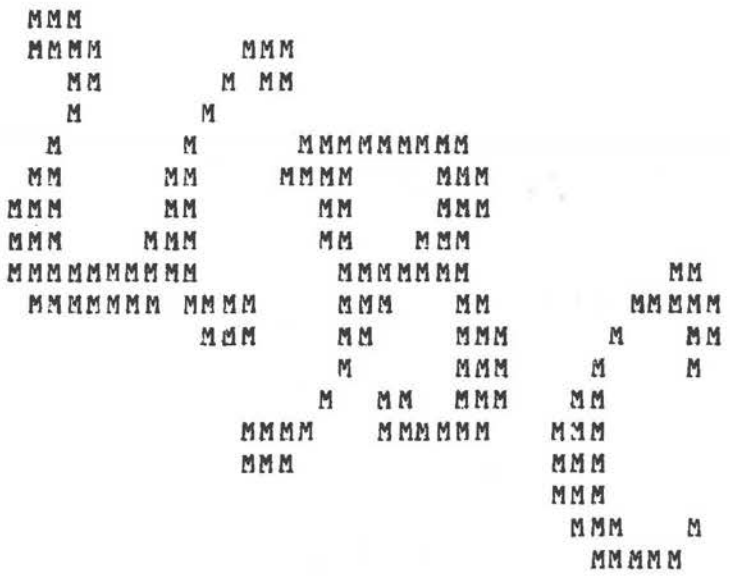

Unlanguage Grammars and Their Uses

by

R. A. Fraley

Technical Report 77-6

August 1977

Department of Computer Science
University of British Columbia
Vancouver, B. C.

Unlanguage Grammars and Their Uses

R. A. Fraley
University of British Columbia
August 17, 1977

ABSTRACT

A new technique is presented for using context free grammars for the definition of programming languages. Rather than accumulating a number of specialized statement formats, generalized productions specify the format of all statements. A sample unlanguage grammar is presented, and the use of this grammar is described. Some of the difficulties in parsing the language are described.

Unlanguage Grammars and Their Uses

This paper does not define another class of grammars or parsing algorithm. It describes a new way of using grammars for the definition of programming languages. A favorite parsing algorithm, or even an ad hoc technique, may be used to process an Unlanguage grammar.

Most programming language definitions are modelled after the Algol 60 report [6]. They present a grammar which defines all statements of the language. After careful consideration, the language designers have defined a language, written a grammar describing that language, and possibly produced a parser for the language using the grammar and a suitable parser generator. Unlanguages work the other way around: after the grammar has been specified and the parser built, the language is defined.

Lisp [5] resembles an Unlanguage. Here is a simplified grammar for Lisp:

```
<element> ::= <atom>
            |  (  <list>  )

<list>    ::= <EMPTY>
            |  <list>  <element>

<atom>    ::= <word>  |  <number>
```

This grammar describes the complete (for our purposes) grammar of Lisp. But it doesn't give any notion about the functions available in Lisp; it deals strictly with the language structure. All functions are defined elsewhere. By contrast, Algol 60's grammar contains the names of all legal statements.

The grammar for Lisp contains two undefined constructs <word> and <number>. While we could provide syntactic descriptions of these notions, compilers typically recognize these using a scanner rather than a parser. We may consider these to be "class terminals", because they specify classes of symbols rather than specific character strings.

The Lisp grammar is not a "pure" unlanguage grammar, because it contains two absolute terminal characters, "(" and ")". The only terminals in a pure Unlanguage grammar are class terminals. This is the origin of its name: the grammar defines only a structure, but doesn't specify actual statements or other language elements. An Unlanguage grammar defines pure form with no content.

Before looking at more examples of Unlanguage grammars, let's consider class terminals in more detail. The grammar places no restriction on the symbols which belong to different classes. Two classes (such as <identifier> and <array identifier> of the Algol 60 report [6]) may contain tokens which are syntactically indistinguishable. Other classes (such as <relational operator>) can contain tokens which have a radically different syntactic structure (such as "<=" and "LE"). In the discussion which follows, we shall never attach any syntactic significance to the class terminals.

