CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Administrivia

Agenda

- Review:
  - Context free grammars
  - Ambiguous grammars
  - Parsing
- LR Parsing
- Table-driven Parsers
- Parser States Shift-Reduce and Reduce-Reduce conflicts
- LR(0) state construction
- FIRST, FOLLOW, and nullable
- Variations: SLR, LR(1), LALR(1)
- Top-down parsing

Reading: Cooper & Torczon – chapter 3

(The Dragon book, ch 4. is also particularly strong on grammars and languages)

Credits For Course Material

- **Big thank you to UW CSE faculty member, Hal Perkins**
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, ...)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, ...)

Review: Syntactic Analysis / Parsing

- Goal: Convert token stream to an abstract syntax tree
- Abstract syntax tree (AST):
  - Captures the structural features of the program
  - Primary data structure for next phases of compilation

Review: Context-Free Grammars

- Formally, a *grammar* G is a tuple $<N,\Sigma,P,S>$ where
  - N is a finite set of *non-terminal* symbols
  - Σ is a finite set of *terminal* symbols (alphabet)
  - P is a finite set of *productions*
    - A subset of $N \times (N \cup \Sigma)^*$
  - S is the *start symbol*, a distinguished element of N
    - If not specified otherwise, this is usually assumed to be the non-terminal on the left of the first production

Review: Standard Notations

a, b, c elements of Σ

w, x, y, z elements of Σ^*

A, B, C elements of N

X, Y, Z elements of $N \cup \Sigma$

α, β, γ elements of $(N \cup \Sigma)^*$

$A \rightarrow \alpha$ or $A ::= \alpha$ if $\langle A, \alpha \rangle \in P$

Review: Derivation Relations (1)

- $\alpha\ A\ \gamma \Rightarrow \alpha\ \beta\ \gamma$ iff A ::= β in *P*
  - derives
- A =>* α if there is a chain of productions starting with A that generates α
  - transitive closure

Review: Derivation Relations (2)

- $w\ A\ \gamma \Rightarrow_{lm} w\ \beta\ \gamma$ iff $A ::= \beta$ in P
  - derives leftmost
- $\alpha\ A\ w \Rightarrow_{rm} \alpha\ \beta\ w$ iff $A ::= \beta$ in P
  - derives rightmost
- We will only be interested in leftmost and rightmost derivations – not random orderings

Review: Ambiguity

- Grammar G is *unambiguous* if and only if (iff) every w in $L(G)$ has a unique leftmost (or rightmost) derivation
  - Fact: unique leftmost or unique rightmost implies the other
- A grammar without this property is *ambiguous*
  - Note that other grammars that generate the same language may be unambiguous, i.e., ambiguity is a property of grammars, not languages
- We need unambiguous grammars for parsing

Review: Common Parsing Orderings

- Top-down
  - Start with the root
  - Traverse the parse tree depth-first, left-to-right (leftmost derivation)
  - LL(k), recursive-descent
- Bottom-up
  - Start at leaves and build up to the root
    - Effectively a rightmost derivation in reverse(!)
  - LR(k) and subsets (LALR(k), SLR(k), etc.)

Bottom-Up Parsing

Bottom-Up Parsing

- Idea: Read the input left to right
- Whenever we've matched the right hand side of a production, reduce it to the appropriate non-terminal and add that non-terminal to the parse tree
- Definition 1 – Frontier:

  The upper edge of this partial parse tree is known as the *frontier*

LR(1) Parsing

- We will look at LR(1) parsers
  - Left to right scan, Rightmost derivation, 1 symbol lookahead
  - Almost all practical programming languages have a LR(1) grammar
  - LALR(1), SLR(1), etc. – subsets of LR(1)
    - LALR(1) can parse most real languages, tables are more compact, and is used by YACC/Bison/CUP/etc.

LR Parsing in Greek

- The bottom-up parser reconstructs a reverse rightmost derivation
- Given the rightmost derivation

  $S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$

  the parser will first discover $\beta_{n-1} \Rightarrow \beta_n$, then $\beta_{n-2} \Rightarrow \beta_{n-1}$, etc.
- Parsing terminates when
  - β_1 reduced to S (start symbol, success), or
  - No match can be found (syntax error)

How Do We Parse with This?

- Key: given what we've already seen and the next input symbol (the lookahead), decide what to do.
- Choices:
  - Perform a reduction
  - Look ahead further
- Can reduce $A \Rightarrow \beta$ if both of these hold:
  - $A \Rightarrow \beta$ is a valid production, *and*
  - $A \Rightarrow \beta$ is a step in *this* rightmost derivation
- This is known as a *shift-reduce parser*

Sentential Forms

- **Definition 2 – Sentential form:**

  If S =>* α, the string α is called a *sentential form* of the grammar

- In the derivation
  $S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$
  each of the β_i are sentential forms
- A sentential form in a rightmost derivation is called a right-sentential form (similarly for leftmost and left-sentential)

Handles

- Informally: **a handle -** a production whose right hand-side matches a substring of the tree frontier *that is part of the rightmost derivation of the current input string*

  (i.e., the "correct" production)
  - Even if A ::= β is a production, it is a handle only if β matches the frontier at a point where A ::= β was used in *this specific* derivation
  - β may appear in many other places in the frontier without designating a handle
- Bottom-up parsing is all about finding handles

Handle Example

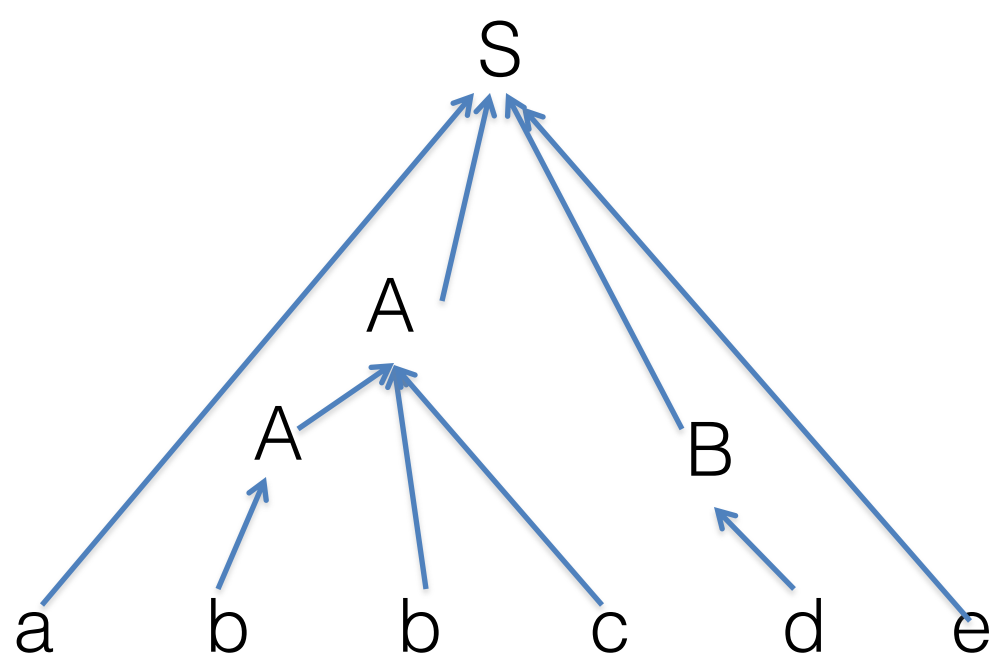
- Grammar

  S ::= aAB e

  A ::= Abc | b

  B ::= d

- Bottom-up Parse



Handle Examples

- In the derivation

  $S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

  – abbcde is a right sentential form whose handle is A::=b at position 2

  – aAbcde is a right sentential form whose handle is A::=Abc at position 4

    - Note: some books take the left end of the match as the position

Handles – The Dragon Book Definition

- Definition 3 – Handle:
- Formally: a *handle* of a right-sentential form γ is a production *A* ::= β and a position in γ where β may be replaced by *A* to produce the previous right-sentential form in the rightmost derivation of γ

Implementing Shift-Reduce Parsers

- **Key data structures**
  - A **stack** holding the frontier of the tree
  - A **string** with the remaining input (tokens)
- We also need something to encode the rules that tell us what action to take next, given the state of the stack and the lookahead symbol
  - Typically a table that encodes a finite automata

Shift-Reduce Parser Operations

- *Reduce* – if the top of the stack is the right side of a handle A::=β, pop the right side β and push the left side A
- *Shift* – push the next input symbol onto the stack
- *Accept* – announce success
- *Error* – syntax error discovered

Shift-Reduce Example

Stack	Input	Action
\$	abbcde\$	*shift*
\$a	bbcde\$	shift
\$ab	bcde\$	reduce
\$aA	bcde\$	shift
\$aAb	cde\$	shift
\$aAbc	de\$	reduce
\$aA	de\$	shift
\$aAd	e\$	reduce
\$aAB	e\$	shift
\$aABe	\$	reduce
\$S	\$	accept

S ::= aABe
A ::= Abc | b
B ::= d

How Do We Automate This?

