ב"ה

Course Notes on Formal Languages and Compilers

R.B. Yehezkael

Jerusalem College of Technology

Havaad Haleumi 21, Jerusalem 91160, Israel

E-mail address: rafi@mail.jct.ac.il

Fax: 02-6422075   Tel: 02-6751111

December 2004 - טבת תשס"ה

CONTENTS

1. LANGUAGES AND GRAMMARS

We shall now formalize the notion of language and grammar.

Strings and sets of strings

If V is a set, then V* denotes the set of all finite strings of elements of V including the empty string which will be denoted by ε.
e.g. $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001,\dots\}$

The set of all non empty strings of elements of V is denoted by V^+.
Usually, $V^+ = V^* \setminus \{\varepsilon\}$, but when $\varepsilon \in V$, $V^+ = V^*$.
e.g. $\{0,1\}^+ = \{0, 1, 00, 01, 10, 11, 000, 001,\dots\}$
but $\{\varepsilon, 0, 1\}^+ = \{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001,\dots\}$

If $x \in V^*$ and $y \in V^*$ then xy will denote their concatenation, that is, the string consisting of x followed by y.

If $x \in V^*$ then $x^n = \underbrace{\underline{xxx.....x}}_{\text{n-times}}$ $\quad n \geq 0$

We assume $x^0 = \varepsilon$ the empty string.

e.g. $\{a\}^* = \{\varepsilon, a, a^2, a^3, \dots a^n, \dots\} = \{a^n : n \geq 0\}$
$\{a\}^+ = \{a, a^2, a^3, \dots a^n, \dots\} = \{a^n : n \geq 1\}$

Similarly, if X, Y are sets of strings, then their concatenation is also denoted by XY. Of course $XY = \{xy : x \in X \text{ and } y \in Y\}$.
Also, $X^n = \underbrace{\underline{XXX.....X}}_{\text{n-times}}$ $\quad n \geq 0$. Of course $X^0 = \{\varepsilon\}$.

e.g. $\{0, 1\}\{a, b, c\} = \{0a, 0b, 0c, 1a, 1b, 1c\}$
$\{0, 1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

If x is a string, then $|x|$ denotes the length of x, and this is the number of indivisible symbols in x. Of course $|\varepsilon| = 0$.

Exercises

1) Determine the following sets.
(a) $\{0,1\}\{\varepsilon, a, ba\}$ (b) $\{b, aa\}^*$

2) Let V be a set of strings. Does $V^+ = V V^*$?

Vocabulary and language

A vocabulary (or alphabet or character set or word list) is a finite nonempty set of indivisible symbols (letters, digits, punctuation marks, operators, etc.).

A language over a vocabulary V is any subset L of V* which has a finite description. There are two approaches for making this mathematically precise. One is to use a grammar - a form of inductive definition of L. The other is to describe a method for recognizing whether an element $x \in L$ is in the language L and automata theory is based on this approach.

Examples of languages:

1) Any finite subset of V* is a language as it can be described explicitly by listing its elements.

2) $L = \{x \in V^* : |x| \text{ is even}\}$.

Non languages

It is reasonable to ask if every subset of V* is a language over a vocabulary V, that is, can every subset of V* be described finitely. The answer to this question is that there are subsets of V* which are not languages as the following argument by contradiction shows.

Suppose every subset of V* is a language where lets say $V = \{a\}$. It may be shown that all these languages may be enumerated $L_0, L_1, \ldots, L_n \ldots$. Consider the set $S = \{a^n : a^n \notin L_n\}$. The description of S is finite and $S \subseteq V^*$. So S is a language over V. Thus, $S = L_k$ for some k as $L_0, L_1, \ldots L_n, \ldots$ is supposed to enumerate all languages.

Now $a^k \in S$, if and only if $a^k \notin L_k$ - by definition of S
if and only if $a^k \notin S$ - as $S = L_k$.

This is a contradiction. Thus there are subsets of V* which are not languages. NOTE: there is no way of giving an example of a non language because its description is not finite. Yet we can prove their existence as shown above by reductio ad absurdum.

Grammars

G=(N,T,P,S) is said to be a phrase structure grammar if:

1) N,T are disjoint vocabularies that is, $N \cap T = \varnothing$. T is the set of terminal symbols which are the "characters" or "words" from which the language is formed. N is the set of non terminals which are used for describing the structure of the language.

2) P is a finite set of rules (replacement rules, productions) each of which has the form $x \to y$ where $x, y \in (N \cup T)^*$.

3) S is the starting set (initial set) of the grammar and is a finite subset of $(N \cup T)^*$ that is $S \subseteq (N \cup T)^*$.

Derivations and language of a Grammar

Let $G=(N,T,P,S)$ be any <u>phrase structure grammar</u> and let $u,v \in (N \cup T)^*$. We write $u \Rightarrow v$ and say v is <u>derived in one step</u> from u by the rule $x \to y$, providing that $u = pxq$ and $v = pyq$. (Here the rule $x \to y$ is used to replace x by y in u to produce v. Note that $p,q \in (N \cup T)^*$.)

If $u_1 \Rightarrow u_2 \Rightarrow u_3 \ldots\ldots \Rightarrow u_n$ we say u_n is <u>derived</u> from u_1 in G and write $u_1 \overset{+}{\Rightarrow} u_n$.

Also if $u_1=u_n$ or $u_1 \overset{+}{\Rightarrow} u_n$ we write $u_1 \overset{*}{\Rightarrow} u_n$

L(G) the language of G is defined by:

$$\begin{aligned} L(G) &= \{t \in T^*: Z \overset{*}{\Rightarrow} t \text{ for some } Z \in S\} \\ &= \{t \in T^*: t \in S \text{ or } Z \overset{+}{\Rightarrow} t \text{ for some } Z \in S\}. \end{aligned}$$

So the elements of L(G) are those elements of T^* which are elements of S or which are derivable from elements of S.

Classification of Phrase Structure Grammars

The following table describes the various types of grammars, and their relative advantages and disadvantages. Historically this classification of grammars arose because of the need to find efficient recognition algorithms for the languages of grammars.

Type of Grammar	*Restrictions on G = (N,T,P,S)*	*Advantages and Disadvantages*
Phrase Structure (Type 0)	No further restrictions	Can describe any language but mathematically proveable that no general recognition algorithms exist.
Context Sensitive (Type 1)	Rules only take the form $x \rightarrow y$ where $\|x\| \leq \|y\|$	All programming language features can be described with these grammars, and general recognition algorithms exist but they are inefficient.
Context Free (Type 2)	Rules only take the form $A \rightarrow y$ where $A \in N$ and $y \in (N \cup T)^*$	Efficient general recognition methods exist and this type is widely used in practice. However the matching of use of variables and their declaration can't be expressed in this grammar. Neither can type matching be expressed.
Regular Grammars (Type 3)	Rules only take the form $A \rightarrow \varepsilon$ or $A \rightarrow Ba$ where $A, B \in N$ and $a \in T$. The start set must satisfy $S \subseteq N$.	Sequential recognition methods exist with either left to right or right to left scan. These grammars can only describe language features having a sequential structure, e.g., numbers or identifiers. Nested tree-like structures can't be described, e.g., block structure, expressions, if then else statements, etc.

<u>Example</u>: Let $L = \{a^n b^n c^n d^n : n \geq 0\}$. We shall describe L by means of an inductive definition and by means of a phrase structure grammar.

Inductive Definition	(a) As $a^0b^0c^0d^0 \in L$ and $a^0b^0c^0d^0 = \varepsilon$ (empty string), thus $\varepsilon \in L$.
	(b) if $a^n b^n c^n d^n \in L$ then $a^{(n+1)}b^{(n+1)}c^{(n+1)}d^{(n+1)} \in L$
	(c) Nothing else is in L.
Phrase Structure Grammar	$G_1=(N_1,T_1,P_1,S_1)$ where $N_1=\{X, Y\}$ $T_1=\{a,b,c,d\}$, $S_1=\{abcd\}$ and P_1 consists of the rules:

$$\begin{array}{ccc} & abcd\to\varepsilon & \\ & bc\to XbcY & \\ bX\to Xb & & Yc\to cY \\ aX\to aab & & Yd\to cdd \end{array}$$

We shall not prove that $L(G_1) = \{ a^nb^nc^nd^n: n\geq 0\}$ but will give examples of derivations of ε and $a^3b^3c^3d^3$ from the start set. Incidentally $abcd\in L(G_1)$ because $abcd\in T_1^*$ and $abcd\in S_1$.

As S_1 has only one element that is, abcd, all derivations must start with abcd. In the following derivations we shall underline the part of the string which is being replaced at each derivation step.

Derivation of ε: $\underline{abcd} \Rightarrow \varepsilon$ directly by a rule

Derivation of $a^3b^3c^3d^3$:

$$\begin{array}{lll} a\underline{bc}d & \Rightarrow & \underline{aX}bcYd \Rightarrow aabbc\underline{Yd} \Rightarrow aab\underline{bc}cdd \\ & \Rightarrow & aa\underline{bX}bcYcdd \Rightarrow aaXbbc\underline{Yc}dd \\ & \Rightarrow & a\underline{aX}bbccYdd \Rightarrow aaabbbcc\underline{Yd}d \\ & \Rightarrow & aaabbbcccddd = a^3b^3c^3d^3 \end{array}$$

<u>Exercise:</u> Continue the above derivation to derive $a^4b^4c^4d^4$.

<u>Example of a Context Sensitive Grammar:</u> The previous grammar G_1 is not a context sensitive grammar. An equivalent context sensitive grammar is $G_2 = (N_2, T_2, P_2, S_2)$ where $N_2=\{X, Y\}$ as before $T_2 = \{a,b,c,d\}$ as before, $S_2=\{abcd,\varepsilon\}$ and P_2 consists of the rules

$$\begin{array}{ccc} & bc\to XbcY & \\ bX\to Xb & & Yc\to cY \\ aX\to aab & & Yd\to cdd \end{array}$$

What we have done is to eliminate the rule $abcd\to\varepsilon$ which is not permitted in a context sensitive grammar and add ε to the starting set of G_2.

<u>Exercises</u>

1) Is G_2 a phrase structure grammar too?

2) Write a context sensitive grammar for the language $\{a^{2n}b^nc^nd^{2n}: n\geq 1\}$. Present a derivation of "aaaabbccdddd" from the starting set of the grammar.

3) Write a context sensitive grammar and a context free grammar for $\{a^m b^m c^n d^n: m \geq 2, n \geq 3\}$. Present derivations of "aaabbbcccccddddd" from the starting sets of the grammars.

4) Let G = ({E}, {v, +, -, (,)}, {E→v, E→(E+E), E→(E-E)}, {E}).

a) What types of grammars is G ?

b) Present derivations of (v+v) and ((v+v)-v) from the starting set of the grammar.

c) Can (v+v-v) be derived from the starting set of the grammar?

d) Describe L(G) in words.

5) Write a regular grammar for $\{ a^m b^n: m,n \geq 0\}$. Present a derivation of "aab" from the starting set of the grammar.

Recognition of Context Sensitive Languages

Let G=(N, T, P, S) be a context sensitive grammar and let $t \in T^*$. Let m=|t| denote the length of t. Here is an algorithm for deciding whether or not $t \in L(G)$.

We shall construct a series of sets $L_0, L_1, \ldots, L_m$, where for each i such that $0 \leq i \leq m$, where L_i is the set of all strings of length i which are elements of S or derivable from S. Each L_i is a subset of the set of all strings of $(N \cup T)^*$ of length i and so must be finite. Initially $L_i = \{z \in S: |z| = i\}$

As G is context sensitive |x|>|y| impossible. So G has two types of rules:

(a) $x \rightarrow y$ where |x|=|y|

(b) $x \rightarrow y$ where |x|<|y|

OBSERVATION: It is impossible to derive t from a string x where |x|>|t|, since the rules never shorten a string when applied to it. Thus all strings of length greater than |t|=m can be ignored. This is the basis of the method.

Do steps A) and B) for i=0,1,2,...., m.

A) Apply all rules of type (a) to elements of L_i, repeatedly adding all new strings derived to L_i. Stop when nothing further can be added. As L_i is finite (see above) infinite looping is impossible.

B) Apply all rules of type (b) to elements of L_i. The new strings derived will be increased in length by |y|-|x| where $x \rightarrow y$ is the rule being applied and |y|>|x|. Add the new strings to $L_{i+|y|-|x|}$ if $i+|y|-|x| \leq m$.

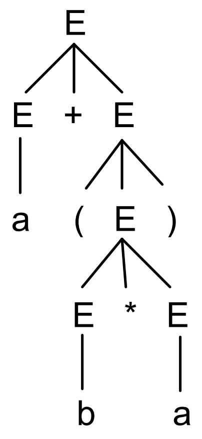
After $L_0, L_1 \ldots, L_m$ have all been generated we can test if $t \in L(G)$ by testing whether or not $t \in L_m$, a finite set.

NOTE This method cannot be applied to phrase structure grammars because our observation above is invalid in the general case.

2. CONTEXT FREE GRAMMARS AND LANGUAGES

Derivations in Context Free Grammars

Derivations in these grammars can be represented by trees. For example, let G=(N, T, P,S) where N=S={E} and T ={a,b,+,*,(,)} and P consists of the rules $E\rightarrow a$, $E\rightarrow b$, $E\rightarrow E+E$, $E\rightarrow E*E$, $E\rightarrow (E)$. A derivation of a+(b*a) can be represented by the tree:



<u>Class discussion</u>: Is there one or are there many derivations corresponding to the above tree?

A derivation in a context free grammar is called a leftmost derivation, if in all substitutions in the derivation, we substitute on the leftmost non terminal. Similarly it is called a rightmost derivation, if in all substitutions in the derivation, we substitute on the rightmost non terminal.

Here are leftmost and rightmost derivations corresponding to the above derivation tree. We have underlined the non terminal which is replaced.

Leftmost: $\underline{E}\Rightarrow\underline{E}+E\Rightarrow a+\underline{E}\Rightarrow a+(\underline{E})\Rightarrow a+(\underline{E}*E)\Rightarrow a+(b*\underline{E})\Rightarrow a+(b*a)$

Rightmost: $\underline{E}\Rightarrow E+\underline{E}\Rightarrow E+(\underline{E})\Rightarrow E+(E*\underline{E})\Rightarrow E+(\underline{E}*a)\Rightarrow \underline{E}+(b*a)\Rightarrow a+(b*a)$

<u>Exercises</u>

1) Write a single derivation equivalent to the previous derivation tree which is not a leftmost derivation and not a rightmost derivation.

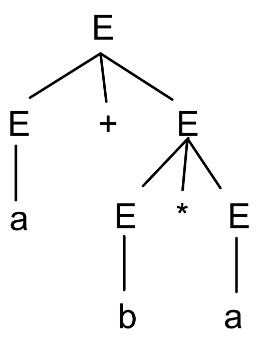
2) Draw a derivation tree for $E \Rightarrow^{+} (b+(b*a))$. How many derivation trees can be drawn in this case?

Ambiguity

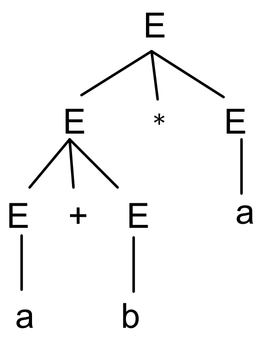
A context free grammar is ambiguous if there is a $t \in L(G)$ which has two different derivation trees. (This is equivalent to the existence of two different leftmost derivations of t or of two different rightmost derivations of t.)

The above grammar is ambiguous because a+b*a has two different derivation trees.

Tree 1



Tree 2



It is important to avoid ambiguity, for this gives rise to double meanings. For example, with reference to the above, a+b*a could either be interpreted as a+(b*a) corresponding to tree 1 or as (a+b)*a corresponding to tree 2.

Unfortunately testing for ambiguity is hard, and it has been proved that there is no algorithm which tests if a context free grammar is ambigous.

Exercises

1) The language $\{ a^m b^m a^n : m, n \geq 0\} \cup \{ a^m b^n a^n : m, n \geq 0\}$ has been proved to be inherently ambigous, that is, every context free grammar defining this language is ambigous. Define a context free grammar for this language and show that your grammar is ambigous.

2) Deleting a rule from an unambigous context free grammar will not make the grammar ambigous. Explain.

Recognition of context free languages

A context free grammar is also context sensitive, if it has no rules of the form $A \to \varepsilon$. If we are able to remove these rules from context free grammars they may be recognized by the context sensitive recognition method.

Lemma: Removing rules of the form $A \to \varepsilon$

Let G be a context free grammar. Then an equivalent context free grammar G' can be found such that G' has no rules of the form $A \to \varepsilon$ and L(G')=L(G).

Proof Let F = {A∈N: $A\overset{+}{\Rightarrow}\varepsilon$}. Assume F can be constructed from G. (Finding a method is an exercise)

Let G=(N,T,P,S) and G'=(N',T',P',S') where N' = N, T' = T.

S' is obtained from S by:

1) Initializing S' to S.
2) Deleting members of F from elements of S' in all possible ways and adding the new elements formed to S'.
   e.g., if ABC∈S' and A,B,C∈F, then this will result in AB, AC, BC, A,B,C,ε being added to S'

Similarly P' is obtained from P by:-

1) Initializing P' to P
2) For each rule A→y in P' we examine y and, as before, delete elements of F from y in all possible ways. Let $y_1,..,y_n$ be the strings obtained, then add $A\to y_1,.., A\to y_n$, to P'. For example,
   if D→ABC is in P' and A,B,C,∈F
   then add the lollowing rules to P': D→BC, D→AB, D→AC,
   D→A, D→B, D→C, D→ε.
3) Delete all rules of the form A→ε from P'.

The previous result allows us to use the context sensitive recognition method for context free grammars. The next two results will help us in obtaining improved recognition methods for context free grammars.

Lemma: Removing rules of the form A→B

Let G be a context free grammar. Then it is possible to construct an equivalent context free grammar G' having no rules of the form A→B where A,B∈N' and such that L(G')=L(G).

Proof Let G=(N,T,P,S) and G'=(N',T',P',S') where N'=N, T'=T, and S'=S..
By the previous lemma we may assume there are no rules of the form A→ε.

P' is obtained from P as follows:-

1) P' is initialized to P with rules A→B removed, A,B∈N.
2) If $A\overset{+}{\Rightarrow}B$ in G and B→y is in P', add A→y to P'

Repeat (2) till nothing further can be added to P'.

Notes:
1) The problem of determining if $A\overset{+}{\Rightarrow}B$ in G is an exercise.
2) If the grammar G is <u>like</u> a regular grammar but with extra rules of the form A→B, the above procedure will work without removing rules of the form A→ε, and in this case we obtain a regular grammar G' equivalent to G.

Exercise: Let G = (N, T, P, S) be a context free grammar and let N = $\{A_1,....,A_n\}$.

(a) Let F = $\{A\in N: A\overset{+}{\Rightarrow}\varepsilon\}$. How can F be constructed from G?

(b) Suppose further that G has no rules of the type $A \rightarrow \varepsilon$ that is F = $\emptyset$. How can we test whether or not $A_i \overset{+}{\Rightarrow} A_j$ for A_i, $A_j \in N$?

