CS 415: Lecture 11

- LR(1) and LALR(1) parsing

SLR(1) (Review)

- Uses DFA to recognize viable prefixes of grammar G
- Each state in the DFA:
  - is the set of LR(0) items valid for a viable prefix
  - "encodes" information about the symbols that have been shifted onto the stack
- Valid LR(0) items are computed by applying the closure and goto functions to the initial, valid item [S’ ::= .S] (this is called the canonical collection of LR(0) items)
- Uses FOLLOW to disambiguate actions
SLR(1) Review

1. If \( A ::= \alpha \cdot a \beta \) is in \( I_k \) and goto\( (I_k, a) = I_j \), set \( \text{actions}[k,a] \) to \( s_j \).
2. If \( A ::= \alpha \cdot \) in \( I_k \) then set \( \text{actions}[k,b] \) to \( r\text{-rule#} \), for all \( b \in \text{FOLLOW}(A) \).
3. If \( S' ::= S \cdot \) is in \( I_k \) then set \( \text{actions}[k,\$] \) to accept.

Rules 1-3 may define conflicting actions for an entry in the actions table. In this case, the grammar is not SLR(1).

Shift/Reduce Conflict

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow A \cdot b \mid d \cdot c \mid b \cdot A \cdot c \\
A & \rightarrow d
\end{align*}
\]

A very simple language = \{db, dc, bdc\}
Follow(\( S \)) = \{\$\}, Follow(\( A \)) = \{b, c\}

Form part of the SLR(1) parser:

\[
\begin{array}{c}
I_0: S' \rightarrow S \\
S & \rightarrow A \cdot b \\
S & \rightarrow d \cdot c \\
S & \rightarrow b \cdot A \cdot c \\
A & \rightarrow d
\end{array}
\]

\[
\begin{array}{c}
I_1: S \rightarrow d \cdot c \\
A & \rightarrow d.
\end{array}
\]

But since \( c \) is in \text{FOLLOW}(A), we don’t know whether to reduce or shift in state.
\( I_1 \) if \( c \) is next input symbol!

Deriv1: \( S' \rightarrow S \rightarrow dc \)
Deriv2: \( S' \rightarrow S \rightarrow b \cdot A \cdot c \rightarrow bdc \)
Reduce/Reduce Conflict

\[ S' \rightarrow S \]
\[ S \rightarrow b\ A\ e \mid b\ B\ d \mid A\ c \]
\[ A \rightarrow d \]
\[ B \rightarrow E\ c \]
\[ E \rightarrow d \]

Deriv1: \[ S' \rightarrow S \rightarrow Ac \rightarrow dc \]
Deriv2: \[ S' \rightarrow S \rightarrow bBd \rightarrow bEc \rightarrow bcd \]
Deriv3: \[ S' \rightarrow S \rightarrow bAe \rightarrow bde \]

\[ I_0: S' \rightarrow S \]
\[ S \rightarrow .\ A\ e \]
\[ S \rightarrow .\ B\ d \]
\[ S \rightarrow .\ c \]
\[ A \rightarrow .\ d \]

\[ I_1: S \rightarrow b\ A\ e \]
\[ S \rightarrow b\ B\ d \]
\[ A \rightarrow .\ d \]
\[ B \rightarrow .\ E\ c \]
\[ E \rightarrow .\ d \]

\[ I_2: A \rightarrow .\ d \]
\[ E \rightarrow .\ d \]

Which reduction to take? Follow set too imprecise here to decide.

LR(1)

- Solution: keep more information about what next input symbol can be as part of DFA state
  - Idea: keep an input look-ahead as part of each item (these are called LR(1) items)
  - More precise than Follow sets which essentially unions these look-ahead symbols for non-terminal A over all sentential forms in which A appears
  - Potentially gives rise to much bigger parsers than SLR(1) (more states)
LR(k) Items

- The table construction algorithm for an LR(k) parser uses 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 symbol at some position in the rhs.
  - \(\beta\) is a look-ahead string containing \(k\) symbols (terminals or $).
- Example LR(1) item:
  \([A ::= X \cdot Y Z, a]\)
  - where \(a\) is the symbol that we would expect next.

