ON CHARACTER SET REDUCTION

by

Reino Kurki-Suonio

Carnegie Institute of Technology
August, 1965

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

Abstract

Character sets of languages can be reduced by representing several different characters by one in case this does not cause ambiguities. For finite-state languages the best reductions can always be found, whereas for context-free languages the problem is, in general, unsolvable. Easily invertible reductions are obtained, however, if the language is embedded into a finite-state language. A method for this is suggested which uses only information on the possible adjacent character pairs.

The parsing problem of the reduced language can be returned to that of the original. The implications of this to precedence grammars are discussed.

Finally, results of some programs for character set reductions are demonstrated.

1. Introduction

The limitations of the available character sets cause many difficulties in designing symbolic languages. Some of the difficulties can, of course, be postponed to the implementations by assuming an unlimited source of symbols. This was done e.g. in designing Algol 60 [6], whose character set includes 116 symbols, 24 of which are represented in the reference language by underlined (boldface) letter strings. In most other languages (and in most implementations of Algol) a great number of reserved words and phrases is used, instead. This leads, of course, to some restrictions on the invention of names for local purposes in a program. In general, the wider and the more "English-like" the language is, the more severe these restrictions are. (Note the corollary: the farther your own language is from English, the less nuisance you have.)

In any case, writing and keypunching phrases like " IF A IS GREATER THAN B" or "DIVIDE B INTO A GIVING C, ON SIZE ERROR GO TO L" may feel convenient for the first time, but being forced to repeat them for the length of any practical program resembles the old punishment to write down 100 times the lesson you did not learn.

In the following we will assume that an extension of the available alphabet is used in the reference language rather than reserved words. The possibility of reducing this character set by combining several characters will be investigated. No attention will be paid to the illegibility of the reduced language. One possible application would be in on-line conversational communication with a computer. One or more symbols could be assigned to each available key. When pushing a key, the computer should decide from the context which symbol you mean and type you back a reasonable string representation for it.

2. Notations

Familiarity with context-free grammars [1], finite-state machines [4] and their use as recognizers [7] is assumed. The language (set of strings) defined by a grammar G will be denoted by L(G). The alphabet of a language (the terminal alphabet of a context-free grammar or the input alphabet of a finite-state machine) will be denoted by $T = \{a,b\ldots\}$. The set of finite strings on T will be $F_T = \{x,y,\ldots\}$. The nonterminals of a context-free grammar will be denoted by S,A,B,... where S is the designated one. Application of one rewriting rule (production) will be denoted by $\rightarrow$; and $\Rightarrow$ will be a sequence of $\rightarrow$'s. When writing down a context-free grammar, we will adopt the symbol $|$ from [6] to indicate repetition of the previous left-hand side of a production.

By the predicate $P(a,b)$ we will mean that there exist strings x, y such that $xaby$ belongs to the given language.

Terminators $\vdash$, $\dashv$ will be used to indicate the left and right end of a string. So $P(\vdash, a)$ and $P(a, \dashv)$ are equivalent to the existence of a string x such that ax or xa belongs to the language, respectively. If the null string belongs to the language, $P(\vdash, \dashv)$ is true.

3. Character Set Reductions

A mapping $f: T \to T'$ of the character set of a language into another character set T' is called a reduction if no two sentences have the same image in $F_{T'}$. This means that f is invertible for sentences and that the reduced representation of the language has exactly the same ambiguities as the original. A reduction is called a left-to-right reduction if no two initial strings of sentences have the same image in $F_{T'}$.

Let L be a finite-state language. We can then find a deterministic finite-state machine M to recognize L. For simplicity, M is allowed to be incomplete, and it is assumed to contain only those states which are reachable from the initial state and from which at least one of the admissible final states is reachable. A mapping f will then transform M into a finite-state machine M', which is, in general, non-deterministic. The mapping f is now a reduction of L in exactly those cases when M' is unambiguous. Futthermore, f is a left-to-right reduction if M' is deterministic. Both these criteria are decidable. Especially, we can then find the reductions with the least number of different characters.

To M' we can add output in T so that in each transition the output is the input symbol of the corresponding transition in M. The inverse mapping of f can now be performed by M' (or by any machine equivalent to it). If f is not a left-to-right reduction, the inversion cannot be performed from left to right without a lookahead, and there is no finite bound for the length of the necessary lookahead.

