Top-Down Parsing (Objectives)

- Given a grammar, the student will be able to convert the grammar to LL(1) form if possible.
- Given an LL(1) grammar, the student will be able to construct a corresponding predictive parser for the grammar

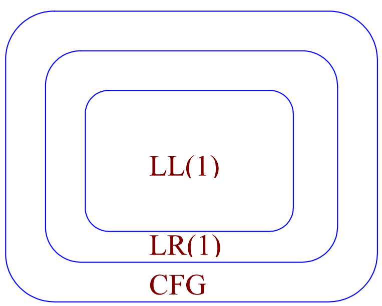
Context-Free Grammars

- A context-free grammar G is a quadruple, (N,T,P,S) where N is a set of nonterminals, T is a set of terminals, P is a set of productions and S∈N is the start symbol.
- Example

E → T + E
  | T - E
  | T
T → F * T
  | F / T
  | F
F → num
  | id
  | id [E]

N = {E, T, F}
T = {+, -, *, /, num, id}
S = E

Types of Context-Free Grammars



Top-Down Parsers

- Start at the root of the parse tree and fill in the children
  - expand the grammar from the start symbol
- pick a production and try to match the input token
- may require backtracking
  - if the picked production doesn't match the input at some point
- predictive recursive descent parsers do not require backtracking
- predictive parsers need LL(k) grammars
  - Left-to-right scan, Leftmost derivation, k symbols of lookahead
  - we will look at LL(1) grammars

Top-down Parsers w/o Lookahead

- To build a parse tree start with the root of the parse tree labeled with the start symbol
- Repeat the following steps until the left-to-right ordering of the leaves matches the input string
  1. At a node labeled A select a production with A on its lhs and for each symbol on its rhs construct a parse tree.
  2. When a terminal is added to a leaf of the parse tree that does not match the input string, backtrack up the tree to where a different choice could have been made that may lead to a correct derivation.
  3. Find the next node to be expanded and go to step 1.
- **Key**: select the right production in step 1.

Example Grammar

G →	E	(1)
E →	E + T	(2)
|	E - T	(3)
|	T	(4)
T →	T * F	(5)
|	T / F	(6)
|	F	(7)
F →	num	(8)
|	id	(9)
|	id [E]	(10)

- Parse the string x - 2

Problems with No Lookahead

- Termination
  - should not depend on the choice of production
  - parsers should always terminate
- Determinism
  - parsers should be deterministic
  - there should be only one choice at each step of the parse
- Solution
  - add lookahead to the parsing algorithm
  - fix grammar so that infinite loops are not possible

Writing an LL(1) Grammar

- eliminate infinite loops
  - make it impossible to recursively expand a grammar symbol either immediately or through a chain of expansions
  - eliminate left recursion
- eliminate ambiguity
  - make at most one choice for expanding each grammar symbol based upon the next input character
  - left factor the grammar

Eliminating Left Recursion

- A grammar is left recursive if

  $\exists A \in N \mid A \Rightarrow^{*} A\alpha$

  for some string α
- Eliminating immediate left recursion

  $A \rightarrow A\alpha \mid \beta$ becomes $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon$
- We must eliminate indirect left recursion too.

Algorithm

```
// grammar must have production S' → S
arrange non-terminals in some order
for i = 1 to n do
  for j = 1 to i-1 do // make sure no Aj in rhs
    replace each Ai → Ajγ by Ai → δ1γ | δ2γ | … | δkγ
      where Aj → δ1 | δ2 | … | δk
  end // only immediate recursion left for Ai
  eliminate immediate left recursion for Ai
end
```


Example

S → Aa
A → Bb
B → Sc | c

$$
\begin{aligned}
G &\rightarrow E \\
E &\rightarrow E + T \\
  &\mid E - T \\
  &\mid T \\
T &\rightarrow T * F \\
  &\mid T / F \\
  &\mid F \\
F &\rightarrow \text{num} \\
  &\mid \text{id} \\
  &\mid \text{id [E]}
\end{aligned}
$$

Left Factoring

- Restructure the grammar so FIRST sets of possible productions do not intersect

```
while a common prefix for alternatives exists for some A
  do
    find the longest prefix α common to two or more of
        A's alternatives
    replace productions A → αβ1 | αβ2 | … | αβn with
        A → αL
        L → β1 | β2 | … | βn
  end
```

