

## *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  $\gamma_i$  is a sentential form
  - > If  $\alpha$  contains only terminal symbols,  $\alpha$  is a **sentence** in  $L(G)$
  - > If  $\alpha$  contains  $\geq 1$  non-terminals,  $\alpha$  is a **sentential form**
- To get  $\gamma_i$  from  $\gamma_{i-1}$ , expand some NT  $A \in \gamma_{i-1}$  by using  $A \rightarrow \beta$ 
  - > Replace the occurrence of  $A \in \gamma_{i-1}$  with  $\beta$  to get  $\gamma_i$
  - > 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  $S$

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

To reduce  $\gamma_i$  to  $\gamma_{i-1}$  match some *rhs*  $\beta$  against  $\gamma_i$   
 then replace  $\beta$  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  $\beta$  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		<i>Goal</i>	→	<u>a</u> A B <u>e</u>
2		A	→	A <u>b</u> <u>c</u>
3				<u>b</u>
4		B	→	<u>d</u>

<i>Sentential Form</i>	<i>Next Red'n</i>	
	<i>Prod'n</i>	<i>Pos'n</i>
<u>abcde</u>	3	2
<u>a</u> A bcde	2	4
<u>a</u> A <u>d</u> e	4	3
<u>a</u> A B <u>e</u>	1	4
<i>Goal</i>	—	—

And the input string abcde

*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  $\beta$  of the tree's frontier that

*matches some production  $A \rightarrow \beta$  that occurs as one step  
in the rightmost derivation* ( $\Rightarrow \beta \rightarrow A$  is in RRD)

Informally, we call this substring  $\beta$  a *handle*

Formally,

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

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

Because  $\gamma$  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  $\gamma_i$  from  $\gamma_{i-1}$
- 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)*

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

The expression grammar

Handles for rightmost derivation of  $x = 2 * y$

**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

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

Apply the following simple algorithm

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

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

Replace  $\beta_i$  with  $A_i$  to generate  $\gamma_{i-1}$

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  $A \rightarrow \beta$ 
    then /* reduce  $\beta$  to  $A$  */
      pop  $|\beta|$  symbols off the stack
      push  $A$  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

## Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	<u>id</u> - num * id	none	shift
\$ <u>id</u>	- num * id		

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

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		

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

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		

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

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		

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

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			

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

Back to  $x = 2 * y$

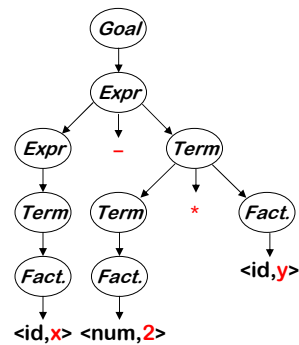
Stack	Input	Handle	Action
\$	id = num * id	none	shift
\$ id	= num * id	9,1	red. 9
\$ Factor	= num * id	7,1	red. 7
\$ Term	= num * id	4,1	red. 4
\$ Expr	= num * id	none	shift
\$ Expr =	num * id	none	shift
\$ Expr = num	* id	8,3	red. 8
\$ Expr = Factor	* id	7,3	red. 7
\$ Expr = Term	* id	none	shift
\$ 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	red. 9
\$ Expr = Term * id		red. 5
\$ 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 ⇒ 2x work*

CMSC430 Spring 2007

17

## An Important Lesson about Handles

- To be a handle, a substring of a sentential form  $\gamma$  must have two properties:
  - > It must match the right hand side  $\beta$  of some rule  $A \rightarrow \beta$
  - > There must be some rightmost derivation from the goal symbol that produces the sentential form  $\gamma$  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

CMSC430 Spring 2007

18

### *Finding Handles*

---

- **Assumption:** in a well-formed grammar, every non-terminal symbol can be found in some legal sentential form
  - > That is, given a non-terminal  $A$  there is a derivation that produces a sentential form with  $A$  somewhere in it
  - > **Consequence:** there is a rightmost derivation that produces a sentential form  $\alpha A \delta$  with  $A$  as the last non-terminal.
  - > **Consequence:** If  $A \rightarrow \beta$  is a production in the grammar, during shift-reduce parsing,  $\beta$  on the stack is a handle when followed by  $\delta$  in the input.
  - > **Special case:** let  $\underline{d}$  be the first character of  $\delta$ . For some grammars,  $\beta$  on the stack followed by  $\underline{d}$  will always be a handle.
  - > **Even more special case:** Let  $Z$  be the last symbol (terminal or non-terminal) of  $\beta$ . In some restricted grammars, called *simple precedence grammars*,  $Z$  on the stack followed by  $\underline{d}$  in the input is always the end of a handle.

