Parsing Algorithms 2: LR Parsing

CS 4447 / CS 9545 -- Stephen Watt
University of Western Ontario

Readings

- Purple Dragon Chapter 3. Lexical analysis
- Purple Dragon Chapter 4. Parsing

LR(k) Parsing

Left-to-right scan, **R**ight-most derivation,
with **k** tokens of look-ahead.

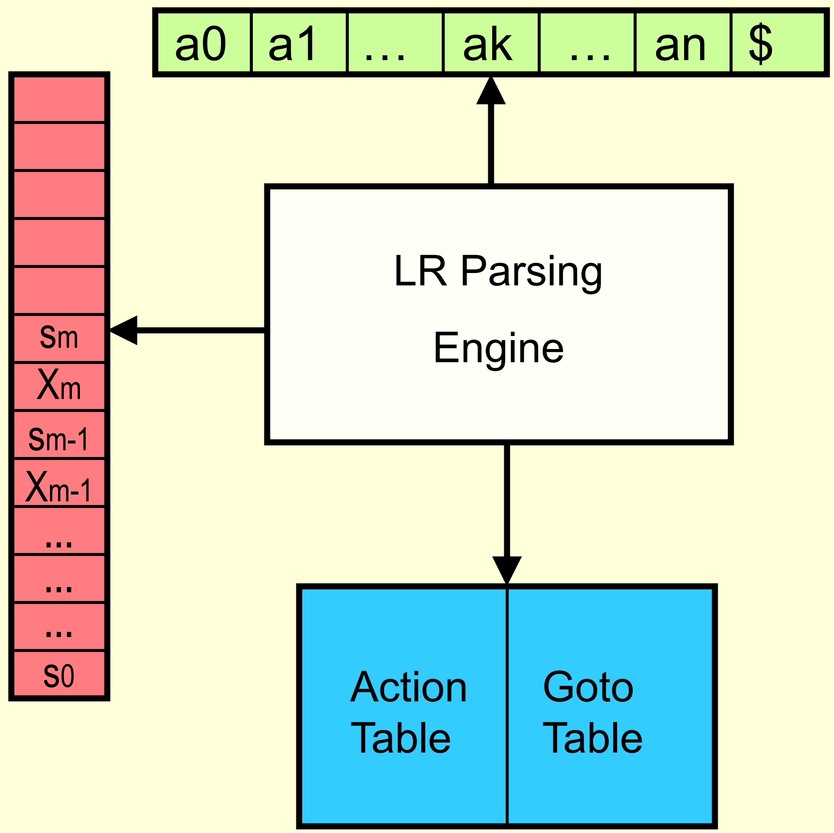
+ Most general non-backtracking shift-reduce parsers
+ Larger class of grammars than LL parsing
+ Detect syntax errors as soon as possible with left-to-right scan
- Tables not suitable to build by hand

Three Methods

- SLR – simple LR.
  Easiest to implement; least powerful.
- Canonical LR.
  Hard to implement; most powerful.
- LALR – look ahead LR.
  Relatively easy to implement; quite powerful.

Overall Idea

- Input string of tokens
- Stack of parser states
- Action and Goto tables
- Parsing engine



The LR Parsing Engine

- Stack contains
  - X[i] grammar symbols
  - s[i] “states”

- Action table gives, for each (s[i], a[j]) pair, one of
  - *shift* s[j], for some state j
  - *reduce* $A \rightarrow \beta$, for some production of the grammar
  - *accept* parsing is finished
  - *error* parser has discovered an error

- Goto table gives, for each state + grammar symbol, a new state.

Parser Configurations

- A pair
  - Stack contents
  - Rest of input
- For our figure

  $(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m\ ,\ a_k\ a_{k+1}\ \ldots\ a_n\ \$)$
- This corresponds to a mid-derivation form

  $X_1\ X_2\ \ldots\ X_m\ a_k\ a_{k+1}\ \ldots\ a_n\ \$$

  interleaved with parser states.

