LR Parsing
LALR Parser Generators

Outline

- Review of bottom-up parsing
- Computing the parsing DFA
- Using parser generators

Bottom-up Parsing (Review)

- A bottom-up parser rewrites the input string to the start symbol
- The state of the parser is described as

$$\alpha \mid \gamma$$

  - α is a stack of terminals and non-terminals
  - γ is the string of terminals not yet examined

- Initially: $\mid x_1 x_2 \ldots x_n$

The Shift and Reduce Actions (Review)

Recall the CFG: $E \rightarrow E + (E) \mid \text{int}$

A bottom-up parser uses two kinds of actions:

- Shift pushes a terminal from input on the stack

$$E + (\mid \text{int}) \Rightarrow E + (\text{int} \mid)$$

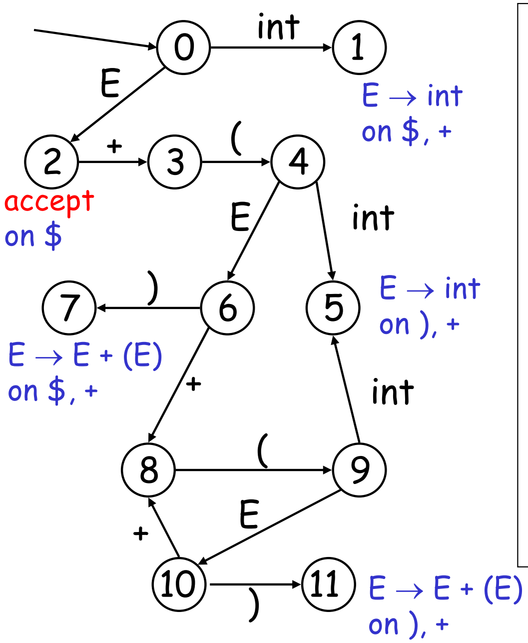
- Reduce pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS)

$$E + (\underline{E + (E)} \mid) \Rightarrow E + (\underline{E} \mid)$$

Key Issue: When to Shift or Reduce?

- Idea: use a deterministic finite automaton (DFA) to decide when to shift or reduce
  - The input is the stack
  - The language consists of terminals and non-terminals
- We run the DFA on the stack and we examine the resulting state X and the token tok after ∣
  - If X has a transition labeled tok then <u>shift</u>
  - If X is labeled with "$A \rightarrow \beta$ on tok" then <u>reduce</u>

LR(1) Parsing: An Example



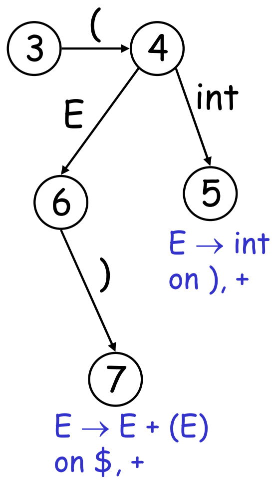
∣ int + (int) + (int)$	shift
int ∣ + (int) + (int)$	E → int
E ∣ + (int) + (int)$	shift (x3)
E + (int ∣) + (int)$	E → int
E + (E ∣) + (int)$	shift
E + (E) ∣ + (int)$	E → E+(E)
E ∣ + (int)$	shift (x3)
E + (int ∣)$	E → int
E + (E ∣)$	shift
E + (E) ∣ $	E → E+(E)
E ∣ $	accept

Representing the DFA

- Parsers represent the DFA as a 2D table
  (Recall table-driven lexical analysis)
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: the action table
  - Those for non-terminals: the goto table

Representing the DFA: Example

The table for a fragment of our DFA:



	int	+	(	)	$	E
…						
3			s4			
4	s5					g6
5		$r_{E \rightarrow int}$		$r_{E \rightarrow int}$		
6	s8		s7			
7		$r_{E \rightarrow E+(E)}$			$r_{E \rightarrow E+(E)}$	
…						

sk is shift and goto state k
$r_{X \rightarrow \alpha}$ is reduce
gk is goto state k

The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated
- To avoid this, we remember for each stack element on which state it brings the DFA
- LR parser maintains a stack

$$\langle sym_1, state_1 \rangle \ldots \langle sym_n, state_n \rangle$$

$state_k$ is the final state of the DFA on $sym_1 \ldots sym_k$

The LR Parsing Algorithm

```
let I = w$ be initial input
let j = 0
let DFA state 0 be the start state
let stack = ⟨ dummy, 0 ⟩
  repeat
    case action[top_state(stack), I[j]] of
      shift k: push ⟨ I[j++], k ⟩
      reduce X → A:
        pop |A| pairs,
        push ⟨ X, goto[top_state(stack), X] ⟩
      accept: halt normally
      error: halt and report error
```


Key Issue: How is the DFA Constructed?

