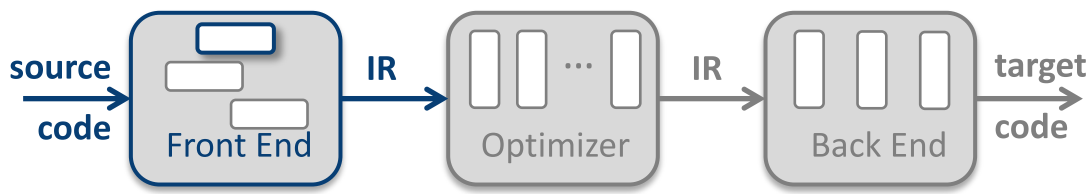
COMP 412
FALL 2018

Midterm Exam: Thursday October 18, 7PM Herzstein Amphitheater

Syntax Analysis, V

Bottom-up Parsing & The Magic of Handles

Comp 412



Chapter 3 in EaC2e

Lab 2 Schedule

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
9	10	11	12	13	14 Lab 2 Specs available	15
16	17	18 Tutorial 5 PM McMurtr	19	20	21 Deadline: Code Check 1	22
23	24 Tutorial 5 PM McMurtry	25 Dan Grove Talk (Dart Group)	26	27	28	29
30	1 Deadline: Code Check 2	2	3 Advice: Pay attention to the timing / scaling tests	4	5	6
7	8 Deadline: Lab 2 Code	9	10	12 Deadline: Lab Report	13	14

Focus on connecting to Lab 1 IR, and Rename

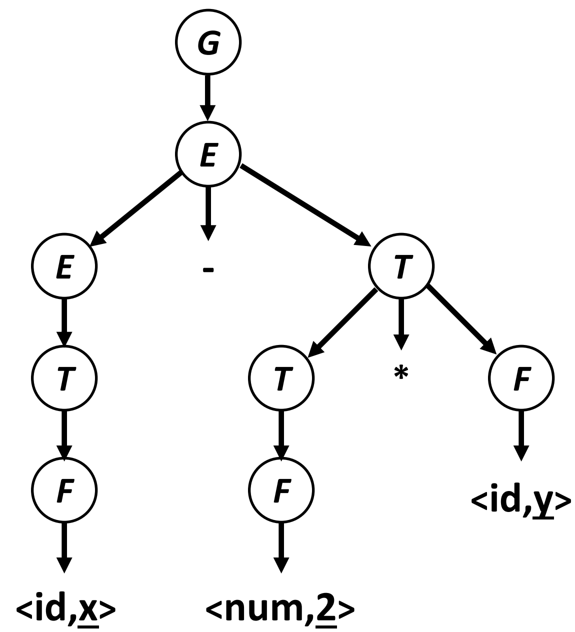
Allocate correctly

Improve performance & allocation

Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free



Parse tree for **x - 2 * y**

Bottom-up parsers *(LR(1), operator precedence)*

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- We can make the process deterministic

Bottom-up parsers can recognize a strictly larger class of *grammars* than can top-down parsers. See exercise 3.12 in EaC2e.

Bottom-up Parsing (definitions)

The point of parsing is to construct a *derivation*

A derivation consists of a series of rewrite steps

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

- Each γ_i is a sentential form
  - If γ contains only terminal symbols, γ is a **sentence** in *L(G)*
  - If γ contains 1 or more non-terminals, γ is a **sentential form**
- To get γ_i from γ_{i-1}, expand some NT $A \in \gamma_{i-1}$ by using $A \rightarrow \beta$
  - Replace the occurrence of $A \in \gamma_{i-1}$ with β to get γ_i
  - In a leftmost derivation, it would be the first NT $A \in \gamma_{i-1}$

A ***left-sentential form*** occurs in a <u>*leftmost*</u> derivation

A ***right-sentential form*** occurs in a <u>*rightmost*</u> derivation

Bottom-up, LR(1) parsers build a rightmost derivation in reverse

Bottom-up Parsing (definitions)

