An extended BNF for specifying the syntax of declarations

by GORDON WHITNEY

Western Electric Engineering Research Center
Princeton, New Jersey

"I recommend the use of suitable special metalanguages as a part of the defining reports. The Backus notation is probably as good as anything we have at present, but it still leaves a great deal to be desired . . . many syntactic rules cannot be expressed in it. The language reporter should . . . invent and use such tools wherever the special metalanguage is easier to work with than natural language." Peter Naur,1 1963.

INTRODUCTION

The syntax of ALGOL is specified in its defining report by a grammar whose rules are given in Backus-Naur Form (BNF). Because of the need for precision in the specification of complex systems, BNF has been used to record the syntax of many other programming languages. The rules of BNF are equivalent to context-free production rules. However, due to its declarations, ALGOL is context-dependent, and its syntax cannot be fully specified by a grammar limited to context-free rules. Other grammars for ALGOL are more compact than the BNF of the defining report, but they are generatively no more powerful. Because of this context-dependency of declarations, even an assembly language cannot be fully specified in BNF.

This paper defines an extension to BNF which permits the specification of the syntax of declarations, while retaining the definitional power of BNF as a subset. PL/I has been formally defined by specifying a model which will execute programs written in the language. This model specifies the semantics of the language but does not include a grammar for the syntax of declarations. Recent papers define systems capable of specifying the syntax of declarations. This paper is an extension and exposition of one of these models* (called "table-grammar") in the hope that language reporters and implementors will be able to utilize the model in practical work with programming languages. Consistent with this purpose, a system of notation has been adopted (with an attendant loss of compactness) which is both compatible with BNF and oriented for the human reader of the syntax.

Table-language generation

A grammar defines a language by specifying the means by which legal strings in the language can be generated starting from some fixed initial configuration. For each class of language there exists a corresponding class of processors which can carry out the generative process. Such a processor is called a generator. This section will define the form of a table-grammar and the means by which they generate strings.

* The table-grammar presented here is similar but not identical to a previous model. The differences are as follows:

1. The table-functions using Greek letters as function names have been changed to expression format using operators with English names. Thus the \( r \)-function is replaced by the use of create, retrieve or recreate. The \( p \)-function becomes replaces. The \( g \)-function becomes illegal.

2. The use of two auxiliary storage tapes rather than one gives the new model increased generative power. This also allows BNF grammars to remain nearly intact when table operations are added to the grammar. The table operations copytable and erasetable are new and are necessary to allow multiple tables in a two-tape system.

3. The definition of the generation operators to perform recognition is new, as is the operator copy and both the definitions and the properties of "Regular Table Expressions."
Notation

Certain definitions will be used throughout. They will not be given further local definition. These definitions are:

- \(a, b, c, \ldots\) terminals symbols.
- \(e\) a terminal-string of length zero.
- \(x, y, z\) terminal-strings, having an arbitrary value, or a range of values.
- \(+\) this post-fix operator denotes \(e\)-free closure for catenation, i.e., \((X)^+ = \{ \sum X^i \text{ where } i=1 \}
- \(*\) this post-fix operator denotes full closure for catenation, i.e., \((X)^* = \{ (X)^+ \cup e \}.
- \(\lambda\) this symbol denotes a string of either letters or blanks and is called a letter-string.

Definition of a table-grammar

A table-grammar is a 4-tuple: 
\[ G = (\text{categories}, \text{terminals}, \text{static-rules}, \text{sentence-symbol}) \]

where:

1. The set of categories and the set of terminals are finite and disjoint.
2. The set of categories has two disjoint subsets: grammar-categories and table-categories.
3. The set of static-rules is finite. Each rule has the form: \(\langle \lambda \rangle :: = (\text{grammar-category U terminal U table-expression}) \ast \) where:
   - a. The leftside of a rule is a grammar-category.
   - b. The rightside is called an arbitrary-string.
4. A table-expression consists of a generation-operator with its respective operands. A generation-operator has either zero, one or two operands, and is respectively referred to as a niladic, monadic or dyadic operator. The operands for table-expressions are table-categories whose form is \(\langle \cdot \cdot \lambda \rangle\). For the presentation of table-expression schema it will be convenient to use \(\langle \cdot \cdot \lambda \rangle\) to represent an incident table-category (i.e., one entering the table) and \(\langle \cdot \cdot \cdot \rangle\) to represent a resident table-category (i.e., one already present in the table). The following are the schemas for the seven different forms of table-expressions:
   - a. The niladic expressions are: copytable and erase-table.
   - b. The monadic expressions are: create \(\langle \cdot \cdot \cdot \rangle\, \text{retrieve} \langle \cdot \cdot \cdot \rangle\) and illegal \(\langle \cdot \cdot \cdot \rangle\).
   - c. The dyadic expressions are:
     \(\langle \cdot \cdot \cdot \rangle \text{recreates} \langle \cdot \cdot \cdot \rangle\) and \(\langle \cdot \cdot \cdot \rangle \text{replaces} \langle \cdot \cdot \cdot \rangle\)

Table expressions, when evaluated, have the following characteristics:

- d. Successful evaluation may be subject to certain restrictions.
- e. A terminal-string is returned as the value of the expression.
- f. Side-effects may be produced in the active-table.
   (The concept of active-table will be defined below.)

5. The sentence-symbol is a unique grammar-category.

Concerning the evaluation of table-expressions, if the specified conditions are not satisfied, the current generation step is blocked and an alternate sequence of steps must be utilized. Thus invalid programs are eliminated at the point in the generation sequence where an incorrect construction would have appeared.

