Top-Down Parsing

Parsing:

- Context-free syntax is expressed with a context-free grammar.
- The process of discovering a derivation for some sentence.

Recursive-Descent Parsing

- 1. Construct the root with the starting symbol of the grammar.
- 2. Repeat until the fringe of the parse tree matches the input string:
  - Assuming a node labelled A, select a production with A on its left-hand-side and, for each symbol on its right-hand-side, construct the appropriate child.
  - When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack.
  - Find the next node to be expanded.

The key is picking the right production in the first step: that choice should be guided by the input string.

Example: Parse *x-2*y*

Example:

1. *Goal* $\rightarrow$ *Expr*
2. *Expr* $\rightarrow$ *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. *Term* $\rightarrow$ *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. *Factor* $\rightarrow$ *number*
9. | *id*

Rule	Sentential Form	Input

Example: Parse *x-2*y*

Example:

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*

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
4	*Term + Term*	| x – 2*y
7	*Factor + Term*	| x – 2*y
9	*id + Term*	| x – 2*y
Fail	*id + Term*	x | – 2*y
Back	*Expr*	| x – 2*y
3	*Expr – Term*	| x – 2*y
4	*Term – Term*	| x – 2*y
7	*Factor – Term*	| x – 2*y
9	*id – Term*	| x – 2*y
Match	*id – Term*	x – | 2*y
7	*id – Factor*	x – | 2*y
9	*id – num*	x – | 2*y
Fail	*id – num*	x – 2 | *y
Back	*id – Term*	x – | 2*y
5	*id – Term * Factor*	x – | 2*y
7	*id – Factor * Factor*	x – | 2*y
8	*id – num * Factor*	x – | 2*y
match	*id – num * Factor*	x – 2* | y
9	*id – num * id*	x – 2* | y
match	*id – num * id*	x – 2*y |

Example: Parse *x-2*y*

Example:

1. *Goal* $\rightarrow$ *Expr*
2. *Expr* $\rightarrow$ *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. *Term* $\rightarrow$ *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. *Factor* $\rightarrow$ *number*
9. | *id*

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
2	*Expr + Term + Term*	| x – 2*y
2	*Expr + Term + Term + Term*	| x – 2*y
2	*Expr + Term + Term + … + Term*	| x – 2*y

- Wrong choice leads to non-termination!
- This is a bad property for a parser!
- Parser must make the right choice!

Left-Recursive Grammars

- **Definition**: A grammar is left-recursive if it has a non-terminal symbol A, such that there is a derivation $A \Rightarrow A\alpha$, for some string α.
- A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.

Eliminating left-recursion:

- In many cases, it is sufficient to replace $A \rightarrow Aa \mid b$ with $A \rightarrow bA'$ and $A' \rightarrow aA' \mid \varepsilon$
- **Example**:

$$Sum \rightarrow Sum{+}number \mid number$$

would become:

$$Sum \rightarrow number\ \ Sum'$$

$$Sum' \rightarrow {+}number\ \ Sum' \mid \varepsilon$$

Eliminating Left Recursion

Example:

1. *Goal* $\rightarrow$ *Expr*
2. *Expr* $\rightarrow$ *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. *Term* $\rightarrow$ *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. *Factor* $\rightarrow$ *number*
9. | *id*

Applying the transformation to the Grammar of the Example we get:

Expr $\rightarrow$ *Term Expr'*

Expr' $\rightarrow$ *+Term Expr'* | *– Term Expr'* | ε

Term $\rightarrow$ *Factor Term'*

Term' $\rightarrow$ **Factor Term'* | */ Factor Term'* | ε

(*Goal* $\rightarrow$ *Expr* and *Factor* $\rightarrow$ *number* | *id* remain unchanged)

Non-intuitive, but it works!

Where are we?

- We can produce a top-down parser, but:
  - if it picks the wrong production rule it has to backtrack.
- **Idea**: look ahead in input and use context to pick correctly.
- How much lookahead is needed?
  - In general, an arbitrarily large amount.
  - Fortunately, most programming language constructs fall into subclasses of context-free grammars that can be parsed with limited lookahead.

Predictive Parsing

- Basic idea:
  - For any production $A \to a \,|\, b$ we would like to have a distinct way of choosing the correct production to expand.
- *FIRST* sets:
  - For any symbol A, *FIRST(A)* is defined as the set of terminal symbols that appear as the first symbol of one or more strings derived from A.

E.g. $Expr \to Term\ Expr'$

$$Expr' \to +Term\ Expr' \;|\; -\ Term\ Expr' \;|\; \varepsilon$$
$$Term \to Factor\ Term'$$
$$Term' \to *Factor\ Term' \;|\; /\ Factor\ Term' \;|\; \varepsilon$$

($Goal \to Expr$ and $Factor \to number \,|\, id$

$FIRST(Expr') = \{+,-,\varepsilon\}$, $FIRST(Term') = \{*,/,\varepsilon\}$, $FIRST(Factor) = \{number, id\}$

The LL(1) property

- If $A \to a$ and $A \to b$ both appear in the grammar, we would like to have: $FIRST(a) \cap FIRST(b) = \varnothing$.
- This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

Left Factoring

What if my grammar does not have the LL(1) property?

Sometimes, we can transform a grammar to have this property.

