LR Parsing

Lecture Notes by Profs Aiken and Necula (UCB)

Outline

- Review of SLR parsing
- Limits of SLR parsing
- LR parsing
- LALR parsing
- Implementation of semantic actions
- Using parser generators

Review of SLR(1) Parsing

- LR parser maintains a stack
  \((\text{sym}_1, \text{state}_1), \ldots, (\text{sym}_n, \text{state}_n)\)
  \(\text{state}_n\) is the final state of the DFA on \(\text{sym}_1 \ldots \text{sym}_n\)
- **Goto table**: the transition function of the DFA
  - \(\text{Goto}[i, A] = j\) if \(\text{state}_i \rightarrow^A \text{state}_j\)
- **Action table**: for each state and terminal:
  - \(\text{Shift } j\)
  - \(\text{Reduce } X \rightarrow \alpha\)
  - \(\text{Accept}\)
  - \(\text{Error}\)

LR Parsing Algorithm

Let \(I = w$$\) be initial input
Let \(j = 0\)
Let DFA state 1 have item \(S' \rightarrow S\)
Let stack = \(<\text{dummy}, 1>\)

repeat
  case \(\text{Action}[\text{top}\_state(\text{stack}), I[j]]\) of
    - \(\text{Shift } k\): push \((I[j+1], k)\)
    - \(\text{Reduce } X \rightarrow \alpha\):
      pop \(|\alpha|\) pairs,
      push \((X, \text{Goto}(X, \text{top}\_state(\text{stack})))\)
    - \(\text{Accept}\): halt normally
    - \(\text{Error}\): halt and report error

Review Items

- An item \([X \rightarrow \alpha, \beta]\) says that
  - the parser is looking for an \(X\)
  - it has an \(\alpha\) on top of the stack
  - Expects to find a string derived from \(\beta\) next in the input

- Notes:
  - \([X \rightarrow \alpha, \beta]\) means that \(\alpha\) should follow. Then we can shift it and still have a viable prefix.
  - \([X \rightarrow \alpha, \beta]\) means that we could reduce \(X\)
    - But this is not always a good idea!

SLR(1) Action Table

For each state \(s_i\) and terminal \(a\)
- If \(s_i\) has item \(X \rightarrow \alpha, \beta\) and \(\text{Goto}(i, a) = j\) then
  - \(\text{Action}[i, a] = \text{Shift } j\)
- If \(s_i\) has item \(S' \rightarrow S\), then \(\text{Action}[i, $$] = \text{Accept}\)
- If \(s_i\) has item \(X \rightarrow \alpha\) and \(a \in \text{Follow}(X)\) and \(X = S\)
  then \(\text{Action}[i, a] = \text{Reduce } X \rightarrow \alpha\)
- Otherwise, \(\text{Action}[i, a] = \text{Error}\)
Limits of SLR Parsing

• SLR(1) is the simplest LR parsing method
• SLR(1) is almost powerful enough, but ...
• ... some common programming language constructs are not SLR(1).

Consider the grammar

\[ S \rightarrow L = E \mid E \]
\[ L \rightarrow \ast E \mid \text{id} \]
\[ E \rightarrow L \]

Limits of SLR Parsing (cont.)

• Consider two states of the DFA for recognizing viable prefixes

\[ S' \rightarrow S \]
\[ S \rightarrow L \ast E \rightarrow \ast E \rightarrow E \]
\[ S \rightarrow E \]
\[ L \rightarrow \ast E \rightarrow \ast E \rightarrow E \]
\[ E \rightarrow \text{id} \rightarrow E \rightarrow \ast E \rightarrow E \]

SLR(1) parser on input “=”
• shift (item \[L \ast E\])
• reduce by \[E \rightarrow L\] (since “=” \(\in\) Follow(E))

What’s The Problem?

• The grammar is not SLR(1), but why?
• Focus on the reduce move in the second state
  - We are in the context of \[S \rightarrow E \rightarrow L\]
  - No “can follow E in this context
  - Even though “\(\in\) Follow(E) (in \[S \rightarrow L = E \rightarrow \ast E \rightarrow E\])
  - The reduce move should not happen if an “follows in this context.

What’s The Problem? (Cont.)