String processing devices

A string processing device is a machine with a finite control, with a system of one or more auxiliary tapes providing potentially infinite storage, and with either an input-tape or with an output-tape or with both. The device is called deterministic if in every possible configuration it has only one possible move to its next configuration. If the device has more than one possible move, it is called nondeterministic. A device with an input-tape is called a recognizer. A device with an input and output tape is called a transducer. A device with only an output-tape is called a generator. The sets defined by a context-free grammar are exactly the sets generated by some nondeterministic generator whose auxiliary storage is a pushdown and whose output-tape is one way.

If an input (or output) tape of a device is one-way (usually this is left to right) then the device is called on-line and its input (or output) need not be written on a tape at all but need only be received (or transmitted) one character at a time without the use of any local storage.

A table-language generator

Figure 1 shows the type of device necessary to generate table-languages. The output is one way and the output-string is not counted as a tape. In a one pass processor, as declarations of identifiers are encountered, they must be recorded in some form of auxiliary storage to allow for subsequent retrieval. The table-tape stores a sequence of declaration tables as a pushdown list of
Figure 1—A nondeterministic two-tape generator for a language defined by a table-grammar

tables. The top most table is designated as the active-table. When a block structure is not required, only the active-table will appear on the table-tape. The operation of the table is subject to four restrictions:

1. Entries must be dynamic-rules of the form:
   
   \[(\cdot \alpha \cdot) \rightarrow \text{identifier}\].

2. \((\cdot \alpha \cdot)\) is a table-category. A simple set of table-categories would be
   
   \{\langle \cdot \text{integer} \cdot \rangle, \langle \cdot \text{real} \cdot \rangle, \langle \cdot \text{Boolean} \cdot \rangle\}.

3. An identifier is an element from a regular set. This means that an identifier can be recognized by a finite automaton. A simple set would be \langle \text{letter} \rangle (\langle \text{digit} \rangle)^*\) where a3 and j49 would be identifiers.

4. The identifiers within the active-table must be unique. The same identifier may not appear twice. Note by way of contrast that the same table-category may appear many times. Thus:
   
   \[(\cdot \text{real} \cdot) \rightarrow \text{a56}\](\cdot \text{integer} \cdot) \rightarrow \text{a56}\]

is a valid table; while

\[(\cdot \text{real} \cdot) \rightarrow \text{a56}\](\cdot \text{integer} \cdot) \rightarrow \text{a56}\]

is not a valid table.

An identifier is said to be declared if it appears in the active-table, otherwise it is undeclared. Each new declaration which is added to the active-table must contain an identifier which is undeclared.

**Table-grammar generation sequences**

The operator copytable explained in the section "Table operators for embedded blocks". When this operator is not used in a grammar, only a single table copy will appear on the table-tape. The description of generation sequences for this portion of the paper will be limited to those generations in which only one table copy is present on the table-tape. A generation sequence is a finite list of transitive steps. Each step has the form:

\[\text{ID}_1 \Rightarrow \text{ID}_2\]

where ID stands for instantaneous description and \(\Rightarrow\) indicates the rewriting of ID; in accordance with an evaluation algorithm to produce (i.e., generate) ID. An instantaneous description is a triple of the form:

\[(\text{terminal-string}, \text{arbitrary-string}, \text{table-configuration})\]

where arbitrary-string represents the concatenation of unreduced rightsides of static-rules and where a table-configuration is one distinct arrangement of declarations out of a potentially infinite number of such configurations labeled \(T_0, T_1, T_2, \ldots\). In actual sequences, the parentheses and commas may be omitted with no loss of meaning by the use of disjoint alphabets. The active-table is erased on the last step of a generation sequence.

The evaluation algorithm for a step has three parts:

1. If \(\text{ID}_1 = (x, \alpha, \gamma, T_i)\), then \(\text{ID}_2 = (x, \alpha, \gamma, T_i)\).
2. If \(\text{ID}_1 = (x, \alpha, \gamma, T_i)\) and \(\alpha \vdash = \beta\) is a static-rule of G, then one possible value is \(\text{ID}_2 = (x, \beta, \gamma, T_i)\).
3. If \(\text{ID}_1 = (x, \alpha, \gamma, T_i)\) and the table-expression is such that given an active-table \(T_i\), the expression returns as its value y and as a side-effect changes \(T_i\) to \(T_j\), then \(\text{ID}_2 = (xy, \gamma, T_j)\).

It is now possible to give a precise definition of a table-language.

**Definition of a table-language**

A table-language is a set of strings generated by a function \(L\) having two arguments:

\[L(G, R) = \{x \mid G \text{ is a table-grammar and } R \text{ is a regular set, and where } (e, \text{sentence-symbol}, T_0) \Rightarrow x, \text{ using the static-rules of } G \text{ and using } R \text{ as an identifier set, and where } \Rightarrow \text{ is the transitive closure of } \Rightarrow \}\].

**Formal properties**

Table-languages can be defined by means of a number of alternative models. The formal properties listed below pertain only to the model defined above with a unified identifier set and where the identifiers in the table are isomorphic to the terminal identifiers:
1. The languages are properly contained in the context-sensitive languages.
2. The languages properly contain the context-free languages.
3. The languages are closed under *, homomorphism, and intersection with a regular set.
4. The languages are not closed under union, catenation, or inverse homomorphism.