A bottom-up parser builds a derivation by working from the input sentence *back* toward the start symbol *S*

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

$\xleftarrow{\hspace{6em}}$ bottom-up

To **reduce** γ_i to γ_{i-1} match some *rhs* β against γ_i then replace β with its corresponding *lhs, A* (*assuming the reduction is* $A \rightarrow \beta$)

In terms of the parse tree, it works from leaves to root

- Nodes with no parent in a partial tree form its *upper fringe*
- Since each replacement of β with A shrinks the upper fringe, we call it a **reduction**.
- "Rightmost derivation in reverse" processes words ***left to right***

The parse tree need not be built, it can be simulated

$$|\textit{parse tree nodes}| = |\textit{terminal symbols}| + |\textit{reductions}|$$

"Shrinks the upper fringe" implies that the terminals are all instantiated, at least implicitly.

Finding Reductions

Consider the grammar

0	*Goal*	→	a *A* *B* e
1	*A*	→	*A* b c
2		|	b
3	*B*	→	d

And the input string abbcde

Sentential Form
abbcde
a *A* bcde
a *A* de
a *A* *B* e
Goal

derivation

- The trick is scanning the input and finding the next reduction.
- The mechanism for doing this must be efficient.

* The reductions are obvious from the derivation. Of course, building the derivation is not a practical way to find it.

$a\ b^+\ (bc)^*\ de$

Finding Reductions

Consider the grammar

0	*Goal*	→	a *A* *B* e
1	*A*	→	*A* b c
2		|	b
3	*B*	→	d

And the input string abbcde

parse

Sentential Form	*Next Reduction Prod'n*	*Pos'n*
abbcde	2	2
a *A* bcde	1	4
a *A* de	3	3
a *A* *B* e	0	4
Goal	—	—

- The trick is scanning the input and finding the next reduction
- The mechanism for doing this must be efficient

"Position" specifies where the right end of β occurs in the current sentential form.

While the process of finding the next reduction appears to be almost oracular, it can be automated in an efficient way for a large class of grammars.

Finding Reductions (Handles)

At each step, the parser needs to find a substring β of the tree's upper frontier that *derives from an expansion by A $\rightarrow \beta$ in the previous step in the rightmost derivation*

Informally, we call this substring β a **handle**

By convention, we will use ***k*** to mark the **right end** of the handle

Formally,

A **handle** of a right-sentential form γ is a pair $<A \rightarrow \beta, k>$ where $A \rightarrow \beta \in P$ and k is the position in γ of β's rightmost symbol.

If $<A \rightarrow \beta, k>$ is a handle, then replacing β at k with A produces the right sentential form from which γ is derived in the rightmost derivation.

Because γ is a right-sentential form, the substring to the right of a handle contains **only terminal symbols**

$\Rightarrow$ the parser doesn't need to scan (*much*) past the handle

*Handles are the **most mystifying** aspect of bottom-up, shift-reduce parsers. It usually takes a couple lectures to comprehend.*

*Assume, **WLOG**, that we can find handles easily …*

Using Handles: a Bottom-up Parser

As with top-down parsers, we will use a stack to hold the fringe of the partially completed parse tree. In this case, it is the *upper fringe*.

A simple shift-reduce parser:

```
push INVALID
word ← NextWord( )
repeat until (top of stack = Goal and word = EOF)
   if the top of the stack forms a handle A→β then
      // reduce β to A
      pop |β| symbols off the stack
      push A onto the stack
   else if (word ≠ EOF) then     // shift
      push word
      word ← NextWord( )
   else                // need to shift, but out of input
      report an error
report success   // accept
```

What happens on an error?

- Parser fails to find a handle
- Thus, it keeps shifting
- Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.

To fix this issue, the parser must recognize the failure to find a handle earlier.

To make shift-reduce parsers practical, we need good error localization in the handle-finding process.

Example

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	$\rightarrow$	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	$\rightarrow$	<u>(</u> *Expr* <u>)</u>
8		|	<u>number</u>
9		|	<u>id</u>

The Left-recursive, Left-associative Classic Expression Grammar

Bottom-up parsers work with either left-recursive or right-recursive grammars.

