Parsing Wrapup

Roadmap (Where are we?)

Last lecture

- Shift-reduce parser
- LR(1) parsing
  - → LR(1) items
  - → Computing closure
  - → Computing goto
  - → LR(1) canonical collection

This lecture

- LR(1) parsing
  - → Building ACTION / GOTO tables
  - → Shift/reduce and reduce/reduce conflicts
  - → SLR(1), LALR(1), operator precedence
  - → Error recovery

Filling in the LR(1) ACTION and GOTO Tables

The algorithm

x is the number of the state for s_x

$\forall$ *set* $s_x \in S$
  $\forall$ *item* $i \in s_x$
    if i *is* $[A \to \beta \cdot \underline{a}\gamma, \underline{b}]$ *and* $goto(s_x, \underline{a}) = s_k$, $\underline{a} \in T$ — // • to left of terminal a
      then ACTION$[x, \underline{a}] \leftarrow$ "*shift k*" — // → shift if lookahead = a
    else if i *is* $[S' \to S \cdot, \$]$ — // start production done,
      then ACTION$[x, \$] \leftarrow$ "*accept*" — // → accept if lookahead = $
    else if i *is* $[A \to \beta \cdot, \underline{a}]$ — // • all the way to right
      then ACTION$[x, \underline{a}] \leftarrow$ "*reduce* $A \to \beta$" — // → production done
  $\forall$ $n \in NT$ — // reduce if lookahead = a
    if $goto(s_x, n) = s_k$
      then GOTO$[x, n] \leftarrow k$ — // store transitions for nonterminals

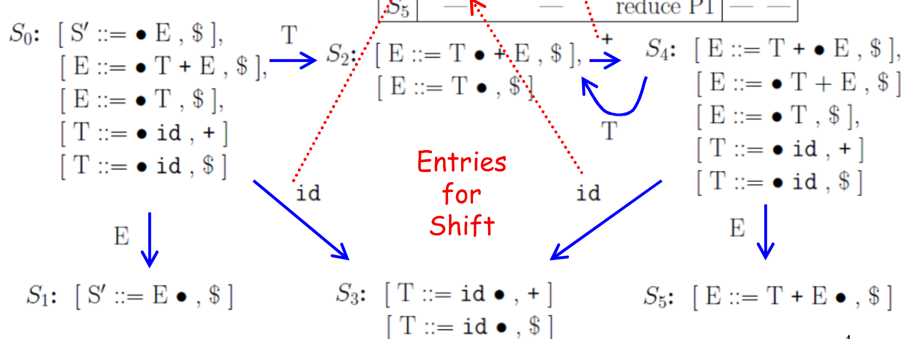
Many items generate no table entry

e.g., $[A \to \beta \cdot B\alpha, \underline{a}]$ does not, but *closure* ensures that all the rhs' for *B* are in s_x

Example – Building LR(1) ACTION and GOTO Table

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

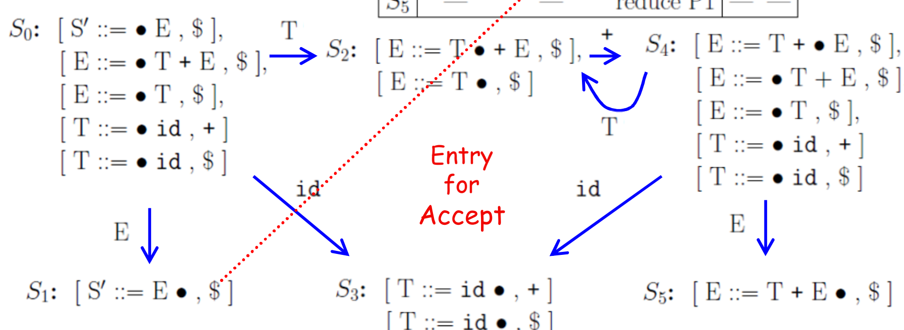
	ACTION			GOTO	
	id	+	$	E	T
S_0	shift 3	—	—	1	2
S_1	—	—	accept	—	—
S_2	—	shift 4	reduce P2	—	—
S_3	—	reduce P3	reduce P3	—	—
S_4	shift 3	—	—	5	2
S_5	—	—	reduce P1	—	—



