Lecture 9

Operator-precedence parsing
Operator-precedence parsing

• For a small but important class of grammars we can easily construct efficient shift-reduce parsers by hand. These grammars have the property (among other essential requirements) that:
  - no production right side is \( \Lambda \);
  - has not two adjacent nonterminals.

A grammar with the latter property is called an operator grammar.
Operator grammars

- The grammar
  \[ E \rightarrow EAE \mid (E) \mid -E \mid \text{id}, \quad A \rightarrow + \mid - \mid * \mid / \mid \uparrow \]

  is not an operator grammar, because the right side EAE has two (in fact three) consecutive nonterminals;

- However, if we substitute for A each of its alternatives, we obtain the following operator grammar:
  \[ E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid \text{id} \]
Operator-precedence parsing

• Historically, the technique was first described as a manipulation on tokens without any reference to an underlying grammar.

• In fact, once we finish building an operator-precedence parser from a grammar, we may effectively ignore the grammar, using the nonterminals on the stack only as placeholder for attributes associated with the nonterminals.
Operator-precedence parsing

As a general parsing technique, operator-precedence parsing has a number of disadvantages:

1) it is hard to handle tokens like the minus sign (-), which has two different precedence (depending on whether it is unary or binary);

2) the relationship between a grammar for the language being parsed and the operator-precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language.

3) only a small class of grammars can be parsed using operator-precedence techniques.
In operator-precedence parsing, we define three disjoint precedence relations: < · , = , · > , between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

- \( a < \cdot b \) - \( a \) yields precedence to \( b \);
- \( a = b \) - \( a \) has the same precedence as \( b \);
- \( a > \cdot b \) - \( a \) takes precedence over \( b \).
Remark.
We should caution the reader that while these relations may appear similar to the arithmetic relations "less than", "equal to" and "greater than", the precedence relations have quite different properties:

• For example, we could have \( a<\cdot b \) and \( a\cdot>b \) for the same language, or we might have none of \( a<\cdot b \) and \( a=b \) and \( a\cdot>b \) holding for some terminals \( a \) and \( b \).
Operator-precedence parsing

There are two common ways of determining what precedence relations should hold between a pair of terminals:

- The first method we discuss is intuitive and is based on the traditional notions of associativity and precedence of operators;

- The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.
**Intuitive method**

Intuitive and is based on the traditional notions of associativity and precedence of operators.

Example:

If * is to have higher precedence than +, we make +<·,* and *·>+. This approach will be seen to resolve the ambiguities of grammar

\[ E \rightarrow E + E | E - E | E \cdot E | E / E | E \uparrow E | (E) | -E | id \]

and it enables us to write an operator-precedence parser for it (although the unary minus sign causes problems).
Method of selecting operator-precedence relations

A method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

For example, for frequently encountered source of ambiguity "handing else" model may be grammar:.......
Method of selecting operator-precedence relations

\[ \text{instr} \rightarrow \text{fit\_instr} \]
\[ \quad \mid \text{mismatch\_instr} \]

\[ \text{fit\_instr} \rightarrow \text{if } \text{wyr} \text{ then } \text{fit\_instr} \text{ else } \text{fit\_instr} \]
\[ \quad \mid \text{other} \]

\[ \text{mismatch\_instr} \rightarrow \text{if } \text{wyr} \text{ then } \text{instr} \]
\[ \quad \mid \text{if } \text{wyr} \text{ then } \text{fit\_instr} \text{ else } \text{mismatch\_instr} \]
Method of selecting operator-precedence relations

Having obtained an unambiguous grammar, there is a mechanical method for constructing operator-precedence relations from it.

These relations may not be disjoint, and they may parse a language other than that generated by the grammar, but with the standard sorts of arithmetic expressions, few problems are encountered in practice.
Operator-precedence relations

The intention of the precedence relations is to delimit the handle of a right-sentential form, with `< · ` marking the left end, `= ` appearing in the interior of the handle, and `· > ` marking the right end.
Operator-precedence relations

Suppose we have a right-sentential form of an operator grammar.

