

LALR(1) parsing (Lison & others)

Recall: LR(1) parser may be huge

LALR(1) is a compromise:

- Reduced info included in states, resolving some conflicts with SLR
- Same number of states as SLR (mergeable)

Definition: "3 state fine" method for constructing an LALR(1) parser:

1. Construct a (canonical) LR(1) parser.
2. Merge states with the same "core" into one state.

LALR(1) item:

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

and core

If 2 sets of items look the same when ignoring the lookahead, then merge them.

Ex:

$$S \rightarrow S \quad I_1: S' \rightarrow S, \$$$

$$S \rightarrow XX \quad S \rightarrow X, \$$$

$$X \rightarrow aX \quad X \rightarrow aX, a$$

$$X \rightarrow b \quad X \rightarrow aX, a/b$$

$$X \rightarrow b, a/b$$

$I_1: S' \rightarrow S, \$$

$I_2: S \rightarrow X, \$$

$X \rightarrow aX, \$$

$X \rightarrow b, \$$

Apr 20-11:00 AM

$I_3: X \rightarrow a \cdot X, a/b$

$X \rightarrow aX, a/b$

$X \rightarrow b, a/b$

$I_4: X \rightarrow b, a/b$

$I_5 = \text{trans}(I_3, X):$

$$S \rightarrow XX, \$$$

$I_6 = \text{trans}(I_4, a):$

$$X \rightarrow aX, \$$$

$$X \rightarrow aX, \$$$

$$X \rightarrow b, \$$$

$I_7 = \text{trans}(I_6, b):$

$$X \rightarrow b, \$$$

Merge  $I_3$  &  $I_6$  into one state

$I_{36}: X \rightarrow aX, \$/a/b$

$$X \rightarrow aX, \$/a/b$$

$$X \rightarrow b, \$/a/b$$

Merge  $I_4$  &  $I_7$  into

$I_{47}: X \rightarrow b, a/b$

More efficient method: merge states (if possible) as they are constructed

[check core items against those of previous states]

Apr 20-11:18 AM

Notes:

1. Merging maintains integrity of the trans table. Still well-defined.
2. Building action table is same in LALR(1) as in LR(1)
3. Merging states will not produce new shifts/reduce conflicts.
4. May produce new reduce/reduce conflicts.
5. Defn: A grammar G is LALR(1) if the LALR(1) parser produced in conflict-free.

Final list for Proj III

Intermediate actions on core conflicts.

$$A \rightarrow X_1 X_2 X_3$$

$$A \rightarrow X_1 X_2 X_3$$

$$A \rightarrow X_1 X_2 X_3$$

$$A \rightarrow X_1 X_2 X_3$$

act1: { }

act2: { }

Solution: introduce yourself a new non-terminal too.

$$A \rightarrow X_1 X_2 X_3 \mid \text{foo} \{ \}$$

$$A \rightarrow X_1 X_2 X_3 \mid \text{foo} \{ \}$$

Apr 20-11:28 AM

Code generation/optimization

Choose an abstract target language — simple, unstructured, low-level, machine indep.

Examples include: triplex quadruples

→ 3-address code

Sequence of instructions

Instructions include

$x := y$  or  $x := y$ , where  $x$  is a variable,  $y, z$  are either vars or constants and  $op$  is some binary operator:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$

$x, y, z$  are all of simple numerical type.

$x := y$

$op$  is unary  $+$ ,  $-$

$x := y$

$a[y] := z$

$x := a[y]$

Instructions may be preceded by one or more labels

if  $x op y$  goto L

goto L

Here,  $op$  is one of the 6 numerical comparison operators, L is label.

Apr 20-11:45 AM

Compiling high-level language (C or Pascal) into 3-address code

Syntax-directed translation (you know this, essentially)

Instead of a control stack, most compilers use a stack of "virtual registers" for temporary values during expression evaluation.

$t1, t2, t3, \dots$

$z = (x + y) * (z - 3)$

$t1 := x + y$

$t2 := z - 3$

$z := t1 * t2$

Code optimization — necessary because optimal means best.

But only guarantee best, not efficient code by this

frontal stack

back

3-address rule

3-address rule

Briefly: front-end optimizations:

1. constant folding
2. shared sub-expressions
3. algebraic transformations

Apr 20-11:53 AM

2.  $z = (x - y) * (x - y)$

expr dag

instead of

3. Algebraic transformations:

$z = 0 * y \Rightarrow z = 0$

$z = (x + y) + z$

$t = x + z + 2 * y$

$z = (x + 3) - (y + 4)$

$t = x - y + (3 - 4)$

$z = x - y - 1$

Optimization of 3-address code

Peephole

Control flow analysis

Liveness analysis

Peephole

$x := y$

looks for local improvements within a sliding window. Multiple passes until nothing changes

General comment: optimization must preserve the semantics.

Apr 20-12:05 PM