LINGO, A Dynamic Syntax Directed Compiler

by

Edward A. Burfine

Computer Science 239, Stanford University

Abstract

LINGO is a table driven syntax directed compiler which translates sentences in a simple precedence phrase structure programming language to symbolic code. The tables which direct translation from language text to symbolic code can be modified during translation. Table modification is directed by meta-language text. Many blocks of language text, where each block is composed of sentences of a different simple precedence phrase structure programming language, can be translated at one time. This is done by changing tables between the translation of blocks of language text.
Introduction

All previous syntax directed compilers translate language text in a programming language to symbolic code or machine code. The programming language may change for each translation, but it is fixed during the translation.

LINGO allows the programming language to change during the translation. This is done by changing the tables which control language text to symbolic code translation. The number of changes and the position in the language text where the changes occur are left to the programmer. With a fixed environment, i.e., the memory of the digital computer, these tables define a simple precedence phrase structure programming language. A change in these tables changes the programming language.

Changes in the tables may 1) extend the programming language to include new data spaces or grammatical constructs, 2) remove the definition of certain data spaces or grammatical constructs from the programming language, or 3) reinstate a previously defined programming language.

A programmer using LINGO defines the language he wants to use by meta-language text. He then writes sentences in that programming language until a modification of the programming language is necessary for efficient programming. He then modifies the definition of the programming language using meta-language text, and proceeds to write sentences in the new programming language. This modification process can be repeated as often as necessary.
In some ways this work parallels the work of Wijngaarden and deBakker.\textsuperscript{1,2,3} It differs in three major respects, however, which are:

1) A more general method of defining languages by syntactic rules and semantic interpretation rules is presented. 2) The meta-language in LINGO is a simple precedence phrase structure programming language which is parsed in the same way as the programming languages. The semantic interpretation rules are fixed and represent operations on the tables which drive the compiler. 3) The compiler is as efficient as the programmer makes it. There is no pyramid of definitions, as in the method of recursive definitions of Wijngaarden\textsuperscript{1} and deBakker\textsuperscript{3}. 
Phrase Structure Programming Languages

Using the notation of Wirth and Weber\(^4\), a phrase structure programming language \( \mathcal{L}_p(\mathcal{V}, \mathcal{F}, \mathbb{B}, A, \Psi, \mathcal{E}) \) is completely specified by

1) \( \mathcal{V} \) - the vocabulary,
2) \( \mathcal{F} \) - the finite set of syntactic rules,
3) \( \mathbb{B} \) - the set of terminal symbols,
4) \( A \) - the initial non-terminal symbol,
5) \( \Psi \) - the set of interpretation rules, and
6) \( \mathcal{E} \) - the environment for elements of \( \mathcal{L} \).

The environment \( \mathcal{E} \) for language text in LINGO is fixed and is the memory of the digital computer. The initial non-terminal symbol \( A \) for the language is fixed at \( \langle \text{PROGRAM} \rangle \).

In a context free grammar, the vocabulary \( \mathcal{V} \) and the set of terminal symbols \( \mathbb{B} \) may be obtained from \( \mathcal{F} \). Thus, with \( \mathcal{E} \) and \( A \) fixed, \( \mathcal{L}_p \) is defined by \( \mathcal{F} \) and \( \Psi \).

\( \mathcal{F} \) may be thought of as a set of syntactic rules in Backus Normal Form and \( \Psi \) as a set containing groups of assembly code. Let \( T \) be a translation which adds, removes, or changes rules in \( \mathcal{F} \) and adds, removes or changes the corresponding assembly code in \( \Psi \).

If \( T \mathcal{F}_0 = \mathcal{F}_1 \) and \( T \Psi_0 = \Psi_1 \)

then \( \mathcal{L}_p(\mathcal{V}_1, \mathcal{F}_1, \mathbb{B}_1, A, \Psi_1, \mathcal{E}) = \mathcal{L}_p(\mathcal{V}(T \mathcal{F}_0), T \mathcal{F}_0, \mathbb{B}(T \mathcal{F}_0), A, T \Psi_0, \mathcal{E}) \),

where \( \mathcal{V}(T \mathcal{F}_0) = \mathcal{V}_1 \) and \( \mathbb{B}(T \mathcal{F}_0) = \mathbb{B}_1 \).

Thus, a new phrase structure programming language may be formed from
the old one by performing a transformation on $\mathcal{L}$ and $\psi$. To maintain a precedence phrase structure programming language, the transformation $T$ must be restricted such that the resulting language $\mathcal{L}_f$ is a precedence phrase structure programming language. If this restriction is violated, LINGO will terminate translation and give an error message.

The transformation $T$ is represented in LINGO by meta-language text.

Translators

LINGO is composed of two translators.

The first translator is used for the translation of language text in a particular programming language to symbolic code. It is driven by a set of tables which can be modified. This translator is written as an ALGOL procedure called COMPILER.

