LR(k) parsing (pg. 1)

I. Intro
II. Table construction LR(∅) SLR(1)
III LR parsing
IV LR(1), LALR(1)

I. Intro

L ⇒ left-to-right
R ⇒ right most derivation
k ⇒ # look ahead usually k=0
LR(0) - SLR, LALR   easy to construct
LR(1) -   more powerful, less practical

Advantages:
- powerful
- most general non-back tracking method known
- detect errors at earliest posible point
- class of grammars superset of grammars handled by predictive parsers

Drawbacks:
- hard to hand code
- hard to debug
- error recovery problematic

II Table Construction

LR(k) automaton:
- 2 basic moves: shift, reduce
- 2 other moves: accept, error
- uses a stack to track parsing progress

Concepts:
- an LR(0) item is a rule with a dot at some position in the RHS., e.g.:
  A → .BCD, A → B.CD, A → BC.D, A → BCD.
  A → . (empty / ε rule)
- An LR(0) state is a set of items
- notation: lowercase letters terminals
  : uppercase letters non-terminals
  : greek letters strings

Canonical LR(∅) collection: basis of LR(∅) deterministic finite automaton used to make parsing decisions

for grammar G, need:
(A) augmented grammar G'
(B) closure function
(C) GOTO function

(A) for augmented grammar just add rule S' → S

(B) given a set of items I
closure(I):
1. add I to closure of I
2. if $A \to \alpha . B\beta \in$ closure(I)
   then add $B \to . \gamma$ to closure(I)

keep applying until nothing can be added

Example:
1) E' → E
2) E → T
3) T → T * F
4) T → F
5) F → (E)
6) F → id

Let I = {[E' → .E]}

I_0 = closure(I) = { [E' → .E],   } kernel set
[E → .E + T],
[E → .T]
[T → .T * F],
[T → .F]
[F → .(E)],
[F → .id] }

kernel items form [E' → .E] or [A → α.xβ] where α ≠ ε
non-kernel items form [A → .α]

(C) Given a set of items I
∀x in set of symbols following dot
GOTO(I, x) =
closure({[A → α x . β] | [A → α . x β] ∈ I})

I_1 = GOTO(I_0, E) = {[E' → E.],
[E' → E. + T]}

I_2 = GOTO(I_0, T) = {[E → T.],
[T → T. * F]}

I_3 = GOTO(I_0, F) = {[T → F.]}

I_4 = GOTO(I_0, ( ) = {[F → (.E)],
[E → .E + T],
[E → .T],
[T → .T * F],
[T → .F],
[F → .(E)],
[F → .id]}

I_5 = GOTO(I_0, id) = {[F → id.]}

LR(k) parsing (pg. 2)

Algorithm for constructing canonical LR(∅) set of items:

initialize $C = \{ \text{closure}(\{ [S' \to .S] \}) \}$

repeat for all $I \in C$, for all x,

if $\text{GOTO}(I, x) \neq \emptyset$ and $\text{GOTO}(I,x) \notin C$

add $\text{GOTO}(I,x)$ to C

In practice, find repeating items, e.g. $\text{GOTO}(I_4, T)$ happens to be in I_2

A final item is an item with a dot at the end

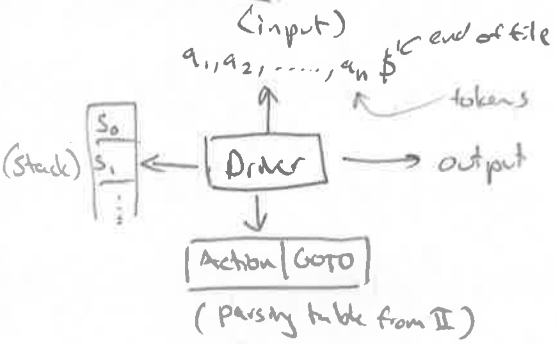
If all final items are in states by themselves, then the collection is LR(∅), otherwise there is a shift-reduce conflict in LR(∅) or a reduce-reduce conflict in LR(∅)

Doing that yields an SLR(1) table

Even in SLR(1), can still have conflicts, since follow sets may overlap

See hand-outs for graphical and table representations of the running example

III LR parsing



Stack → summary of a path in the automaton.

LR parser configuration

S_0	S_1	...	S_n

a_i	a_{i+1}	...	a_n	$

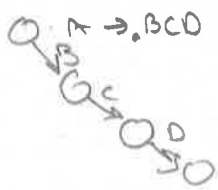
If next move is:

a) shift j: move past symbol a_i in input
push s_j into stack $s_m \xrightarrow{a} s_j$

b) reduce $A \to \beta$: pop $|\beta|$ states from stack
push $\text{GOTO}(S_{m-|\beta|}, A)$ on stack

c) accept: Done!

d) error: I Quit!



Example (see table from handout)

s_0	a + b * c $	s5
$s_0\ s_5$	+ b * c $	r6
$s_0\ s_3$	+ b * c $	r4
$s_0\ s_2$	+ b * c $	r2
$s_0\ s_1$	+ b * c $	s6
$s_0\ s_1\ s_6$	b * c $	s5
$s_0\ s_1\ s_6, s_5$	* c $	r6
$s_0\ s_1\ s_6, s_3$	* c $	r4
$s_0\ s_1\ s_6\ s_9$	* c $	s7
$s_0\ s_1\ s_6\ s_9\ s_7$	c $	s5
$s_0\ s_1\ s_6\ s_9\ s_7\ s_5$	$	r6
$s_0\ s_1\ s_6\ s_9\ s_7\ s_{10}$	$	r3
$s_0\ s_1\ s_6\ s_9$	$	r1
$s_0\ s_1$	$	acc

Can use translation scheme propagating synthesized attributes during reduce actions to build AST

The LR driver:

push s_0 onto empty stack
read first input token
repeat:
  let s = the state at the top of stack
  if action(s, a) = shift t:
    push t onto the stack
    let a be next symbol
  else if action(s, a) = reduce $A \to \beta$:
    pop $|\beta|$ states off stack
    let t be new top of stack
    push GOTO(t, A) on stack
    synthesize attributes for $A \to \beta$
  else if action(s, a) == accept
    break
  else
    call error recovery

The same LR driver works for all the variants of LR parse tables

LR(k) parsing (pg. 3)

IV LR(1), LALR(1)

An LR(1) item is a pair consisting of
an LR(0) item and a look-ahead,
e.g., $[A \to \alpha . \beta \;, a]$

Starting point: $[S' \to .S, \$]$

Example: $[E' \to .E, \$]$ \$

closure: $[E \to .E + T, \; \$, +$

$[E \to .T \quad \$, +$

$T \to .T * F, \; \$, +, *$

$T \to .F \quad , \$, +, *$

$F \to .(E), \quad \$, +, *$

$F \to .id. \quad \$, +, *$

General rule for an LR(1) canonical collection:

closure(I):

repeat for all $[A \to \alpha . B\beta, a]$ in I,

for all $[B \to \gamma$ in G',

for all b in first (βa)

add $[B \to .\gamma, b]$ to set I

until no more items can be added

goto(I, x):

$J = \emptyset$

for all $[A \to \alpha . x\beta, a] \in I$

add $[A \to \alpha x . \beta, a]$ to J

return closure(J)

Unlike with LR(Ø) → SLR step, no need
to deal with ambiguous states in LR(1)

For LALR, systematically merge certain
LR(1) steps, to get more compact table

to build LALR, given LR(Ø) state, consider
path that led to it: more efficient
parser table construction.