• Problem: the SLR table has too many reduce actions.
  - Using Follow is too coarse.
• In any given context, only some elements of Follow can actually follow a non-terminal.
  - For example:
    - Follow(E) = \{=, $\}, but
    - In context \[S \rightarrow E\] \(\rightarrow\) only $ can follow E
    - In context \[S \rightarrow L \ast E \rightarrow \ast E \rightarrow E\] \(\rightarrow\) only = can follow \[E \rightarrow L\]

One Way to Fix The Problem: LR(1) Items

• Idea:
  - refine Follow based on context.
  - The context is described through items.
• An LR(1) item is a pair
  \[ [X \rightarrow \alpha \beta, a] \]
  where \(X \rightarrow \alpha \beta\) is a production and \(a\) is the lookahead token or $.
• LR(k) is similar but with \(k\) tokens of lookahead
  - In practice, \(k = 1\)

LR(1) Items, Intuition

• \([X \rightarrow \alpha \beta, a]\) describes a state of the parser:
  - We are trying to find an \(X\), and
  - We have \(\alpha\) already on top of the stack, and
  - We expect to see a prefix derived from \(\beta\)
• Back to reduce actions: have an \([X \rightarrow \alpha \gamma, a]\)
  - Perform the reduce only if next token is \(a\!\)
  - Will have fewer reduce actions
  - Not for all \(b \in\) Follow(X)
Constructing Sets of LR(1) Items (1)

- Similar to construction for LR(0).
- The states of the NFA are the LR(1) items of \( G \).
- The start state is \([S' \rightarrow . S, $]\)

Constructing Sets of LR(1) Items (2)

1. For each LR(1) item \([Y \rightarrow \alpha X \beta, a]\)
   - Add an \( X \)-transition \([Y \rightarrow \alpha X \beta, a] \rightarrow X [Y \rightarrow \alpha X \beta, a]\)
2. For each LR(1) item \([Y \rightarrow \alpha X \beta, a]\)
   - For each production \( X \rightarrow \gamma \)
   - For each terminal \( b \in \text{First}(\beta a) \)
   - Add an \( \varepsilon \)-transition \([Y \rightarrow \alpha X \beta, a] \rightarrow \varepsilon [X \rightarrow \gamma, b]\)

NFA for Viable Prefixes in Detail (1)

\[
\begin{align*}
S' &\rightarrow . S, $ \\
S &\rightarrow . E, \$ \\
S' &\rightarrow . S \$ \\
S &\rightarrow . L = E, \$ \\
S &\rightarrow . \varepsilon, \$ \\
S &\rightarrow . L . = E \\
S &\rightarrow . \varepsilon \\
S &\rightarrow . E \\
E &\rightarrow . L \\

NFA for Viable Prefixes in Detail (2)

\[
\begin{align*}
S' &\rightarrow . S, $ \\
S &\rightarrow . E, \$ \\
S' &\rightarrow . S \$ \\
S &\rightarrow . L = E, \$ \\
S &\rightarrow . \varepsilon, \$ \\
S &\rightarrow . L . = E \\
S &\rightarrow . \varepsilon \\
S &\rightarrow . E \\
E &\rightarrow . L \\
L &\rightarrow . \text{id} = \\
L &\rightarrow . * E = \\
S &\rightarrow . L . \\
E &\rightarrow . L . \\

NFA for Viable Prefixes in Detail (3)

\[
\begin{align*}
S' &\rightarrow . S, $ \\
S &\rightarrow . E, \$ \\
S' &\rightarrow . S \$ \\
S &\rightarrow . L = E, \$ \\
S &\rightarrow . \varepsilon, \$ \\
S &\rightarrow . L . = E \\
S &\rightarrow . \varepsilon \\
S &\rightarrow . E \\
E &\rightarrow . L \\
L &\rightarrow . \text{id} = \\
L &\rightarrow . * E = \\
S &\rightarrow . L . \\
E &\rightarrow . L . \\
L &\rightarrow . * E = \\
S &\rightarrow . L . \\

NFA for Viable Prefixes in Detail (4)

\[
\begin{align*}
S' &\rightarrow . S, $ \\
S &\rightarrow . E, \$ \\
S' &\rightarrow . S \$ \\
S &\rightarrow . L = E, \$ \\
S &\rightarrow . \varepsilon, \$ \\
S &\rightarrow . L . = E \\
S &\rightarrow . \varepsilon \\
S &\rightarrow . E \\
E &\rightarrow . L \\
L &\rightarrow . \text{id} = \\
L &\rightarrow . * E = \\
S &\rightarrow . L . \\
E &\rightarrow . L . \\
L &\rightarrow . * E = \\
S &\rightarrow . L . \\

\]
An Example Revisited

• Consider the state from last slide

\[ S \rightarrow L \cdot = E \cdot $ \]

\[ E \rightarrow L \cdot $ \]

• LR(1) parser on input "="
  • only shift (item \( L \cdot = E \cdot \))

Constructing LR(1) Parsing Tables

1. Add a dummy \( S' \rightarrow S \) production
2. Construct the NFA of LR(1) items as before
3. Convert the NFA into a DFA
4. Goto is defined exactly as before:
   \[ \text{Goto}[i, A] = j \text{ if state}_i \rightarrow^* \text{state}_j \]
   (the transition function of the DFA)

Constructing LR(1) Parsing Tables (Cont.)

