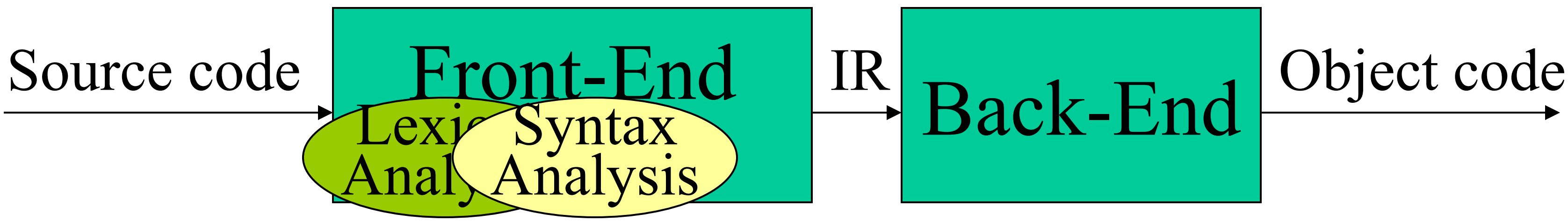
Lecture 8: Top-Down Parsing



Parsing:

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

Today's lecture:

Top-down parsing

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.

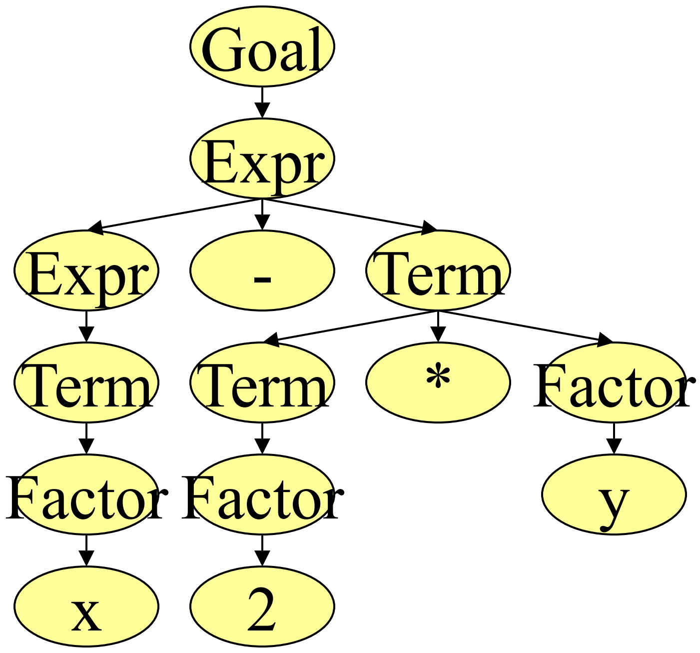
<u>Example:</u>

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

Example: Parse *x-2*y*

Steps (one scenario from many)

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 |



Other choices for expansion are possible:

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 $A \Rightarrow Aa$, for some string a.
- 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

Applying the transformation to the Grammar of the Example in Slide 2 we get:

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

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

Non-intuitive, but it works!

<u>General algorithm</u>: works for non-cyclic, no ε-productions grammars

1. Arrange the non-terminal symbols in order: $A_1, A_2, A_3, \ldots, A_n$

2. For i=1 to n do

for j=1 to i-1 do

I) replace each production of the form $A_i \rightarrow A_j \gamma$ *with*

the productions $A_i \rightarrow \delta_1\ \gamma \mid \delta_2\ \gamma \mid \ldots \mid \delta_k\ \gamma$

where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ *are all the current* A_j *productions*

II) eliminate the immediate left recursion among the A_i

Where are we?

- We can produce a top-down parser, but:
  - if it picks the wrong production rule it has to backtrack.
- **<u>Idea</u>**: 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 $A \rightarrow a \mid b$ 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 <u>the set of terminal symbols that appear as the first symbol of one or more strings derived from A</u>.

    E.g. (grammar in Slide 5): *FIRST(Expr')={+,-,ε}, FIRST(Term')={*,/,ε}, FIRST(Factor)={number, id}*
- The LL(1) property:
  - If $A \rightarrow a$ and $A \rightarrow b$ both appear in the grammar, we would like to have: $FIRST(a) \cap FIRST(b) = \emptyset$. This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

The Grammar of Slide 5 has this property!

Recursive Descent Predictive Parsing

(a practical implementation of the Grammar in Slide 5)

```
Main()
  token=next_token();
  if (Expr()!=false)
    then <next_compilation_step>
  else return false;

Expr()
  if (Term()==false)
    then result=false
  else if (EPrime()==false)
    then result=false
  else result=true
  return result

EPrime()
  if (token=='+' or '-') then
    token=next_token()
    if (Term()==false)
      then result=false
    elseif (EPrime()==false)
      then result=false
    else result=true
  else result=true /* ε */
  return result

Term()
  if (Factor()==false)
    then result=false
  else if (TPrime()==false)
    then result=false
  else result=true
  return result

TPrime()
  if (token=='*' or '/') then
    token=next_token()
    if (Factor()==false)
      then result=false
    else if (TPrime()==false)
      then result=false
    else result=true
  else result=true
  return result

Factor()
  if (token=='number' or 'id')then
    token=next_token()
    result=true
  else
    report syntax_error
    result=false
  return result
```

No backtracking is needed!

check :-)

Left Factoring

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

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

Algorithm:

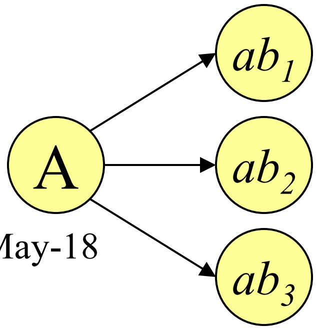
1. For each non-terminal A, find the longest prefix, say a, common to two or more of its alternatives

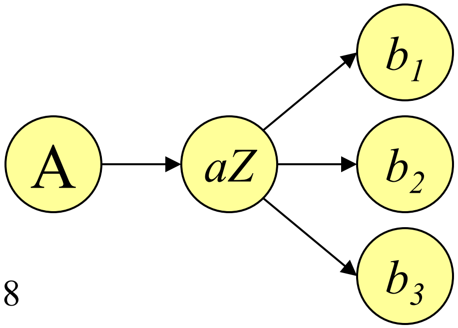
2. if $a \neq \varepsilon$ then replace all the A productions, $A \rightarrow ab_1|ab_2|ab_3|\ldots|ab_n|\gamma$, where γ is anything that does not begin with a, with $A \rightarrow aZ \mid \gamma$ and $Z \rightarrow b_1|b_2|b_3|\ldots|b_n$

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

(NB: this is a different grammar from the one in Slide 2)

$Goal \rightarrow Expr$

$\boldsymbol{Expr \rightarrow Term + Expr}$

$\boldsymbol{\quad \mid Term - Expr}$

$\boldsymbol{\quad \mid Term}$

$\boldsymbol{Term \rightarrow Factor * Term}$

$\boldsymbol{\quad \mid Factor \,/\, Term}$

$\boldsymbol{\quad \mid Factor}$

$Factor \rightarrow number$

$\quad \mid id$

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*

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.
- <u>Next time</u>: Bottom-Up Parsing
- <u>Reading</u>: Aho2, Sections 4.3.3, 4.3.4, 4.4; Aho1, pp. 176-178, 181-185; Grune pp.117-133; Hunter pp. 72-93; Cooper, Section 3.3.