Bottom-Up Parsing

Parsing Techniques

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

- Start at root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad “pick” $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

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

- Start at the leaves and grow toward root
- As input consumed, encode possibilities in internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Bottom-up Parsing (definitions)

The point of parsing is to construct a <u>derivation</u>

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

Bottom-up Parsing (definitions)

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

- 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 first NT $A \in \gamma_{i-1}$

Bottom-up Parsing (definitions)

A *left-sentential form* occurs in a *leftmost* derivation

A *right-sentential form* occurs in a *rightmost* derivation

Bottom-up parsers build rightmost derivation in reverse

Bottom-up Parsing (definitions)

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

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

bottom-up

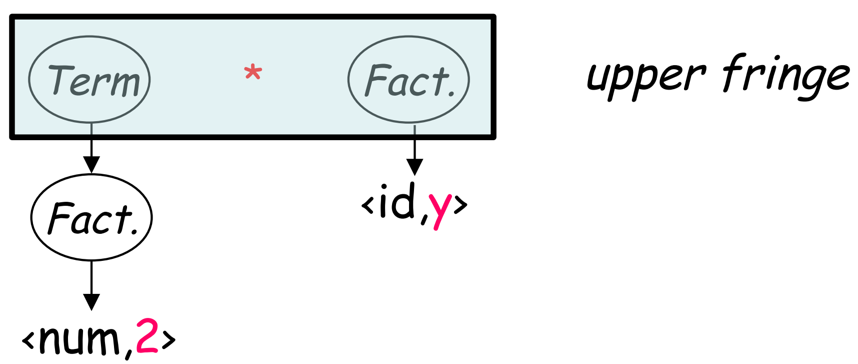
assuming $A \rightarrow \beta$, *match some rhs* β here

replace β with its corresponding *lhs*, *A here*

Bottom-up Parsing (definitions)

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

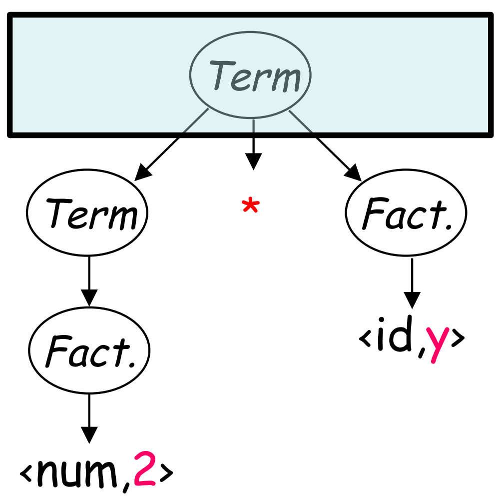
- Nodes with no parent in partial tree form *upper fringe*
- Each replacement of β with *A* shrinks the upper fringe, we call this a *reduction*.
- "Rightmost derivation in reverse" processes words *left to right*



Bottom-up Parsing (definitions)

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

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



reduction

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	Next Reduction Production	Next Reduction Position
abbcde	2	2
a *A* bcde		

"Position" specifies where the right end of β occurs in the current sentential form. We call this position k.

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	*Next Reduction Production*	*Next Reduction Position*
abbcde	2	2
a *A* bcde	1	4
a *A* de	3	3
a *A* *B* e	0	4
Goal	—	—

"Position" specifies where the right end of β occurs in the current sentential form. We call this position k.

Finding Reductions (Handles)

Parser must find substring β at parse tree's frontier that *matches some production* $A \to \beta$

($\Rightarrow \beta \to A$ is in Reverse Rightmost Derivation)

We call substring β a *handle*

Finding Reductions (Handles)

Formally,

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

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

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

On ChalkBoard 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>number</u>
8		|	<u>id</u>
9		|	<u>(</u> *Expr* <u>)</u>

A simple left-recursive form of the classic expression grammar

Bottom up parsers can handle either left-recursive or right-recursive grammars.