The First Unlanguage

The First Unlanguage (FUNL) grammar describes the structure of many programming languages:

```
<program>          ::=  <expr>  <expr sequence>

<expr>             ::=  <function>
                      |  <expr>  <operator>

<expr sequence>    ::=  <EMPTY>
                      |  <expr sequence>  <expr>

<expr option>      ::=  <EMPTY>
                      |  <expr>

<function>         ::=  <fn item>  <phrase sequence>

<fn item>          ::=  <FN1>
                      |  <FN2>  <expr>
                      |  <FN3>  <expr option>
                      |  <FN4>  <expr sequence>

<operator>         ::=  <op item>  <phrase sequence>

<op item>          ::=  <OP1>
                      |  <OP2>  <expr>
                      |  <OP3>  <expr option>
                      |  <OP4>  <expr sequence>

<phrase sequence>  ::=  <EMPTY>
                      |  <phrase sequence>  <phrase>

<phrase>           ::=  <PH1>
                      |  <PH2>  <expr>
                      |  <PH3>  <expr option>
                      |  <PH4>  <expr sequence>
```

The symbols <PHi>, <FNi>, and <OPi> are class terminals. Methods for dealing with ambiguities in this grammar shall be discussed in the section "Disambiguation". The appendix

describes some extensions which might be made to this grammar.

It is not obvious how common structures of standard programming languages are represented by the FUNL grammar. Here are some examples:

Variables
A simple (unsubscripted) variable lies in the <FN1> class.

Constants
Numeric and character constants are also in <FN1>.

Infix Operators
An infix operator (such as "/") is in <OP2>.

Example

Suppose that "A", "B", and "C" are variables, and that "+", "-", "*", and "/" are infix operators. The sentence "A * B + C" will be parsed as follows:

Parse Stack	Explanation
A	From input
<FN1>	By definition of variable
<fn item>	
<fn item> *	Input
<fn item> <OP2>	Definition of infix operator
<function> <OP2>	
<expr> <OP2>	
<expr> <OP2> B	Input
<expr> <OP2> <FN1>	Definition of variable
<expr> <OP2> <FN1> +	Input
<expr> <OP2> <FN1> <OP2>	Infix operator
. . .	
<expr> <OP2> <expr> <OP2>	Several reduction steps
<expr> <op item> <OP2>	Disambiguation (precedence)
<expr> <operator> <OP2>	(Null <phrase sequence>)
<expr> <OP2>	
<expr> <OP2> C	Input
. . .	
<expr> <OP2> <expr>	Several reduction steps
<expr> <op item>	
<expr> <operator>	
<expr>	
<expr> <expr sequence>	
<program>	

Notice that operators are combined with their right operand before being joined with the left. This is purely a function of the context free representation. The parsing mechanism is

isolated from the rest of the system: semantic processing occurs only after the completion of <expr> and <program> productions. Such isolation is important from a structured programming standpoint. (Semantic action can occur at each phrase word, but this action is usually reserved for changing name scopes. A 1-pass compiler could generate code at these points.)

One might think that a general grammar could say absolutely nothing. This is not the case. While the above strings were accepted as input, the string below is illegal:

A + / * B C

Prefix Operators

A prefix operator is in the terminal class <FN2>.

It may seem strange to refer to a prefix operator as a function. But prefix operators have the same format as functions, except for parentheses: there is a function name followed by a parameter. The lexical form of the name is irrelevant in Unlanguage parsing.

Grouping Parentheses

"(" lies in <FN2>. It must be followed by a phrase ")" having structure <PH1>.

This is our first use of the syntactic structure <phrase sequence>. Operators and functions need not consist of a single word; they may include regular expressions of phrases. Regular expressions, together with the recursion provided within the <expr> structure, are adequate for defining most computer languages. This should not be surprising, because regular expressions are used for describing statements in many language manuals. There will be no attempt to prove that all computer languages can be described in this manner, but a survey of a dozen languages, performed by the author, has shown that most language constructs can be handled by FUNL. (FUNL is intended to be a language design tool, so some features of some languages might not fit its structure.)

