SLR(1)

- Idea: start with LR(0) configuration sets and then add lookahead.
- Why? If lookahead symbol l signals that production B $\rightarrow \alpha$ $\bullet$ should be reduced, then l must legally be able to follow B (after reduction they will be adjacent).
- Algorithm idea: reduce configuration B $\rightarrow \alpha$ $\bullet$ if lookahead symbol is in Follow(B)
- Reduce$_i$ if B $\rightarrow \alpha$ $\bullet$ $\in$ s, a $\in$ Follow(B), and production i is B $\rightarrow \alpha$
- If A $\rightarrow \alpha$ $\bullet$ a β $\in$ s for a $\in$ V$_t$, then Shift else Error
- If more than one action is possible, have conflict (shift/reduce or reduce/reduce) and grammar is not SLR(1).

Number of states? Size of action table?

Sample Grammar - G_3

S $\rightarrow$ E $\vdash$
E $\rightarrow$ E + T | T
T $\rightarrow$ T * P | P
P $\rightarrow$ id | (E)

LR(1) and not LR(0).

CSFM for G_3

Conflicts in states 7 and 11.

Figure 6.18 on p. 164

SLR(1) action function for G_3

Figure 6.19 on p. 165

G_3 Parse Table

Figure 6.35, p. 191

LALR - Theory

2 ways to construct LALR(1) Tables

- Construct LR(1) parser and merge states
- Similar to SLR(1) construction, start with LR(0) parser and propagate *lookahead* sets

Any state in LR(1) can be mapped to a state in CFSM by removing the lookahead:

A $\rightarrow$ a $\bullet$, $\{$b, c$\}$

B $\rightarrow$ a $\bullet$, $\{$d$\}$

is mapped many-to-one to $\bar{s}$

A $\rightarrow$ a $\bullet$

B $\rightarrow$ a $\bullet$

Cognate($\bar{s}$) = $\{$c $\mid$ c $\in$ s, Core(s) = $\bar{s}\}$

Construct action table as:

- Reduce$_i$ if B $\rightarrow \alpha \bullet$, a $\in$ Cognite(s) and production i is B $\rightarrow \alpha$
- If A $\rightarrow \alpha \bullet$ a $\beta \in$ s for a $\in$ V$_t$, then Shift else Error

Example - Cognate States for G_3

LALR(1) Cognate State	LR(1) States with Common Core
State 3	States 3, 17
State 4	States 4, 14
State 5	States 5, 10
State 6	States 6, 18
State 7	States 7, 19
State 8	States 8, 21
State 9	States 9, 22
State 10	States 13, 15
State 11	States 11, 20
State 12	States 12, 16

LALR(1) Machine for G_3

Figure 6.22, p. 169

When is LALR(1) Needed?

G_5: prog $\rightarrow$ stmts

stmts $\rightarrow$ stmt; stmts

stmt $\rightarrow$ ID

stmt $\rightarrow$ var := expr

var $\rightarrow$ ID

var $\rightarrow$ ID [expr]

expr $\rightarrow$ var

Problem: ; $\in$ Follow(stmt) and

; $\in$ Follow(var) since expr $\rightarrow$ var

Part of the CFSM for G_5

Why will this work for LALR(1)?

Optimizing LR Parsers

- Optimizing LR(1) and LALR(1) parser tables has received a lot of attention. Some of the optimization techniques listed below
- Merge states with identical parsing actions
- Make a bit matrix of errors. Then fill the parse tables with actions that make states identical so they can be merged.
- Eliminate insignificant single reductions, i.e.