Chomsky normal form

A context free grammar G=(N, T, P, S) is in Chomsky Normal Form if it only has rules of the type $A\rightarrow BC$ or $A\rightarrow a$ where $A,B,C\in N$ and $a\in T$

Theorem: Conversion to Chomsky normal form

Any context free grammar G=(N,T,P,S) can be converted into an equivalent grammar G'=(N', T', P', S') in Chomsky normal form such that L(G)=L(G')

Proof By the previous two lemmas we may assume without loss of generality that G has no rules of the form $A\rightarrow\varepsilon$ or $A \rightarrow B$, $A,B\in N$.
G' is obtained from G as follows S'=S, T'=T. Initially P'=P and N'=N, but these may be changed in the following. Consider each rule $A\rightarrow y$ in P'. In view of the fact that there are no rules of the form $A\rightarrow\varepsilon$ and $A\rightarrow B$, $A,B,\in N$ we can say that either $y\in T$ or $|y|\geq 2$. If $y\in T$ the rule $A\rightarrow y$ is permitted in Chomsky Normal Form so $A\rightarrow y$ can be left in P'. Otherwise $|y|\geq 2$. So y can be written $B_1\ B_2.....B_n$ where $n=|y|\geq 2$ and $B_i\in N\cup T$.

Thus the rule we are dealing with is $A\rightarrow B_1\ B_2\ ...\ B_n$. This is transformed as follows.

1) Ensure that there are no non terminals on the right hand side by replacing $A\rightarrow B_1\ B_2.....B_n$ by $A\rightarrow B_1'\ B_2'\ ...\ B_n'$ where $B_i' = B_i$ if B_i is in N, but if $B_i\in T$ B_i' is a new non terminal which is added to N'. Also when $B_i\in T$ we add $B_i' \rightarrow B_i$, a valid Chomsky normal form rule to P'.
2) Now deal with the rule $A\rightarrow B_1'\ B_2'...B_n'$ $n\geq 2$ as follows. If n=2 rule is in Chomsky Normal form as each $B_i'\in N'$. If n>2 replaces $A\rightarrow B_1'...B_n'$ by the rules.
$A\rightarrow B_1'\ C_1$
$C_1 \rightarrow B_2'\ C_2......$ $C_{n-3}\rightarrow B'_{n-2}\ C_{n-2}$
$C_{n-2}\rightarrow B'_{n-1}\ B_n'$.
Where $C_1, ..., C_{n-2}$, are new non terminals which are added to N'.

This completes the description of the method. The reason why it works is where we had

$A \rightarrow B_1 B_2 \ldots B_n$, in G, we have instead
$A \Rightarrow B_1' C_1' \Rightarrow B_1' B_2' C_2 \ldots \Rightarrow B_1' B_2' \ldots B_n' \overset{+}{\Rightarrow} B_1 B_2 \ldots B_n$ in G'
that is $A \overset{+}{\Rightarrow} B_1 B_2 \ldots B_n$. Thus L(G) is unchanged and so equals L(G').

Example:

Transform G = ({E,T,I}, {a,+}, P, {E})
where P consists of $E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow aI$
$I \rightarrow \varepsilon$
$I \rightarrow aI$

into Chomsky Normal form. Initially G'= G. Then G' is transformed as follows

Stage 1 eliminate rules of type $A \rightarrow \varepsilon$,
that is remove $I \rightarrow \varepsilon$ from P'. To compensate for the removal, $I \rightarrow a$, $T \rightarrow a$ are added to P'.

Stage 2 Remove rules of type $A \rightarrow B$,
that is remove $E \rightarrow T$ from P'. To compensate for the removal, $E \rightarrow aI$, $E \rightarrow a$, are added to P'.

Stage 3 deal with rules $A \rightarrow y$ where $|y| \geq 2$

Substage A - Make right hand sides consist entirely of non terminals.

$E \rightarrow T\ +'\ E$	$+' \rightarrow +$
$E \rightarrow a'\ I$	$a' \rightarrow a$
$T \rightarrow a'\ I$	
$I \rightarrow a'\ I$	where +', a' are new non terminals of G'.

Substage B - Handle right hand sides with length greater than 2.

$E \rightarrow T\ C_1$
$C_1 \rightarrow +'\ E$ where C_1 is a new non terminal of G'.

Thus now grammar G'= ({E, T, I, +', a', C_1}, {a,+}, P', {E})
Where P' consists of:

$I \rightarrow a$	$E \rightarrow a$	$E \rightarrow T\ C_1$
$T \rightarrow a$	$E \rightarrow a'\ I$	$C_1 \rightarrow +'\ E$
$+' \rightarrow +$	$T \rightarrow a'\ I$	
$a' \rightarrow a$	$I \rightarrow a'\ I$	

Derivation In Chomsky Normal Form And Syntax analysis

Derivations have a binary tree structure in Chomsky normal form and this can be put to use when trying to recognise if a terminal string t is derivable from a non terminal C. Here is a recursive algorithm for this purpose.

```
function derivable (C: non_terminal; t: terminal_string)
        return boolean is

-- SPECIFICATION:
-- derivable (C,t) is true if and only if t is derivable from C, that is, $C \overset{+}{\Rightarrow} t$.

begin
        if t = $\varepsilon$
        then return false;
        elsif C $\to$ t is a rule
        then return true;
        elsif t is a single terminal  (that is |t| = 1)
        then return false
        else - - |t| $\geq$ 2, try all splits of t into two parts.
                for k in 1..|t| - 1 -- k is a local variable
                loop
                        for all rules of the form C$\to$AB in
                                the grammar G -- G is a global variable
                        loop
                                if derivable (A, t (1..k)) and then
                                  derivable (B,t (k+1..|t|))
                                then return true;
                                endif;
                        endloop;
                endloop;
                return false;
        endif;
end derivable;
```

Importance of Chomsky Normal Form

- Derivations easily represented by binary trees.
- Elegant syntax analysis methods available.
- Any context free grammar can be converted into Chomsky Normal Form.

Exercises

1) (a) Construct Context free Grammars for L_1, L_2 where

$L_1 = \{a^m b^m c^n : m, n \geq 1\}$

$L_2 = \{ a^m b^n c^m : m, n \geq 1\}$

(b) Explain why $L_1 \cap L_2 = \{a^n b^n c^n : n \geq 1\}$.

(It has been proved that $\{ a^n b^n c^n : n \geq 1\}$ is not a context free language, and from this it follows that the intersection of two context free languages need not be context free.
It can also be shown that the complement of a context free language need not be context free.)

2) Let G = ({L,B} , { (,) } , P, {L}) be a context free grammar where P consists of the rules:-

$L \rightarrow \varepsilon$ $\quad\quad$ $L \rightarrow LB$ $\quad\quad$ $B \rightarrow (L)$

(a) Is "()(())((()))" in L(G)?
(b) Is "())(" in L(G)?
(c) Describe the set L(G) in English
(d) Transform G into Chomsky Normal Form.

3) A Non terminal is useful if and only if

(a) it can be reached from the start set, and
(b) some terminal string can be derived from it

A non terminal is useless if it is not useful. Explain how the useless non terminals of a context free grammar may be removed. Suggestion: Treat (a) and (b) separately.

4) Let G be any context free grammar. Explain how to convert G into an equivalent context free grammar G' such that the start set of G' has only one element which is a single non terminal symbol.

5) Let L_1, L_2 be context free languages. This means of course that there are context free grammars G_1, G_2 such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$. Prove that the following are also context free languages.

(a) $L_1 \cup L_2$ $\quad\quad$ (b) $L_1 L_2$
(c) L_1^* , L_2^* $\quad\quad$ (d) L_1^R , L_2^R (the reversals of L_1, L_2)

Forward Deterministic Context Free Grammars

A Context Free Grammar G = (N,T,P,S) is said to be forward deterministic providing that:-

(a) S has only one element which is a non terminal.
(b) All rules have the form A $\rightarrow$ ux where u is a terminal and x is a string of terminals and/or non terminals. (Note that x may be empty.)
(c) For each non terminal A, all the right hand sides of its rules start with different terminals. This means that we never have A $\rightarrow$ ux and A $\rightarrow$ uy for x$\neq$y.

(Later on when discussing top down syntax analysis, we shall generalize this concept and allow rules of the form A $\rightarrow$ ux where u is a non terminal and also rules of the form A $\rightarrow$ ε.)

These restrictions allow us to analyze the input deterministically using a stack. We start with the start non terminal on the stack and substitute on it the appropriate right hand side which matches the input, and then delete the matching symbols from the input and from the stack. If no right hand side matches, the syntax analysis fails. We carry on in this way with the non terminal on the top of the stack until no further substitutions are possible. If at the end, both stack and input are empty the input is accepted, otherwise it is rejected.

<u>Exercise</u>: Write the previous paragraph in algorithmic style.

<u>Example of a Forward Deterministic Context Free Grammar</u>

G= ({C,E} , {if, then, else, endif, statement, boolean}, P , {C})

The rules in P are

C $\rightarrow$ if boolean then C E
C $\rightarrow$ statement
E $\rightarrow$ else C endif
E $\rightarrow$ endif

Here are two examples of syntax analysis, one succesful and one failing. (The top of the stack is on the left.)

Stack	Input
C	if boolean then statement endif
if boolean then C E	if boolean then statement endif
C E	statement endif
statement E	statement endif
E	endif
endif	endif
ε	ε

Input is accepted as Stack and Input are empty.

Stack	Input
C	if boolean then else statement endif
if boolean then C E	if boolean then else statement endif
C E	else statement endif

Substitutions now fail and the input is rejected.

<u>Exercise:</u> Show the states of Stack and Input when analyzing:-
(i) if boolean else statement
(ii) if boolean then if boolean then statement endif endif

Discussion

There is a whole theory of stack automata including deterministic and non deterministic forms. Stack automata (including the non deterministic ones) are in fact equivalent to Context Free Grammars. We will not present this theory but we shall use stacks (or pushdowns) for explaining, in a practical style, various algorithms for recognizing context free languages.

3. REGULAR GRAMMARS AND LANGUAGES

Consider the regular grammar $G_1 = (\{A, L\}, \{l,d\}, P_1, \{A\})$

where P_1 consists of

$A \rightarrow Al$
$A \rightarrow Ad$
$A \rightarrow Ll$
$L \rightarrow \varepsilon$

Over here, "l" can be thought of as representing a letter and "d" a digit

$L(G_1)$ is the set of identifiers and here is an example derivation:

$A \Rightarrow Ad \Rightarrow Ald \Rightarrow Llld \Rightarrow lld$

This derivation can be constructed systematically by working backwards. This is possible whenever the right hand sides of rules are all different.

Consider the regular grammar $G_2 = (\{N,D\}, \{d\}, P_2, \{N\})$
where we wish to make $L(G_2)$ to be $\{d^n : n \geq 1\}$

At least two possibilities exist for P_2

(A)	$N \rightarrow Dd$	(B)	$N \rightarrow Nd$
	$D \rightarrow Dd$		$N \rightarrow Dd$
	$D \rightarrow \varepsilon$		$D \rightarrow \varepsilon$

Let's say we try to use rules (A) for recognizing if $ddd \in L(G)$ by forming derivation backwards:

$$\begin{tikzcd}
D \arrow[r, Rightarrow] & Dd \arrow[r, Rightarrow] & Ddd \arrow[r, Rightarrow] & Dddd \arrow[r, Rightarrow] & ddd \\
N \arrow[ur, Rightarrow] & Nd \arrow[ur, Rightarrow] & Ndd \arrow[ur, Rightarrow] & &
\end{tikzcd}$$

Several dead ends need to be explored to make sure all possibilities are tried. This is far more complex than using rules (B) which has a unique derivation when developed backwards:

$N \Rightarrow Nd \Rightarrow Ndd \Rightarrow Dddd \Rightarrow ddd$

Backward Deterministic Regular Grammars

A regular grammar is called backward deterministic if all rules have different right hand sides. These grammars have the property that the derivation can be easily constructed by working backwards - see previous examples.

Recognition Algorithm for Backward Deterministic Regular Grammars

Say we wish to recognise whether a string

$a_1a_2\text{..........}a_n \in L(G)$ where each $a_j \in T$

We proceed as follows:

1) Initially set u to $Aa_1a_2\text{..........}a_n$ where A is <u>the</u> nonterminal such that $A \rightarrow \varepsilon$.
If A exists it is unique.
If no such A exists then $L(G)= \varnothing$, and therefore $a_1a_2\text{..........}a_n \notin L(G)$.

2) While u can be reduced by a rule, do the reduction (and repeat process) that is if u has the form Aa_ja_{j+1} .... a_n and $B \rightarrow Aa_j$ is a rule, then reduce u to Ba_{j+1} ....a_n. Note, if such a rule exists, it is unique as all rules have different right hand sides.

3) Once step 2 can't be repeated we can say

$a_1a_2\text{..........}a_n \in L(G)$ if and only if $u \in S$ (the start set of G).

Theorem: Conversion to backward deterministic form

Any regular grammar G can be transformed into a backward deterministic regular grammar G' such that L(G') = L(G).

(What this means is that any regular language is easily recongnised by a sequential syntax analysis process).

<u>Proof:</u> Let G = (N, T, P, S) be a regular grammar. A backward deterministic regular grammar. G' = (N', T, P', S') is constructed from G using the following method which is called the <u>set construction</u>.

A non terminal of G' is a non empty set of non terminals of G,

that is, $N' = \{U: U \subseteq N \text{ and } U \neq \varnothing\}$.

Equivalently, $U \in N'$ if and only if $U \subseteq N$ and $U \neq \varnothing$.

$S' = \{ U \subseteq N : U \cap S \neq \emptyset$, that is there is an element of S which is also in U.$\}$

We now construct P'.

(i) Initially $P' = \emptyset$.

(ii) Let $U_0 = \{A : A \rightarrow \varepsilon$ is in P$\}$. If $U_0 \neq \emptyset$ add $U_0 \rightarrow \varepsilon$ to P'.
(Note there is at most one such rule in P'.)

(iii) For every $V \in N'$ and every $a \in T$ add the rule $U \rightarrow Va$ to P' only when $U = \{A : A \rightarrow Ba$ is in P for $B \in V\}$ and $U \neq \emptyset$.

This construction ensures that the grammar is regular and that all the right hand sides are diferent that is the grammar is in backward deterministic form.

It can be shown that for any $t \in T^*$, if $U \Rightarrow^{+} t$ in G' then for all $A \in U$, $A \Rightarrow^{+} t$ in G.

It can also be shown that for $t \in T^*$, if $A \Rightarrow^{+} t$ in G then for some U such that $A \in U$ we have $U \Rightarrow^{+} t$ in G'.

Applying these results $Z \in S$ and $Z' \in S'$ it follows that

$Z \Rightarrow^{+} t$ if and only if $Z' \Rightarrow^{+} t$ for some $Z \in Z'$

Therefore L(G') = L(G) where G' is backward deterministic. Q. E. D.

Some comments on the set construction

1) A terminal string can be derived from at most one non terminal in G'.

2) The number of non terminals may be very large. This number can be reduced by first removing useless nonterminals from the original grammar G and then starting the set construction from the non terminal U_0 described at stage (ii) in the proof and working backwards from this non terminal until no new non terminal are created. In this way the creation of useless non terminals is avoided.

3) The set construction is interesting in that it replaces synchronous execution of a parallel process (that is trying all ways back) by a sequential process involving sets.

Theorem: Closure properties of regular languages

Let L_1, L_2 be regular language. This means of course that there are regular grammars G_1, G_2 such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$. Then the following are also regular languages.

(a) $L_1 \cup L_2$ (b) $L_1 \cap L_2$
(c) $L_1 \setminus L_2$ (d) L_1L_2
(e) L_1^* , L_2^* (f) L_1^R , L_2^R (the reversals of L_1, L_2)

<u>Proofs</u>: In the following we assume that G_1, G_2 are regular grammars
$G_1 = (N_1, T_1, P_1, S_1)$ and $G_2 = (N_2, T_2, P_2, S_2)$.
and $L_1 = L(G_1)$, $L_2 = L(G_2)$.

(a) If neccessary rename the non terminals so that the non terminals of G_1, G_2 are all different, that is $N_1 \cap N_2 = \varnothing$.

Define $G = (N_1 \cup N_2, T_1 \cup T_2, P_1 \cup P_2, S_1 \cup S_2)$.

Clearly for any non terminal $A \in N_1 \cup N_2$ and any string $t \in (T_1 \cup T_2)^*$ either $A \in N_1$ or $A \in N_2$ but not both. So clearly $A \overset{+}{\Rightarrow} t$ in G if and only if $A \overset{+}{\Rightarrow} t$ in G_1 or $A \overset{+}{\Rightarrow} t$ in G_2

Therefore $t \in L(G)$ if and only if $t \in L(G_1)$ or $t \in L(G_2)$
that is $L(G) = L(G_1) \cup L(G_2) = L_1 \cup L_2$. Thus as G is a regular grammar, $L_1 \cup L_2$ is a regular language. Q.E.D.

(b) Let $G = (N_1 \times N_2, T_1 \cap T_2, P, S_1 \times S_2)$ where P contains all rules of the form:-

$(A_1, A_2) \rightarrow \varepsilon$ whenever $A_1 \rightarrow \varepsilon$ is in P_1 and $A_2 \rightarrow \varepsilon$ is in P_2.

$(A_1, A_2) \rightarrow (B_1, B_2)a$ whenever $A_1 \rightarrow B_1a$ is in P_1 and $A_2 \rightarrow B_2a$ is in P_2.

Clearly $(A_1, A_2) \Rightarrow (B_1, B_2)a_1 \Rightarrow (C_1, C_2)a_2\, a_1 ...$ in G if and only if
$A_1 \Rightarrow B_1a_1 \Rightarrow C_1a_2a_1 ...$ in G_1 and
$A_2 \Rightarrow B_2a_1 \Rightarrow C_2a_2a_1 ...$ in G_2.

So $t \in L(G)$ if and only if $t \in L(G_1)$ and $t \in L(G_2)$
that is $L(G) = L(G_1) \cap L(G_2) = L_1 \cap L_2$

So as G is a regular grammar it follows that $L_1 \cap L_2$ is a regular language. Q.E.D.

(c) Again, let us arrange that $N_1 \cap N_2 = \varnothing$. Let us start by using the grammar G from (a) which defined $L_1 \cup L_2$. Let us now convert G to an equivalent backward deterministic regular grammar G' using the set contsruction described earlier. So far $L(G') = L_1 \cup L_2$. As mentioned earlier, a terminal string can be derived from at most one non terminal in G'. Also recall that for $Z \in S$ and $Z' \in S'$ that $Z \overset{+}{\Rightarrow} t$ in G if and only if $Z' \overset{+}{\Rightarrow} t$ in G' for some $Z \in Z'$.

In view of this it follows that if Z' is a non terminal which has an element of S_1 but no element of S_2, then any terminal string derivable from Z' is derivable from an element of S_1. However it can not be derived from an element of S_2.

So if we change the definition of S' to:-

$S' = \{ U \in N : U \cap S_1 \neq \varnothing$ and $U \cap S_2 = \varnothing\}$, that is U contains an element of S_1 but no elements of S_2.

It follows that now $L(G') = L_1 \setminus L_2$ and that since G' is a regular grammar this means that $L_1 \setminus L_2$ is a regular language. Q. E. D.

(d) Again let us rename non terminals if neccessary so that $N_1 \cap N_2 = \varnothing$.

Let $G = (N_1 \cup N_2, T_1 \cup T_2, P, S_2)$. We wish to form P so that we start our derivation from S_2, that is in G_2 but instead of completing the derivation in G_2 by a rule of the form $A_2 \rightarrow \varepsilon$ we instead continue the derivation in G_1 and complete it in G_1, thereby forming t_1t_2, where $t_1 \in L(G_1)$ and $t_2 \in L(G_1)$. Of course t_2 is derived first and then preceded by t_1. We therefore construct P as follows:

(i) Initially $P = P_1 \cup P_2$
(ii) Every rule $A_2 \rightarrow \varepsilon$ in P where $A_2 \in N_2$ is replaced by $A_2 \rightarrow Z_1$ for all $Z_1 \in S_1$.
(iii) the rules $A_2 \rightarrow Z_1$ are eliminated as explained earlier. (See note (2) of the lemma in chapter 2 about removing rules of the form $A \rightarrow B$ from context free grammars.)

