tr>
<th></th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>eof</td>
</tr>
<tr>
<td>$S_0$</td>
<td>shift 3</td>
<td>—</td>
</tr>
<tr>
<td>$S_1$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$S_2$</td>
<td>—</td>
<td>shift 4</td>
</tr>
<tr>
<td>$S_3$</td>
<td>—</td>
<td>reduce 3</td>
</tr>
<tr>
<td>$S_4$</td>
<td>shift 3</td>
<td>—</td>
</tr>
<tr>
<td>$S_5$</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>id + id eof</td>
<td>shift 3</td>
</tr>
<tr>
<td>$0$ id 3</td>
<td>+ id eof</td>
<td>reduce 3 (T ::= id)</td>
</tr>
<tr>
<td>$0$ T 2</td>
<td>+ id eof</td>
<td>shift 4</td>
</tr>
<tr>
<td>$0$ T 2 + 4 id 3</td>
<td>id eof</td>
<td>shift 3</td>
</tr>
<tr>
<td>$0$ T 2 + 4 id 3</td>
<td>eof</td>
<td>reduce 3 (T ::= id)</td>
</tr>
<tr>
<td>$0$ T 2 + 4 T 2</td>
<td>eof</td>
<td>reduce 2 (E ::= T)</td>
</tr>
<tr>
<td>$0$ T 2 + 4 E 5</td>
<td>eof</td>
<td>reduce 1 (E ::= T + E)</td>
</tr>
<tr>
<td>$0$ E 1</td>
<td>eof</td>
<td>accept</td>
</tr>
</tbody>
</table>
What can go wrong?

Example: A simple grammar

1. \( S' ::= S \)
2. \( S ::= L = R \)
3. \( S ::= R \)
4. \( L ::= *R \)
5. \( L ::= \text{id} \)
6. \( R ::= L \)

Canonical LR(0) collection

\[ I_0 : \{ [S' ::= \bullet S], [S ::= \bullet L = R], [S ::= \bullet R], [L ::= \bullet * R], [L ::= \bullet \text{id}], [R ::= \bullet L] \} \]

\[ I_1 : \{ [S' ::= S\bullet] \} \]

\[ I_2 : \{ [S ::= L\bullet = R], [R ::= L\bullet] \} \]

\[ I_3 : \{ [S ::= R\bullet] \} \]

\[ I_4 : \{ [L ::= *\bullet R], [R ::= \bullet L], [L ::= \bullet * R], [L ::= \bullet \text{id}] \} \]

\[ I_5 : \{ [L ::= \text{id}\bullet] \} \]

\[ I_6 : \{ [S ::= L = \bullet R], [R ::= \bullet L], [L ::= \bullet * R], [L ::= \bullet \text{id}] \} \]

\[ I_7 : \{ [L ::= *R\bullet] \} \]

\[ I_8 : \{ [R ::= L\bullet] \} \]

\[ I_9 : \{ [S ::= L = R\bullet] \} \]
Consider the set of items $I_2$. The action table is defined as follows:

\[
[S ::= L \bullet = R] \text{ implies } \text{ACTION}[2, =] = "shift 6"
\]

\[
[R ::= L \bullet] \text{ implies } \text{ACTION}[2, =] = "reduce 6"
\]

Due to multiple definitions of the position in the action table, the grammar is not $SLR(1)$. 

---

**SLR(1) table construction**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S'$</td>
<td>${ \text{id}, \ast }$</td>
<td>${ \text{eof} }$</td>
</tr>
<tr>
<td>$S$</td>
<td>${ \text{id}, \ast }$</td>
<td>${ \text{eof} }$</td>
</tr>
<tr>
<td>$L$</td>
<td>${ \text{id}, \ast }$</td>
<td>${ =, \text{eof} }$</td>
</tr>
<tr>
<td>$R$</td>
<td>${ \text{id}, \ast }$</td>
<td>${ =, \text{eof} }$</td>
</tr>
</tbody>
</table>
What can go wrong?