**Compound rules**

The static-rule schemas given above can be called simple. BNF uses a rule schema which will be called compound. If \( \langle a \rangle ::= \beta \) and \( \langle a \rangle ::= \gamma \) are rules in a grammar then they can be replaced by the single compound rule \( \langle a \rangle ::= \beta|\gamma \). Compound rules will be used where convenient. They are generatively no more powerful than simple rules.

**Table entry and access**

Two basic table operations are the recording and the retrieval of a declaration. These operations are achieved by the expressions \( \text{create} \langle \cdot i \rangle \) and \( \text{retrieve} \langle \cdot r \rangle \) respectively, where \( \langle \cdot i \rangle \) stands for an incident table-category (i.e., one entering the table) and \( \langle \cdot r \rangle \) stands for a resident table-category (i.e., one already present in the table).

**Table-entry**

The operator \( \text{create} \) is employed to specify the syntax of the declaration of an identifier. The expression \( \text{create} \langle \cdot i \rangle \), when evaluated, has the following characteristics:

1. The evaluation cannot be blocked if any additional undeclared identifiers exist.
2. The selected-identifier must be undeclared and is returned as the value which replaces the invoking expression within the intermediate-string of the generation sequence. (This will be made more explicit in the sequel by examples.)
3. The dynamic rule, \( \langle \cdot i \rangle \rightarrow \langle \cdot r \rangle \), is added to the top of the active-table.

In summary, the expression \( \text{create} \langle \cdot i \rangle \) when evaluated selects an undeclared identifier, places it in the output-string and adds to the active-table the indicated declaration.

**Table-access**

The operator \( \text{retrieve} \) is used to specify the syntax of an identifier which can appear only if it has been previously declared. The expression \( \text{retrieve} \langle \cdot r \rangle \), when evaluated, has the following characteristics:

1. The evaluation is blocked if the specified resident category \( \langle \cdot r \rangle \) does not appear in the active-table.
2. The identifier in the selected dynamic rule is returned as the value.
3. There are no side-effects in the active-table.

In summary, the expression \( \text{retrieve} \langle \cdot r \rangle \) when evaluated selects a dynamic rule whose leftside is \( \langle \cdot r \rangle \) and places in the output-string the identifier appearing on the rightside of the selected rule.

**Examples of context-sensitive table-languages**

The context-sensitive set with strings of the form \( w_1 w_2 \ldots w_n \) where \( w_i = w_2 \) and where \( w_1 \) and \( w_2 \) are in the set \((0 U 1)^+\) can be generated by the table-grammar:

\[
\langle a \rangle ::= \text{create} (\cdot b \cdot) (c) \\
\langle c \rangle ::= d \\
\text{retrieve} (\cdot b \cdot)
\]

where the identifier set is \((0 U 1)^+\). A typical sequence is:

\[
\langle a \rangle T_0 \Rightarrow 01101 \quad \langle c \rangle T_1 \Rightarrow 01101d01101
\]

where \( T_1 = \langle (\cdot b \cdot) \rightarrow 01101 \rangle \).

The context-sensitive set with strings of the form \( w_1 w_2 \ldots w_n \) where \( w_i \neq w_j \) for all \( i \neq j \) and where \( w_i \) is in the set \((0 U 1)^+\) is generated by the table-grammar:

\[
\langle a \rangle ::= \text{create} (\cdot b \cdot) (\langle a \rangle) \\
\langle c \rangle ::= \epsilon
\]

where the identifier set is \((0 U 1)^+\). A typical sequence is:

\[
\langle a \rangle T_0 \Rightarrow 010d \quad \langle a \rangle T_1 \Rightarrow 010d011d (\langle a \rangle) T_2 \\
\Rightarrow 010d011d11d (\langle c \rangle) T_3 \Rightarrow 010d011d11d
\]

where \( T_1 = \langle (\cdot b \cdot) \rightarrow 010d \rangle \),
\( T_2 = \langle (\cdot b \cdot) \rightarrow 1011d (\cdot b \cdot) \rightarrow 010d \rangle \), and
\( T_3 = \langle (\cdot b \cdot) \rightarrow 11d (\cdot b \cdot) \rightarrow 1011d (\cdot b \cdot) \rightarrow 010d \rangle \).

**Table operators for contextual declarations**

Programming languages utilize three different methods of associating table-categories with identifiers: explicit declaration, implicit declaration and contextual declaration. Explicit declaration of variables is used in Algol to assure that the declaration of a variable precedes its use. Syntax for explicit declaration preceding use can be specified by the operator sequence:

\[
\text{create} (\cdot a \cdot) \ldots \text{retrieve} (\cdot a \cdot)
\]
An instance of contextual declaration, is the associa­
tion of the category “label” with an identifier by means
of the context in which the identifier appears. Two
distinct contexts are used to cause such a declaration and
these can be meaningfully distinguished by the use of the
table-categories ⟨·proper label·⟩ and ⟨·improper label·⟩. For example, if the integer “25” appears in
columns 1 to 5 of a Fortran statement, then the integer
“25” has been contextually declared to be a ⟨·proper label·⟩. On the other hand, if the integer “25” appears
in a GO TO statement prior to its declaration as a
⟨·proper label·⟩, then the integer “25” will initially be
declared as an ⟨·improper label·⟩. Two additional
operators, recreate and illegal, are provided to handle
the table activity associated with this kind of declara­
tion.

