Solving systems of regular expression equations

Algorithm 2.1
Solving a set of regular expression equations in standard form.

Input. A set \( Q \) of regular expression equations in standard form over \( \Delta \), whose coefficients are regular expressions over alphabet \( \Sigma \). Let \( \Delta \) be the set \( \{X_1, X_2, \ldots, X_n\} \).

Output. A set of solutions of the form \( X_i = \alpha_i, 1 \leq i \leq n \), where \( \alpha_i \) is a regular expression over \( \Sigma \).

Method. The method is reminiscent of solving linear equations using Gaussian elimination.

Step 1: Let \( i = 1 \).

Step 2: If \( i = n \), go to step 4. Otherwise, using the identities of Lemma 2.1, write the equation for \( X_i \) as \( X_i = \alpha X_i + \beta \), where \( \alpha \) is a regular expression over \( \Sigma \) and \( \beta \) is a regular expression of the form \( \beta_0 + \beta_1 X_{i+1} + \cdots + \beta_n X_n \) with each \( \beta_j \) a regular expression over \( \Sigma \). We shall see that this will always be possible. Then in the equations for \( X_{i+1}, \ldots, X_n \), we replace \( X_i \) on the right by the regular expression \( \alpha^* \beta \).

Step 3: Increase \( i \) by 1 and return to step 2.

Step 4: After executing step 2 the equation for \( X_i \) will have only symbols in \( \Sigma \) and \( X_n, \ldots, X_i \) on the right. In particular, the equation for \( X_n \) will have only \( X_n \) and symbols in \( \Sigma \) on the right. At this point \( i = n \) and we now go to step 5.

Step 5: The equation for \( X_i \) is of the form \( X_i = \alpha X_i + \beta \), where \( \alpha \) and \( \beta \) are regular expressions over \( \Sigma \). Emit the statement \( X_i = \alpha^* \beta \) and substitute \( \alpha^* \beta \) for \( X_i \) in the remaining equations.

Step 6: If \( i = 1 \), end. Otherwise, decrease \( i \) by 1 and return to step 5. \( \square \)
Consider the following DFSM

For each transition from state $Q$ to state $R$ under input $a$, we create a production of the form $Q \rightarrow Ra$ and for each final state, $P$, a production of the form $P \rightarrow \varepsilon$. Productions lead to equations in the obvious way.

So, we have the equations

$A_1 = 1A_1 + 0A_2$

$A_2 = 0A_3 + 1a_1$

$A_3 = 0A_3 + 1A_1 + \varepsilon$
These equations can be solved as follows:
First solve for $A_3$

$$A_3 = 0^* 1A_1 + 0^*$$

Then replace $A_3$ in the equation for $A_2$

$$A_2 = 00^* 1A_1 + 00^* + 1A_1$$

finally replacing $A_2$ in the equation for $A_1$ we have

$$A_1 = 1A_1 + 000^* 1A_1 + 000^* + 01A_1$$

$$= (1 + 000^* 1 + 01)^* 000^*$$

the obvious answer!
Parsing

- In our compiler model, the parser accepts tokens and verifies that the string can be generated by the grammar.

- Three types of parsers
  - tabular
  - top-down
  - bottom up
Backtrack parsing

- Backtrack parsing simulates a nondeterministic machine.
- There are both top-down and bottom up forms of backtrack parsing.
- Both forms may take an exponential amount of time to parse.
- Let us only discuss the top down version.
- The name top-down parsing comes from the idea that we attempt to produce a parse tree for the input string starting from the top (root) and working down to the leaves.
• For top-down backtrack parsing, we begin with a tree containing one node labeled $S$, the start symbol of the grammar.

• We assume that the alternate productions for every nonterminal have been ordered. That is, if $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ are all the $A$ productions in the grammar, we assign some ordering to the $\alpha_i$’s (the *alternates* for $A$).
• Recursively proceed as follows
  -- If the active node is labeled by a nonterminal, say $A$, then choose the first alternate, say $X_1 \ldots X_k$, for $A$ and create $k$ direct descendants for $A$ labeled $X_1$, $X_2$, ..., $X_k$. Make $X_1$ the active node. If $k = 0$, then make the node immediately to the right of $A$ active.
  -- If the active node is labeled by a terminal, say $a$, then compare the current input symbol with $a$. If they match, then make active the node immediately to the right of $a$ and move the input pointer one symbol to the right. If $a$ does not match the current input symbol, go back to the node where the previous production was applied, adjust the input pointer if necessary and try the next alternate. If no alternate is possible, go back to the next previous node, and so forth.
Consider the grammar \( S \rightarrow aSbS | aS | c \) and the input string \( aacbc \). The sequence of parsing trees would be as follows:
The Cocke-Younger-Kasami Algorithm

- Tabular methods include: Cocke-Younger-Kasami algorithm and Earley’s algorithm.

