Bottom-Up Parsing II

Lecture 8
Review: Shift-Reduce Parsing

Bottom-up parsing uses two actions:

*Shift*

\[ ABC|xyz \Rightarrow ABCx|yz \]

*Reduce*

\[ Cbxy|ijk \Rightarrow CbA|i\bar{jk} \]
Recall: The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |  

- Shift pushes a terminal on the stack

- Reduce
  - pops 0 or more symbols off of the stack
    - production rhs
  - pushes a non-terminal on the stack
    - production lhs
Key Issue

• How do we decide when to shift or reduce?

• Example grammar:
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \ast T \mid \text{int} \mid (E) \]

• Consider step \text{int} \mid \ast \text{int} + \text{int}
  - We could reduce by \text{T} \rightarrow \text{int} giving \text{T} \mid \ast \text{int} + \text{int}
  - A fatal mistake!
    • No way to reduce to the start symbol \text{E}
    • Basically, no production that looks like \text{T} \ast
So What This Shows

• We definitely don’t always want to reduce just because we have the right hand side of a production on top of the stack
• To repeat, just because there’s the right hand side of a production sitting on the top of the stack, it might be a mistake to do a reduction
• We might instead want to wait and do our reduction someplace else
• And the idea about how we decide is that we only want to reduce if the result can still be reduced to the start symbol
Recall Some Terminology

• If $S$ is the start symbol of a grammar $G$, and $\alpha$ is such that $S \rightarrow^* \alpha$, then $\alpha$ is a sentential form of $G$
  - Note $\alpha$ may contain both terminals and non-terminals

• A sentence of $G$ is a sentential form that contains no nonterminals.

• Technically speaking, the language of $G$, $L(G)$, is the set of sentences of $G$
Handles

• Intuition: Want to reduce only if the result can still be reduced to the start symbol

• Informally, a handle is a substring that matches the body of a production, and whose reduction (using that production) represents one step along the reverse of a rightmost derivation
Handles

• Intuition: Want to reduce only if the result can still be reduced to the start symbol

• Assume a rightmost derivation

\[ S \rightarrow^* \alpha X \omega \rightarrow \alpha \beta \omega \]

Remember, parser is going from right to left here

• Then production \( X \rightarrow \beta \) is a handle of \( \alpha \beta \omega \)
  - What is means is that you can reduce using this production and still potentially get back to the start symbol
Handles (Cont.)

• Handles formalize the intuition
  - A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)

• We only want to reduce at handles
  - Clearly, since if we do a reduction at a place that is not a handle, we won’t be able to get back to the start symbol!

• Note: We have said what a handle is, not how to find handles
  - Finding handles consumes most of the rest of our discussions of parsing
Techniques for Recognizing Handles

• **Bad news:** There is no known efficient algorithm that recognizes handles in general
  - So for arbitrary grammar, we don’t have a fast way to find handles when we are parsing

• **Good news:** There are heuristics for guessing handles
  - AND for some fairly large classes of context-free grammars, these heuristics always identify the handles correctly
  - For the heuristics we will use, these are the SLR (simple LR) grammars
    • L (**Left-to-right**) R (**Rightmost derivation (in reverse)**)
Important Fact #2

Important Fact #2 about bottom-up parsing:

*In shift-reduce parsing, handles appear only at the top of the stack, never inside*.

Recall that Important Fact #1 was in previous slide set: A bottom-up parser traces a rightmost derivation in reverse.
Why?

- Informal induction on # of reduce moves:
  - True initially, stack is empty
  - Immediately after reducing a handle
     - right-most non-terminal on top of the stack
     - next handle must be to right of right-most non-terminal, because this is a right-most derivation
     - Sequence of shift moves reaches next handle
Summary of Handles

• In shift-reduce parsing, handles always appear at the top of the stack

• Handles are never to the left of the rightmost non-terminal
  - Therefore, shift-reduce moves are sufficient; the need never move left

• Bottom-up parsing algorithms are based on recognizing handles
Recognizing Handles

• There are no known efficient algorithms to recognize handles

• Solution: use heuristics to guess which stacks are handles

• On some CFGs, the heuristics always guess correctly
  - For the heuristics we use here, these are the SLR grammars
  - Other heuristics work for other grammars
Classes of GFGs (equivalently, parsers)

- **Unambiguous CFG**
  - Only one parse tree for a given string of tokens
  - Equivalently, **ambiguous** grammar is one that produces more than one leftmost tree or more than one rightmost tree for a given sentence