- Cannot use telepathy in a real parser (alas…)
- **Definition 4 – Viable Prefix:**

  A prefix of a right-sentential form that can appear on the stack of the shift-reduce parser

  – **Equivalent:** a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form

  – In Greek: γ is a *viable prefix* of *G* if there is some derivation $S \Rightarrow^*_{rm} \alpha A w \Rightarrow^*_{rm} \alpha\beta w$ and γ is a prefix of $\alpha\beta$.

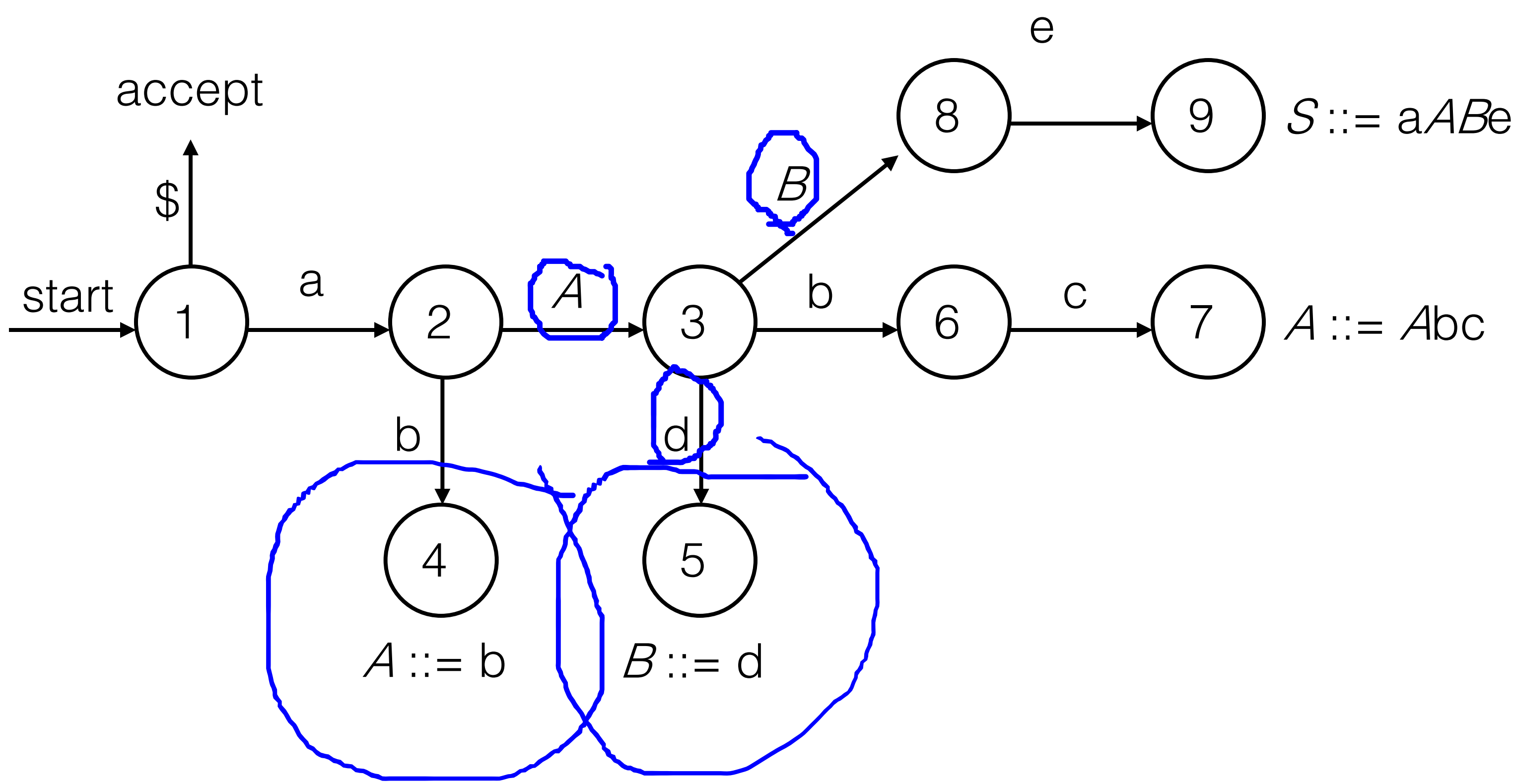
  – The occurrence of β in $\alpha\beta w$ is a *handle* of $\alpha\beta w$

How Do We Automate This?

- Fact: the set of viable prefixes of a CFG is a regular language(!)

- Idea: Construct a DFA to recognize viable prefixes given the stack and remaining input
  - Perform reductions when we recognize them

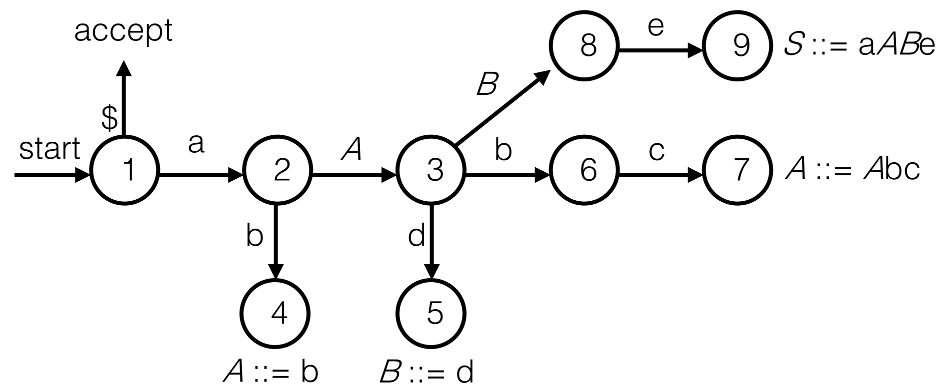
Example: DFA for prefixes of $S ::= aABe$ $A ::= Abc \mid b$ $B ::= d$



Trace

S ::= aABe
A ::= Abc | b
B ::= d

Stack	Input	State
$	abbcde$	1
$a	bbcde$	2
$ab	bcde$	4
$aA	bcde$	3
$aAb	cde$	6
$aAbc	de$	7
$aA	de$	3
$aAd	e$	5
$aAB	e$	8
$aABe	$	9
$S	$	accept



Observations

- Way too much backtracking
  - We want the parser to run in time proportional to the length of the input
- Where the heck did this DFA come from anyway?
  - From the underlying grammar
  - Defer construction details for now

Avoiding DFA Rescanning

- Observation 1: no need to restart DFA after a shift. Stay in the same state and process next token.
- Observation 2: after a reduction, the contents of the stack are the same as before except for the new non-terminal on top
  - ∴ Scanning the stack will take us through the same transitions as before until the last one
  - ∴ If we record state numbers on the stack, we can go directly to the appropriate state when we pop the right hand side of a production from the stack

Stack

- Idea: change the stack to contain pairs of states and symbols from the grammar

  $\$s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_n\ s_n$

  – State s_0 represents the accept (start) state

    (Not always explicitly on stack – depends on particular presentation)

  – When we push a symbol on the stack, push the symbol plus the FA state

  – When we reduce, popping the handle will reveal the state of the FA just prior to reading the handle

- Observation: in an actual parser, only the state numbers are needed since they implicitly contain the symbol information. But for explanations / examples it can help to show both.