- These are essentially dynamic programming methods and are discussed mainly because of their simplicity.

- However, they are highly inefficient (n^3 time and n^2 space). They are the parsing equivalent to “bubble sort”.
• **Definition**: A Context Free Grammar \((N, \Sigma, P, S)\) is said to be in Chomsky normal form (CNF) if each production in \(P\) is one of the forms

1. \(A \rightarrow BC\) with \(A, B, C\) in \(N\)
2. \(A \rightarrow a\) with \(a\) in \(\Sigma\), or
3. if \(\varepsilon\) is in \(L(G)\) then \(S \rightarrow \varepsilon\) is a production and \(S\) does not appear on the right hand side of any production.

• **Theorem**: Let \(L\) be a context free language (i.e. a language that can be generated by a context free grammar). Then \(L = L(G)\) for some CFG \(G\) in Chomsky Normal Form.
Algorithm 2.12
Conversion to Chomsky normal form.

Input. A proper CFG $G = (N, \Sigma, P, S)$ with no single productions.

Output. A CFG $G'$ in CNF, with $L(G) = L(G')$.

Method. From $G$ we shall construct an equivalent CNF grammar $G'$ as follows. Let $P'$ be the following set of productions:

1. Add each production of the form $A \rightarrow a$ in $P$ to $P'$.
2. Add each production of the form $A \rightarrow BC$ in $P$ to $P'$.
3. If $S \rightarrow e$ is in $P$, add $S \rightarrow e$ to $P'$.
4. For each production of the form $A \rightarrow X_1 \cdots X_k$ in $P$, where $k > 2$, add to $P'$ the following set of productions. We let $X'_i$ stand for $X_i$ if $X_i$ is in $N$, and let $X'_i$ be a new nonterminal if $X_i$ is in $\Sigma$.

\[
A \rightarrow X'_1\langle X_2 \cdots X_k \rangle \\
\langle X_2 \cdots X_k \rangle \rightarrow X'_2\langle X_3 \cdots X_k \rangle \\
\vdots \\
\langle X_{k-2} \cdots X_k \rangle \rightarrow X'_{k-2}\langle X_{k-1}X_k \rangle \\
\langle X_{k-1}X_k \rangle \rightarrow X'_{k-1}X_k
\]

where each $\langle X_1 \cdots X_k \rangle$ is a new nonterminal symbol.

5. For each production of the form $A \rightarrow X_1X_2$, where either $X_1$ or $X_2$ or both are in $\Sigma$, add to $P'$ the production $A \rightarrow X'_1X'_2$.

6. For each nonterminal of the form $a'$ introduced in steps (4) and (5), add to $P'$ the production $a' \rightarrow a$. Finally, let $N'$ be $N$ together with all new nonterminals introduced in the construction of $P'$. Then our desired grammar is $G' = (N', \Sigma, P', S)$. \qed
Example: The GFG, G, defined by

\[
S \rightarrow aAB \mid BA \\
A \rightarrow BBB \mid a \\
B \rightarrow AS \mid b \\
\]

can be converted to the equivalent CNF grammar, G’

\[
S \rightarrow a' <AB> \mid BA \\
A \rightarrow B <BB> \mid a \\
B \rightarrow AS \mid b \\
<AB> \rightarrow AB \\
<BB> \rightarrow BB \\
a' \rightarrow a \\
\]

