LALR parsing

LALR stands for look ahead left right. It is a technique for deciding when reductions have to be made in shift/reduce parsing.

Often, it can make the decisions without using a look ahead. Sometimes, a look ahead of 1 is required.

Most parser generators (and in particular Bison and Yacc) construct LALR parsers.

In LALR parsing, a deterministic finite automaton is used for determining when reductions have to be made. The deterministic finite automaton is usually called prefix automaton. On the following slides, I will explain how it is constructed.

Items

Let $\mathcal{G} = (\Sigma, R, S)$ be a grammar.

Definition Let $\sigma \in \Sigma,\ w_1, w_2 \in \Sigma^*$. If $\sigma \rightarrow w_1 \cdot w_2 \in R$, then $\sigma \rightarrow w_1.w_2$ is called an item.

An item is a rule with a dot added somewhere in the right hand side.

The intuitive meaning of an item $\sigma \rightarrow w_1.w_2$ is that w_1 has been read, and if w_2 is also found, then rule $\sigma \rightarrow w_1 w_2$ can be reduced.

Items

Let $a \rightarrow bBc$ be a rule. The following items can be constructed from this rule:

$$a \rightarrow .bBc, \quad a \rightarrow b.Bc, \quad a \rightarrow bB.c, \quad a \rightarrow bBc.$$

For a given grammar G, the set of possible items is finite.

Operations on Itemsets (1)

Definition: An itemset is a set of items.

Because for a given grammar, there exists only a finite set of possible items, the set of itemsets is also finite.

Let I be an itemset. The closure $\mathrm{CLOS}(I)$ of I is defined as the smallest itemset J, s.t.

- $I \subseteq J$,
- If $\sigma \rightarrow w_1.Aw_2 \in J$, and there exists a rule $A \rightarrow v \in R$, then $A \rightarrow .v \in J$.

Operations on Itemsets (2)

Let I be an itemset, let $\alpha \in \Sigma$ be a symbol. The set $\mathrm{TRANS}(I, \alpha)$ is defined as

$$\{\sigma \rightarrow w_1 \alpha . w_2 \mid \sigma \rightarrow w_1 . \alpha w_2 \in I \ \}.$$

The Prefix Automaton

Let $\mathcal{G} = (\Sigma, R, S)$ be a grammar. The prefix automaton of $\mathcal{G}$ is the deterministic finite automaton $\mathcal{A} = (\Sigma, Q, Q_s, Q_a, \delta)$, that is the result of the following algorithm:

- Start with $\mathcal{A} = (\Sigma, \{\text{CLOS}(I)\}, \{\text{CLOS}(I)\}, \emptyset, \emptyset)$, where $I = \{\hat{S} \rightarrow .S\ \#\}$, $\hat{S} \notin \Sigma$ is a new start symbol, S is the original start symbol of $\mathcal{G}$, and $\# \notin \Sigma$ is the EOF symbol.

- As long as there exists an $I \in Q$, and a $\sigma \in \Sigma$, s.t. $I' = \text{CLOS}(\text{TRANS}(I, \sigma)) \notin Q$, put

$$Q := Q \cup \{I'\}, \quad \delta := \delta \cup \{(I, \sigma, I')\}.$$

- As long as there exist $I, I' \in Q$, and a $\sigma \in \Sigma$, s.t. $I' = \text{CLOS}(\text{TRANS}(I, \sigma))$, and $(I, \sigma, I') \notin \delta$, put

$$\delta := \delta \cup \{(I, \sigma, I')\}.$$

The Prefix Automaton (2)

The prefix automaton can be big, but it can be easily computed.

Every context-free language has a prefix automaton, but not every language can be parsed by an LALR parser, because of the look ahead sets.

Parse Algorithm (1)

```
std::vector< state > states;
   // Stack of states of the prefix automaton.

std::vector< token > tokens;
   // We assume that a token has attributes, so
   // we don't encode them separately.

std::dequeue< token > lookahead;
   // Will never be longer than one.

states. push_back( q0 ); // The initial state.

while( true )
{
```

Parse Algorithm (2)

```
decision = unknown;

state topstate = states. back( );
if(topstate has only one reduction R and no shifts)
   decision = reduce(R);

// We know for sure that we need lookahead:

if( decision == unknown && lookahead. size( ) == 0 )
{
   lookahead. push_back( inputstream. readtoken( ));
}
```

Parse Algorithm (3)

```
if( lookahead. front( ) == EOF )
{
   if( topstate is an accepting state )
      return tokens. back( );
   else
      return error, unexpected end of input.
}
```