Example

Let "-" be a prefix operator. We shall show the parse of the sentence:

(- B + C)

Parse Stack	Explanation
(	Input
<FN2>	Definition
<FN2> -	Input
<FN2> <FN2>	Definition
<FN2> <FN2> B	Input
<FN2> <FN2> <FN1> +	Variable def and input
. . .	
<FN2> <FN2> <expr> <OP2>	Several reduction steps
<FN2> <fn item> <OP2>	Disambiguation (precedence)
. . .	
<FN2> <expr> <OP2>	Several reduction steps
<FN2> <expr> <OP2> C	Input
<FN2> <expr> <OP2> <FN1>	Definition
<FN2> <expr> <OP2> <FN1>)	Input
<FN2> <expr> <OP2> <FN1> <PH1>	Definition
<FN2> <expr> <op item> <PH1>	Disambiguation (precedence)
<FN2> <expr> <operator> <PH1>	
. . .	
<FN2> <expr> <phrase>	Several reductions
<fn item> <phrase>	
<fn item> <phrase sequence>	
<function>	
. . .	
<program>	Several reductions

Statements

The regular expression below defines an IF statement:

IF # THEN # [ELIF # THEN #]* [ELSE #] FI

The underlined words are <PH2> tokens if they are followed by "#" and are <PH1> tokens otherwise. An exception to this is IF, which is a <FN2> token. Square brackets represent optional phrases or phrase sequences, while "*" indicates zero or more repetitions. This definition language shall be used throughout this paper. (With FUNL, a number of definition languages can be invented for describing statement formats.)

Most statements of most languages can be easily defined using the regular expression notation. There are a number of minor problems, especially when optional separators appear in the statement or when expressions may be written without delimiters. The CASE statement of Pascal [4] is one of the least natural for the regular phrase expression notation.

CASE # OF # [, #]* : #
[; # [, #]* : #]* END

The notation is unnatural for a number of reasons. Expressions

(indicated by "#") are associated with "OF" and ";" when they should really be associated with the phrase ":" which follows. A more serious problem is that extra ";"s, included in the Pascal syntax, cannot be provided for in the regular notation. Consider, for example, the optional ";" which can be placed before the END. This ";" is not followed by an expression, so it must be a <PH1> token. However, in a left to right scan we might encounter the ";" which starts the optional repetition of CASE value statements. This ";" is a <PH2> token. Inclusion of the optional ";" would be ambiguous, so it is not allowed. On close examination, the reader will find other faults in this implementation of the CASE statement.

There are a number of ways we could improve the CASE statement representation. New productions could be added to <phrase> to implement new statement formats, or a different characterization of statements might be found.

A second alternative is to avoid describing the complete CASE statement in a single regular expression. We may use the representation below:

$$\underline{\text{CASE}}\ \#\ \ \underline{\text{OF}}\ [\#]\ \ [;\ [\#]]^*\ \ \underline{\text{END}}$$

The bracketed "#" indicates that the preceding phrase ("OF" or ";") belongs to the token class <PH3>. Tokens of this class are optionally followed by an expression. Between the words "OF" and "END" we define infix operators ",", and ":" (in order of decreasing precedence). The "," operator is used to form value lists, while ":" attaches a value list to a statement. Operators may easily be defined (or redefined) over a limited program segment due to the flexible symbol table structure of FUNL. This structure is described in a later section.

Subscripted Variables

Before investigating the inner structure of the FUNL compiler, let's examine one more source construct--the subscripted variable. One implementation of subscripted variables gives them a complex syntactic structure:

$$\underline{\text{W}}\ [\lfloor\ \#\ [\ ,\ \#\ ]^*\ \rfloor\ ]$$

The subscript portion of the variable is optional, so that we may refer to an entire array if desired. We could share this definition among all array variables.

An alternate implementation defines the <operator> below:

$$\lfloor\ \#\ [\ ,\ \#\ ]^*\ \rfloor$$

(In this case, the first symbol is a terminal in <OP2>.) A subscript is defined as a postfix operator. (This postfix

operator contains some expressions. APL has several operators with internal expressions.) Of the two techniques, this is superior because it allows the subscript to be applied to array expressions and constants as well as variables.

The Symbol Table

A description of the FUNL symbol table may seem inappropriate for a discussion of Unlanguage syntax, but it plays a part in parsing. It serves the obvious goal of defining the terminal symbol classes of tokens, and also serves as a finite-state machine for the parser.

The symbol table structure facilitates the definition of block structured languages. There are data objects called blocks, and each symbol is defined within a specified block. Normally a symbol has only one definition within a block.

A scope is a list of blocks. A symbol may have many definitions within a scope, but the definition in one block has priority over definitions in later blocks. Scopes resemble the nest of blocks within a block-structured language. Like some of the recent modular languages, there may be many scopes in existence at a given time. A block may belong to any number of scopes.

