International Institute for Applied Systems Analysis
IIASA www.iiasa.ac.at

The Use of Translator Implementation Methods for Writing Nonprocedural Interfaces to Application Software Systems

Melichar, B.

IIASA Working Paper

WP-82-015

February 1982

THE USE OF TRANSLATOR IMPLEMENTATION METHODS FOR WRITING NONPROCEDURAL INTERFACES TO APPLICATION SOFTWARE SYSTEMS

Borivoj Melichar

February 1982
WP-82-15

PREFACE

One of the results of advances in computer hardware technology is a wider use of computers in almost all areas of society. There is a need to make it possible for many people to use application software systems that are produced for different areas. For people without data processing backgrounds it would seem wise to use a nonprocedural interface to these systems. The implementation of a nonprocedural interface as a part of an application software system can be facilitated using aspects of the theory and practice of the translator construction for programming languages.

The purpose of this paper is to introduce programmers who lack theoretical background and/or practical experience in the area of translator design and implementation to the relevant aspects of its theory and practice.

ACKNOWLEDGMENTS

I would like to thank Goeran Fick for inspiring me to do this work, Ronald Lee and Donald Costello for the many useful and helpful discussions as well as their comments. Thanks also to Miyoko Yamada for her help in the preparation of this paper.

CONTENTS

THE USE OF TRANSLATOR IMPLEMENTATION METHODS FOR WRITING NONPROCEDURAL INTERFACES TO APPLICATION SOFTWARE SYSTEMS

Borivoj Melichar

1. INTRODUCTION

Advances in microelectronics during the past decade have dramatically decreased the cost of computer hardware. One of the results of the availability of low cost computers is a potential for much wider use of computers in almost all areas of society. This development is dependent on the existence of an easy-to-use interface between the user and the computer. The manner of an interface between a user and the computer is mainly determined by the medium used for communication. One basic media for this communication is alphanumeric text or graphics. More advanced media for communication (speech, eye movement, hand written script, brain wave control, etc.) are being researched, but they are not yet in common use. Here we focus on alphanumeric texts as a medium for user-computer communication.

For different areas of application of computers many application software systems have been or will be produced. Among others we can mention some application software systems like:

- computer based text editing and text formatting systems,
- office automation systems,
- information and data base systems,
- decision support systems,

— teleconferencing systems,
— business data processing systems,
— computer aided design systems, etc.

Our concern here is the problem of communication between user and application software.

Conventionally the user must translate his problem solution from terms meaningful in his problem domain into the terms of the application software available to him. Frequently the manner of application software use have been dictated by computer considerations rather than by application characteristics. In general this translation process involves an algorithmization of the user's problem requiring the procedural specification of numerous details not relevant to the original problem.

With the great diffusion of cheap computer hardware it is expected that the most rapidly expanding category of users is the category of users with little or no data processing background. For such people it is very hard and very time consuming to learn a method of construction and description of algorithms in programming procedure oriented languages.

Such users, called by Schneiderman (1978) "non-trained intermittent users" (vs. skilled frequent users) are interested in communicating to the computer what results should be produced, not how in detail computer should operate. In other words this means that it is highly desirable for such categories of users to communicate with an application software in a nonprocedural manner.

According to McCracken (1978), we can characterize non-procedural languages as follows:

a) The user cannot take any care of storing data. Decisions that relate only indirectly to the calculation are considered to be part of the internal functioning of the system. These include decisions about internal representation of numbers (fixed point, floating point, octal, decimal), dimensions of quantities that occur only as intermediate results, input and output formats, etc. The representation of data is selected by the system itself, and the description of the data representation with the data. This is called *data independence*.

b) The user cannot tell the computer *how to do* a process to obtain desirable results. Rather, he/she tells the computer only what he/she wants. This means that user input does not involve the loops and branches which make up most of the computational steps in a program written in procedural language. The user cannot even explicitly specify the order in which operation are to be performed. This we can call *control independence*.

In a non-procedural language the computational process is specified by the desired result and this specification is data and control independent. In practice, however, some vestiges of control or data dependencies remain. The languages in which the computational process is specified by the desired result and this specification is in a limited manner control or data dependent are often called non-procedural oriented languages.

Melichar (1981) describes an annotated classification of available types of text oriented non-procedural communication languages between the users and application software. Non-procedural languages are classified as follows:

a) answer languages,

b) command languages

c) query languages,

d) natural like languages,

e) special purpose languages, and

f) two-dimensional positional languages.

