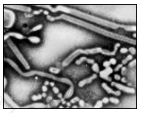
COMP 181

Lecture 9

LR parsing

October 3, 2006

Prelude



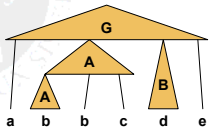
- What is this?
  Micrograph of influenza A
- Where does influenza A come from?
  - Birds – aka "Avian Flu"
  - **BUT**, it's extremely rare for humans to be infected
- So, how do people get influenza A
  - Pigs can get both avian and human flu strains
  - Virus recombines in pigs

Bottom-up parsing

- Start with input stream
  - "Leaves" of parse tree
- Build up towards goal symbol
  - Construct the reverse derivation

#	Production rule
1	G → a A B e
2	A → A b c
3	| b
4	B → d

Rule	Sentential form
-	abbcde
3	aAbcde
2	aAde
4	aABe
1	G



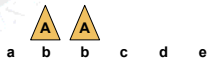
Bottom-up parsing

- **Problem**:
  - Not good enough to simply find production right-hand sides and reduce
  - *Example*:

Rule	Sentential form
-	abbcde
3	aAbcde
2	aAAcde
?	…now what?

#	Production rule
1	G → a A B e
2	A → A b c
3	| b
4	B → d

  - *"**aAAcde**" is not part of any sentential form*



LR parsing

- State of the parser:

  α | γ

  - α is a stack of terminals and non-terminals
  - γ is string of unexamined terminals
- Two operations:
  - *Shift* – read next terminal, push on stack

    E + (| int) → E + (int |)
  - *Reduce* – pop RHS symbols off stack, push LHS

    E + (E + (E) |) → E + (E |)

#	Production rule
1	E → E + (E)
2	| int

LR parsing

repeat
  if top symbols on stack match β for some A → β
    Reduce: *"found an A"*
      Pop those symbols off
      Push A on stack
  else Get next token from scanner
    if token is useful
      Shift: *"still working on something"*
        Push on stack
  else **error**
until stack contains goal **and** no more input

Example

1. | int + (int) + (int) — Nothing on stack, get next token

Stack

Example

1. | int + (int) + (int) — Nothing on stack, get next token
2. int | + (int) + (int) — *Shift*: push int

Top of stack matches E → int

Stack int

Example

1. | int + (int) + (int) — Nothing on stack, get next token
2. int | + (int) + (int) — *Shift*: push int
3. int | + (int) + (int) — *Reduce*: pop int

Top of stack matches E → int

Stack

int

Example

1. | int + (int) + (int) — Nothing on stack, get next token
2. int | + (int) + (int) — *Shift*: push int
3. int | + (int) + (int) — *Reduce*: pop int, push E

Top of stack matches E → int

Stack E

int

Example

1. | int + (int) + (int) — Nothing on stack, get next token
2. int | + (int) + (int) — *Shift*: push int
3. int | + (int) + (int) — *Reduce*: pop int, push E
4. int + | (int) + (int) — *Shift*: push +
5. int + (| int) + (int) — *Shift*: push (
6. int + (int |) + (int) — *Shift*: push int

Top of stack matches E → int

Stack E + (int

Example

1. | int + (int) + (int) — Nothing on stack, get next token
2. int | + (int) + (int) — *Shift*: push int
3. int | + (int) + (int) — *Reduce*: pop int, push E
4. int + | (int) + (int) — *Shift*: push +
5. int + (| int) + (int) — *Shift*: push (
6. int + (int |) + (int) — *Shift*: push int
7. int + (int |) + (int) — *Reduce*: pop int, push E

Stack E + (E

Example

1. | int + (int) + (int) — Nothing on stack, get next token
2. int | + (int) + (int) — *Shift*: push int
3. int | + (int) + (int) — *Reduce*: pop int, push E
4. int + | (int) + (int) — *Shift*: push +
5. int + (| int) + (int) — *Shift*: push (
6. int + (int |) + (int) — *Shift*: push int
7. int + (int |) + (int) — *Reduce*: pop int, push E
8. int + (int) | + (int) — *Shift*: push)

