Pushdown Automata

Pushdown automata (PDAs) have the same expressive power as CFGs (i.e., they accept CFLs).

A pushdown automaton is like an NFA but with an additional “memory stack” which can hold sequences of symbols from a memory alphabet.

Automaton scans an input from left to right - at each step it may push a symbol onto the stack, or pop the stack. It cannot read other elements of the stack.

Start with empty stack; accept if at end of string state is in subset \( T \subseteq Q \) of accepting states and stack is empty.

Transitions

The notation \( \mathcal{P} = (Q, A, M, \delta, i, T) \) where \( M \) is stack alphabet and \( \delta \) is transition function.

Action taken by machine is allowed to depend on top element of stack, input letter being read, and state. Action consists of new state, and possibly push/pop the stack.

Formally:
\[
\delta : Q \times (A \cup \{\varepsilon\}) \times (M \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q \times (M \cup \{\varepsilon\}))
\]

i.e. for each combination of state, letter being read, and topmost stack symbol, we are given a set of allowable new states, and actions on stack.

For example:

\[
\delta(q, a, m) = \{(q_2, m_2), (q_3, m_3)\}
\]

means that in state \( q \), if you read \( a \) with \( m \) at top of stack, you may move to state \( q_2 \) and replace \( m \) with \( m_2 \). Alternatively you may move to state \( q_3 \) and replace \( m \) with \( m_3 \).

\[
\delta(q, a, \varepsilon) = \{(q_2, m_2)\}
\]

means in state \( q \) with input \( a \), go to state \( q_2 \) and push \( m_2 \) on top of stack.
Example: Palindromes

Input alphabet $A = \{a, b, c\}$
Use stack alphabet $M = \{a', b', c'\}$

states $Q = \{f, s\}$ ($f$ is “reading first half”, $s$ is “reading second half”)
Initial state $f$.
Accepting states $T = \{s\}$

Transitions:
\[
\delta(f, a, \epsilon) = \{(f, a'), (s, \epsilon), (s, a')\}
\delta(f, b, \epsilon) = \{(f, b'), (s, \epsilon), (s, b')\}
\delta(f, c, \epsilon) = \{(f, c'), (s, \epsilon), (s, c')\}
\delta(s, a, a') = \{(s, \epsilon)\}; \delta(s, b, b') = \{(s, \epsilon)\}; \delta(s, c, c') = \{(s, \epsilon)\};
\delta(\text{anything else}) = \emptyset
\]

Example of a Deterministic PDA

Consider palindromes over $\{a, b, c\}$ which contain exactly one $c$.
Use stack alphabet $M = \{a', b'\}$

states $Q = \{f, s\}$ ($f$ is “reading first half”, $s$ is “reading second half”)
Initial state $f$, accepting states $T = \{s\}$

Transitions:
\[
\delta(f, a, \epsilon) = (f, a')
\delta(f, b, \epsilon) = (f, b')
\delta(f, c, \epsilon) = (s, \epsilon)
\delta(s, a, a') = (s, \epsilon); \delta(s, b, b') = (s, \epsilon); \delta(\text{anything else}) \text{ is undefined (reject input)}.\]

An Accepting Computation

PDA to recognise “well-formed” strings of parentheses
- A single state $s$ (accepting)
- Input alphabet $\{(, )\}$
- Memory alphabet $\{x\}$

\[
\delta(s, (, \epsilon) = \{(s, x)\}
\delta(s, ), x) = \{(s, \epsilon)\}
\]

Comments
- The number of $x$’s on the stack is the number of (’s read so far minus number of )’s read.
PDA’s and CFG’s...

...define the same set of languages. To prove this,

1. Given a CFL, construct a PDA that accepts it
2. and given a PDA, construct a CFG that defines the language

of that PDA.

We define “extended PDA”, a generalisation of PDA. But the
extension will not allow extra languages to be accepted. Then we
can show equivalence between extended PDA’s and Greibach
normal form grammars.

Extended PDAs

Allow transitions that write more than 1 memory symbol to the
stack.

e.g.

\[ \delta(s, a, X) = \{(t, UVW)\} \]

meaning: in state s, with input a and X on top of stack, change to
state t, replace the X with UVW.