Example – Building LR(1) ACTION and GOTO Table

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

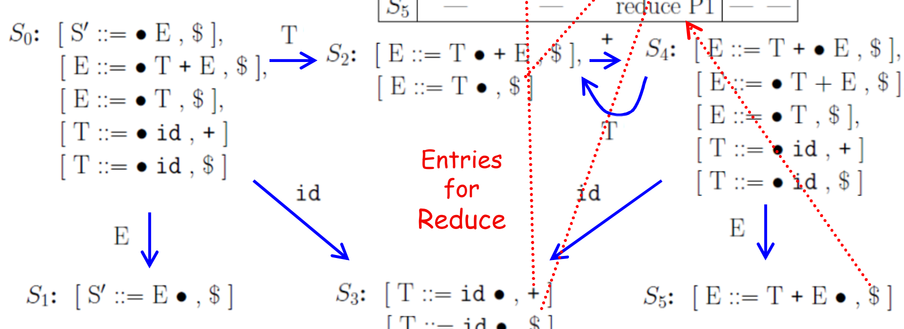
	ACTION			GOTO	
	id	+	$	E	T
S_0	shift 3	—	—	1	2
S_1	—	—	accept	—	—
S_2	—	shift 4	reduce P2	—	—
S_3	—	reduce P3	reduce P3	—	—
S_4	shift 3	—	—	5	2
S_5	—	—	reduce P1	—	—



Example – Building LR(1) ACTION and GOTO Table

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

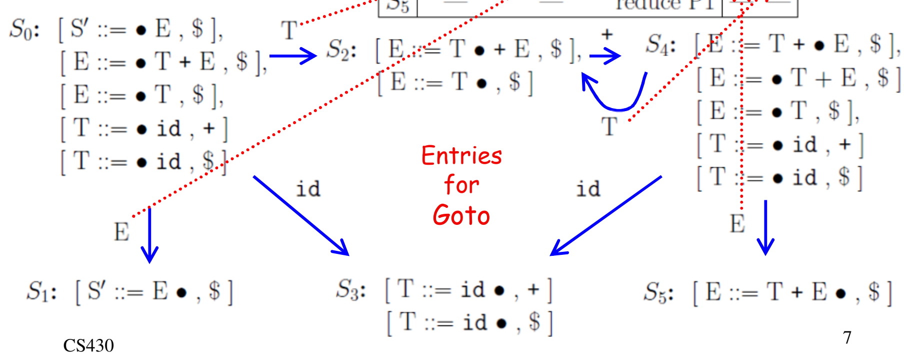
	ACTION			GOTO	
	id	+	$	E	T
S_0	shift 3	—	—	1	2
S_1	—	—	accept	—	—
S_2	—	shift 4	reduce P2	—	—
S_3	—	reduce P3	reduce P3	—	—
S_4	shift 3	—	—	5	2
S_5	—	—	reduce P1	—	—



Example – Building LR(1) ACTION and GOTO Table

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	—	—



What can go wrong?

What if set *s* contains [*A*→β•aγ,b] and [*B*→β•,a] ?

- First item generates "shift", second generates "reduce"
- Both define ACTION[s,a] — cannot do both actions
- This is a fundamental ambiguity, called a *shift/reduce conflict*

What if set *s* contains [*A*→γ•, a] and [*B*→γ•, a] ?