Top of stack matches E --> E + (int)

Stack E + (E)

Example

1. | int + (int) + (int) — Nothing on stack, get next token
2. int | + (int) + (int) — *Shift*: push int
3. int | + (int) + (int) — *Reduce*: pop int, push E
4. int + | (int) + (int) — *Shift*: push +
5. int + (| int) + (int) — *Shift*: push (
6. int + (int |) + (int) — *Shift*: push int
7. int + (int |) + (int) — *Reduce*: pop int, push E
8. int + (int) | + (int) — *Shift*: push)
9. int + (int) | + (int) — *Reduce*: pop x 5, push E

Stack E

Example

....

9. int + (int) | + (int) — *Reduce*: pop x 5, push E
10. int + (int) + | (int) — *Shift*: push +
11. int + (int) + (| int) — *Shift*: push (
12. int + (int) + (int |) — *Shift*: push int

Stack E + (int

Example

....

9. int + (int) | + (int) — *Reduce*: pop x 5, push E
10. int + (int) + | (int) — *Shift*: push +
11. int + (int) + (| int) — *Shift*: push (
12. int + (int) + (int |) — *Shift*: push int
13. int + (int) + (int |) — *Reduce*: pop int, push E
14. int + (int) + (int) | — *Shift*: push)

Stack E + (E)

Example

....

9. int + (int) | + (int) — *Reduce*: pop x 5, push E
10. int + (int) + | (int) — *Shift*: push +
11. int + (int) + (| int) — *Shift*: push (
12. int + (int) + (int |) — *Shift*: push int
13. int + (int) + (int |) — *Reduce*: pop int, push E
14. int + (int) + (int) | — *Shift*: push)
15. int + (int) + (int) | — *Reduce*: pop x 5, push E

DONE!

Stack E

Key problems

- (1) Will this work?
  How do we know that shifting and reducing using a stack is sufficient to compute the reverse derivation?
- (2) How do we know when to shift and reduce?
  - Can we efficiently match top symbols on the stack against productions?
    - *Right-hand sides of productions may have parts in common*
  - Will shifting a token move us closer to a reduction?
    - *Are we making progress?*
    - *How do we know when an error occurs?*

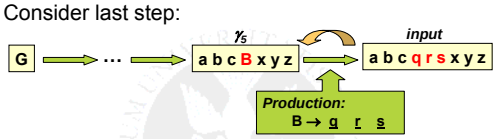
Key problems

- (1) Will this always work?
  - Yes, for unambiguous grammars
- Why?
  - Unambiguous:
    - Unique right-most derivation for every string
    - At each parsing step, one possible reduction

$$G \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \rightarrow \gamma_5 \rightarrow \text{input}$$

Shift-reduce parsing

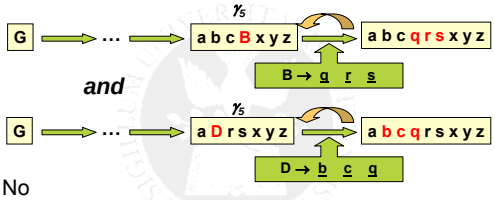
- Consider last step:



- To reverse this step:
  - **Shift** until **q**, **r**, **s** on top of stack
  - **Reduce**: pop **q**, **r**, **s**, push **B**
  - Parsing state: *Input:* a b c q r s | x y z; *Stack:* a b c B

Right-most derivation

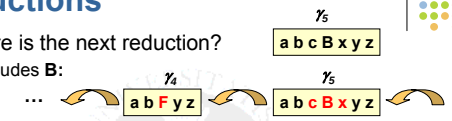
- Could there be an alternative reduction?



- No
  - Two right-most derivations for the same string
  - I.e., the grammar would be ambiguous

Reductions

- Where is the next reduction?
  - Includes **B:**



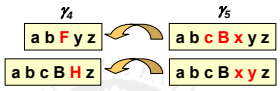
  - Later in the input stream:

  - Could it be earlier?

    - No – this is not the right-most derivation!