Because of the great variety of complexity of the non-procedural languages from the point of view of the implementation, we shall divide these languages into the following three groups:

1. *Trivial languages*, the structure of which is very simple. Examples of such languages are binary language, menu selection systems, or other simple answer languages.
2. *Formal languages* like command languages, query languages or special purpose languages, which can be uniquely defined by formal systems. A definition of such a language has two parts:
   - definition of syntax, and
   - definition of semantics.
3. *Natural languages*, the structure of which is very complex. A definition of them consists of:
   - syntax definition,
   - semantics definition, and
   - semantics interpretation with respect to a world model.

Here, we shall deal with methods for the description and implementation of the *formal language interface. The reason for that is* that trivial languages are very simple and it is not appropriate to use sophisticated methods for their implementation. On the other side, the implementation of the natural language interface needs different approach than one described here.

An application software system with non-procedural user interface is shown on the following figure (Figure 1).

The non-procedural interface in Figure 1 we can comprehend as a translator from a user's non-procedural language into an internal procedural language of the system. This internal language involves statements or instructions that are used for calling of individual procedures of an application software to perform operations desired by a user. The application software is usually the library of procedural programs written in some procedural programming language.

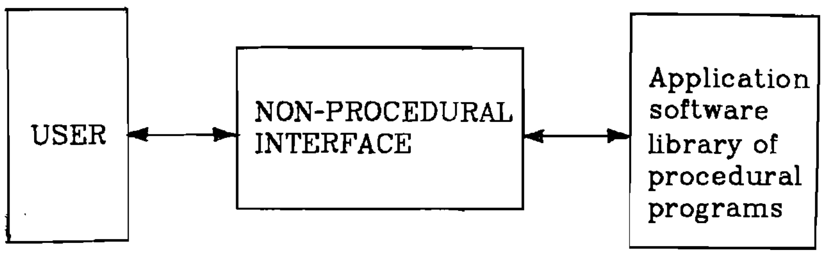

Figure 1. Application software system

It is often true that many application software products are not written by programming specialist familiar with the current state of the programming technology, but by experts in the application areas, whose programming skills are secondary to their main purpose. The result is that the application procedures are not separated from the user interface. In such a system the code describing these two terms are mixed altogether. But there are several good reasons for separating the application software from the user interface:

- It is the first step to divide complex programming task into the two simpler subtasks.
- It is possible to use for the implementation of the user interface available tools (see below).
- Often it is desirable to have different interfaces for different classes of users. A casual user has different requirements than the intensive user.
- The requirements for a user interface are not stable over time. Individual users evolve from casual to intensive mode, and vice versa. This means that it is necessary to change user interface without changing application software.
- The application software itself might be changed in time due to the change of computer architecture or a new development in the application area.

As soon as the user interface is separated from application software it is possible to implement it as a translator. Therefore it is possible to use for the user interface implementation, methods and tools already developed for the translators of programming languages implementation.

Since the late 1950's there has been a considerable interest in development of theoretical and methodological tools useful for a description of formal languages and for a description of a process of translation. This interest has been provoked by compiler writers who have been implemented programming languages like FORTRAN, ALGOL, etc.

As a result of the effort exerted many theoretical issues have appeared during the 1960's and 1970's. The most important theories of our concern here are

- theory of formal languages,
- formal methods for description of semantics,
- theory of automata, and
- theory of translation.

These theoretical advances facilitate the description of formal languages and description of a process of translation from one formal language into another.

Though the theoretical development was motivated and its results are used mainly in the field of programming (procedural) languages, there is a possibility to use known techniques in the design and implementation of nonprocedural interface as well.

The theories mentioned above facilitate not only the description of a language or a process of translation. The important issue is the fact that a formal description of a language can be directly converted into an algorithm of analysis of sentences in this language. In the same way, the formal description of a translation process can be converted into an algorithm of the translation.

The latter issue is the basis for an automatic construction of translators. If we have a description of the translation process, we can automatically convert the formal description of the translation into a translator by means of a special program, called *constructor*. The role of the constructor is shown in the following figure (Figure 2).