Encoding the DFA in a Table

- A shift-reduce parser's DFA can be encoded in two tables
  - One row for each state
  - *action* table encodes what to do given the current state and the next input symbol
  - *goto* table encodes the transitions to take after a reduction

Actions (1)

- Given the current state and input symbol, the main possible actions are
  - s_i – shift the input symbol and state *i* onto the stack (i.e., shift and move to state *i*)
  - r_j – reduce using grammar production *j*
    - The production number tells us how many <symbol, state> pairs to pop off the stack (= number of symbols on rhs of production)

Actions (2)

- Other possible *action* table entries
  - *accept*
  - blank – no transition – syntax error
    - A LR parser will detect an error as soon as possible on a left-to-right scan
    - A real compiler needs to produce an error message, recover, and continue parsing when this happens

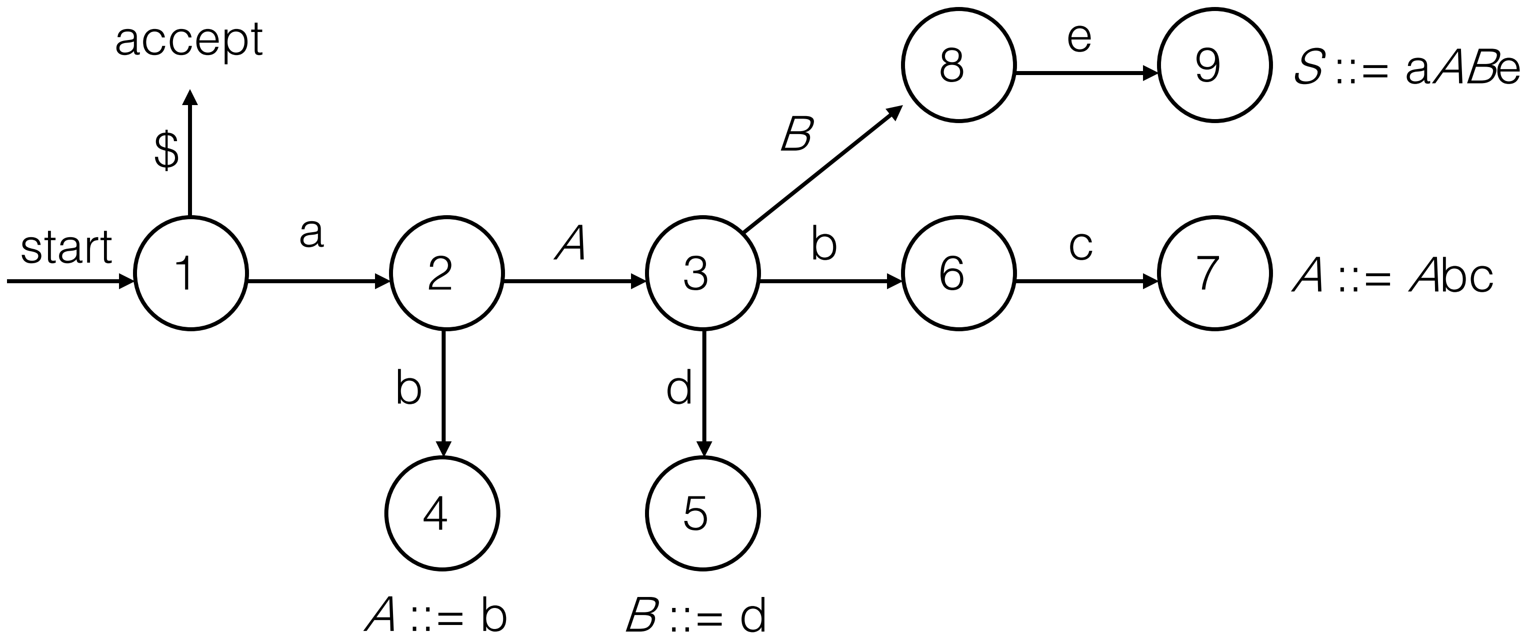
Goto

- When a reduction is performed using A ::= β, we pop |β| <symbol, state> pairs from the stack revealing a state *uncovered_s* on the top of the stack
- goto[*uncovered_s* , *A*] is the new state to push on the stack when reducing production *A* ::= β (after popping handle β and pushing A)

LR States

- Idea is that each state encodes
  - The set of all possible productions that we could be looking at, given the current state of the parse, and
  - *Where* we are in the right hand side of each of those productions

Reminder: DFA for $S ::= aABe$, $A ::= Abc \mid b$, $B ::= d$



LR Parse Table for

1. S ::= aABe
2. A ::= Abc
3. A ::= b
4. B ::= d

State	*action*						*goto*		
	a	b	c	d	e	$	A	B	S
0						acc			
1	s2								g0
2		s4					g3		
3		s6		s5				g8	
4	r3	r3	r3	r3	r3	r3			
5	r4	r4	r4	r4	r4	r4			
6			s7						
7	r2	r2	r2	r2	r2	r2			
8					s9				
9	r1	r1	r1	r1	r1	r1			

Example

1. S ::= aABe
2. A ::= Abc
3. A ::= b
4. B ::= d

Stack	Input
$0	abbcde$
$0 s2	bbcde$
$ 0 a2 b4	bcde$
$ 0 a2 A3	bcde$
$ 0 a2 A3 b6	cde$
$ 0 a2 A3 b6 c7	de$
$ 0 a2 A3	de$
$ 0 a2 A3 d5	e$
$ 0 a2 A3 B8	e$
$ 0 a2 A3 B8 e9	$
$ 0 S	$

S	*action*						*goto*		
	a	b	c	d	e	$	A	B	S
0	s2					ac			
1	s2								g0
2		s4					g3		
3		s6		s5				g8	
4	r3	r3	r3	r3	r3	r3			
5	r4	r4	r4	r4	r4	r4			
6			s7						
7	r2	r2	r2	r2	r2	r2			
8					s9				
9	r1	r1	r1	r1	r1	r1			

LR Parsing Algorithm

```
tok = scanner.getToken();
while (true) {
    s = top of stack;
    if (action[s, tok] = si ) {
       push tok; push i
    (state);
       tok = scanner.getToken();
    } else if (action[s, tok] =
    rj ) {
       pop 2 * length of right
    side of
     production j  (2*|β|);
       uncovered_s = top of
    stack;
       push left side A of
    production j ;
       push state
    goto[uncovered_s, A];
    }
```

```
    } else if (action[s, tok] =
      accept ) {
      return;
    } else {
      // no entry in action
      table
      report syntax error;
      halt or attempt recovery;
    }
```

Items

- Definition 5 – Item:

  An *item* is a production with a dot in the right hand side

- Example: Items for production $A ::= X\ Y$

  $A ::= .\ X\ Y$

  $A ::= X\ .\ Y$

  $A ::= X\ Y\ .$

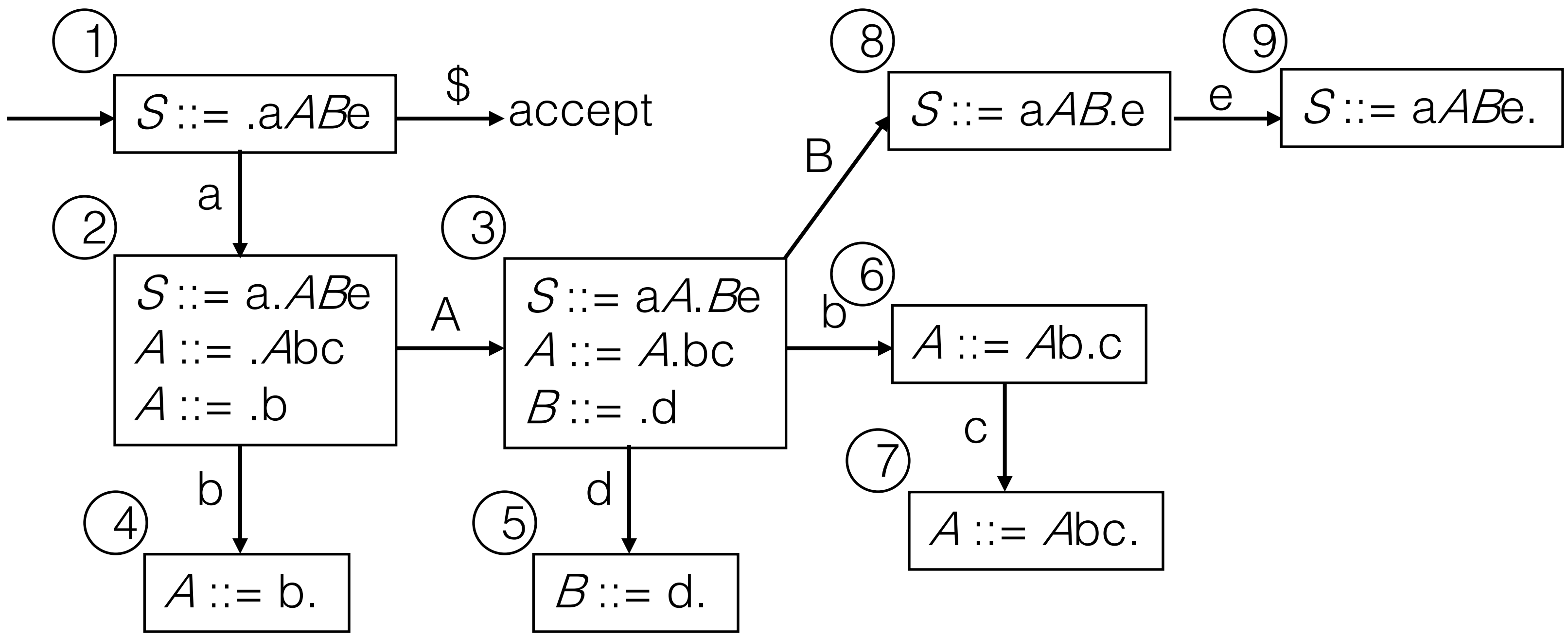
- Idea: The dot represents a position in the production

