LALR(1) parsing

LR(1) parsers have many more states than SLR(1) parsers(approximately factor of ten for Pascal).

LALR(1) parsers have same number of states as SLR(1) parsers, but with more power due to lookahead in states.

Define the core of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.

Thus the two sets

- $\{[A \Rightarrow \alpha \bullet \beta, a], [A \Rightarrow \alpha \bullet \beta, b]\}$, and
- $\{[A \Rightarrow \alpha \bullet \beta, c], [A \Rightarrow \alpha \bullet \beta, d]\}$,

have the same core.

Key Idea:

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.

LALR(1) table construction

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

Approach 1: Build LR(1) sets of items, then merge.

1. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union
2. Update the goto function to reflect the replacement sets

The resulting algorithm has large space requirements

LALR(1) table construction

A more space efficient algorithm can be derived by observing that:

- We can represent I_i by its kernel, those items that are either the initial item [S' $\rightarrow \bullet$ S, eof] or do not have the $\bullet$ at the left end of the rhs.
- We can compute shift, reduce, and goto actions for the state derived from I_i directly from kernel(I_i)

This method avoids building the complete canonical collection of sets of LR(1) items.

Approach 2: Build LR(0) sets of items, then generate lookahead information.

1. Constuct kernels of LR(0) sets of items
2. Initialize lookaheads of each kernel item
3. Compute when lookahead propagate
4. Propagate lookaheads

LALR(1) properties

LALR(1) parsers have same number of states as SLR(1) 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.

Operator precedence parsers

Another approach to shift-reduce parsing is to use operator precedence.

Given $S \Rightarrow^{*} \alpha S_1 S_2 \beta$, there are three possible precedence relations between S_1 and S_2.

1. S_1 in handle, S_2 not $\quad S_1 > S_2$

   (S_1 reduced before S_2)

2. both in handle $\quad S_1 = S_2$

   (reduced at the same time)

3. S_2 in handle, S_1 not $\quad S_2 > S_1$

   (S_2 reduced before S_1)

A handle is thus composed of:

$<>$, $< = >$, $< = = >$, …

To decide whether to shift or reduce, compare top of stack with lookahead(ignoring nonterminals):

- Shift if $<$ or $=$
- Reduce if $>$

Left end of handle is marked by first $<$ found

Parsing example

The grammar

E ::= E + E | E * E | id

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>		>
$	<	<	<	>

Stack	Input	Precedence
$	id + id * id $	$ < id
$ < id	+ id * id $	id > +
$ < E	+ id * id $	$ < +
$ < E +	id * id $	+ < id
$ < E + < id	* id $	id > *
$ < E + < E	* id $	+ < *
$ < E + < E *	id $	* < id
$ < E + < E * id	$	id > $
$ < E + < E * E	$	* > $
$ < E + E	$	+ > $
$ < E	$	$ > $

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.

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	error detection no ambiguity
LR(1)	det. Langs. early error det. automatable	Working sets Table size error recovery

Error recovery in LL(1) parsers

Key notion:

- for each non-terminal, construct a set of terminals on which the parser can synchronize.
- When an error occurs looking for A, scan until an element of SYNCH(A) is found, then pop A and continue.

Building SYNCH:

1. $a \in$ FOLLOW(A) => $a \in$ SYNCH(A)
2. place keywords that start statements in SYNCH(A)
3. add symbols in FIRST(A) to SYNCH(A)

If we can't match a terminal on the top of stack:

1. pop the terminal
2. print a message saying the terminal was inserted
3. continue the parse

Error recovery in shift-reduce parsers

The problem

- encounter an invalid token
- bad pieces of tree hanging from stack
- incorrect entries in symbol table

We want to parse the rest of the file

Restarting the parser

- find a restartable state on the stack
- move to a consistent place in the input
- print an informative message(line number)

Error recovery in yacc

Yacc's error mechanism

- designated token error
- valid in any production
- error shows synchronization

When an error is discovered

- pops the stack until error is legal
- skips input tokens until it matches 3 tokens
- error productions can have actions

This mechanism is fairly general

Error recovery in yacc

stmt_list : stmt
    | stmt_list ; stmt

can be augmented with error

stmt_list : stmt
    | error
    | stmt_list ; stmt

this should

- throw out the errorneous statement
- synchronize at “;” or “end”
- invoke yyerror(“syntax error”)

Other “natural” places for errors

- all the “lists”
- missing parentheses or brackets
- extra operator or missing operator