This could be replaced with:

\[ \delta(s, a, X) = \{(s', \epsilon)\} \]

\[ \delta(s', \epsilon, \epsilon) = \{(s'', \epsilon)\} \]

\[ \delta(s'', \epsilon, \epsilon) = \{(t, \epsilon)\} \]

Converting GNF grammar to Ext. PDA

Use two states: initial state \( i \), accepting state \( t \)
Variable symbols in grammar become the elements of the stack
alphabet
Let \( S \) denote start symbol of grammar. Rules such as

\[ S \rightarrow a TUVW \]

become transitions

\[ \delta(i, a, \epsilon) = \{(t, TUVW), \ldots\} \]

Rules such as

\[ X \rightarrow a TUVW \]

become transitions

\[ \delta(t, a, X) = \{(t, TUVW), \ldots\} \]

Finally, to allow the empty string to belong to the language
accepted by the PDA (when the grammar has the production
\( S \rightarrow \epsilon \)):

Include transition

\[ \delta(i, \epsilon, \epsilon) = \{(t, \epsilon)\} \]

So, to translate from a CFG to a PDA:

1. Convert grammar to Greibach normal form
2. Convert to extended PDA
3. Convert ext. PDA to a standard PDA
Find a PDA that accepts “valid” expressions that use (,),+,x
e.g.
\[
  x + x + (x + x) \\
  (x + (x + x))
\]
Let us disallow consecutive pairs of nested parentheses e.g.
\[
  x + ((x + x))
\]
also disallow singleton x in a pair of parentheses e.g.
\[
  x + (x) + x
\]

Grammar

Start symbol E. Variable symbol F represents an expression with
no matching pair of parentheses on the outside.
\[
  E \rightarrow x \\
  E \rightarrow (F) \\
  E \rightarrow E + E \\
  F \rightarrow E + E
\]
(We don’t have e.g. \( F \rightarrow x \), since \((x)\) was disallowed as a
substring.)
Need to convert to GNF.

Conversion to Greibach Normal Form

Eliminate leading variables on RHS’s of productions. Luckily this is
fairly simple (for this particular grammar), but we need to be
careful in arguing new grammar is equivalent.
\[
  E \rightarrow x \\
  E \rightarrow (F) \\
  E \rightarrow (F) + E \\
  E \rightarrow x + E \\
  F \rightarrow x + E \\
  F \rightarrow (F) + E \\
  F \rightarrow (F) + E + x \\
  F \rightarrow x + E + E
\]
The last 2 F rules are redundant; we can remove them.

We now have the grammar
\[
  E \rightarrow x \mid (F) \mid (F) + E \mid x + E \\
  F \rightarrow x + E \mid (F) + E
\]
Now introduce some variables to represent non-leading alphabet
symbols
\[
  X_j \rightarrow ) \\
  X_+ \rightarrow +
\]
\[
  E \rightarrow (FX_j \mid x \mid (FX_j X_+ E \mid x X_+ E \\
  F \rightarrow x X_+ E \mid (FX_j X_+ E
\]
This is now in Greibach Normal Form.
Convert to (extended) PDA

states $Q = \{i, t\}$  (initial state, accepting state)
Input alphabet $A = \{(), +, x\}$
Memory alphabet $M = \{X, X+, E, F\}$

Transitions:
- $\delta(i, (, \epsilon) = \{(t, FX)\}$
- $\delta(i, x, \epsilon) = \{(t, X, E)\}$
- $\delta(t, X) = \{(t, \epsilon)\}$
- $\delta(t, +, X+) = \{(t, \epsilon)\}$
- $\delta(t, x, X) = \{(t, X+E)\}$
- $\delta(t, X) = \{(t, FX)\}$
- $\delta(t, x, E) = \{(t, X, E)\}$
- $\delta(t, x, F) = \{(t, FX)\}$

A Computation

Conversion from PDA to CFG

The Associated Leftmost Derivation

\[
E \Rightarrow (FX) \\
\Rightarrow (x X, E) \\
\Rightarrow (x + E) \\
\Rightarrow (x + (FX)X) \\
\Rightarrow (x + (x X, E)X) \\
\Rightarrow (x + (x + E)X) \\
\Rightarrow (x + (x + x) X)
\]

Grammar:

\[
E \rightarrow (FX) \mid x \mid FX X \mid X E \\
F \rightarrow x X E \mid (FX) X E \\
X \rightarrow x
\]

Problem: PDAs constructed from CFGs have just 2 states \{i, t\}, with the PDA only in state i at the start, afterwards always in state t.

How do we convert from a PDA with \(>2\) states?