$L(G) = L_1L_2$ and G is a regular grammar, so L_1L_2 is a regular language. Q.E.D.

(e) A regular grammar G defining L_1^* is defined as follows:-

(i) Initially $G = (N, T, P, S)$ where $N = N_1$, $T = T_1$, $P = P_1$, $S = S_1$.
(ii) To enable us to generate a sequence of elements of L_1, whenever $A \rightarrow \varepsilon$ is in P (that is end of derivation) we send it back to the start, that is <u>add</u> to P the rules $A \rightarrow Z$ for every $Z \in S$.

(iii) The rules $A \to Z$ are eliminated as explained earlier. (See note (2) following the lemma in chapter 2 about removing rules of the form $A \to B$ from context free grammars.)
(iv) To ensure that ε is in L(G) add a new non terminal Z' to N and to S and add the rule $Z' \to \varepsilon$ to P. Q.E.D.

(f) A grammar G for the reversal L_1^R is constructed based on the desired property that $A_0 \Rightarrow A_1a_1 \Rightarrow A_2a_2a_1 \Rightarrow \ldots \Rightarrow A_na_n\ldots a_1 \Rightarrow a_n\ldots a_2a_1$ in G_1 if and only if $A_n \Rightarrow A_{n-1}a_n \Rightarrow A_{n-2}a_{n-1}a_n \Rightarrow \ldots\ a_1a_2\ldots a_n$ in G.

We achieve this by making the following reversals when constructing G from G_1.

$G = (N_1, T_1, P, S)$ where.

(i) A is in S if and only if $A \to \varepsilon$ is in P_1.
(ii) $A \to \varepsilon$ is in P if and only if $A \in S_1$.
(iii) $A \to Ba$ is in P if and only if $B \to Aa$ is in P_1.

Note (i), (ii) switch start and finish whereas (iii) reverses the string derived. Q.E.D.

This completes the proof of properties (a) to (f) for regular languages.

<u>Exercises</u>:

1) Let $G = (\{A, L\}, \{d, l\}, \{ A \to Ll, L \to Ll, L \to Ld, L \to \varepsilon\}, \{A\})$.
   a) Describe L(G) in set form.
   b) Convert G to backward deterministic form using the set construction.

2) Write regular grammars for :
   a) T* where T is a finite set of terminals
   b) $\{ a^m : m \text{ is even} \}$
   c) $\{ b^n : n \text{ is odd} \}$

3) Apply appropriate constructions from the previous theorem and your answers from the previous question to obtain regular grammars for:-
   a) $L = \{ a^mb^n : m \text{ is even and } n \text{ is odd} \}$
   b) L*
   c) L^R
   d) Use the set construction to convert your grammar for L to backward deterministic form.

4) The grammar G=(N, T, P, S) has rules of the form $A \to Bt$ and $A \to t$ where $A, B \in N$ and $t \in T^*$. Explain how to convert G into an equivalent regular grammar. (Also take care to handle the starting set S.)

5) Let $L_1 = \{a^{2n}: n\geq 0\}$ and $L_2 = \{a^{3n}: n\geq 0\}$.

a) Write regular grammars for L_1, L_2.

b) Construct a regular grammar for $L_1 \cap L_2$ using the method described in the previous theorem.

6) Explain in general terms how to recognize a regular language from right to left. (Hint: Try defining "reversed regular grammar" and "backward deterministic reversed regular grammar".)

Pumping Lemma for regular languages

Let L be any regular language. Then there is an M such that if $t\in L$ and $|t|\geq M$ then t can be written in the form $t=uvw$ where $v\neq\varepsilon$ and $|vw|\leq M$ and for all $i\geq 0$, $uv^iw\in L$

<u>Proof</u>: Since L is a regular language there is a regular grammar $G=(N,T,P,S)$ such that $L(G)=L$. Let M be the number of non terminals in N.

Consider any $t\in L(G)$ such that $|t|\geq M$. If no such t exists, then there is nothing to prove. If such a t exists, then its derivation from a starting non terminal Z must have a loop in it; that is some non terminal which we will denote by A must be repeated. So $Z\Rightarrow^{*}Aw\Rightarrow^{+}Avw\Rightarrow^{+}uvw=t$.

Clearly $v\neq\varepsilon$ as there is at least one step in the sub derivation $A\Rightarrow^{+}Av$. Also by choosing the first such A which repeats and its first repetition, we must have $|vw|\leq M$ as there are only M non terminals.

Of course the sub derivation $A\Rightarrow^{+}Av$ can be omitted altogether, or written once, or twice or as many times as we wish. So this allows to derive in G, $Z\Rightarrow^{+} uv^iw$. So $uv^iw\in L(G)$. Q.E.D.

<u>Example</u>: The pumping lemma can be used to <u>prove by contradiction</u> that a language is not regular e.g. $L=\{a^nb^n: n\geq 0\}$ is not a regular language.

If L were regular, then there is an M satisfying the conditions of the pumping lemma. Let us consider a^Mb^M. Certainly $|a^Mb^M|\geq M$ and so we can write it in the form $a^Mb^M=uvw$ where $|vw|\leq M$ and $v\neq\varepsilon$.

However as $|vw|\leq M$, vw must consists only of b's so certainly v must contain only b's. So if we form uv^iw for $i\geq 2$, these strings will have more b's than a's and so cannot be of the form a^nb^n and so $uv^iw\notin L$. A contradiction to the pumping lemma. So L can not be regular. Q.E.D.

<u>Exercise</u>: Let $L=\{a^n$: such that n is a perfect square$\}$.
Use the pumping lemma to show that L is not a regular language.

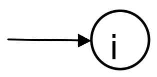
4. FINITE (STATE) AUTOMATA

We will not present this formally as these are entirely equivalent to regular grammars. We will indicate the connection later. A finite automaton can be defined by a (state) transition diagram.

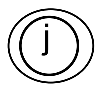
An automaton scans it input from left to right and changes state according to the diagram. It starts from an initial state and scans the first symbol. If it scans the entire input and finishes in a final state it accepts the input. If it can not complete the scan or completes the scan but can not finish in a final state, it rejects the input. The following notation is used.

State:

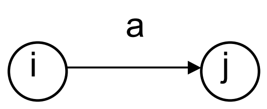
Initial State:



Final State:

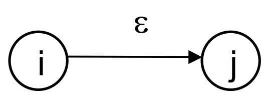


Transition:



where a is a terminal (element of the input alphabet). This means that when in state i and current input is "a" change to state j and input next symbol.

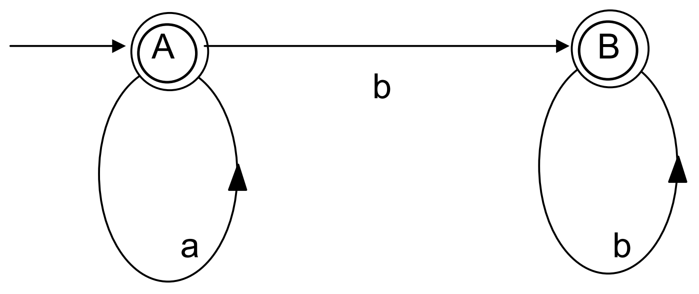
ε-transition:



This means move from state i to state j without input of a new symbol.

The language of a finite automaton is the set of terminal strings it is possible to accept (see above).

Example of a finite automaton accepting $\{a^m b^n : m,n \geq 0\}$



Here A is the initial state and both A and B are final states.

Class discussion: Are the strings ε, aabbb, bba, bbb, accepted by the above automaton?

A finite automaton can also be described in a mathematical style (without a diagram). A state transition function δ gives the new state.

that is δ(state, input) = new state.

This function is typically described by a table

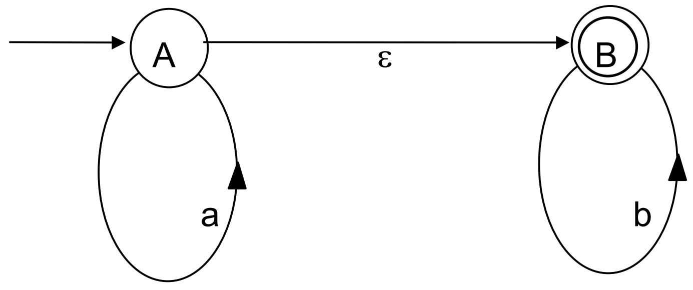
For example, the previous diagram is equivalent to the following description.

Terminals (Input Alphabet):	{a,b}
States:	{A,B}
Set of initial States:	{A}
Set of Final States:	{A,B}

Table defining δ(state, input) which gives the new state:

	Input	
State	a	b
A	A	B
B	-	B

Here is another automaton for $\{a^m b^n : m,n \geq 0\}$



Exercise: Write the above automaton using a table as above without a diagram.

Equivalence between Finite Automata and Regular Grammars

In fact, any finite automaton can be converted into a regular grammar and also vice versa. The following correspndence is used:

Regular Grammar	Finite Automaton
T	Terminals (Input Alphabet)
N	States
S	Final States
$A \rightarrow \varepsilon$	A is an initial state
$A \rightarrow Ba$	$B \xrightarrow{a} A$

If the automaton has an ε - transition such as $\begin{tikzcd} B \arrow[r, "\varepsilon"] & A \end{tikzcd}$,
then the grammar rule will be $A \to B$, but this must be removed, as already explained, so that the grammar will be regular.

In this way we can convert between Regular Grammars and Finite Automata in both directions.

For example, the grammars G_1, G_2 corresponding to the previous two automata are:-

G_1 = ({ A, B}, {a,b}, P_1, {A,B}) where the rules in P_1 are

$A \to Aa$ $\qquad$ $A \to \varepsilon$
$B \to Bb$ $\qquad$ $B \to Ab$

G_2 = ({A,B}, {a, b}, P_2, {B}) where the rules in P_2 are

$A \to Aa$ $\qquad$ $A \to \varepsilon$
$B \to Bb$ $\qquad$ $B \to A$

Notes: 1) G_1 is a backward deterministic regular grammar.
2) Eliminating $B \to A$ from G_2 will give us a regular grammar which is not in backward deterministic form.

More analogies between finite automata and regular grammars:

1) ε - transitions can always be eliminated - similar to removing rules of the form $A \to B$ from grammars.

2) A <u>deterministic</u> finite automaton has no transitions of the form

$$\begin{tikzcd} A \arrow[r, "a"] \arrow[d, "a"] & C \\ B & \end{tikzcd}$$

where $B \neq C$.

It also has no ε - transitions and has at most one initial state. Otherwise, the finite automaton is non-deterministic (The first automaton we presented is deterministic, the second is non-deterministic).

Deterministic finite automata correspond to backward deterministic regular grammars.

3) Similarly any finite automaton can be made deterministic by using the set construction. (Similar to converting regular grammars to backward deterministic form using the set construction.)

<u>Exercises</u>:

1) (a) Without converting finite automata to regular grammar, explain how ε-transitions may be removed.

(b) Construct a finite automaton without ε-transitions for the second automaton we presented.

2) Present the derivation of aabbb backwards according to G_1 and forwards according to the first automaton and see the similarity.

3) The set construction for converting a regular grammar into backward deterministic form can be rewritten so as to convert a non deterministic finite state automaton into deterministic form. Rewrite the set construction for finite automata with the help of the table showing the equivalence between finite automata and regular grammars.

Extensions of finite automata (Moore and Mealy Machines)

It is useful to associate an "output" or "action" with finite automata to extend their use to applications such as translation (use output) or real time (use action) or for entering data into compiler tables (use action). Two approaches have been used and have been shown to be equivalent. Moore proposed associating the output or action with the state and Mealy proposed associating the output or action with the transition.

5. REGULAR EXPRESSIONS

These expressions denote sets and are defined as follows:

(i) $\varnothing$ is a regular expression:
(ii) $\{\varepsilon\}$ is a regular expression
(iii) {a} is a regular expression where "a" is a terminal symbol.
(iv) If R and S are regular expressions then the following are also regular expressions.

(a) R|S	- the union
(b) RS	- the concatenation
(c) R* (also S*)	- the transitive closures

<u>In practice</u> we will write ε and not $\{\varepsilon\}$, "a" and not {a}. Of course the set {a,b,c,d} is then written (a|b|c|d) and of course $\{a,b\} \cup \{c,d\}$ would be written (a|b) | (c|d) as regular expressions.
Incidentally, $\varnothing^* = \varepsilon$ in regular expressions.
Regular expressions are useful for defining the structure of reserved words, identifiers, numbers occuring in programming languages but are not suitable for defining the structure of blocks, expressions, statements etc.

Examples of Regular Expressions:

1) Unsigned Integers:
(0|1|2|...|9)(0|1|2...|9)*

2) Integers with an optional sign
(ε|+|-)(0|1|2|...|9)(0|1|2|...9)*

3) Identifiers:
(a|b|c|...|z)(a|b|c|...|z|0|1|2...|9)*

<u>Exercises</u>: 1) Write a regular expression defining the structure of a real number which may include a sign, a decimal point and an exponent which are optional.

2) Change the definition of the first example above so as to exclude numbers with unneccessary leading zeroes, i.e. to exclude numbers such as 000, 00156, etc.

Equivalence between regular expressions and regular grammars

We shall now show that the sets defined by regular expressions and regular grammars are equivalent.

Theorem: Converting regular expressions to regular grammars

Any set defined by a regular expression is definable by a regular grammar.

<u>Proof:</u> The sets $\varnothing$, $\{a\}$, $\{\varepsilon\}$ can clearly be defined by regular grammars. Also we proved earlier that if L_1 ,L_2 are regular languages then so too are $L_1 \cup L_2$ L_1L_2, L_1^*, L_2^*.

But $L_1 \cup L_2$ is just another way of writing $L_1 \mid L_2$. Therefore $L_1|L_2$, L_1L_2, L_1^*, L_2^* are all regular languages. So every set defined by a regular expression uses only these operations and so must be a regular language, that is definable by a regular grammar.

Theorem: Converting regular grammars to regular expressions

The language of any regular grammar can be defined by a regular expression.

<u>Proof:</u> Let L be a language defined by a regular grammar, say G=(N,T,P,S). That is L=L(G). Suppose that G is in backward deterministic form. (This can always be arranged.) There are two cases to consider.

a) There is no non terminal such that $A \rightarrow \varepsilon$, or, $S=\varnothing$.
   In these cases, $L(G) = \varnothing$. Clearly $\varnothing$ is a regular expression.

b) There is such a non terminal say A such that $A \rightarrow \varepsilon$ and in addition $S \neq \varnothing$.

Now as G is backward deterministic, A is in fact unique. Suppose that all the non terminals are numbered A_1, A_2, ..., A_m in such a way that A_e is <u>the</u> non terminal such that $A_e \rightarrow \varepsilon$. Let us also suppose that the non terminal in the start set S are all at the front of this numbering that is $S = \{A_1, A_2, ..., A_s\}$ for a suitable s.

Let us now define for k = 0,1,...,m
$L^k_{ij} = \{t \in T^*$: $A_j \Rightarrow^* A_i t$ such that the non terminals used in the intermediate steps (if any) all have index less than or equal to k.$\}$

We shall show (by induction on k) that all the L^k_{ij} can be defined by regular expressions. (Afterwards we show how to define L(G) in terms of L^k_{ij})

<u>Basis:</u> k = 0 As there are no non terminals with index 0, the only derivations are direct (that is one step).
So $L^0_{ij} = \{t:(A_j \rightarrow A_i t$ is a rule) or ($t= \varepsilon$ and $i =j)\}$

This must be a finite set consisting of terminals or ε and so can be written as a regular expression of the form $(t_1|t_2|...t_p)$ where t_1, t_2, ... t_p are the elements of L^0_{ij}.

So clearly L^0_{ij} can be defined by a regular expression for all i, j.

Inductive step: Let us suppose that all the sets L^k_{ij} can be defined by a regular expression.

What about the sets $L^{(k+1)}_{ij}$?

In fact $L^{(k+1)}_{ij} = L^k_{ij} \mid (L^k_{i\,(k+1)}\ (L^k_{(k+1)\,(k+1)})^*\ L^k_{(k+1)\,j})$

which is in fact is a regular expression built from the regular expressions for L^k_{ij}.

Hence by induction, L^k_{ij} can be defined by a regular expression for k = 0,1,2...m.

Now recall that A_e is the only non terminal for which $A_e \rightarrow \varepsilon$ and that $S = \{A_1, A_2, ...A_s\}$

So $L(G) = \{t \in T^*: A_j \Rightarrow^* t \text{ for } 1 \le j \le s\}$
$= \{t \in T^*: A_j \Rightarrow^* A_e t \Rightarrow t \text{ for } 1 \le j \le s\}$
(This is because the last step must use $A_e \rightarrow \varepsilon$.)
$= L^m_{e1} \mid L^m_{e2} \ldots \mid L^m_{es}$
which is a regular expression. Q.E.D.

Exercise: It is highly desirable not to convert the grammar to backward deterministic form in the above, as the set construction can create large grammars. Explain how you would rewrite the previous proof without having to assume that the grammar is backward deterministic.
Hint: Assume that that the non terminals are numbered $A_1, A_2, ..., A_m$ in such a way that $S = \{A_1, ..., A_s\}$ and $\{A: A \rightarrow \varepsilon\} = \{A_e, A_{e+1}, ..., A_f\}$. Why is this always possible? Now how do we proceed?

Example: Consider the grammar $G = (\{A_1, A_2\}, \{d\}, P, \{A_1\})$ where P consists of

$A_1 \rightarrow A_1 d \qquad A_1 \rightarrow A_2 d \qquad A_2 \rightarrow \varepsilon$

G is backward deterministic and the non terminals are numbered as required by the theorem. Of course m = 2, e = 2, s=1.

The regular expressions for L^k_{ij} are as follows

k	i	j	L^k_{ij}
0	1	1	$(d \mid \varepsilon)$
0	1	2	$\varnothing$
0	2	1	d
0	2	2	ε
1	1	1	$L^0_{11} \mid (L^0_{11} (L^0_{11})^* L^0_{11})$
1	1	2	$L^0_{12} \mid (L^0_{11} (L^0_{11})^* L^0_{12})$
1	2	1	$L^0_{21} \mid (L^0_{21} (L^0_{11})^* L^0_{11})$
1	2	2	$L^0_{22} \mid (L^0_{21} (L^0_{11})^* L^0_{12})$
2	1	1	...
2	1	2	...
2	2	1	...
2	2	2	...

<u>Exercises:</u>

1) (a) Complete the above table by substituting for L^0_{ij} from first four rows of the table to obtain L^1_{ij}. Carry on in the same way to get L^2_{ij}.

(b) How is L(G) determined from L^k_{ij} ?

2) Suppose that the sets S_1 and S_2 can be defined by regular expressions. Give a general explanation regarding how to construct regular expressions for $S_1 \cap S_2$ and $S_1 \setminus S_2$.

Another Formulation of Regular Languages

Certain kinds of equivalence relations can also be used to define the regular languages (Myhill and Nerode). They used their formulation to find a minimal finite automata for a regular languages as well as showing that the minimal automata is essentially unique, apart from renaming of the states. We shall not present this formulation.