Summary: Forms, Handles, Prefixes & Items

- If *S* is the start symbol of some grammar *G*, then:
  1. If $S \Rightarrow^* \alpha$ then α is a *sentential form* of *G*
  2. γ is a *viable prefix* of *G* if there is some derivation $S \Rightarrow^*_{rm} \alpha A w \Rightarrow^*_{rm} \alpha\beta w$ and γ is a prefix of $\alpha\beta$.
  3. The occurrence of β in $\alpha\beta w$ is a *handle* of $\alpha\beta w$.
  4. An *item* is a marked production (a . at some position in the right hand side) [*A* ::= . *X Y*]   [*A* ::= *X* . *Y*]   [*A* ::= *X Y* .]

DFA for

$S ::= \mathrm{a}AB\mathrm{e}$
$A ::= A\mathrm{bc} \mid \mathrm{b}$
$B ::= \mathrm{d}$



Problems with Grammars

Problems with Grammars

- Grammars can cause problems when constructing a LR parser
  - Shift-reduce conflicts
  - Reduce-reduce conflicts

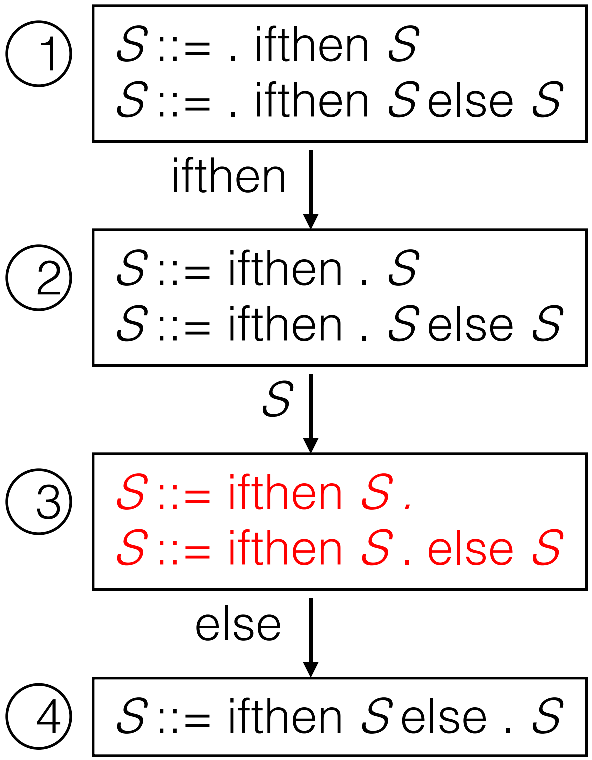
Shift-Reduce Conflicts

- **Situation:** both a shift and a reduce are possible at a given point in the parse (equivalently: in a particular state of the DFA)
- Classic example: if-else statement

  S ::= ifthen *S* | ifthen *S* else *S*

Parser States for

1. S ::= ifthen S
2. S ::= ifthen S else S



- State 3 has a shift-reduce conflict
  - Can shift past else into state 4 (s4)
  - Can reduce (r1)

    S ::= ifthen S

(Note: other S ::= . ifthen items not included in states 2-4 to save space)

Solving Shift-Reduce Conflicts

- Fix the grammar
  - Done in Java reference grammar, others
- Use a parse tool with a "longest match" rule – i.e., if there is a conflict, choose to shift instead of reduce
  - Does exactly what we want for if-else case
  - Guideline: a few shift-reduce conflicts are fine, but be sure they do what you want (and that this behavior is guaranteed by the tool specification)

Reduce-Reduce Conflicts

- Situation: two different reductions are possible in a given state
- Contrived example

$$S ::= A$$

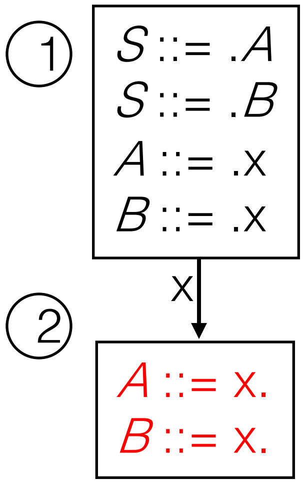
$$S ::= B$$

$$A ::= \mathrm{x}$$

$$B ::= \mathrm{x}$$

Parser States for

1. $S ::= A$
2. $S ::= B$
3. $A ::= x$
4. $B ::= x$



- State 2 has a reduce-reduce conflict (r3, r4)

Handling Reduce-Reduce Conflicts

- These normally indicate a serious problem with the grammar.
- Fixes
  - Use a different kind of parser generator that takes lookahead information into account when constructing the states
    - Most practical tools use this information
  - Fix the grammar

Another Reduce-Reduce Conflict

- Suppose the grammar tries to separate arithmetic and boolean expressions

  expr ::= aexp | bexp

  *aexp ::= aexp * aident | aident*

  bexp ::= bexp && bident | bident

  aident ::= id

  bident ::= id

- This will create a reduce-reduce conflict after recognizing *id*

Covering Grammars

- A solution is to merge *aident* and *bident* into a single non-terminal like *ident* (or just use *id* in place of *aident* and *bident* everywhere they appear)
- This is a *covering grammar*
  - Will generate some programs (sentences) that are not generated by the original grammar
  - Use the type checker or other static semantic analysis to weed out illegal programs later

LR Constructions

LR State Machine

- Idea: Build a DFA that recognizes handles
  - Language generated by a CFG is generally not regular, but
  - Language of viable prefixes for a CFG is regular
    - So a DFA can be used to recognize handles
  - LR Parser reduces when DFA accepts a handle

Building the LR(0) States

- Example grammar

  $S' ::= S\ \$$

  $S ::= (\ L\ )$

  $S ::= \mathrm{x}$

  $L ::= S$

  $L ::= L\ ,\ S$

  – We add a production S' with the original start symbol followed by end of file ($)
    - We accept if we reach the end of this production

  – Question: What language does this grammar generate?

0. $S' ::= S\ \$$
1. $S ::= (\ L\ )$
2. $S ::= \mathrm{x}$
3. $L ::= S$
4. $L ::= L\ ,\ S$

Start of LR Parse

- Initially
  - Stack is empty
  - Input is the right hand side of *S'*, i.e., *S* $
  - Initial configuration is [*S'* ::= . *S* $]
  - But, since position is just before *S*, we are also just before anything that can be derived from *S*

0. $S' ::= S\,\$$
1. $S ::= (\ L\ )$
2. $S ::= \text{x}$
3. $L ::= S$
4. $L ::= L\ ,\ S$

Initial state

$$\boxed{\begin{array}{l} S' ::= .\ S\,\$ \\ S ::= .\ (\ L\ ) \\ S ::= .\ \text{x} \end{array}}$$

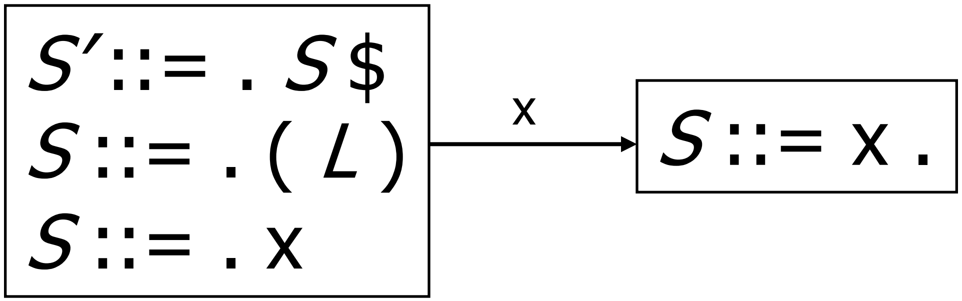
start

completion

- A state is just a set of items
  - Start: an initial set of items
  - Completion (or closure): additional productions whose left hand side appears to the right of the dot in some item already in the state