CMSC430 Spring 2007

19

### *Precedence - Definitions*

---

- $a < b$  if  $X \rightarrow aY$  and  $Y \Rightarrow bw$  for some string  $w$
- $a = b$  if  $X \rightarrow w_1 a b w_2$  for strings  $w_1$  and  $w_2$
- $a > b$  if  $X \rightarrow Yb$  and  $Y \Rightarrow wa$  for some string  $w$ 
  - or
  - if  $X \rightarrow w_1 Y Z w_2$  and  $Y \Rightarrow w_3 a$  and  $Z \Rightarrow b w_4$
- For **ANY** context free grammar we can compute the precedence relations between any 2 symbols in the grammar
- But: If there is at most one precedence relation between any 2 symbols in the grammar, AND the right hand side of every production is unique, then the grammar is a precedence grammar.
- We can use this fact to generate a simple table driven bottom up parser.

CMSC430 Spring 2007

20

### General precedence

	FIRST	LAST
$S \rightarrow \perp E \perp$	$\perp$	$\perp$
$E \rightarrow E+T \mid T$	ETFin(	TFin)
$T \rightarrow T^*F \mid F$	TFin(	Fin)
$F \rightarrow i \mid n \mid (E)$	In)	In)

	S	E	T	F	+	*	i	n	(	)	$\perp$
S											
E					=						=
T					>	=				>	>
F					>	>				>	>
+			$\Leftarrow$	<			<	<	<		
*				=			<	<	<		
i					>	>				>	>
n					>	>				>	>
(		$\Leftarrow$	<	<			<	<	<		
)					>	>				>	>
$\perp$		$\Leftarrow$	<	<			<	<	<		

$\perp = E = \perp$
$\perp < \text{first}(E) \perp < \text{EFTin}(\text{Last}(E) > \perp \text{TFin}) > \perp$
$E = + = T$
$\text{Last}(E) > + \text{TFin} > +$
$+ < \text{first}(T) + < \text{Tfin}(\text{T} = * = F$
$\text{Last}(T) > * \text{Fin} > *$
$* < \text{first}(F) * < \text{in}(\text{E} = E =$
$( < \text{first}(E) ( < \text{EFTin}(\text{Last}(E) > \text{TFin}) >$

Conflicts over E and T – so not precedence

CMSC430 Spring 2007

21

### Remove ambiguity

	FIRST	LAST
$S \rightarrow \perp E' \perp$	$\perp$	$\perp$
$E' \rightarrow E$	ETT'Fin(	ETT'Fin)
$E \rightarrow E+T' \mid T'$	ETT'Fin(	TT'Fin)
$T' \rightarrow T$	TFin(	TFin)
$T \rightarrow T^*F \mid F$	TFin(	Fin)
$F \rightarrow i \mid n \mid (E')$	in)	in)

No conflicts in table – so a precedence grammar

	S	E	T	F	+	*	i	n	(	)	$\perp$	E'	T'
S													
E			>	>	=		>	>		>	>		
T					>	=				>	>		
F					>	>				>	>		
+			<	<			<	<	<			=	
*				=			<	<	<				
i					>	>				>	>		
n					>	>				>	>		
(		<	<	<			<	<	<			=	
)					>	>				>	>		
$\perp$		<	<	<			<	<	<			=	
E'											=	=	
T'					>					>	>		

CMSC430 Spring 2007

22



## Operator Precedence Parse Algorithm

```

let Stack contain "#";
nextToken = first input token;
while (topTerm(Stack) ≠ "#" and input ≠ "#") do begin
  p = precedence[topTerm, nextToken];
  if p == "<" or p == "=" then /* shift */
    shift nextToken onto stack and advance input;
  else if p == ">" then begin /* reduce */
    find the shallowest pair of terminals d and s on the stack
    such that d < s, where d is the deeper terminal;
    pop everything above d off the stack;
    push N, the general non-terminal, onto the stack;
  end
  else if p == "acc" then exit loop; /* accept */
  else /* precedence undefined */ report error; /* error */
end
end