---

LR(k) Items

- What’s the point of the look-ahead symbols?
- Carry them along to allow us to choose correct reduction when there is any choice.
- Look-ahead symbols are bookkeeping unless item has a at the right end:
  - In \([A ::= X \cdot Y Z, a]\), \(a\) has no direct use.
  - In \([A ::= X Y Z, a]\), \(a\) is useful.
- The point: for \([A ::= \alpha, a]\) and \([B ::= \alpha, b]\), we can decide between reducing to \(A\) or 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 \( S' ::= S, \) using \( \text{goto} \) and \( \text{closure} \)
- We say that an LR(1) item \([A ::= \alpha, \beta, a]\) is valid for a viable prefix \( \gamma \) if there is a derivation
  \[
  S \Rightarrow^*_{r} \delta Aw \Rightarrow_{r} \delta \alpha \beta w
  \]
  where
  - \( \gamma = \delta \alpha, \) and
  - either \( a \) is the first symbol of \( w, \) or \( w \) is \( \varepsilon \) and \( a \) is $.

 LR(1) closure

- Given an LR(1) item \([A ::= \alpha, B\beta, a]\), its closure contains the item and any other LR(1) items that can generate legal sub-strings to follow \( \alpha. \)
- Thus, if the parser has the suffix \( \alpha \) of a viable prefix on its stack, a sub-string of the input should reduce to \( B\beta \) (or some \( \gamma \) for some other item \([B ::= \gamma, b]\) in the closure).
LR(1) Closure

function closure(I)
    repeat
        new_item <- false
        for each item \([A ::= \alpha.B\beta,a] \in I\),
            each production \(B ::= \gamma \in G\),
            and each terminal \(b \in \text{FIRST}(\beta,a)\),
            if \([B ::= \gamma,b] \notin I\) then
                add \([B ::= \gamma,b]\) to I
                new_item <- true
            endif
    endfor
    until (new_item = false)
    return I

LR(1) goto

Let I be a set of LR(1) items and X be a grammar symbol.
Then, goto(I,X) is the closure of the set of all items
\([A ::= \alpha.X.\beta,a] such that [A ::= \alpha.X\beta,a] \in I\]

That is, if I is the set of valid items for some viable prefix \(\gamma\), then goto(I,X) is the set of valid items for the viable prefix \(\gamma X\)
LR(1) goto

function goto(I, X)
    J <- set of items \([A ::= \alpha X \beta, a]\)
        such that \([A ::= \alpha X \beta, a] \in I\)
    J' <- closure(J)
    return J'

Canonical Collection of Sets of LR(1) Items

- We start the construction of the canonical collection of sets of LR(1) items with the item \([S' ::= .S, \$]\), where
  - \(S'\) is the start symbol of the augmented grammar \(G\)
  - \(S\) is the start symbol of \(G\)
  - \(\$\) is the right end of sentence marker

Rutgers University, DCS
CS 415: Compilers
Canonical Collection of Sets of LR(1) Items

Procedure items(G')
C <- closure({[S' ::= .S,$]})
repeat
  new_item <- false
  for each set of items I in C and each grammar symbol X such that
  goto(I, X) ≠ ∅ and
  goto(I, X) ∉ C
    add goto(I, X) to C
    new_item <- true
  endfor
until (new_item = false)