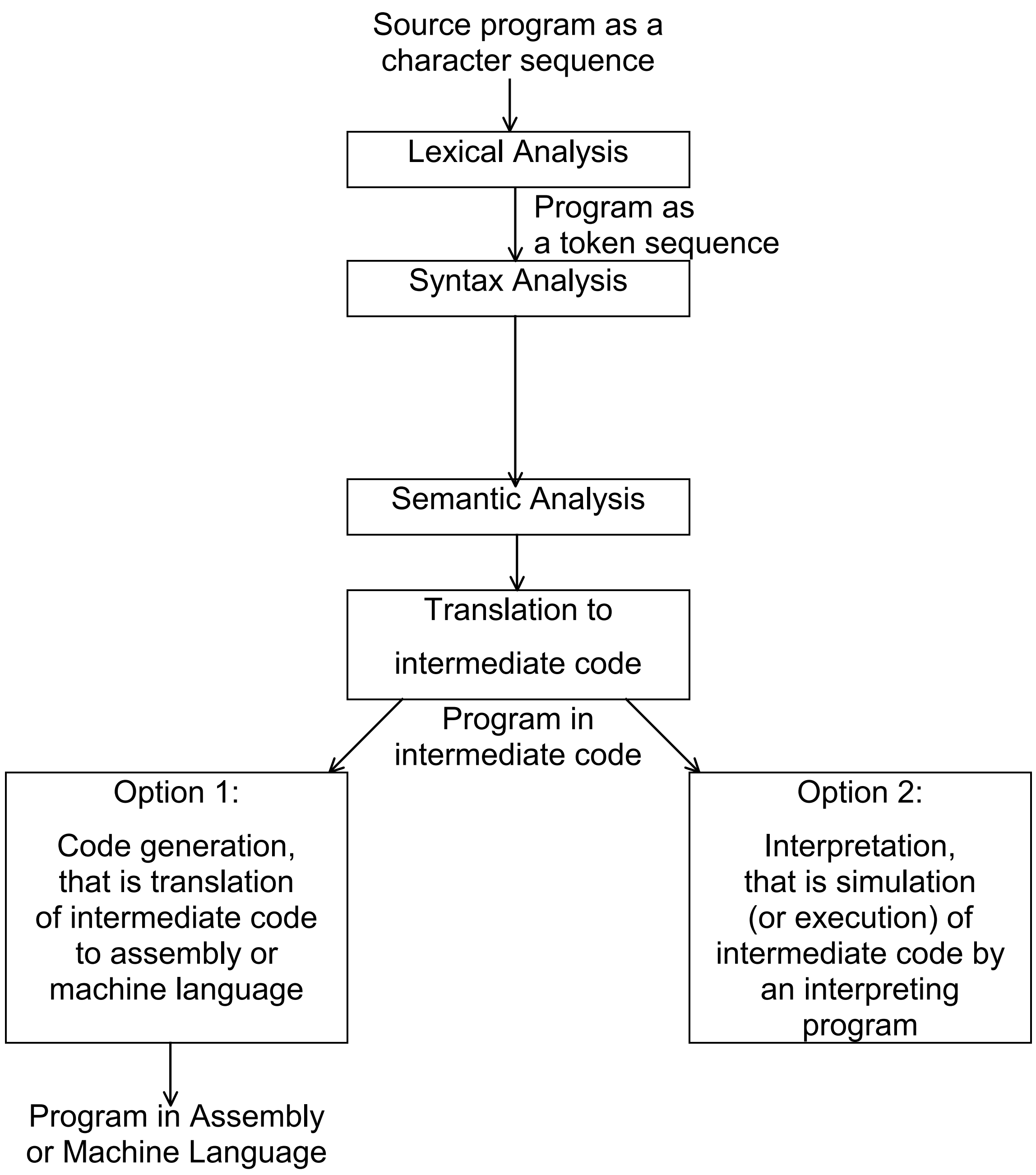
Discussion

We have presented three formalisms for regular languages (regular grammars, finite automata and regular expressions), and shown their equivalence. The formalism of Myhill and Nerode using certain equivalence relations is a further equivalent formalism.

The advantage of having different formalisms for regular languages is that one can use whatever is most convenient e.g. a regular expression to define the set and a finite automaton to recognize the set. Also, the existence of different formalisms is an indicator of the importance of this topic.

As the conversion from one formalism to another can be laborious, software is used to make this conversion. For example the Lex software takes regular expressions (together with actions which are C statements) and converts it to a deterministic finite automata with actions which are C statements. This software is useful for building the lexical analyzer phase of compilers.

6. SURVEY OF COMPILER AND INTERPRETER STAGES


Source program as a character sequence
Lexical Analysis
Program as a token sequence
Syntax Analysis
Semantic Analysis
Translation to intermediate code
Program in intermediate code
Option 1: Code generation, that is translation of intermediate code to assembly or machine language
Option 2: Interpretation, that is simulation (or execution) of intermediate code by an interpreting program
Program in Assembly or Machine Language

The above diagram shows the stages of compilation and interpretation of a program in a high level language. It presents the logical view. The arrows do not necessarily represent sequential execution. Phases can be run in parallel by using a pipe or buffer between them for data transfer. This reduces the number of compiler passes that is the number of times the source program needs to be scanned.

<u>Class discussion:</u> Why is a compiler organized in this way and what are the benefits.

Lexical Analysis

This stage is responsible for carrying out the folloowing actions:
- Representing the program as a token sequence.
- Building the table of tokens (symbol table).

For example for the program fragment,

```
SUM:= 0;
FOR I:= 1 T0 25 DO SUM:=SUM + I;
```

the token sequence and table of token entries are:

<u>Token sequence</u>

0
1
2
6
3
4
1
5
7
10
8
0
1
0
9
4
6

<u>Table of Tokens</u> (Symbol Table)

	Token	Properties
0	SUM	identifier
1	:=	operator
2	0	constant
3	FOR	keyword
4	I	identifier
5	1	constant
6	;	punctuation
7	TO	keyword
8	DO	keyword
9	+	operator
10	25	constant

Syntax Analysis

The compiler has to check that the program is constructed according to the rules of the grammar. There are two main approaches to syntax analysis, namely, bottom up (shift reduce) and top down (predictive) methods. (Later, we also discuss the recursive descent method which is in fact a kind of top down method.)
Here is an example of the two main methods for the input: ((id+id)+id)
using the grammar G = ({ E }, { (,), +, id }, { E$\rightarrow$id, E$\rightarrow$(E+E) }, { E })

Bottom Up

Stack (Top at right)	Input	Comments
ε	((id+id)+id)	shifts
((id	+id)+id)	reduce
((E	+id)+id)	shifts
((E+id	)+id)	reduce
((E+E	)+id)	shift
((E+E)	+id)	reduce
(E	+id)	shifts
(E+id	)	reduce
(E+E	)	shift
(E+E)	ε	reduce
E	ε	accept

Top Down

Stack (Top at left)	Input	Comments
E	((id+id)+id)	replace
(E+E)	((id+id)+id)	erase
E+E)	(id+id)+id)	replace
(E+E)+E)	(id+id)+id)	erase
E+E)+E)	id+id)+id)	replace
id+E)+E)	id+id)+id)	erase
E)+E)	id)+id)	replace
id)+E)	id)+id)	erase
E)	id)	replace
id)	id)	erase
ε	ε	accept

Semantic Analysis

Here the following tasks are performed by the compiler:

1) Each definition of an identifier is replace by a unique name (unique token), and information about each unique name is added to the table of tokens. Also each occurrance of an identifier is replaced by the appropriate unique name. For example, suppose that the first occurence of TOTAL denotes a procedure, the second and third occurences denotes an integer, and the fourth occurence of TOTAL denotes a procedure. Then by using two entries for TOTAL in the table of tokens, we can correctly hold the different definitions of these names.

Here is how the token sequence and table of tokens will change from lexical analysis to semantic analysis to handle this kind of situation.

Token sequence — Table of Tokens

After	12 ...	
Lexical	12 ...	
Analysis	12 ...	
	12 ...	

	Token	Properties
12	TOTAL	identifier

After	12 ...	
Semantic	201 ...	
Analysis	201 ...	
	12 ...	

	Token	Properties
12	TOTAL	procedure
....	.....	.....
201	TOTAL	integer

2) A check is made that there is no improper use of any name - for example that we do not add A+B where A,B are procedure names.

Note the distinction between identifier and names. The names are unique but the identifiers need not be unique. The name is determined from the identifier and the context.

Exercise: How many entries do we need in the table of tokens for "-"?
Can "-" be a part of other tokens and if so which ones?

Intermediate Code Generation

There are several possibilities for the intermediate code form of the program.

1) The derivation tree itself may be used
2) A postfix (reverse Polish) form may be used
3) Three address code may be used, which we shall discuss further.

Three address code can be written in either of the styles "A:= B op C", or "op A,B,C", where "op" is any operator. For example the expression A/B*C+D would be written as follows in these styles.

Style 1	Style 2	
T1:= A/B	DIV	T1, A, B
T2:= T1*C	MUL	T2, T1, C
T3:= T2 + D	ADD	T3, T2, D

Note the need for temporary varables T1, T2, T3 when translating to three address code.

Exercise: Translate A/B*C+D to postfix. Is there a need for temporary variables in the translation?

Code generation (Option 1, in the diagram)

The intermediate code is translated into assembly or machine language. For example for a single register machine, the intermediate code

```
A:=B+C    --   ADD A,B,C
```

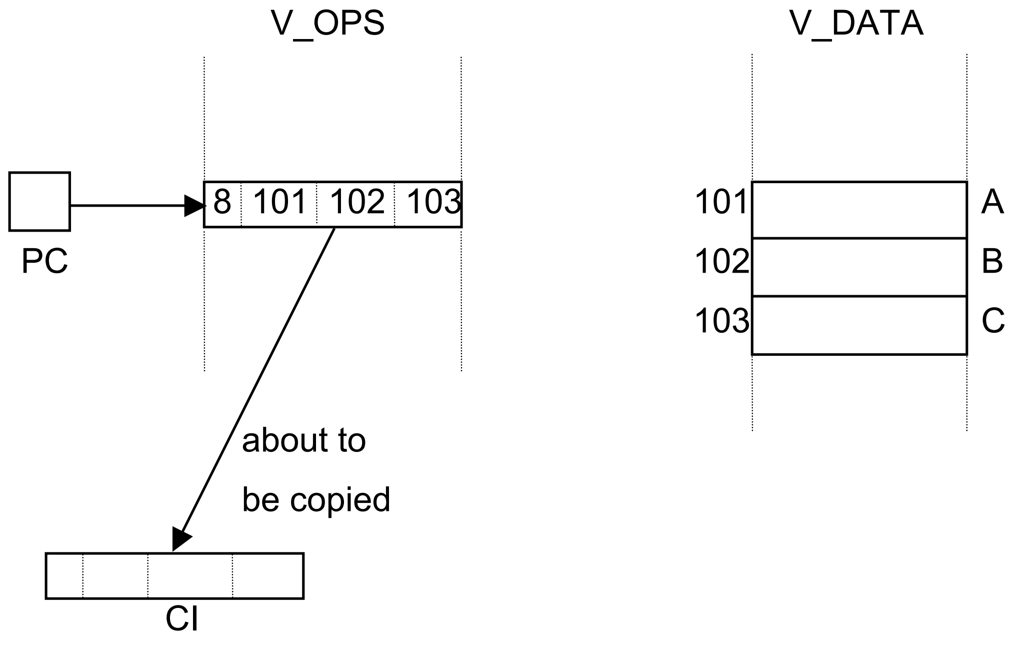
would be translated to:-

```
        LOAD B
        ADD C
        STORE A
```

Code generation may appear simple. However, using all the registers efficiently, is a complicated task.

Interpretation (Option 2 in the diagram).

An interpreting program has to simulate or execute the operations and commands of the intermediate code. Typically V_OPS a vector of operations is intermediate code and V_DATA a vector of data will be used together with a variable PC simulating the program counter and a variable CI holding the instruction to be interpreted (executed or simulated). Suppose for example that A:= B+C - - ADD A,B,C is about to be interpreted (executed or simulated) and that A,B,C are at positions 101, 102, 103 in the data vector and that the opcode of + is 8. Then we will have a situation such as:-



Here is how a typical interpreter would use the information in PC, CI, V_OPS and V_DATA to simulate or execute operations in intermediate code.

```
PC:= 0;       - -    initialization
LOOP
        CI:= V_OPS(PC);
        PC:= PC + 1; - - advance simulated program counter
        CASE CI.opcode OF
                WHEN 0 => -- HALT operation
                        EXIT;
                . . . .
                WHEN 8 => --ADD operation
                        V_DATA(CI.address1):= V_DATA(CI.address2) +
                                               V_DATA(CI.address3);
                . . . .
                WHEN 13 => -- JUMP instruction uses only the field address1.
                        PC:= CI.address1;
                . . . .
        OTHERWISE => error;
        END CASE;
END LOOP;
```

Exercise: Define the variables PC, CI, V_DATA, V_OPS including their type definitions. (For simplicity assume that basic data are of type integer.)

Optimization

In our diagram of the phases of a compiler this did not appear. This task may be carried out in the source language before compilation, in intermediate code, or in assembly or machine language. The compiler run's slower but the translated program run's faster and/or may use less memory. Here are two examples of optimization.

Loop optimization in the source program:

```
FOR R IN 1..100
LOOP  - - LOTS OF
        - - CALCULATIONS
        PI:= 4 * ARCTAN (1);
        AREA:= PI*R*R;
ENDLOOP;
```

The execution is more efficient if the statement PI:= 4 * ARCTAN (1) is moved before the FOR loop and in principle it can move quite far in the program. Loop optimization is an example of global optimization and requires working on large parts of the program.

Peephole optimization in assembly or machine language:

For example if the intermediate code for

```
A:= B + C        - -        ADD A,B,C
X:= A * Z        - -        MUL X,A,Z
```

is translated to assembly or machine language we obtain:

```
LOAD B
ADD C
STORE A
LOAD A
MUL Z
STORE X
```

The operation LOAD A is unnecessary and can be eliminated by examining through a peephole pairs (or a few) instructions and making improvements in the peephole only. Here delete LOAD A after a STORE A. Peephole optimization is harder to perform in machine language in view of the need to update jump and data addresses for example. Peephole optimization is an example of local optimization.

Table handling

This routine (not shown in the diagram) updates or accesses information in the table of tokens. It is used by the other compiler phases.

Error handling

Each phase has the potential of detecting and reporting an error. Such a routine would collect these reports, sort them according to position of occurrence and give an orderly listing of all errors. This routine (not shown in the diagram) is responsible for deciding how to continue after an error is detected. However in an interactive environment it is possible to stop on the first error detected and immediately activate the editor and position its cursor at the postition of the error in the program.

Exercise: What are the relative advantages and disadvantages of fixed as apposed to free program format?

7. SIMPLIFYING THE PROCESS OF BUILDING A COMPILER

A great deal of effort is needed to build a compiler, and we shall present two methods for saving much effort.

Notation

Let us use the notation ${}^{L}C^{M}_{A}$ to denote a compiler which translates from language L to language M and is written in language A. (Yes - writing a compiler may involve you working with three languages.)
When this compiler is executing and translating a program P_L in language L to the same program P_M but in language M we shall write

$P_L \rightarrow \boxed{{}^{L}C^{M}_{A}} \rightarrow P_M$ (The boxed item is being executed.)

Bootstrapping a compiler

Suppose we wish to build the compiler ${}^{L}C^{A}_{A}$ from high level language L to assembly or machine language A.
The following approach will save much effort.
(i) Identify a subset S of the language L which is easier to compile.
(ii) Write the compiler ${}^{S}C^{A}_{A}$.
(iii) Write the compiler ${}^{L}C^{A}_{S}$.
(iv) Execute the following.

$${}^{L}C^{A}_{S} \rightarrow \boxed{{}^{S}C^{A}_{A}} \rightarrow {}^{L}C^{A}_{A}$$

Cross compilation

Here are two uses of this technique.

1) If the machine (e.g. microcontroller) with language M does not have the resources to do the compilation itself then we can use another machine with language A and run the compiler on machine with language A and just transfer the translated program onto the machine with language M. For example when ${}^{L}C^{M}_{A}$ runs on machine with language A then execute

$$P_L \rightarrow \boxed{{}^{L}C^{M}_{A}} \rightarrow P_M$$

and transfer P_M to run on machine with language M. This technique allows the use of high level languages on machines with very limited resources by using an auxiliary computer to do the compilation.

2) Another use of this concept is to create a compiler for a new machine by using a compiler for an existing machine. Suppose a machine exists which can run ${}^{L}C^{A}_{A}$ where L is a high level language. Suppose

we wish to build ${}^{L}C^{B}_{B}$ for a new machine which runs the language B. The following method greatly simplifies the task.

(a) Write ${}^{L}C^{B}_{L}$ - that is a compiler is written in the high level language L.

(b) Execute the following.

$$ {}^{L}C^{B}_{L} \rightarrow \boxed{{}^{L}C^{A}_{A}} \rightarrow {}^{L}C^{B}_{A} $$

$$ {}^{L}C^{B}_{L} \rightarrow \boxed{{}^{L}C^{B}_{A}} \rightarrow {}^{L}C^{B}_{B} $$

The subtlety in this method is that the compiler ${}^{L}C^{B}_{L}$ we write is translated twice, and we obtain both a cross compiler ${}^{L}C^{B}_{A}$ and the compiler ${}^{L}C^{B}_{B}$.

<u>Exercises</u>:

1) Suppose that the compilers ${}^{L}C^{A}_{B}$ and ${}^{L}C^{B}_{L}$ are available. Suppose also that you have access to two computers for running machine language A and machine language B. How can these resources be used to construct the compiler ${}^{L}C^{B}_{B}$.

2) Is there any value in writing a Compiler ${}^{L}C^{L}_{L}$ where L is a high level language?

8. LEXICAL ANALYSIS

The following example regular expression defines tokens of a language. It is written in a form that each line in the regular expression describes a different token in the language.

```
BEGIN |
END |
IF |
THEN |
ELSE |
letter (letter | digit)* |
digit digit* |
< |
<= |
= |
<> |
> |
>=
```

The following two pages are from “Principles of Compiler Writing” by Aho and Ullman and show non-deterministic and deterministic finite state automata for the above regular expression.