0. $S' ::= S\,\$$
1. $S ::= (\ L\ )$
2. $S ::= \mathrm{x}$
3. $L ::= S$
4. $L ::= L\ ,\ S$

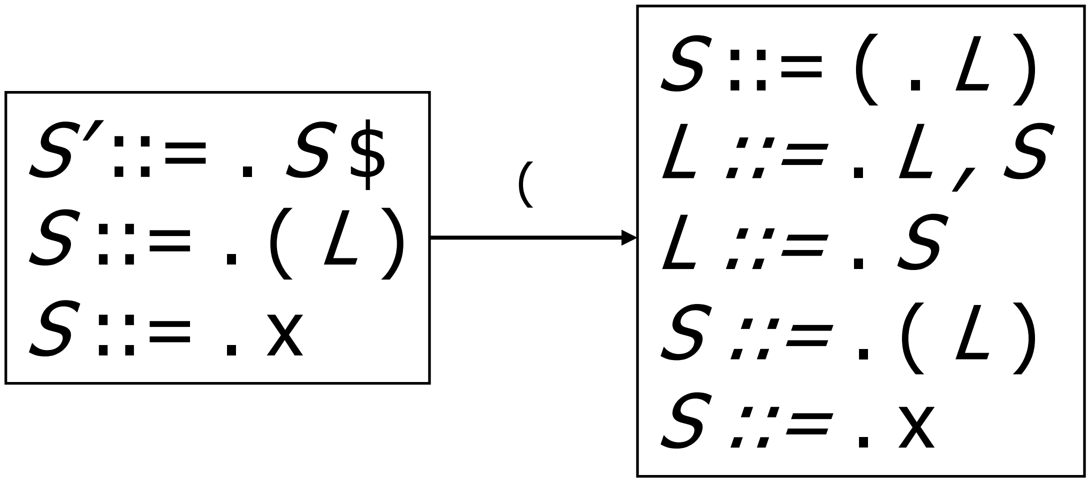
Shift Actions (1)



- To shift past the x, add a new state with appropriate item(s), including their closure
  - In this case, a single item; the closure adds nothing
  - This state will lead to a reduction since no further shift is possible

0. $S' ::= S\,\$$
1. $S ::= (\ L\ )$
2. $S ::= \mathrm{x}$
3. $L ::= S$
4. $L ::= L\ ,\ S$

Shift Actions (2)



- If we shift past the (, we are at the beginning of *L*
- The closure adds all productions that start with *L*, which also requires adding all productions starting with *S*

0. $S' ::= S\,\$$
1. $S ::= (\ L\ )$
2. $S ::= \mathrm{x}$
3. $L ::= S$
4. $L ::= L\ ,\ S$

Goto Actions

$$
\begin{tikzcd}
\boxed{\begin{array}{l} S' ::= .\ S\,\$ \\ S ::= .\ (\ L\ ) \\ S ::= .\ \mathrm{x} \end{array}} \arrow[r, "S"] & \boxed{S' ::= S\ .\ \$}
\end{tikzcd}
$$

- Once we reduce S, we'll pop the rhs from the stack exposing the first state. Add a goto transition on S for this.

Basic Operations

- *Closure* (*S*)
  - Adds all items implied by items already in *S*
- *Goto* (*I*, *X*)
  - *I* is a set of items
  - *X* is a grammar symbol (terminal or non-terminal)
  - *Goto* moves the dot past the symbol *X* in all appropriate items in set *I*

Closure Algorithm

- *Closure* (*S*) =

```
repeat
   for any item [A ::= α . B β] in S
      for all productions B ::= γ
    add [B ::= . γ] to S
until S  does not change
return S
```

- Classic example of a fixed-point algorithm

Goto Algorithm

- *Goto* (*I*, *X*) =

```
 set new  to the empty set
 for each item [A ::= α . X  β]
in I
    add [A ::= α X . β] to new
 return Closure (new )
```

  - This may create a new state, or may return an existing one

LR(0) Construction

- First, augment the grammar with an extra start production *S'* ::= *S* $
- Let *T* be the set of states
- Let *E* be the set of edges
- Initialize *T* to *Closure* ([*S'* ::= . *S* $])
- Initialize *E* to empty

LR(0) Construction Algorithm

repeat
  for each state I in T
    for each item $[A ::= \alpha \,.\, X \;\beta]$ in I
      Let *new* be *Goto*(I, X)
      Add *new* to T if not present
      Add $I \xrightarrow{X}$ *new* to E if not present
until E and T do not change in this iteration

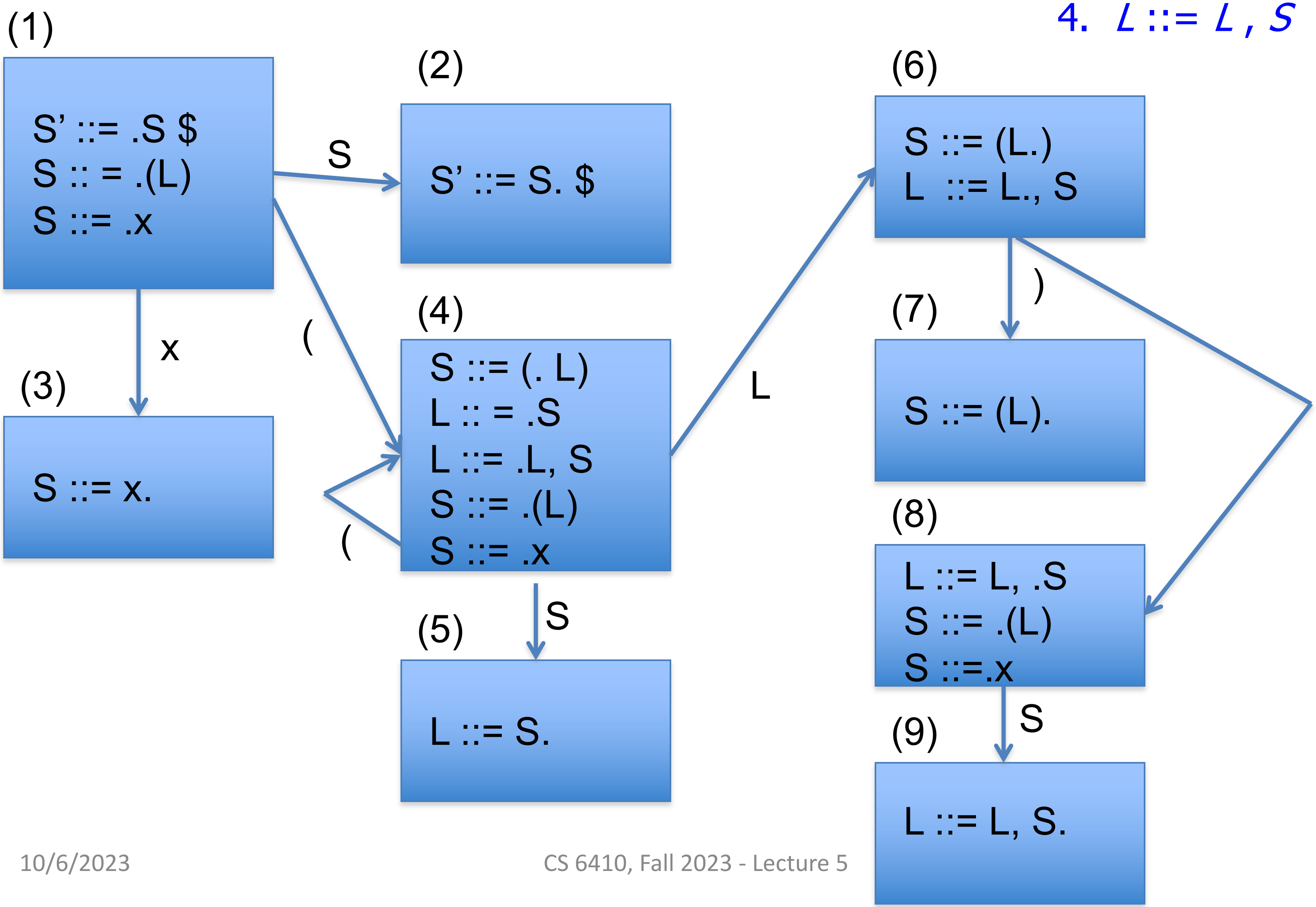
- Footnote: For symbol \$, we don't compute *goto*(I, \$); instead, we make this an *accept* action.