For efficient processing, the implementation reverses this arrangement. All of the symbols are placed in a single table to facilitate a rapid search algorithm. Each symbol entry yields a chain (or tree) of definitions, and each definition is distinguished by a block identifier. After finding a symbol, we search for a definition whose block lies in the current name scope. The name scope is a list of block identifiers. With this organization, we can also access all definitions of a symbol when errors occur.

A symbol may have two definitions within a given block. One definition must be a <function>, while the other is an <operator>. The selection between these definitions is discussed under the section "Disambiguation".

Sequencing Through Regular Expressions

While parsing, there are always two name scopes which are active. One of these is the traditional environment, which defines the current functions and operators. The second contains those phrases which may currently be recognized. Each time a new function is entered or a new phrase found, this scope changes.

Let's consider the "IF" statement defined earlier. After finding "IF", we process an expression until "THEN" is found. While the expression is being parsed, the phrase scope contains a single block, and that block contains only the symbol "THEN". After "THEN" has been found in the input, the phrase scope changes. This new scope contains two blocks. The first contains the words "ELIF" and ELSE", while the second contains "FI". Any of these three words may now appear as a phrase. If the word "ELSE" appears, the phrase scope changes again. This scope contains only the "FI" block. After "FI" is found, the phrase scope for any containing statement is restored.

This process may seem time consuming, but it is equivalent to a finite state machine. The blocks are the states of the machine, and scopes with multiple blocks correspond to null transitions. A change in scope is implemented by changing a single pointer variable. We need no specialized routines for transition and state selection.

Disambiguation

There are many ambiguities in the mechanism presented above. This section examines these ambiguities and explains techniques for resolving them.

Syntactic Ambiguity

The most obvious ambiguities lie in the FUNL grammar. A construct

<expr> <OP2> <expr> <OP2> <expr>
or
A + B + C

could be parsed in two distinct ways. We use the traditional notion of precedence to resolve the ambiguity. In our formulation, each operator has two precedence levels: one as seen from the left, the other as seen from the right. For standard operators these levels are the same. A right associative operator has its right level slightly lower than its left.

In FUNL, functions and phrases need levels as well. Consider this segment of code:

if B then A := C; C := D

This code could have been extracted from either an Algol 60 program or an Algol 68 program [7]. In Algol 60 the ";" terminates the if statement, so that "C := D" is always performed. In Algol 68, the if statement must be terminated by

fi. Therefore the second assignment must still be a part of the if.

Let us assume that ";" is defined as an operator. A fairly natural extension may be made to the concept of operator precedence: we can define "operand" precedence. For both types of if the ";" has a lower precedence than ":=", so the assignment statement is properly grouped together. For Algol 60, the precedence of ";" is also lower than that of then. The entire if statement becomes the left operand of ";". For the Algol 68 if statement, the precedence of ";" is higher than the precedence of "then", so it binds together the two assignment statements within the then portion of the if statement.

Parentheses illustrate a situation where the left and right precedence of a token are quite different. From the inside, the parentheses have a low precedence, permitting operators of higher precedence to reside within. But from the outside, the parentheses must remain as a unit, so they have a very high precedence. This allows them to be operands for operators of lower precedence.

Symbolic Ambiguity

Symbolic ambiguity is more difficult to deal with than syntactic ambiguity was. Symbolic ambiguities arise when a word has several definitions. One of these must be chosen for the current instance of the word. At times, name scoping is inadequate for making this selection.

We begin by making the simplifying assumptions that:

-- No tokens are in the classes <OP3>, <OP4>, <FN3>, etc.

-- The construct <expr sequence> always matches the empty string.

As a result of these changes, FUNL parses only operator precedence languages. This is the "safe" subset of FUNL.

Two major ambiguities arise in this subset. One of these is the selection between an operator and a function defined in a single block; the second is the choice between a phrase and an operator having the same name.

The beauty of the subset language is that we always know whether we need a function or an operator. Grammar analysis shows that <FNi> and <OPi> never occur in the same state. When there are functions and operators of the same name (such as "-" and "(") which lie in the same state, we interrogate the grammar state to find out which one can be used. The appropriate definition is selected.