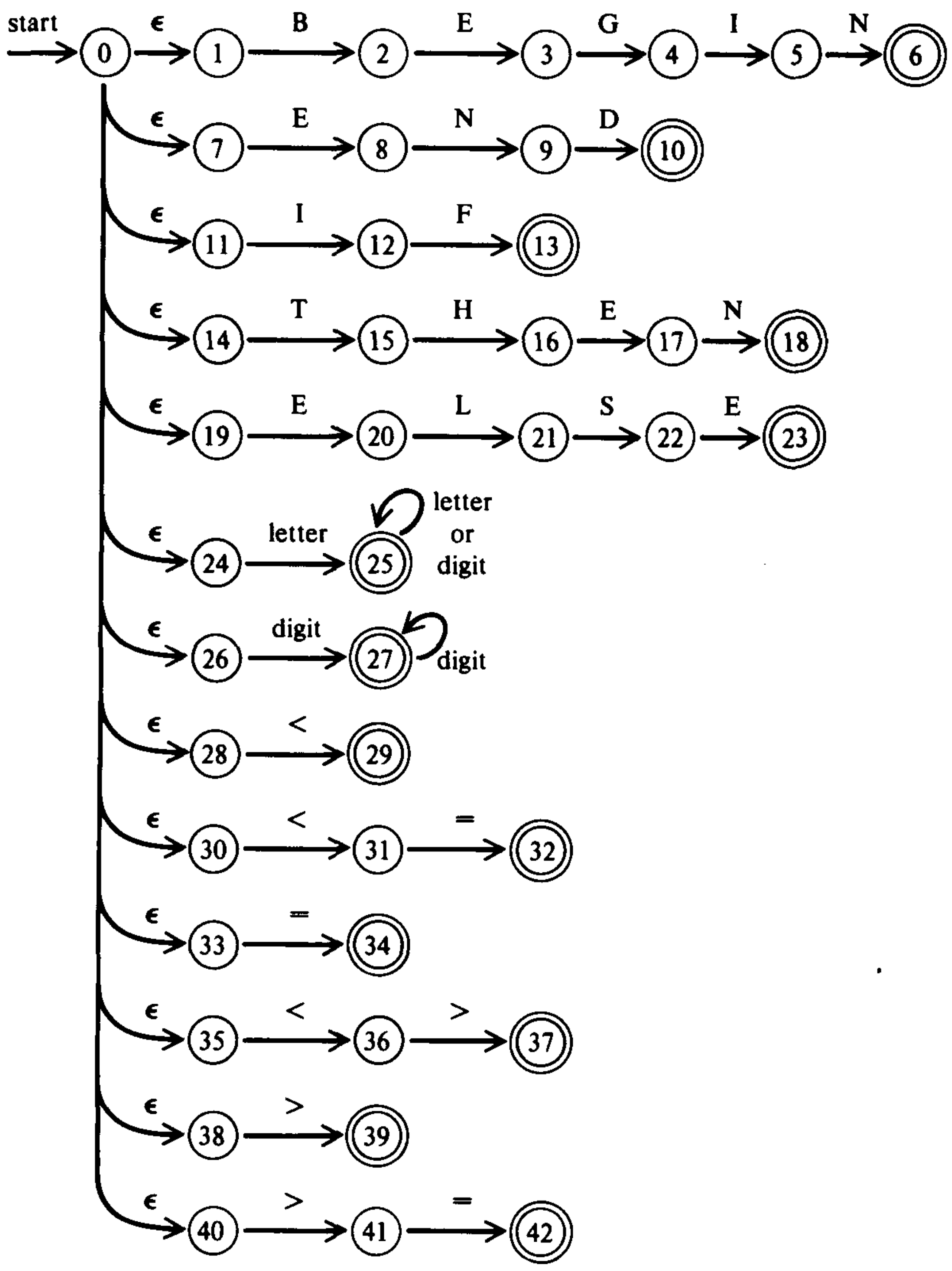


Fig. 3.21. Combined NFA for tokens.

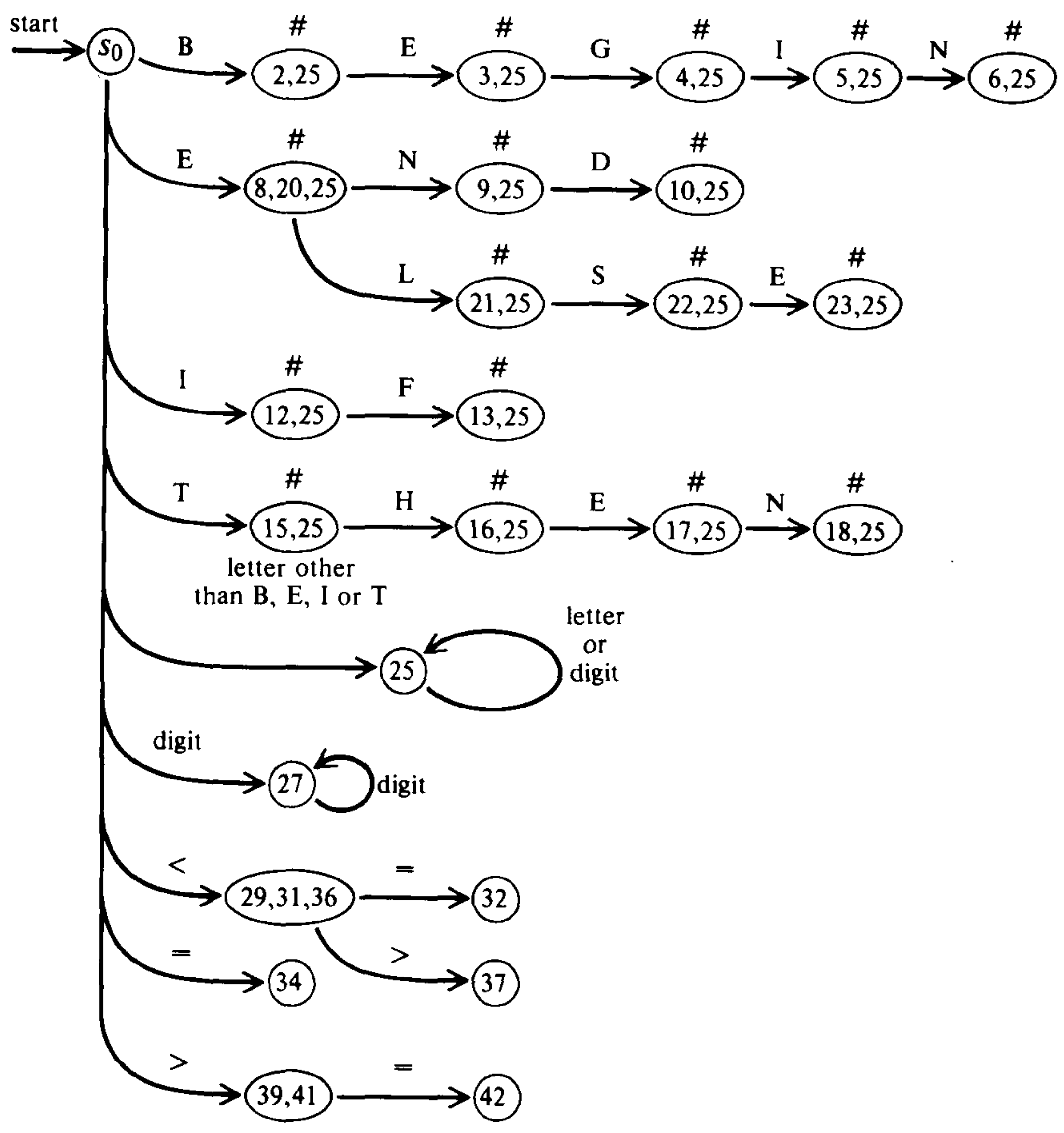


Fig. 3.22. DFA constructed from Fig. 3.20.

Note that "#" means "otherwise go to state 25".

5 <u>Exercise</u>:

Convert the regular expression (a|b)* | (cab)* into a backward deterministic regular grammar, and then into a deterministic finite automaton.

9. PROGRAMMING LANGUAGE DESIGN AIMS

1) It should not be difficult to write programs which are easy for anyone to understand.
2) It should not be difficult to write programs to solve the problems you need to solve.
3) Effective methods should be available for defining the syntax and semantics of programming languages.
4) Efficient algorithms should be available for recognizing legal programs.
5) Programs should be easy to translate to any machine language.
6) The translated programs should run efficiently.
7) Preferably programming errors should be prevented or easily detected and corrected.

However, programming is hard and designing and implementing programming languages is very much harder. Nonetheless, there has been progress in achieving these aims.

Class discussion: Which of these aims are important to the programmer, his supervisor, the language designer, the writer of a compiler or interpreter, a computer manufacturer, a software producer? Give examples illustrating some design aims.

Exercise: In ADA the index of a FOR loop is a local variable of the loop. What advantages does this give to the compiler writer?

The definition of programming languages

Syntax: The syntax of assembly languages (except for expressions) can be defined by a regular expression or grammar. However, for high level languages context free grammars are needed, but are not fully adequate. So additional restrictions concerning the form of a legal program are defined in natural language (English).

Semantics: In practice the meaning of the program is defined in natural language (English). The following formal (mathematical) methods have been proposed but are rarely used in practice.

1) Translational Semantics 2) Operational semantics
3) Axiomatic definition 4) Extensible definition
5) Mathematical or Denotational semantics

Aids for the compiler writer

The kinds of languages which are easy to recognize are now well understood, and tools are available for building recognizers directly from the language definition, for example:-

LEX -	builds a deterministic finite automata from regular expressions. This is useful for building a lexical analyser.
YACC -	builds a bottom up (shift reduce) syntax analyser from a context free grammar.
DCG -	Enables top down syntax analysis for a context free grammar from the PROLOG language.

10. PROGRAMMING LANGUAGE IMPLEMENTATION ISSUES

A high level rogramming language has a hierarchic structure for statements expressions and definitions of functions, procedures, data and their types. Assembly language has a sequential structure (except for expressions) and does not support a large variety of data types. Machine language is completely sequential in structure and has no concept at all of data type. There is therefore a large gap between a high level language and machine or and the main problem to be handled is this large gap. It is for this reason that there are many phases in compilation and interpretation, where each phase handles a part of the problem. The multiphase design makes also it easier to transfer the compiler to another machine. This is because the code generation phase and the interpretation phase are the only really machine dependent phases in the compiler.

Let us now give examples of different programming language features, associated problems and some solutions.

Code generation or interpretation - Associating properties with names

Recall first the distinction between identifier and name, that is names are unique but identifiers can have multiple definitions. Certain languages (such as Fortran IV) allow determining all the properties of a name at compile turns e.g. type of data, its size, and its absolute run time address. Other languages such a Pascal, need the absolute address to be determined at run time even though sizes are all known. At compile time, relative addresses are known e.g. relative to top of a stack.
Other languages such as ADA, the size of data is not known and even this needs to be determined at run time e.g. on block entry. In languages such as LISP and PROLOG the type may only be known at run time. In view of these approaches to languages, different implementations are used. Code generation can be used when the types of the data are known at compile time but is much harder or impossible to use when the data types are known at run time, and interpretation is a simpler solution in this case.

Type declarations and checking

Here are three possibilities for programming languages.

<u>Strongly typed language:</u> Type declarations and compile time type checking. e.g. Pascal, ADA
<u>Weakly typed language:</u> Type declarations but no type checking.
e.g. assembly language, C.
<u>Utyped language:</u> No type declarations but run time type checking.
e.g. LISP, PROLOG

Basic data types and their operations

Data type	Operation
Numbers:	+,-,*,/,**,=,<,>,…
real	
complex	
multiprecision	
Boolean	not, and, or, = , < , > etc
Character	=, < , >, etc
Pointers	↑X : the value at address X.
	@Y, addr(Y) : the address of variable Y.
	=, ≠

There is no major difficulty here except that operations have more than one meaning and the technique we described in the section on semantic analysis for handling multiple definitions of identifiers is suitable for dealing the multiple meaning of operations on basic data types. Care also needs to be taken regarding the priorities of the operators and whether they associate to the left and right and so the language definition has to be clearly understood. For example does
-I**Z mean - (I**Z) or (-I)**Z?
Does A**B**C mean (A**B)**C or A**(B**C)?
A more complex operation is the condtional expression, such as,
"(if A>0 then B else C) + D".

Exercises:
1) Explain what is the value of the expression (if A>0 then B else C) + D.
2) What is the difference between a conditional expression and conditional statement
3) Write a conditional statement without a conditional expression equivalent to
R := (if A>0 then B else C) + D .

Compound data structures and their operations

Data structure	Operation
Arrays (fixed or varying size)	indexing
Records or struct	field selection
Strings (fixed or varying size)	concatenation, substitution, search, substring
Linked structures (lists, trees, graphs)	creation, insertion, deletion, update, search

A common problem here is how do we calculate the address of an element in the structure. How do we allocate space for a newly created structure (NEW) and how is space freed (DISPOSE or garbage collection). In the next section we will give an example of address calculation as part of an assignment statement, and later in the secton and storage allocation we will discuss how storage is allocated and freed.

Exercise: Write an algorithm for calculating the address of a field in a nested record with respect to the start of the outermost record.

Assignment

Care needs to be taken here as a variable has two meanings. On the left hand side of the assignment a variable denotes an address (also called l-value) while on the right hand side it denotes a value (also called r-value). Assignment on basic data types is simple to handle but it is more difficult to handle on compound data structures. In the general situation the following needs to be done.
1) Calculate the value of the right hand side and store it in a register.
2) If neccessary, convert the type of the right hand side to the type of the left hand side.
3) Calculate the address of the left hand side of the assignment.
4) Move the value calculated at step 1, to the address calculated at step 3.

For example on the assumption that each array element occupies 4 bytes and X, Y are of the same type, the assignment X(I):= Y(J) would be translated by the compiler to something like:-

```
LOAD J, R2
MULT #4, R2          -- calculate relative position
LOAD Y(R2), R1       -- get value
LOAD I, R2
MULT #4, R2          -- calculate relative postition
STORE R1, X(R2)      -- store value
```

Parameter passing

We describe four methods.

1) Call by reference - the address of the actual parameter is passed to the function or procedure
2) Call by value - the value of the actual parameter is transferred in a local variable to the function or procedure.
3) Copy in - Copy out - like call by value on procedure entry, but in addition the value in the local procedure variable is copied back outside to the original variable.
4) Call by name - the parameter is calculated every time it needs to be used. It is possible that its value is calculated many times on each procedure or function call, or it is possible that its value is never calculated.

Call by reference is economical on memory usage but it is harder to exploit parallelism as there is only one copy of the data. Call by value, and copy in / copy out use more memory but they enable greater parallelism. Call by name is the least efficient of the methods but it enables the substitution of a procedure body in place of the call without changing the meaning of the program, and it is easier to reason about programs with this calling method.

Exercises:

1) Which of these calling methods are available in ADA, PASCAL and how are they used?

2) Find an example where "copy in - copy out" gives different results to "call by reference".

3) Suppose that the vector A(1), A(2),...A(10) contains the values 1^2, 2^2,...10^2, and suppose that the value of I is 4.
Suppose S is defined by:-

```
PROCEDURE S(X,Y);
BEGIN
  Y:= X;
  X:= Y;
END S;
```

What will happen when we execute S(A(I),I) with these four methods of procedure call?

Storage Allocation

Three methods for allocating storage are described: "static storage allocation", "automatic storage allocation", and "controlled or dynamic storage allocation".

1) Static Storage Allocation: When the sizes of all the data and variables are known at compile time and providing that there are no pointer variables, no recursion and no re-entrant (parallel) execution of the same procedure or function, then the absolute addresses of all data and variables can be calculated and fixed at compile time.

Question: Can static storage allocation always be used for global data?

2) Automatic Storage Allocation: When the programming language allows the definitions of variables which are local to a function, a procedure, or a block, storage can be allocated automatically to these variables on a stack on entry to the function, procedure, or block, and be freed on exit from the function, procedure or block. As functions, procedures or blocks may be

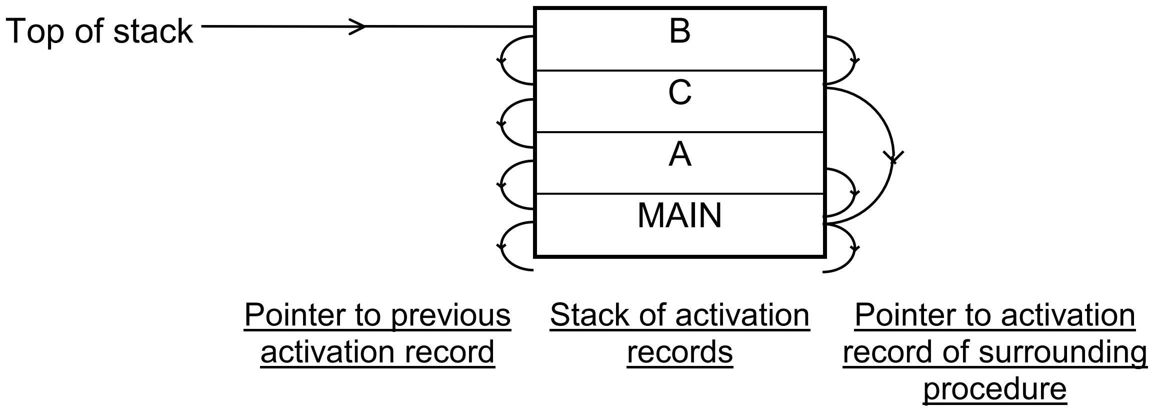
nested, a mechanism needs to be provided for an inner procedure to access the variables of an outer procedure, and similarly for functions and blocks. Let us now discribe by means of an example, <u>two methods</u> of handling the allocation of storage of local variables of procedures. (Functions and blocks are similarly handled).
Here is the example which will be used to illustrate <u>both methods</u>.
Suppose that the procedure MAIN calls procedure A whcih calls procedure C which calls procedure B where the nesting of procedure definitions is as follows:

```
PROCEDURE MAIN IS                    -- level 0
        PROCEDURE C IS                       -- level 1
                PROCEDURE B IS                       -- level 2
                BEGIN ... END B;
        BEGIN ... B; ... END C;
        PROCEDURE A IS                       -- level 1
        BEGIN ... C; ... END A;
BEGIN ... A; ... END MAIN;
```

<u>Method A: Activation records with pointers:</u> When a procedure is entered an activation record is created on the top of the stack for all the storage needs of the procedure and pointers to enable the procedure to access the activation record of the surrounding procedure as well as a pointer to the previous activation record in the stack which is used to reset the stack on procedure exit. The return address is also stored as part of the activation address. So the activation record for <u>every</u> procedure activated will have a sturcture such as:

Return Address (old PC)
Pointer to previous activation record (old top of stack)
Pointer to activation record of surrounding procedure
Local variables
Parameters

The stack of activation records for the example described earlier would look something like this.


Top of stack
B
C
A
MAIN
Pointer to previous activation record
Stack of activation records
Pointer to activation record of surrounding procedure

So when B executes it can access the varables of surrounding procedures using the pointer chain starting from the right of B in the above diagram that is B can access the variables of C and MAIN but not of A. When B exits, the top of stack is reset from the pointer on the left of B.

Exercise: What are the differences in the activation records of blocks, procedures, functions and lambda expression.

Method B: Display vector, activation record with previous execution status.
The display vector (DV) is a vector of pointers to activation records of the current and surrounding procedures.

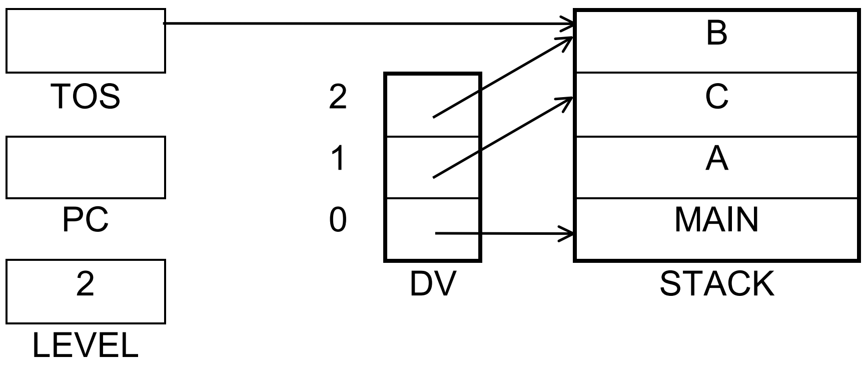
The execution status consists of the following elements:-

display vector (DV)
level of the procedure being executed (LEVEL)
top of stack pointer (TOS)
program counter (PC)

The activation record for every procedure activated would now have structure something like:-

Previous execution status
Local Variables
Parameters

The stack of activation records (STACK) and current execution status for our previous example would look like:

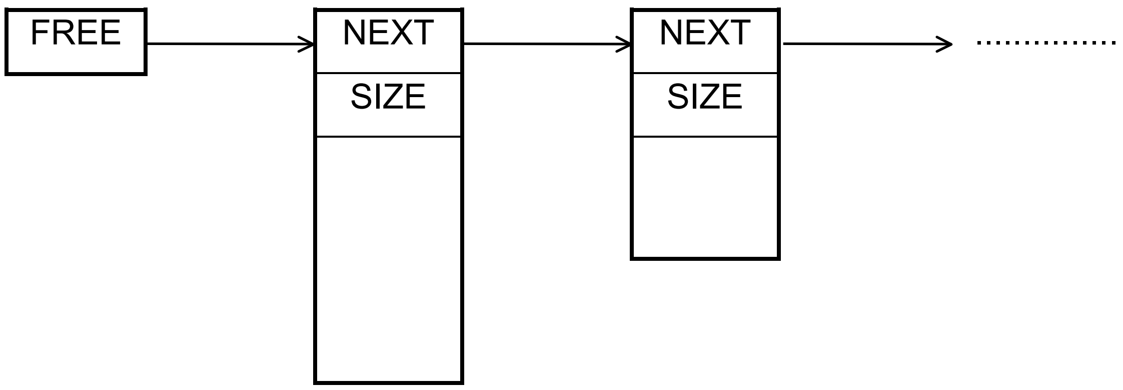


So just before entering B, the old execution status is stored in the stack as part of the activation record of B. When B exists, the old execution status is restored, and execution of the previous procedure C continues with its execution status.