Altering a table-entry

The operator recreate is used to alter a table-category
as a side-effect of generating a declared identifier. The
expression ⟨·i· recreate ⟨·r·⟩, when evaluated, has the
following characteristics:
1. The evaluation is blocked if the specified resident
category ⟨·r·⟩ does not appear in the active-table.
2. The identifier in the selected dynamic-rule is
returned as the value.
3. In the selected dynamic-rule, the incident category
 ⟨·i·⟩ replaces the resident category ⟨·r·⟩.
In summary, the expression ⟨·i· recreate ⟨·r·⟩ when
evaluated selects a dynamic-rule whose leftside is
 ⟨·r·⟩, changes ⟨·r·⟩ to ⟨·i·⟩, and places in the output-string
the identifier appearing on the rightside of the selected
rule.

Scan for an invalid category

The operator illegal is used to specify a scan for an
illegal table-category. The expression illegal⟨·r·⟩, when
evaluated, has the following characteristics:
1. The evaluation is blocked if the specified resident
category ⟨·r·⟩ is present in the active-table.
2. The value returned is the empty-string, ε.
3. There are no side-effects in the active-table.
In summary, the expression illegal⟨·r·⟩ when evaluated
is blocked if ⟨·r·⟩ is in the active-table, otherwise it
returns ε and has no side-effects.

Example of a micro-assembly language (MAL)

An example in the form of a micro-assembly language
(abbreviated MAL) will be used to illustrate the
definitional power of the four operators given above. It
will further show how a BNF grammar can be altered to
become a table-grammar while preserving most of the
original grammar-categories. The resulting table-gram­
mar is only slightly larger than the original BNF
grammar. MAL is not intended to be useful as a
programming language. A free-form syntax has been
chosen to permit a definition which is not sensitive to
blanks or lines. MAL has “dc” for “define constant” and
“ds” for “define storage.” Statements are terminated by
semicolons, labels have a colon suffix, and the period is
used as a separator.

Sample program in Micro-Assembly Language (MAL)

```mal
a: dc . 12;
b: ds;
c: load . a;
    store . b;
    goto . c;
end;
```

BNF grammar for MAL

The syntax of BNF can be defined by the following
BNF rules:

```bnf
(program) ::= ⟨body⟩ end;
(body) ::= ⟨statement⟩; | ⟨statement⟩; ⟨body⟩
⟨statement⟩ ::= ⟨declarative statement⟩ | ⟨imperative statement⟩
⟨declarative statement⟩ ::= ⟨data label⟩ : dc . ⟨integer⟩ | ⟨data label⟩ : ds
⟨imperative statement⟩ ::= ⟨imperative label⟩ : ⟨unlabeled imperative⟩ |
                         ⟨unlabeled imperative⟩
⟨unlabeled imperative⟩ ::= goto . ⟨imperative label⟩ |
                          ⟨operation⟩ . ⟨data label⟩
⟨operation⟩ ::= load | add | store
⟨imperative label⟩ ::= ⟨name⟩
⟨data label⟩ ::= ⟨name⟩
⟨name⟩ ::= ⟨letter⟩ | ⟨name⟩ ⟨digit⟩ | ⟨name⟩ ⟨letter⟩
⟨integer⟩ ::= ⟨digit⟩ | ⟨digit⟩ ⟨integer⟩
⟨letter⟩ ::= a|b|c|z
⟨digit⟩ ::= 0|1|9
```

Semantic restraints on MAL programs

Certain restraints on MAL programs have in the past
been classed as “semantics.” However their formaliza­
tion by a table-grammar shows them to be syntactic
though not context-free. These restraints are:
1. All labels must be unique.

From the collection of the Computer History Museum (www.computerhistory.org)
2. \(\langle \text{data label} \rangle\) must be declared by a \(\text{dc}\) or \(\text{ds}\) statement before it is used in an \(\langle \text{imperative statement} \rangle\). The op-codes "load," "add" and "store" must reference a \(\langle \text{data label} \rangle\) not an \(\langle \text{imperative label} \rangle\).

3. A "goto" op-code must reference an \(\langle \text{imperative label} \rangle\), not a \(\langle \text{data label} \rangle\) or an improper label (i.e., one for which there is no proper label declaration).

**Unrestrained BNF generations**

The following generations of illegal programs are possible using the BNF grammar for MAL when these so-called "semantic" restraints are not observed:

**The table-grammar for MAL**

The following table-grammar for MAL is a revision of the BNF grammar given above. Rules preceded by a * have been altered. Unmarked rules are retained unchanged. Rules which do not appear have been omitted.

* (program) ::= =
* (body) \(\Rightarrow\) illegal \(\langle \text{improper label} \rangle\) end;
* (body) ::= = \(\langle \text{statement} \rangle\) ; \(\langle \text{statement} \rangle\) ; \(\langle \text{body} \rangle\)
* (statement) ::= = \(\langle \text{declarative statement} \rangle\)
* (declarative statement) ::= =
  * (declarative statement) ::= =
    create \(\langle \text{data label} \rangle\) : \(\text{dc}\) . \(\langle \text{integer} \rangle\)
    create \(\langle \text{data label} \rangle\) : \(\text{ds}\)
  * (imperative statement) ::= = \(\langle \text{unlabeled imperative} \rangle\)
  create \(\langle \text{imperative label} \rangle\) : \(\langle \text{unlabeled imperative} \rangle\)
  \(\langle \text{imperative label} \rangle\) \(\text{recreates}\) \(\langle \text{improper label} \rangle\)
    \(\langle \text{unlabeled imperative} \rangle\)
* (unlabeled imperative) ::= = goto . create \(\langle \text{improper label} \rangle\)
  goto . \(\text{retrieve}\) \(\langle \text{improper label} \rangle\)
  goto . \(\text{retrieve}\) \(\langle \text{imperative label} \rangle\)
  \(\langle \text{operation}\rangle\) . \(\text{retrieve}\) \(\langle \text{data label} \rangle\)
  \(\langle \text{operation}\rangle\) ::= = load | add | store
  \(\langle \text{integer}\rangle\) ::= = \(\langle \text{digit} \rangle\) | \(\langle \text{digit}\rangle\) (integer)
  \(\langle \text{digit}\rangle\) ::= = 0 | 9