Phrase names and operators, on the other hand, can occur in the same state. If a given word has active definitions as both an operator and a phrase, FUNL always chooses the phrase definition. One way that we may justify this approach is to regard the statement as implicitly declaring its phrase words in a new inner block. The phrase word hides any previous definitions. Another view is that the phrase words should have been reserved. The ability to use phrase words as identifiers is strictly a convenience; it is the user's responsibility to see that no conflict occurs.

The FUNL system includes two capabilities for preventing user confusion which might result from the arbitrary preference towards phrases. The first is a standard reserved word technique. This could be applied to operators only, or to functions and operators. This is viewed as an undesirable solution, because a new language module needed for one portion of a program might invalidate the variable names used in another portion. The second choice is to check each usage of a word for ambiguity, and only flag those places where ambiguity has actually occurred. This method requires a little more time during symbol lookup, but seems more realistic than the traditional reserved word technique.

Consider the complete FUNL grammar. The concepts <expr sequence> and <expr option> were added to the basic grammar to provide additional flexibility. Along with this flexibility comes additional ambiguity. Can we justify this loss of safety? We think so. A number of languages, such as SIMSCRIPT II.5 [2], COBOL [3], and LISP [5] use lists without separators and occasionally have optional expressions. They also make sure that the use of such constructs is unambiguous. (Infix operators are not allowed if there is an identical prefix operator.) Including the constructs in FUNL enables us to investigate further the requirements for safe use of the constructs. They may be excluded at the definition language level when safety is desired.

The full FUNL grammar has places where a function, operator, or phrase might be legal. We need to select one definition of the input word. There may also be several choices as to where the resulting expression should be used in the grammar. The general principle used in resolving these ambiguities is: use the closest location at hand. This principle shall be illustrated below.

Some language might like to allow lists (such as parameter lists) in which the separating character (such as ",") is optional. This could be accomplished with the <expr sequence> construct.

```
(A      -B      (I+J)*C)
```

This example shows how such a list might appear in a general

purpose language. Because such a language does not have the syntactic safeguards, we might interpret the above list as:

(A-B(I+J)*C)

Because our language processing ability does not currently allow subtle spacing information to take part in parsing, we must live with separator characters, rigid spacing rules, restricted operator ability, or potential ambiguity.

The choice of closest meaning we interpret, in this case, to require that an infix operator in the current expression be preferred to beginning a new expression in the list. For the example above, the net result of this decision would be the single expression interpretation instead of the list. For those languages such as COBOL or SIMSCRIPT II.5 which use such lists, infix operators could be removed from the current scope to permit the list interpretation. If the subscript operator remains, it will automatically be chosen over the use of parentheses for grouping.

Earlier we saw that phrase definitions were preferred to operator definitions. Likewise, they are preferred to function definitions when an expression is optional. Suppose that "ELSE" is a variable name in a language having an ALGOL 60 "IF" statement. Compare the statements below:

```
IF B THEN ELSE A := C

IF B THEN ELSE := C
```

Because FUNL decides upon definitions without the use of look-ahead, we must decide upon the definition of "ELSE" with identical information. Our preference for phrases makes us accept the first and reject the second.

Another example arises from the interaction between expression options and lists. Suppose that "!" is a postfix operator which may optionally take a second parameter. If we use it in a list which has no separators, we find that

```
( A ! B )
```

has two possible interpretations: the B can go with "!" or can become a separate expression.

One situation avoids the simplistic disambiguation which we have used so far. Consider the Algol 68 [7] expression

```
begin a := b ; - c end
```

As before, we regard ";" as having an optional second parameter. The trained human eye will immediately see that "- c" is the second operand of ";". But the naive machine might be tempted

to say that ";" has only a left operand, and "a := b ;" is the left operand of the infix operator "-". We could use our rule of "closeness" to choose the prefix definition of "-", and we would make the proper choice in this case. But suppose that "-" has a lower precedence than "!". Shouldn't the interpretation of "B! - C" be (B!)-C? Our rule, on the other hand, would produce "B!(-C)". This latter interpretation could be illegal in some implementations. It suggests that precedence is an important consideration for disambiguation.

A precedence check would ascertain that the selected operator or function has precedence levels compatible with its environment. If the usual choice is not compatible, the alternate is used.