Class discussion: Do we need to store the entire previous DV in the activation record or can we manage with staring just one element (which one) of the previous DV. What are the advantages and disadvantages? Is special care needed when passing a procedure as a parameter?

Exercise: How can a display vector be of benefit with the first method using activation records with pointers?

3) Controlled or dynamic storage allocation: Storage can be allocated dynamicallly under programmer control for example using the NEW statement in PASCAL or ADA. Storage can be freed by the programmer using for example the DISPOSE statement in PASCAL, but in ADA the storage is freed automatically by a "garbage collection routine" when the system detects that the storage has become inaccessible to the program. The simplest way of handling this form of storage allocation is to use a linked list of free blocks where each block will contain a size field and a pointer field to the next free block, such as



When storage needs to be allocated a search can be made for a free block large enough to hold the data and the above list is updated appropriately e.g. a block size may be reduced or a block may be deleted completely. Similarly when storage is freed, it may be added as a new bock to the free list, or if it is adjacent to an existing block, it can be included into the adjacent block by updating its size and perhaps pointers. Sometimes though there is enough memory to there may not be a large enough block to be found e.e. 1000 bytes is needed but there are three non adjacent free blocks of 500 bytes. When this occurs the allocation may fail or a "garbage compaction routine" can be used to move data and update program pointers as appropriate producing a single free block of 1500 bytes, and then the storage of 1000 bytes can be allocated. However, garbage compaction is a slow process.

Notes and questions:

1) When searching for a free block, three possibilities exist

FIRST FIT - allocate from first block found which is large enough to hold the data.

BEST FIT - allocate from block which is large enough and closest in size to the size of the data.

WORST FIT - allocate from the largest free block providingng it is large enough for the data.

2) The free list may be sorted by block size or address of the free block.

3) The free blocks may be organized in the form of an ordered tree or heap.

What are the relative advantages and disadvatages of the above?

11. SYNTAX ANALYSIS METHODS - A GENERAL DESCRIPTION

We shall describe in general terms the bottom up (shift reduce) syntax analysis method and the top down (predictive) syntax analysis method. With both methods the input is scanned from left to right. With both methods a table and stack can be used to guide the syntax analysis. (Another name for "Syntax analysis" is "parsing".)

The bottom up method is based on reducing the right hand side of a rule of the grammar (found in the stack) to the left hand side, and finds a rightmost derivation, which is produced backwards.

The top down method replacing the left hand of a grammar rule (found in the stack) by the right hand side and finds a leftmost derivation, which is produced forwards. We shall also discuss the recursive descent method which is in fact a kind of top down method.

With both methods all inputs accepted by the syntax analysis method are in the language of the grammar. However, the converse may not be true as the use of the tables may restrict the use of the rules of the grammar. Therefore it is highly desirable to use systematic methods to build the tables for guiding the syntax analysis so as to avoid this problem.

Conventions

From now on we only deal with context free grammars which have only a single starting non terminal. When giving examples of context free grammars we often write only the rules. When we do this the reader must understand that the non terminals are on the left side of the arrow, the non terminals on the left hand side of the first rule is the only starting non terminal. All other symbols apart from the arrow and ε are terminals. So when we write that the grammar is:-

$E \rightarrow E+T$	$E \rightarrow T$
$T \rightarrow T*F$	$T \rightarrow F$
$F \rightarrow (E)$	$F \rightarrow id$

it is understood that the set of non terminals is { E,T,F }, the starting set of non terminals is { E } and the set of terminals is { (,), +, *, id }.

Example Grammar Used in Descriptions

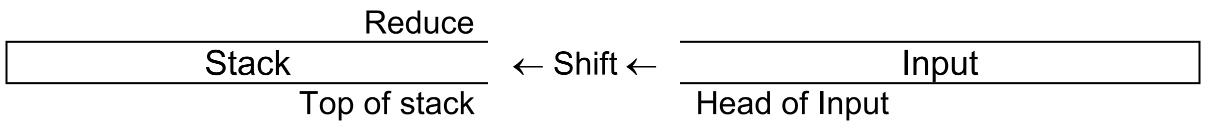
Let us now describe in general terms the bottom up (shift reduce) method and the top down (predictive) method. Let us use the following simple grammar which is suitable to illustrate both these methods.

1) $E \rightarrow id$
2) $E \rightarrow (E+E)$

Let us analyze the input (id+(id+id)) using both these methods.

Bottom up (shift reduce) syntax analysis

Initially the stack is empty and terminals are shifted from the input to the stack. When the right hand side of a rule is found on the top of the stack, it may be reduced to the left hand side, and then the process continues. If at the end, the input is empty and the stack contains only the starting non terminal then the input is accepted. Otherwise it is rejected.



Here is an example syntax analysis (parse).

Stack	Input	Comments
ε	(id+(id+id))	shift, shift
(id	+(id+id))	reduce 1
(E	+(id+id))	shift, shift, shift
(E+(id	+id))	reduce 1
(E+(E	+id))	shift, shift
(E+(E+id	))	reduce 1
(E+(E+E	))	shift
(E+(E+E)	)	reduce 2
(E+E	)	shift
(E+E)	ε	reduce 2
E	ε	accept

The bottom up analysis creates a rightmost derivation but backwards, in this case: $\underline{E} \rightarrow (E+\underline{E}) \rightarrow (E+(E+\underline{E})) \rightarrow (E+(\underline{E}+id)) \rightarrow (\underline{E}+(id+id)) \rightarrow (id+(id+id))$
(The substituted non terminal is underlined)

The syntax analysis can be guided by means of a table such as

Top of stack (non terminals, terminals or ε)	Head of input (terminals or ε) id	(	)	+	ε
id			r1	r1	r1
(	s	s			
+	s	s			
)			r2	r2	r2
E			s	s	Final test
ε	s	s			

Key: s - shift
r n - reduce using rule n.
empty space - error
Final test - if the stack consists only of the start non terminal E then accept the input, otherwise error.
(Of course the input is empty when this test is performed.)

This table was produced using the following observations and common sense.

1) A legal expression can only start with "id" or "(".
2) An "id" can only be followed by a "+" or ")" or ε (end of input).
3) A "+" may only be followed by a "id" or "(".
4) A "(" can only be followed by a "id" or "("
5) A ")" can only be followed by a "+" or ")" or ε (end of input).
6) An "E" can only be followed by a "+" or ")" or ε (end of input).

Systematic methods for building such tables for bottom up syntax analysis can be used for certain kinds of grammars (see later).

Notes:
1) The top of the stack is empty only when the stack itself is empty and similarly for the head of the input.
2) With other methods, states are pushed onto the stack and over here grammar symbols are pushed onto the stack.

Top down (predictive) syntax analysis

Initially the stack contains only the starting non terminal. When there is a non terminal on the top of the stack, then based on the head of the input it is replaced by a suitable right hand side of a rule. When the top of stack and head of input are the same terminal they are both erased and when they are different terminals an error is raised. If at the end both stack and input are empty then the input is accepted. Otherwise it is rejected.

Stack	Input
Top of stack	Head of Input

Replace non terminal according to head of input
Erase when same terminal. Raise error when different terminals.

Here is an example syntax analysis (parse).

Stack	Input	Comments
E	(id+(id+id))	replace 2
(E+E)	(id+(id+id))	erase
E+E)	id+(id+id))	replace 1
id+E)	id+(id+id))	erase, erase
E)	(id+id))	replace 2
(E+E))	(id+id))	erase
E+E))	id+id))	replace 1
id+E))	id+id))	erase, erase
E))	id))	replace 1
id))	id))	erase, erase, erase
ε	ε	accept

The top down analysis creates a leftmost derivation forwards, in this case

$\underline{E}\rightarrow(\underline{E}+E)\rightarrow(id+\underline{E})\rightarrow(id+(\underline{E}+E))\rightarrow(id+(id+\underline{E}))\rightarrow(id+(id+id))$

The following table for guiding the syntax analysis was built using common sense that if E is on the top of stack then rule 1 is appropriate when the head of input is "id" and rule 2 is appropriate when the head of input is "(". For certain kinds of grammars, such tables for top down syntax analysis can be built systematically from the grammar (see later).

	Head of input (terminals or ε)				
	id	(	)	+	ε
Top of stack (non terminal)					
E	id	(E+E)			

The entries the table show which right hand side of a rule should replace the non terminal. As before a blank means an error.

Note: There is no need to write into the table when to erase as this can be written into the algorithm itself, (check if top of stack = head of input).

Recursive descent syntax analysis

The following recursive procedure, which can systematically be constructed from the grammar and the table for top down syntax analysis, will analyze the input.

```
PROCEDURE      E;
-- SPECIFICATION:
-- Check for and skip over text derivable from E.

BEGIN
        IF              current_token = 'id'
        THEN            get_next_token;
        ELSIF           current_token = '('
        THEN            get_next_token;
                        E;                      -- A
                        IF current_token = '+'
                        THEN get_next_token
                        ELSE  error
                        ENDIF;
                        E;                      -- B
                        IF current_token = ')'
                        THEN get_next_token;
                        ELSE error
                        ENDIF;
        ELSE error
        ENDIF;
END E;
```

Exercises:

1) Analyze ((id+id)+id+id) using the previous tables using both top down and bottom up methods. (It should be rejected.)

2) Present the states of the stack of procedure calls and the input when executing:- E; -- C where initially the input is (id+(id+id)). Compare and see the similarity to the stack of the top down method.

3) Consider the grammar:

$C \rightarrow$ if T then C else C endif
$C \rightarrow$ s
$T \rightarrow$ id O id
$O \rightarrow$ =
$O \rightarrow$ <
$O \rightarrow$ >

a) Build a table for bottom up syntax analysis for this grammar.
b) Analyze the following inputs using the table you built.
i) if id = id then s endif
ii) if id < id then s else if id > id then s else s endif endif

4) Consider the grammar:

$C \rightarrow$ if t then C L endif
$C \rightarrow$ s
$L \rightarrow$ elsif t then C L
$L \rightarrow$ else C

a) Build a table for top down syntax analysis for this grammar.
b) Analyze the following inputs using the table you built.
i) if t then s endif
ii) if t then s elsif t then s else s endif

5) Use the grammar and the table you built in the previous question, to write recursive procedures for syntax analysis.

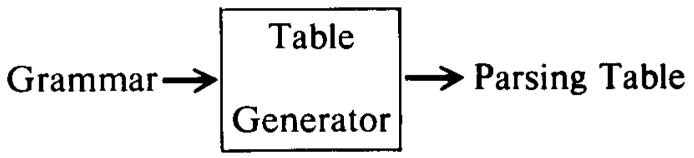
6) How could bottom up and top down syntax analysis be carried out without a stack but with a single string, a pointer to the active position in the string, and string replacement operations? Is this an efficient approach compared to using a stack?

12. BOTTOM UP SYNTAX ANALYSIS

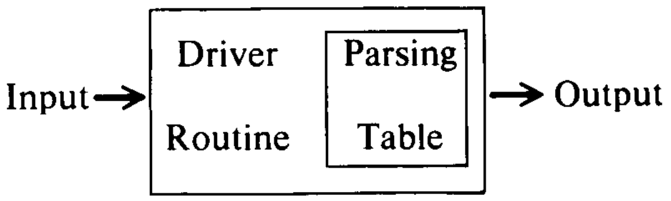
We shall now describe methods of bottom up syntax analysis and systematic methods of building the table to guide the syntax analysis. Many methods are known but we shall describe two of them, namely, LR syntax analysis and operator precedence syntax analysis.

LR Syntax Analysis

From “Principles of Compiler Writing” by Aho and Ullman



(a) Generating the parser.



(b) Operation of the parser.

Fig. 6.1. Generating an LR parser.

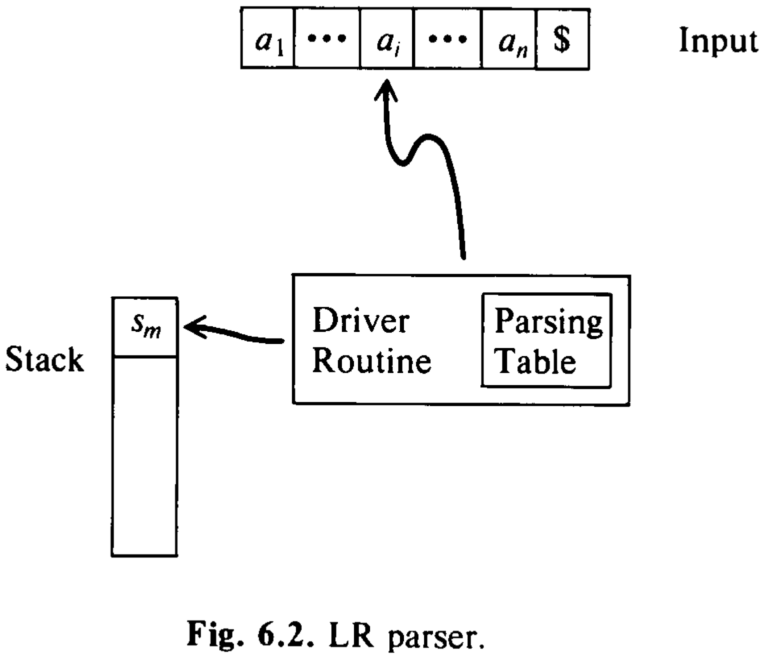


Fig. 6.2. LR parser.

The program driving the LR parser behaves as follows. It determines s_m, the state currently on top of the stack, and a_i, the current input symbol. It then consults ACTION$[s_m,\ a_i]$, the parsing action table entry for state s_m and input a_i. The entry ACTION$[s_m,\ a_i]$ can have one of four values:

1. shift s
2. reduce $A \rightarrow \beta$
3. accept
4. error

The function GOTO takes a state and grammar symbol as arguments and produces a state. It is essentially the transition table of a deterministic finite automaton whose input symbols are the terminals and nonterminals of the grammar.

A *configuration* of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \cdots\ X_m\ s_m,\quad a_i\ a_{i+1}\ \cdots\ a_n\ \$)$$

The next move of the parser is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting the parsing action table entry ACTION$[s_m,\ a_i]$. The configurations resulting after each of the four types of move are as follows:

1. If ACTION$[s_m,\ a_i]$ = shift s, the parser executes a shift move, entering the configuration

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \cdots\ X_m\ s_m\ a_i\ s,\quad a_{i+1}\ \cdots\ a_n\ \$)$$

Here the parser has shifted the current input symbol a_i and the next state $s = \text{GOTO}[s_m, a_i]$ onto the stack; a_{i+1} becomes the new current input symbol.

2. If $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

   $$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_{m-r}\ s_{m-r}\ A\ s,\ \ a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

   where $s = \text{GOTO}[s_{m-r}, A]$ and r is the length of β, the right side of the production. Here the parser first popped $2r$ symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}. The parser then pushed both A, the left side of the production, and s, the entry for $\text{ACTION}[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, $X_{m-r+1} \cdots X_m$, the sequence of grammar symbols popped off the stack, will always match β, the right side of the reducing production.

3. If $\text{ACTION}[s_m, a_i] = \text{accept}$, parsing is completed.

4. If $\text{ACTION}[s_m, a_i] = \text{error}$, the parser has discovered an error and calls an error recovery routine.

The LR parsing algorithm is very simple. Initially the LR parser is in the configuration $(s_0,\ a_1\ a_2\ \ldots\ a_n\ \$)$ where s_0 is a designated initial state and $a_1a_2 \ldots a_n$ is the string to be parsed. Then the parser executes moves until an accept or error action is encountered. All our LR parsers behave in this fashion. The only difference between one LR parser and another is the information in the parsing action and goto fields of the parsing table.

Example 6.1. Figure 6.3 shows the parsing action and goto functions of an LR parser for the grammar

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

The codes for the actions are:

1. s*i* means shift and stack state *i*,
2. r*j* means reduce by production numbered *j*,
3. acc means accept,
4. blank means error.

State	Action						Goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 6.3. Parsing table.

	Stack	Input
(1)	0	**id** * **id** + **id** $
(2)	0 **id** 5	* **id** + **id** $
(3)	0 *F* 3	* **id** + **id** $
(4)	0 *T* 2	* **id** + **id** $
(5)	0 *T* 2 * 7	**id** + **id** $
(6)	0 *T* 2 * 7 **id** 5	+ **id** $
(7)	0 *T* 2 * 7 *F* 10	+ **id** $
(8)	0 *T* 2	+ **id** $
(9)	0 *E* 1	+ **id** $
(10)	0 *E* 1 + 6	**id** $
(11)	0 *E* 1 + 6 **id** 5	$
(12)	0 *E* 1 + 6 *F* 3	$
(13)	0 *E* 1 + 6 *T* 9	$
(14)	0 *E* 1	$

Fig. 6.4. Moves of LR parser on **id** * **id** + **id**.

The parse table is constructed by creating items and then grouping them into sets.

The items are obtained from the original grammar by adding a new nonterminal E' which will be the new starting non terminal and a rule $E'\rightarrow E$ wher E is the old starting non terminal.

From the new and original rules, items are constructed by inserting one dot on the right hand side in all possible positions. for example, the rule $E\rightarrow E+T$ gives rise to four items $E\rightarrow \cdot E+T$, $E\rightarrow E\cdot +T$, $E\rightarrow E+\cdot T$, $E\rightarrow E+T\cdot$ where the dot represents where we have reached in parsing. These items are used to construct a non-deterministic finite automaton. The non-deterministic finite automaton is converted to a deterministic finite automaton by using the set construction. The parsing table is constructed from the deterministic finite automaton which can be minimized. In practice, these steps are carried out by a program which generates parse tables.

Here are the sets of items and deterministic finite automaton for the grammar given in example 6.1.

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E{+}T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T{*}F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot\mathbf{id}$

I_1: $E' \rightarrow E\cdot$
$E \rightarrow E\cdot{+}T$

I_2: $E \rightarrow T\cdot$
$T \rightarrow T\cdot{*}F$

I_3: $T \rightarrow F\cdot$

I_4: $F \rightarrow (\cdot E)$
$E \rightarrow \cdot E{+}T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T{*}F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot\mathbf{id}$

I_5: $F \rightarrow \mathbf{id}\cdot$

I_6: $E \rightarrow E{+}\cdot T$
$T \rightarrow \cdot T{*}F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot\mathbf{id}$

I_7: $T \rightarrow T{*}\cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot\mathbf{id}$

I_8: $F \rightarrow (E\cdot)$
$E \rightarrow E\cdot{+}T$

I_9: $E \rightarrow E{+}T\cdot$
$T \rightarrow T\cdot{*}F$

I_{10}: $T \rightarrow T{*}F\cdot$

I_{11}: $F \rightarrow (E)\cdot$

Fig. 6.7. Collection of sets of items.