On ChalkBoard 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>number</u>
8		|	<u>id</u>
9		|	(*Expr*)

A simple left-recursive form of the classic expression grammar

Prod'n	*Sentential Form*	*Handle*
8	<id,<u>x</u>> - <num,<u>2</u>> * <id,<u>y</u>>	8,1
	Factor - <num,<u>2</u>> * <id,<u>y</u>>	

parse

Handles for rightmost derivation of <u>x</u> <u>-</u> <u>2</u> <u>*</u> <u>y</u>

On ChalkBoard Example

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*)

A simple left-recursive form of the classic expression grammar

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
6	*Expr* - *Factor* * <id,y>	6,3
8	*Expr* - *Term* * <id,y>	8,5
4	*Expr* - *Term* * *Factor*	4,5
2	*Expr* - *Term*	2,3
0	Expr	0,1
-	*Goal*	-

parse

Handles for rightmost derivation of x - 2 * y

Bottom-up Parsing (Abstract View)

A bottom-up parser repeatedly finds a handle $A \rightarrow \beta$ in current right-sentential form and replaces β with A.

To construct a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$$

Apply the following conceptual algorithm

for $i \leftarrow n$ *to 1 by −1*

Find the handle $\langle A_i \rightarrow \beta_i , k_i \rangle$ in γ_i

Replace β_i *with* A_i *to generate* γ_{i-1}

of course, *n* is unknown until the derivation is built

This takes *2n* steps

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

LR parsing

- Keep upper fringe of the partially completed parse tree on a **stack**
  - Stack makes position information irrelevant
  - Handles appear at top of the stack (TOS)

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

More on Handles

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

* <id,y>

Rest of input from scanner

7	*Factor*	→	number

K=3

stack
<num,2> ← TOS
-
Expr

stack

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
  - Located handle (rhs) on top of stack
  - Pop handle off stack & push appropriate *lhs*

Shift is just a push and a call to the scanner

Reduce means found a handle, 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.

Shift-Reduce Parsing

- *Accept* — stop parsing & report success
- *Error* — call an error reporting/recovery routine

Accept if no input and Goal symbol on top of stack (TOS)

Error otherwise

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.

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.

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.

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.

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.

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*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	$\rightarrow$	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	$\rightarrow$	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		

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 reducing now will cause backtracking.

While that statement sounds like oracular, 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		

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

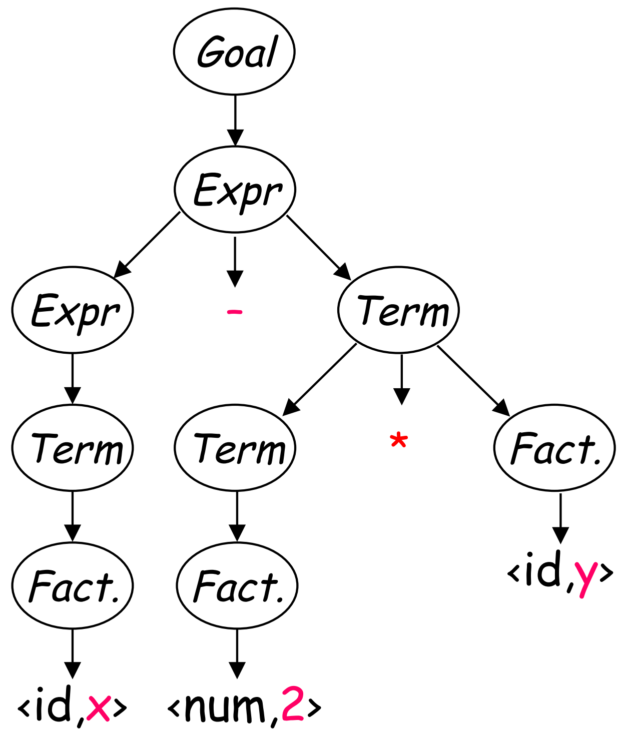
0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	number
8		|	id
9		|	(*Expr*)

5 shifts +
9 reduces +
1 accept

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

- 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?

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.)
  - 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.

Informal definition:

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

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

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