6 Parsing 2 - Bottom up, operator precedence parsing

Bottom up parsing is based on the reverse process to top down. Instead of expanding successive non-terminals according to production rules, to predict the legal next symbol, a current string or right-sentential form is collapsed each time, until the start non-terminal is reached. This can be regarded as a series of reductions.

The approach is known as *shift-reduce* parsing. The techniques for shift-reduce parsing use certain concepts which we define here.

A right sentential form is a sequence within the current string of terminals and reduced non- terminals that matches the form of the right hand side of a production in the grammar. Thus, a sequence which is right sentential may be reduced to the non-terminal on the left hand side of that production.

A handle is a right sentential sequence which is the *rightmost derivation* within the string being parsed. The shift-reduce sequence results in the reverse of a series of rightmost derivations to produce the string of terminals forming the program.

Take the grammar

$$S \rightarrow \text{a } A \; B \text{ e} \quad (1)$$
$$A \rightarrow A \text{ b c} \mid \text{b} \quad (2)$$
$$B \rightarrow \text{d} \quad (3)$$

and the sentence **abbcde**. Parsing by bottom up methods gives the following series of reductions.

abbcde	
aAbcde	by (2)
aAde	by (2)
a AB e	by (3)
S	by (1)

Reversed this gives

$$\begin{aligned} S &\Rightarrow \\ \text{a}AB\text{e} &\Rightarrow \\ \text{a}A\text{de} &\Rightarrow \\ \text{a}A\text{bcde} &\Rightarrow \\ \text{abbcde} & \end{aligned}$$

which is clearly a series of rightmost derivations. The question is how do we decide which reduction corresponds to a rightmost derivation? It is not the rightmost reduction, necessarily, nor always the leftmost. How, then do we define the handles at the various stages?

A second problem arises when the same right sentential form is found in more than one production. How do we choose which to reduce by? For instance, in some languages both

$$\textit{proc-call} \;\rightarrow\; \textbf{id}\ (\ (\ \textit{exp}\ (\ ,\ \textit{exp}\ )^{*}\ )^{?}\ )$$

$$\textit{array-var} \;\rightarrow\; \textbf{id}\ (\ \textit{exp}\ (\ ,\ \textit{exp}\ )^{*}\ )$$

are productions, one as a procedure or function call, the other as a subscripted variable. The sub-string `x(3)` fits both.

A final problem concerns ambiguous grammars, where more than one rightmost derivation can lead to a particular string. Bottom up parsing has to be able to choose between two or more valid handles. Thus, for the grammar

$$\begin{aligned}
\textit{exp} &\rightarrow \textit{exp} + \textit{exp}\\
\textit{exp} &\rightarrow \textit{exp} * \textit{exp}\\
\textit{exp} &\rightarrow (\ \textit{exp})\\
\textit{exp} &\rightarrow \text{id}
\end{aligned}$$

we can have the two following rightmost derivations

$$\begin{aligned}
\textit{exp} &\rightarrow \textit{exp} + \textit{exp}\\
&\rightarrow \textit{exp} + \textit{exp} * \textit{exp}\\
&\rightarrow \textit{exp} + \textit{exp} * \textit{id}_3\\
&\rightarrow \textit{exp} + \textit{id}_2 * \textit{id}_3\\
&\rightarrow \textit{id}_1 + \textit{id}_2 * \textit{id}_3
\end{aligned}
\qquad
\begin{aligned}
\textit{exp} &\rightarrow \textit{exp} * \textit{exp}\\
&\rightarrow \textit{exp} * \textit{id}_3\\
&\rightarrow \textit{exp} + \textit{exp} * \textit{id}_3\\
&\rightarrow \textit{exp} + \textit{id}_2 * \textit{id}_3\\
&\rightarrow \textit{id}_1 + \textit{id}_2 * \textit{id}_3
\end{aligned}$$

In the first, $*$ has higher precedence, in the second $+$ has.

Implementing shift reduction

The normal way to view a shift reduce parser uses the notion of

an input stream , containing

a phrase to be parsed and

a stack holding symbols.

The input stream holds terminals, the stack can hold a mixture of terminals and non-terminals, the latter generated by earlier reductions.

The operation *shift* moves a symbol from the input to the stack. The operation *reduce* combines the sequence ending with the last terminal shifted, to form a non-terminal on the stack. When the string is exhausted, the single start symbol should be present, assuming all reductions have been performed.

The question is how to decide when to shift and when to reduce and, if we choose to reduce, which reduction to apply.

	Sequence of shift/reduce/accept for a string	
Stack	Input	Action
$	id1 + id2 * id3 $	shift
$id1	+ id2 * id3 $	reduce using exp → id
$ exp	+ id2 * id3 $	shift
$ exp +	id2 * id3 $	shift
$ exp + id2	* id3 $	reduce using exp → id
$ exp + exp	* id3 $	shift
$ exp + exp *	id3 $	shift
$ exp + exp *id3	$	reduce using exp → id
$ exp +exp *exp	$	reduce using exp → exp * exp
$ exp + exp	$	reduce using exp → exp + exp
$ exp	$	accept

$ represents the end of the stack or input

Operator precedence parsing

An operator precedence grammar may be used to create a shift/reduce parser. It is not a simple technique to apply to most languages, but where a suitable grammar may be produced it provides an easy technique to implement.