- **LR(k) CFG**
  - Bottom-up parser that scans from left-to-right, produces a rightmost derivation, and uses k tokens of lookahead
Classes of GFGs

• LALR(k) CFG
  - Look-ahead LR grammar
  - Basically a more memory efficient variant of an LR parser (at time of development, memory requirements of LR parsers made them impractical)
  - Most parser generator tools use this class

• SLR(k) CFG
  - Simple LR parser
  - Smaller parse table and simpler algorithm
Grammars

Strict containment relations here

All CFGs

Unambiguous CFGs

LR(k) CFGs

LALR(k) CFGs

SLR(k) CFGs

will generate conflicts
Viable Prefixes

• It is not obvious how to detect handles

• At each step the parser sees only the stack, not the entire input; start with that . . .

\( \alpha \) is a viable prefix if there is an \( \omega \) such that

\( \alpha | \omega \) is a valid state of a shift-reduce parse

stack rest of input - parser might know first token of \( \omega \) but typically not more than that
Huh?

• What does this mean? A few things:

  - A viable prefix does not extend past the right end of the handle
  - It’s a viable prefix because it is a prefix of the handle
  - As long as a parser has viable prefixes on the stack no parsing error has been detected

This all isn’t very deep. Saying that $\alpha$ is a viable prefix is just saying that if $\alpha$ is at the top of the stack, we haven’t yet made an error. (These are the valid states of a shift reduce parse.)
Important Fact #3

Important Fact #3 about bottom-up parsing:

*For any grammar, the set of viable prefixes is a regular language*

This is an amazing fact, and one that is the key to bottom-up parsing. Almost all the bottom up parsing tools are based on this fact.
Important Fact #3 (Cont.)

• Important Fact #3 is non-obvious

• We show how to compute automata that accept viable prefixes
  - But first, we’ll need a number of additional definitions
Items

• An *item* is a production with a “.” somewhere on the rhs
  - We can create these from the usual productions

• E.g., the items for $T \to (E)$ are
  
  $T \to (.E)$
  $T \to (.E)$
  $T \to (.E)$
  $T \to (E.)$
  $T \to (E).$
Items (Cont.)

• The only item for $X \rightarrow \varepsilon$ is $X \rightarrow \epsilon$.
  - This is a special case

• Items are often called “LR(0) items”
Intuition

• The problem in recognizing viable prefixes is that the stack has only bits and pieces of the rhs of productions
  - If it had a complete rhs, we could reduce

• These bits and pieces are always prefixes of rhs of productions
It Turns Out...

- ...that what is on the stack is not random
  - It has a special structure: these bits and pieces are always prefixes of the right-hand sides of productions
  - That is, in any successful parse, what is on the stack always has to be a prefix of the right hand side of some production or productions
Example

Consider the input \((\text{int})\)

- Then \((E|)\) is a valid state of a shift-reduce parse
  
  - \((E\) is a prefix of the rhs of \(T \rightarrow (E)\)
    - Will be reduced after the next shift
  
  - Item \(T \rightarrow (E.)\) says that so far we have seen \((E\) of this production and hope to see \)
    - I.e., this item describes this state of affairs (what we've seen and what we hope to see)
    - Note: we may not see what we hope to see
Stated Another Way...

Left of dot tells what we’re working on, right of dot tells what we’re waiting to see before we can perform a reduction
Generalization

• The stack always has the structure of (possibly multiple) prefixes of rhs’s
  - Several prefixes literally stacked up on the stack

  \[ \text{Prefix}_1 \text{Prefix}_2 \ldots \text{Prefix}_{n-1}\text{Prefix}_n \]
Generalization

- Let $\text{Prefix}_i$ be a prefix of rhs of $X_i \rightarrow \alpha_i$
  - $\text{Prefix}_i$ will eventually reduce to $X_i$
  - The missing part of $\alpha_{i-1}$ starts with $X_i$ (i.e., the missing suffix of $\alpha_{i-1}$ must start with $X_i$)
  - That is, when we eventually replace $\alpha_i$ with $X_i$, $X_i$ must extend prefix $\alpha_{i-1}$ to be closer to a complete rhs of production $X_{i-1} \rightarrow \alpha_{i-1}$
  
  - i.e. there is a production $X_{i-1} \rightarrow \text{Prefix}_{i-1} X_i \beta$ for some $\beta$ (which may be empty)
Generalization

