Definitions

Shift-reduce parsing

Shift-reduce parsers

LR(1) items

An LR(0) item is a string \( a \), where \( \text{LR}(0) \) items

\[ ZAX =:: \cdot A \]
\[ Z \cdot AX =:: \cdot A \]
\[ AX \cdot =:: \cdot A \]
\[ ZAX\cdot =:: \cdot A \]

Generate a LR(0) item, no lookahead.

Z derivable from \( AX \) and is looking for one string derivable from \( AX \). The \( Z \) indicates that the parser has seen a string derivable from \( AX \) and is looking for one string derivable from \( ZAX\cdot \). The \( Z \) indicates that the parser is looking for a string derivable from \( ZAX\cdot \).

at a given state in the parse.

The \( \cdot \) indicates how much of an item we have seen.

A production from \( G \) with a \( \cdot \) at some position in the rhs is a production from \( G \) with a \( \cdot \) at some position in the rhs.

Shift-reduce parsing information in DFA.

Construct DFA for recognizing viable prefix.

TAR(1)

Storing lookahead information in DFA.

Construct DFA for recognizing viable prefix.

SLR(1) = LR(0) + FOLLOW

Reductions

Define precedence between operators to guide parses.

Shift-reduce parsers

LR(1) items

An LR(0) item is a string \( a \), where \( \text{LR}(0) \) items

\[ ZAX =:: \cdot A \]
\[ Z \cdot AX =:: \cdot A \]
\[ AX \cdot =:: \cdot A \]
\[ ZAX\cdot =:: \cdot A \]

Generate a LR(0) item, no lookahead.

Z derivable from \( AX \) and is looking for one string derivable from \( AX \). The \( Z \) indicates that the parser has seen a string derivable from \( AX \) and is looking for one string derivable from \( ZAX\cdot \). The \( Z \) indicates that the parser is looking for a string derivable from \( ZAX\cdot \).

at a given state in the parse.

The \( \cdot \) indicates how much of an item we have seen.

A production from \( G \) with a \( \cdot \) at some position in the rhs is a production from \( G \) with a \( \cdot \) at some position in the rhs.
The Grammar

1d = ::= P3
1l = ::= P2
E + E = ::= P1
E = ::= P0

The Augmented Grammar

S = ::= E
E = ::= T
T = ::= id

The Grammar

1d = ::= P3
1l = ::= P2
E + E = ::= P1
E = ::= P0

Definitions

LR(0) items
LR items

We can build SLR parsers using LR(0) items and

LR(1) items

Several LR(1) items may have the same core.

Example LR(0) states

\[ \{ p, q \} \]
\[ q, p \]
\[ r, q \]
\[ r, p \]
The point of all these lookahead symbols?

What's the point of all these lookahead symbols?
For a string of grammar symbols $\alpha$, define $\text{FIRST}(\alpha)$ as the set of terminal symbols that begin strings derived from $\alpha$. For example, $\text{LR}(1)$ states:

$\text{FIRST}(\alpha)$ contains the set of tokens valid in the first position of a string of a grammar. If $\alpha \in \text{FIRST}(\alpha)$, then $\text{FIRST}(\alpha)$ is non-semantic, and the set of terminal symbols that begin strings derived from $\alpha$.

For a string of grammar symbols $\alpha$, define $\text{FIRST}(\alpha)$ as the set of terminal symbols that begin strings derived from $\alpha$. The set of terminal symbols $\text{FIRST}(\alpha)$ contains the set of tokens valid in the first position of a string of a grammar. If $\alpha \in \text{FIRST}(\alpha)$, then $\text{FIRST}(\alpha)$ is non-semantic, and the set of terminal symbols that begin strings derived from $\alpha$.
The augmented grammar