```

CMSC430 Spring 2007

25

## Operator Precedence Example

- Recall the simple grammar:

```

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

```

Operator Precedence Table

	a	b	c	d	e	#
#	<					acc
a		<		=		
b		>	=	>		
c		>		>		
d					=	
e						>

- Operator grammar:

```

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

```

Sentential Form	Next Red'n	
	Prod'n	Pos'n
<u>abbcde</u>	3	2
<u>a</u> A <u>bcde</u>	2	4
<u>a</u> A <u>de</u>	1	4
Goal	—	—

CMSC430 Spring 2007

26

### Operator Precedence Parse Tables for Expressions

---

1	<i>Goal</i> → <i>Expr</i>										
2	<i>Expr</i> → <i>Expr</i> + <i>Term</i>	<b>#</b>	<	<	<	<	<	<	<	<	<b>acc</b>
3	<i>Expr</i> - <i>Term</i>	<b>id</b>		>	>	>	>	>	>	>	>
4	<i>Term</i>	<b>num</b>		>	>	>	>	>	>	>	>
5	<i>Term</i> → <i>Term</i> * <i>Factor</i>	<b>+</b>	<	<	>	>	<	<	<	>	>
6	<i>Term</i> / <i>Factor</i>	<b>-</b>	<	<	>	>	<	<	<	>	>
7	<i>Factor</i>	<b>*</b>	<	<	>	>	>	>	<	>	>
8	<i>Factor</i> → <u>number</u>	<b>/</b>	<	<	>	>	>	>	<	>	>
9	<u>id</u>	<b>(</b>	<	<	<	<	<	<	<	=	>
10	( <i>Expr</i> )	<b>)</b>		>	>	>	>	>	>	>	>

### Operator Precedence Parse of $x-2*y$

---

Stack	Prec	Input	Action
#	<	<b>id = num * id #</b>	shift
# <u>id</u>	>	<b>= num * id #</b>	reduce
# <i>N</i>	<	<b>= num * id #</b>	shift
# <i>N</i> =	<	<b>num * id #</b>	shift
# <i>N</i> = <u>num</u>	>	<b>* id #</b>	reduce
# <i>N</i> = <i>N</i>	<	<b>* id #</b>	shift
# <i>N</i> = <i>N</i> *	>	<b>id #</b>	shift
# <i>N</i> = <i>N</i> * <u>id</u>	>	<b>#</b>	reduce
# <i>N</i> = <i>N</i> * <i>N</i>	>	<b>#</b>	reduce
# <i>N</i> = <i>N</i>	>	<b>#</b>	reduce
# <i>N</i>	<b>acc</b>	<b>#</b>	accept

### Computing Operator Precedence Relations

---

- Define the following relations
  - > **NBEFORE**  $\underline{t}$  iff there is some production  $A \rightarrow \beta$  in which non-terminal  $N$  occurs immediately before terminal  $\underline{t}$
  - > **NATER**  $\underline{t}$  iff there is some production  $A \rightarrow \beta$  in which non-terminal  $N$  occurs immediately after terminal  $\underline{t}$
  - > **N<sub>1</sub>FIRST**  $N_2$  iff there is some production  $N_1 \rightarrow \beta$  in which non-terminal  $N_2$  occurs as the first symbol on the rhs
  - > **N<sub>1</sub>LAST**  $N_2$  iff there is some production  $N_1 \rightarrow \beta$  in which non-terminal  $N_2$  occurs as the last symbol on the rhs
  - > **NFIRSTTERM**  $\underline{t}$  iff there is some production  $N \rightarrow \beta$  in which  $\underline{t}$  is the first terminal on the rhs
  - > **NLASTTERM**  $\underline{t}$  iff there is some production  $N \rightarrow \beta$  in which  $\underline{t}$  is the last terminal on the rhs