Implications

- Cases:



- Parsing state: *Input:* a b c q r s | x y z; *Stack:* a b c B
- *Key*: next reduction must consume top of stack
  Possibly after shifting more terminal symbols
- How does this help?
  - Can consume terminal symbols in order
  - Never need to search inside the stack

We can perform LR parsing using only stack operations

Key problems

- (2) How do we know when to shift or reduce?
  - Reductions
    - Good news:
      - At any given step, reduction is unique
      - Matching production occurs at top of stack
    - *Problem*:
      - How to efficiently find the right production
  - Shifts
    - Default behavior: shift when there's no reduction
    - Still need to handle errors

When to shift or reduce

- What is on the stack?
  Either:
  - A right sentential form
  - A prefix of a right sentential form (missing some terminals)
    Called a ***viable prefix***

Input: a b c q r s | x y z
Stack: a b c B

- **Idea**: a DFA that recognizes viable prefixes
  - Input: stack contents (a mix of terminals, non-terminals)
  - Each state represents either
    - A right sentential form – labeled with the reduction to apply
    - A viable prefix – labeled with tokens to expect next

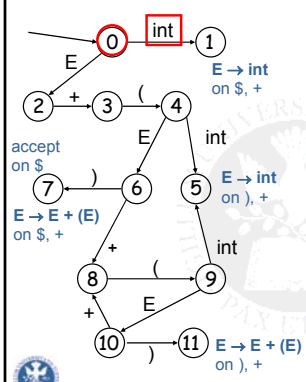
Shift/reduce DFA

- Using the DFA
  - At each parsing step run DFA on stack contents
  - Examine the resulting state X and the token t immediately following | in the input stream
    - If X has an outgoing edge labeled t, then **shift**
    - if X is labeled "**A → β on t**", then **reduce**
- Example:

#	Production rule
1	E → E + (E)
2	| int

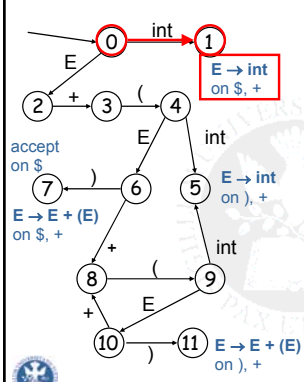
  - First, we'll look at how to use such a DFA…

Example



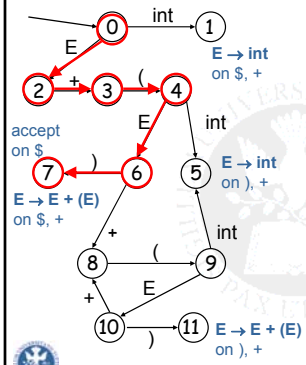
| int + (int) + (int)$

Example



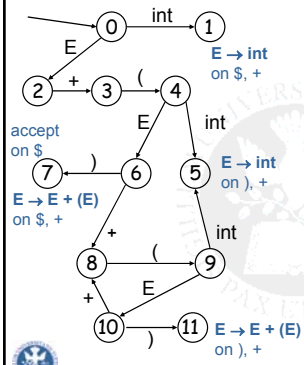
| int + (int) + (int)$ shift
int | + (int) + (int)$

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)$

Example

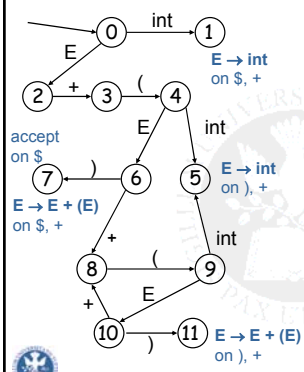


| int + (int) + (int)$ shift
int | + (int) + (int)$ E --> int
E | + (int) + (int)$ shift(x3)
E + (int |) + (int)$ E --> int
E + (E |) + (int)$ shift
E + (E) | + (int)$ E --> E+(E)
E | + (int)$ shift (x3)
E + (int |)$ E --> int
E + (E |)$ shift
E + (E) | $ E --> E+(E)
E | $ accept