- The stack describes the context of the parse
  - What non-terminal we are looking for
  - What production RHS we are looking for
  - What we have seen so far from the RHS
- Each DFA state describes several such contexts
  - E.g., when we are looking for non-terminal E, we might be looking either for an int or an E + (E) RHS

LR(0) Items

- An <u>LR(0) item</u> is a production with a "I" somewhere on the RHS
- The LR(0) items for T → (E) are

  T → I (E)

  T → (I E)

  T → (E I)

  T → (E) I
- The only LR(0) item for $X \to \varepsilon$ is X → I

LR(0) Items: Intuition

- An item $[X \to \alpha \mid \beta]$ says that the parser
  - is looking for an X
  - has an α on top of the stack
  - expects to find a string derived from β next in the input
- Notes:
  - $[X \to \alpha \mid a\beta]$ means that a should follow
    - Then we can shift it and still have a viable prefix
  - $[X \to \alpha \mid]$ means that we could reduce X
    - But this is not always a good idea !

LR(1) Items

- An LR(1) item is a pair:

$$X \to \alpha \mid \beta, \; a$$

  - $X \to \alpha\beta$ is a production
  - a is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal
- $[X \to \alpha \mid \beta, a]$ describes a context of the parser
  - We are trying to find an X followed by an a, and
  - We have (at least) α already on top of the stack
  - Thus we need to see next a prefix derived from βa

Note

- The symbol $\mid$ was used before to separate the stack from the rest of input
  - $\alpha \mid \gamma$, where α is the stack and γ is the remaining string of terminals
- In items, $\mid$ is used to mark a prefix of a production RHS:

$$X \to \alpha \mid \beta, \quad a$$

  - Here β might contain non-terminals as well
- In either case the stack is on the left of $\mid$

Convention

- We add to our grammar a fresh new start symbol S and a production $S \to E$
  - Where E is the old start symbol
- The initial parsing context contains:

$$S \to \mid E \;, \$$$

  - Trying to find an S as a string derived from E$
  - The stack is empty

LR(1) Items (Cont.)

- In context containing

  $E \to E + \mid (E)\ , +$

  - If (follows then we can perform a shift to context containing

    $E \to E + (\mid E)\ , +$

- In context containing

  $E \to E + (E) \mid\ , +$

  - We can perform a reduction with $E \to E + (E)$
  - But only if a + follows

LR(1) Items (Cont.)

- Consider the item

  $E \to E + (\mid E)\ , +$

- We expect a string derived from E) +
- Our example has two productions for E

  $E \to int$ and $E \to E + (E)$

- We describe this by extending the context with two more items:

  $E \to \mid int\ ,)$

  $E \to \mid E + (E)\ ,)$

The Closure Operation

- The operation of extending the context with items is called the closure operation

```
Closure(Items) =
  repeat
    for each [X → α ı Yβ, a] in Items
      for each production Y → γ
        for each b in First(βa)
          add [Y → ı γ, b] to Items
  until Items is unchanged
```


Constructing the Parsing DFA (1)

$E \to E + (E) \mid int$

- Construct the start context:

  Closure({$S \to \mid E, \$$})

  $S \to \mid E\ , \$$
  $E \to \mid E+(E), \$$
  $E \to \mid int\ , \$$
  $E \to \mid E+(E), +$
  $E \to \mid int\ , +$

- We abbreviate as:

  $S \to \mid E\ , \$$
  $E \to \mid E+(E)\ , \$/+$
  $E \to \mid int\ , \$/+$

Constructing the Parsing DFA (2)