- Each generates "reduce", but with a different production
- Both define ACTION[s,a] — cannot do both reductions
- This fundamental ambiguity is called a *reduce/reduce conflict*

In either case, the grammar is not LR(1)

Solutions

- Modify grammar to eliminate conflict
- Specify how conflict should be resolved
  - → Can be used to assign precedence & associativity (to operators)

Shift/reduce Error Example

- Grammar

P1	S	$\rightarrow$	E
P2	E	$\rightarrow$	E + E
P3		|	a

- State in canonical collection of LR(1) items

$[\ E \rightarrow E + E \bullet\ ,\ \{\$,+\}\ ]$
$[\ E \rightarrow E \bullet + E\ ,\ \{\$,+\}\ ]$

- Entry in ACTION / GOTO table for lookahead +

shift 3
reduce P2
($E \rightarrow E + E$)

Reduce/reduce Error Example

- Grammar

P1 $S \rightarrow E$
P2 $E \rightarrow A$
P3 $\quad | \ a$
P4 $A \rightarrow a$

- State in canonical collection of LR(1) items

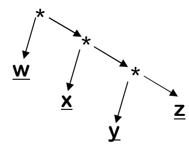
[E → a •, $]
[A → a •, $]

- Entry in ACTION / GOTO table for lookahead $

reduce P3 (E → a)
reduce P4 (A → a)

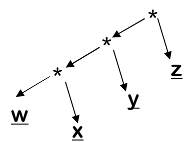
Left Recursion versus Right Recursion

- Right recursion
  - → Required for termination in top-down parsers
  - → Produces right-associative operators



w * (x * (y * z))

- Left recursion
  - → Works fine in bottom-up parsers
  - → Limits required stack space
  - → Produces left-associative operators

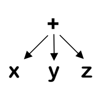


((w * x) * y) * z

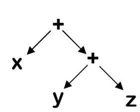
- Rule of thumb
  - → Left recursion for bottom-up parsers
  - → Right recursion for top-down parsers

Associativity

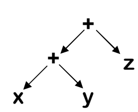
- Normally defined by programming language
- What difference does it make?
- Can change answers in floating-point arithmetic
- Exposes a different set of common subexpressions
- Consider x+y+z



Ideal operator



Right association



Left association

- What if y+z occurs elsewhere? Or x+y? or x+z?
- What if x = 2 & z = 17 ? Neither left nor right exposes 19
  - → I.e., optimizer cannot replace x+z with 19 in output code

Resolving Conflicts

- Precedence and associativity may be used to resolve conflicts in ambiguous grammars
- When comparing operator on stack with lookahead
  - → Shift if lookahead has
    - Higher precedence
    - Same precedence, right associative
  - → Reduce if lookahead has
    - Lower precedence
    - Same precedence, left associative
- Advantage
  - → Can use smaller (ambiguous) grammars
  - → Example
    - E → E + E | E - E | E * E | E / E | - E | id | num

Operator Precedence Grammars

- Alternative approach to shift-reduce parsing
- Given a sentential form aABb, three possibilities
  1. A in handle, B not in handle — A > B
     - → A is reduced before B
  2. A and B both in handle — A = B
     - → A and B reduced at same time
  3. B in handle, A not in handle — B > A
     - → B is reduced before A
- Handle is composed of
  - → < >, < = >, < = = >, < = = = >, etc…
- 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 closest < to top of stack)

Operator Precedence Grammar 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	$	$ > $

Shrinking the ACTION/GOTO Tables

Three options:

- Combine terminals such as number & identifier, + & -, * & /
  - → Directly removes a column, may remove a row
  - → For expression grammar, 198 (vs. 384) table entries
- Combine rows or columns (*table compression*)
  - → Implement identical rows once & remap states
  - → Requires extra indirection on each lookup
  - → Use separate mapping for ACTION & for GOTO
- Use another construction algorithm
  - → Both SLR(1) and LALR(1) produce smaller tables
  - → Implementations are readily available

