Shift-reduce parsing

Definitions

- a *right-sentential form* is any string that may occur in a legal rightmost derivation
- a *viable prefix* of a right-sentential form is any prefix that does not continue past the right end of its rightmost handle

Shift-reduce parsers

- operator precedence

  define precedence between operands to guide reductions
- SLR(1) = LR(0) + FOLLOW

  construct DFA for recognizing viable prefix, use FOLLOW to guide reductions
- LR(1)

  construct DFA for recognizing viable prefix, storing lookahead information in DFA
- LALR(1)

  construct DFA for recognizing viable prefix, propagating lookahead information in DFA

LR(0) items

An *LR(0) item* is a string $[\alpha]$, where

α is a production from G with a $\bullet$ at some position in the *rhs*

The $\bullet$ indicates how much of an item we have seen at a given state in the parse.

$[A ::= \bullet XYZ]$ indicates that the parser is looking for a string that can be derived from XYZ

$[A ::= XY \bullet Z]$ indicates that the parser has seen a string derived from XY and is looking for one derivable from Z

LR(0) Items (*no lookahead*)

$A ::= XYZ$ generates 4 *LR(0)* items.

1. $[A ::= \bullet XYZ]$
2. $[A ::= X \bullet YZ]$
3. $[A ::= XY \bullet Z]$
4. $[A ::= XYZ \bullet]$

LR(0) machine

Definitions

- *closure* of $[A ::= \alpha \bullet B\beta]$ contains itself and any items of form $[B ::= \bullet\gamma]$, repeat for new items.
- *goto*(X) of $[A ::= \alpha \bullet X\beta]$ contains the closure of $[A ::= \alpha X \bullet \beta]$.

LR(0) DFA construction

1. begin with closure of start symbol $[S ::= \bullet\alpha]$
2. for each state, calculate *goto*(X) for all grammar symbols X, generating states
3. repeat step 2 for all newly generated states

Properties

- states in the DFA are sets of *LR*(0) items
- states represent viable prefixes of productions
- to recognize viable prefixes of language, save state of current production on stack when reducing new nonterminal

LR(0) items

The Grammar

P1	E	::=	T + E
P2		|	T
P3	T	::=	id

The Augmented Grammar

P0	S′	::=	E
P1	E	::=	T + E
P2		|	T
P3	T	::=	id

Example LR(0) states

S_0: [S′ ::= • E],
[E ::= • T + E],
[E ::= • T],
[T ::= • `id`]

S_1: [S′ ::= E •]

S_2: [E ::= T • + E],
[E ::= T •]

S_3: [T ::= `id` •]

S_4: [E ::= T + • E],
[E ::= • T + E],
[E ::= • T],
[T ::= • `id`]

S_5: [E ::= T + E •]

LR(1) items

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

But, we can get more powerful parsers by keeping track of lookahead information in the states of the LR parser.

An *LR(k) item* is a pair $[\alpha, \beta]$, 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`)

LR(1) items

- example: $[A ::= X \bullet YZ, \texttt{a}]$
- several *LR*(1) items may have the same *core*

  $[A ::= X \bullet YZ, \texttt{a}]$
  $[A ::= X \bullet YZ, \texttt{b}]$

  we represent this as

  $[A ::= X \bullet YZ, \{\texttt{a, b}\}]$

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 $\bullet$ at right end.
  - — in $[A ::= X \bullet YZ, \texttt{a}]$, a has no direct use
  - — in $[A ::= XYZ\bullet, \texttt{a}]$, a is useful
- allows use of (non-invertible) grammars where productions have the same *rhs*

The point

For $[A ::= \alpha\bullet, \texttt{a}]$ and $[B ::= \alpha\bullet, \texttt{b}]$, we can decide between reducing to A and to B by looking at limited right context!

