Bottom-Up Parsing

More on Handles

Bottom-up parsers finds rightmost derivation

- Process input left to right
- Handle always appears at upper fringe of partially completed parse tree

More on Handles

- We can keep the prefix of the upper fringe of the partially completed parse tree on a **<u>stack</u>**
  - The stack makes the position information irrelevant
  - Handles appear at the top of the stack

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

More on Handles

- Handles appear at the top of the stack

Prod'n	Sentential Form	Handle
8	<id,x> - <num,2> * <id,y>	8,1
6	*Factor* - <num,2> * <id,y>	6,1
3	*Term* - <num,2> * <id,y>	3,1
7	*Expr* - <num,2> * <id,y>	7,3

Rest of input from scanner

* <id,y>

TOS →

stack
<num,2>
-
Expr

stack

7	*Factor*	→	number

Shift-reduce Parsing

To implement a bottom-up parser, we adopt the shift-reduce paradigm

A shift-reduce parser is a stack automaton with four actions

- *Shift* — next word is shifted onto the stack
- *Reduce* — right end of handle is at top of stack
  - Locate left end of handle within the stack
  - Pop handle off stack & push appropriate *lhs*
- *Accept* — stop parsing & report success
- *Error* — call an error reporting/recovery routine

Accept & *Error* are simple

Shift is just a push and a call to the scanner

Reduce takes |*rhs*| pops & 1 push

But how does parser know when to shift and when to reduce? It shifts until it has a handle at the top of the stack.

Bottom-up Parser

A simple *shift-reduce parser:*

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

What happens on an error?

- It 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.

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*	→	number
8		|	id
9		|	(*Expr*)

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	8,1	*reduce 8*

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

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	8,1	*reduce 8*
$ *Factor*	- 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*	→	number
8		|	id
9		|	(*Expr*)

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	8,1	*reduce 8*
$ *Factor*	- num * id	6,1	*reduce 6*

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

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	8,1	*reduce 8*
$ *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*	→	number
8		|	id
9		|	(*Expr*)

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	8,1	*reduce 8*
$ *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*	→	number
8		|	id
9		|	(*Expr*)

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	8,1	reduce 8
$ *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*	→	number
8		|	id
9		|	(*Expr*)

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	8,1	reduce 8
$ *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	7,3	reduce 7
$ *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*	→	number
8		|	id
9		|	(*Expr*)

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	8,1	reduce 8
$ *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	7,3	reduce 7
$ *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*	→	number
8		|	id
9		|	(*Expr*)

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	8,1	reduce 8
$ *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	7,3	reduce 7
$ *Expr* - *Factor*	* id	6,3	reduce 6
$ *Expr* - *Term*	* id	none	shift
$ *Expr* - *Term* *	id	none	shift
$ *Expr* - *Term* * id		8,5	reduce 8
$ *Expr* - *Term* * *Factor*		4,5	reduce 4
$ *Expr* - *Term*		2,3	reduce 2
$ *Expr*		0,1	reduce 0
$ *Goal*		none	accept

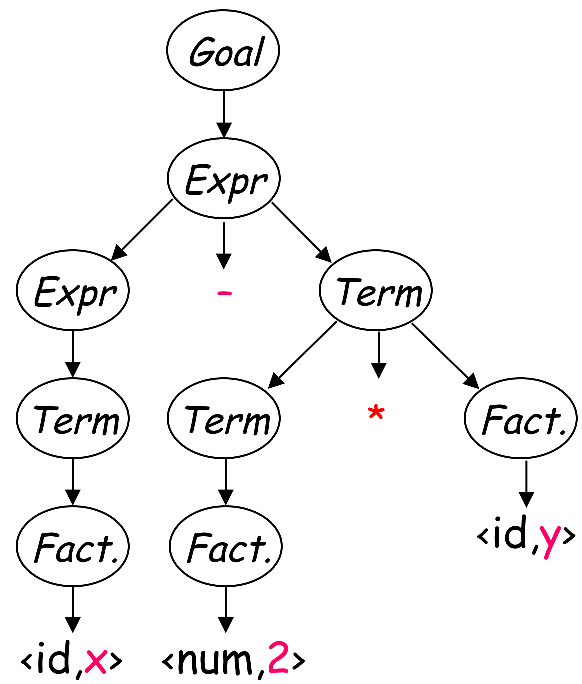
0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	number
8		|	id
9		|	(*Expr*)

5 shifts +
9 reduces +
1 accept

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	Action
$	id - num * id	shift
$ id	- num * id	reduce 8
$ Factor	- num * id	reduce 6
$ Term	- num * id	reduce 3
$ Expr	- num * id	shift
$ Expr -	num * id	shift
$ Expr - num	* id	reduce 7
$ Expr - Factor	* id	reduce 6
$ Expr - Term	* id	shift
$ Expr - Term *	id	shift
$ Expr - Term * id		reduce 8
$ Expr - Term * Factor		reduce 4
$ Expr - Term		reduce 2
$ Expr		reduce 0
$ Goal		accept



Corresponding Parse Tree

An Important Lesson about Handles

A handle must be a substring of a sentential form γ such that :

- Must match rhs β of some rule $A \rightarrow \beta$; and
- Must be some rightmost derivation from goal symbol that produces sentential form γ with $A \rightarrow \beta$ as last production applied

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

An Important Lesson about Handles

- 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 at the next word in the input. (Formally, 1 word beyond the handle.)
  - Parser states are derived by reachability analysis on grammar
  - We build all of this knowledge into a handle-recognizing DFA

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

LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- The class of grammars that these parsers recognize is called the set of LR(1) grammars

LR(1) means left-to-right scan of the input, rightmost derivation (in reverse), and 1 word of lookahead.

Informal definition:

A grammar is LR(1) if, given a rightmost derivation

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

We can

1. *isolate the handle of each right-sentential form* γ_i, and
2. *determine the production by which to reduce,*

by scanning γ_i from *left-to-right*, going at most *1* symbol beyond the right end of the handle of γ_i

Formally,

A *handle* of a right-sentential form γ is a pair $\langle A \rightarrow \beta, k\rangle$ where

$A \rightarrow \beta \in P$ and k is the position in γ of β's rightmost symbol.

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

Finding Reductions (Handles)

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