For context-free languages we will show, at first, that the best reductions cannot be found, in general. This problem is obviously equivalent to deciding whether a given mapping f increases the ambiguity of a language or not. Let G be a context-free grammar. We can then find a standard grammar G' (cf. [5]) where all productions are of the form $A \rightarrow aA_1 \ldots A_n (n \geq 0)$ and which generates exactly the same sentences as G. Replacing G' by a grammar in which all terminal characters in the productions are different, we get an unambiguous grammar G''. If we could decide whether the language $L(G')$ is a reduction of $L(G'')$ or not, we could solve the ambiguity problem of G', which is, in general, unsolvable [2].

The solvability of the reduction problem for finite-state languages suggests embedding a context-free language (or any non-finite-state language) into a finite-state language and requiring the reduction to be a reduction of this wider language. Obviously there is, however, no smallest finite-state language L' covering a non-finite-state language L, since $x \in L'$, $x \notin L$ implies that $L' - \{x\}$ is also a finite-state language.

A simple finite-state language covering a given language L is the language consisting of all strings $a_1 \ldots a_n$ $(n \geq 0)$ such that after extending them with $a_0 = \vdash$, $a_{n+1} = \dashv$ we have $P(a_i, a_{i+1})$ for $i = 0, \ldots, n$. This language will be denoted by $R(L)$, and reductions of $R(L)$ are called <u>simple reductions</u> of L.

For $R(L)$ a finite-state machine M can be constructed easily by taking the set of states $\{S_a ; a \in T\} \cup \{\vdash\}$ and letting a symbol $a \in T$ to cause a state transition from S_b to S_a if and only if $P(b,a)$. The initial state and the admissible final states are $S_{\vdash}$ and $\{a ; P(a, \dashv)\}$, respectively. Since both the inputs and outputs of M' depend only on the states to be entered, the simplification of M' is easy. As an example we consider the language L defined by the grammar P_3 of [3]:

$$
\begin{aligned}
S &\to A \\
A &\to A - B \mid B \\
B &\to B * C \mid C \\
C &\to \theta\ D \mid D \\
D &\to (A) \mid \lambda
\end{aligned}
$$

The finite-state machine M for R(L) is given in Fig. 1.

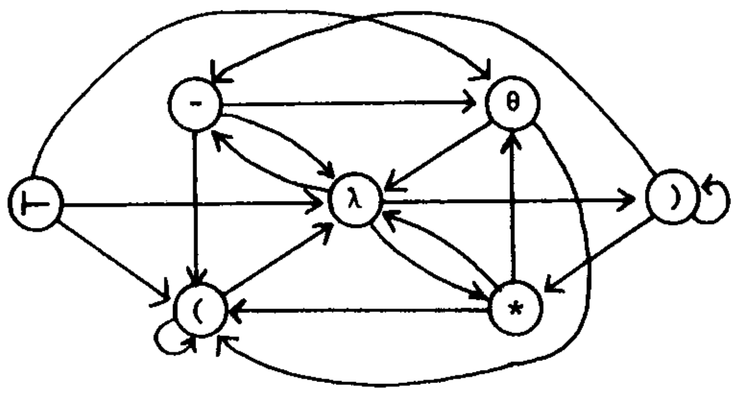

Fig. 1

Let f now be a mapping which maps -, *, θ, (,) and λ to a, b, a, b, c and c, respectively. Since the resulting M' is deterministic, f is a left-to-right reduction. The inverse mapping of f is performed most easily by the two-state machine of Fig. 2, which is the minimal machine equivalent to M'. The two states correspond to whether we are expecting an operator or an operand.