LR(1) machine

Definitions

- *closure* of $[A ::= \alpha \bullet B\beta, \texttt{a}]$ contains itself and any items of form $[B ::= \bullet\gamma, \text{FIRST}(\beta\ \texttt{a})]$, repeat for new items.
- *goto*(X) of $[A ::= \alpha \bullet X\beta, \texttt{a}]$ contains the closure of $[A ::= \alpha X \bullet \beta, \texttt{a}]$.

LR(1) DFA construction

1. begin w/ closure of start symbol $[S ::= \bullet\alpha, \texttt{eof}]$
2. for each state, calculate *goto*(X) for all grammar symbols X, generating states
3. repeat step 2 for all newly generated states

Properties

- $[A \rightarrow X \bullet YZ, \alpha] \Rightarrow$ have recognized X & YZ would be valid
- $[A \rightarrow X \bullet YZ, \alpha] \Rightarrow [Y \rightarrow \bullet\beta, \gamma]$ & $[Y \rightarrow \bullet\delta, \eta]$ are also valid, where $\gamma, \eta \in \text{FIRST}(Z\alpha)$
- recognizing Y takes parser to $[A \rightarrow XY \bullet Z, \alpha]$

The FIRST set

For a string of grammar symbols α, define FIRST(α) as

- the set of terminal symbols that begin strings derived from α
- if $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in$ FIRST(α)

FIRST(α) contains the set of tokens valid in the first position of α

To build FIRST(X):

1. if X is a terminal, FIRST(X) is $\{X\}$
2. if $X ::= \epsilon$, then $\epsilon \in$ FIRST(X)
3. if $X ::= Y_1 Y_2 \cdots Y_k$ then put FIRST(Y_1) in FIRST(X)
4. if X is a non-terminal and $X ::= Y_1 Y_2 \cdots Y_k$, then $a \in$ FIRST(X) if $a \in$ FIRST(Y_i) and $\epsilon \in$ FIRST(Y_j) for all $1 \leq j < i$
   (If $\epsilon \notin$ FIRST(Y_1), then FIRST(Y_i) is irrelevant, for $1 < i$)

Example LR(1) states

S_0: [S′ ::= • E , $],
[E ::= • T + E , $], FIRST(ϵ $) = $
[E ::= • T , $], FIRST(ϵ $) = $
[T ::= • id , +] FIRST(+ E $) = +
[T ::= • id , $] FIRST(ϵ $) = $

S_1: [S′ ::= E • , $]

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

S_3: [T ::= id • , +]
[T ::= id • , $]

S_4: [E ::= T + • E , $],
[E ::= • T + E , $], FIRST(ϵ $) = $
[E ::= • T , $], FIRST(ϵ $) = $
[T ::= • id , +] FIRST(+ E $) = +
[T ::= • id , $] FIRST(ϵ $) = $

S_5: [E ::= T + E • , $]

LR(1) table construction

The Algorithm

1. if S appears on *rhs* of production, create augmented grammar G' by adding $S' ::= S$
2. construct the collection of sets of *LR(1)* items for G'.
3. State i of the parser is constructed from I_i.
   (a) if $[A ::= \alpha \bullet \texttt{a}\beta, \texttt{b}] \in I_i$ and $\text{goto}(I_i, \texttt{a}) = I_j$, then set `action[i, a]` to "*shift j*". (`a` must be a terminal)
   (b) if $[A ::= \alpha\bullet, \texttt{a}] \in I_i$, then set `action[i, a]` to "*reduce* $A ::= \alpha$".
   (c) if $[S' ::= S\bullet, \texttt{eof}] \in I_i$, then set `action[i, eof]` to "*accept*".
4. If $\text{goto}(I_i, A) = I_j$, then set `goto[i, A]` to j.
5. All other entries in `action` and `goto` are set to "*error*"
6. The initial state of the parser is the state constructed from the set containing the item $[S' ::= \bullet S, \texttt{eof}]$.