Improvements

- Each DFA state represents stack contents
  - At each step, we rerun the DFA to compute the new state
  - Can we avoid this?
  - Two actions:
    - Shift: Push a new token
    - Reduce: Pop some symbols off, push a new symbol
- **Idea**:
  - For each symbol on the stack, remember the DFA state that represents the contents up to that point
    - Push a new token = go forward in DFA
    - Pop a sequence of symbols = "unwind" DFA to previous state

Example



I int + (int) + (int)$ shift
int I + (int) + (int)$ E --> int
E I + (int) + (int)$ shift(x3)
At state 2
go forward in DFA
2-->3-->4-->5
E + (int I) + (int)$ E --> int
Back up to state 4
Go forward with E
4-->6

Algorithm components

- Stack
  - String of the form : $\langle sym_1, state_1 \rangle \ldots \langle sym_n, state_n \rangle$
  - sym_i : grammar symbol (left part of string)
  - $state_i$: DFA state
    - Intuitively: represents what we've seen so far
    - $state_k$ is the final state of the DFA on $sym_1 \ldots sym_k$
    - And, captures what we're looking for next
- Represent as two tables:
  - *action* – whether to shift, reduce, accept, error
  - *goto* – next state

Tables

- **Action**

  Given state and the next token, ***action****[s_i,a] =*
  - **Shift s'**, where s' is a state
  - **Reduce** by a grammar production **A → β**
  - **Accept**
  - **Error**
- **Goto**

  Given a state and a grammar symbol, ***goto****[s_i,X] =*
  - Transition to next state (after recognizing an X)

Algorithm

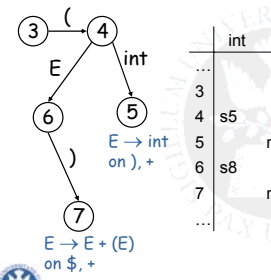
```
push s0 on stack
token = scanner.next_token()
repeat
   s = state at top of stack
   if action[s, token] = reduce A → β then
      pop | β |  pairs (X_i, s_m) off the stack
      s' = top of stack
      push A on stack
      push goto[s', A] on stack
   else if action[s, token] = shift s' then
      push token on stack
      push s on stack
      token = scanner.next_token()
   else if action(s, token) = accept  then
      return true
   else error()
```

Top of stack is handle A → β

- Work
  - Shift each token
  - Pop each token
- Errors
  - Input exhausted
  - Error entry in table

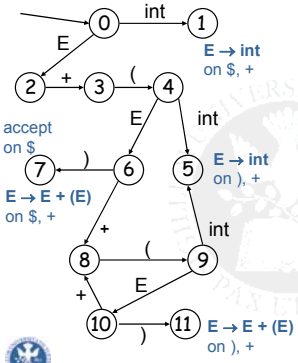
Representing the DFA

- Combined table:



	action(state, token)					*goto*
	int	+	(	)	$	E
…						
3			s4			
4	s5					g6
5		$r_{E\text{-->int}}$		$r_{E\text{-->int}}$		
6	s8		s7			
7		$r_{E\text{-->E+(E)}}$			$r_{E\text{-->E+(E)}}$	
…						

Example



push <G,0>
I int + (int) + (int)$ shift
push <int,1>
int I + (int) + (int)$ E --> int
pop <int, 1>
push <E, goto(0, E)=2>
E I + (int) + (int)$ shift(x3)
push <+,3>, <(,4>, <int,5>
E + (int I) + (int)$ E --> int
pop <int,4>
push <E, goto(4, E) = 6>
etc….