- Recursively, \( \text{Prefix}_{k+1} \ldots \text{Prefix}_n \) must eventually reduce to the missing part of \( \alpha_k \)
  - That is, all the prefixes above prefix \( k \) must eventually reduce to the missing part of \( \alpha_k \)

- So the image:
  - A stack of prefixes.
  - Always working on the topmost (shifting and reducing)
  - But every time we perform a reduction, that must extend the prefix immediately below it on the stack
  - And when a bunch of these have been removed via reduction, then we get to work on prefixes lower in the stack
An Example

Consider the string \((\text{int} \ast \text{int})\):

\((\text{int} \ast | \text{int})\) is a state of a shift-reduce parse

“\(\)’’ is a prefix of the rhs of \(T \rightarrow (E)\)

Seen \(\), working on rest of this production

“\(\varepsilon\)’’ is a prefix of the rhs of \(E \rightarrow T\)

We’ve seen none of this

“\(\text{int} \ast\)’’ is a prefix of the rhs of \(T \rightarrow \text{int} \ast T\)

We’ve seen the start of this
An Example (Cont.)

Can record this with the “stack of items”

\[ T \rightarrow (.E) \]
\[ E \rightarrow .T \]
\[ T \rightarrow \text{int} \ast .T \]

Says

We've seen “(” of \( T \rightarrow (E) \)
We've seen \( \varepsilon \) of \( E \rightarrow T \)
We've seen \( \text{int} \ast \) of \( T \rightarrow \text{int} \ast T \)

Note how lhs of each production is going to become part of rhs of previous production
Recognizing Viable Prefixes

Idea: To recognize viable prefixes, we must

- Recognize a sequence of partial right hand sides of productions, where

- Each sequence can eventually reduce to part of the missing suffix of its predecessor
Now, onto the algorithm for implementing this idea.

That is, we finally get to the technical highlight of bottom-up parsing!
An NFA Recognizing Viable Prefixes

• We’re going to build an NFA that recognizes the viable prefixes of $G$
• This should be possible, since we claim that the set of viable prefixes of $G$ are a regular language
• Input to the NFA is the stack (the NFA reads the stack)
  • Reads the stack from bottom to top
• Output of NFA is either yes (viable prefix) or no (not viable prefix)
An NFA Recognizing Viable Prefixes

• But let’s be clear what this is saying:
  • Yes: This stack is OK, we could end up parsing the input
  • No: What we’ve got on the stack now doesn’t resemble any valid stack for any possible parse of any input string for this grammar.
An NFA Recognizing Viable Prefixes

1. Add a dummy production $S' \rightarrow S$ to $G$
2. The NFA states are the items of $G$
   - Including the extra production
3. For item $E \rightarrow \alpha \cdot X \beta$ add transition
   \[ E \rightarrow \alpha \cdot X \beta \rightarrow^X E \rightarrow \alpha X \beta \]

And remember, items are the states of the NFA, so this is telling us when to move from one state to another (e.g., a transition)

X can be terminal or non-terminal here

Simplifies some things

means “on input $X$”
An NFA Recognizing Viable Prefixes

4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add

$$E \rightarrow \alpha.X\beta \rightarrow \epsilon \quad X \rightarrow .\gamma$$

Is it possible that there could be a valid configuration of the parser where we saw $\alpha$ but then $X$ didn’t appear next?

Yes: stack is sequence of partial rhs

Could be that all that’s on the stack right now for this production is $\alpha$ and the next thing on the stack is eventually going to reduce to $X$
An NFA Recognizing Viable Prefixes

What does this mean? It means that whatever is there on the stack has to be derived from \( X \), (has to be something that can be generated by using a sequence of \( X \) productions), because eventually it’s going to reduce the \( X \)

So, for every item that looks like this, and for every production for \( X \), we’re going to say that if there is no \( X \) on the stack, we make an \( \epsilon \) move, we shift to a state where we can try to recognize the rhs plus something derived from \( X \)
An NFA Recognizing Viable Prefixes

1. Add a dummy production $S' \rightarrow S$ to $G$

2. The NFA states are the items of $G$
   - Including the extra production

3. For item (NFA state) $E \rightarrow \alpha.X\beta$ add transition
   $E \rightarrow \alpha.X\beta \rightarrow X E \rightarrow \alpha.X\beta$

4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add
   $E \rightarrow \alpha.X\beta \rightarrow^\varepsilon X \rightarrow .\gamma$

point must mark the start of another rhs
An NFA Recognizing Viable Prefixes (Cont.)