Environment Ambiguity

As mentioned earlier, the FUNL system has the capability of changing the current naming environment at any symbol of the input. Fortunately, the semantics routines can restrict the use of this facility. We still must be sure that changes in the name scoping produce no bad effects while parsing. So far, only one rather obscure difficulty has been discovered. Let us assume that our language has a structure resembling the class of Simula 67, but allowing the definition of arbitrary operators. Let us further assume that there is a statement resembling the Simula inspect statement, which opens the class name space on top of the current environment. The change in name scope is effected within a when phrase, which contains a single statement. The low-precedence operator ";" is normally used to terminate the inspect statement.

Suppose that in our class we define an operator "%" having the same precedence as ";". Within some inspect statement, we open up the environment of the class. On finding the operator "%", we must terminate the inspect statement (since "%" has the same precedence as ";"). But now that the inspect has been terminated, "%" is no longer defined. Should we issue an error message? How can we recover from such an error, since the inspect has already been terminated? If "%" was already defined in the outer environment, should we use this definition or the class definition? What if the outside operator has a precedence high enough that the inspect should not have been terminated? We avoid the problem by making the convention that a token's first definition is used, and we never look for another. While we could get some strange effects in this obscure case, a definition language could check for changes of scope in an open statement.

Human Ambiguity

The most interesting form of ambiguity is human ambiguity: language constructs which lead the programmer into writing syntactically correct programs for which his interpretation differs from that of the compiler. The classic statement of this sort is the Algol 60 if statement [1]. While the original syntax of this statement is ambiguous, an unambiguous syntax may be written (using constructs <closed statement> and <open statement>). The natural definition of the Algol 60 if statement in FUNL will associate else with the innermost if. But while the ambiguity can be eliminated technically, programmers may occasionally write an else clause for the outermost if without thinking about the nested if. Although the language is unambiguous, it has not provided the redundancy needed to detect the author's incorrect interpretation. FUNL has the capability of flagging ambiguous occurrences of the phrase words like else, but the language designer should be warned when specifying such a statement. The definition language may easily check for this situation.

Do we really need to warn the language designers? Hasn't the Algol 60 if problem been around long enough that the error is no longer made? The answer is "NO"! For example, the designers of Simula 67 [3] eliminated the if problem by forbidding the use of an if statement following then. But they went on to produce another statement -- the inspect statement -- which has identical problems. If the when clause of one inspect has another inspect contained within, additional when and a concluding otherwise are absorbed by this inner inspect.

Are there other constructs which are syntactically unambiguous but lead to human error? If so, then when they are discovered they may be incorporated into the definition languages without requiring a change to the Unlanguage system.

The Use of FUNL

The FUNL parser is intended to be part of a larger system -- the FUNL Unlanguage system. This system has mechanisms for defining the semantics as well as the syntax of a language. This semantics model is common to all FUNL implementations.

The Unlanguage system organization can be compared to micro-coded computers: the syntactic constructs (macro commands) must be implemented in terms of a given semantic model (the micro instructions). Through source code, the user can access only those semantic primitives which have been given a syntax by the language designer. A "GO TO" semantic primitive could be provided for implementation of control structures, but the language designer might not provide a direct syntactic access to this primitive.

Just as separately written library packages can be used by a single program, a language processor will consist of separately coded modules. The modules will generally contain a syntax definition and a semantics definition. The semantics will generally transform the source concepts into semantic primitives, though in some cases they may invoke routines from other modules. When several languages share a number of semantic notions, purely semantic modules may be generated to provide a higher-level model for these languages.

A language implemented under FUNL can be extended by building a new module which contains the extensions and including it with the old compiler. This is analogous to extending the instruction set of a computer by adding micro code. Unlike current macro or preprocessor systems, the extensions need not be defined in terms of existing language constructs. They are defined in terms of Unlanguage semantic model concepts. As these concepts are generally at a lower level than those of the source language, the implementation can often be more efficient. With the Unlanguage approach, a module of the existing language could be phased out by providing a replacement module. This module would need to provide the semantic procedures needed by other modules, but could have a reworked syntax.

Consider Fortran. It has been around for a long time. One feature which has aided Fortran's longevity is the library facility. By getting a library developed elsewhere, one is providing a semantic extension to Fortran. But syntactically Fortran is static. Even surface changes, such as replacing ".LT." by "<" are not possible. If some installation made such a change to its compiler, the user programs would no longer be transportable to other installations. Many people dislike Fortran because its syntactic structure is archaic. This structure cannot be easily changed. Syntactic extensions cannot be added using library packages. With languages implemented under FUNL, this will not be true. FUNL languages will be able to evolve. Learning a new language feature will be like learning a new library package. Programmers don't want to leave Fortran because it means learning a new language and destroying their old programs. With the FUNL system, a programmer would only need to learn a new module rather than a new language. His old programs are still useable as long as the old modules remain in the library.