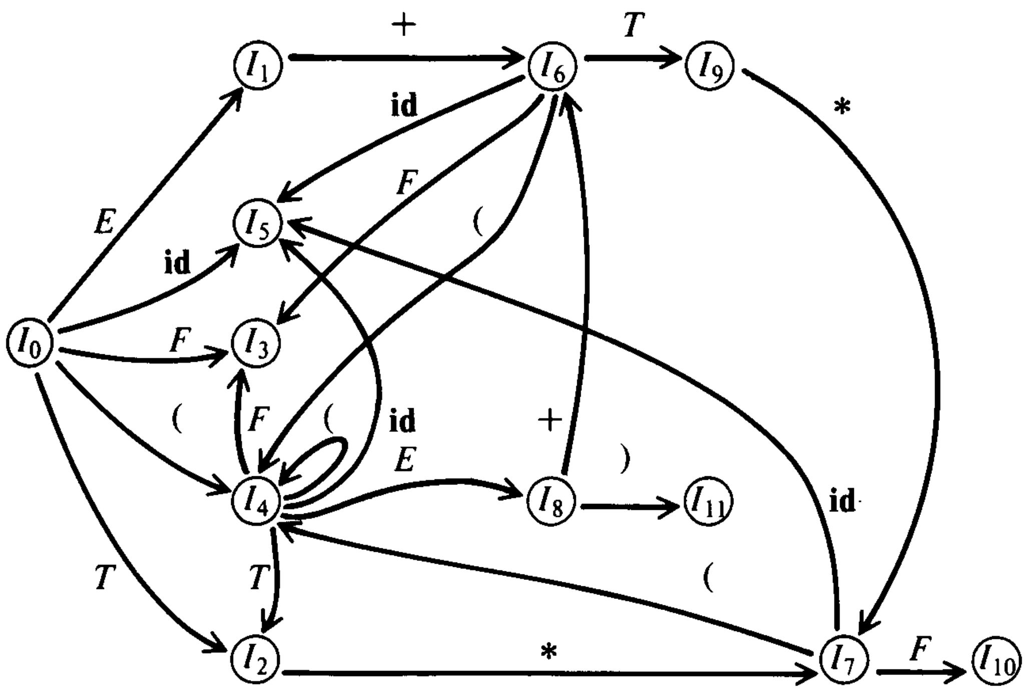


Fig. 6.8. Deterministic finite automaton *D*.

Operator Precedence Syntax Analysis

This method can be used with operator grammars which have the following restrictions regarding the right hand sides of the grammar rules.

1) The right hand side is not ε.
2) The right hand side does not consist of a single non terminal.
3) Two non terminals do not occur adjacent to each other in the right hand side (i.e. they are separated by at least one terminal).

e.g. the rule $E \rightarrow E+E$ can be used in an operator grammar but the rule $E \rightarrow + E\ E$ can't. (Here + is a terminal.)

Relations $\doteq$, $<\bullet$, $\bullet>$ are defined on terminal symbols so as to identify right hand side of the rule to be used in the reduction so as to build a rightmost derivation backwards. (Recall bottom up syntax analysis builds rightmost derivations backwards.)

$<\bullet$	-	identifies left hand end of right hand side.
$\bullet>$	-	identifies right hand end of right hand side.
$\doteq$	-	identifies interior of right hand side.

These relations can be defined either practically using additional knowledge about the terminals, but when the grammar is unambiguous, formal definitions can be given, but they have limitations. Let us discuss both these approaches.

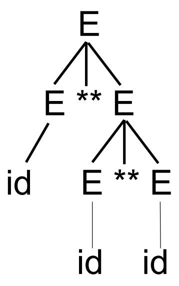
Practical approach to defining operator precedence relations

Let us consider the grammar

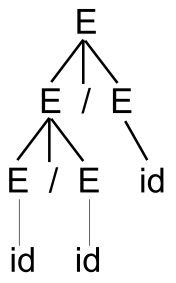
$E \rightarrow E/E$	$E \rightarrow E{**}E$
$E \rightarrow (E)$	$E \rightarrow id$

which of course is an ambiguous grammar. (Why?) It is possible to resolve these ambiguities and carry out syntax analysis uniquely by specifying additional knowledge about the operators /, **, namely, their priorities and whether they associate to the left or to the right.

Let us assume that ** has higher priority than /. Let us assume that ** is right associative i.e. id**id**id should have the following derivation tree.



Also let us assume that / is left associate i.e. id/id/id should have the following derivation tree.



The following table of operator precedence relations can be built by using common sense as explained later.

	Head of input (terminals or ε)					
Topmost terminal in stack or ε	/	**	(	)	id	ε
/	•>	<•	<•	•>	<•	•>
**	•>	<•	<•	•>	<•	•>
(	<•	<•	<•	$\doteq$	<•	
)	•>	•>		•>		•>
id	•>	•>		•>		•>
ε	<•	<•	<•		<•	final test

Note: As before a blank denotes an error.

Algortithm for syntax analysis with an operator precedence table

```
WHILE     the input is not empty
LOOP
IF        (topmost terminal in stack ≐ head of input)    OR
          (topmost terminal in stack <• head of input)
THEN      shift one terminal from input to stack
ELSIF     (topmost terminal of stack •> head of input)
THEN      reduce right hand side of rule on top of stack delimited by
          <•...........•> to the left hand side and if not possible then error.
ELSE      error
ENDIF;
ENDLOOP;
```

```
-- FINAL TEST (input is now empty)
IF          the stack consists only of the start non terminal
THEN        accept
ELSE        error
ENDIF
```

Notes:

1) If there are no terminals in the stack then topmost terminal means ε.

2) If the input is empty then head of input means ε.

3) In using such a table there is a danger we may not accept the full language defined by the grammar but in practice, this does not occur for arithmetic expressions

Here is an example syntax analysis of id/id**id using this table. For ease of understanding we enter the relations $\doteq$, <•, •> in the stack, but this is not essential.

Stack (top on right)	Input	Comments
ε <•	id/id**id	shift
ε<•id•>	/id**id	reduce
ε<•E	/id**id	shift
ε<•E/<•	id**id	shift
ε<•E/<•id•>	**id	reduce
ε<•E/<•E	**id	shift
ε<•E/<•E**<•	id	shift
ε<•E/<•E**<•id•>	ε	reduce
ε<•E/<•E**E•>	ε	reduce
ε<•E/E•>	ε	reduce
ε E	ε	accept

<u>Exercise</u>: Draw the derivation tree corresponding to the above analysis.

Determining operator precedence relations using common sense

Here are rules for determining operator precedence relations. However, there may be contradictions between the rules and that is when common sense is needed.

Let t_1 and t_2 be <u>any</u> terminals.

1) When t_1 and t_2 have the same priority and they both associate to the left then make t_1 •> t_2 and t_2 •> t_1 (i.e. the left symbol is stronger). Also note that this includes the possibility that t_1 , t_2 are the same terminal.

2) When t_1 and t_2 have the same priority and they both associate to the right then define t_1 <• t_2 and t_2 <• t_1 (i.e. the right symbol is stronger). Also note that this includes the possibility that t_1 , t_2 are the same terminal.

3) When t_1 and t_2 have the same priority but one associates to the left and the other to the right then put an error entry (blank entry) in the table.

4) If t_1 is of higher priority that t_2 then make t_1 •> t_2 and t_2 <• t_1 (Similarly if t_1 is of lower priority than t_2 make t_1 <• t_2 and t_2 •> t_1)

5) When t_1, t_2 occur on the right hand side of a rule with t_2 following t_1 directly or separated by at most one non terminal then make $t_1 \doteq t_2$.

6) If for all derivations from the starting non terminal Z, the terminals t_2 never directly follows t_1 (or separated by one non terminal) then make an error entry (blank entry) in the table.

7) For terminals t occuring in rules in the form $A \rightarrow$....t or $A \rightarrow$....tB where B is a non terminal, make t•> any terminal.

8) For terminals t occuring in rules of the form $A \rightarrow$ t.... or $A \rightarrow$Bt.... where B is a non terminal, make any terminal <•t.

9) For any terminal t, make ε<•t if t can occur at the left hand end or one non terminal away from the left hand end in a derivation from the starting non terminal.

10) For any terminal make t•>ε if t can occur at the right hand end or one non terminal away from the right hand end in a derivation from its start non terminal.

11) For ε, with ε, put "final test" in the table.

Formally defining operator precedence relations

When the operator grammar is unambiguous, the following definitions can be used for determining the operator precedence relations. However, there may exist a, b such that a, b are terminals or ε and more than one of the relations $a \doteq b$, a<•b, a•>b hold. In such a case a parse table can not be built without making suitable changes to the grammar.

Let t_1, t_2 be any terminal symbols.

1) $t_1 \doteq t_2$ means as before that t_1 occurs on the right hand side of a rule with t_2 directly folowing t_1 or separated by one non terminal.

2) t_1 <• t_2 means we can have some right hand side of a rule of the form ...t_1A... and we can derive from the non terminal $A \overset{+}{\Rightarrow} \gamma t_2 \delta$ where γ is either ε or a single non terminal and δ is any string.

3) t_1 •> t_2 means we have for some right hand side of a rule of the form ...At_2... and we can derive from the non terminal $A \overset{+}{\Rightarrow} \gamma t_1 \delta$ where δ is ε or one non terminal and γ is any string.

4) For any terminal t, when $Z \overset{+}{\Rightarrow} \gamma t \delta$ where Z is the start non terminal, γ is ε or a single non terminal, and δ is any string make, then ε <• t. Similarly make t •> ε when δ is ε or a single non terminal and γ is any string.

However for bottom up deterministic parsing, in addition to unambiguity, it is required that all the right hand side of the rules are different. (This is similar to what was required for backward deterministic regular grammars.)

<u>Example</u>: Here again is the ambiguous grammar for which we built a table of operator precedence relations.

$E \rightarrow E/E$	$E \rightarrow E**E$
$E \rightarrow (E)$	$E \rightarrow id$

Here is an equivalent unambiguous grammar in which the operator / is left associative, the operator ** is right associative, and the operator ** has higher priority than /.

$E \rightarrow E/P$	$E \rightarrow P$
$P \rightarrow F**P$	$P \rightarrow F$
$F \rightarrow (E)$	$F \rightarrow id$

<u>Class discussion</u>: What principles were used in constructing this unambiguous grammar?

Unfortunately the above is not an operator grammar. Rules of the form $A \rightarrow B$ can be removed and an operator grammar obtained. However all the right hand sides of the new rules will not be different. So there will remain difficulties to use the new rules for bottom up deterministic parsing.

Exercises:

1) For the ambiguous grammar above, analyze the input “id ** id ** id” with the help of the table. Is there only one possibility for analysis when the table is used?

2) Build a table of precedence relations for the unambigous grammar above based on the formal definition of operator precedence relations.

3) The following grammar is ambiguous:

C $\rightarrow$ if t then C
C $\rightarrow$ if t then C else C
C $\rightarrow$ s

a) Draw two derivation trees for: "if t then if t then s else s".

b) Build a table of operator precedence relations so as to ensure that the "else" is associated with the closest "if" before the "else".

c) Analyze "if t then if t then s else s" with the help of the table you built.

d) Construct on unambiguous grammar equivalent to the above grammar.

4) How can a transitive closure algorithm (e.g. Warshall's) be used to determine <• and •> from the formal definitions of these relations?

13. TOP DOWN SYNTAX ANALYSIS

We shall present two methods of top down syntax analysis. With the first method, predictive syntax analysis a stack and table are used. With the second method, recursive descent syntax analysis there seems to be no stack and no table and instead (recursive) procedures are used. However, what has hapened is that the information in the table has been incorporated into the procedures and the stack needed to handle procedure calls, corresponds to the explicit stack used in the first method.

For us to use either of these metods the grammar must satisfy the following requirements.

1) There is no left recursion in the grammar i.e there is no non terminal A such that $A \overset{+}{\Rightarrow} A\alpha$. (This prevents the stack expanding indefinitely.)

2) For any non terminal, all its right hand sides start differently. This means that we never have different rules $A \rightarrow \alpha\beta$ and $A \rightarrow \alpha\gamma$ where $\alpha \neq \varepsilon$. The purpose of this requirement is to make it easy to identify which right hand side to use based on the head of the input.

It can be quite easy to correct the grammar so it satisfies both these requirements. First left recursion is eliminated and then left factoring is used. (Additional steps may be needed.) A simplified explanation follows.

Left Recursion Removal

If the first requirement is not met and we have left recursion in the grammar, for example $A \rightarrow A\alpha$ $A \rightarrow \beta$, then these rules can be replaced by

$A \rightarrow \beta A'$ $A' \rightarrow \alpha A'$ $A' \rightarrow \varepsilon$ where A' is a new non terminal.

In this way, left recursion has been replaced by right recursion.

For example the left recursive rules $E \rightarrow E + T$ $E \rightarrow T$ would be replaced by $E \rightarrow T\ E'$ $E' \rightarrow \varepsilon$ $E' \rightarrow +T\ E'$.

Left Factoring

If the second requirement is not met and we have rules $A \rightarrow \alpha\beta$ $A \rightarrow \alpha\gamma$, then these rules can be replaced by

$A \rightarrow \alpha A'$ $A' \rightarrow \beta$ $A' \rightarrow \gamma$ where A' is a new non terminal.

Here α is the common initial part of the two right hand sides, $\alpha \neq \varepsilon$, and this ensures that β, γ start with different symbols.

For example the rules

C → if t then C endif
C → if t then C else C endif

would be replaced by

C → if t then C C'
C' → endif
C' → else C endif

Predictive syntax analysis

Let us illustrate this by an example and then we will explain how the table is built. Consider the grammar

E → T E'	E' → +T E'	E' →ε
T → F T'	T' → *F T'	T'→ε
F → id	F → (E)	

(Incidentally this grammar is similar to the grammar in the section "LR Syntax Analysis", but left recursion has been removed).

Here is the table for syntax analysis where the entries show which right hand side is to be used for the non terminal when it appears on the top of stack. As before, a blank entry signifies an error.

	Head of input (terminals or ε)					
	id	+	*	(	)	ε
Top of stack (non terminals)						
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	id			(E)		

Here is the algorithm for predictive syntax analysis.

```
Initialize stack to contain starting non terminal only
WHILE   stack is not empty
LOOP
    WHILE top of stack is a non terminal
    LOOP
    examine head of input and use syntax analysis table to replace the
    non terminal on the top of the stack by the right hand side of the rule
    in the table or if there is a blank in the table raise an error.
    ENDLOOP;
    IF top of stack ≠ head of input
    THEN error
    END IF;
    WHILE (stack ≠ ε) AND (top of stack = head of input)
    LOOP
    erase one terminal from stack and input
    END LOOP;
ENDLOOP;
IF input and stack are empty THEN accept ELSE error ENDIF;
```

<u>Example</u>: Analysis of id+(id * id)

Stack (top on left)	Input	Remarks
E	id +(id*id)	replace
TE'	id +(id*id)	replace
FT'E'	id +(id*id)	replace
id T'E'	id +(id*id)	erase
T' E'	+ (id*id)	replace
E'	+ (id*id)	replace
+TE'	+ (id*id)	erase
TE'	(id*id)	replace
FT'E'	(id*id)	replace
(E)T'E'	(id*id)	erase
E) T'E'	id*id)	replace
TE') T'E'	id*id)	replace
FT'E') T'E'	id*id)	replace
id T'E') T'E'	id*id)	erase
T'E') T' E'	*id)	replace
*FT'E') T'E'	*id)	erase
FT'E') T'E'	id)	replace
id T'E') T'E'	id)	erase
T'E') T'E'	)	replace
E') T'E'	)	replace
)T'E'	)	erase
T'E'	ε	replace
E'	ε	replace
ε	ε	accept

Note: When a non terminal is replaced by ε it is erased.

Class discussion: What is the significance of the word “predictive” in the name of this method ?

Exercise: Present a similar analysis of id * id + id.

Recursive descent syntax analysis

We can convert the information in the table used for predictive syntax analysis into a set of (indirectly recursive) procedures. Each non terminal becomes a procedure name and its procedure is responsible for checking if something derivable from the non terminal is to be found at the front of the input and skip over what was found, and raise an error if it is not possible. In explaining this method we shall assume that "next" is a procedure for moving forward one terminal in the input.

In addition to a procedure for each non terminal we need a main procedure which calls the procedure for the starting non terminal and then checks if the input is empty. Here are some procedures.

```
PROCEDURE  MAIN IS
BEGIN
E ;     - - a
IF      input is empty
THEN accept
ELSE  error
ENDIF;
END MAIN ;

PROCEDURE  E IS
BEGIN
IF      head of input = 'id'      OR
        head of input = '('
THEN T ;   - - b
         E' ;   - - c
ELSE error
ENDIF;
END E ;
PROCEDURE E' is
BEGIN
IF      head of input = '+'
THEN next ;
        T ;     - - d
        E' ;    - - e
ELSIF           head of input = ')' OR
                head of input = ε
THEN  - - do nothing
ELSE error
ENDIF;
END E' ;
.      .
.      .
PROCEDURE F IS
IF      head of input = 'id'
THEN next
ELSIF head of input = '('
THEN next ;
        E ;     - -  f
        IF      head of input = ')'
        THEN  next
        ELSE error
        ENDIF;
ELSE error
ENDIF;
END  F;
```

<u>Class discussion:</u> What is the connecton between the previous procedures and the table used for syntax analysis. How is replacement by ε handled in the procedures?

<u>Exercises</u>:

1) Write similar procedures for T and T'.

2) Analyze the input "(id)" using the recursive procedures presenting the states of the stack of procedure calls and of the input.

3) Analyze the same input "(id)" using predictive syntax analysis and compare it to the analysis you wrote in answer to the previous qustion and see the similarities.

4) Apply the methods of left recursion removal and of left factoring to the following grammar to make it suitable for top down syntax analysis.

$C \rightarrow$ s
$C \rightarrow$ if t then C endif
$C \rightarrow$ if t then C else C endif
$C \rightarrow$ if t then C elsif t then L else C endif
$L \rightarrow C$
$L \rightarrow$ L elsif t then C

5) Consider the following unambiguous grammar for expressions in prefix notation.

$E \rightarrow +EE$	$E \rightarrow -EE$
$E \rightarrow *EE$	$E \rightarrow /EE$
$E \rightarrow$ id	

a) Write a "recursive descent" procedure for syntax analysis.

b) For the input "/ * id id - id id" present the states of the stack and input when the procedure you wrote is executed.

c) Present the syntax analysis of the above input using top down and bottom up syntax analysis methods.

d) How would you write the expression "/ * id id - id id" in normal arithmetic notation (that is infix operators).

e) Write a procedure similar to the procedure you wrote in your answer to a) translate an expression in prefix notation to fully backeted infix notation.

f) Is the above grammar an operator grammar?

6) a) Write a grammar defining lists of digits. (By lists we also mean nested lists or an empty list).

For example (0 1 (3 4 ( ) 5 6) 1 9)

b) Write "recursive descent" procedures for syntax analysis according to the grammar you wrote in your answer to a). (Be careful to make sure that your grammar is suitable for top down syntax analysis.)

Defining FIRST and FOLLOW sets

These sets will be used for building syntax analysis tables for the top down method. Let u be a string and A be a non terminal.

FIRST (u) = { t: (t is a terminal and u $\Rightarrow^{*}$ tα) or (t=ε and u $\Rightarrow^{*}$ ε)}

FOLLOW (A) = { t: (t is a terminal and Z $\Rightarrow^{*}$ αAtβ) or
(t = ε and Z$\Rightarrow^{*}$ αA)
where Z is the starting non terminal }

Class discussion: How can common sense be used to construct these sets.