Example

- Left factor the expression grammar

Left Recursion and Left Factoring

$G \to E$
$E \to E + T$
$\quad | \; E - T$
$\quad | \; T$
$T \to T * F$
$\quad | \; T / F$
$\quad | \; F$
$F \to num$
$\quad | \; id$
$\quad | \; id\,[E]$

$G \to E$
$E \to T\,E'$
$E' \to +\,T\,E'$
$\quad | \; -\,T\,E'$
$\quad | \; \varepsilon$
$T \to F\,T'$
$T' \to *\,F\,T'$
$\quad | \; /\,F\,T'$
$\quad | \; \varepsilon$
$F \to num$
$\quad | \; id\;\; F'$
$F' \to [E]$
$\quad | \; \varepsilon$

FIRST Sets

- For a production $A \to \alpha \,|\, \beta$ we would like a distinct way to choose the correct production.
  - Answer: use lookahead
- FIRST sets
  - For some rhs of a production α, FIRST(α) is the set of tokens that appear as the first symbol in some string derived from α

$$x \in FIRST(\alpha) \Leftrightarrow \exists x \in \Sigma \,|\, \alpha \Rightarrow^* x\gamma$$

FIRST Sets

- KEY PROPERTYS: Whenever two productions $A \to \alpha \,|\, \beta$ both appear in the same grammar
  1. $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
  2. At most one of α and β derives ε
  3. If $\beta \Rightarrow^* \varepsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A). (Similar for α)

  This allows the parser to make the right choice of productions with one symbol of lookahead
- The text book define predict set:
  - predict($A \to \alpha$) = FIRST(α) $\cup$ (if $\alpha \Rightarrow^* \varepsilon$ then Follow(A) else $\emptyset$)
- Use left factoring to try to obtain this property

Computing First Sets

- To build FIRST(X)

  if X is a terminal then
  FIRST(X) = {X}
  else if $X \rightarrow \varepsilon$ then
  FIRST(X) ∪= {ε}
  else if $X \rightarrow Y_1 Y_2 \ldots Y_k$ then
  FIRST(X) ∪= FIRST(Y_1)
  $\forall i \mid \varepsilon \in \text{FIRST}(Y_j),\ 1 \le j < i,\ \text{FIRST}(X) \cup= \text{FIRST}(Y_i)$

Example

- Compute FIRST sets for the expression grammar

FOLLOW Sets

- To construct FOLLOW sets:
  1. place \$ in FOLLOW(S')
  2. for $A \rightarrow \alpha B \beta$ add FIRST(β)-{ε} to FOLLOW(B)
  3. for $A \rightarrow \alpha B$ add FOLLOW(A) to FOLLOW(B)
  4. for $A \rightarrow \alpha B \beta$ if $\varepsilon \in$ FIRST(β) add FOLLOW(A) to FOLLOW(B)

Example

- Compute FOLLOW sets for the expression grammar

First and Follow Sets

	First	Follow
G	{num, id}	{$}
E	{num, id}	{$,]}
E'	{+, -, ε}	{$,]}
T	{num, id}	{$, + , -,]}
T'	{*, /, ε}	{$, + , -,]}
F	{num, id}	{$, + , -, *, /,]}
F'	{ [, ε}	{$, + , -, *, /,]}

LL(1)?

- Show that the converted grammar is LL(1)

Predictive Parser

- Construct a top-down predictive parser
  - Each non-terminal becomes a function
  - When a terminal is the first symbol on the rhs of a production, match that terminal with the next input symbol

Predictive Parser

```
G() {
    token = NextToken()
    if (E() == ERROR) then
      return ERROR;
    }
E() {
    if (T() == ERROR)
      return ERROR;
    else return E'();
    }
T() {
    if (F() == ERROR) then
      return ERROR
    else return T'();
    }
```

```
F() {
    if (token == NUM ||
        token == ID) then {
          ptoken = token;
          token = NextToken();
          if (ptoken == NUM) then
            return OK;
          else return F'(); }
    else return ERROR;}
E'() {
    if (token == PLUS ||
        token == MINUS) then {
      token = NextToken();
      if (T() == ERROR) then
        return ERROR
      else
        return E'();}
    else return OK;
    }
// T' is similar
```