The examples will use the obvious left-recursive, left-associative form of the classic expression grammar.

I prefer the obvious left-recursive grammar because its associativity matches the standard rules that we were all taught as children.

Example

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	$\rightarrow$	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	$\rightarrow$	(*Expr*)
8		|	number
9		|	id

The Left-recursive, Left-associative Classic Expression Grammar

derivation

Prod'n	*Sentential Form*
—	*Goal*
0	*Expr*
2	*Expr* - *Term*
4	*Expr* - *Term* * *Factor*
9	*Expr* - *Term* * <id,y>
6	*Expr* - *Factor* * <id,y>
8	*Expr* - <num,2> * <id,y>
3	*Term* - <num,2> * <id,y>
6	*Factor* - <num,2> * <id,y>
9	<id,x> - <num,2> * <id,y>

Rightmost derivation of **x – 2 * y**

Example

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

The Left-recursive, Left-associative Classic Expression Grammar

Prod’n	Sentential Form
—	*Goal*
0	*Expr*
2	*Expr* - *Term*
4	*Expr* - *Term* * *Factor*
9	*Expr* - *Term* * <id,y>
6	*Expr* - *Factor* * <id,y>
8	*Expr* - <num,2> * <id,y>
3	*Term* - <num,2> * <id,y>
6	*Factor* - <num,2> * <id,y>
9	<id,x> - <num,2> * <id,y>

parse

Rightmost derivation of x – 2 * y

Example

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	$\rightarrow$	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	$\rightarrow$	(*Expr*)
8		|	number
9		|	id

The Left-recursive, Left-associative Classic Expression Grammar

Prod'n	*Sentential Form*
—	*Goal*
0	*Expr*
2	*Expr - Term*
4	*Expr* - *Term * Factor*
9	*Expr* - *Term* * <id,y>
6	*Expr* - *Factor* * <id,y>
8	*Expr* - <num,2> * <id,y>
3	*Term* - <num,2> * <id,y>
6	*Factor* - <num,2> * <id,y>
9	<id,x> - <num,2> * <id,y>

parse

Handles for rightmost derivation of x – 2 * y

Back to x - 2 * y

Stack	Input	Handle	Action
$	id - num * id	*none*	*shift*
$ id	- num * id		

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

By convention, $ represents **INVALID**

1. Shift until the top of the stack is the right end of a handle

2. Find the left end of the handle and reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id - num * id	*none*	*shift*
$ id	- num * id	9,1	*reduce 9*
$ *Factor*	- num * id	6,1	*reduce 6*
$ *Term*	- num * id	3,1	*reduce 3*
$ *Expr*	- num * id		

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

1. Shift until the top of the stack is the right end of a handle

2. Find the left end of the handle and reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id - num * id	*none*	*shift*
$ id	- num * id	9,1	*reduce 9*
$ *Factor*	- num * id	6,1	*reduce 6*
$ *Term*	- num * id	3,1	*reduce 3*
$ *Expr*	- num * id		

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

Expr is not a handle at this point because it does not occur at this point in the derivation.

While that statement sounds like **oracular mysticism**, we will see that the decision can be automated efficiently.

1. Shift until the top of the stack is the right end of a handle

2. Find the left end of the handle and reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id - num * id	*none*	*shift*
$ id	- num * id	9,1	*reduce 9*
$ *Factor*	- num * id	6,1	*reduce 6*
$ *Term*	- num * id	3,1	*reduce 3*
$ *Expr*	- num * id	*none*	*shift*
$ *Expr* -	num * id	*none*	*shift*
$ *Expr* - num	* id		