Example ACTION and GOTO tables

The Augmented Grammar

P0	S′	::=	E
P1	E	::=	T + E
P2		|	T
P3	T	::=	id

	ACTION			GOTO	
	id	+	$	E	T
S_0	shift 3	—	—	1	2
S_1	—	—	accept	—	—
S_2	—	shift 4	reduce P2	—	—
S_3	—	reduce P3	reduce P3	—	—
S_4	shift 3	—	—	5	2
S_5	—	—	reduce P1	—	—

The "reduce" actions are determined by the lookahead entries in the LR(1) items

LR(1) grammars

Informally, we say that a grammar G is LR(1) if we can find the sequence of handles for a reverse rightmost derivation using at most 1 token of lookahead past the end of the handle.

Properties

- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- efficient shift-reduce parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers

LALR(1) parsers

Problem

- LR(1) parsers are powerful, but have many more states than LR(0)(approximately ×10 for Pascal)
- larger state tables longer to construct, run

LALR(1) parsers

- define the *core* of a set of $LR(1)$ items to be the set of $LR(0)$ items derived by ignoring the lookahead symbols.
- example of two sets of LR(1) items with same core:
  - $\{[A \Rightarrow \alpha \bullet \beta, \texttt{a}], [A \Rightarrow \alpha \bullet \beta, \texttt{b}]\}$, and
  - $\{[A \Rightarrow \alpha \bullet \beta, \texttt{c}], [A \Rightarrow \alpha \bullet \beta, \texttt{d}]\}$
- 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
- almost as powerful as LR(1), same size as LR(0)

LALR(1) parsers

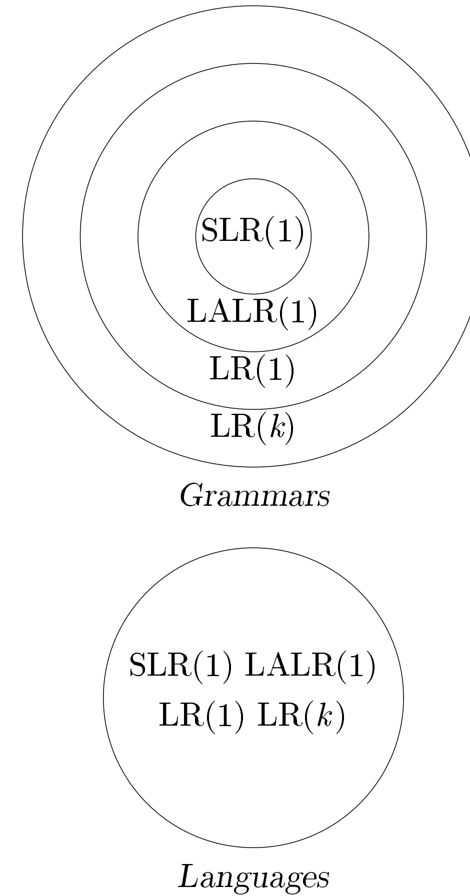
There are two approaches to constructing LALR(1) parsing tables

1. build LR(1) sets of items, then merge states with same core
2. build LR(0) sets of items, then propagate lookahead information.

LALR(1) properties

- LALR(1) parsers have same number of states as LR(0) parsers (core LR(0) items are the same)
- may perform *reduce* rather than *error*, but will catch error before more input is processed
- LALR derived from LR with no shift-reduce conflict will also have no shift-reduce conflict
- LALR may create reduce-reduce conflict not in LR from which LALR is derived
- used by utilities such as `yacc`, `bison`, `cup`

LR(k) languages



Parsing review

Recursive Descent A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

LL(k) An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.

Operator Precedence An ad hoc shift-reduce parser suitable for small expression grammars.

LR(k) An LR(k) 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.

Parsing review

	Advantages	*Disadvantages*
top-down recursive descent	fast locality simplicity error detection	hand-coded maintenance no left recursion associativity
LL(1)	simple method fast automatable	LL(1) $\subset$ LR(1) no left recursion associativity
operator precedence	simple method very fast small table associativity	L(G) $\neq$ L(parser) error detection no ambiguity
LR(1)	fast *det. langs.* early error det. automatable associativity	working sets table size error recovery