The second translator accepts text which directs the modification of the tables in COMPILER. It is also driven by tables. In this case the tables are fixed and define the meta-language when the environment is the tables in COMPILER. This translator is written as an ALGOL procedure called METACOMPILER.

Some of the tables in COMPILER are modified directly by METACOMPILER, while others are recalculated using an ALGOL procedure SYNTAXPROCESSOR and the algorithm of Floyd. Parsing is directed by the precedence functions of Floyd and is accomplished by the algorithm of Wirth and Weber.
Every table in METACOMPILER is fixed. In COMPILER, every table is treated as data and can be changed during translation. There are six types of tables in LINGO. They are:

1) A symbol table containing the vocabulary $\mathcal{V}$. This table is subdivided into $\mathcal{B}$ and $\mathcal{V} - \mathcal{B}$.
2) A table of syntactic rules (in numeric form).
3) A master table which relates symbols in the symbol table to syntactic rules.
4) The two precedence functions $F$ and $G$ of Floyd.
5) A table containing syntactic rules of $\Phi$ in Backus Normal Form.
6) A table containing the corresponding semantic interpretation rules of $\Psi$ such that there is a one-to-one correspondence between semantic interpretation rules and syntactic rules.

Tables of type 1) through 4) are found in both COMPILER and META-COMPILER. Tables of type 5) and 6) are only found in COMPILER.

Table Modification

Meta-language text translated by METACOMPILER directs modifications in tables of types 5) and 6) by adding, changing, defining, or deleting syntactic rules and their associated semantic interpretation rules. Thus the meta-language is itself a simple precedence phrase structure programming language where the semantic interpretation rule for each syntactic unit is an operation on tables of type 5) and 6) or a signal to SYNTAX-PROCESSOR to form tables of types 1) through 4).

Control of the Source Code

The Source Code is composed of both blocks of language text and blocks of meta-language text. At the beginning of the translation, the text is assumed to be language text and is processed by COMPILER.

When the word delimiter SAVE is encountered the tables in COMPILER and the local pushdown stack (state of COMPILER) are saved in a pushdown store and control is passed to METACOMPILER. In Fig. 1 this may be
thought of as placing the switch in the down position. This is implemented as a procedure call of METACOMPILER by COMPILER.

METACOMPILER processes one block of meta-language text. This block must be syntactically correct. When parsing of the block is complete, a procedure call of COMPILER by METACOMPILER is executed. In Fig. 1 this is represented by placing the switch in the up position. This recursion allows LINGO to move up and down a tree of languages in a recursive manner. When COMPILER is called, language text in the new language is translated.

A return through the recursion is performed when the word delimiter RETURN is encountered in COMPILER. At this time the language text in the present language is parsed to completion. If no syntax error occurs, the tables and local pushdown stack which define the previous language and the previous state of COMPILER are restored. A procedure return from COMPILER through METACOMPILER to COMPILER is then executed.

The Meta-language

The meta-language is based on a structure composed of a single block. A block is composed of sentences of the form

\[ \text{operator} \quad \text{syntax} \quad \text{IS} \quad \text{semantic interpretation rule} ; \]

The block may contain as many sentences as necessary to define the new language. The initial language in LINGO is quite simple and is defined only to give the programmer a language to operate on with meta-language text.
It is

\[
\text{<program>} ::= \text{<block>}
\]

\[
\text{<block>} ::= \text{<block body>}
\]

\[
\text{<block body>} ::= \text{<identifier>}
\]

In a production compiler, the initial language would be an ALGOL-like language.

There are four meta-language operators. They are

1) **DEFINE**
   - define a new metalinguistic variable with a metalinguistic equation and a semantic interpretation rule,

2) **APPEND**
   - add to the definition of a previously defined metalinguistic variable by appending the right-hand side of the given metalinguistic equation to a previous equation and appending the given semantic interpretation rule,

3) **CHANGE**
   - delete all previous definitions of the metalinguistic variable and define it as in 1),

4) **DELETE**
   - delete the definition of the metalinguistic variable from the grammar.

Syntax is defined in a modified Backus Normal Form. Word delimiters and delimiters are enclosed in quotations to differentiate them from metalinguistic delimiters.

Semantic interpretation rules are represented by machine language code. The symbol *, used as an address in the symbolic code, represents the identifier which was last seen by the parsing procedure. Identifiers are stored
by the parsing algorithm in a pushdown stack and removed as used.

Several examples of sentences in the meta-language will help to clarify this discussion. Consider the sentences

\begin{verbatim}
DEFINE <statement> ::= "IDENT". "=". <leftpart> ;" IS STO*;
DEFINE <leftpart> ::= <expression> IS EMPTY;
\end{verbatim}

The equivalent Backus Normal Form for the syntactic rules is

\begin{verbatim}
<statement> ::= <identifier> ::= <leftpart> ;
<leftpart> ::= <expression>
\end{verbatim}

The semantic interpretation rules are

STO * : Store the value of the stack in the location specified by the last identifier seen in the parsing algorithm,

EMPTY: Null operation.