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

1. Shift until the top of the stack is the right end of a handle

2. Find the left end of the handle and reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id - num * id	*none*	*shift*
$ id	- num * id	9,1	*reduce 9*
$ *Factor*	- num * id	6,1	*reduce 6*
$ *Term*	- num * id	3,1	*reduce 3*
$ *Expr*	- num * id	*none*	*shift*
$ *Expr* -	num * id	*none*	*shift*
$ *Expr* - num	* id	8,3	*reduce 8*
$ *Expr* - *Factor*	* id	6,3	*reduce 6*
$ *Expr* - *Term*	* id		

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

1. Shift until the top of the stack is the right end of a handle

2. Find the left end of the handle and reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id - num * id	*none*	*shift*
$ id	- num * id	9,1	*reduce 9*
$ *Factor*	- num * id	6,1	*reduce 6*
$ *Term*	- num * id	3,1	*reduce 3*
$ *Expr*	- num * id	*none*	*shift*
$ *Expr* -	num * id	*none*	*shift*
$ *Expr* - num	* id	8,3	*reduce 8*
$ *Expr* - *Factor*	* id	6,3	*reduce 6*
$ *Expr* - *Term*	* id	*none*	*shift*
$ *Expr* - *Term* *	id	*none*	*shift*
$ *Expr* - *Term* * id			

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

1. Shift until the top of the stack is the right end of a handle

2. Find the left end of the handle and reduce

Back to x - 2 * y

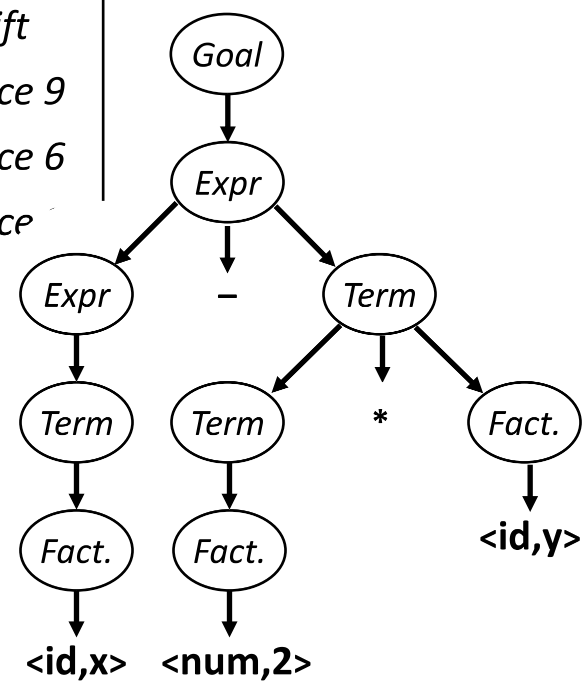
Stack	Input	Handle	Action
$	id - num * id	*none*	*shift*
$ id	- num * id	9,1	*reduce 9*
$ *Factor*	- num * id	6,1	*reduce 6*
$ *Term*	- num * id	3,1	*reduce 3*
$ *Expr*	- num * id	*none*	*shift*
$ *Expr* -	num * id	*none*	*shift*
$ *Expr* - num	* id	8,3	*reduce 8*
$ *Expr* - *Factor*	* id	6,3	*reduce 6*
$ *Expr* - *Term*	* id	*none*	*shift*
$ *Expr* - *Term* *	id	*none*	*shift*
$ *Expr* - *Term* * id		9,5	*reduce 9*
$ *Expr* - *Term* * *Factor*		4,5	*reduce 4*
$ *Expr* - *Term*		2,3	*reduce 2*
$ *Expr*		0,1	*reduce 0*
$ *Goal*		*none*	*accept*

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

5 shifts +
9 reduces +
1 accept

Back to x - 2 * y

Stack	Input	Handle	Action
$	id - num * id	*none*	*shift*
$ id	- num * id	9,1	*reduce 9*
$ *Factor*	- num * id	6,1	*reduce 6*
$ *Term*	- num * id	3,1	*reduce*
$ *Expr*	- num * id	*none*	*shi*
$ *Expr* -	num * id	*none*	*sh*
$ *Expr* - num	* id	8,3	*redu*
$ *Expr* - *Factor*	* id	6,3	*redu*
$ *Expr* - *Term*	* id	*none*	*sh*
$ *Expr* - *Term* *	id	*none*	*sh*
$ *Expr* - *Term* * id		9,5	*redu*
$ *Expr* - *Term* * *Factor*		4,5	*redu*
$ *Expr* - *Term*		2,3	*reduc*
$ *Expr*		0,1	*reduce 0*
$ *Goal*		*none*	*accept*