where G=(N,\{a,b\},P,S) and G’=(N’,\{a,b\},P’,S). Here P and P’ correspond to the two sets of productions above and N’ ={S, A, B, <AB>, <BB>, a}. 
- let \( w = a_1a_2...a_n \) be an input string which is to be parsed according to \( G \). We assume that each \( a_i \) is in \( \Sigma \) for \( 1 \leq i \leq n \). The essence of the algorithm is the construction of a parse table \( T \), whose elements we denote \( t_{ij} \) for \( 1 \leq i \leq n \) and \( 1 \leq j \leq n-i+1 \). Each \( t_{ij} \) will have a value which is a subset of \( \mathbb{N} \). \( A \) will be in \( t_{ij} \) iff \( A \rightarrow^+ a_i a_{i+1}...a_{i+j-1} \), that is, if \( A \) derives the \( j \) input symbols beginning at position \( i \). As a special case, the input string \( w \) is in \( L(G) \) iff \( S \) is in \( t_{1n} \).
Algorithm 4.3

Cocke–Younger–Kasami parsing algorithm.

Input. A Chomsky normal form CFG $G = (N, \Sigma, P, S)$ with no $e$-production and an input string $w = a_1a_2 \cdots a_n$ in $\Sigma^*$.

Output. The parse table $T$ for $w$ such that $t_{ij}$ contains $A$ if and only if $A \Rightarrow^* a_ia_{i+1} \cdots a_{i+j-1}$.

Method.

1. Set $t_{ii} = \{A | A \rightarrow a_i \text{ is in } P\}$ for each $i$. After this step, if $t_{ii}$ contains $A$, then clearly $A \Rightarrow^* a_i$.

2. Assume that $t_{ij'}$ has been computed for all $i$, $1 \leq i \leq n$, and all $j'$, $1 \leq j' < j$. Set $t_{ij} = \{A | \text{for some } k, 1 \leq k < j, A \rightarrow BC \text{ is in } P,$ $B \text{ is in } t_{ik}, \text{ and } C \text{ is in } t_{i+k,j-k-1}\}^+$.

Since $1 \leq k < j$, both $k$ and $j - k$ are less than $j$. Thus both $t_{ik}$ and $t_{i+k,j-k}$ are computed before $t_{ij}$ is computed. After this step, if $t_{ij}$ contains $A$, then

$$A \Rightarrow^* BC \Rightarrow^* a_i \cdots a_{i+k-1}C \Rightarrow^* a_i \cdots a_{i+k-1}a_{i+k} \cdots a_{i+j-1}.$$ 

3. Repeat step (2) until $t_{ij}$ is known for all $1 \leq i \leq n$, and $1 \leq j \leq n - i + 1$. □
Example 4.8

Consider the CNF grammar $G$ with productions

\[
S \rightarrow AA | AS | b \\
A \rightarrow SA | AS | a
\]

Let $abaab$ be the input string. The parse table $T$ that results from Algorithm 4.3 is shown in Fig. 4.8. From step (1), $t_{11} = [A]$ since $A \rightarrow a$ is in $P$ and $a_1 = a$. In step (2) we add $S$ to $t_{31}$, since $S \rightarrow AA$ is in $P$ and $A$ is in both $t_{31}$ and $t_{41}$. Note that, in general, if the $t_{ij}$'s are displayed as shown, we can

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
i & 1 & 2 & 3 & 4 & 5 \\
\hline
j & A & S & A & S & \\
\hline
\end{array}
\]

compute $t_{ij}$, $i > 1$, by examining the nonterminals in the following pairs of entries:

\[(t_{i1}, t_{i+1,j-1}), (t_{i2}, t_{i+2,j-2}), \ldots, (t_{i,j-1}, t_{i,j-1,i})\]

Then, if $B$ is in $t_{ik}$ and $C$ is in $t_{i+k,j-k}$ for some $k$ such that $1 \leq k < j$ and $A \rightarrow BC$ is in $P$, we add $A$ to $t_{ij}$. That is, we move up the $i$th column and down the diagonal extending to the right of cell $t_{ij}$ simultaneously, observing the nonterminals in the pairs of cells as we go.

Since $S$ is in $t_{13}$, $abaab$ is in $L(G)$. □
Top Down parsing.

A non-backtracking form of top-down parser is called a predictive parser (section 2.4 of the textbook).