Two cases arise

**shift/reduce**

This is called a *shift/reduce* conflict. In general, it indicates an ambiguous construct in the grammar.

- can modify the grammar to eliminate it
- can resolve in favor of shifting

**classic example:** dangling else

**reduce/reduce**

This is called a *reduce/reduce* conflict. Again, it indicates an ambiguous construct in the grammar.

- often, no simple resolution
- parse a nearby language

**classic example:** PL/I call and subscript
**LR(1)**

**SLR(1)** parsers may not be able to parse some **LR** grammars.

Problem is that lookahead information is added to **LR(0)** parser at the end of construction.

We can get more powerful parser by keeping track of lookahead information in the states of the parser.

*If, in a single left-to-right scan, we can construct a reverse rightmost derivation, while using at most a single token lookahead to resolve ambiguities, then the grammar is **LR(1)***
**LR\((k)\) items**

The table construction algorithms use \(LR(k)\) items to represent the set of possible states in a parse.

An \(LR(k)\) item is a pair \([\alpha, \beta]\), where

- \(\alpha\) is a production from \(G\) with a \(\bullet\) at some position in the rhs
- \(\beta\) is a lookahead string containing \(k\) symbols (terminals or \texttt{eof})

What about \(LR(1)\) items?

- example \(LR(1)\) item: \([A ::= X \bullet YZ, a]\)
- \(LR(1)\) items have lookahead strings of length 1
- several \(LR(1)\) items may have the same core
  \[[A ::= X \bullet YZ, a]
  \[A ::= X \bullet YZ, b]\]
  we represent this as
  \([A ::= X \bullet YZ, \{a, b\}]\)
**LR(1) lookahead**

What’s the point of all these lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping unless item has • at right end.
  - in \([A ::= X \bullet YZ, a]\), a has no direct use
  - in \([A ::= XYZ\bullet, a]\), a is useful

Recall, the SLR(1) construction uses LR(0) items!

---

**The point**

For \([A ::= a\bullet, a]\) and \([B ::= a\bullet, b]\), we can decide between reducing to \(A\) and to \(B\) by looking at limited right context!
Canonical \( LR(1) \) items

The canonical collection of sets of LR(1) items:

- sets of valid items for viable prefixes of the grammar
- sets of items derivable from \( [S' ::= \bullet S, \text{eof}] \) using \textit{goto} and \textit{closure} functions — both functions preserve validity.

A \( LR(1) \) item \( [A ::= \alpha \bullet \beta, a] \) is valid for a viable prefix \( \gamma \) if there is a derivation \( S \Rightarrow^*_rm \delta A w \Rightarrow_{rm} \delta \alpha \beta w \), where

- \( \gamma = \delta \alpha \), and
- either \( a \) is the first symbol of \( w \), or \( w \) is \( \epsilon \) and \( a \) is \textit{eof}.

Essentially,

- Each \( LR(1) \) item in a set in the canonical collection represents a state in an NFA that recognizes viable prefixes.
- Grouping these items together is really the DFA subset construction.
LR(1) closure

Given an item \([A ::= \alpha \bullet B\beta, a]\), its closure contains the item and any other items that can generate legal substrings to follow \(\alpha\).

Thus, if the parser has viable prefix \(\alpha\) on its stack, a substring of the input should reduce to \(B\beta\) (or \(\gamma\) for some other item \([B ::= \bullet\gamma, b]\) in the closure).

To compute closure\((I)\)

\[
\text{function closure}(I) \\
\text{repeat} \\
\quad \text{new\_item} \leftarrow \text{false} \\
\quad \text{for each item } [A ::= \alpha \bullet B\beta, a] \in I, \\
\quad \quad \text{each production } B ::= \gamma \in G', \\
\quad \quad \text{and each terminal } b \in \text{FIRST}(\beta a), \\
\quad \quad \text{if } [B ::= \bullet\gamma, b] \notin I \text{ then} \\
\quad \quad \quad \text{add } [B ::= \bullet\gamma, b] \text{ to } I \\
\quad \quad \quad \text{new\_item} \leftarrow \text{true} \\
\quad \text{endif} \\
\quad \text{until (new\_item = false)} \\
\text{return } I
\]

Aho, Sethi, and Ullman, Figure 4.38
Let $I$ be a set of $LR(1)$ items and $X$ be a grammar symbol.