General idea: convert it to an extended PDA with 2 states as above. We are arguing that anything we can do with a PDA can be done with a 2-state extended PDA.

(Alternative conversion to CFG is also possible)
Converting to a 2-state Extended PDA

Use a bigger memory alphabet, where each symbol represents a symbol from original memory alphabet, together with a state and a letter.

Technically, if $\mathcal{P} = (Q, A, M, \delta, i, T)$ we define

$$\mathcal{P}' = ([i, t], A, M \times Q \times A, \delta', i, \{t\})$$

Need to define $\delta'$, the new transition function.

Constructing new transition function

For example:

Suppose $\delta(q, a, m) = \{(q', m'), \ldots\}$. Then we would say, for all $a' \in A$,

$$\delta'(i, a, (m, q, a')) = \{(t, (m', q', a')), \ldots\}$$

$\delta'(t, a, (m, q, a')) = \{(t, (m', q', a')), \ldots\}$

If we have $\delta(q, a, \epsilon) = \{(q', m')\}$, then for all $m \in M, a' \in A$

$$\delta'(t, a, (m, q, a')) = \{(t, (m', q', a'')(m, q, a')), \ldots\}$$

for any $a''$ satisfying $\delta(q', a'', m') = \{(q, \epsilon), \ldots\}$.

The Pumping Lemma for CFLs

Given a CFL, any sufficiently long string in that CFL has two substrings (at least one of which is non-empty) such that if both of these substrings are “pumped” you generate further words in that CFL.

More formally...

Given a CFL $L$, there exists a number $N$ such that any string $\alpha \in L$ with $|\alpha| \geq N$ can be written as

$$\alpha = sxtyu$$

such that $sx^2ty^2u, sx^3ty^3u, sx^4ty^4u, \ldots$ are all members of $L$, $|xty| \leq N$ and $|xy| > 0$ (which we need for all strings in this collection to be distinct).

How do we find suitable substrings $x$ and $y$?

Consider the following (Chomsky normal form) grammar

$$S \rightarrow XY$$

$$U \rightarrow a$$

$$V \rightarrow ZX | a | b$$

$$X \rightarrow VW | a$$

$$Y \rightarrow b | c$$

$$Z \rightarrow a | c$$

$$W \rightarrow UZ | b$$

The string cabaab belongs to the language. Also it contains "suitable substrings" $x$ and $y$ which we can find by looking at a derivation tree.
We find two X’s on the same path. We can say:

\[ X \Rightarrow^* cXaa \]

(via the sequence \( X \Rightarrow VW \Rightarrow ZXW \implies cXW \Rightarrow cXUZ \Rightarrow cXaZ \Rightarrow cXaa \))

Generally: \( X \Rightarrow^* cXaaaa \Rightarrow^* cccXaaaaa \ldots \)

The substrings \( c \) and \( aa \) can be pumped, because a derivation of \( cabaab \) can go as follows:

\[
\begin{align*}
S & \Rightarrow Xb \\
& \Rightarrow^* cXaab \\
& \Rightarrow^* cabaab 
\end{align*}
\]

but the underlined \( X \) could have been used to generate extra \( c \)'s and \( aa \)'s on each side of it.

Example

Prove the following is not a CFL:

\{ 1, 101, 101001, 1010010001, 10100100010001, \ldots \} 

Proof by contradiction: suppose string \( s \) (in above set) has substrings \( x \) and \( y \) that can be pumped.

\( x \) and \( y \) must contain at least one \( 1 \), or else all new strings generated by repeating \( x \) and \( y \) would have same number of \( 1 \)'s, a contradiction.

But if \( x \) contains a \( 1 \), then any string that contains \( x^3 \) as a substring must have 2 pairs of \( 1 \)'s with the same number of \( 0 \)'s between them, also a contradiction.
Another example

Prove the following language is not a CFL: Let \( L \) be words of the form \( ww \) (where \( w \) is any word over \( \{a, b, c\} \))
(e.g. \( aa \), \( abcabc \), \( baaabaaa \), ...)  
Let \( N \) be "sufficiently large" word length promised by pumping lemma.  
Choose \( w = a^{N+1}b^{N+1}a^{N+1}b^{N+1} \), so \( w \in L \)  
We can argue that there is no subword of \( w \) of length \( N \) which contains any pair of subwords which, if repeated once, give another member of \( L \).