We consider its use to parse arithmetic expressions, with the following grammar.

$$\begin{aligned} exp &\rightarrow exp\ operator\ exp \mid (\ exp\ ) \mid -\ exp \mid \mathrm{id} \\ operator &\rightarrow + \mid - \mid * \mid / \mid \uparrow \end{aligned}$$

Rewriting this as an operator grammar is straightforward.

$$\begin{aligned} exp &\rightarrow exp\ +\ exp \mid exp\ -\ exp \mid exp\ *\ exp \mid exp\ /\ exp \mid exp\ \uparrow\ exp \\ &\mid (\ exp\ ) \mid -\ exp \mid \mathrm{id} \end{aligned}$$

Here no two consecutive non-terminals are found in the production.

We now define precedence relations between certain pairs of terminals. These relations are defined by disjoint relation symbols $\lessdot$, $\doteq$ and $\gtrdot$

Roughly speaking, where we wish $*$ to have higher precedence that $+$, we define the relations:

$+ \lessdot *$

and

$* \gtrdot +$

In other cases we may need to define

$term1 \gtrdot term2$

and

$term1 \lessdot term2$

This is not a contradiction. The relations are disjoint.

Finally, for some pairs no precedence may be defined.

The idea is to build a table of relations, such that a handle will be defined by a sequence with $\lessdot$ marking its start, $\doteq$ possibly the middle and $\gtrdot$at the end. A part of the table for our

grammar is given here.

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

$ is a special, imaginary terminal marking each end of the string to be parsed. $ is always $\lessdot$ any other terminal and any other terminal is always $\gtrdot$$, as in the table.

To parse our string we can now apply these relations, ignoring the non-terminals. Thus

id1 + id2 * id3 $\Rightarrow$ $ $\lessdot$ id1 $\gtrdot$+ $\lessdot$ id2 $\gtrdot$* $\lessdot$ id3 $\gtrdot$$

We scan the resulting sequence, left to right, until the first $\gtrdot$and then backtrack to the immediately left $\lessdot$, ignoring any $\doteq$ relations. The $\lessdot$ and $\gtrdot$enclose the handle, in terms of terminals. Any intervening or adjacent non-terminals are also taken to be in the handle. In our example there are, initially, no non-terminals.

Clearly id1 is the first handle and this is reduced to exp giving:

exp + id2 * id3

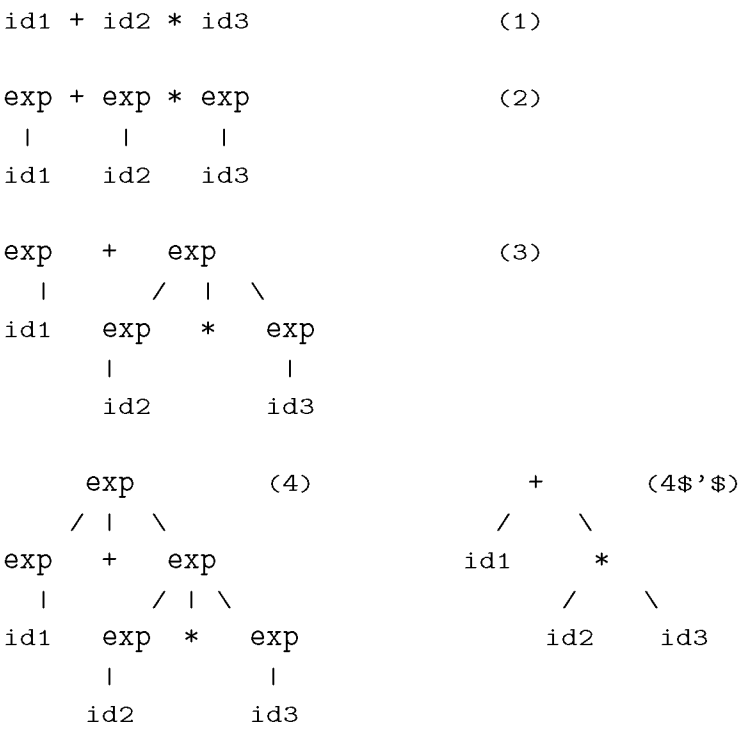
Applying the same process twice more gives us

exp + exp * exp

From this we derive

$ $\lessdot$+ $\lessdot$ * $\gtrdot$$

which has as its handle exp * exp. This is then reduced, reflecting its greater precedence compared to +. Lastly we will find exp + exp as a handle and reduce this. The reductions generate a parse tree in reverse. In each reduction a new node is formed with links down to the symbols from which it was reduced.