Corresponding Parse Tree

|Parse tree nodes| = |shifts| + |reduces|

Took | *Parse tree nodes* | + 1 steps

More on Handles

Shift reduce parsers find a rightmost derivation in reverse order

- Rightmost derivation ⇒ rightmost **NT** expanded at each step in the derivation
- Processed in reverse ⇒ parser proceeds left to right

These two statements are somewhat counter-intuitive

This slide, with no graphics and lots of text, is actually critical to understanding handles.

More on Handles

Shift-reduce parsers find a reverse rightmost derivation

- Process input left to right
  - Upper fringe of partially completed parse tree is *(NT | T)* T**
  - The handle always appears with its right end at the junction between *(NT | T)** and *T** (*the hot spot for LR parsing*)
  - We can keep the prefix of the upper fringe of the partially completed parse tree on a stack — that is, *(NT | T)**
- Handles appear at the top of the stack
  - Right end of handle is *always* at the top of the stack
  - The stack makes the position information irrelevant
- All the information for the decision is at the hot spot
  - The next word in the input stream
  - The rightmost **NT** on the fringe & its immediate left neighbors
  - An **LR** parser keeps additional information on the stack
    - → *the "state" of a handle-recognizing automaton*

Handles

Consider x – 2 * y with the expression grammar

derivation

Sentential Form				
Goal				
Expr				
Expr	—	*Term*		
Expr	—	*Term*	*	*Factor*
Expr	—	*Term*	*	<id,y>
Expr	—	*Factor*	*	<id,y>
Expr	—	<num,2>	*	<id,y>
Term	—	<num,2>	*	<id,y>
Factor	—	<num,2>	*	<id,y>
<id,x>	—	<num,2>	*	<id,y>

parse

Unambiguous grammar implies unique rightmost derivation

- At each step, we have one step that leads to x – 2 * y
- Any other choice leads to another distinct expression

A bottom-up parse reverses the rightmost derivation

- Each step has a unique reduction
- The key is finding the reduction at each step that leads to the derivation

Handles

Consider x – 2 * y with the expression grammar

Sentential Form				
Goal				
Expr				
Expr	—	*Term*		
Expr	—	*Term*	*	*Factor*
Expr	—	*Term*	*	<id,y>
Expr	—	*Factor*	*	<id,y>
Expr	—	<num,2>	*	<id,y>
Term	—	<num,2>	*	<id,y>
Factor	—	<num,2>	*	<id,y>
<id,x>	—	<num,2>	*	<id,y>

Now, look at the sentential forms in the example

- They have a specific form
- *NT* * (*NT* | *T*)* *T**
- Handles are found in the (*NT* | *T*)* portion
  - Track right end of region
  - Search left from there
  - Finite set of rhs strings

We know that each step has a unique reduction

That reduction is the **handle**

Handles

Consider x – 2 * y with the expression grammar

Sentential Form					*Reduction*
Goal					
Expr					*Goal* → *Expr*
Expr	—	*Term*			*Expr* → *Expr* — *Term*
Expr	—	*Term*	*	*Factor*	*Term* → *Term* * Factor
Expr	—	*Term*	*	<id,y>	*Factor* → id
Expr	—	*Factor*	*	<id,y>	*Term* → *Factor*
Expr	—	<num,2>	*	<id,y>	*Factor* → num
Term	—	<num,2>	*	<id,y>	*Expr* → *Term*
Factor	—	<num,2>	*	<id,y>	*Term* → *Factor*
<id,x>	—	<num,2>	*	<id,y>	*Factor* → id

Finding Reductions

We saw this slide earlier. It deserves a second take.