The categories \(\langle \text{Name} \rangle\) and \(\langle \text{Letter} \rangle\) have been deleted from the BNF grammar and are replaced by the identifier set \(\langle \text{letter} \rangle\) \(\langle \text{letter} \rangle\) \(\cup\) \(\langle \text{digit} \rangle\) *. Identifiers then are any alphameric string beginning with a letter. There are three table-categories: \(\langle \text{data label} \rangle\), \(\langle \text{imperative label} \rangle\) and \(\langle \text{improper label} \rangle\). The categories data label and imperative label have been changed from grammar-categories to table-categories and their original BNF rules have been deleted. The following list itemizes the way in which the table-grammar satisfies the three "semantic" restraints given above for MAL programs:

1. Identifiers are generated only by table-expressions. Uniqueness of declared identifiers is assured by the definition of the operator create.
2. The table-category \(\langle \text{data label} \rangle\) is entered into the table only by the rule for \(\langle \text{declarative statement} \rangle\). Declaration must precede use because in the rule for \(\langle \text{unlabeled imperative} \rangle\), the grammar-category \(\langle \text{operation}\rangle\) refers to an identifier only by accessing the table through "\(\text{retrieve}\) \(\langle \text{data label} \rangle\)" and \(\text{retrieve}\) returns as its value only identifiers previously declared.
3. In the rule for \(\langle \text{unlabeled imperative} \rangle\) the following cases are handled:
   a. create \(\langle \text{improper label} \rangle\) allows a label to enter the table before it is properly declared.
   b. retrieve \(\langle \text{improper label} \rangle\) allows a label which was the argument of a prior goto, to be retrieved as a reference by another goto, still prior to its proper declaration.
   c. retrieve \(\langle \text{imperative label} \rangle\) allows a label which has been previously properly declared to be used as the argument of a goto.
   d. create \(\langle \text{imperative label} \rangle\) allows the contextual declaration of a label to precede its use in a goto.
   e. \(\langle \text{imperative label} \rangle\) \(\text{recreates}\) \(\langle \text{improper label} \rangle\) allows a label previously referred to by a goto to be redeclared as a proper label.
   f. \(\text{illegal}\) \(\langle \text{improper label} \rangle\) blocks the generation sequence in a case when goto references have not been redeclared as a proper label.

**Table operators for embedded blocks**

Certain programming languages allow a block structure in which declarations outside of a block have a scope which extends inward, while declarations within a block are local to the block. An identifier already declared outside a block may be redeclared within the block to...
refer to a new address which will not agree with the one assigned to the same identifier when used outside the block. These requirements indicate that the table-tape must be able to operate as a pushdown store in which tables are handled as units of information. The top most table is designated the “active-table.” All operators access only this table, while lower copies are inaccessible to the operators.

Table-tape pushdown operations

The niladic operators copytable and erasetable are used to control the table-tape as a pushdown store whose unit of information is a table. When evaluated, these operators have the following characteristics:

1. They cannot be blocked.
2. They return as a value the empty-string, $\varepsilon$.
3. copytable has the side-effect of placing a duplicate copy of the active-table on the top of the table-tape.
4. erasetable has the side-effect of erasing the active-table from the top of the table-tape.

Control of local and nonlocal identifiers

The need for a special operator to provide initialization for nonlocal identifiers is best explained by an example. Consider a table-grammar having only the table-category $\langle \cdot r \cdot \rangle$, and whose identifiers are alphanemic strings beginning with a letter. Let a generation sequence proceed until a new inner block has just been initialized, a copytable has just been executed, and the table-tape contains $T_1 T_1$. Horizontal snapshots of the table-tape are oriented so that the active-table is on the left.

$$T_1 = (\langle \cdot r \cdot \rangle \rightarrow a5)$$

Let the active-table $T_1$ be updated via create $\langle \cdot r \cdot \rangle$ so that the table-tape contains $T_2 T_1$ and

$$T_2 = (\langle \cdot r \cdot \rangle \rightarrow k9)$$

Within the active-table there is no information which indicates that $k9$ is local and that both $a5$ and $cd$ are nonlocal to the current block. In order that such a distinction could be achieved, a new expression must be defined. This expression is $\langle \cdot i \cdot \rangle$ replaces $\langle \cdot r \cdot \rangle$, which when evaluated has the following characteristics:

1. The evaluation cannot be blocked.
2. The value returned is the empty-string, $\varepsilon$.
3. As a side-effect, each instance of the specified resident category $\langle \cdot r \cdot \rangle$, within the active-table, is changed to the specified incident-category $\langle \cdot i \cdot \rangle$.

