LR(1)

SLR(1) parsers may not be able to parse some LR grammars.

Problem is that lookahead information is added to LR(0) parser at the end of construction.

We can get more powerful parser by keeping track of lookahead information in the states of the parser.

If, in a single left-to-right scan, we can construct a reverse rightmost derivation, while using at most a single token lookahead to resolve ambiguities, then the grammar is LR(1)

Of course, we would like a more formal definition.
Unfortunately, that requires some more notation.

LR(1) gammars

Given these definitions, we can formally define an LR(1) grammar. An augmented grammar G is LR(1) if the three conditions

1. Start $=>^*$ $\alpha A w$ $=>^*$ $\alpha\beta w$,
2. Start $=>^*$ $\gamma B x$ $=>^*$ $\alpha\beta y$,
3. FIRST(w) = FIRST(y)

imply that $\alpha A w = \gamma B x$
(That is, $\alpha = \gamma$, A = B, and w = x)

To extend this to LR(k) grammars, we define $\text{FIRST}_k(\alpha)$ as the leading k symbols that begin strings derived from α
The definition extends naturally by changing rule 3

(Note) The "augmented grammar" is one where the start symbol appears only on the lhs of productions
For the rest of LR parsing, we will assume the grammar is augmented with a production S' ::= S

LR(k) items

The table construction algorithm use LR(k) items to represent the set of possible states in a parse

An LR(k) item is a pair [α,β], where

α is a production from G with a • at some position in the rhs
β is a lookahead string containing k symbols(terminals or eof)

What about LR(1) items?

- Example LR(1) item: [A ::= X•YZ, a]
- LR(1) items have lookahead strings of length 1
- several LR(1) items may have the same *core*

  [A ::= X•YZ, a]
  [A ::= X•YZ, b]

  we represent this as

  [A ::= X•YZ, {a, b}]

LR(1) lookahead

What's the point of all these lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping, unless item has • at right end.
    - in [A ::= X•YZ, a], a has no direct use
    - in [A ::= XYZ•, a], a is useful
- allows use of grammars that are not uniquely invertible

(Note) G is uniquely invertible if no two productions have the same rhs

Recall, the SLR(1) construction uses LR(0) items!

The point

For [A ::= α•, a] and [B ::= α•, b], we can decide between reducing to A and to B by looking at limited right context.

Canonical LR(1) items

The canonical collection of LR(1) items:

- set of items derivable from [S' ::= •S, eof]
- set of all items that can derive the final configuration

Essentially,

- each set in the canonical collection of sets of LR(1) items represents a state in an NFA that recognizes viable prefixes.
- Grouping together is really the subset construction, see 3.6

To construct the canonical collection we need two functions:

- closure(I)
- goto(I, X)

LR(1) closure

Given an item [A ::= $\alpha \bullet B\beta$, a], its closure contains the item and any other items that can generate legal substrings to follow α.

Thus, if the parser has viable prefix α on its stack, the input should reduce to $B\beta$

(or γ for some other item [B ::= $\bullet\gamma$, b] in the closure).

To compute closure(I)

```
function closure(I)
    repeat
        new_item ← false
        for each item [A ::= α•Bβ, a] ∈ I,
            each production B ::= γ ∈ G',
            and each terminal b ∈ FIRST(βa),
            if [B ::= •γ, b] ∉ I then
                add [B ::= •γ, b] to I
                new_item ← true
            endif
    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

$[A ::= \alpha X\bullet\ \beta, a]$ such that $[A ::= \alpha\bullet X\beta, a] \in I$

If I is the set of valid items for some viable prefix γ, then goto(I, X) is the set of valid items for the viable prefix γX.

goto(I, X) represents state after recognizing X in state I.

To compute goto(I, X)

```
function goto(I, X)
        J ← set of items [A ::= αX•β, a]
                such that [A ::= α•Xβ, a] ∈ I
        J' ← closure(J)
        return J'
```

Collection of sets of LR(1) items

We start the construction of the collection of sets of LR(1) items with the item [S’ ::= • S, eof], where

S’ is the start symbol of the augmented grammar G’
S is the start symbol of G, and eof is the right end of string marker

To compute the collection of sets of LR(1) items