We may write the right-sentential form a

$$\beta_0 a_1 \beta_1 a_2 \beta_2 \ldots a_n \beta_n$$

where each $\beta_i$ is either $\epsilon$ (the empty string) or a single nonterminal, and each $a_i$ is a single terminal, $i=1,\ldots,n$.

Suppose that between $a_i$ and $a_{i+1}$ exactly one of the relations: $\prec, = \lub \succ$ holds.
Further, let us use $ to mark each end of the string, and define (for all terminal b)

\[ \langle \cdot \rangle b \text{ and } b \langle \cdot \rangle \].

Now suppose we remove the nonterminals from the string and place the correct relation \( \langle \cdot = \text{ or } \rangle \), between each pair of terminals and between the endmost terminals and the $'s marking the ends of the string.

For example, suppose we initially have the right-sentential form:

\[ \text{id} + \text{id} \ast \text{id} \]

we have:

\[ \langle \cdot \rangle \text{id} \rangle + \langle \cdot \rangle \text{id} \rangle \ast \langle \cdot \rangle \text{id} \rangle \$. 


Operator-precedence relations

The precedence relations are those given

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td></td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>+</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>*</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>$</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
</tbody>
</table>
Operator-precedence relations

The handle can be found by the following process.

1. Scan the string from the left end until the first $\cdot >$ is encountered.
   \[ $\cdot id \cdot > + \cdot id \cdot > * \cdot id \cdot > $. \]

2. Then scan backwards (to the left) over any $=$'s until a $\cdot <$ is encountered.
   \[ $\cdot id \cdot > + \cdot id \cdot > * \cdot id \cdot > $. \]

3. The handle contains everything to the left of the first $\cdot >$ and to the right of the $\cdot <$ encountered in step (2), including any intervening or surrounding nonterminal.

REMARCH: The inclusion of surrounding nonterminals is necessary so that two adjacent nonterminals do not appear in a right-sentential form.
Operator-precedence relations

Zatem uchwytem jest w tym przypadku id.
We remember the grammar:

\[ E \rightarrow E + E \mid E - E \mid E \times E \mid E / E \mid E^+ E \mid (E) \mid -E \mid id \]

If we are dealing with grammar, we then reduce E to id. At this point we have the right-sentential form

\[ E + id * id \]

Next after reducing in sequence of nonterminals and after using of relations symbol we have

\[ \langle \cdot + < \cdot id \cdot > \ast < \cdot id \cdot > \rangle \]

After reducing the two remaining id's to E by the same steps, we obtain the right-sentential form

\[ E + E * E \]

Next after reducing in sequence of nonterminals and after using of relations symbol we have

\[ \langle \cdot + < \cdot \ast \cdot > \rangle \]

This indicate that the left end of the handle lies between + and * and the right end between * and $\ast$. These precedence relations indicate that, in the right-sentential form \[ E + E \ast E \], the handle is \[ E \ast E \].
Operator-precedence relations

It may appear from the discussion above that the entire right-sentential form must be scanned at each step to find the handle. Such is not the case if we use a stack to store the input symbols already seen and if the precedence relations are used to guide the actions of a shift-reduce parser.

If the precedence relation \(<\) or \(=\) holds between the topmost terminal symbol on the stack and the next input symbol, the parser shifts; it has not yet found the right end of the handle.

If the relation \(>\) holds, a reduction is called for. At this point the parser has found the right end of the handle, and the precedence relations can be used to find the left end of the handle in the stack.

If no precedence relation holds between a pair of terminals (id id), then a syntactic error has been detected and an error recovery routine must be invoked.
Operator-precedence parsing algorithm

Input. An input string \( \omega \) and a table of precedence relations.

Output. If \( \omega \) is well formed, a skeletal parse tree, with a placeholder nonterminal E labeling all interior nodes; otherwise, an error indication.
Operator-precedence parsing algorithm

Method: Initially, the stack contains $ and the input buffer the string $\omega$.