5. Every state is an accepting state
   If the automaton successfully consumes the entire stack, then the stack is viable

   Note not every state is going to have a transition on every symbol (so lot of possible stacks will be rejected)

6. Start state is \( S' \rightarrow .S \)
   Dummy production added so we can conveniently name the start state
An Example

\[ S' \rightarrow E \]
\[ E \rightarrow T + E \mid T \]
\[ T \rightarrow \text{int} \times T \mid \text{int} \mid (E) \]

The usual, augmented with the extra production
NFA for Viable Prefixes of the Example

\[ T \rightarrow .(E) \]
\[ T \rightarrow (.E) \]
\[ T \rightarrow (E.) \]
\[ T \rightarrow (E). \]
\[ S' \rightarrow E. \]
\[ S' \rightarrow .E \]
\[ E \rightarrow .T+E \]
\[ E \rightarrow T.+E \]
\[ E \rightarrow T+.E \]
\[ E \rightarrow T+.E. \]
\[ T \rightarrow .int \]
\[ T \rightarrow .int * T \]
\[ T \rightarrow int * T \]
\[ T \rightarrow int * .T \]
\[ T \rightarrow int * T. \]
The Example Automata

• It’s actually quite complicated
  – Many states and transitions
  – And this is for a relatively simple grammar, so…
• These NFAs for recognizing viable prefixes for grammars are actually quite elaborate

Let’s build it!
So, this is saying we’re hoping to see an E on the stack. If we don’t, then we’re hoping to see something derived from E.
If we see and E on the stack, we move the dot over, saying we’ve read the E on the stack. (This would indicate we’re probably done with parsing - you would have reduced the old start symbol and be about to reduce to the new start symbol.)

Lacking E at top of stack, hope for either a T or a T + E (use Rule 4)
Notice how we crucially use the power of nondeterministic automata. We don’t know which rhs of a production is going to appear on the stack (and this grammar isn’t even left-factored) so we just use the guessing power of the NFA, let it choose which one to use (because it can always guess correctly).
Another way of describing the intuition here: From the red box, we’d like to see an E as the next input symbol. If we do, then we can reduce to the start state $S'$. But, if we don’t see an E next, then we’d like to see something that can be derived from E, because then we could eventually reduce that derived thing back to E, and then back to $S'$. The $\varepsilon$-transitions take us to things that can be derived from E.
Of course we can compile this down to a DFA, but at this point it’s really nice to not have to know which choice to make.
NFA for Viable Prefixes in Detail (3)

Same as before. Note that when we see a dot at the end of a rhs, we've got a handle that we might want to reduce.

We'll talk about this later, but this is kind of how we're going to go about recognizing handles.
Note that in these newly added links, the dot is all the way to the left, indicating that we haven’t actually seen anything from the rhs of the production yet.

\[
S' \rightarrow E
\]

\[
E \rightarrow .T+E
\]

\[
T \rightarrow .\text{int}
\]

\[
E \rightarrow .T
\]

\[
T \rightarrow .\text{int} \ast T
\]

\[
E \rightarrow T.
\]

\[
S' \rightarrow E
E \rightarrow T + E \mid T
T \rightarrow \text{int} \ast T \mid \text{int} \mid (E)
\]
NFA for Viable Prefixes in Detail (4)

Note that when we see a non-terminal to the right of the dot, we create a link corresponding to that non-terminal being on top of the stack, and other links (ε-transitions) to the other productions that derive things from that non-terminal.
Note that when we see a terminal to the right of the dot, there is only one possible move: we create a link corresponding to that terminal being on top of the stack.
NFA for Viable Prefixes in Detail (7)
NFA for Viable Prefixes in Detail (8)
NFA for Viable Prefixes in Detail (9)
NFA for Viable Prefixes in Detail (10)
NFA for Viable Prefixes in Detail (11)
NFA (for Viable Prefixes) to DFA