An Example of Language Translation in LINGO

In the following example, numbers are used to represent blocks of text to be discussed. In some cases the block is a single word delimiter.

Initially, language text is translated by COMPILER in the basic language of LINGO.

1) The first identifier encountered by COMPILER is SAVE. The tables defining the basic language and the state of COMPILER are stored in the pushdown stack and METACOMPILER is called.

2) The block of meta-language text from START to FINISH is translated by METACOMPILER and a new language is defined. Sentences of this
language are ALGOL-like statements with the right part a simple arithmetic expression using only addition.

3) A sentence in the language is parsed and machine language emitted.

4) SAVE is encountered by COMPILER. Parsing of the previous statement is completed. The tables defining the language and the state of COMPILER are saved in the pushdown store and METACOMPILER is called.

5) The block of meta-language text from START to FINISH is translated by METACOMPILER and a new language is defined. Sentences of this language now include arithmetic expressions with both terms and factors. Addition has been replaced by subtraction to make it clear that 3) and 8) are not sentences in this language.

6) A sentence in the language is parsed and machine language emitted.

7) RETURN is encountered in COMPILER. Parsing is completed. The tables which define the previous language and the local pushdown stack (which represents the previous state of COMPILER) are reinstated. A procedure return is executed from COMPILER through METACOMPILER to COMPILER.

8) COMPILER can now translate sentences in the language defined in 2). A sentence is parsed and the machine language emitted.

9) RETURN is encountered in COMPILER. Parsing is completed and the basic language of LINGO is reinstated.

10) An End-of-Program is encountered. The program is syntactically correct and is stored on a magnetic tape ready for execution.

The above example used only simple languages for clarity. There is no restriction on the complexity of the languages used in LINGO as long
DYNAMIC SYNTAX DIRECTED COMPILER

1. SAVE

START
CHANGE <BLCK>+<BLCKBODY> IS EMPTY;
CHANGE <BLCKBODY>+<STATEBLOCK> IS EMPTY;
DEFINE <STATEBLOCK>+<STATEMENT>.<STATEBLOCK> IS EMPTY;
APPEND <STATEBLOCK>+<STATEMENT> IS EMPTY;
DEFINE <STATEMENT>"IDENT"."+"<LEFTPART>.";" IS STO *;
DEFINE <LEFTPART>+<EXPRESSION> IS EMPTY;
DEFINE <EXPRESSION>+<EXPRESSION>."+"."IDENT" IS STK *, ADD;
APPEND <EXPRESSION>+"IDENT" IS STK *

FINISH
3. X + Y + Z + W;
STK Y 00100001
STK Z 00100002
ADD 00100003
STK W 00100004
ADD 00100005

4. SAVE

STO ( X ); 00100006

START
CHANGE <EXPRESSION>+<EXPRESSION>."-"<TERMPART> IS SUB;
APPEND <EXPRESSION>+<TERMPART> IS EMPTY;
DEFINE <TERMPART>+<TERM> IS EMPTY;
DEFINE <TERM>+<TERM>."x"<FACTOR> IS MUL;
APPEND <TERM>+<FACTOR> IS EMPTY;
DEFINE <FACTOR>"IDENT" IS STK *
APPEND <FACTOR>"("<LEFTPART>.,")" IS EMPTY

FINISH
6. X + X - Y X (Q - R);
STK X 00100007
STK Y 00100008
STK Q 00100009
STK R 00100010
SUB 00100011
MUL 00100012
SUB 00100013

7. RETURN

STO ( X ); 00100014

START
APPEND <EXPRESSION>+<TERMPART> IS EMPTY;
DEFINE <TERMPART>+<TERM> IS EMPTY;
DEFINE <TERM>"IDENT" IS STK *
APPEND <TERM>"("<LEFTPART>.,")" IS EMPTY

FINISH
8. X + Y + Z + W;
STK Y 00100015
STK Z 00100016
ADD 00100017
STK W 00100018
ADD 00100019

9. RETURN

STO ( X ); 00100020

10. S
as they are simple precedence phrase structure languages.

Evaluation

LINGO provides the programmer with a very powerful tool. It is the ability of the programmer to match programming techniques with programming tasks. The programmer must pay for the generalized ability in several ways.

Complex precedence phrase structure programming languages are difficult to define and can be obtained only with great care on the programmer's part. LINGO provides debugging aides to help in this task, but translation is terminated if the language is not a precedence language or if the precedence functions do not exist. A simple modification in the program will allow the use of the precedence matrix rather than the precedence functions for parsing.

Conclusions

LINGO is the first attempt to allow the programming language to change during translation. Dynamic language definition is possible through the formal definition of simple precedence phrase structure programming languages.
Bibliography