```
procedure items(G’)
   C ← closure({[S’ ::= • S, eof]})
   repeat
      new_item ← false
      for each set of items I in C and
          each grammar symbol X such that
         goto(I, X) ≠ ϕ and goto(I, X) ∉ C
            add goto(I, X) to C
            new_item = true
      endfor
   until (new_item = false)
```

LR(1) table construction

The Algorithm:

1. construct the collection of sets of LR(1) items for G'
2. State i of the parser is constructed from I_i
   A. if $[A ::= \alpha \bullet a\beta, b] \in I_i$ and $goto(I_i, a) = I_j$, then set action[i, a] to "shift j". (a must be a terminal)
   B. if $[A ::= \alpha \bullet, a] \in I_i$, then set action[i, a] to "reduce $A ::= \alpha$" .
   C. if $[S' ::= S\bullet, eof] \in I_i$, then set action[i, eof] 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 $[S' ::= \bullet\ S, eof]$

Example

The Grammar

1	goal	::=	expr
2	expr	::=	term + expr
3		|	term
4	term	::=	factor * term
5		|	factor
6	factor ::= id		

	Action				GOTO		
	id	+	*	eof	expr	term	factor
S_0	s4	-	-	-	1	2	3
S_1	-	-	-	acc	-	-	-
S_2	-	s5	-	r3	-	-	-
S_3	-	r5	s6	r5	-	-	-
S_4	-	r6	r6	r6	-	-	-
S_5	s4	-	-	-	7	2	3
S_6	S4	-	-	-	-	8	3
S_7	-	-	-	r2	-	-	-
S_8	-	r4	-	r4	-	-	-

Example

Step 1

$I_0 \leftarrow \{[g ::= \bullet\ e, eof]\}$

$I_0 \leftarrow closure(I_0)$

{ [g ::= • e, eof], [e ::= • t + e, eof],
[e ::= • t, eof], [t ::= • f * t, +],
[t ::= • f * t, eof], [t ::= • f, +],
[t ::= • f, eof], [f ::= • id, +],
[f ::= • id, eof], [f ::= • id, *]}

Iteration 1

$I_1 \leftarrow goto(I_0, e)$

$I_2 \leftarrow goto(I_0, t)$

$I_3 \leftarrow goto(I_0, f)$

$I_4 \leftarrow goto(I_0, id)$

Iteration 2

$I_5 \leftarrow goto(I_2, +)$

$I_6 \leftarrow goto(I_3, *)$

Iteration 3

$I_7 \leftarrow goto(I_5, e)$

$I_8 \leftarrow goto(I_6, t)$

Example

I_0: [g ::= • e, eof], [e ::= • t + e, eof],
[e ::= • t, eof], [t ::= • f * t,{+, eof}],
[t ::= • f, {+, eof}], [f ::= • id, {+, *, eof}]

I_1: [g ::= e •, eof]

I_2: [e ::= t •, eof], [e ::= t • + e, eof]

I_3: [t ::= f •, {+, eof}], [t ::= f • * t,{+, eof}]

I_4: [f ::= id •, {+, *, eof}]

I_5: [e ::= t + • e, eof], [e ::= • t + e, eof],
[e ::= • t, eof], [t ::= • f * t,{+, eof}],
[t ::= • f, {+, eof}], [f ::= • id, {+, *, eof}]

I_6: [t ::= f * • t,{+, eof}], [t ::= • f * t,{+, eof}],
[t ::= • f, {+, eof}], [f ::= • id, {+, *, eof}]

I_7: [e ::= t + e •, eof]

I_8: [t ::= f * t •,{+, eof}]

LR(1) table construction algorithm

1. build I, the canonical collection of sets of LR(1) items
   A. $I_0 \leftarrow$ closure({[S' $\rightarrow$ •S, eof]})
   B. Repeat until no sets are added
      for $I_j \in$ I and X $\in$ NT $\cup$ T