Parser Action

Config = (s0 X1 s1 X2 s2 … Xm sm, ak ak+1 … an $)

- If Action(s[m], a[k]) == shift s.
  s = Goto(s[m], a[k])
  Config = (s0 X1 s1 X2 s2 … Xm sm ak s, ak+1 … an $)

- If Action(s[m], a[k]) == reduce A → β
  r = |β|
  s = Goto(s[m-r], A)
  Config = (s0 X1 s1 X2 s2 … Xm-r sm-r A s, ak+1 … an $)

- If Action(s[m], a[k]) == accept
  accept

- If Action(s[m], a[k]) == error
  halt, or attempt error recovery

Example

1. E → E "+" T
2. E → T
3. T → T "*" F
4. T → F
5. F → "(" E ")"
6. F → **id**

State	Action						Goto		
	Id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Constructing Parsing Tables

- Build a Finite Automaton to recognize viable prefixes of rules.
- LR(0) items:

  $E \rightarrow \bullet\ E$ "+" T

  $E \rightarrow E \bullet$ "+" T

  $E \rightarrow E$ "+" $\bullet\ T$

  $E \rightarrow E$ "+" $T \bullet$

- Indicates how much of production has been seen
- Can be represented as (production #, dot position)

Closure of an Item Set

Given set of items I for grammar G, *closure*(I) is the set formed by:

- All elements of I are in closure(I)
- If $A \rightarrow \alpha \bullet B\ \beta$ is in closure(I) and $B \rightarrow \gamma$ is a production in G, then $B \rightarrow \bullet\ \gamma$ is in closure(I)

closure(I) captures the idea of finding all the rules that might be applicable at a given point.

Closure Example

- Augment previous grammar with E’ → E.
- Closure of {E’ → •E} is

  {E’ → •E,
  E → •E “+” T,   E → •T,
  T → •T “*”F,   T → •F,
  F → • “(” E “)”,   F → • **id**}

The Goto Operation

- Goto($\mathbb{I}$,X) for an item set $\mathbb{I}$ and grammar symbol X, is the set of items obtained by "moving the dot past X" in the items.

$\mathbb{J}$:= { }

for all $A \rightarrow \alpha \bullet X \beta$ in $\mathbb{I}$, add $A \rightarrow \alpha X \bullet \beta$ to $\mathbb{J}$.

return closure($\mathbb{J}$).

Constructing the Automaton

Initialize T to { closure({S' → • S $}) }

Initialize E to { }

repeat

    for each state I in T

        for each item A → α • X β in I

            J := Goto(I,X)

            T := T U { J }

            E := E U { I →[X] J }

until E and T do not change

Note that for X = $ we do not compute Goto(I,$).

Instead we make an accept action.

Constructing the Tables

- For each edge I $\rightarrow$[X] J
  - If X is a terminal,
    put the action *shift J* at position (I,X) of the table.
  - If X is a nonterminal,
    put *goto J* at position (I,X)
- For each state I containing an item S' $\rightarrow$ S • $,
  - put an accept action at (I, $)
- For a state containing A $\rightarrow$ γ •
  (production *n* with a dot at the end),
  put *reduce n* at (I, K) for every token K.

LR(1) Items

- Some languages cannot be handled with LR(0).
  Some look-ahead is needed.
- An LR(1) item is of the form

  [$A \rightarrow \alpha \bullet \beta$, a],

  for A a non-terminal, a a terminal, α and β strings of terminals and non-terminals.
- The terminal a is the “look-ahead”.

  It has no effect when β is non-empty.

  When β *is* empty, *i.e.* for [$A \rightarrow \alpha \bullet$, a], the item says to reduce the production $A \rightarrow \alpha \bullet$

Closure with LR(1) Items

- Compute the closure of a set I of LR(1) in items with grammar G' as follows:

Closure(I) == {
  repeat
    for each item [$A \rightarrow \alpha \bullet B\beta$, a] in I repeat
      for each production $B \rightarrow \gamma$ in G' repeat
        for each terminal b in FIRST(βa) repeat
          I := I $\cup$ { [$B \rightarrow \bullet\gamma$, b] }
  until I stops growing
  return I
}

Goto with LR(1) Items

- Compute the goto of a set I of LR(1) in items with grammar G' as follows:

```
Goto(I, X) == {
   J := { }
   for each item [ A →α • Xβ, a] in I  repeat
            J := J ∪ { [ A →α X• β, a] }
   return Closure(J)
}
```

Computing Valid LR(1) Items

- Many potential LR(1) items will not be used.
  Compute the needed ones as follows:

```
Items(G’) == {
  C := { Closure( { [S’ → • S, $] } ) }
  repeat
     for each item set  I in C repeat
        for each grammar symbol X repeat
           J := Goto[I,X]
           if J nonempty and J not in C then C := C ∪ { J }
  until C stops growing
```

Constructing an LR(1) Parser

- To build the automaton, use the new definitions of Closure and Goto in the previous algorithm.

- To build the tables, change

  For a state containing $A \to \gamma \bullet$
  (production *n* with a dot at the end),
  put *reduce n* at (I, K) for every token K.

  to

  For a state containing $A \to [\gamma \bullet, a]$
  (production *n* with a dot at the end),
  put *reduce n* at (I, a).