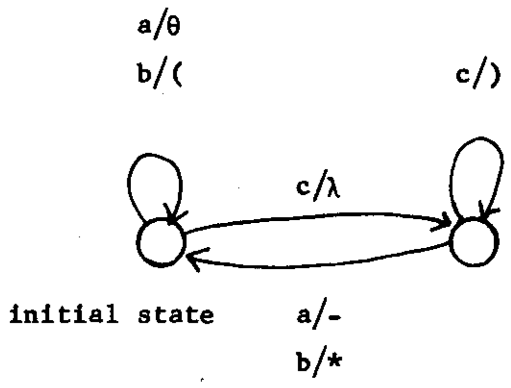

Fig. 2

4. Character Set Extensions

It is sometimes desirable to consider a given language L as a reduction $f(L_0)$ of another language L_0. E.g., a parsing algorithm might be easier to express for L_0 than for L (cf. next section). Also better reductions of L might be found as simple reductions of L_0, since $f(R(L_0))$ is included in $R(L)$.

As an example let us consider the language $L = \{ab^nc^n, db^ne^n ; n > 0\}$. Let a, b, c, d and e be mapped by f to a', b', c', d' and c', respectively. Obviously f is a reduction. It is not a simple reduction, since e.g., a'b'c' is the image of abc and abe, both of which belong to R(L). However, L is a simple reduction $g(L_0)$ of a language L_0 where b is denoted differently depending on the first character, and f is a reduction of the finite-state language $g(R(L_0))$.

Languages are most often given by their grammars. Reducing the character set means then to perform the given character set mapping in the grammar. On the other hand, it is to be noted that a character set extension of a language is not always realizable by a replacement of the terminal characters in the grammar. E.g., in the above example the given extension could be realized for the context-free grammar

$$S \rightarrow aA \mid dB$$
$$A \rightarrow bc \mid bAc$$
$$B \rightarrow be \mid bB$$

but not for the grammar

$$\begin{aligned} S &\to aA \mid dB \\ A &\to Cc \mid CAc \\ B &\to Ce \mid CBe \\ C &\to b. \end{aligned}$$

When speaking of character set reductions and extensions of grammars, we will always assume that no other changes are allowed in the grammar except replacing the occurrences of terminal characters by some other terminals.

For a given grammar G we can now find an extension G_0 such that G_0 is a simple (left-to-right) reduction $f(G_0)$ of G_0 and that the finite-state language $f(R(L(G_0)))$ is as small as possible. To express the necessary and sufficient requirements of G_0 we adopt the following notations:

$a,b,\ldots$ will denote different occurrences of terminal characters in G or in G_0 rather than the different symbols; terminators $\vdash$, $\dashv$ are also treated as terminal characters;

$a = b$ will denote that a and b are represented by the same symbol in G ;

$a \approx b$ will denote that a and b are represented by the same symbol in G_0 . Note that $a \approx a$ for all a.

As G is required to be a simple reduction of G_0 the following must be satisfied:

$$(1) \qquad \begin{cases} \text{if } a_0 \approx b_0 \text{ , } a_n \approx b_n \text{ , } a_i = b_i \text{, } P(a_i, a_{i+1}) \text{ ,} \\ P(b_i, b_{i+1}) \text{ for } i = 0,\ldots,n-1 \text{, then} \\ a_i \approx b_i \text{ for } i = 1,\ldots,n-1 \text{ .} \end{cases}$$

For simple left-to-right reductions (1) reduces to

$$(1') \qquad \text{if } a \approx b \text{ , } P(a,c) \text{ , } P(b,d) \text{ then } c \approx d \text{ .}$$

In section 6 we will discuss some experiments in reducing G by simple reductions of $R(L(G_0))$.

5. Implications To Precedence Grammars

The parsing problem of a reduced language can be returned to that of the original, if a device for inverting the reduction is provided. So reductions may be considered as operations extending any family of grammars for which a given parsing method can be used.

In [3] a family of context-free grammars , precedence grammars (PGs), was introduced, for which the syntax analysis can be formulated in a particularly simple way. To be more exact, the problem of finding all phrases in a sentence is simple, though assigning the proper nonterminals for them might not be straightforward or even unique. Precedence grammars are a special case of operator grammars, in which no productions $A \rightarrow \ldots BC \ldots$ with adjacent nonterminals are allowed. It was shown in [5] that any context-free grammar can be transformed into this form. Slightly differing from [3] we define the following precedence relations for an operator grammar:

$$(2)\quad \begin{cases} a \doteq b & \text{if there is a production } A \rightarrow \ldots aBb\ldots, \\ a \lessdot b & \quad\text{"}\qquad A \rightarrow \ldots aB\ldots \text{ and } B \Rightarrow Cb\ldots, \\ a \gtrdot b & \quad\text{"}\qquad A \rightarrow \ldots Bb\ldots \text{ and } B \Rightarrow \ldots aC, \end{cases}$$

$$(3)\quad \begin{cases} a \dot{\equiv} b & \text{if there is a production } A \rightarrow \ldots ab\ldots, \\ a \trianglelefteq b & \quad\text{"}\qquad A \rightarrow \ldots aB\ldots \text{ and } B \Rightarrow b\ldots, \\ a \trianglerighteq b & \quad\text{"}\qquad A \rightarrow \ldots Bb\ldots \text{ and } B \Rightarrow \ldots a \end{cases}$$

For $\vdash$ and $\dashv$ the relations will be defined as if the designated nonterminal would be S' with a production $S' \rightarrow \vdash S \dashv$. Obviously the relations (3) are different cases of $P(a,b)$.

In [3] the corresponding relations of groups (2) and (3) were not separated, and the criterion for precedence grammars was that for any pair a,b at most one of the relations holds. A trivial relaxation of this is to allow at most one relation of each group (maybe different members of the groups) to hold for any pair. This family can now be further extended by simple (left-to-right) reductions. These grammars will be called (left-to-right) RPG's. It is decidable whether an operator grammar is a (left-to-right) RPG or not.

An example of a left-to-right RPG is the grammar P_2 of [3], which differs from P_3 discussed above only by having one character for both - and θ . In [3] it was pointed out that P_3

is a PG , and we showed above that even a further reduction of P_2 into three characters is a simple left-to-right reduction of P_3.

One of the properties of (left-to-right) RPG's is that they can be embedded into each other so that the resulting grammar is also a (left-to-right) RPG. As an example we consider the following left-to-right RPG's

G:	G':	G":
$S \to S + A \mid A$	$S' \to S' * A' \mid A'$	$S'' \to S'' + A \mid S'' * A'' \mid A''$
$A \to A * B \mid B$	$A' \to A' + B' \mid B'$	$A'' \to (S'') \mid a$
$B \to (S) \mid a$	$B' \to (S') \mid a$	

which generate the same language with different precedence relations between * and + . If we want to be able to convert the precedence relations of G inside an expression to those of G' , we can add the productions

$$B \to [S']$$
$$B' \to [S]$$

where [and] are new delimiters indicating the change of the precedence relations. If we want to be able to use any of the three possible precedence relations, we can add the productions

$$B \to [S'] \mid \{S''\}$$
$$B' \to [S''] \mid \{S\}$$
$$B'' \to [S] \mid \{S'\} \; .$$

It is to be noted, that in order to preserve the grammar as an RPG, the new delimiters must be used so that, when knowing the current subgrammar and scanning a delimiter, no extra information is needed to know which subgrammar is to be entered.

It is to be noted that a character set extension of an RPG is not necessarily an RPG. Hence the set of RPG's is not closed under the reduction technique introduced in section 4. On the other hand, this technique does not give all those reductions which are RPG's themselves, since some of them might not be simple reductions of the same PG's as the original. However, the ambiguity problem is decidable for RPG's and hence the best RPG-reductions of an RPG can always be found.

6. Experiments With Programs For Character Set Reduction

The above ideas were implemented as a series of related programs*. The main interest was in programs for finding character set reductions of context-free grammars with the method introduced in section 4.

The programs were applied to the Algol-like language defined in [3]. This language was selected instead of Algol 60 because of its more concise syntax. (Some minor changes were made to the syntax; e.g., it was transformed into a strict "Backus normal form".) Denoting the language defined by this grammar by L, the

*The CDC G-21 computer at Carnegie Tech was used. All programming was done in Algol.

best simple left-to-right reductions of R(L) which were found had 17 characters less than L. One possibility for such reduction is to combine (~) characters as follows:

real	~	∨	~	false	a
integer	~	⊃	~	true	
Boolean	~	≡	~	¬	b
function	~	value	~	∧	c
,	~	array			
until	~	switch			
while	~	constant			d
:	~	for			
↑	~	$_{10}$			
do	~	if			
else	~	procedure			e
comment	~	step			f
]	~	begin			g

When allowing the program to consider L as a simple left-to-right reduction of another language*, two more characters could be reduced, e.g.,

then	~	go to	h
:=	~	.	i

*The extension was, in fact, one in which digits were represented differently in identifiers and digit strings.