At each step, the parser needs to find a substring β of the tree's upper frontier that *derives from an expansion by A $\rightarrow \beta$ in the previous step in the rightmost derivation*

Informally, we call this substring β a **handle**

By convention, we will use ***k*** to mark the **right end** of the handle

Formally,

A **handle** of a right-sentential form γ is a pair $<A \rightarrow \beta, k>$ where $A \rightarrow \beta \in P$ and k is the position in γ of β's rightmost symbol.

If $<A \rightarrow \beta, k>$ is a handle, then replacing β at k with A produces the right sentential form from which γ is derived in the rightmost derivation.

Because γ is a right-sentential form, the substring to the right of a handle contains **only terminal symbols**

$\Rightarrow$ the parser doesn't need to scan (*much*) past the handle

*Handles are the **most mystifying** aspect of bottom-up, shift-reduce parsers. It usually takes a couple lectures to comprehend.*

*Assume, **WLOG**, that we can find handles easily …*

Handles Are Unique

Theorem:

*If G is unambiguous, then every right-sentential form has a **unique** handle.*

Recall: Right sentential form is a string that appears as one step in a rightmost derivation.

Sketch of Proof:

1 G is unambiguous $\Rightarrow$ rightmost derivation is unique

2 $\Rightarrow$ a unique production $A \rightarrow \beta$ applied to derive γ_i from γ_{i-1}

3 $\Rightarrow$ a unique position ***k*** at which $A \rightarrow \beta$ is applied

4 $\Rightarrow$ a unique handle $\langle A \rightarrow \beta, \boldsymbol{k} \rangle$

This all follows from the definitions

If we can find the handles, we can build a derivation!

The handle always appears with its right end at the stack top.

⇒ We can make the handles relative to the stack top, which makes the "position" implicit. Now the handle is just the "right" RHS at stack top.

From Handles to Parsers

The sentential forms in the derivation have the form:

$NT^* \quad (NT \mid T)^* \quad T^*$

- They form the upper fringe of partially completed parse tree
- The suffix consisting of *T** is, at each step, the unread input
  - *The first word in the trailing string of terminals is the current word or token*

The shift-reduce parser operates by:

- Keeping the portion NT^* $(NT \mid T)^*$ on a stack
  - Leftmost symbol at bottom of stack, rightmost at stack top
- Handles always appear with right end at the top of stack
  - *The border between NT* (NT | T)* and T * is the critical spot*
- Searching for handles from stack top to stack bottom
- If search fails, shift another terminal onto stack

All the info that the parser needs
to decide is at TOS

The Critical Lesson about Handles

Simply looking for right hand sides that match is not good enough

Critical Question: How can we know when we have found a handle without generating lots of different derivations?

- **Answer:** We use left context, encoded in the sentential form, left context encoded in a "parser state", and a lookahead — the next word in the input. (Formally, 1 word beyond the handle.)
- Parser states are derived by reachability analysis on grammar that resembles the subset construction from Chapter 2.
- The set of handles is *finite* ⇒ we can recognize handles with a **DFA**
  - — Invoke **DFA** recursively to find terminal symbols as "sub goals"
  - — Store **DFA** states (at recursive invocation) on the stack as "parser state"

The additional left context is precisely the reason that LR(1) grammars express a superset of the languages that can be expressed as LL(1) grammars

The Critical Lesson about Handles

Simply looking for right hand sides that match is not good enough

Critical Question: How can we know when we have found a handle without generating lots of different derivations?

- **Answer:** We use left context, encoded in the sentential form, left context encoded in a "parser state", and a lookahead — the next word in the input. (Formally, 1 word beyond the handle.)
- Parser states are derived by reachability analysis on grammar that resembles the subset construction from Chapter 2.
- The set of handles is *finite* ⇒ we can recognize handles with a **DFA**
  - — Invoke **DFA** recursively to find terminal symbols as "sub goals"
  - — Store **DFA** states (at recursive invocation) on the stack as "parser state"