         if goto(I_j, X) is a new set, add it to I
2. iterate through $I_j \in$ I, filling in the ACTION table
3. fill in the GOTO table

What does I_j “mean”?

LR(1) parser example

The Grammar

1	E	::=	T + E
2		|	T
3	T	::=	id

The Augmented Grammar

0	S’	::=	E
1	E	::=	T + E
2		|	T
3	T	::=	id

Symbol	FIRST	FOLLOW
S’	{id}	{eof}
E	{id}	{eof}
T	{id}	{+, eof}

Example LR(0) states

S_0: [S' ::= • E, $],
[E ::= • T + E, $],
[E ::= • T, $],
[T ::= • id, +],
[T ::= • id, $]

S_1: [S' ::= E •, $]

S_2: [E ::= T • + E, $],
[E ::= T •, $]

S_3: [T ::= id •, +]
[T ::= id •, $]

S_4: [S' ::= T + • E, $],
[E ::= • T + E, $],
[E ::= • T, $],
[T ::= • id, +]
[T ::= • id, $]

S_5: [E ::= T + E •, $]

Example GOTO function

Start

$S_0 \leftarrow$ closure({[S ::= $\bullet$ E]})

Iteration 1

goto(S_0, E) = S_1

goto(S_0, T) = S_2

goto(S_0, id) = S_3

Iteration 2

goto(S_2, +) = S_4

Iteration 3

goto(S_4, id) = S_3

goto(S_4, E) = S_5

goto(S_4, T) = S_2

Example Action and GOTO tables

The Augmented Grammar

0	S’	::=	E
1	E	::=	T + E
2		|	T
3	T	::=	id

	Action			GOTO	
	Id	+	$	E	T
S_0	shift 3	-	-	1	2
S_1	-	-	accept	-	-
S_2	-	shift 4	reduce 2	-	-
S_3	-	reduce 3	reduce 3	-	-
S_4	shift 3	-	-	5	2
S_5	-	-	reduce 1	-	-

The “reduce” actions are determined by the lookahead entries in the LR(1) items(instead of FOLLOW as in SLR parsers)

The dfa, ACTION and GOTO tables have the exact same format for both SLR(1) and LR(1) parsers

Resolving parse conflicts

Parse conflicts possible when certain LR items are found in the same state.

Depending on parser, may choose between LR items using lookahead.

Legal lookahead for LR items must be disjoint, else conflict exists.

	Shift-Reduce [$A ::= \alpha \bullet , \delta$] [$B ::= \beta \bullet \gamma, \eta$]	Reduce-Reduce [$A ::= \alpha \bullet , \delta$] [$B ::= \beta \bullet , \eta$]
LR(0)	conflict	conflict
SLR(1)	FOLLOW(A) $\cap$ FIRST(γ)	FOLLOW(A) $\cap$ FOLLOW(B)
LR(1)	$\delta \cap$ FIRST(γ)	$\delta \cap \eta$

SLR(1) parsing example

The Grammar

0	S’	::=	G
1	G	::=	E = E | id
2	E	::=	E + T | T
3	T	::=	T * id | id

S_0: [S’ ::= • G]
[G ::= • E = E]
[G ::= • id]
[E ::= • E + T]
[E ::= • T]
[T ::= • T * id]
[T ::= • id]

S_1: [G ::= id •] FOLLOW(G) = {$}
[T ::= id •] FOLLOW(T) = {$, *, +, =}

Reduce-reduce conflict in S_1 for lookahead $!

LR(1) parsing example

The Grammar

0	S’	::=	G
1	G	::=	E = E | id
2	E	::=	E + T | T
3	T	::=	T * id | id

S_0: [S’ ::= • G, {$}]
[G ::= • E = E, {$}]
[G ::= • id, {$}]
[E ::= • E + T, {=, +}]
[E ::= • T, {=, +}]
[T ::= • T * id, {=, +, *}]
[T ::= • id, {=, +, *}]

S_1: [G ::= id •, {$}]
[T ::= id •, {=, +, *}]

Reduce-reduce conflict in S_1 resolved by lookahead