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 represents several such contexts
  - E.g., when we are looking for non-terminal **E**, we might be looking either for an **int** or a **E + (E)** on the RHS
  - We want to capture all the productions we ***could be*** working on

LR Items

- An ***LR(1) item*** is a pair:

  [A → α • β, a]
  - **A → αβ** is a production
  - a is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal
- **[A → α • β, a]** describes a context of the parser
  - We are trying to find an **A** followed by an a, and
  - We have **α** on top of the stack
  - We need to see a prefix derived from **β a**

LR Items

- In context containing

  [E → E + • (E), +]
  - If **(** next then we can a ***shift*** to context containing

  [E → E + (• E), +]
- In context containing

  [E → E + (E) •, +]
  - We can ***reduce*** with **E → E + (E)**
  - But only if a **+** follows

LR Items

- Consider the item

  E → E + (• E) , +
  - What could we see next?
  - We expect a string derived from **E) +**
  - There are two productions for E

  E → int *and* **E → E + (E)**
- We extend the context with two more items:

  E → • int,)

  E → • E + (E) ,)

Closure operation

- The operation of extending the context with items is called the ***closure*** operation
- Closure function:
  - Given a set of items
  - Compute all other items that could represent the current parsing state
- Observation:
  - At **A → α•Bβ** we expect to see **Bβ** next
  - If **B → γ** is a production, then we could also see a **γ**

Closure operation

- ***Algorithm:***

closure(Items) =
  repeat
    for each [A → α • Bβ, a] in Items
      for each production **B** → γ

        add [B → • γ, ?] to Items
  until Items is unchanged

What is the lookahead?

Closure operation

- **Algorithm:**

closure(Items) =
  repeat
    for each [A → α • Bβ, a] in Items
      for each production **B** → γ
        for each **b** ∈ **FIRST**(βa)
          add [B → • γ, b] to Items
  until Items is unchanged

Building the DFA – part 1

- Starting context = ***closure***({S → • E, $})

  S → • E, $
  E → • E+(E), $
  E → • int, $
  E → • E+(E), +
  E → • int, +

- Abbreviated:

  S → • E, $
  E → • E+(E), $/+
  E → • int, $/+

Building the DFA – part 2

- DFA states
  - Each DFA state is a closed set of LR(1) items
  - Start state: **closure**({S → • E, $})
- Reductions
  - Label each item [A → αβ •, x] with "Reduce with **A** → αβ on **x**"
- What about transitions?

DFA transitions

- Idea:
  - If the parser was in state [A → α•Xβ] and then recognized an instance of X, then the new state is [A → αX•β]
  - *Note*: X could be a terminal or non-terminal
- **Algorithm**:

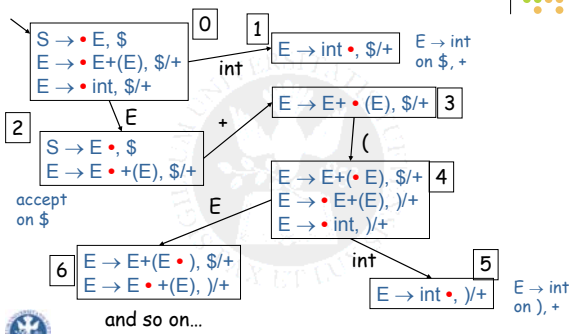
  transition(I, X) =
    J = {}
    for each [A → α•Xβ, b] ∈ I
      add [A → αX•β , b] to J
    return ***closure***(J)

DFA construction

- Data structure:
  - **T** – set of states (each state is a set of items)
  - **E** – edges of the form I $\xrightarrow{X}$ J
    where I, J ∈ T and X is a terminal or non-terminal
- **Algorithm**:

  T = {**closure**({S → • E, $}, E = {}
  repeat
    for each state I in T
      for each item [A → α•Xβ, b] ∈ I
        let J = ***transition***(I, **X**)
        T = T + J
        E = E + { I $\xrightarrow{X}$ J }
  until E and T no longer change

Example DFA



To form into tables

- Two tables
  - action(I, token)
  - goto(I, symbol)
- Layout:
  - One row for each states – each I in T
  - One columns for each symbol
- Entries:
  - For each edge **I $\xrightarrow{x}$ J**
    - If X is a terminal, add shift J at position (I, X) in action
    - if X is a non-terminal, add goto J at position (I, X) goto
  - For each state **[A → αβ •, x]** in I
    - Add reduce n at position (I, x) in action (where n is |rhs|)

Next time…

- Some issues with bottom-up parsing
- Parser-generator tools
- New programming assignment