Parsers, Parser generators

As noted, brute-force parsing of a string with respect to CFG \( G \) is (nearly always) impractical.  
We would not want to implement an unrestricted PDA. A deterministic PDA (DPDA) would be OK. (we elaborate on this later)  
A deterministic context-free language (DCFL) is a CFL that can be recognised by a DPDA.  
We look at two kinds of DPDA: LL(1) parser and LR(1) parser (and generalisations: LL(k), LR(k) for positive integers \( k \)) These parsers have the property that they can be generated automatically from given grammars.  
A parser generator is a program that takes a grammar as input and returns a parser (usually one of the kinds named above)  
Examples: javacc – a LL parser generator  
javaCUP – a LR parser generator

Parser Generation

The problem: given a grammar \( G \), convert it to a parser (ie an algorithm that finds a derivation of a string generated by \( G \)). We will look at parser generators that generate LL and LR parsers. Not all CFGs, indeed not all CFLs, have LL/LR parser generators. But you may be able to modify the grammar, and most (nearly all) programming languages have appropriate grammars.  
A brute-force parser is always possible, but we ask for an algorithm that is efficient in the sense that the time taken to find a derivation of a string of length \( n \) should be proportional to \( n \), ie \( \Theta(n) \).
If we can
- convert to a deterministic pushdown automaton
- interpret each transition of DPDA as a derivation step
then we are done because it is not hard to see that the time taken
to run a DPDA on a string of length \( n \) is indeed \( \Theta(n) \).

**LL(1) parser**

LL(1): left-to-right scan, leftmost derivation, one symbol lookahead.

Let \( G \) be the grammar of interest
Scan input word \( w \) from left to right
Each time a new letter (element of alphabet \( A \)) is scanned, you have to be able to identify which production in the grammar has been used to generate that letter
Let \( w' \) be the suffix of \( w \) reached at some stage
That production cannot necessarily be applied in reverse to \( w' \) - use a stack of variables (elements of \( V \)) and letters in \( A \) to keep track of what \( w' \) has to be derivable from.

**The two kinds of (efficient) parser**

LL(k): left-to-right scan, leftmost derivation, \( k \) symbols of lookahead
LR(k): left-to-right scan, rightmost derivation, \( k \) symbols of lookahead
Usually we use 1 symbol of lookahead, so that LL and LR can usually be taken to mean LL(1) and LR(1) respectively.
Observe that “left-to-right scan” is descriptive of pushdown automaton. We observed that palindromes need a general ie. non-deterministic PDA to recognise them. Palindromes can be efficiently recognised, but not by scanning from left to right with a DPDA.

**Example**

\[
\begin{align*}
S & \rightarrow a X \mid b Y \\
X & \rightarrow c Z c c Z \\
Y & \rightarrow a X \\
Z & \rightarrow d X \mid e
\end{align*}
\]

Given a grammar symbol to be parsed and an alphabet symbol (the lookahead symbol, ie the next symbol in a word that is being scanned) we want to know which rule to apply. We can write down a table:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( S \rightarrow a X )</td>
<td>( S \rightarrow b Y )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( X )</td>
<td></td>
<td></td>
<td>( X \rightarrow c Z c c Z )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Y )</td>
<td>( Y \rightarrow a X )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Z )</td>
<td></td>
<td></td>
<td></td>
<td>( Z \rightarrow d X )</td>
<td>( Z \rightarrow e )</td>
</tr>
</tbody>
</table>
Apply the table to problem of parsing input:

bacdcecece

stack represents the remaining stuff to be parsed (the unscanned portion of the input)

<table>
<thead>
<tr>
<th>rule</th>
<th>stack</th>
<th>rest of input</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → bY</td>
<td>bY bacdcecece</td>
<td>bacdcecece</td>
</tr>
<tr>
<td>Y → aX</td>
<td>aX acdcecece</td>
<td>acdcecece</td>
</tr>
<tr>
<td>X → cZccZ</td>
<td>cZccZ dcecece</td>
<td>dcecece</td>
</tr>
</tbody>
</table>

continued...

input: bacdcececece

rule stack rest of input

S bacdcececece
S → bY bacdcececece
Y → aX aX acdcececece
X → cZccZ cZccZ dcececece
Z → dX dXccZ dcececece
X → cZccZ cZccZccZ ececece
Z → e eccZccZ ececece
Z → e eccZ ecece
Z → e e e ecece

...we are done. The sequence of rules down the first column gives a leftmost derivation.

