Parsing IV
Bottom-up Parsing

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.

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" $\Rightarrow$ 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 is consumed, encode possibilities in an 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 derivation

A derivation consists of a series of rewrite steps

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

- Each γ_i is a sentential form
  - If γ contains only terminal symbols, γ is a sentence in L(G)
  - If γ contains $\geq$ 1 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 leftmost derivation

A right-sentential form occurs in a rightmost derivation

Bottom-up Parsing

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

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

← 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 production $A \rightarrow \beta$)

In terms of the parse tree, this is working 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.

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

$$|\text{parse tree nodes}| = |\text{words}| + |\text{reductions}|$$

Finding Reductions

Consider the simple grammar

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

And the input string abbcde

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

The trick is scanning the input and finding the next reduction

The mechanism for doing this must be efficient

Finding Reductions (Handles)

The parser must find a substring β of the tree's frontier that matches some production $A \to \beta$ that occurs as one step in the rightmost derivation ($\Rightarrow \beta \to A$ is in RRD)

Informally, we call this substring β a handle

Formally,

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

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

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 past the handle (very far)

Finding Reductions (Handles)

Critical Insight (Theorem?)

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

If we can find those handles, we can build a 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 $\langle A \rightarrow \beta, k \rangle$

This all follows from the definitions

Example

(a very busy slide)

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

The expression grammar

Prod'n.	Sentential Form	Handle
—	Goal	—
1	Expr	1,1
3	Expr - Term	3,3
5	Expr - Term * Factor	5,5
9	Expr - Term * <id,y>	9,5
7	Expr - Factor * <id,y>	7,3
8	Expr - <num,2> * <id,y>	8,3
4	Term - <num,2> * <id,y>	4,1
7	Factor - <num,2> * <id,y>	7,1
9	<id,x> - <num,2> * <id,y>	9,1

Handles for rightmost derivation of x - 2 * y

This is the inverse of Figure 3.9 in EaC

Handle-pruning, Bottom-up Parsers

The process of discovering a handle & reducing it to the appropriate left-hand side is called handle pruning

Handle pruning forms the basis for a bottom-up parsing method

To construct a rightmost derivation

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

Apply the following simple algorithm

for $i \leftarrow n$ to 1 by -1

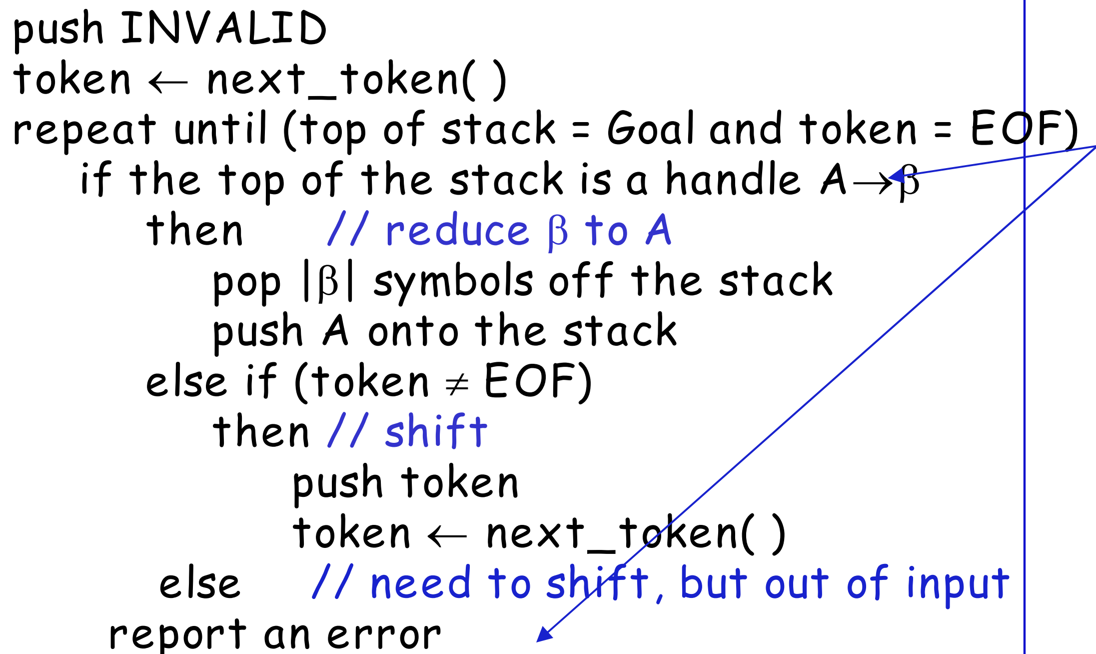
Find the handle $\langle A_i \rightarrow \beta_i , \mathbf{k}_i \rangle$ in γ_i

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

This takes 2n steps

Handle-pruning, Bottom-up Parsers

One implementation technique is the shift-reduce parser



How do errors show up?