Example: States for

0. $S' ::= S\,\$$
1. $S ::= (\ L\ )$
2. $S ::= \mathrm{x}$
3. $L ::= S$
4. $L ::= L\ ,\ S$

Example: States for

0. $S' ::= S\,\$$
1. $S ::= (\ L\ )$
2. $S ::= \mathrm{x}$
3. $L ::= S$
4. $L ::= L\ ,\ S$



Building the Parse Tables (1)

- For each edge $I \rightarrow_X J$
  - if X is a terminal, put s*j* in column X, row *I* of the action table (shift to state *j*)
  - If X is a non-terminal, put g*j* in column X, row *I* of the goto table (go to state *j*)

Building the Parse Tables (2)

- For each state *I* containing an item [*S'* ::= *S* . $], put *accept* in column $ of row *I*
- Finally, for any state containing [*A* ::= γ .] put action *rn* (reduce) in every column of row *I* in the table, where *n* is the *production* number (*not* a state number)

Example: Lookup Table for

0. $S' ::= S\,\$$
1. $S ::= (\ L\ )$
2. $S ::=$ x
3. $L ::= S$
4. $L ::= L\ ,\ S$

	x	(	)	,	$	S	L
1	s3	s4	-	-	-	g2	
2	-	-	-	-	accept		
3	r2	r2	r2	r2	r2		
4	s3	s4	-	-	-	g5	g6
5	r3	r3	r3	r3	r3		
6	-	-	s7	s8	-		
7	r1	r1	r1	r1	r1		
8	s3	s4	-	-	-	g9	
9	r4	r4	r4	r4	r4		

Where Do We Stand?

- We have built the LR(0) state machine and parser tables
  - No lookahead yet
  - Different variations of LR parsers add lookahead information, but basic idea of states, closures, and edges remains the same
- A grammar is LR(0) if its LR(0) state machine (equiv. parser tables) has no shift-reduce or reduce-reduce conflicts.

A Grammar That is Not LR(0)

- Build the state machine and parse tables for a simple expression grammar

  S ::= E \$

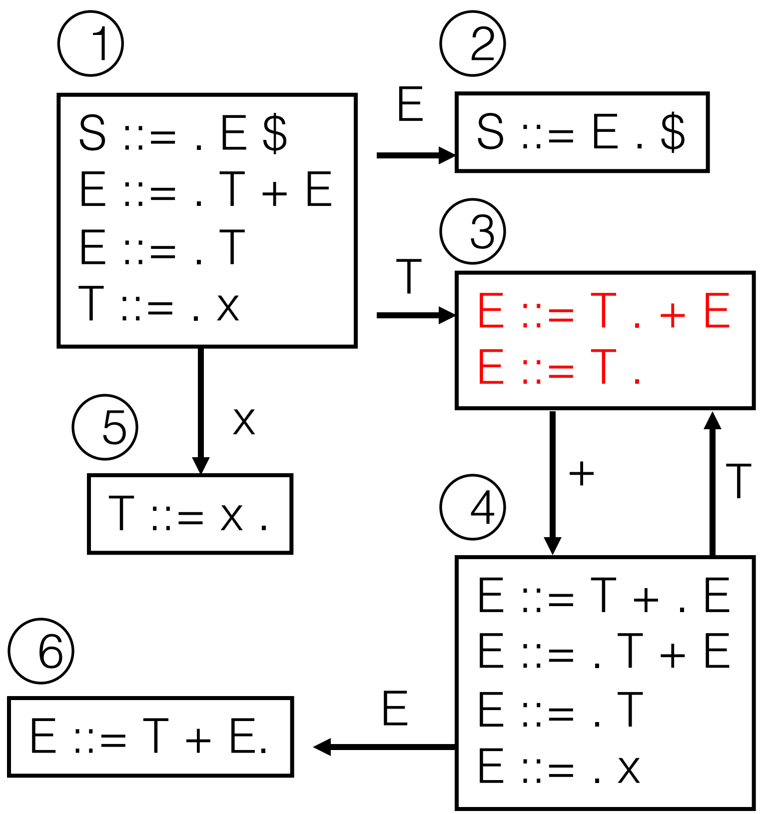
  E ::= T + E

  E ::= T

  T ::= x

LR(0) Parser for

0. $S ::= E\,\$$
1. $E ::= T + E$
2. $E ::= T$
3. $T ::= \mathrm{x}$



	x	+	$	E	T
1	s5			g2	G3
2			acc		
3	r2	s4,r2	r2		
4	s5			g6	G3
5	r3	r3	r3		
6	r1	r1	r1		

- State 3 is has two possible actions on +
  - shift 4, or reduce 2
- ∴ Grammar is not LR(0)

How can we solve conflicts like this?

- Idea: look at the next symbol after the handle before deciding whether to reduce
- Easiest: SLR – Simple LR
  - Reduce only if next input terminal symbol could follow the nonterminal on the left of the production in some possible derivation(s)
- More complex: LR and LALR
  - Store lookahead symbols in items to keep track of what can follow a particular instance of a reduction

SLR Parsers

- Idea:
  1. Use information about what can follow a non-terminal to decide if we should perform a reduction
  2. Don't reduce if the next input symbol can't follow the resulting non-terminal
- We need to be able to compute FOLLOW(*A*) – the set of symbols that can follow *A* in any possible derivation
  - i.e., t is in FOLLOW(*A*) if any derivation contains *A*t
  - To compute this, we need to compute FIRST(γ) for strings γ that can follow *A*

Calculating FIRST(γ)

- Sounds easy... If $\gamma = X\ Y\ Z$, then FIRST(γ) is FIRST(X), right?
  - But what if we have the rule X ::= ε?
  - In that case, FIRST(γ) includes anything that can follow X, i.e. FOLLOW(X), which includes FIRST(Y) and, if Y can derive ε, FIRST(Z), and if Z can derive ε, ...
  - So computing FIRST and FOLLOW involves knowing FIRST and FOLLOW for other symbols, as well as which ones can derive ε.

FIRST, FOLLOW, and nullable

- nullable(X) is true if X can derive the empty string
- Given a string γ of terminals and non-terminals, FIRST(γ) is the set of terminals that can begin any strings derived from γ
  - For SLR we only need this for single terminal or non-terminal symbols, not arbitrary strings γ
- FOLLOW(X) is the set of terminals that can immediately follow X in some derivation
- All three of these are computed together

Computing FIRST, FOLLOW, and nullable (1)

- Initialization
  - Set FIRST and FOLLOW to be empty sets
  - Set nullable to false for all non-terminals
  - Set FIRST[a] to a for all terminal symbols a
- Repeatedly apply four simple observations to update these sets
- Stop when there are no further changes
  - Another fixed-point algorithm

Computing FIRST, FOLLOW, and nullable (2)

repeat
  for each production $X := Y_1\ Y_2 \ldots Y_k$
   if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$)
    set nullable[X] = true
   for each i from 1 to k and each j from i +1 to k
    if $Y_1 \ldots Y_{i-1}$ are all nullable (or if $i = 1$)
     add FIRST[Y_i] to FIRST[X]
    if $Y_{i+1} \ldots Y_k$ are all nullable (or if $i = k$)
     add FOLLOW[X] to FOLLOW[Y_i]
    if $Y_{i+1} \ldots Y_{j-1}$ are all nullable (or if i+1=j)
     add FIRST[Y_j] to FOLLOW[Y_i]
Until FIRST, FOLLOW, and nullable do not change

Example

- Grammar

Z ::= d

Z ::= $X\ Y\ Z$

Y ::= ε

Y ::= c

X ::= Y

X ::= a

	nullable	FIRST	FOLLOW
X	*No→*	*a, c*	*c, d, a*
Y	*No→Yes*	*c*	*d, c, a*
Z	*No*	*d, c, a*	

LR(0) Reduce Actions (review)

- In a LR(0) parser, if a state contains a reduction, it is unconditional regardless of the next input symbol
- Algorithm:

Initialize *R* to empty

for each state *I* in *T*

for each item [A ::= α .] in *I*

add (I, A ::= α) to *R*

SLR Construction

- This is identical to LR(0) – states, etc., except for the calculation of reduce actions
- Algorithm:

Initialize *R* to empty

for each state *I* in *T*

  for each item [A ::= α .] in *I*

    for each terminal a in FOLLOW(*A*)

    add (*I*, a, A ::= α) to *R*

– i.e., reduce α to *A* in state *I* only on lookahead a

On To LR(1)

- Many practical grammars are SLR
- LR(1) is more powerful yet
- Similar construction, but notion of an item is more complex, incorporating lookahead information

LR(1) Items

- An LR(1) item [A ::= α . β, a] is
  - A grammar production (A ::= $\alpha\beta$)
  - A right hand side position (the dot)
  - A lookahead symbol (a)
- Idea: This item indicates that α is the top of the stack and the next input is derivable from βa.