Things to note:

The leftmost derivation is:

S ⇒ bY ⇒ baX ⇒ bacZccZ ⇒ bacdXccZ
⇒ bacdZccZccZ ⇒ bacdceccZccZ ⇒ bacdcecececcZ
⇒ bacdcececececece

The stack contains a sequence of grammar symbols which is supposed to derive the remainder of the input.

The algorithm is a top-down approach - the stack initially contains the initial symbol of the grammar, and we use the first letter of the string to tell us which rule to apply first.

Example 2

Prolog terms composed from the functors \( f(\cdot, \cdot) \) and \( g(\cdot) \) and atoms \( a, b, c \).

Examples

a
f(a, g(a, b))
g(f(a, g(a, b)))
f(f(c, g(a)), g(a, b))

Grammar: alphabet \{ a, b, c, f, g(\cdot), \cdot \}, variables \{ T \} (necessarily \( T \) is the start symbol).

Productions:

\( T \rightarrow f(T, T) \mid g(T) \mid a \mid b \mid c \)
Example 2 (notes on parse table)

meaning that when I read a I assume it was generated by production $T \to a$ and so in order to get rid of that a I need to reverse that production on it. Easy to see in this example that in fact a can only be generated by a usage of $T \to a$ (and similarly for other symbols). The $T$ labelling the row says that we apply an appropriate rule when $T$ is at the top of the stack. If a constant is at top of stack we have to just check it’s equal to head of string.

LL(1) parser generation

The problem: need general method for inserting grammar rules into table entries.

define “FIRST” sets, “FOLLOW” sets and “nullable”, all features of a variable symbol.

Let $X$ be a variable symbol.

- $\text{FIRST}(\alpha)$ is the set of all constants (i.e. grammar symbols other than variables) that can begin any string derived from $\alpha$.
- $\text{FOLLOW}(X)$ is the set of all constants that can come after $X$ is a string derivable from the start symbol.
- $X$ is nullable if $X \Rightarrow^* \epsilon$, i.e. $X$ can derive the empty string.

If we can work out what these sets are for all variable symbols, then we have a rule for inserting rules into parse table.

Example 2 (continued)

Productions:

$T \rightarrow f(T,T) \mid g(T) \mid a \mid b \mid c$

LL(1) parse table has rows labelled by variables and columns labelled by constants:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$T \rightarrow a$</td>
<td>$T \rightarrow b$</td>
<td>$T \rightarrow c$</td>
<td>$T \rightarrow f(T,T)$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow g(T)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$T \Rightarrow f(g(g(a)), b)$

where $f$ is lookahead token

$f(T,T) \Rightarrow f(g(g(a)), b)$

2 constants on stack match first 2 symbols:

$T \rightarrow g(g(a)), b)$

now $g$ is lookahead token

$g(T), T) \Rightarrow g(g(a)), b)$

$T), T) \Rightarrow g(a)), b)$

$T), T) \Rightarrow a)), b)$

$a)), T) \Rightarrow a)), b)$

$T) \Rightarrow b)$

$b) \Rightarrow b)$

and we are done (the string is parsed).
FIRST and FOLLOW sets; nullable symbols

Building the parse table from them:
Extend definition of “nullable” to strings: \( \alpha \) is nullable if all symbols in it are nullable.
For each production \( X \rightarrow \alpha \), enter that production into row \( X \) column \( a \) if \( a \in \text{FIRST}(\alpha) \). Also, if \( \alpha \) is nullable, insert \( X \rightarrow \alpha \) into row \( X \) column \( a \) for each \( a \in \text{FOLLOW}(X) \).
If you do this and some entries contain more than one production, then the grammar was not LL(1) parsable.

How to compute them:
Initially FIRST and FOLLOW sets are all set to empty and nullable is set to false, for all variable symbols.
Next slide: “iterate until nothing happens” approach seen in DFA simplification method.