1) set ip to pint to the first symbol of $\omega$.
2) Repeat forever
3) if $ is on top of the stack and ip points to $ then
   Return
4) else begin
5) let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by ip;
6) if a $\leq$ b or a = b then begin
7)   push b onto the stack;
8)   advance ip to the next input symbol; end;
9) else if a $\cdot$ b then /*reduction*/
10)   Repeat
11)     pop the stack
12)   until the top stack terminal is related by $<$ to the terminal most recently popped
13) else blad();
end
Creating operator-precedence relations

1) If operator $\theta_1$ has higher precedence than operator $\theta_2$, make $\theta_1 \cdot > \theta_2$ and $\theta_2 < \cdot \theta_1$

For example, if * has higher precedence than +, make $* +> +$ and $+ \leftarrow *$.
These relations ensure that, in an expression of the form $E + E^* E + E$, the central $E^* E$ is the handle that will be reduced first.

2) If $\theta_1$ and $\theta_2$ are operators of equal precedence (they may in fact be the same operator), then make:

$\theta_1 \cdot > \theta_2$ and $\theta_2 \cdot > \theta_1$ – if the operators are left-associative, or
$\theta_1 < \cdot \theta_2$ and $\theta_2 < \cdot \theta_1$ – if the operators are right-associative.

For example, if + and - are left-associative, then make
$+ \cdot > +$, $+ \cdot > -$, $- \cdot > -$, $- \cdot > +$
If operator ↑ is right associative, then make, ↑ < · ↑

(These relations ensure that $E-E + E$ will have handle $E-E$ selected and $E \uparrow E \uparrow E$ will have the last $E \uparrow E$ selected $E \uparrow E$.)
Creating operator-precedence relations

3) Make

\[ \theta \prec \text{id} , \text{id} \succ \theta , \theta \prec ( , ( \succ \theta , \theta \succ ) ) , \theta \succ \$ \prec \theta \]

for all operators \( \theta \). Also let:

\[ ( = ) \quad \$ \prec ( \quad \$ \prec \text{id} \]
\[ ( \prec ( \quad \text{id} \succ \$ \quad ) \quad \prec \ \text{id} \]
\[ ( \prec \text{id} \quad \text{id} \succ ) \quad ) \quad \succ ) \]

These rules ensure that both \text{id} and \text{(E)} will be reduced to \( E \). Also \$, serves as both the left and right endmarker, causing handles to be found between $'s wherever possible $.
Operator-precedence relations

Assuming that:

1. Operator ↑ is of highest precedence and right-associative;

2. Operators * and / are of next highest precedence and left-associative, and

3. Operators + and - are of lowest precedence and left-associative.
Handling Unary Operators

We have a two kind of unary operator:

1) ¬ (logical negation) - which is not also a binary operator;

1) The sing – which is both a unary operator prefix (eg. -6), how and operator of two arguments.
Handling Unary Operators

¬ (logical negation) – this operator, we can incorporate it into the above scheme for creating operator-precedence relations.

A. Supposing ¬ to be a unary prefix operator, we make

\[ \theta < \cdot \neg \]

for any operator \( \theta \), whether unary or binary;

We make ¬ has higher precedence than \( \theta \), then:

\[ \neg \cdot \theta \]

and if not:

\[ \neg < \cdot \theta \]

B. The rule for unary postfix operators is analogous.
Handling Unary Operators

¬ (logical negation) – example

If ¬ has higher precedence than &, and & is left-associative, we would group:

\[ E \& \neg E \& E \]

as:

\[ ( E \& (\neg E) ) \& E \]

by the rules.
Handling Unary Operators

The sign - is both unary prefix (eg. -6) and binary infix.

Even if we give unary and binary minus the same precedence the table1, will fail to parse strings like id*-id correctly.

The best approach in this case is to use the lexical analyzer to distinguish between unary and binary minus, by having it return a different token when it sees unary minus. Unfortunately, the lexical analyzer cannot use lookahead to distinguish the two; it must remember the previous token.

In Fortran, for example, a minus sign is unary if the previous token was an operator, a left parenthesis, a comma, or an assignment symbol.
Precedence function

Compilers using operator-precedence parsers need not store the table of precedence relations.