LR(1) Tradeoffs

- LR(1)
  - Pro: extremely precise; largest set of grammars
  - Con: potentially ***very*** large parse tables with many states

LALR(1)

- Variation of LR(1), but merge any two states that differ only in lookahead
  - Example: these two would be merged

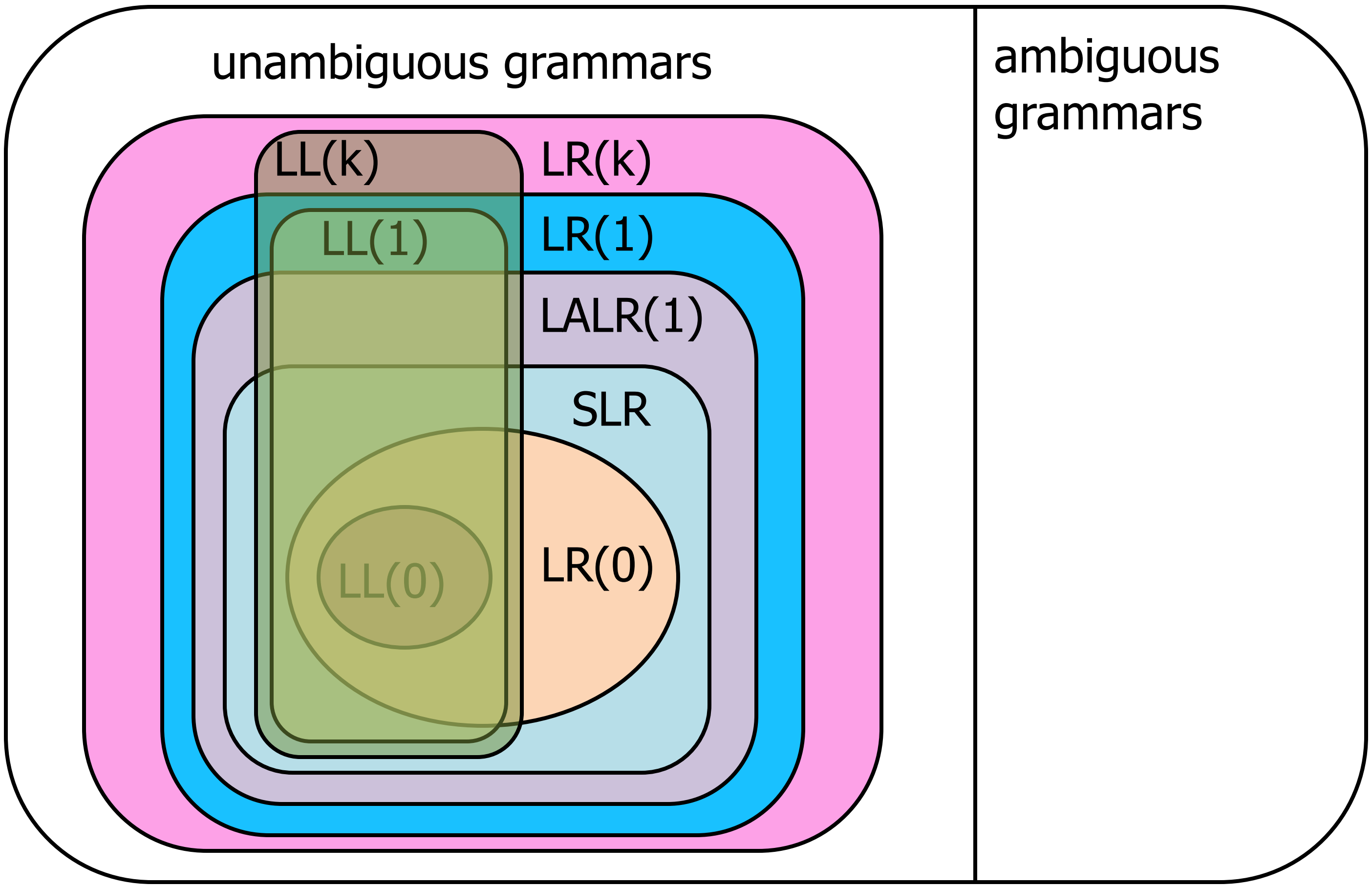
    [*A* ::= x . , a]

    [*A* ::= x . , b]

LALR(1) vs LR(1)

- LALR(1) tables can have many fewer states than LR(1)
  - Somewhat surprising result: will actually have same number of states as SLR parsers, even though LALR(1) is more powerful
  - After the merge step, acts like SLR parser with "smarter" FOLLOW sets (can be specific to particular handles)
- LALR(1) may have reduce conflicts where LR(1) would not (but in practice this doesn't happen often)
- Most practical bottom-up parser tools are LALR(1) (e.g., yacc, bison, CUP, …)

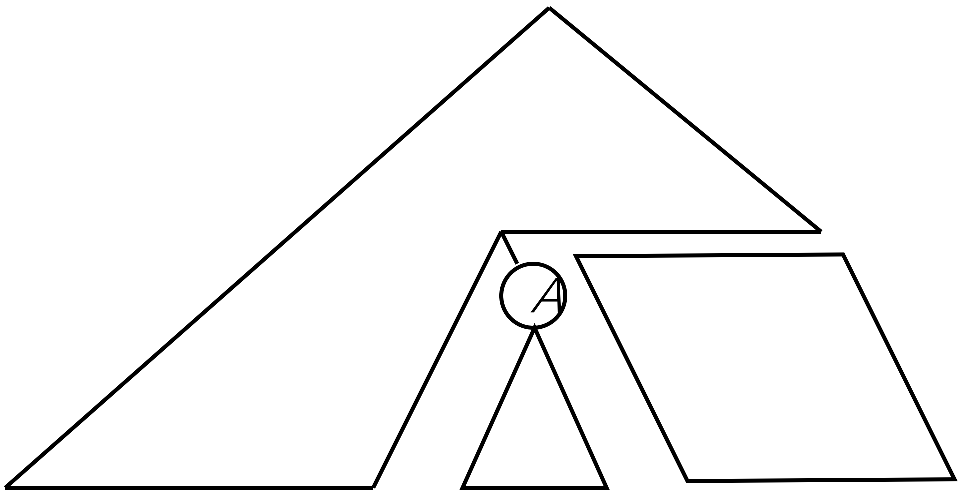
Language Hierarchies



Top-Down Parsing Strategies

Basic Parsing Strategies

- Top-Down
  - Begin at root with start symbol of grammar
  - Repeatedly pick a non-terminal and expand
  - Success when expanded tree matches input
  - LL(k)



Top-Down Parsing

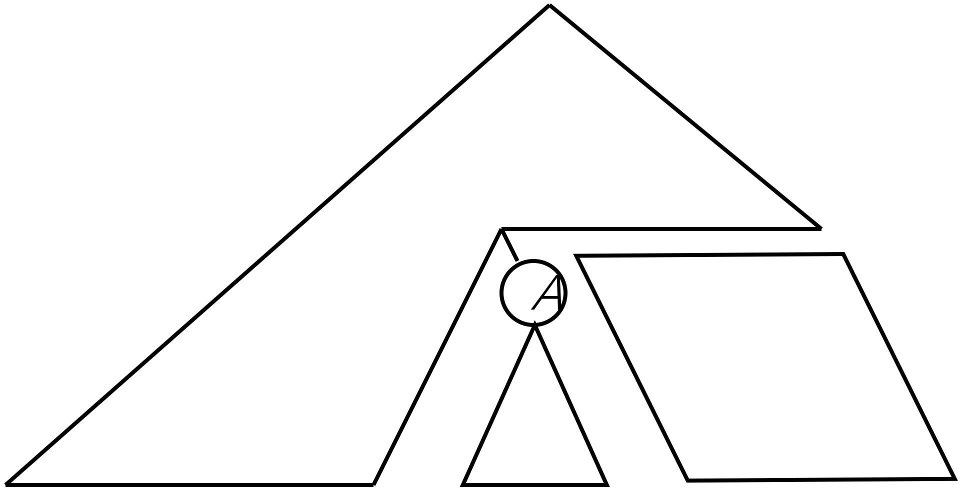
- Situation: have completed part of a left-most derivation