Parse Algorithm (4)

```
if( decision == unknown &&
    topstate has only one reduction R with
       lookahead. front( ) &&
    no shift is possible with lookahead. front( ))
{
   decision = reduce(R);
}
if( decision == unknown &&
    topstate has only a shift Q with
       lookahead. front( ) &&
    no reduction is possible with lookahead. front())
{
   decision = shift(Q);
}
```

Parse Algorithm (5)

```
if( decision == unknown )
{
   // Either we have a conflict, or the parser is
   // stuck.

   if( no reduction/no shift is possible )
      print error message, try to recover.
```

Parse Algorithm (6)

```
    // A conflict can be shift/reduce, or
    // reduce/reduce:

    Let R, from the set of possible reductions,
    (taking into account lookahead. front( )),
    be the rule with the smallest number.

    decision = reduce(R);
}
```

Parse Algorithm (7)

```
if( decision == push(Q))
{
   states. push_back( Q );
   tokens. push_back( lookahead. front( ));
   lookahead. pop_front( );
}
else
{
   // decision has form reduce(R)

   unsigned int n =
      the length of the rhs of R.
```

Parse Algorithm (8)

```
token lhs =
   compute_lhs( R,
      tokens. begin( ) + tokens. size( ) - n,
      tokens. begin( ) + tokens. size( ));
         // this also computes the attribute.

for( unsigned int i = 0; i < n; ++ i )
{
   states. pop_back( );
   tokens. pop_back( );
}
```

Parse Algorithm (9)

```
        // The shift of the lhs after a reduction is
        // also called 'goto'

        topstate = states. back( );
        state newstate =
           the state reachable from topstate under lhs.

        states. push_back( newstate );
        tokens. push_back( lhs );
     }
  }

  // Unreachable.
```

Lookahead Sets

We already have seen lookahead sets in action.

If a state has more than one reduction, or a reduction and a shift, the parser looks at the lookahead symbol, in order to decide what to do next.

$\mathrm{LA}(I,\ \sigma \rightarrow w) \subseteq \Sigma$ is defined a set of tokens. If the parser is in state I, and the lookahead $\in \mathrm{LA}(I,\ \sigma \rightarrow w)$, then the parser can reduce $\sigma \rightarrow w$.

When should a token σ be in $\mathrm{LA}(I,\ \sigma \rightarrow w)$?

Lookahead Sets (2)

Definition:

$s \in \mathrm{LA}(I,\ \sigma \to w)$ if

1. $\sigma \to w. \ \in I$ (obvious)
2. There exists a correct input word $w_1 \cdot s \cdot w_2 \cdot \#$, such that
3. The parser reaches a state with state stack $(\ldots, I)$ and token stack $(\ldots, w)$, the lookahead (of the parser) is s, and
4. the parser can reduce the rule $\sigma \to w$, after which
5. it can read the rest of the input w_2 and reach an accepting state.

Computing Look Ahead Sets

For every rule $A \rightarrow w$ of the grammar $\mathcal{G}$, such that there exist states I_1, I_2, I_3, s.t. $A \rightarrow .w \in I_1, \quad A \rightarrow w. \in I_2$, there exists a path from I_1 to I_2 in the prefix automaton using w, and there is a transition from I_1 to I_3 based on A, the following must hold:

- For every symbol $\sigma \in \Sigma$, for which a transition from I_3 to some other state is possible in the prefix automaton,
  $\sigma \in \mathrm{LA}(I_2, \quad A \rightarrow w.)$.
- For every item of form $B \rightarrow v. \ \in I_3$,
  $\mathrm{LA}(I_3, \quad B \rightarrow v.) \subseteq \mathrm{LA}(I_2, \quad A \rightarrow w.)$

Compute the LA as the smallest such sets.

Computing Look Ahead Sets (2)

Example

$$S \rightarrow Aa,$$

$$A \rightarrow B,$$

$$A \rightarrow Bb,$$

$$B \rightarrow C,$$

$$B \rightarrow Cc,$$

$$C \rightarrow d.$$

The algorithm on the previous slides can sometimes compute too big look ahead sets. You will see this in the exercises.

Computing the Correct Sets

I don't want to say much about this, because it is complicated.

Definition: An LR(1)-item has form $\sigma \rightarrow w_1.w_2/s$, where $\sigma \rightarrow w_1 w_2$ is a rule of the grammar, and $s \in S$.

STEP remains the same.

CLOS has to be modified.