Example

Write down a LL(1) parse table for the following context-free grammar. \( \{S, A, B, C\}, \{a, b, c, d\}, P, S \) where \( P \) is the set of rules:
\[
S \quad \rightarrow \quad ABCd \\
A \quad \rightarrow \quad aS \mid B \\
B \quad \rightarrow \quad bCC \mid \epsilon \\
C \quad \rightarrow \quad cC \mid d
\]

Computing FIRST and FOLLOW sets; nullable symbols

for each terminal symbol \( a \), \( \text{FIRST}(a) = \{ a \} \)
repeat
- for each production \( X \rightarrow Y_1 Y_2 \ldots Y_k \)
  - if all the \( Y_i \) are nullable
    - then \( \text{nullable}(X) = \text{true} \)
  - if all of \( Y_1, \ldots, Y_{i-1} \) are nullable
    - then \( \text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(Y_i) \)
  - if all of \( Y_{i+1}, \ldots, Y_k \) are nullable
    - then \( \text{FOLLOW}(Y_i) = \text{FOLLOW}(Y_i) \cup \text{FOLLOW}(X) \)
if \( Y_{j+1}, \ldots, Y_{j-1} \) are all nullable
then \( \text{FOLLOW}(Y_j) = \text{FOLLOW}(Y_j) \cup \text{FIRST}(Y_j) \)
until FIRST, FOLLOW and nullable do not change in some iteration.

some (not all) analysis of example

Nullable symbols? \( B \Rightarrow \epsilon \) so \( B \) is nullable. \( A \Rightarrow \epsilon \) so \( A \) is nullable. \( S \) and \( C \) are not (look at RHS’s of their rules).
FIRST\((S)\)=\{a, b, c, d\}; FIRST\((A)\)=\{a, b\}; etc
FOLLOW\((S)\)=\{b, c, d\} (e.g. \( S \Rightarrow ABCd \Rightarrow aSBCd \Rightarrow aSbCCCd \), hence \( b \in \text{FOLLOW}(S) \).)
FOLLOW\((A)\) contains c and d

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td>a</td>
<td>B</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>a</td>
<td></td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>b</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>c</td>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>
Eliminating left recursion

Parser generation succeeds iff no entry to parse table contains more than one production. How to change non-LL(1) parsable into LL(1) parsable grammar?

Two examples (from A.W. Appel: *Modern compiler implementation in Java*):

Suppose a grammar contains the following rules

\[ E \rightarrow E + T \]
\[ E \rightarrow T \]

Any token in \( \text{FIRST}(T) \) with also be in \( \text{FIRST}(E + T) \). So given a stack with \( E \) at the top and one of the tokens at front of input, don’t know which rule to use. (The algorithm just given would insert both rules into the parse table entry.) Replace with

\[ E \rightarrow TE' \]
\[ E' \rightarrow T + E' \]
\[ E' \rightarrow \epsilon \]

Left factoring

Suppose a grammar contains the rules:

\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
\[ S \rightarrow \text{if } E \text{ then } S \]

Clearly we don’t know which to use if we want to parse an \( S \) and we see the next token is an “if”. Replace with:

\[ S \rightarrow \text{if } E \text{ then } S X \]
\[ X \rightarrow \epsilon \]
\[ X \rightarrow \text{else } S \]

this equivalent grammar fragment is LL(1) parsable.

LL(k) parsers

Each row is a variable symbol (as before); each row is a sequence of \( k \) constants. Hence the action taken at each step depends on the first \( k \) tokens in the input, as well as the top of the stack.

A simple generalisation of the LL(1) parser generator to LL(k) (with \( |A|^k \) columns in the parse table) would be slow; “local lookahead” with LL(1) is preferable.

Example

Prolog terms composed from the functors \( f(\cdot, \cdot) \) and \( g(\cdot) \) and atoms \( a, b, c, f, g \). (like earlier example but now \( f \) and \( g \) may be atoms.

Grammar: alphabet \{a, b, c, f, g, (, ,)\}, variables \{T\} (necessarily \( T \) is the start symbol).

Productions:

\[ T \rightarrow f(T, T) \mid g(T) \mid a \mid b \mid c \mid f \mid g \]

Suppose we want to parse a \( T \) at top of stack and we see an \( f \) at beginning of input. Do we think it was generated by \( T \rightarrow f \) or by \( T \rightarrow f(T, T) \)?

If we look at what comes next, we would know: If the next symbol is ( then we should use \( T \rightarrow f(T, T) \) otherwise we should use \( T \rightarrow f \).

(some parser generators a “local lookahead” facility for incorporating facts like that, alternatively we could use LL(2) parser, but you end up with a big parse table.)
Contrasting this with LR parsing (which I will not do in detail)

An LR parser is a DPDA where the stack contains a collection of symbols that can derive the prefix scanned so far (as opposed to symbols that can be used to derive the suffix). On reading a symbol, the symbol can simply be transferred to the stack. Sometimes a rule is applied in reverse to a sequence of stack symbols that correspond to a RHS of a grammar rule.