Many of the claims being made for Unlanguages are identical to those made for UNCOL's. UNCOL's are impossible because they require universality. An Unlanguage makes no such claim: if your source language or target machine finds the semantic or syntactic model unusable, then invent another. The resulting Unlanguage must be implemented on various machines in order to allow transportability, much as a new language must be implemented now. If the new Unlanguage is a minor modification of the old, transporting source language to the new system

should be quite easy.

Proliferation of Unlanguages should be much less than the proliferation of languages is today. New languages are invented when the notation of an existing language is cumbersome for the problem at hand. But an Unlanguage will only change when the implementation of languages becomes too inefficient within its semantic model, or new hardware features cannot be accommodated. Language experts cannot rid themselves of Fortran, but an archaic Unlanguage can always be replaced by rewriting the definition modules to produce code within the new semantic model. Most users would not need to know of the change.

Proliferation of languages is another matter. Everyone can become a language writer. But this should pose no real problem. After the novelty wears off, certtain language modules will become standard. These will tend to be used by everyone. A language designer will tend to use existing modules for most of his language so that he is free to write his changes to other aspects of the old language. Extension modules for special applications should be no worse than a new library package is today, even though new syntax is introduced. (This assumes that the extension language imposes suitable structure on these extensions so that they bear some resemblance to the original language.)

Appendix: Extended FUNL grammar

The grammar described in this paper has a number of limitations. This appendix describes extensions which might be made to allow more efficient implementation of constructs found in current programming languages.

1.

COBOL has two basic forms of MOVE statements:

MOVE A TO B.

and

MOVE CORRESPONDING A TO B.

The MOVE statement may be expressed by the syntax:

MOVE [#] [CORRESPONDING #] TO #

This does not describe the COBOL syntax precisely, since it allows the statement:

MOVE A CORRESPONDING B TO C.

A semantic check can be used to detect this error.

The FUNL grammar can be augmented to make a precise definition of the MOVE statement syntax. A new lexical class <MOD> can be created, and a sequence of <MOD>s can be permitted after a phrase (or function) identifier and before its corresponding expression. The word "CORRESPONDING" is defined to be a <MOD>. The MOVE statement definition becomes:

MOVE CORRESPONDING # TO #

2.

A number of languages, such as COBOL and SIMSCRIPT, allow noise words to be inserted at arbitrary locations in a program. These words may usually be treated as operators, optional phrases, or modifiers. In certain cases, it is convenient to eliminate such words in the scanner or permit them explicitly in the grammar between any two existing constructs.

3.

The COBOL relational operators may be described most easily by creating the class <FUNCTOR>. A functor takes an operator as its parameter, and produces a new operator as the result. The word NOT is a functor: it operates on the "=" operator to produce a new operator which checks for inequality. "IS" is the identity functor. Without the <functor> concept, a description of relational operators requires substantially more space.

```
<operator>  ::=  <op item> <phrase sequence>
              |  <FUNCTOR1>  <operator>
              |  <OPi>  <FUNCTOR2>  <operator>      (i>1)
```

Bibliography

[1] Abrahams, P., "A Final Solution to the Dangling else of ALGOL 60 and Related Languages", CACM, 9:9, 679-682 (1966).

[2] CACI, Simscript II.5 Reference Handbook, CACI, Los Angeles, 1973.

[3] IBM, DOS Full American National Standard COBOL, IBM Corporation, GC28-6394-4, 1973.

[4] Jensen, K. and N. Wirth, Pascal User Manual and Report, Springer-Verlag, New York, 1976.

[5] McCarthy, J. et al., LISP 1.5 Programmers Manual, 2nd edition, MIT Press, Cambridge, Mass., 1965.

[6] Naur, P. ed., "Revised Report on the Algorithmic Language Algol 60", CACM, 6:1 (1973).

[7] van Wijngaarden, A. et al., Revised Report on the Algorithmic Language Algol 68, Springer-Verlag, New York, 1976.