  E $\leftarrow$ T ; T $\leftarrow$ F ; F $\leftarrow$ P ; P $\leftarrow$ id $\quad\Rightarrow\quad$ E $\leftarrow$ id

LR Parsing & Actions

- For LALR(1) and SLR(1) parsers semantic and/or code generation actions can only be invoked when a reduction is applied.
  This is equivalent to requiring that such actions occur only at the right end of a rule.
- It is usually necessary to add extra productions to a grammar that provide a hook for attaching actions. yacc does this automatically.
- Example

  ifStatement $\rightarrow$ ifHead statement elsePart **end** C9

  ifHead $\rightarrow$ **if** expression **then** S13 S11 C8

  elsePart $\rightarrow$

  $\rightarrow$ elseHead statement

  elseHead $\rightarrow$ **else** C7 C12 C9

Parser Error Recovery

- Both LL(1) and LR(1) have the *valid prefix* property:
  if no error has been detected then
  the input thus far is a valid prefix of one or more programs.
- Both kinds of parsers can give error messages in terms of input symbols without mentioning the grammar (i.e. names of nonterminal symbols)
- Just look along current row of parse table for non-error entries and list the column headings.
- Changing the parser stack can cause problems in later compiler phases, e.g. compiler internal data structures may be left in an inconsistent state.
  If and only if the parser has the valid prefix property, we can avoid changing the parser stack.

Syntax Error Repair

- Goal: transform syntactically incorrect program into the *closest* correct program, mainly to be able to continue parsing.
  Try to maximize the number of *useful* error messages per compilation.
- Measure of closeness: smallest number of changes. For example:
  insert 1 token or delete 1 token or change 1 token.
- Problems
  - closest correct program may not be unique e.g. delete **begin** or insert **end** ?
  - Algorithms to find a closest correct program are non-linear.
  - Any measure of closeness may not give the intuitively closest program in all cases.
    Examples

PL/I	DOWHILE(I = 1);	→	DOWHILE(I) = 1 ;
Turing	gut S	→	**get** S
		→	**put** S

Syntax Error Repair Strategy

- General strategy: isolate the error in a *replaceable phrase*. Replaceable implies that the parse can continue.
- Making the *closest correction* may require changing the parser stack.
  Example:
  A = B **then** I = 1 **else** I = 0
  The closest correction would be to insert an **if** and compile A = B as a boolean expression. But it may have already been compiled as an assignment statement before we know about the error.
- Three (of many) possible strategies are the *recovery token strategy*, the *panic strategy* and *spelling correction*.

Syntax Error Repair Strategy

- Recovery token repair strategy
  - For each row in the table, one input token is designated as the *recovery token*.
  - Reserved words, identifiers, numbers and strings are *long* tokens.
    All other tokens are *short* tokens
  - if the incorrect token is long and the recovery token is short
    insert the recovery token in front of the input.
  - Otherwise, replace the current input symbol with the recovery token.
- Panic repair strategy
  - Some tokens are designated as *hard* tokens. For example **;** , **end**
    All other tokens are *soft*
  - Discard input up to and including the first hard token.
  - Pop the parser stack down to a corresponding token or state.
- Spelling correction strategy:
  - Replace identifier that is close to a reserved word with the reserved word.

LL(1) vs. LR(1)

- Almost all compilers use LL(1) or LALR(1) (or some close relative like recursive descent or SLR(1)).
- For grammars: LL(k) $\subset$ LL(K+1) LR(k) $\subset$ LR(k+1) LL(k) $\subset$ LR(k)
  For languages: LL(k) $\subset$ LL(k+1) $\subset$ LR(1) $=$ LR(k)
- LR(1) is much more powerful than LL(1), but LL(1) may be strong enough.
- If you have no parser generator, try to make the grammar LL(1) and then build an LL(1) parser or use recusive descent.
- If you have a parser generator use it.
- if you don't have a parser generator *do not* try and build an LALR(1) parser from scratch.
- If you have both LL(1) and LALR(1) parser generators use LALR(1)

food for slide eater

Short Review

- Lexical analysis: regular expressions, finite automata, determinism/non-determinism
  - Chapter 3. Problems 3, 4, 6, 16, 17, 20, 21
  - Handout.
- Context-free grammars, Chomsky hierarchy
  - Chapter 4. Problems 4, 7, 12
- Parsing: top-down, bottom-up, operator precedence (basic understanding), LL parsing - trace from table, LR parsing - trace from table
  Example on Fig. 5.10
  Chapter 6. Problems 1, 3, 4, 7, 8, 13, 14, 15, 19, 35, 36, 37
- Parsing: construction of LR(0) and LR(1) tables. Differences between the tables

food for slide eater