5. For each state \( s \) of the DFA and terminal \( a \)
   - If \( s \) has item \( [X \rightarrow \alpha, a, \beta] \) and \( \text{Goto}[s, a] = j \) then
     \( \text{action}[j, a] = \text{shift} j \)
   - If \( s \) has item \( [X \rightarrow \alpha, a] \) and \( X \neq S \) then
     \( \text{action}[j, a] = \text{reduce} X \rightarrow \alpha \)
   - If \( s \) has item \( [S' \rightarrow S, \$] \) then
     \( \text{action}[j, \$] = \text{accept} \)
   - Otherwise,
     \( \text{action}[j, a] = \text{error} \)

• LR(1) grammar \( \Leftrightarrow \text{action}[j, a] \) uniquely defined

LALR Parsing

• Two bottom-up parsing methods: SLR and LR
  - Which one we use? Neither
    - SLR is not powerful enough.
    - LR parsing tables are too big (1000's of states vs. 100's of states for SLR).
  - In practice, use LALR(1)
    - Stands for Look-Ahead LR
    - A compromise between SLR(1) and LR(1)

LALR Parsing (Cont.)

• Rough intuition: A LALR(1) parser for \( G \) has
  - The number of states of an SLR parser.
  - Some of the lookahead discrimination of LR(1).

• Idea: construct the DFA for the LR(1).
  - Then merge the DFA states whose items differ only in the lookahead tokens
    - We say that such states have the same core.

The Core of a Set of LR Item

• Definition: The core of a set of LR items is the set of first components.

• Example: the core of
  \[ \{ [X \rightarrow \alpha, \beta, b], [Y \rightarrow \gamma, \delta, d] \} \]
  is
  \[ \{ X \rightarrow \alpha, \beta, Y \rightarrow \gamma, \delta \} \]

• The core of an LR item is an LR(0) item.
A LALR(1) DFA

• Repeat until all states have distinct core.
  - Choose two distinct states with same core.
  - Merge the states by creating a new one with the union of all the items.
  - Point edges from predecessors to new state.
  - New state points to all the previous successors.

LALR vs. LR Parsing

• LALR languages are not natural.
  - They are an efficiency hack on LR languages

• Any reasonable programming language has an LALR(1) grammar.

• LALR(1) has become a standard for programming languages and for parser generators.

The LALR Parser Can Have Conflicts

• Consider for example the LR(1) states
  \([X \rightarrow \alpha., a], [Y \rightarrow \beta., b])\]
  \([X \rightarrow \alpha., b], [Y \rightarrow \beta., a])\]

• And the merged LALR(1) state
  \([X \rightarrow \alpha/b], [Y \rightarrow \beta/a/b])\]

• Has a new reduce-reduce conflict.

• In practice such cases are rare.

Semantic Actions

• We can now illustrate how semantic actions are implemented for LR parsing.

• Keep attributes on the stack.

• On shift \(a\), push attribute for \(a\) on stack.

• On reduce \(X \rightarrow \alpha\)
  - pop attributes for \(\alpha\)
  - compute attribute for \(X\)
  - and push it on the stack

Performing Semantic Actions, Example

• Recall the example from earlier lecture

\[
\begin{align*}
E & \rightarrow T + E_1 \\
& \quad (E.val = T.val + E_1.val) \\
T & \rightarrow int * T_1 \\
& \quad (T.val = int.val * T_1.val) \\
T & \rightarrow int \\
& \quad (T.val = int.val)
\end{align*}
\]

• Consider the parsing of the string \(3 * 5 + 8\)
Performing Semantic Actions, Example

| int * int + int | shift |
| int * int + int | shift |
| int * int + int | shift |
| int * int + int | shift |
| int * int + int | reduce T -> int |
| int * int + int | reduce T -> int * T |
| T15 * int | shift |
| T15 * int | shift |
| T15 * int | reduce T -> int |
| T15 * int | reduce E -> T |
| T15 + E8 | reduce E -> T + E |
| T15 | accept |

Notes

- The previous discussion shows how synthesized attributes are computed by LR parsers.
- It is also possible to compute inherited attributes in an LR parser.

Using Parser Generators

- Most common parser generators are LALR(1).
- A parser generator constructs a LALR(1) table.
- And reports an error when a table entry is multiply defined:
  - A shift and a reduce. Called shift/reduce conflict
  - Multiple reduces. Called reduce/reduce conflict
- An ambiguous grammar will generate conflicts.
- What do we do in that case?

Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar.
- Classic example: the dangling else

```
S -> if E then S | if E then S else S | OTHER
```

- Will have DFA state containing
  - [S -> if E then S., else]
  - [S -> if E then S. else S.
  - if else follows, then we can shift or reduce
- Default (bison, CUP, etc.) is to shift
- Default behavior is as needed in this case.

More Shift/Reduce Conflicts

- Consider the ambiguous grammar
  - E -> E * E | E * E | int
- We will have the states containing
  - [E -> E * E, +] [E -> E * E, +]
  - [E -> E * E, +] => [E -> E * E, +]
  - Again we have a shift/reduce on input +
    - We need to reduce ("" binds more tightly than ")
    - Recall solution: declare the precedence of "" and ""

More Shift/Reduce Conflicts

- In bison, declare precedence and associativity:
  - %left +
  - %left *
- Precedence of a rule = that of its last terminal
  - See bison manual for ways to override this default.
- Resolve shift/reduce conflict with a shift if:
  - no precedence declared for either rule or terminal
  - input terminal has higher precedence than the rule
  - the precedences are the same and right associative
Using Precedence to Solve S/R Conflicts

- Back to our example:
  \[ E \rightarrow E \cdot E, + \] \[ E \rightarrow E + E, + \] \[ E \rightarrow \cdot E + E, + \] \[ E \rightarrow E \cdot + E, + \] \[ E \rightarrow E + + E \] \[ E \rightarrow \cdot E + + E \] \[ E \rightarrow E \cdot + + E \] \[ E \rightarrow E + + + E \] 
- Will choose reduce because precedence of rule \( E \rightarrow E \cdot E \) is higher than of terminal +.

Using Precedence to Solve S/R Conflicts

- Same grammar as before
  \[ E \rightarrow E \cdot E \mid E + E \mid \text{int} \]
- We will also have the states
  \[ E \rightarrow E \cdot E, + \] \[ E \rightarrow E + E, + \] \[ E \rightarrow \cdot E + E, + \] \[ E \rightarrow E \cdot + E, + \] \[ E \rightarrow E + + E \]
- Now we also have an S/R conflict on input +.
  - We choose reduce because \( E \rightarrow E \cdot E \) and + have the same precedence and + is left-associative.

Reduce/Reduce Conflicts

- Usually due to gross ambiguity in the grammar
- Example: a sequence of identifiers
  \[ S \rightarrow \varepsilon \mid id \mid id \cdot S \]
- There are two parse trees for the string \( id \cdot id \cdot S \cdot id \)
  - S → id
  - S → id S → id
- How does this confuse the parser?

More on Reduce/Reduce Conflicts

- Consider the states
  \[ S \rightarrow \cdot id \cdot, \$ \]
  \[ S \rightarrow S, \$ \]
  \[ S \rightarrow id, \$ \]
  \[ S \rightarrow id S, \$ \]
- Reduce/reduce conflict on input $ S \rightarrow S \rightarrow id S \rightarrow id$
- Better rewrite the grammar: \( S \rightarrow \varepsilon \mid id \cdot S \)

Strange Reduce/Reduce Conflicts

- Consider the grammar
  \[ S \rightarrow P \cdot R, \]
  \[ N \rightarrow N \mid N, NL \]
  \[ P \rightarrow T \mid NL : T \]
  \[ R \rightarrow T \mid N : T \]
  \[ N \rightarrow id \]
  \[ T \rightarrow id \]
- P - parameters specification
- R - result specification
- N - a parameter or result name
- T - a type name
- NL - a list of names
Strange Reduce/Reduce Conflicts

- In P an id is a
  - N when followed by , or :
  - T when followed by id
- In R an id is a
  - N when followed by :
  - T when followed by ,
- This is an LR(1) grammar.
- But it is not LALR(1). Why?
  - For obscure reasons

A Few LR(1) States

```
P → T . id
P → NL : T . id
NL → N
NL → N , NL
N → id :
N → id ,
T → id id
R → T ,
R → N : T ,
R → . id bogus,
```

LALR reduce/reduce conflict on ","

Different cores ⇒ no LALR merging

What Happened?

- Two distinct states were confused because they have the same core.
- Fix: add dummy productions to distinguish the two confused states.
- E.g., add
  
  \[ R → \text{id bogus} \]
  
  - bogus is a terminal not used by the lexer.
  - This production will never be used during parsing.
  - But it distinguishes R from P.

A Few LR(1) States After Fix

```
P → T . id
P → NL : T . id
NL → N
NL → N , NL
N → id :
N → id ,
T → id id
R → T ,
R → N : T ,
R → . id bogus,
```

Different cores ⇒ no LALR merging

Notes on Parsing

- Parsing
  - A solid foundation: context-free grammars
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - LALR(1) parser generators
- Next time we move on to semantic analysis.