Then, $\text{goto}(I, X)$ is the closure of the set of all items

$$[A ::= \alpha X \bullet \beta, a] \text{ such that } [A ::= \alpha \bullet X \beta, a] \in I$$

If $I$ is the set of valid items for some viable prefix $\gamma$, then $\text{goto}(I, X)$ is the set of valid items for the viable prefix $\gamma X$.

goto($I, X$) represents state after recognizing $X$ in state $I$.

To compute $\text{goto}(I, X)$

$$\text{function goto}(I, X)$$

$$J \leftarrow \text{set of items } [A ::= \alpha X \bullet \beta, a]$$

such that $[A ::= \alpha \bullet X \beta, a] \in I$

$$J' \leftarrow \text{closure}(J)$$

return $J'$

Aho, Sethi, and Ullman, Figure 4.38
Collection of sets of $LR(1)$ items

We start the construction of the canonical collection of $LR(1)$ items with the item $[S' ::= \bullet S, \text{eof}]$, where

- $S'$ is the start symbol of the augmented grammar $G'$
- $S$ is the start symbol of $G$, and
- $\text{eof}$ is the right end of string marker

To compute the collection of sets of $LR(1)$ items

```markdown
procedure items($G'$)
    $C \leftarrow \{\text{closure}([S' ::= \bullet S, \text{eof}])\}$
    repeat
        new_item $\leftarrow$ false
        for each set of items $I$ in $C$ and each grammar symbol $X$ such that $\text{goto}(I, X) \neq \emptyset$ and $\text{goto}(I, X) \notin C$
            add $\text{goto}(I, X)$ to $C$
        endfor
        new_item $\leftarrow$ true
    until (new_item = false)
```

Aho, Sethi, and Ullman, Figure 4.38
**LR(1) table construction**

The Algorithm

1. construct the collection of sets of LR(1) items for $G'$.

2. State $i$ of the parser is constructed from $I_i$.
   
   (a) if $[A ::= \alpha \cdot a\beta, b] \in I_i$ and $\text{goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift $j". (a must be a terminal)
   
   (b) if $[A ::= \alpha \cdot a] \in I_i$, then set $\text{ACTION}[i, a]$ to "reduce $A ::= \alpha$".
   
   (c) if $[S' ::= S\cdot, \text{eof}] \in I_i$, then set $\text{ACTION}[i, \text{eof}]$ to "accept".

3. If $\text{goto}(I_i, A) = I_j$, then set $\text{GOTO}[i, A]$ to $j$.

4. All other entries in $\text{ACTION}$ and $\text{GOTO}$ are set to "error"

5. The initial state of the parser is the state constructed from the set containing the item $[S' ::= \cdot S, \text{eof}]$.

Aho, Sethi, and Ullman, Algorithm 4.10
Example

Our simple grammar

1. \( S' ::= S \)
2. \( S ::= L = R \)
3. \( S ::= R \)
4. \( L ::= *R \)
5. \( L ::= \text{id} \)
6. \( R ::= L \)
Canonical LR(1) collection

$I_0 : \{ [S' := \bullet S, \text{eof}] , [S := \bullet L = R, \text{eof}] ,
\quad [S := \bullet R, \text{eof}] , [L := \bullet * R, \{=, \text{eof}\}] ,
\quad [L := \bullet \text{id}, \{=, \text{eof}\}] , [R := \bullet L, \text{eof}] \} $

$I_1 : \{ [S' := S \bullet, \text{eof}] \} $

$I_2 : \{ [S := L \bullet = R, \text{eof}] , [R := L \bullet, \text{eof}] \} $

$I_3 : \{ [S := R \bullet, \text{eof}] \} $

$I_4 : \{ [L := \ast \bullet R, \{=, \text{eof}\}] , [R := \bullet L, \{=, \text{eof}\}] ,
\quad [L := \bullet * R, \{=, \text{eof}\}] , [L := \bullet \text{id}, \{=, \text{eof}\}] \} $