- failure to find a handle
- hitting EOF & needing to shift (final else clause)

Either generates an error

Figure 3.7 in EAC

Back to x - 2 * y

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

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id		

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id			

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to x - 2 * y

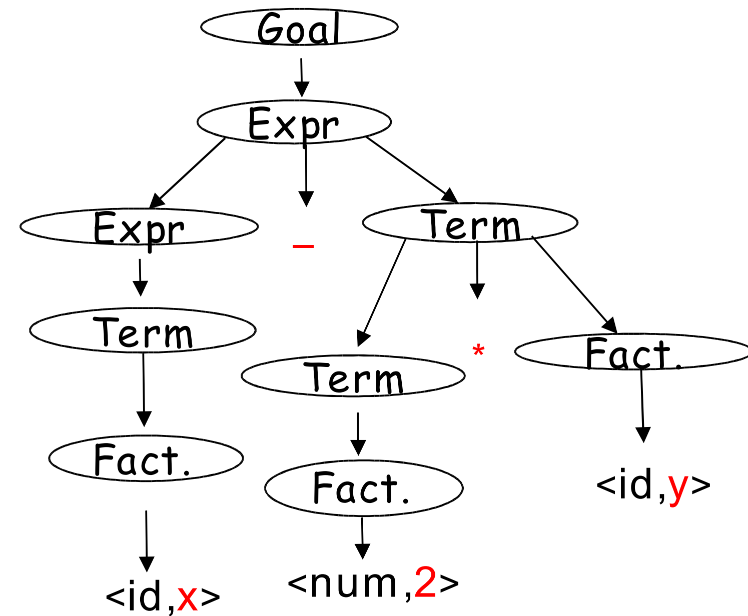
Stack	Input	Handle	Action
$	id – num * id	*none*	shift
$ id	– num * id	9,1	red. 9
$ *Factor*	– num * id	7,1	red. 7
$ *Term*	– num * id	4,1	red. 4
$ *Expr*	– num * id	*none*	shift
$ *Expr* –	num * id	*none*	shift
$ *Expr* – num	* id	8,3	red. 8
$ *Expr* – *Factor*	* id	7,3	red. 7
$ *Expr* – *Term*	* id	*none*	shift
$ *Expr* – *Term* *	id	*none*	shift
$ *Expr* – *Term* * id		9,5	red. 9
$ *Expr* – *Term* * *Factor*		5,5	red. 5
$ *Expr* – *Term*		3,3	red. 3
$ *Expr*		1,1	red. 1
$ *Goal*		*none*	accept

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 & reduce

Example

Stack	Input	Action
$	id – num * id	shift
$ id	– num * id	red. 9
$ *Factor*	– num * id	red. 7
$ *Term*	– num * id	red. 4
$ *Expr*	– num * id	shift
$ *Expr* –	num * id	shift
$ *Expr* – num	* id	red. 8
$ *Expr* – *Factor*	* id	red. 7
$ *Expr* – *Term*	* id	shift
$ *Expr* – *Term* *	id	shift
$ *Expr* – *Term* * id		red. 9
$ *Expr* – *Term* * *Factor*		red. 5
$ *Expr* – *Term*		red. 3
$ *Expr*		red. 1
$ *Goal*		accept



Shift-reduce Parsing

Shift reduce parsers are easily built and easily understood

A shift-reduce parser has just 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

Handle finding is key
- handle is on stack
- finite set of handles

$\Rightarrow$ use a DFA !

Accept & Error are simple

Shift is just a push and a call to the scanner

Reduce takes |rhs| pops & 1 push

If handle-finding requires state, put it in the stack $\Rightarrow$ 2x work

An Important Lesson about Handles

To be a handle, a substring of a sentential form γ must have two properties:

- It must match the right hand side β of some rule $A \rightarrow \beta$
- There must be some rightmost derivation from the goal symbol that produces the sentential form γ with $A \rightarrow \beta$ as the last production applied

- Simply looking for right hand sides that match strings 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 look ahead in the grammar along with tables produced as the result of analyzing the grammar.
  - LR(1) parsers build a DFA that runs over the stack & finds them

Extra Slides Start Here

An Important Lesson about Handles

- To be a handle, a substring of a sentential form γ must have two properties:
  - It must match the right hand side β of some rule $A \rightarrow \beta$
  - There must be some rightmost derivation from the goal symbol that produces the sentential form γ with $A \rightarrow \beta$ as the last production applied
- We have seen that simply looking for right hand sides that match strings 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 look ahead in the grammar along with tables produced as the result of analyzing the grammar.
    - There are a number of different ways to do this.
    - We will look at two: operator precedence and LR parsing