For this reduction a 4-state machine was found for the inverse mapping. This machine is given in Fig. 3. The characters a, a',..., n represent sets of output symbols associated with the state-transitions. The sets a ... i are given above; the dashes indicate the column, e.g., a = {real integer}, a' = $\{\vee \supset\}$ and a" = {false true}. The sets j ... n are:

$j = \{ \; [\; (\; + \; - \; * \; / \; \div \; = \; < \; > \; \neq \; \leq \; \geq \; \}$,

k = {) end } ,

l = { ; } ,

m = { 0 ... 9 } ,

n = { a ... z } .

The initial state of the machine is the state 1.

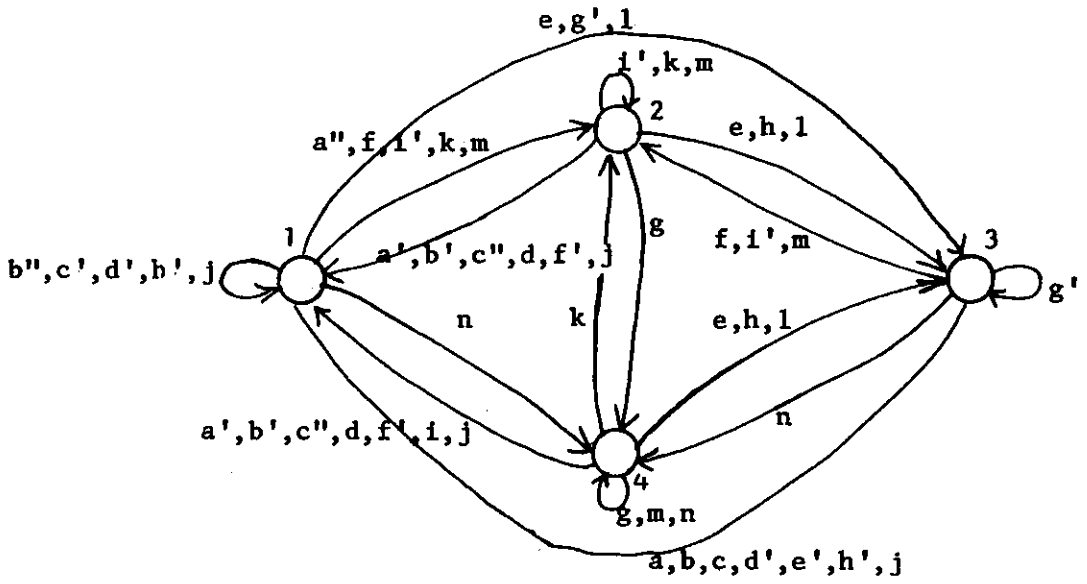

Fig. 3

When the reduction was not required to be a left-to-right reduction, one more reduction could be made, e.g.:

$\underline{\text{end}} \quad \sim \quad \leq$

To reverse this final reduction, a 6-state non-deterministic machine could be constructed, in which only a one character lookahead is required to distinguish between end and $\leq$.

REFERENCES

1. Chomsky, N., "On Certain Formal Properties of Grammars", Information and Control 2 (1959), 137-167.

2. Floyd, R. W., "On Ambiguity In Phrase Structure Languages", Comm. ACM 5 (Oct. 1962), 526.

3. Floyd, R. W., "Syntactic Analysis and Operator Precedence", J. ACM 10 (July 1963), 316-333.

4. Gill, A., Introduction To The Theory of Finite-State Machines. McGraw-Hill, New York, 1962.

5. Greibach, S. A., "A New Normal-Form Theorem For Context-Free Phrase Structure", J. ACM 12 (Jan. 1965), 42-52.

6. Naur, P. (Ed.), "Revised Report On The Algorithmic Language ALGOL 60", Comm. ACM 6 (Jan. 1963), 1-17.

7. Rabin, M. O. and Scott, D., "Finite Automata and Their Decision Problems", IBM J. Res. Develop. 3 (1959), 114-125.