Start symbol
NFA (for Viable Prefixes) to DFA
NFA (for Viable Prefixes) to DFA
NFA (for Viable Prefixes) to DFA
Translation to the DFA

\[ S' \rightarrow E \]
\[ E \rightarrow T \]
\[ E \rightarrow T + E \]
\[ T \rightarrow (E) \]
\[ T \rightarrow \text{int} * T \]
\[ T \rightarrow \text{int} \]

\[ E \rightarrow T + E \]
\[ E \rightarrow . T \]
\[ E \rightarrow . T + E \]
\[ T \rightarrow (E) \]
\[ T \rightarrow \text{int} * T \]
\[ T \rightarrow \text{int} \]
Lingo

The states of the DFA are

“canonical collections of items”

or

“canonical collections of LR(0) items”

The Dragon book gives another way of constructing LR(0) items

(It’s a little more complex, and a bit more difficult to understand if you’re seeing this for the first time.)
Valid Items

Item $X \rightarrow \beta.\gamma$ is *valid* for a viable prefix $\alpha\beta$ if

$$S' \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\gamma\omega$$

by a right-most derivation

After parsing $\alpha\beta$, the valid items are the possible tops of the stack of items

Remember what this item is saying: “I hope to see $\gamma$ at the top of the stack so that I can reduce $\beta\gamma$ to $X$.” Saying the item is valid is saying that doing that reduction leaves open the possibility of a successful parse
Simpler Way: Items Valid for a Prefix

For a given viable prefix $\alpha$, the items that are valid (in that prefix) are exactly the items that are in the final state of the DFA after it reads $\alpha$.

(An item $I$ is valid for a given viable prefix $\alpha$ if the DFA that recognizes viable prefixes terminates on input $\alpha$ in a state $s$ containing $I$)

The items in $s$ describe what the top of the item stack might be after reading input $\alpha$

(So valid items describe what might be on top of stack after you've seen stack $\alpha$)
Valid Items Example

• An item is often valid for many prefixes

• Example: The item $T \rightarrow (\cdot E)$ is valid for prefixes

(  
(  
(  
(  
(  
(  
...  

To see this, look at the DFA...
Valid Items for (((...
So Now...

- We’re finally going to give the actual bottom-up parsing algorithm. But first (because we like to keep you in suspense), we give a few minutes to a very weak, but very similar, bottom-up parsing algorithm.

- All this builds on what we’ve been talking about so far.
LR(0) Parsing Algorithm

• Idea: Assume
  - stack contains $\alpha$
  - next input is $t$
  - DFA on input $\alpha$ terminates in state $s$. (i.e., $s$ is a final state of the DFA) Then...

• Reduce by $X \rightarrow \beta$ if
  - $s$ contains item $X \rightarrow \beta$. (we’ve seen entire rhs here)

• Shift if
  - $s$ contains item $X \rightarrow \beta.t\omega$ (OK to add $t$ to the stack)
  - equivalent to saying $s$ has a transition labeled $t$
LR(0) Conflicts (i.e. When does LR(0) get into trouble)

- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
    - $X \rightarrow \beta$. and $Y \rightarrow \omega$. (two complete rhs, could do either)
    - (might not be able to decide between two possible reduce moves)

- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
    - $X \rightarrow \beta$. and $Y \rightarrow \omega.\delta$
    - (Do we shift or reduce?)
    - Only an issue if $\delta$ is the next item in the input
if next input is +, can do shift or reduce here

LR(0) Conflicts

Two shift/reduce conflicts with LR(0) rules
if next input is *, can do shift or reduce here

**LR(0) Conflicts**

Two shift/reduce conflicts with LR(0) rules
SLR

- LR = “Left-to-right scan”
- SLR = “Simple LR”

- SLR improves on LR(0) shift/reduce heuristics
  - Add one small heuristic so that fewer states have conflicts
SLR Parsing

• Idea: Assume
  - stack contains $\alpha$
  - next input is $t$
  - DFA on input $\alpha$ terminates in state $s$

• Reduce by $X \rightarrow \beta$ if
  - $s$ contains item $X \rightarrow \beta$. ($\beta$ is on top of stack and valid)
  - $t \in \text{Follow}(X)$

• Shift if
  - $s$ contains item $X \rightarrow \beta. t \omega$
So what’s going on here?

• We have our stack contents
• It ends in a $\beta$
• And now we’re going to make a move (a reduction) and replace that $\beta$ by $X$
• So things look like this $\beta \mid \tau$
• And we’re going to make it look like $X \mid \tau$
• And this makes no sense whatsoever if $\tau$ cannot follow $X$
• So we add the restriction that we only make the move if $\tau$ can follow $X$
SLR Parsing (Cont.)