SLR(1) Parser

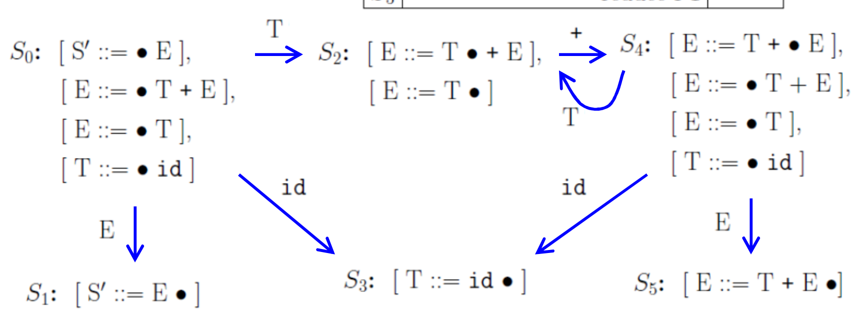
- Build ACTION / GOTO table using LR(0) items
  - → Problem – when to perform reduction for $A \to \beta$?
  - → Solution – reduce $A \to \beta$ only when lookahead ∈ FOLLOW(A)
- Algorithm

∀ *set* $s_x \in S$
  ∀ *item* $i \in s_x$
    if *i is* [$A \to \beta \bullet \underline{a} \gamma$] *and goto*($s_x$,$\underline{a}$) = s_k, $\underline{a} \in T$ // • to left of terminal a
      then ACTION[*x*,$\underline{a}$] ← "*shift k*" // → shift if lookahead = a
    else if *i is* [$S' \to S \bullet$] // start production done,
      then ACTION[*x* , $] ← "*accept*" // → accept if lookahead = $
    else if *i is* [$A \to \beta \bullet$] and a ∈ FOLLOW(A) // • all the way to right
      then ACTION[*x*,$\underline{a}$] ← "*reduce* $A \to \beta$" // → reduce if lookahead
  ∀ $n \in NT$ // is in FOLLOW(A)
    if *goto*(s_x ,*n*) = s_k
      then GOTO[*x*,*n*] ← *k* // store transitions for nonterminals

Example – SLR(1) Parser

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	—	—



LALR(1) Parser

- Core of set of LR(1) items
  - → Set of LR(0) items derived by ignoring lookahead symbols
  - → Example

LR(1) state		LR(0) state
[E → a •, b]	→	[E → a •]
[A → a •, c]		[A → a •]

- LALR(1) parser
  - → Merge two sets of LR(1) items (states), if same core
- Result
  - → Potentially much smaller set of states
    - Same as SLR(1) parser
  - → May introduce reduce/reduce conflicts
  - → Will not introduce shift/reduce conflicts

Example – LALR(1) Parser

- Merging states

[E → a •, b] [A → ba •, c]	**can be merged with**	[E → a •, d] [A → ba •, b]

- Resulting state

  [E → a •, b]
  [A → ba •, b]
  [E → a •, d]
  [A → ba •, c]

- Introduces reduce/reduce error
  - → Can reduce either **E → a or A → ba** for lookahead = b

LR(k) versus LL(k) (*Top-down Recursive Descent*)

Finding Reductions

LR(*k*) ⇒ Each reduction in the parse is detectable with
- → Complete left context (everything to left of handle)
- → Reducible phrase (handle) itself
- → *k* terminal to right of handle

LL(*k*) ⇒ Parser must select the next rule based on
- → Complete left context (everything up to & including NT to expand)
- → *k* terminals to right of nonterminal to expand

Thus, LR(*k*) is more powerful since it examines more context

For example, consider grammar **S → ab | aa**
- → LL(1)?
  - No, cannot decide which production with lookahead = a
- → LR(1)?
  - Yes, can shift ab or aa onto stack before deciding on reduction

LR(k) Parsers

- Properties
  - → Strictly more powerful than LL(k) parsers
  - → Most general non-backtracking shift-reduce parser
  - → Detects error as soon as possible in left-to-right scan of input
    - Contents of stack are viable prefixes
      - Possible for remaining input to lead to successful parse

LR(k) versus LL(k) Summary

	Advantages	*Disadvantages*
Top-down recursive descent	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

LR(0) versus SLR(1) versus LR(1)

Example grammar

$S' \to S$
$S \to S\ ;\ a\ \mid\ a$

LR(0) ?

LR(1) ?

SLR(1) ?