The main idea is that at each point it is possible to know by looking at the next input token which production should be applied. For most programming languages, the fact that each type of statement is initiated by a different keyword (e.g. for, while, if) facilitates the development of top-down predictive parsers.
Consider the following grammar

\[
\begin{align*}
type & \rightarrow \text{simple} \mid * \text{id} \mid \text{array} [ \text{simple} ] \text{ of type} \\
\text{symple} & \rightarrow \text{integer} \mid \text{char} \mid \text{num : num}
\end{align*}
\]

- Here the tokens are in bold.

- Starting from \textit{type}, it is always clear which production to apply.
  - Thus, if the next token is \textit{integer}, \textit{char}, or \textit{num}, the production \textit{type} \rightarrow \textit{simple} should be applied.
  - If it is *, the production \textit{type} \rightarrow * \textit{id} should be applied.
  - Finally, if it is \textit{array}, the production \textit{type} \rightarrow \textit{array} [ \text{simple} ] \text{ of type} should be applied.

- Clearly, no need for backtracking ever arises.
• Predictive parsing relies on the information about what first symbols can be generated by the right hand side of a production.

• Thus,
  -- FIRST(\textit{simple}) = \{\text{integer, char, num}\}
  -- FIRST(\* \textit{id}) = \{ * \}
  -- FIRST(\textit{array} [ \textit{simple} ] \text{of type}) = \{\text{array}\}
procedure match(t: token);
begin
  if lookahead = t then
    lookahead := nextoken
  else error
end;

procedure type;
begin
  if lookahead is in { integer, char, num } then
    simple
  else if lookahead = '†' then begin
    match ('†'); match(id)
  end
  else if lookahead = array then begin
    match (array); match ('>'); simple; match ('['); match (of); type
  end
  else error
end;

procedure simple;
begin
  if lookahead = integer then
    match (integer)
  else if lookahead = char then
    match (char)
  else if lookahead = num then begin
    match (num); match (dotdot); match (num)
  end
  else error
end;

Fig. 2.17. Pseudo-code for a predictive parser.
Problems with predictive parsing.

Consider the grammar

\[
S \rightarrow S \ a \\
S \rightarrow b
\]

By just looking at the next token it is impossible to determine how many times the first production should be applied.

For example, if the input is \textit{baaaaa} then the first production must be applied five times before the second one is.

Grammars like this are said to be left recursive.

Left recursion can be eliminated by changing the grammar. In this case to:

\[
S \rightarrow b \ S' \\
S' \rightarrow a \ S' \mid \epsilon
\]
A second problem is that it may not be possible to determine by just looking at the next token which production to apply.

Consider the following grammar:

\[
\begin{align*}
S & \rightarrow i \ E \ t \ S | \ i \ E \ t \ S \ e \ S \ | \ a \\
E & \rightarrow \ b
\end{align*}
\]

By just looking at the next token it is impossible to determine whether the first or the second production should be applied.

Again, changing the grammar can solve this problem:

\[
\begin{align*}
S & \rightarrow i \ E \ t \ S \ S' \ | \ a \\
S' & \rightarrow e \ S \ | \ \epsilon \\
E & \rightarrow \ b
\end{align*}
\]
Example

Consider the grammar:

$A \rightarrow \text{id} = E$

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$E \rightarrow -T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{id}$

$F \rightarrow \text{number}$

After eliminating the left recursion, we obtain
From where we can create the recursive descent parser.
procedure A()
{
    match(id);
    t=idnum;
    match('=');
    expression();
    emit("fstore");
    emitln(t);
}

procedure E()
{
    if ( lookahead == "-" ) {
        match("-");
        T();
        emit("fneg");
    }
    else T();
    E'();
}

procedure E'()
{
    if ( lookahead == "-" ) {
        match("-");
        T();
        emit("fsub");
        E'();
    }
    elseif ( lookahead == "+" ) {
        match("+");
        T();
        emit("fadd");
        E'();
    }
}

procedure T()
{
    F();
    T'();
}

procedure T'()
{
    if ( lookahead == "*" ) {
        match("*");
        F();
        emitln("fmul");
        T'();
    }
    elseif ( lookahead == "/" ) {
        match("/");
        F();
        emitln("fdiv");
        T'();
    }
}
procedure F() {
    if ( lookahead == num ) {
        emit("ldc");
        emitln(value);
        match (num);
    }
    elseif lookahead == "(" {
        match("(");
        E();
        match(")");
    }
    else {
        match(id);
        emit("fload");
        emitln(idnum);
    }
}