Parsing IV
Bottom-up Parsing

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 *(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

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 non-terminals, γ is just a sentential form
- To get γ_i from γ_{i-1}, expand some $N \in \gamma_{i-1}$ by using $N \rightarrow \beta$
  - Replace the occurrence of $N \in \gamma_{i-1}$ with β to get γ_i
  - In a leftmost derivation, it would be the first $N \in \gamma_{i-1}$

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*

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

To derive γ_{i-1} from γ_i, it matches $\gamma_i = \alpha\, b\, d$ against γ_i, then replaces b with its corresponding *lhs*, N, to yield $\gamma_{i-1} = \alpha\, N\, d$.

(assuming $N \rightarrow b$)

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 in $\alpha\, b\, d$ with N shrinks the upper fringe, we call it a *reduction*.

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

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

Finding Reductions

Consider the simple grammar

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

Sentential Form	*Next Red'n*	
	Prod'n	*Pos'n*
a b bcd e	3	2
a *A* b c de	2	4
a *A* d e	4	3
a *A B* e	1	4
Goal	—	—

And the input string abbcde

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 $\alpha\, b\, d$ of the tree's frontier that

> *matches some production* $N \rightarrow b$ *that occurs as one step*
> *In the rightmost derivation* *(* $\Rightarrow N \rightarrow b$ *is in RRD)*

Informally, we call this substring β a *handle*

Formally,

> A *handle* of a **right**-sentential form γ is a pair $\langle N \rightarrow \beta, \boldsymbol{k} \rangle$ where $N \rightarrow \beta \in P$ and $\boldsymbol{k}$ is the position in γ of β's rightmost symbol.
>
> If $\langle N \rightarrow \beta, \boldsymbol{k} \rangle$ is a handle, then replacing β at $\boldsymbol{k}$ with N produces the right sentential form preceding γ in the rightmost derivation.

Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols *(convoluted, but true)*

$\Rightarrow$ 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 $N \rightarrow \beta$ applied to take γ_{i-1} to γ_i
3. $\Rightarrow$ a unique position ***k*** at which $N \rightarrow \beta$ is applied
4. $\Rightarrow$ a unique handle $<N \rightarrow \beta, \boldsymbol{k}>$

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

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

from Cooper and Torczon

See Figure 3.5 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 $\langle N_i \rightarrow b_i, \mathbf{k}_i \rangle$ in γ_i

Replace b_i *with* N_i *to generate* γ_{i-1}

This takes *2n* steps

Handle-pruning, Bottom-up Parsers

One implementation technique is the *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 N → β
        then  /* reduce β to N */
            pop |β| symbols off the stack
            push N onto the stack
        else if (token ≠ EOF)
            then /* shift */
                push token
                token ← next_token( )
```

How do errors show up?

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

Either generates an error

from Cooper and Torczon

Figure 3.7 in EAC

Back to x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	*non e*	shif t
$ id	– num * id	9,1	red . 9
$ *Fac to r*	– num * id	7,1	red . 7
$ *Te rm*	– num * id	4,1	red . 4
$ *Ex pr*	– num * id	*non e*	shif t
$ *Ex pr* –	num * id	*non e*	shif t
$ *Exp r* – num	* id	8,3	red . 8
$ *Ex pr* – *Fac to r*	* id	7,3	red . 7
$ *Exp r* – *Te rm*	* id	*non e*	shif t
$ *Exp r* – *Te rm* *	id	*non e*	shif t
$ *Ex pr* – *Te rm* * id		9,5	red . 9
$ *Ex pr* – *Te rm* * *Fac to r*		5,5	red . 5
$ *Ex pr* – *Te rm*		3,3	red . 3
$ *Ex pr*		1,1	red . 1
$ *Goal*		*non e*	ac c ept

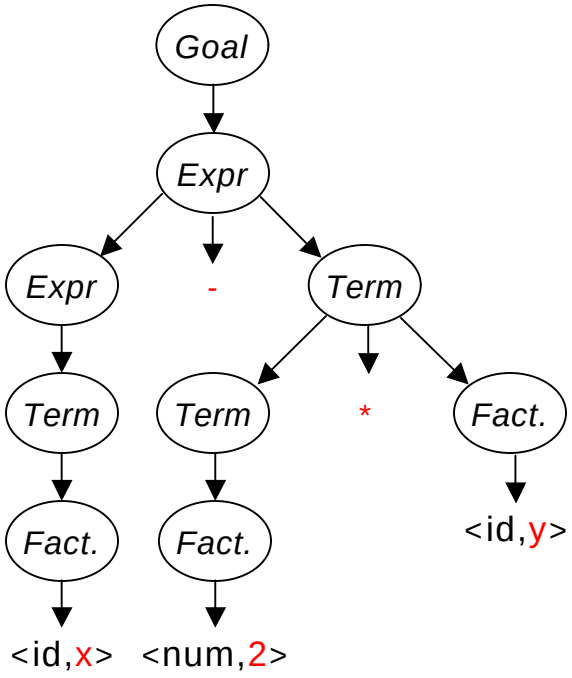
5 shifts +
9 reduces + 1
accept

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

from Cooper and Torczon

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

⇒ 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 Þ *2x work*