LALR(1) versus LR(1)

Example grammar

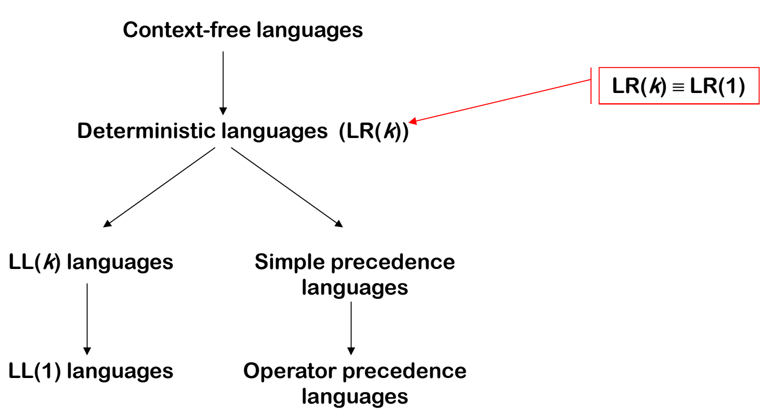
$S' \to S$
$S \to aAd \mid bBd \mid aBe \mid bAe$
$A \to c$
$B \to c$

LR(0) ?

LR(1) ?

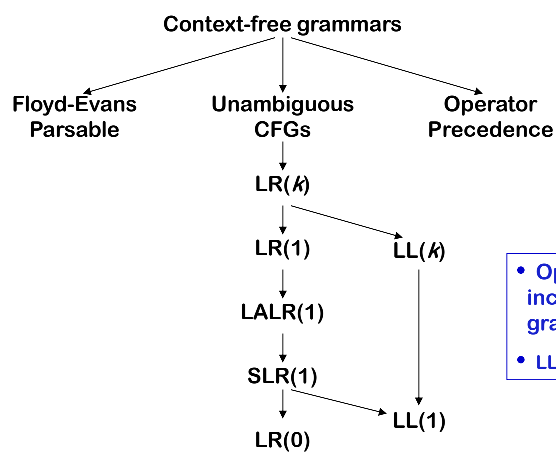
LALR(1) ?

Hierarchy of Context-Free Languages



The inclusion hierarchy for context-free languages

Hierarchy of Context-Free Grammars



- Operator precedence includes some ambiguous grammars
- LL(1) is a subset of SLR(1)

The inclusion hierarchy for context-free grammars

Error Recovery in Shift-Reduce Parsers

The problem: parser encounters an invalid token

Goal: Want to parse the rest of the file

Basic idea (panic mode):

- → Assume something went wrong while trying to find handle for nonterminal *A*
- → Pretend handle for *A* has been found; pop "handle", skip over input to find terminal that can follow *A*

Restarting the parser (panic mode):

- → find a restartable state on the stack (has transition for nonterminal *A*)
- → move to a consistent place in the input (token that can follow *A*)
- → perform (error) reduction (for nonterminal *A*)
- → print an informative message

Error Recovery in YACC (ASU p.264)

Yacc's (bison's) error mechanism (note: version dependent!)

- designated token **error**
- used in error productions of the form
  $A \rightarrow \beta$ **error** α
- α specifies synchronization points

When error is discovered

- pops stack until it finds state where it can shift the **error** token
- resumes parsing to match α
  special cases:
  - → α = w, where w is string of terminals: skip input until w has been read
  - → $\alpha = \varepsilon$: skip input until state transition on input token is defined
- error productions can have actions

Error Recovery in YACC

```
cmpdstmt: BEG stmt_list END
stmt_list : stmt
          | stmt_list ';' stmt
          | error { yyerror("\n***Error: illegal statement\n");}
```

This should

- throw out the erroneous statement
- synchronize at ";" or "end" (implicit: $\alpha = \varepsilon$)
- writes message "***Error: illegal statement" to stderr

Example: begin a & 5 | hello ; a := 3 end

↑ ↑ resume parsing

***Error: illegal statement