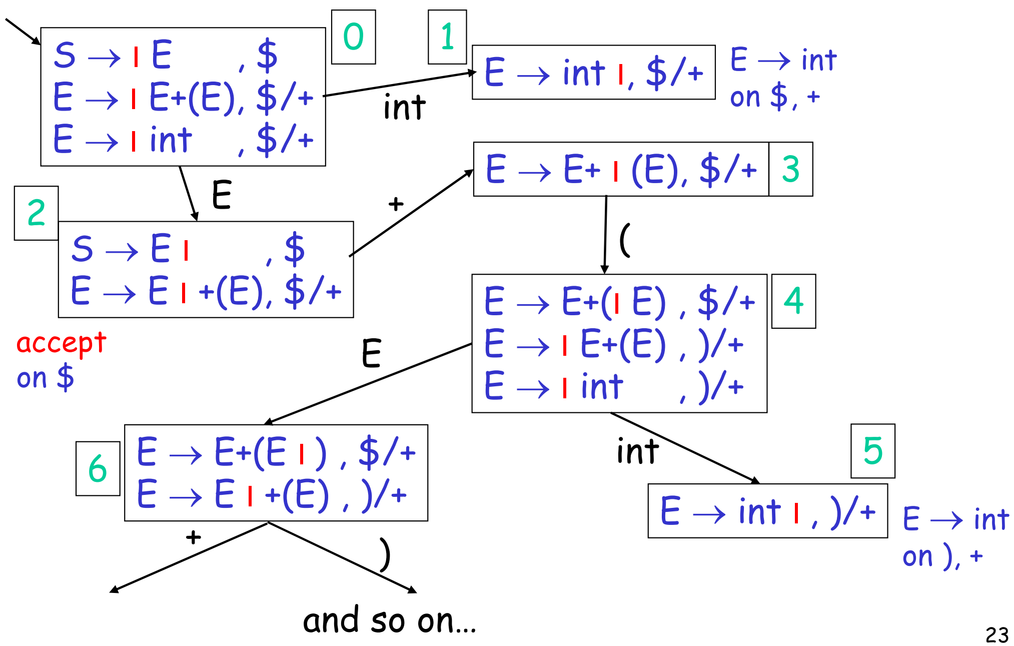
- A DFA state is a closed set of LR(1) items
- The start state contains $[S \to \bullet E \ , \$]$
- A state that contains $[X \to \alpha \bullet, b]$ is labeled with "reduce with $X \to \alpha$ on b"
- And now the transitions …

The DFA Transitions

- A state "State" that contains $[X \to \alpha \bullet y\beta, b]$ has a transition labeled y to a state that contains the items "**Transition**(State, y)"
  - y can be a terminal or a non-terminal

```
Transition(State, y)
  Items = ∅
  for each [X → α • yβ, b] in State
      add [X → αy • β, b] to Items
  return Closure(Items)
```


Constructing the Parsing DFA: Example



LR Parsing Tables: Notes

- Parsing tables (i.e., the DFA) can be constructed automatically for a CFG
- But we still need to understand the construction to work with parser generators
  - E.g., they report errors in terms of sets of items
- What kind of errors can we expect?

Shift/Reduce Conflicts

- If a DFA state contains both
  $[X \to \alpha \mid a\beta,\ b]$ and $[Y \to \gamma \mid,\ a]$
- Then on input "a" we could either
  - Shift into state $[X \to \alpha a \mid \beta,\ b]$, or
  - Reduce with $Y \to \gamma$
- This is called a *shift-reduce conflict*

Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar
- Classic example: the dangling else
  S → if E then S | if E then S else S | OTHER
- Will have DFA state containing
  [S → if E then S |,   else]
  [S → if E then S | else S,   x]
- If else follows then we can shift or reduce
- Default (yacc, ML-yacc, bison, etc.) is to shift
  - Default behavior is as needed in this case

More Shift/Reduce Conflicts

- Consider the ambiguous grammar
  E → E + E | E * E | int
- We will have the states containing
  [E → E * | E, +]   [E → E * E |, +]
  [E → | E + E, +] $\Rightarrow^{E}$ [E → E | + E, +]
  …   …
- Again we have a shift/reduce on input +
  - We need to reduce (* binds more tightly than +)
  - Recall solution: declare the precedence of * and +