Algorithm:

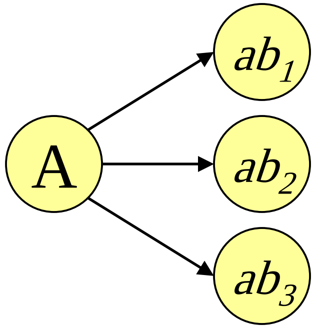
1. For each non-terminal A, find the longest prefix, say a, common to two or more of its alternatives

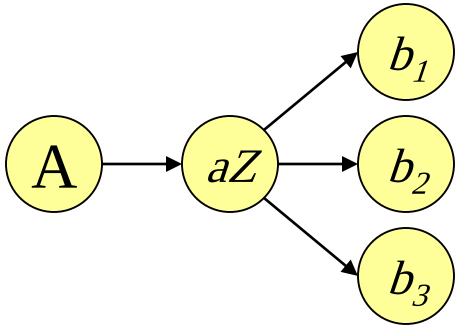
2. if $a \neq \varepsilon$ then replace all the A productions, $A \rightarrow ab_1 | ab_2 | ab_3 | \ldots | ab_n | \gamma$, where γ is anything that does not begin with a, with $A \rightarrow aZ \mid \gamma$ and $Z \rightarrow b_1 | b_2 | b_3 | \ldots | b_n$

Repeat the above until no common prefixes remain

Example: $A \rightarrow ab_1 \mid ab_2 \mid ab_3$ would become $A \rightarrow aZ$ and $Z \rightarrow b_1 | b_2 | b_3$

Note the graphical representation:





Example

$Goal \rightarrow Expr$

$\boldsymbol{Expr \rightarrow Term + Expr}$
$\boldsymbol{\quad | \; Term - Expr}$
$\boldsymbol{\quad | \; Term}$

$\boldsymbol{Term \rightarrow Factor * Term}$
$\boldsymbol{\quad | \; Factor / Term}$
$\boldsymbol{\quad | \; Factor}$

$Factor \rightarrow number$
$\quad | \; id$

We have a problem with the different rules for *Expr* as well as those for *Term.* In both cases, the first symbol of the right-hand side is the same (*Term* and *Factor*, respectively). E.g.:

$FIRST(Term) = FIRST(Term) \cap FIRST(Term) = \{number, id\}.$

$FIRST(Factor) = FIRST(Factor) \cap FIRST(Factor) = \{number, id\}.$

Applying left factoring:

$Expr \rightarrow Term\ Expr'$ $Expr' \rightarrow + Expr \mid - Expr \mid \varepsilon$	$FIRST(+)=\{+\}$; $FIRST(-)=\{-\}$; $FIRST(\varepsilon)=\{\varepsilon\}$; $FIRST(-) \cap FIRST(+) \cap FIRST(\varepsilon) = =\emptyset$
$Term \rightarrow Factor\ Term'$ $Term' \rightarrow * Term \mid / Term \mid \varepsilon$	$FIRST(*)=\{*\}$; $FIRST(/)=\{/\}$; $FIRST(\varepsilon)=\{\varepsilon\}$; $FIRST(*) \cap FIRST(/) \cap FIRST(\varepsilon) = =\emptyset$

Example (cont.)

1. *Goal* $\rightarrow$ *Expr*
2. *Expr* $\rightarrow$ *Term Expr´*
3. *Expr´* $\rightarrow$ *+ Expr*
4. | *- Expr*
5. | ε
6. *Term* $\rightarrow$ *Factor Term´*
7. *Term´* $\rightarrow$ ** Term*
8. | */ Term*
9. | ε
10. *Factor* $\rightarrow$ *number*
11. | *id*

The next symbol determines each choice correctly. No backtracking needed.

Rule	Sentential Form	Input

Example (cont.)

1. *Goal* → *Expr*
2. *Expr* → *Term Expr´*
3. *Expr´* → + *Expr*
4. | - *Expr*
5. | ε
6. *Term* → *Factor Term´*
7. *Term´* → * *Term*
8. | / *Term*
9. | ε
10. *Factor* → *number*
11. | *id*

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Term Expr´*	| x – 2*y
6	*Factor Term´ Expr´*	| x – 2*y
11	*id Term´ Expr´*	| x – 2*y
Match	*id Term´ Expr´*	x | – 2*y
9	*id* ε *Expr´*	x | – 2*y
4	*id – Expr*	x | – 2*y
Match	*id – Expr*	x – | 2*y
2	*id – Term Expr´*	x – | 2*y
6	*id – Factor Term´ Expr´*	x – | 2*y
10	*id – num Term´ Expr´*	x – | 2*y
Match	*id – num Term´ Expr´*	x – 2 | *y
7	*id – num * Term Expr´*	x – 2 | *y
Match	*id – num * Term Expr´*	x – 2* | y
6	*id – num * Factor Term´ Expr´*	x – 2* | y
11	*id – num * id Term´ Expr´*	x – 2* | y
Match	*id – num * id Term´ Expr´*	x – 2*y |
9	*id – num * id Expr´*	x – 2*y |
5	*id – num * id*	x – 2*y |

The next symbol determines each choice correctly. No backtracking needed.

Conclusion

- Top-down parsing:
  - recursive with backtracking (not often used in practice)
  - recursive predictive
- Nonrecursive Predictive Parsing is possible too: maintain a stack explicitly rather than implicitly via recursion and determine the production to be applied using a table (Aho, pp.186-190).
- Given a Context Free Grammar that doesn’t meet the LL(1) condition, it is undecidable whether or not an equivalent LL(1) grammar exists.
- Next time: Bottom-Up Parsing