Returning to the example given at the beginning of this section, if the expression $\langle \cdot n r \rangle$ replaces $\langle \cdot r \cdot \rangle$ were evaluated following copytable, then the table-tape contains $T_3 T_1$ and

$$T_3 = (\langle \cdot n r \rangle \rightarrow a5)$$

A subsequent evaluation of create $\langle \cdot r \cdot \rangle$ would give a table-tape of $T_4 T_1$,

$$T_4 = (\langle \cdot r \cdot \rangle \rightarrow k9)$$

In $T_4$, the desired distinction between local and nonlocal identifiers has been obtained.

Examples of embedded blocks

The BNF grammar

$$(a) ::= [\langle c \rangle] | [\langle c \rangle (a)]$$

$$(c) ::= (d) | (d), \langle c \rangle$$

$$(d) ::= 0 | 1 | 1 (d) | 0 (d)$$

generates the strings:

$$\langle a \rangle T^* \Rightarrow [0, 1, 101 [1, 010, 0]]$$

$$\langle a \rangle T^* \Rightarrow [0, 1, 101 [101, 1011]].$$

Now add the restraint that values for $\langle d \rangle$ appearing in the rule for $\langle c \rangle$ may appear only once at each level of self-embedding. This is equivalent to the restriction that an identifier be declared only once on each level and that identifiers declared on an outer level may be redeclared on an inner level. Given this restraint, generation (4) is well formed but generation (5) is not, because 101 is repeated on the lowest level. The following table-grammar generates (4) but will not generate (5) thus satisfying the restraint stated above.

$$(a) ::= [copytable (b) \langle a \rangle \langle b \rangle \langle c \rangle \langle erasetable \rangle]$$

$$(b) ::= (\langle n r \rangle \langle c \rangle \langle l r \rangle \langle c \rangle)$$

From the collection of the Computer History Museum (www.computerhistory.org)
\[ \langle c \rangle ::= \langle d \rangle \mid \langle d \rangle, \langle c \rangle \]  
(2)

\[ \langle d \rangle ::= \text{create}\langle\cdot \text{local}\cdot\rangle \mid \text{recreate}\langle\cdot \text{nonlocal}\cdot\rangle \]  
(3)

Note that (2) and (3) are revisions for the table-grammar of (1) and (3) respectively, and that the table-category \langle\cdot \text{nonlocal}\cdot\rangle enters the active-table in the rule for \langle d \rangle and is utilized in the rule for \langle d \rangle.

The following is the table-grammar generation sequence for the string given in (4) above.

\[ (a)\overrightarrow{T_0} \Rightarrow [b] (a) \text{erasetable}] T_0 T_0 \]  
(7)

\[ \Rightarrow [0, 1, 101 (a) \text{erasetable}] T_1 T_0 \]  
(8)

\[ \Rightarrow [0, 1, 101 [b] \text{erasetable}] \text{erasetable}] T_1 T_1 T_0 \]  
(9)

\[ \Rightarrow [0, 1, 101 [c] \text{erasetable}] \text{erasetable}] T_2 T_1 T_0 \]  
(10)

\[ \Rightarrow [0, 1, 101 [1, 010, 0 \text{erasetable}] \text{erasetable}] T_2 T_1 T_0 \]  
(11)

\[ \Rightarrow [0, 1, 101 [1, 010, 0] \]  
(12)

where:

\[ T_0 = e \]

\[ T_1 = (\cdot t \cdot) \rightarrow 101)(\cdot t \cdot) \rightarrow 1)(\cdot t \cdot) \rightarrow 0 \]

\[ T_3 = (\cdot n \cdot) \rightarrow 101)(\cdot n \cdot) \rightarrow 1)(\cdot n \cdot) \rightarrow 0 \]

\[ T_3 = (\cdot t \cdot) \rightarrow 010)(\cdot n \cdot) \rightarrow 101)(\cdot t \cdot) \rightarrow 1 \]

\[ (\cdot t \cdot) \rightarrow 0 \]

and where \langle\cdot t \cdot\rangle stands for \langle\cdot \text{local}\cdot\rangle and \langle\cdot n \cdot\rangle stands for \langle\cdot \text{nonlocal}\cdot\rangle.

Recognizers for table-languages

Much work has been done on the problem of designing syntax-directed recognizers\(^4\) for languages defined by BNF grammars. This section will indicate that existing syntax-directed techniques for BNF can be extended to table-languages.

Recognition-operators

A syntax-directed recognizer needs a means of carrying out each event in the recognition sequence which corresponds to its dual in the generation sequence. For a table grammar this requires that a recognition-operator be defined for each grammar-operator. While the same names will be used for both types of operators, the context will make clear which type of operator is intended. An additional recognition-operator, \text{copy}, is also provided to allow a limited form of deterministic recognition. The recognition-operators have access to a workspace which is placed on the top of the table-tape. The recognition-operators are defined as follows:

\text{copy}:  
This monadic operator has as its operand a terminal character from a set of check-out-characters. The set of check-out-characters is fixed for a particular grammar. \text{"copy c"} reads characters from the input and places them in the workspace until one of the check-out-characters is reached. The operation succeeds if this character is \"c\" otherwise it fails. If the operation succeeds, the control is passed along this path with the workspace initialized for use by a subsequent table-operation. If the operation fails the workspace is erased and recognition must proceed along an alternate path.

\text{create, retrieve, recreate}:  
These recognizer-operators utilize the workspace as their identifier input. If these operators succeed, they erase the workspace in addition to their usual side effects in the active-table. Since these three recognition-operators require that the identifier be already present in the workspace, they must be preceded by the execution of \text{copy} which initializes the workspace.

\text{illegal, replaces, copytable, erasetable}:  
These four recognition-operators are identical to their duals in a grammar. They neither utilize nor affect the contents of the workspace.