Constructing Syntax Analysis Tables from FIRST and FOLLOW sets

Once FIRST and FOLLOW are constructed.
the table is constructed using the following algorithm:

```
FOR   every rule $A \rightarrow \alpha$
LOOP -- update the row of non terminal A
        (1)    For each terminal in FIRST ($\alpha$)
               put $\alpha$ as the action in row A for this terminal.
        (2)    If $\varepsilon$ is in FIRST ($\alpha$)    (that is $\alpha \Rightarrow^{*} \varepsilon$)
               then add $\alpha$ as the action in row A
               for every element of FOLLOW(A).
END LOOP
```

If an entry has two right hand sides then during the analysis both need to be tried. So it is necessary to modify the algorithm we gave and use additional backtracking steps. However it is preferable that the grammar should be modified to avoid this.

Exercise: Consider the grammar

C $\rightarrow$ if t then C L endif
C $\rightarrow$ s
L $\rightarrow$ ε
L $\rightarrow$ else C
L $\rightarrow$ elsif t then C L

a) Use common sense to determine the first and follow sets and build the syntax analysis table for predictive syntax analysis.

b) From the table, build procedures for syntax analysis according to the recursive descent method.

Constructing FIRST and FOLLOW sets

The FIRST sets for terminals, non terminals, and ε may be constructed thus:-

```
Initialization:-
(1) For every terminal X define FIRST (X) = {X}. Similarly FIRST (ε) = {ε}.
(2) For every non terminal X define FIRST (X) = { }.
(3) For every rule X→ε add ε to FIRST (X).
(4) For every rule X→ tα such that t is a terminal, add t to FIRST (X).

Iteration:-
loop
for every rule X → Y1 Y2....Yk    (Here each Yi is a terminal or non terminal.)
    loop
    (1) if ε is in FIRST (Yi) for each i = 1,2...,k
        then add ε to FIRST (X)
        { Here we have X → Y1Y2 ...Yk ⇒* ε }
    (2) Add every terminal symbol in FIRST (Y1) to FIRST (X).
    (3) if ε is in FIRST (Y1), FIRST (Y2)....FIRST (Yj-1) for any j
         then add all terminal symbols of FIRST (Yj) to FIRST (X).
         end if;
    end loop
exit when nothing new is added to any FIRST set.
end loop;
```

Once we have the FIRST sets defined for terminals, non terminals, and ε we can define then for strings as follows:

```
FIRST (Y1 Y2...Yk) = FIRST (Y1) - {ε}
j = 1
while ε is in FIRST (Yj) and j < k
loop
        j := j + 1;
        add all elements of FIRST (Yj) - {ε} to FIRST (Y1, Y2....Yk).
end loop;
if j = k and ε is in FIRST (Yk) then add ε to FIRST (Y1, Y2....Yk) end if;
```

Construction of FOLLOW sets for Non terminals.
We assume that there are no useless non terminals.

(1) Initialization: Set FOLLOW (Z) = $\{\varepsilon\}$ where Z is the starting non terminal but FOLLOW (A) is empty for other non terminals A.

(2) For every rule $A \rightarrow \alpha B \beta$, every terminal in FIRST (β) is added to FOLLOW (B).

(3) If there is a rule of the form $A \rightarrow \alpha B \beta$ where ε is in FIRST (β),
that is $\beta \Rightarrow^{*} \varepsilon$, then everthing in FOLLOW (A) is added to FOLLOW (B).
Note this means that $A \Rightarrow^{*} \alpha B$.

(The special case $\beta = \varepsilon$ i.e. the rule is $A \rightarrow \alpha B$, is included in the above.)

<u>Step 3</u> is repeated until nothing new is added to any FOLLOW set.

<u>Review:</u> To build these tables for top down syntax analysis, we proceed as follows.

1) Determine FIRST (α) for all rules $A \rightarrow \alpha$.
2) Determine FOLLOW (A) for all rules $A \rightarrow \alpha$ such that ε is in FIRST (α).
3) Build the table from the first and follow sets.

14. TRANSLATION TO INTERMEDIATE CODE

We shall illustrate the recursive descent method of translating various kinds of statements. This is based on the recursive decent syntax analysis method. We shall define "Simple Structured Programming Language" (SSPL) and "Machine Language for Simple Structured Programming" (MLSSP). We shall then give procedures for recursive descent syntax analysis for SSPL, as this will give us the structure for the translation procedures. Then we will give the actual procedures for the translation form SSPL to MLSSP, where SSPL can be viewed as the "high level language" and MLSSP can be viewed as the intermediate code. Of course MLSSP can either be interpreted or translated to machine code but we shall not give these details.

Simple Structured Programming Language (SSPL)

Here is the grammar defining this language

```
1) program → declarations block
2) declarations → INTEGER namelist;
3) namelist → name [, name] *
4) name → [A | B... | Z] [A | B.. | Z | 0 | 1 | ... | 9]*
5) statement_list → statement [statement]*
6) statement →  comment | null | assignment |
                conditional | block | exit_block | repeat_block |
                input  | output
7) comment → NOTE <any text except “;”> ;
8) null → ;
9) assignment → name := expression;
10) conditional → IF boolean _ expn
                  THEN statement _list
                  ENDIF;
11) conditional → IF boolean _ expn
                  THEN statement_list
                  ELSE statement_list
                  ENDIF;
12) block → BEGIN statement_list END;
13) exit_block → EXIT;
14) repeat_block → REPEAT;
15) input → READ namelist;
16) output →  WRITE namelist;
17) expression → name | number | (expression op expression)
18) number → [ε | + | -] [0 | 1... | 9] [0 | 1.... | 9]*
19) op → + | - | * |  /
20) boolean_expn → expression comp expression
21) comp → < | > | =
```

Exercises:

1) What are the non terminals, start non terminal and terminals of this grammar.

2) Is X*Y-7 a legal expression according to this grammar and if not correct it to conform to the rules of the grammar.

3) May an "IF....ENDIF occur within a BEGIN....END and vice versa?

4) Draw a derivation tree for the program:-
INTEGER I; BEGIN READ I; WRITE I; END;

Recursive Descent Syntax analysis of SSPL

We shall assume that CS is a global variable giving the current token (or terminal) of the input, that "next" is a procedure to move onto the next token of the input (error if there is none), and that "check" is a procedure with the following definition.

```
procedure check (t : terminal);
begin
if CS = t
then next;
else Syntax error;
endif;
end check;
```

We give procedures for each non terminal which are responsible for checking if something derivable from that non terminal is to be found at the start of the input onwards, and skipping over it, or, raising an error if nothing derivable was found. An additional procedure called "main" is responsible for starting and terminating the syntax analysis.

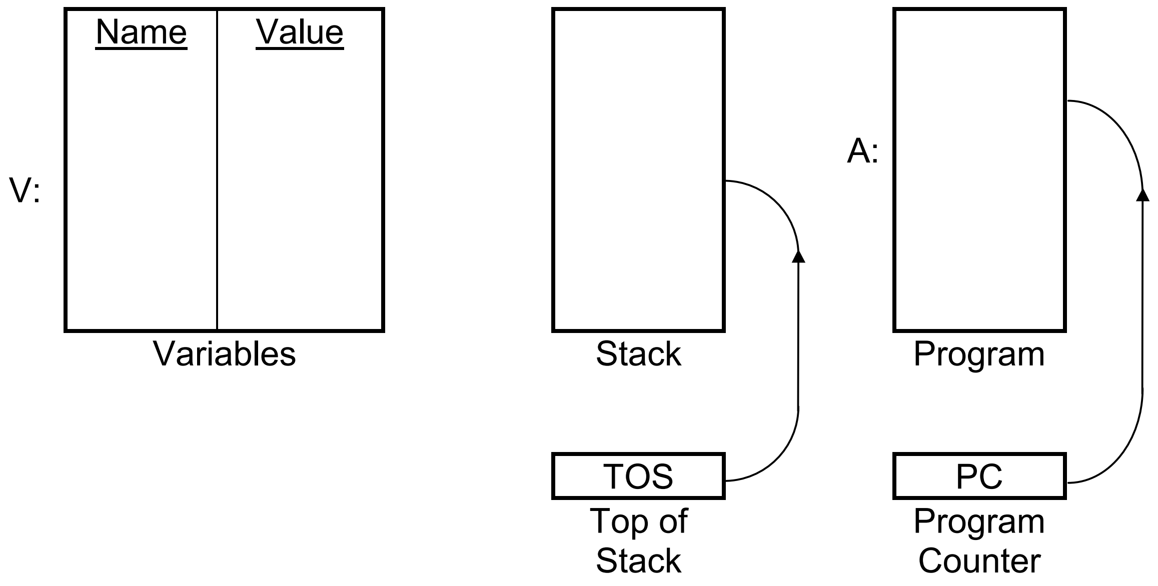
```
procedure main;
begin
program;
if input is not empty then
error ("Program continues after final END;");
endif;
end main;

procedure program;
begin
declarations;
block;
```

```
end program;

procedure declarations;
begin
check ("INTEGER");
name_list;
check (";");
end declarations;

procedure name_list;
begin
name;
while CS =","
loop
next;  -- skip over ","
name;
end loop;
end name_list

procedure name;
begin
if CS is a name  -- access table of tokens to determine the type of CS
then next;
else error;
endif;
end name;

procedure statement_list;
begin
loop
statement;
exit when end of statement list
that is when CS is "ENDIF" or "ELSE" or "END";
end loop;
end statement_list;

procedure statement;
begin
case CS of
when "NOTE"  =>           comment;
when ";"  =>              null;
when name  =>             assignment;
when "IF"  =>             conditional;
when "BEGIN"  =>          block;
when "EXIT"  =>           exit_block;
when "REPEAT"  =>         repeat_block;
when "READ"  =>           input;
when "WRITE"  =>          output;
otherwise error;
end case;
end statement;

procedure comment;
begin
next; -- skip over "NOTE"
while CS <> ";"
loop
next;
end loop;
next; -- skip over ";"
end comment;

procedure null;
begin
next; -- skip over ";"
end null;

procedure assignment;
begin;
next; -- skip over "name"
check (":=");
expression;
check(";");
end assignment;

procedure conditional;
begin
next; -- skip over "IF"
boolean_expn;
check ("THEN");
statement_list;
case CS of
when "ENDIF" =>  next; -- skip over ENDIF
                 check  (";");
when "ELSE" =>   next; -- skip over "ELSE"
                 statement_list;
                 check ("ENDIF");
                 check (";");
otherwise error;
end case;
end conditional;

procedure expression;
begin
```

```
case CS of
when name  =>     next;
when number  =>  next;
when "("  =>        next; -- skip over "("
                        expression;
                        op;
                        expression;
                        check (")");
otherwise error;
end case;
end expression;
…
…
…
```

<u>Exercise</u>: Write a similar procedure for checking a block.

Machine Language for Simple Structured Programming (MLSSP)



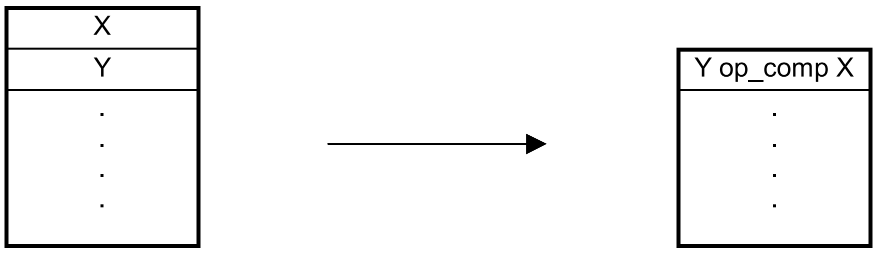
Here are the instructions of the machine language with the above architecture.

(1) PUSH (V), push the value of the variable at address V onto the top of the stack. (Error if the stack is full).

(2) PUSHC(K), push the value of the constant K onto the top of the stack. (Error if the stack is full.)

(3) POP(V), copy the top cell of the stack into the value field of the variable at address V, and then remove the top cell from the stack. (Error if the stack is empty.)

(4-10) +,-,*,/,=,<,>, artithmetic operation or comparison on top two element of stack. The value of the operation or comparison replaces the top two elements of the stack as in the diagram below. (Error if there are fewer than two elements in the stack.)



(11) JF (A), jump to address A in the program if the value on the top of the stack is false (equal to 0) otherwise continue to the next instruction. In any case remove one cell from the stack. (Error if the stack is empty).

(12) JT (A), like 11 but jump when the value at the top of stack is true (different from zero).

(13) J (A), unconditional jump to address A in the program.

(14) READ (V), prompt with the name of the variable at address V to the user and then read number into the value field of the variable at address V.

(15) WRITE (V) , write the name and, value of the variable at address V (e.g. in the form name = value).

(16) SETNAME (V,S), set the name field of the variable at address V to the string S.

(17) HALT, halt the execution.

<u>Exercise:</u>

Write similar definitions for the following instructions.

POP - like POP(V) but no copying of top of stack.
CALL (A)
RETURN,
CHANGE_TOP_OF_STACK (V)
COPY_TOP_OF_STACK(V)

Recursive descent translation to MLSSP

The procedures for translation have a similar structure to the procedures for recursive descent syntax analysis. Let us now assume the following

1) The source program is free of lexical, syntactic and semantic errors. (Recall when translation occurs in compilation.)

2) As before we will use the global variable CS and the procedure "next".

3) We shall also use the global variable CTA - current translation address and a procedure build (instruction, address) for building an MLSSP instruction at a given address.

4) Each procedure corresponding to a non terminal is responsible for translating and skipping over the part of the program derivable from non terminal (to be found at the start of the input onwards) and updating the value of CTA appropriately.

Here then are the procedures for translating from SSPL to MLSSP where for the more complicated procedures, we give the structure, of the translation before giving the translation itself.

```
procedure t_program;  -- translate program
begin
CTA:=0; -- initial value
t_declarations;
t_block;
build (HALT, CTA); CTA:= CTA + 1;
end t_program;

procedure t_declarations;
VA:  -- variable address;
begin
VA:= 0;
next;  -- skip over "INTEGER"
loop
 -- CS is a name
set translation address of CS in table of tokens to VA.
build (setname (VA, CS), CTA);
VA:= VA + 1; CTA:= CTA + 1;
next;  -- skip over name
if CS=";" then next; exit; endif;
next;  -- skip over ","
end loop;
end t_declaration;
```

```
procedure t_statement_list;
begin
loop
t_statement;
exit when end of statement list
that is when CS is "ENDIF" or "ELSE" or "END";
end loop;
end t_statement_list;

procedure t_statement;
begin
case  CS of
when "NOTE" =>          t_comment;
when ";" =>             t_null;
when name =>            t_assignment;
when "IF" =>            t_conditional;
....
....
otherwise bug in compiler;
end t_statement;

procedure t_comment; -- As before

procedure t_null; -- As before
```

<u>Structure of translation of assignment</u>

SSPL	MLSSP
name := expression;	translation of expression POP (address of name)

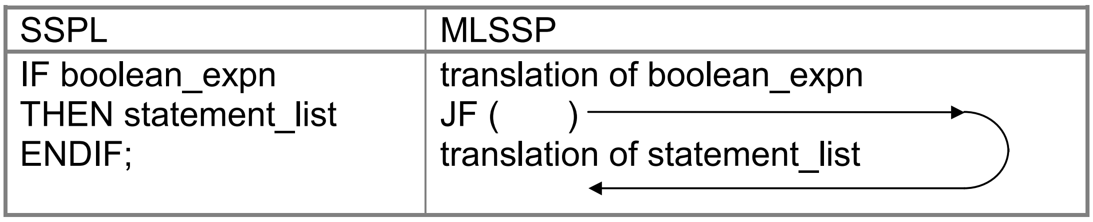
```
procedure t_asignment:
na:  -- address of name
begin
CS is a name so access table
of tokens to determine the address
of the name and store the address in na.
next;  --  skip over name
next;  -- skip over ":="
t_expression;
build (POP (na), CTA)
CTA:= CTA +1
next;  -- skip over ";"
end;
```

Structure of translation of expression

SSPL	MLSSP
name	PUSH (address of name)
number	PUSHC (number)
(expression op expression)	translation of first expression translation of second expression op

```
procedure t_expression:
na:  -- address of name
op:  -- operator
begin
case CS of
when name  =>     find address of CS in table of tokens
                  and store its address in na.
                  build (PUSH (na), CTA); CTA:= CTA +1;
                  next;
when number  =>   build (PUSHC (CS), CTA); CTA:= CTA +1;
                  next;
when "("  =>      next;  -- skip over "("
                  t_expression  -- translate first expression
                  op:=CS; -- save operator in op
                  next;  -- skip over operation
                  t_expression  -- translate second expression
                  build (op, CTA); CTA:= CTA+1;
                  next;  -- skip over ")"
otherwise bug in compiler
end case;
end
```

Structure of translation of conditional assuming no ELSE

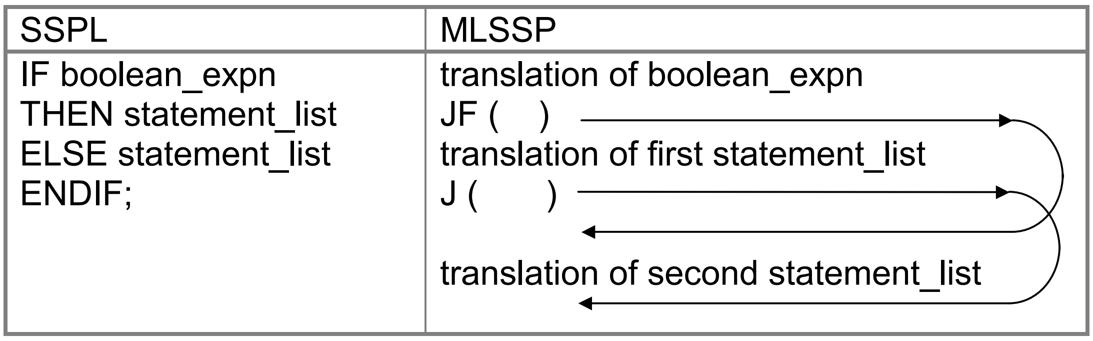


SSPL	MLSSP

IF boolean_expn
THEN statement_list
ENDIF;
	translation of boolean_expn
	JF ( )
	translation of statement_list

```
procedure t_conditional; -- assuming no ELSE
JFA:  -- address of JF
begin
next;  -- skip over "IF"
t_boolean_expn;
JFA:= CTA;  -- Allocate space for JF instruction
CTA:= CTA+1;
next;  -- skip over "THEN"
t_statement_list;
build (JF (CTA), JFA);
next;  -- skip over "ENDIF"
next;  -- skip over ";"
end t_conditional;
```

Structure of translation of conditional assuming an ELSE



SSPL	MLSSP

IF boolean_expn
THEN statement_list
ELSE statement_list
ENDIF;
	translation of boolean_expn
	JF ( )
	translation of first statement_list
	J ( )
	
	translation of second statement_list

<u>Exercises</u>:

1) Write a procedure for translating a conditional on the assumption that there is an ELSE.

2) Write a procedure for translating a conditional whether there is or is not an ELSE.

3) Write three procedures for translating a "block", an "exit_block", a "repeat_block". Remember that a block may be repeated or exited, so information needs to be passed between these procedures.

ACKNOWLEDGEMENT

My sincere thanks to S. Allen, R. Kantor and D. Mohr for their assistance in typing these notes.

WORTHWHILE READING

"Principles of Compiler Design" by Aho, Ullman, and Sethi.

"Introduction to Compiler Construction with Unix" by Schreiner and Friedman.

"Introduction to Automata Theory, Languages and Computation" by Hopcroft and Ullman.

“Compiler Construction: A recursive descent Model” by Elder.