Parsing V
Operator-Precedence Parsing

COMP 412
Fall 2005

Copyright 2005, Keith D. Cooper, Ken Kennedy, & Linda Torczon, all rights reserved. Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

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

Accept & *Error* are simple

Shift is just a push and a call to the scanner

Reduce takes |*rhs*| pops & 1 push

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 ananyzing the grammar.
    - There are a number of different ways to do this.
    - We will look at two: *operator precedence* and *LR* parsing

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, β on the stack is a handle when followed by δ in the input.
  - Special case: let $\underline{d}$ be the first character of δ. For some grammars, β 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 β. 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.

Operator Precedence Parsing

- Even more special case:
  - *Operator grammar:* no production has two non-terminal symbols in sequence on the right-hand side
  - An operator grammar can be parsed using shift-reduce parsing and *precedence relations* between terminal symbols to find handles. This strategy is known as *operator precedence* parsing.
- Precedence relations: given two terminal symbols *x* and *y*
  - We say that they have equal precedence or *x* **8** *y* if they appear on the same right-hand side of a rule in the grammar.
  - We say that *x* has lower precedence than *y* or *x* **A** *y* if *x* can appear as the last terminal symbol before a handle in which *y* appears as the first terminal symbol.
  - We say that *x* has greater precedence than *y* or *x* **S** *y* if *y* can appear as the first terminal symbol after a handle in which *x* appears as the last terminal symbol.

Operator Precedence Parse Algorithm

```
let Stack contain "#";
nextToken = first input token;
while (topTerm(Stack) ≠ "#" and input ≠ "#") do begin
    p = precedence [topTerm, nextToken];
    if p == "A" or p == "8" then /* shift */
        shift nextToken onto stack and advance input;
    else if p == "S" then begin /* reduce */
        find the shallowest pair of terminals d and s on the stack
            such that d A 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
```

Operator Precedence Example

- Recall the simple grammar:

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

- Operator grammar:

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

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	#
#	A					acc
a		A		8		
b		S	8	S		
c		S		S		
d					8	
e						S

- Operator grammar:

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

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	#
#	A					acc
a		A				
b		S	8	S		
c				S		
d					8	
e						S

- Operator grammar:

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

Sentential Form	*Next Red'n Prod'n*	*Next Red'n Pos'n*
abbcde	3	2
a *A* bcde	2	4
a *A* de	1	4
Goal	—	—

Operator Precedence Parse Tables for Expressions

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*)

Operator Precedence Tables for Expressions

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

	id	num	+	–	*	/	(	)	#
#	A	A	A	A	A	A	A		acc
id			S	S	S	S		S	S
num			S	S	S	S		S	S
+	A	A	S	S	A	A	A	S	S
–	A	A	S	S	A	A	A	S	S
*	A	A	S	S	S	S	A	S	S
/	A	A	S	S	S	S	A	S	S
(	A	A	A	A	A	A	A	8	
)			S	S	S	S		S	S

Operator Precedence Parse of x-2*y

Stack	Prec	Input	Action
#	A	id – num * id #	shift
# id	S	– num * id #	

Operator Precedence Parse of x-2*y

Stack	Prec	Input	Action
#	A	id – num * id #	shift
# id	S	– num * id #	reduce
# *N*	A	– num * id #	

Operator Precedence Parse of x-2*y

Stack	Prec	Input	Action
#	A	id – num * id #	shift
# id	S	– num * id #	reduce
# *N*	A	– num * id #	shift
# *N* –	A	num * id #	shift
# *N* – num	S	* id #	

Operator Precedence Parse of x-2*y

Stack	Prec	Input	Action
#	A	id – num * id #	shift
# id	S	– num * id #	reduce
# *N*	A	– num * id #	shift
# *N* –	A	num * id #	shift
# *N* – num	S	* id #	reduce
# *N* – *N*	A	* id #	

Operator Precedence Parse of x-2*y

Stack	Prec	Input	Action
#	A	id – num * id #	shift
# id	S	– num * id #	reduce
# *N*	A	– num * id #	shift
# *N* –	A	num * id #	shift
# *N* – num	S	* id #	reduce
# *N* – *N*	A	* id #	shift
# *N* – *N* *	A	id #	shift
# *N* – *N* * id	S	#	