More Shift/Reduce Conflicts

- In yacc declare precedence and associativity:

```
%left +
%left *
```

- Precedence of a rule = that of its last terminal
  See yacc manual for ways to override this default
- Resolve shift/reduce conflict with a <u>shift</u> if:
  - no precedence declared for either rule or terminal
  - input terminal has higher precedence than the rule
  - the precedences are the same and right associative

Using Precedence to Solve S/R Conflicts

- Back to our example:

  [E → E * | E, +]   [E → E * E |, +]

  [E → | E + E, +] ⇒E [E → E | + E, +]

  …   …

- Will choose reduce because precedence of rule E → E * E is higher than of terminal +

Using Precedence to Solve S/R Conflicts

- Same grammar as before

  E → E + E | E * E | int

- We will also have the states

  [E → E + | E, +]   [E → E + E |, +]

  [E → | E + E, +] ⇒E [E → E | + E, +]

  …   …

- Now we also have a shift/reduce on input +
  - We choose reduce because E → E + E and + have the same precedence and + is left-associative

Using Precedence to Solve S/R Conflicts

- Back to our dangling else example

  [S → if E then S |,   else]

  [S → if E then S | else S, x]

- Can eliminate conflict by declaring else having higher precedence than then
- But this starts to look like "hacking the tables"
- Best to avoid overuse of precedence declarations or we will end with unexpected parse trees

Precedence Declarations Revisited

The term "precedence declaration" is misleading!

These declarations do not define precedence: they define conflict resolutions

I.e., they instruct shift-reduce parsers to resolve conflicts in certain ways

These two are not quite the same!

Reduce/Reduce Conflicts

- If a DFA state contains both

  [X → α ı, a] and [Y → β ı, a]

  - Then on input "a" we don't know which production to reduce

- This is called a *reduce/reduce conflict*

Reduce/Reduce Conflicts

- Usually due to gross ambiguity in the grammar
- Example: a sequence of identifiers

  S → ε | id | id S

- There are two parse trees for the string id

  S → id

  S → id S → id

- How does this confuse the parser?

More on Reduce/Reduce Conflicts

- Consider the states

		[S → id ı, $]
[S' → ı S, $]		[S → id ı S, $]
[S → ı, $]	⇒id	[S → ı, $]
[S → ı id, $]		[S → ı id, $]
[S → ı id S, $]		[S → ı id S, $]

- Reduce/reduce conflict on input $

  S' → S → id

  S' → S → id S → id

- Better to rewrite the grammar as: S → ε | id S

Using Parser Generators

- Parser generators automatically construct the parsing DFA given a CFG
  - Use precedence declarations and default conventions to resolve conflicts
  - The parser algorithm is the same for all grammars (and is provided as a library function)
- But most parser generators do not construct the DFA as described before
  - Because the LR(1) parsing DFA has 1000s of states even for a simple language

LR(1) Parsing Tables are Big

- But many states are similar, e.g.

1: E → int •, $/+ (E → int on $, +) and 5: E → int •,)/+ (E → int on), +)

- Idea: merge the DFA states whose items differ only in the lookahead tokens
  - We say that such states have the same core

- We obtain

1′: E → int •, $/+/) (E → int on $, +,))

The Core of a Set of LR Items

Definition: The core of a set of LR items is the set of first components
- Without the lookahead terminals

- Example: the core of

$$\{[X \to \alpha \bullet \beta, b], [Y \to \gamma \bullet \delta, d]\}$$

is

$$\{X \to \alpha \bullet \beta, Y \to \gamma \bullet \delta\}$$

LALR States

- Consider for example the LR(1) states

$$\{[X \to \alpha \bullet, a], [Y \to \beta \bullet, c]\}$$
$$\{[X \to \alpha \bullet, b], [Y \to \beta \bullet, d]\}$$

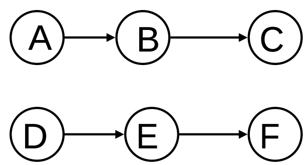
- They have the same core and can be merged
- The merged state contains:

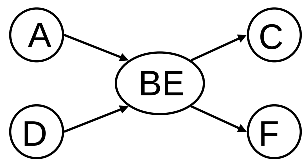
$$\{[X \to \alpha \bullet, a/b], [Y \to \beta \bullet, c/d]\}$$

- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

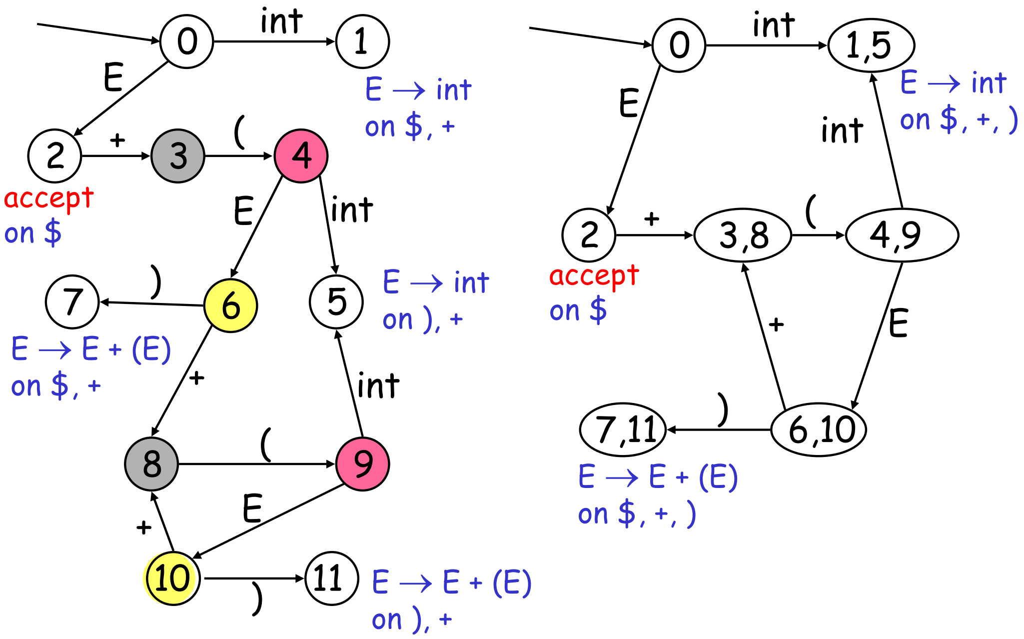
A LALR(1) DFA

- Repeat until all states have distinct core
  - Choose two distinct states with same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from predecessors to new state
  - New state points to all the previous successors





Conversion LR(1) to LALR(1): Example.



The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states

  $\{[X \to \alpha\ \mathrm{I}, a], [Y \to \beta\ \mathrm{I}, b]\}$

  $\{[X \to \alpha\ \mathrm{I}, b], [Y \to \beta\ \mathrm{I}, a]\}$
- And the merged LALR(1) state

  $\{[X \to \alpha\ \mathrm{I}, a/b], [Y \to \beta\ \mathrm{I}, a/b]\}$
- Has a new reduce/reduce conflict

- In practice such cases are rare

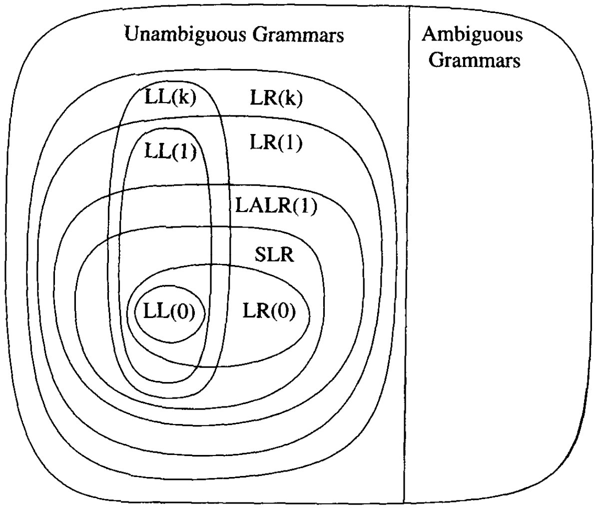
LALR vs. LR Parsing: Things to keep in mind

- LALR languages are not natural
  - They are an efficiency hack on LR languages

- Any reasonable programming language has a LALR(1) grammar

- LALR(1) parsing has become a standard for programming languages and parser generators

A Hierarchy of Grammar Classes



From Andrew Appel, "Modern Compiler Implementation in ML"