**Example ACTION and GOTO tables**

<table>
<thead>
<tr>
<th>State</th>
<th>Action/GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>S^0</td>
<td>Shift 3</td>
</tr>
<tr>
<td>S^1</td>
<td>Shift 4</td>
</tr>
<tr>
<td>S^2</td>
<td>Reduce P1</td>
</tr>
<tr>
<td>S^3</td>
<td>Reduce P3</td>
</tr>
<tr>
<td>P^1</td>
<td>Reduce P2</td>
</tr>
<tr>
<td>P^2</td>
<td>Reduce P3</td>
</tr>
<tr>
<td>S^4</td>
<td>Reduce P1</td>
</tr>
<tr>
<td>P^3</td>
<td>Reduce P1</td>
</tr>
</tbody>
</table>

The **reduce** actions are determined by the **LR** items. The **shift** actions are determined by the lookahead entries in the LR(1) items.

1. **LR(1) Table Construction**

   1. If \( S \) appears on the production, create
   2. Construct the collection of sets of LR(1) items
   3. Add augmented grammar \( G' \) by adding \( S' \)

The initial state of the parser is the state corresponding to the item containing the initial symbol.

4. If goto \( [I] = I^0 \) then set goto \( [I] \) to a.
5. If goto \( [I] = \epsilon \) then set action[I] to "accept".
6. If goto \( [I] \) to "accept", then set action[I] to a.
7. For all other entries in action and goto, set to an error.
Informally, we say that a grammar $G$ is LR(1) if we have two sets of LR(1) items $I_i$ and $I_j$ that can be parsed by a non-backtracking shift-reduce parser.

LR(1) grammars are a proper subset of the languages recognized by LL(1) (predictive) parsers.

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.

LR(0) grammars describe a proper superset of the languages described by a left-to-right scan of the input.

Efficient shift-reduce parsers can be implemented for LR(1) grammars.

For LR(1) grammars, any leftmost derivation using at most 1 token of lookahead past the end of the handle can be found as soon as possible in a left-to-right scan of the input.

LR parsers detect an error as soon as possible in a left-to-right scan of the input.

LR(1) parsers detect an error as soon as possible in a left-to-right scan of the input.

LR parsers detect an error as soon as possible in a left-to-right scan of the input.

LALR(1) parsers, which can be implemented for LR(1) grammars, virtually all context-free programming languages.

Properties

Large state tables longer to construct, run

States than LR(0) approximately 10 for Pascal

LR(1) parsers are powerful, but have many more
There are two approaches to constructing LALR parsers.

1. Build LALR(1) sets of items, then merge states with parse tree tables.

2. Build LR(0) sets of items, then propagate lookahead information.

- LALR may create reduce-reduce conflicts that LR does not have, but will catch errors before more input is processed.
- LALR may perform reduce rather than shift-reduce.
- LALR sets of items have the same number of states as LR sets. (Core LR(0) items are the same.)
- LALR(1) parsers have same number of states as LR(1) parsers.

- LALR properties

- LALR(1) improves over SR(1) parsers

- LALR(1) is useful for languages such as Yacc, Python, C++

- LALR(1) is derived from LR(0) with no shift-reduce conflict

- LALR(1) is derived from LR(0) with no shifty-reduce

- LALR(1) is used by utilities such as yacc, lex, parser...
A hand-coded recursive descent parser directly encodes a grammar. It has most of the linguistic limitations of LL(1), but it also has the virtues of LL(1) parsers. An LL(1) parser must be able to recognize the occurrence of the right-hand side of a production after having seen all that is derived from the right-hand side with k symbols of lookahead. An LL(1) parser must be able to recognize the occurrence of the right-hand side of a production after having seen all that is derived from that right-hand side with k symbols of lookahead. An LL(1) parser must be able to recognize the use of a production after seeing only the first k symbols of its right-hand side. An LL(1) parser must be able to recognize the occurrence of the right-hand side of a production after having seen all that is derived from the right-hand side with k symbols of lookahead. An LL(1) parser must be able to recognize the use of a production after seeing only the first k symbols of its right-hand side.