Operator Precedence Parse of x-2*y

Stack	Prec	Input	Action
#	A	id – num * id #	shift
# id	S	– num * id #	reduce
# *N*	A	– num * id #	shift
# *N* –	A	num * id #	shift
# *N* – num	S	* id #	reduce
# *N* – *N*	A	* id #	shift
# *N* – *N* *	A	id #	shift
# *N* – *N* * id	S	#	reduce
# *N* – *N* * *N*	S	#	

Operator Precedence Parse of x-2*y

Stack	Prec	Input	Action
#	A	id – num * id #	shift
# id	S	– num * id #	reduce
# *N*	A	– num * id #	shift
# *N* –	A	num * id #	shift
# *N* – num	S	* id #	reduce
# *N* – *N*	A	* id #	shift
# *N* – *N* *	A	id #	shift
# *N* – *N* * id	S	#	reduce
# *N* – *N* * *N*	S	#	reduce
# *N* – *N*	S	#	

Operator Precedence Parse of x-2*y

Stack	Prec	Input	Action
#	A	id – num * id #	shift
# id	S	– num * id #	reduce
# *N*	A	– num * id #	shift
# *N* –	A	num * id #	shift
# *N* – num	S	* id #	reduce
# *N* – *N*	A	* id #	shift
# *N* – *N* *	A	id #	shift
# *N* – *N* * id	S	#	reduce
# *N* – *N* * *N*	S	#	reduce
# *N* – *N*	S	#	reduce
# *N*	acc	#	

Operator Precedence Parse of x-2*y

Stack	Prec	Input	Action
#	A	id – num * id #	shift
# id	S	– num * id #	reduce
# *N*	A	– num * id #	shift
# *N* –	A	num * id #	shift
# *N* – num	S	* id #	reduce
# *N* – *N*	A	* id #	shift
# *N* – *N* *	A	id #	shift
# *N* – *N* * id	S	#	reduce
# *N* – *N* * *N*	S	#	reduce
# *N* – *N*	S	#	reduce
# *N*	acc	#	accept

Computing Operator Precedence Relations

- Define the following relations
  - N BEFORE $\underline{t}$ iff there is some production $A \rightarrow \beta$ in which non-terminal N occurs immediately before terminal $\underline{t}$
  - N AFTER $\underline{t}$ iff there is some production $A \rightarrow \beta$ in which non-terminal N occurs immediately after terminal $\underline{t}$
  - N_1 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_1 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
  - N FIRSTTERM $\underline{t}$ iff there is some production $N \rightarrow \beta$ in which $\underline{t}$ is the first terminal on the rhs
  - N LASTTERM $\underline{t}$ iff there is some production $N \rightarrow \beta$ in which $\underline{t}$ is the last terminal on the rhs

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^T · FIRST* · 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 β

- $\underline{t}_1$ GREATERTHAN $\underline{t}_2$
  - GREATERTHAN = (LAST* · LASTTERM)^T · 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 β

Operator Precedence Example

- Recall the operator grammar:

0	*G*	→	# *S* #
1	*S*	→	a *A* d e
2	*A*	→	*A* b c
3		|	b

Operator Precedence Table

	a	b	c	d	e	#
#	A					acc
a		A		8		
b		S	8	S		
c		S		S		
d					8	
e						S

Operator Precedence Example

- Recall the operator grammar:

0	*G*	→	# *S* #
1	*S*	→	a *A* d e
2	*A*	→	*A* b c
3		|	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	#
G	0	0	0	0	0	1
S	0	0	0	0	1	0
A	0	1	1	0	0	0

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

Operator Precedence Example

LASTTERM^T

	G	*S*	*A*
a	0	0	0
b	0	0	1
c	0	0	1
d	0	0	0
e	0	1	0
#	1	0	0

x

BEFORE

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

=

GRTRTHAN

	a	b	c	d	e	#
a	0	0	0	0	0	0
b	0	1	0	1	0	0
c	0	1	0	1	0	0
d	0	0	0	0	0	0
e	0	0	0	0	0	1
#	0	0	0	0	0	0

	a	b	c	d	e	#
#	A					acc
a		A		8		
b		S	8	S		
c		S		S		
d					8	
e						S

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