• If there are conflicts under these rules, the grammar is not SLR

• The rules amount to a heuristic for detecting handles
  - **Defn:** The SLR grammars are those where the heuristics detect exactly the handles
  - So we take into account two pieces of information: (1) the contents of the stack (that’s what the DFA does for us - it tells us what items are possible when we get to the top of the stack) and (2) what’s coming in the input
SLR Conflicts

Follow(E) = { ‘)’, $ }
Follow(T) = { ‘+’, ‘)’, $ }

No conflicts with SLR rules!
Precedence Declarations Digression

• Lots of grammars aren’t SLR
  - including all ambiguous grammars
  - So SLR is an improvement, but not a really very general class of grammar

• We can make SLR parse more grammars by using precedence declarations
  - Instructions for resolving conflicts
• Consider our favorite ambiguous grammar:
  - \( E \rightarrow E + E \mid E \times E \mid (E) \mid \text{int} \)

• The DFA for this grammar contains a state with the following items:
  - \( E \rightarrow E \times E \)
  - \( E \rightarrow E \cdot + E \)
  - shift/reduce conflict!
  - Exactly the question of precedence of + and *

• Declaring “* has higher precedence than +” resolves this conflict in favor of reducing
  - Note that reducing here places the * lower in the parse tree!
Precedence Declarations (Cont.)

- The term “precedence declaration” is misleading
- These declarations do not define precedence; they define conflict resolutions
  - Not quite the same thing!
  - In our example, because we’re dealing with a natural grammar, conflict resolution has same effect as enforcing precedence we want.
  - But more complicated grammar -> more interactions between various pieces of grammar -> you might not do what you want in terms of enforcing precedence
Fortunately, the tools provide means for printing out the parsing automaton, which allows you to see exactly how conflicts are being resolved and whether these are the resolutions you expected.

It is recommended that when you are building a parser, especially a fairly complex one, that you examine the parsing automaton to make sure that it’s doing exactly what you expect.
Naïve SLR Parsing Algorithm

1. Let $M$ be DFA for viable prefixes of $G$
2. Let $|x_1...x_n|S$ be initial configuration
3. Repeat until configuration is $S|$
   • Let $α|ω$ be current configuration
   • Run $M$ on current stack $α$
   • If $M$ rejects $α$, report parsing error
     • Stack $α$ is not a viable prefix
   • If $M$ accepts $α$ with items $I$, let $a$ be next input
     • Shift if $X → β. a γ ∈ I$
     • Reduce if $X → β. ∈ I$ and $a ∈ \text{Follow}(X)$
     • Report parsing error if neither applies
Naïve SLR Parsing Algorithm

1. Let $M$ be DFA for viable prefixes of $G$
2. Let $|x_1...x_n|$ be initial configuration
3. Repeat until configuration is $S|$
   • Let $\alpha|\omega$ be current configuration
   • Run $M$ on current stack $\alpha$
   • If $M$ rejects $\alpha$, report parsing error
     • Stack $\alpha$ is not a viable prefix
   • If $M$ accepts $\alpha$ with items $I$, let $a$ be next input
     • Shift if $X \rightarrow \beta. a \gamma \in I$
     • Reduce if $X \rightarrow \beta. \in I$ and $a \in \text{Follow}(X)$
     • Report parsing error if neither applies

---

not needed: bottom checks will always catch parse errors - we will never form an invalid stack
Notes

- If there is a conflict in the last step, grammar is not SLR(k)

- k is the amount of lookahead
  - In practice k = 1
## Extended Example of SLR Parsing

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Example of SLR Parsing

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
</tbody>
</table>
**SLR Example**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int * int$</code></td>
<td>1</td>
<td><code>shift</code></td>
</tr>
<tr>
<td>`int</td>
<td>* int$`</td>
<td></td>
</tr>
</tbody>
</table>
Configuration int | * int$

\[
\begin{align*}
S' &\rightarrow E. \\
E &\rightarrow T. \\
T &\rightarrow .(E) \\
T &\rightarrow .int * T \\
T &\rightarrow .int
\end{align*}
\]

\[
\begin{align*}
T &\rightarrow T + E. \\
E &\rightarrow T + E \\
T &\rightarrow (E) \\
T &\rightarrow .int
\end{align*}
\]

\[
\begin{align*}
E &\rightarrow T + E. \\
E &\rightarrow T + E \\
T &\rightarrow (E) \\
T &\rightarrow .int
\end{align*}
\]
### SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>* int$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int * int$</td>
<td>3 * not in Follow(T)</td>
<td>shift</td>
</tr>
<tr>
<td>int *</td>
<td>int$</td>
<td></td>
</tr>
</tbody>
</table>
Configuration \textbf{int * | int$}\$