In most cases, the table can be encoded by two precedence functions $f: T \rightarrow \mathbb{Z}$ and $g: T \rightarrow \mathbb{Z}$ that map terminal symbols to integers. We attempt to select $f$ and $g$ so that, for symbols:

- $f(a) < f(b)$, if $a < b$
- $f(a) = f(b)$, if $a = b$
- $f(a) > f(b)$, if $a > b$

Thus the precedence relation between $a$ and $b$ can be determined by a numerical comparison between $f(a)$ and $g(b)$. 
Precedence function

REMARKS.
Note, however, that error entries in the precedence matrix are obscured, since one of (1), (2), or (3) holds no matter what \(f(a)\) and \(g(b)\) are.
The loss of error detection capability is generally not considered serious enough to prevent the using of precedence functions where possible; errors can still be caught when a reduction is called for and no handle can be found.
Not every table of precedence relations has precedence functions to encode it, but in practical cases the functions usually exist.
We have an example \( f(id) < g(id) \) and \( f(*) = 4 < 5 = g(id) \).

Note that \( f(id) > g(id) \), suggests that but, in fact, no precedence relation holds between \( id \) and \( id \).
Algorithm for finding precedence functions

Input: An operator precedence matrix.

Output: Precedence functions representing the input matrix, or an indication that none exist.

Method:
Algorithm for finding precedence functions

STEP I.
Create symbol $f_a$ and $g_a$ for each $a$ that is a terminal or $\$$. 

STEP II.
Partition the created symbols into as many groups as possible, in such a way that: if $a=b$, then $f_a$ and $f_b$ are in the same group.

(Note that we may have to put symbols in the same group even if they are not related by $G$.

For example, if $a=b$ and $c=b$, then $f_a$ and $f_c$ must be in the same group, since they are both in the same group as $g_b$.

If, in addition $c=d$, then $f_a$ and $g_d$ and $x_d$, are in the same group even though $a=d$ may not hold.
Algorithm for finding precedence functions

STEP III.
Create a directed graph whose nodes are the groups found in STEP II. For any a and b:
- if \( a < b \) place an edge from the group of \( g_b \) to the group of \( f_a \).
- if \( a > b \), place an edge from the group of \( f_a \) to that of \( g_b \).
Note that an edge or path from \( f_a \) do \( g_b \) means that \( f(a) \) must exceed \( g(b) \); a path from \( g_b \) to \( f_a \) means that \( g(b) \) must exceed \( f(a) \).

STEP IV.
If the graph constructed in step III has a cycle, then no precedence functions exist.
If there are no cycles, let \( f(a) \) be the length of the longest path \( f(a) \) beginning at the group of \( f_a \);
Let \( g(a) \) be the length of the longest path from the group of \( g_a \).
Operator-precedence parsing

Create relations of operator-precedence

```
id  +  *  $
```

```
id
id  .>  .>  .>
```

```
+  <·  .>  <·  .>
```

```
*  <·  .>  .>  .>
```
Operator-precedence parsing

- There are no cycles, so precedence functions exist;
- As $f_s$ and $g_s$ have no out-edge $f(\$)=g(\$)=0$;
- The longest path from $g_+$ has length 1, so $g(\+)=1$;
- There is a path from $g_{id}$ to $f_*$ to $g_*$ to $f_+$ to $g_+$ to $f_+$, so $g(id)=5$. 

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>*</th>
<th>Id</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Error recover

There are two points in the parsing process at which an operator-precedence parser can discover syntactic errors:

1. If no precedence relation holds between the terminal on top of the stack and the current input.
2. If a handle has been found, but there is no production with this handle as a right side;
Error recover

UWAGI.
Recall that the operator-precedence parsing algorithm appears to reduce handles composed of terminals only. However, while nonterminals are treated anonymously, they still have places held for them on the parsing stack.

Thus when we talk in (2) above about a handle matching a production's right side, we mean that the terminals are the same and the positions occupied by nonterminals are the same.
**Error recover**

**Remarks.**
Just because we find a sequence of symbols $a < b_1 = b_2 = ... = b_k$, on the stack, however, does not mean that $b_1 b_2 ... b_k$ is the string of terminal symbols on the right side of some production. We did not check for this condition in table1, but we clearly can do so, and in fact we must do so if we wish to associate semantic rules with reductions.