LR(1) Table Construction

1. Construct the canonical collection of sets of LR(1) items for G.
2. State i of the parser is constructed from I_i
   a. If [A ::= α.α], b) ∈ I_i and goto(I_i, a) = I_j, then set ACTION[i, a] to "shift j." (a must be a terminal)
   b. If [A ::= α, a] ∈ I_i, then set ACTION[i, a] to "reduce using A ::= α"
   c. If [S' ::= S, $] ∈ I_i, then set ACTION[i, $] to "accept"
3. If goto(I_i, A) = I_j, then set GOTO[i, A] to j
4. All other entries in ACTION and GOTO are set to "error"
5. The initial state of the parser is the state constructed from the set containing the item [S' ::= .S, $]
Example LR(1) Table Construction

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

LALR(1)

- LR(1) more powerful than SLR(1) but many more states (e.g., approximately a factor of 10 for Pascal)
- LALR(1)
  - Define the core of a set of LR(1) items to be the set of LR(0) items derived by ignoring the look-ahead symbols
  - For example, the two sets
    \[
    \{ [A ::= \alpha, \beta, a], [A ::= \alpha, \beta, b] \}
    \]
    \[
    \{ [A ::= \alpha, \beta, c], [A ::= \alpha, \beta, d] \}
    \]
    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 ACTION and GOTO tables
What Can Go Wrong?

- LALR derived from LR with no shift-reduce conflict will also have no shift-reduce conflict
- LALR may create reduce-reduce conflict not in LR from which LALR is derived
- Example

```
S' ::= S
S ::= aAd | bBd | aBe | bAe
A ::= c
c
B ::= e
c

I_1 = \{A ::= c..d, B ::= c..e\} is valid for prefix ac
I_2 = \{A ::= c..e, B ::= c..d\} is valid for prefix bc
Merging these two states gives
I_{1,2} = \{[A ::= c..d, e],[B ::= c..d]\}
conflict!
```

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
LALR(1) Table Construction

Two approaches to constructing LALR(1) parsing tables
1. Build LR(1) sets of items, then merge
   a. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union
   b. Update the goto function to reflect the replacement sets
   This algorithm has large space requirements
2. Build LR(0) sets of items, then generate look-ahead information, Avoids the construction of full LR(1) collection. See ASU, pages 240-244 for details.

Ambiguous Grammars

Precedence and associativity can be used to resolve shift-reduce conflicts in some ambiguous grammars
- Look-ahead with higher precedence => shift
- Same precedence, left associative => reduce

Advantages
- Allow use of more concise but ambiguous grammars
- Shallower parse trees => fewer reductions

Example
- expr ::= expr * expr | expr / expr | expr + expr | expr - expr | (expr) | -expr | id | num
Error Recovery

- **Basic idea (panic mode):**
  - Predetermine a set of non-terminals representing major pieces of a program MS
  - Scan down the stack until a state $s$ is found with a goto on a non-terminal $A$ in $MS$
  - Discard 0 or more input tokens until a symbol $b$ is found, where $b$ is in FOLLOW($A$)
  - Push $(A, \text{goto}[s,A])$ onto stack and resume parsing

Error Recovery in yacc

- **Basic mechanism (version dependent)**
  - Designate error token
  - Used in error productions of the form $A ::= \beta \text{ error } \alpha$
  - $\alpha$ specifies synchronization points
- **When error is discovered**
  - Pops stack until it finds state where it can shift the error token
  - Resumes parsing to match $\alpha$
    - If $\alpha = w$: skip input until $w$ has been read
    - If $\alpha = \epsilon$: skip input until state transition on input token is defined
Example of Error Recovery in yacc

\[
\text{stmt_list ::= stmt | stmt_list ; stmt}
\]

can be augmented with \text{error}

\[
\text{stmt_list ::= stmt | error | stmt_list ; stmt}
\]

This should

- Throw out the erroneous statement
- Synchronize at ";" or "end" (implicit)
-Invoke \text{yyerror("syntax error")} (implicit); user may want to insert explicit call (e.g., error {\text{yyerror("illegal statement")}})

Other "natural" places for error productions

- All the "lists", missing parentheses or brackets, extra operator or missing operator