1. \textbf{S'} \rightarrow \textbf{E.}
2. \textbf{E} \rightarrow \textbf{T.}
3. \textbf{E} \rightarrow \textbf{T. + E}
4. \textbf{T} \rightarrow \textbf{int * T.}
5. \textbf{T} \rightarrow \textbf{int.}
6. \textbf{T} \rightarrow \textbf{.T + E}
7. \textbf{E} \rightarrow \textbf{T + E.}
8. \textbf{T} \rightarrow \textbf{.E}
9. \textbf{T} \rightarrow \textbf{.int * T}
10. \textbf{T} \rightarrow \textbf{.int}
11. \textbf{T} \rightarrow \textbf{.T}

\[ \text{int} \rightarrow \text{int * T.} \]

\[ \text{int} \rightarrow \text{int.} \]

\[ \text{int} \rightarrow \text{.T + E} \]

\[ \text{int} \rightarrow \text{.E} \]

\[ \text{int} \rightarrow \text{.T} \]

\[ \text{int} \rightarrow \text{.int} \]
Configuration int * | int$

\[
\begin{align*}
S' &\rightarrow E. \\
E &\rightarrow T. \\
T &\rightarrow int. \\
E &\rightarrow T + E \\
T &\rightarrow (E) \\
E &\rightarrow int * T \\
T &\rightarrow int \\
\end{align*}
\]
Configuration \( \text{int} \ast | \text{int}\$ \)
### SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>* int$</td>
<td>3</td>
</tr>
<tr>
<td>int *</td>
<td>int$</td>
<td>11</td>
</tr>
</tbody>
</table>
## SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>* int$</td>
<td>3</td>
</tr>
<tr>
<td>int *</td>
<td>int$</td>
<td>11</td>
</tr>
<tr>
<td>int * int</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>
Configuration \texttt{int * int\$}
## SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>* int$</td>
<td>3</td>
</tr>
<tr>
<td>int *</td>
<td>int$</td>
<td>11</td>
</tr>
<tr>
<td>int * int</td>
<td>$</td>
<td>3</td>
</tr>
</tbody>
</table>
### SLR Example

**Configuration** | **DFA Halt State** | **Action**
---|---|---
| `int * int$` | 1 | shift
| `int | * int$` | 3 | * not in Follow(T) | shift
| `int * | int$` | 11 | shift
| `int * int |$` | 3 | $ \in \text{Follow}(T)$ | red. $T \rightarrow \text{int}$
| `int * T |$` | | |
Configuration $\text{int} \ast T | \$
<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>* int$</td>
<td>3</td>
</tr>
<tr>
<td>int *</td>
<td>int$</td>
<td>11</td>
</tr>
<tr>
<td>int * int</td>
<td>$</td>
<td>3</td>
</tr>
<tr>
<td>int * T</td>
<td>$</td>
<td>4</td>
</tr>
</tbody>
</table>

SLR Example
### SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>* int$</td>
<td>3 * not in Follow(T)</td>
</tr>
<tr>
<td>int *</td>
<td>int$</td>
<td>11</td>
</tr>
<tr>
<td>int * int</td>
<td>$</td>
<td>3 $ ∈ Follow(T)</td>
</tr>
<tr>
<td>int * T</td>
<td>$</td>
<td>4 $ ∈ Follow(T)</td>
</tr>
<tr>
<td>T</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>
## SLR Example

### Configuration

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int $</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>* int $</td>
<td>3</td>
</tr>
<tr>
<td>int *</td>
<td>int $</td>
<td>11</td>
</tr>
<tr>
<td>int * int</td>
<td>$</td>
<td>3</td>
</tr>
<tr>
<td>int * T</td>
<td>$</td>
<td>4</td>
</tr>
<tr>
<td>T</td>
<td>$</td>
<td>5</td>
</tr>
</tbody>
</table>
## SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int</td>
<td>* int$</td>
<td>3</td>
</tr>
<tr>
<td>int *</td>
<td>int$</td>
<td>11</td>
</tr>
<tr>
<td>int * int</td>
<td>$</td>
<td>3</td>
</tr>
<tr>
<td>int * T</td>
<td>$</td>
<td>4</td>
</tr>
<tr>
<td>T</td>
<td>$</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>
Configuration | int * int$
Configuration | int * int$