A summary of the definitions for the seven operators for a grammar and the eight operators for a recognizer are given in Appendices 1 and 2 respectively.

A recognizer for MAL programs

The table-grammar for MAL can be revised to place it in right-linear form since self-embedding recursion is not employed. In this form a recognizer or generator would not use the pushdown store and would only need the finite control and the table-tape. A deterministic recognizer for MAL programs is shown in Figure 2. A \text{copy} with reference to the respective delimiter has been inserted preceding each instance of \text{create, recreate and retrieve}. In two cases table-operations provide alternative paths from a node. However in each situation only one of the possible alternatives can succeed.
Figure 2—A deterministic table recognizer for MAL programs

Consider the three alternatives which occur following the path for goto:

1. `create (·improper label ·)`
2. `retrieve (·improper label ·)`
3. `retrieve (·imperative label ·)

If the identifier in the workspace is not in the table then (1) will succeed, and (2) and (3) will fail. If the identifier in the workspace is in the table, (2) will succeed if the selected table-category is `·improper label ·`, while (3) succeeds if the selected table-category is `·imperative label ·`. In each case there is only one path from the node which can succeed. If none succeed the device is blocked and the input-string is rejected.

Languages with self-embedding recursion cannot be placed in right-linear form and would require the use of the pushdown store as part of the recognition scheme. Also not all table-languages will be deterministically recognizable using the `copy` operator with its checkout-character set and workspace.

**Regular table expressions**

A regular table expression is an extension of those regular expressions where operators are limited to union, catenation and closure. Limited regular expressions, when embedded within a context-free grammar, have been shown to be useful both in the representation of the syntax of programming languages and in the construction of processors which automatically create efficient recognizers. This section will indicate that the use of regular expressions within a rule of a grammar can be extended to the domain of regular table expressions.

**Union, catenation, closure & distribution**

A regular table expression is defined recursively as follows:

1. Any element in the set `{a, ε, φ, create (·i·), retrieve (·r·), illegal (·r·), (·i·) recreates (·r·), (·i·) replaces (·r·)}` is a regular table expression.
2. If ρ is a regular table expression, then so are: `copytable ρ erasetable (ρ) *`.
3. If ρ₁ and ρ₂ are regular table expressions defined for the same set of identifiers and having the same table-categories, then so are: `ρ₁ρ₂` and `ρ₁ U ρ₂`.
4. No string is a regular table expression unless its being so follows from 1 to 3 above.

It is claimed without giving the proof that regular table expressions are closed under union, catenation and Kleene closure. However it is necessary to distinguish between the form of an expression and the set it defines. If L₁ is the language defined by ρ and Lₑ by ρₑ, then L₁ ∩ Lₑ = L₁Lₑ it does not follow that Lₑ = (ρₑ) *. This is due to the fact that various table-operators record information in the active-table so that generations to the right of a table-operator may not generate the same strings as if they were on its left. The definition of (ρₑ) * is `ρᵢ₀ ᵗ=₁ ρᵢ` where ρᵢ = ε and ρᵢᵗ⁺ = ρᵢ⁺.

The laws of distribution are obtained by definition. Let ρ, ρ₁, ..., ρₙ be regular table expressions, then:

`ρ(ρ₁ U ρ₂ U ... U ρₙ) = ρ₁ U ρ₂ U ... U ρₙ ,`

and

`(ρ₁ U ρ₂ U ... U ρₙ)ρ₀ = ρ₁ρ U ρ₂ρ U ... U ρₙρ .`

**Replacing recursion by the closure operator**

In clause 1 of the definition of a regular table expression (given above), if the domain of “‘a” is extended to include grammar-categories, then two important identi-
ties can be shown to preserve the generative power of table-grammars (the proof itself is not given).

Let $p_1$, $p_2$ be regular table expressions that have no instance of $(a)$ within them; further let "(" and ")" be grouping brackets for expressions and let them be disjoint from the set of terminals, then:

1. The rule $(a) ::= p_1 | (p_2)(a)$ can be written $(a) ::= (p_2)^* p_1$.
2. The rule $(a) ::= p_2 | (a)(p_2)$ can be written $(a) ::= p_1(p_2)^*$.

In both cases, $(a)$ is now defined without the use of explicit recursion. By the use of a uniform substitution for $(a)$, it can be eliminated from the set of rules of a table-grammar.

Extensions and limitations

The following extensions of table-grammars can be readily achieved within the framework given here. Some limitations are cited which lie outside the present model. Additional extensions which can overcome these limitations may be possible.

A partitioned identifier set

For some languages, the identifier set is partitioned by the definitions of the language. In ASA Fortran, if an identifier has no explicit declaration the type is implied by the rule: "names beginning with letters I to N are type integer, all others are type real." This rule partitions the undeclared identifiers into two disjoint subsets. To handle this case create would be replaced by two operators create-integer and create-real. For a particular grammar these new operators would be defined to select undeclared identifiers from their respective disjoint partition of the total set of identifiers.

Use of an implicit retrieve operator

In a very large grammar, it may be desirable to omit the operator retrieve and let the table-category itself, appearing alone in the grammar, be in effect an implicit call, in which the effective presence of retrieve is to be understood. The generative power of the notation with retrieve omitted would be identical to that where retrieve is explicitly used.