$I_5 : \{ [L := \text{id} \bullet, \{=, \text{eof}\}] \} $

$I_6 : \{ [S := L = \bullet R, \text{eof}] , [R := \bullet L, \text{eof}] ,
\quad [L := \bullet * R, \text{eof}] , [L := \bullet \text{id}, \text{eof}] \} $

$I_7 : \{ [L := \ast R \bullet, \{=, \text{eof}\}] \} $

$I_8 : \{ [R := L \bullet, \{=, \text{eof}\}] \} $

$I_9 : \{ [S := L = R \bullet, \text{eof}] \} $

$I_{10} : \{ [R := L \bullet, \text{eof}] \} $

$I_{11} : \{ [L := \ast \bullet R, \text{eof}] , [R := \bullet L, \text{eof}] ,
\quad [L := \bullet * R, \text{eof}] , [L := \bullet \text{id}, \text{eof}] \} $

$I_{12} : \{ [L := \text{id} \bullet, \text{eof}] \} $

$I_{13} : \{ [L := \ast R \bullet, \text{eof}] \} $
**LR(1) table construction**

*So, does it work now?*

Example: Consider the set of items \( I_2 \). The action table is defined as follows:

\[
[S ::= L \cdot = R, \text{eof}] \Rightarrow \text{ACTION}[2, =] = "shift 6"
\]
\[
[R ::= L \cdot, \text{eof}] \Rightarrow \text{ACTION}[2, \text{eof}] = "reduce 6"
\]

**Yes**, the table construction defines only single entries for each position in the \textbf{ACTION} table.

The price we pay: more states

- construction of table takes longer
- table is larger

Here is an idea: \textit{LALR}(1) parsing: Union states that are sets of \textit{LR}(1) items with the same \textit{core}. 

---

198:515, Spring 2016  Lecture 3, Page 47
**LALR(1) parsing**

LR(1) parsers have many more states than SLR(1) parsers (approximately factor of ten for Pascal).

LALR(1) parsers have same number of states as SLR(1) parsers, but with more power due to lookahead in states.

Define the core of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.

Thus, the two sets

- $\{[A ::= \alpha \bullet \beta, a], [A ::= \alpha \bullet \beta, b]\}$, and
- $\{[A ::= \alpha \bullet \beta, c], [A ::= \alpha \bullet \beta, d]\}$

have the same core.

**Key Idea:**

If two sets of LR(1) items, $I_i$ and $I_j$, have the same core, we can merge the states that represent them in the \texttt{ACTION} and \texttt{GOTO} tables.
Back to our example grammar

1. $S' ::= S$
2. $S ::= L = R$
3. $S ::= R$
4. $L ::= *R$
5. $L ::= id$
6. $R ::= L$

$I_0 : \{ [S' ::= \bullet S, \text{eof}], [S ::= \bullet L = R, \text{eof}],$
$[S ::= \bullet R, \text{eof}], [L ::= \bullet R, \{=, \text{eof}\}],$
$[L ::= \bullet id, \{=, \text{eof}\}], [R ::= \bullet L, \text{eof}] \}$

$I_1 : \{ [S' ::= S\bullet, \text{eof}] \}$

$I_2 : \{ [S ::= L\bullet = R, \text{eof}], [R ::= L\bullet, \text{eof}] \}$

$I_3 : \{ [S ::= R\bullet, \text{eof}] \}$

$I_{4/11} : \{ [L ::= \bullet R, \{=, \text{eof}\}], [R ::= \bullet L, \{=, \text{eof}\}],$
$[L ::= \bullet R, \{=, \text{eof}\}], [L ::= \bullet id, \{=, \text{eof}\}] \}$