CMSC430 Spring 2007

29

### Computing Operator Precedence Relations

---

- $\underline{t}_1$  **EQUAL**  $\underline{t}_2$ 
  - > iff there is some production  $A \rightarrow \beta$  in which  $\underline{t}_1$  immediately precedes  $\underline{t}_2$  on the right hand side or they are separated by a single non-terminal
- $\underline{t}_1$  **LESSTHAN**  $\underline{t}_2$ 
  - > **LESSTHAN** = **AFTER**<sup>T</sup> · **FIRST**<sup>\*</sup> · **FIRSTTERM**
  - >  $N_1$  **AFTER**  $\underline{t}_1$  &  $N_1 \rightarrow^* N_2 \alpha$  &  $N_2 \rightarrow \beta$  &  $\underline{t}_2$  is the first terminal in  $\beta$
- $\underline{t}_1$  **GREATERTHAN**  $\underline{t}_2$ 
  - > **GREATERTHAN** = (**LAST**<sup>\*</sup> · **LASTTERM**)<sup>T</sup> · **BEFORE**
  - >  $N_1$  **BEFORE**  $\underline{t}_2$  &  $N_1 \rightarrow^* \alpha N_2$  &  $N_2 \rightarrow \beta$  &  $\underline{t}_1$  is the last terminal in  $\beta$

CMSC430 Spring 2007

30

### Operator Precedence Example

- Recall the operator grammar:

0	$G \rightarrow \#S\#$
1	$S \rightarrow \underline{a}A\underline{d}e$
2	$A \rightarrow A\underline{b}c$
3	$\underline{b}$

Operator Precedence Table

	a	b	c	d	e	#
#	<					acc
a		<		=		
b		>	=	>		
c		>		>		
d					=	
e						>

### Operator Precedence Example

- Recall the operator grammar:

0	$G \rightarrow \#S\#$
1	$S \rightarrow \underline{a}A\underline{d}e$
2	$A \rightarrow A\underline{b}c$
3	$\underline{b}$

- Relations

LAST	G	S	A
G	0	0	0
S	0	0	0
A	0	0	0

LAST'	G	S	A
G	1	0	0
S	0	1	0
A	0	0	1

LASTTERM	a	b	c	d	e	#	BEFORE	a	b	c	d	e	#
G	0	0	0	0	0	1	G	0	0	0	0	0	0
S	0	0	0	0	1	0	S	0	0	0	0	0	1
A	0	1	1	0	0	0	A	0	1	0	1	0	0

### Operator Precedence Example

LASTTERM <sup>T</sup>	G	S	A		BEFORE						
<u>a</u>	0	0	0	x	<u>G</u>	<u>a</u>	<u>b</u>	<u>c</u>	<u>d</u>	<u>e</u>	<u>#</u>
<u>b</u>	0	0	1		<u>S</u>	0	0	0	0	0	1
<u>c</u>	0	0	1		<u>A</u>	0	1	0	1	0	0
<u>d</u>	0	0	0								
<u>e</u>	0	1	0								
<u>#</u>	1	0	0								

GRTRTHAN	a	b	c	d	e	#	
<u>a</u>	0	0	0	0	0	0	
<u>b</u>	0	1	0	1	0	0	
<u>c</u>	0	1	0	1	0	0	
<u>d</u>	0	0	0	0	0	0	
<u>e</u>	0	0	0	0	0	1	
<u>#</u>	0	0	0	0	0	0	

	a	b	c	d	e	#
<u>#</u>	<					acc
<u>a</u>		<		=		
<u>b</u>		>	=	>		
<u>c</u>		>		>		
<u>d</u>					=	
<u>e</u>						>

CMSC430 Spring 2007

33

### Final Remarks on Operator Precedence

- Developed by Floyd for expression grammar
  - > But has been used for whole languages
  - > Sometimes used in a hybrid parser with top-down recursive descent
- **Abstract syntax trees** are easy to construct
  - > Keep a pointer to the AST for each non-terminal in its *N*node on the stack
  - > When a reduction is performed, create an operator node with pointers to the popped nodes within it — make this the root of the tree pointed to by the non-terminal pushed onto the stack
  - > When parsing stops, a pointer to the AST is on top of the stack
- Full parse trees are hard to construct

CMSC430 Spring 2007

34