1
S' → E.
E → . E
E → . T + E
T → . (E)
T → . int * T
T → . int

2
E → T.
E → T. + E
T → . (E)
T → . int

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T. + E
T → . (E)
T → . int * T
T → . int

7
E → T + E.

8
T → (E.)

9
T → (E.)

10
T → (E.)

11
T → int * . T
T → . (E)
T → . int * T
T → . int

120
### SLR Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>DFA Halt State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int$</td>
<td>1</td>
<td>shift</td>
</tr>
<tr>
<td>int * int$</td>
<td>3</td>
<td>* not in Follow(T)</td>
</tr>
<tr>
<td>int * int$</td>
<td>11</td>
<td>shift</td>
</tr>
<tr>
<td>int * int</td>
<td>$</td>
<td>3</td>
</tr>
<tr>
<td>int * T</td>
<td>$</td>
<td>4</td>
</tr>
<tr>
<td>T</td>
<td>$</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>
Notes

• Rerunning the automaton at each step is wasteful
  - Most of the work is repeated

- Put another way, at each iteration of the algorithm, the change occurs at the top of the stack, but each iteration we repeat through the DFA, which is unnecessary
An Improvement

• Remember the state of the automaton on each prefix of the stack

• Change stack to contain pairs (before it just contained symbols)

〈 Symbol, DFA State 〉
An Improvement (Cont.)

- For a stack

  \[ \langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle \]

  \text{state}_n \text{ is the final state of the DFA on } \text{sym}_1 \ldots \text{sym}_n

- Detail: The bottom of the stack is \( \langle \text{any}, \text{start} \rangle \)

  where

  - \text{any} is any dummy symbol (doesn’t matter what)
  - \text{start} is the start state of the DFA
Goto Table

• Define $\text{goto}[i,A] = j$ if $\text{state}_i \rightarrow^A \text{state}_j$

• $\text{goto}$ is just the transition function of the DFA (written out as an array)
  - One of two parsing tables (other shown a few slides further on)
Refined Parser Has 4 Possible Moves

- **Shift** $x$
  - Push $\langle a, x \rangle$ on the stack
  - $a$ is current input
  - $x$ is a DFA state

- **Reduce** $X \rightarrow \alpha$
  - As before (pop an element (from the rhs) off the stack and push an element (lhs) onto the stack)

- **Accept**

- **Error**
Action Table (the second table)

For each state $s_i$ and terminal $a$

- If $s_i$ has item $X \rightarrow \alpha.a\beta$ and $\text{goto}[i,a] = j$ then action$[i,a] = \text{shift} \; j$

- If $s_i$ has item $X \rightarrow \alpha.$ and $a \in \text{Follow}(X)$ and $X \neq S'$ then action$[i,a] = \text{reduce} \; X \rightarrow \alpha$

- If $s_i$ has item $S' \rightarrow S.$ then action$[i,\$] = \text{accept}$

- Otherwise, action$[i,a] = \text{error}$

Tells us which kind of move to make in every possible state
Indexed by the state of the automaton and next input symbol
SLR Parsing Algorithm

Let I = w$ be initial input
Let j = 0
Let DFA state 1 have item S’ → .S
Let stack = 〈 dummy, 1 〉

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
Notes on SLR Parsing Algorithm

• Note that the algorithm uses only the DFA states and the input
  - The stack symbols are never used!

• However, we still need the symbols for semantic actions
  - Throwing away the stack symbols amounts to throwing away the program, which we’ll obviously need for later stages of compilation!
More Notes

• Some common constructs are not SLR(1)

• LR(1) is more powerful
  - Build lookahead into the items
  - An LR(1) item is a pair: LR(0) item x lookahead
  - \([T \rightarrow \cdot \text{ int } * T, \$]\) means
    - After seeing \(T \rightarrow \text{ int } * T\) reduce if lookahead is \($\)
  - More accurate than just using follow sets
  - Take a look at the LR(1) automaton for your parser!
    - Actually, your parser is an LALR(1) automaton
    - Bit of an optimization over LR(1)

Could be \(k\) lookahead and thus \(k\) items here (and item need not be \($\))

Main difference between SLR and LR