Limitations

A table-grammar, as defined here, is able to specify the declarations of scalar variables and labels. It is unable to specify all the restraints required for subscripted variables and for procedure calls when argument lists are used. In addition, table-grammars are limited to declarations where the identifier set is static. The IMPLICIT statement of Fortran sets up dynamic partitions of the identifier set. Even the extension of create given above is able to handle only static partitions of the identifier set.

ACKNOWLEDGMENTS

The author wishes to thank R. A. Stone of Western Electric, Princeton, C. B. Jones of I.B.M. Vienna, C. N. Moores of Rockford Research, Cambridge, Mass., and M. A. Harrison of the University of California, Berkeley, each of whom contributed constructive suggestions which have been included in the paper.

REFERENCES

1. P Naur
   Documentation problems: ALGOL 60
   Communications of the ACM Vol 6 No 3 March 1963 p 78
2. P Naur
   Revised report on the algorithmic language ALGOL 60
   Communications of the ACM Vol 6 No 1 January 1963 pp 1-17
3. N Chomsky
   On certain formal properties of grammars
   Information and Control Vol 2 1959 pp 137-167
4. S Ginsburg
   The mathematical theory of context-free languages
   McGraw-Hill 1966
5. R W Floyd
   On the nonexistence of a phrase structure grammar for ALGOL-60
   Communications of the ACM Vol 5 No 9 September 1962 pp 483-484
6. K Iverson
   A method of syntax specification
   Communications of the ACM Vol 7 No 10 October 1964 pp 588-589
7. J W Carr J Weiland
   A nonrecursive method of syntax specification
   Communications of the ACM Vol 9 No 4 April 1966 pp 267-269
8. F G Duncan
   Notational abbreviations applied to the syntax of Algol
   SICPLAN Notices ACM Vol 2 No 11 November 1967 pp 28-43
9. K Bandat
   On the formal definition of PL/I
10. S Ginsburg S A Greibach M A Harrison
    Stack automata and compiling
    Journal of the ACM Vol 14 No 1 January 1967 pp 172-201
11. G Whitney
    The generation and recognition properties of table languages
    IFIP Congress Software I August 1968 pp B18-B22
12. G Whitney
    The position of table languages within the hierarchy of nondeterministic on-line tape bounded during machine languages
    IEEE Conference Record Ninth Annual Symposium on Switching and Automata Theory October 1968 pp 120-130
### APPENDIX I

<table>
<thead>
<tr>
<th>Table-expression</th>
<th>Conditions</th>
<th>Value-returned</th>
<th>Side-effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create (·i·)</code></td>
<td>Selected-identifier must be undeclared</td>
<td>Selected-identifier</td>
<td>Add to the active-table the rule: <code>(·i·) → selected-identifier</code></td>
</tr>
<tr>
<td><code>retrieve (·r·)</code></td>
<td>There must exist within the active-table a rule: <code>(·r·) → identifier</code></td>
<td>Selected-identifier</td>
<td>None</td>
</tr>
<tr>
<td><code>(·i·) recreates (·r·)</code></td>
<td>There must exist within the active-table a rule: <code>(·r·) → identifier</code></td>
<td>Selected-identifier</td>
<td>The selected rule is changed to: <code>(·i·) → identifier</code></td>
</tr>
<tr>
<td><code>(·i·) replaces (·r·)</code></td>
<td>None</td>
<td><code>ε</code></td>
<td>Within the active-table each <code>(·r·)</code> is replaced by <code>(·i·)</code></td>
</tr>
<tr>
<td><code>illegal (·r·)</code></td>
<td>The operation is blocked if <code>(·r·)</code> is present in the active-table</td>
<td><code>ε</code></td>
<td>None</td>
</tr>
<tr>
<td><code>copytable</code></td>
<td>None</td>
<td><code>ε</code></td>
<td>A duplicate copy of the active-table is placed on the top of the table-tape</td>
</tr>
<tr>
<td><code>erasetable</code></td>
<td>None</td>
<td><code>ε</code></td>
<td>The active-table is erased</td>
</tr>
</tbody>
</table>

Note: `(·i·)` stands for an incident table-category and `(·r·)` for a resident table-category. `ε` is the empty-string.

**SUMMARY OF DEFINITIONS FOR GENERATION-OPERATORS**
## APPENDIX II

<table>
<thead>
<tr>
<th>Table-expression</th>
<th>Conditions</th>
<th>Side-effects if operation succeeds:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>In active-table</td>
</tr>
<tr>
<td>copy c</td>
<td>Succeeds if the next check-out character is c</td>
<td>None</td>
</tr>
<tr>
<td>create (\cdot i\cdot)</td>
<td>Identifier in the workspace must be undeclared</td>
<td>Add the rule: (\cdot i\cdot) \rightarrow identifier</td>
</tr>
<tr>
<td>retrieve (\cdot r\cdot)</td>
<td>There must exist within the active-table a rule: (\cdot r\cdot) \rightarrow identifier such that identifier = contents of workspace</td>
<td>None</td>
</tr>
<tr>
<td>(\cdot i\cdot) recreates (\cdot r\cdot)</td>
<td>Same as for retrieve (\cdot r\cdot)</td>
<td>The selected rule is changed to: (\cdot i\cdot) \rightarrow identifier</td>
</tr>
</tbody>
</table>

Note: a. the execution of copy must precede evaluation of expressions containing create, retrieve, or recreate.  
b. the definitions for illegal, replaces, copytable and erasetable are the same as given in Appendix I.

### SUMMARY OF DEFINITIONS FOR RECOGNITION-OPERATORS

From the collection of the Computer History Museum (www.computerhistory.org)