The alternative representative form in (4′) moves up the terminals, replacing each *exp* non-terminal, to give a conventional parse tree.

Establishing precedence relationships

The intuitive approach may allow us to assign precedences to conventional operators, but, clearly we need to be able to include all terminals in the grammar. Some more general approach is needed.

Following Bornat, we can use the following method, based on "first operator" and "last operator" lists. It builds a precedence matrix, along the lines of our expression table. (This method is also found in Aho and Ullman, but not in A S and U.)

Firstop+ is a list of all those terminal symbols ("operators") which can appear first on any right hand side of a production. *Lastop+* is a similar list of those terminals which can appear last. The steps are:

1. For each non-terminal, i.e. left hand side, construct a Firstop list containing the first terminal in each production for that non-terminal. Where a non-terminal is the first symbol on the right hand side, include both it and the first terminal following, e.g. for

   $$X \rightarrow \mathrm{a}\ldots|B\mathrm{c}$$

   include a, c and B in X's *Firstop* list.

2. Similarly construct a *Lastop* list for each non-terminal, e.g. for

   $$Y \rightarrow \ldots\ \mathrm{u} \mid \ldots\ \mathrm{v}\ W$$

   include u, v and W in Y's *Lastop* list.

3. Compute the *Firstop+* and *Lastop+* lists using Warshall's Closure Algorithm, as follows:

   (a) Take each non-terminal in turn, in any order and look for it in all the Firstop lists. Add its own first symbol list to any other in which it occurs. Similarly process the Lastop lists.

   (b) The non-terminals may now be deleted from the lists.

4. Construct the precedence matrix by the following rules

   (a) Wherever terminal `a` immediately precedes non-terminal B in any production, put $\mathtt{a} \lessdot \alpha$, where α is any terminal in the *Firstop+* list for B.

   (b) Wherever terminal `b` immediately follows non-terminal C in any production, put $\mathtt{b} \gtrdot \beta$, where β is any terminal in the *Lastop+* list for C.

   (c) Wherever a sequence $\mathtt{a}B\mathtt{c}$ or `ac` occurs in any production, put $\mathtt{a} \doteq \mathtt{c}$.

5. Add the relations $\$ \lessdot \mathtt{a}$ and $\mathtt{a} \gtrdot \$$ for all terminals in the *Firstop+* and *Lastop+* lists, respectively, for S.

Any entries left blank indicate that the two symbols should never occur consecutively in a handle. Such a sequence is a syntax error.

To clarify this, take the grammar, for expressions:

$$
\begin{aligned}
S &\rightarrow A \\
A &\rightarrow T \mid A + T \mid A - T \\
T &\rightarrow F \mid T * F \mid T / F \\
F &\rightarrow P \mid P \uparrow F \\
P &\rightarrow \mathrm{i} \mid \mathrm{n} \mid (A)
\end{aligned}
$$

Applying steps 1 and 2 gives us

Symbol	Firstop	Lastop
S	A	A
A	T + A -	T + -
T	F * T /	F * /
F	P ↑	P ↑ F
P	i n (	i n)

Applying step 3 gives us

Symbol	Firstop+	Lastop+
S	T + A - F * / P ↑ i n (	A T + - F * / P ↑ i n)
A	T + A - F * / P ↑ i n (	T + - F * / P ↑ i n)
T	F * T / P ↑ i n (	F * / P ↑ i n)
F	P ↑ i n (	P ↑ F i n)
P	i n (	i n)

Removing non-terminals gives

Symbol	Firstop+	Lastop+
S	+ - * / ↑ i n (	+ - * / ↑ i n)
A	+ - * / ↑ i n (	+ - * / ↑ i n)
T	* / ↑ i n (	* / ↑ i n)
F	↑ i n (	↑ i n)
P	i n (	i n)

Finally we use steps 4 and 5 to compute our precedence relation matrix.

	$	(	)	i	n	↑	*	/	+	-
$		$\lessdot$		$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$
(		$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$
)	$\gtrdot$		$\gtrdot$			$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
i	$\gtrdot$		$\gtrdot$			$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
n	$\gtrdot$		$\gtrdot$			$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
↑	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$		$\gtrdot$	$\gtrdot$
*	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
/	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
-	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$

Error recovery

There are two basic types of error condition:

- No precedence relation holds between the current top of stack terminal and the next to be input.
- No production has a right sentential form to match the handle found.

To cope with type 1 errors we must change the stack or the input or both. We may change, insert or delete extra symbols to try to produce a "sensible" handle. Insertion could lead to an infinite loop. A "safe" approach is to ensure that whatever is done will allow the next input symbol to be shifted. In general a recovery procedure should be specified for each blank entry in the precedence table.

To cope with type 2 errors we try to produce a "close approximation" to the handle from the legal set of right hand sides and then report the differences as the error. As an example $($ $)$ is a handle and we would like to approximate it to $($ A $)$ and report "Missing expression between parentheses".