Thus we have an opportunity to detect errors in table1, modified at steps (10-12) to determine what production is the handle in a reduction.
We may divide the error detection and recovery routine into several pieces. One piece handles errors of type (2). This routine might pop symbols off the stack just as in steps (10-12) of algorithm.

However, as there is no production to reduce by, no semantic actions are taken; a diagnostic message is printed instead.

To determine what the diagnostic should say, the routine handling case (2) must decide what production the right side being popped "looks like."
**Error recover**

Example:
Suppose abc is popped, and there is no production right side consisting of \(a, b, c\) together with zero or more nonterminals.

Then we might consider if deletion of one of \(a, b\) and \(c\) yields a legal right side (nonterminals omitted).

For example, if there were a right side \(aE\cdot c\cdot E\), we might issue the diagnostic:

*illegal \(b\) on line* (line containing \(b\))
Error recover

Example:
Suppose \texttt{abc} is popped, and there is no production right side consisting of \texttt{a,b,c} together with zero or more nonterminals.

We might also consider changing or inserting a terminal. Thus if \texttt{abEdc} were a right side, we might issue a diagnostic:

\begin{verbatim}
missing d on line (line containing c)
\end{verbatim}

We may also find that there is a right side with the proper sequence of terminals, but the wrong pattern of nonterminal. If \texttt{abc} is popped off the stack with no intervening \texttt{w} surrounding nonterminals, and \texttt{abc} is not a right side but \texttt{aEbc k} is, we might issue a diagnostic:

\begin{verbatim}
missing E on line (line containing b)
\end{verbatim}

Here \texttt{E} stands for an appropriate syntactic category represented by nonterminal \texttt{E}.
In general, the difficulty of determining appropriate diagnostics when no legal right side is found depends upon whether there are a finite or infinite number of possible strings that could be popped in lines (10-12) of algorithm. Any such string $b_1b_2\ldots b_k$, must have $=$ relations holding between adjacent symbols, so $b_1 = b_2 = \ldots = b_k$.

If an operator precedence table tells us that there are only a finite number of sequences of terminals related by $=$, then we can handle these strings on a case-by-case basis. For each such string $x$ we can determine in advance a minimum-distance legal right side $y$ and issue a diagnostic implying that $x$ was found when $y$ was intended.
Error recover

It is easy to determine all strings that could be popped from the stack in steps (10-12) of algorithm. These are evident in the directed graph whose nodes represent the terminals, with an edge from $a$ to $b$ if and only if $a=b$.

Then the possible strings are the labels of the nodes along paths in this graph. Paths consisting of a single node are possible. However, in order for a path $b_1b_2...b_k$ to be "poppable" on some Input:

- there must be a symbol $a$ (possibly $\$) such that $a<· b_1 = b_2 =... =b_k$ ($b_1$ we called initial)
- there must be a symbol $c$ (possibly $\$) such that $b_k>=c$ ($b_k$ we called finaly).

Only then could a reduction be called for and $b_1b_2...b_k$ be the sequence of symbols popped.

If the graph has a path from an initial to a final node containing a cycle, then there are an infinity of strings that might be popped; otherwise, there are only a finite number.
Error recover

We remember of grammar:

\[ E \rightarrow E + E \mid E - E \mid E \times E \mid E / E \mid E^{↑} E \mid (E) \mid -E \mid id \]

Tablica priorytetów

Graph of precedence table

![Graph of precedence table](image)
Specifically, the checker does the following:

If \(+, -, *, /\) or ↑, is reduced, it checks that nonterminals appear on both sides. If not, it issues the diagnostic:

```
missing operand
```

If \(id\) is reduced, it checks that there is no nonterminal to the right or left. If there is, it can warn:

```
missing operand
```

If \((\) is reduced, it checks that there is a nonterminal between the parentheses, if not, it can say

```
no expression between parentheses
```

Also it must check that no nonterminal appears on either side of the parentheses. If one does, it issues the same diagnostic:

```
missing operand
```
End of lecture ninth