Bottom-Up Parsing
Part II

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
```

- Push "INVALID" on the stack. If we run out of input and have not reached "GOAL" then the input is invalid.
- Read input into "token"

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
```

We need "GOAL" at the top of the stack and "EOF" in the token to indicate there is no more input

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
```

Handle is "rhs" of a production rule.

If the top of stack is a handle perform a reduction

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
```

If handle not found then we "shift", i.e. read another token from the input

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		

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		

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*
$ *Term*	- 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*)

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		

Expr is not a handle at this point because reducing now will cause backtracking.

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

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		

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		

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		

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

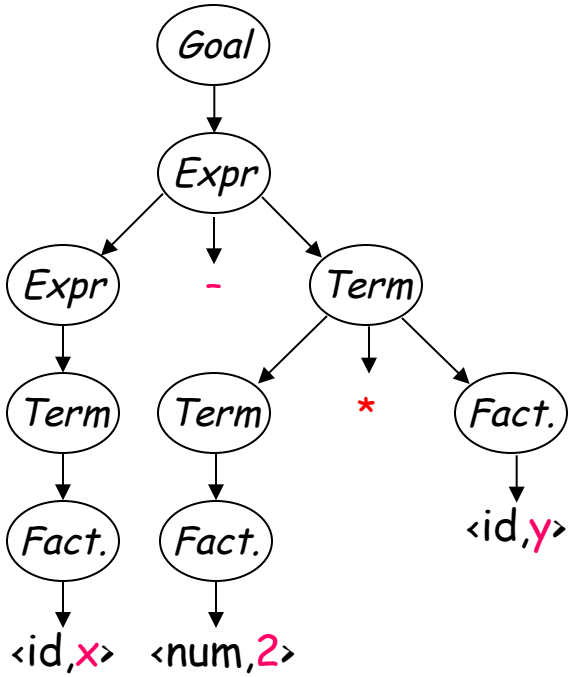
Stack	Input	Handle	Action
$	id - num * id	none	shift
$ id	- num * id	8,1	reduce 8
$ *Factor*	- num * id	6,1	reduce 6
$ *Term*	- num * id	3,1	reduce 3
$ *Expr*	- num * id	none	shift
$ *Expr* -	num * id	none	shift
$ *Expr* - num	* id	7,3	reduce 7
$ *Expr* - *Factor*	* id	6,3	reduce 6
$ *Expr* - *Term*	* 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

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

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.

LR(1) Parsers

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

Finding Reductions (Handles)

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