  $S \Rightarrow^* wA\alpha \Rightarrow^* wxy$

- Basic Step: Pick some production

  $A ::= \beta_1\ \beta_2\ \ldots\ \beta_n$

  that will properly expand A to match the input
  - Want this to be deterministic (i.e., no backtracking)



Predictive Parsing

- If we are located at some non-terminal *A*, and there are two or more possible productions

  $A ::= \alpha$

  $A ::= \beta$

  we want to make the correct choice by looking at just the next input symbol
- If we can do this, we can build a *predictive parser* that can perform a top-down parse without backtracking

Example

- Programming language grammars are often suitable for predictive parsing
- Typical example

  stmt ::= *id* = *exp* ; | return *exp* ;
  | if (*exp*) *stmt* | while (*exp*) *stmt*

If the next part of the input begins with the tokens

IF LPAREN ID(x) …

we should expand *stmt* to an if-statement

LL(1) Property

- A grammar has the LL(1) property if, for all non-terminals A, if productions $A ::= \alpha$ and $A ::= \beta$ both appear in the grammar, then it is true that
  $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \emptyset$
- If a grammar has the LL(1) property, we can build a predictive parser for it that uses 1-symbol lookahead

LL(k) Parsers

- An LL(k) parser
  - Scans the input Left to right
  - Constructs a Leftmost derivation
  - Looking ahead at most k symbols
- 1-symbol lookahead is enough for many practical programming language grammars
  - LL(k) for k > 1 is rare in practice

Table-Driven LL(k) Parsers

- As with LR(k), a table-driven parser can be constructed from the grammar
- Example
  1. S ::= (S) S
  2. S ::= [S] S
  3. S ::= ε
- Table

	(	)	[	]	$
S	1	3	2	3	3

LL vs LR (1)

- Tools can automatically generate parsers for both LL(1) and LR(1) grammars
- LL(1) has to make a decision based on a single non-terminal and the next input symbol
- LR(1) can base the decision on the entire left context (i.e., contents of the stack) as well as the next input symbol

LL vs LR (2)

- ∴ LR(1) is more powerful than LL(1)
  - Includes a larger set of languages
- ∴ (editorial opinion) If you're going to use a tool-generated parser, might as well use LR
  - But there are some very good LL parser tools out there (ANTLR, JavaCC, …) that might win for other reasons (documentation, IDE support, integrated AST generation, local culture/politics/economics etc.)

Recursive-Descent Parsers

- A main advantage of top-down parsing is that it is easy to implement by hand
  - And even if you use automatic tools, the code may be easier to follow and debug
- Key idea: write a function (procedure, method) corresponding to each non-terminal in the grammar
  - Each of these functions is responsible for matching its non-terminal with the next part of the input

Example: Statements

Grammar

stmt ::= *id* = *exp* ;
| return *exp* ;
| if (*exp*) *stmt*
| while (*exp*) *stmt*

Method for this grammar rule

```
// parse stmt ::= id=exp; | …
void stmt( ) {
  switch(nextToken) {
     RETURN: returnStmt(); break;
     IF:  ifStmt(); break;
     WHILE: whileStmt(); break;
     ID: assignStmt(); break;
  }
}
```

Example (more statements)

```
// parse while (exp) stmt
void whileStmt() {
    // skip “while” “(”
    skipToken(WHILE);
    skipToken(LPAREN);

    // parse condition
    exp();

    // skip “)”
    skipToken(RPAREN);

    // parse stmt
    stmt();
}
```

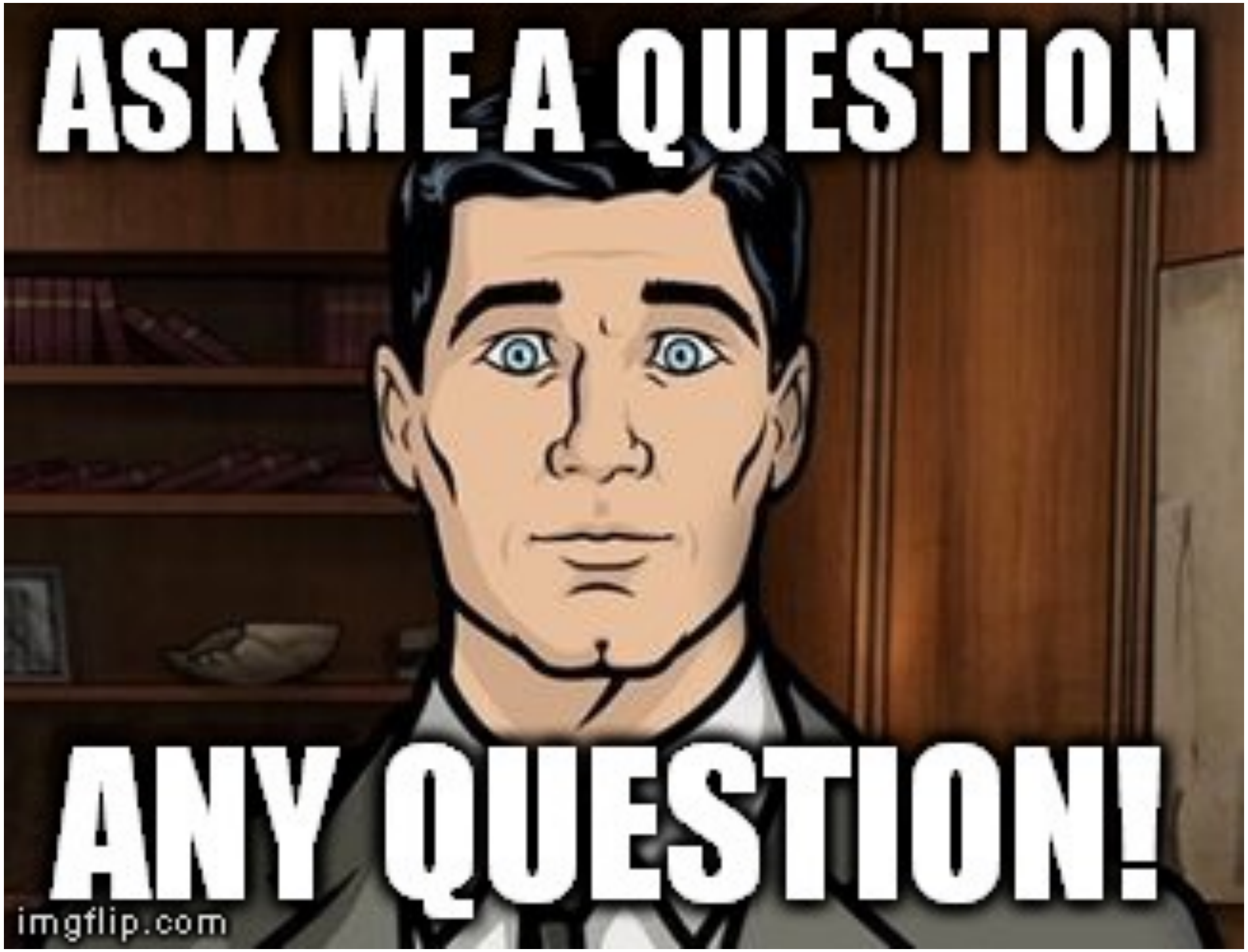
```
// parse return exp ;
void returnStmt() {
    // skip “return”
    skipToken(RETURN);

    // parse expression
    exp();

    // skip “;”
    skipToken(SCOLON);
}


// aux method: advance past expected token
void skipToken(Token expected) {
    if (nextToken == expected)
        getNextToken();
    else error(“token” + expected +
    “expected”);
}
```

Recursive-Descent Recognizer

- Easy!
- Pattern of method calls traces leftmost derivation in parse tree
- Examples only handle valid programs and choke on errors.
- Real parsers need:
  - Better error recovery (don't get stuck on bad token)
  - Semantic checks (declarations, type checking, ...)
  - Some sort of processing after recognizing (build AST, 1-pass code generation, ...)

Invariant for Parser Functions

- The parser functions need to agree on where they are in the input
- Useful invariant: When a parser function is called, the current token (next unprocessed piece of the input) is the token that begins the expanded non-terminal being parsed
  - Corollary: when a parser function is done, it must have completely consumed input correspond to that non-terminal



[Meme credit: imgflip.com]