$I_{5/12} : \{ [L ::= id\bullet, \{=, \text{eof}\}] \}$

$I_6 : \{ [S ::= L = \bullet R, \text{eof}], [R ::= \bullet L, \text{eof}],$
$[L ::= \bullet R, \text{eof}], [L ::= \bullet id, \text{eof}] \}$

$I_{7/13} : \{ [L ::= \bullet R, \{=, \text{eof}\}] \}$

$I_{8/10} : \{ [R ::= L\bullet, \{=, \text{eof}\}] \}$

$I_9 : \{ [S ::= L = R\bullet, \text{eof}] \}$
Example

1. $S' ::= S$
2. $S ::= L = R$
3. $S ::= R$
4. $L ::= *R$
5. $L ::= \text{id}$
6. $R ::= L$

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id * = eof</td>
<td>S L R</td>
</tr>
<tr>
<td>$S_0$</td>
<td>s5/12 s4/11 — —</td>
</tr>
<tr>
<td>$S_1$</td>
<td>— — — acc</td>
</tr>
<tr>
<td>$S_2$</td>
<td>— — s6 r6</td>
</tr>
<tr>
<td>$S_3$</td>
<td>— — — r3</td>
</tr>
<tr>
<td>$S_{4/11}$</td>
<td>s5/12 s4/11 — —</td>
</tr>
<tr>
<td>$S_{5/12}$</td>
<td>— — r5/12 r5/12</td>
</tr>
<tr>
<td>$S_6$</td>
<td>s5/12 s4/11 — —</td>
</tr>
<tr>
<td>$S_{7/13}$</td>
<td>— — r4/11 r4/11</td>
</tr>
<tr>
<td>$S_{8/10}$</td>
<td>— — r6 r6</td>
</tr>
<tr>
<td>$S_9$</td>
<td>— — — r2</td>
</tr>
</tbody>
</table>
LALR(1) properties

LALR(1) parsers have same number of states as SLR(1) parsers (core LR(0) items are the same)

In case of error, LALR(1) parser may perform more reductions than corresponding LR(1) parser, but will catch error before more input is processed.

Example grammar with input "id = id =":

\[
\begin{align*}
\text{LR}(1): & \quad S_0 \rightarrow^s S_5 \rightarrow^r S_2 \rightarrow^s S_6 \rightarrow^s S_{12} \Rightarrow \text{error} \\
\text{LALR}(1): & \quad S_0 \rightarrow^s S_{5/12} \rightarrow^r S_2 \rightarrow^s S_6 \rightarrow^s S_{5/12} \rightarrow^r S_{8/10} \rightarrow^r S_9 \Rightarrow \text{error}
\end{align*}
\]
Summary: Resolving parse conflicts

Parse conflicts possible when certain \( LR \) items are found in the same state.

Depending on parser, may choose between \( LR \) items using lookahead.

Legal lookahead for \( LR \) items must be disjoint, else conflict exists.

<table>
<thead>
<tr>
<th></th>
<th>Shift-Reduce</th>
<th>Reduce-Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ A ::= \alpha \bullet, \Delta ]</td>
<td>[ A ::= \alpha \bullet, \Delta ]</td>
</tr>
<tr>
<td></td>
<td>[ B ::= \beta \bullet \gamma, \Omega ]</td>
<td>[ B ::= \beta \bullet, \Omega ]</td>
</tr>
<tr>
<td>( LR(0) )</td>
<td>conflict</td>
<td>conflict</td>
</tr>
<tr>
<td>( SLR(1) )</td>
<td>( FOLLOW(A) ) ( \cap ) ( FIRST(\gamma) )</td>
<td>( FOLLOW(A) ) ( \cap ) ( FOLLOW(B) )</td>
</tr>
<tr>
<td>( LR(1) )</td>
<td>( \Delta \cap FIRST(\gamma) )</td>
<td>( \Delta \cap \Omega )</td>
</tr>
</tbody>
</table>
Error Recovery, Common Subexpression Elimination, Attribute Grammars, Code Generation, Functional Languages

*For next time, please read:* Scott: please see web site
ALSU: please see web site

First project and third homework will be out by Wednesday. Please see our course website.