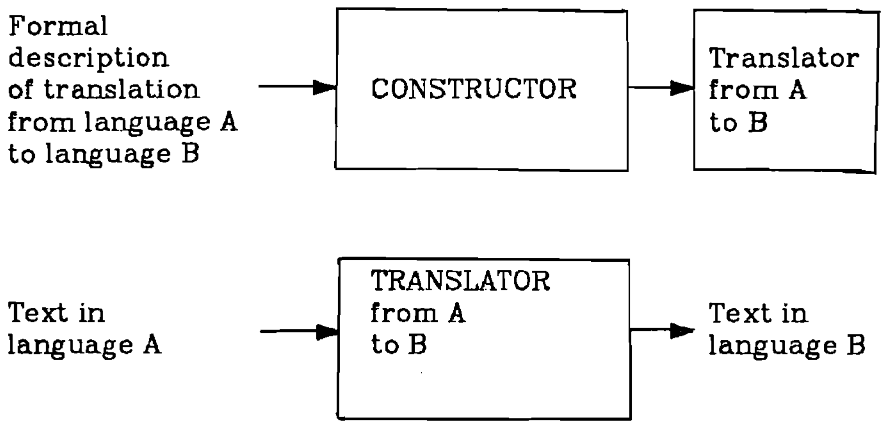

Figure 2. The role of constructor

The development of constructors of translators proceeded in parallel with the development of theoretical basis. The constructors of translators or their parts are called *compiler- compilers* or *translator writing systems*. These systems, primary intended for the implementation of classical translators of programming languages, have proved useful in implementation of a wide variety of application software products as well.

As mentioned above application software systems are written very often by experts in the application area in question. Therefore it is probable that these application programmers have no or little background in the field of the translator design and implementation. Nevertheless the application programmers can use existing tools for translator writing or to ask computer science specialist for a help. In order to start the use of translator writing tools or to start discussion with the computer science specialist the basic knowledge will be helpful.

In the sequel therefore we discuss some basic aspects of the description of formal languages, description of a translation process, basic structures of translators, and finally the principles of translator writing tools. The material is discussed in an informal and tutorial manner. The detailed description of the matter in question is possible to find in many books (Gries 1971, Hopcroft and Ullman 1969, Aho and Ullman 1972,1973, Lewis, Rosenkrantz and Stearns 1976, Aho and Ullman 1977, Backhouse 1979, Barrett and Couch 1979).

2. DEFINITION OF FORMAL LANGUAGES

Input to a non-procedural interface (input text) can be viewed as a string of characters chosen from some alphabet. Definition of which string of characters represents a valid input text is given by rules. These rules are called the *syntax* of the language. It is well known that it is often very difficult to state concisely and precisely what strings are valid input strings, just as it is hard to state which sentences of the particular natural language (for example in English) are proper and which are not. Those languages where their syntax can be uniquely and precisely defined by syntax rules are called formal languages.

A notation called a *grammar* is used for the specification of the syntax of formal languages. This notation has a number of advantages as a method for the syntax specification:

- A grammar gives a concise and easy to understand syntactic specification for statements, groups of statements, or sentences of a particular formal language.
- A grammar describes a structure of texts or sentences that is useful for its translation or interpretation.
- A grammar can be used as a base for description of the meaning of statements or sentences of the languages.
- A grammar is often part of the input used by a constructor to produce translator.

In the following we shall use term *sentence* for a valid user input to an interface.

Once we have defined the syntax of a language, we must specify what is the 'meaning' of each syntactically well formed string in the language. The rules that give specification of meaning of strings in a language are called the *semantics*.

There are essentially four approaches to define semantics:

— operational or interpretative approach, considering an interpreter that executes some string as program on the abstract machine,

— axiomatic approach, considering the input-output relation of a string as a program,

— denotational or mâthematical approach, considering definition of mathematical objects relevant and rules given for translating the string to these mathematical objects,

— translational approach, considering the translation of string into some target language whose semantics we already understand.

The last approach seems to be most natural to our purpose, for non-procedural interface description and implementation.

Often for a given language several different specifications exist (Bochman 1979):

— specification used during the design of the language,

— specification that describes the implementation of the language in terms of translator,

— specification for the user of the language, as written in a user manual.

Often, only the syntax is formally defined in each specification by use of a grammar. Unfortunately, it is very hard to find a specification of semantics that is suitable for all purposes.

Translational grammar and attribute grammars are examples of formal systems, which are popular for semantics specification using the translational approach.

Let us briefly review the notion of a grammar. The English sentence

THE CAT ATE THE MOUSE

has a syntax structure that might be visualized in diagram form as in the following figure (Figure 3).

A sentence diagram like this is called *parse tree*. It describes syntax, or structure of a sentence by breaking it into its constituent parts. To describe this structure we have used new symbols—names of syntactic constructs, e.g., <sentence>, <direct object>, etc. These symbols, enclosed in corner brackets, are called *non- terminal symbols*.

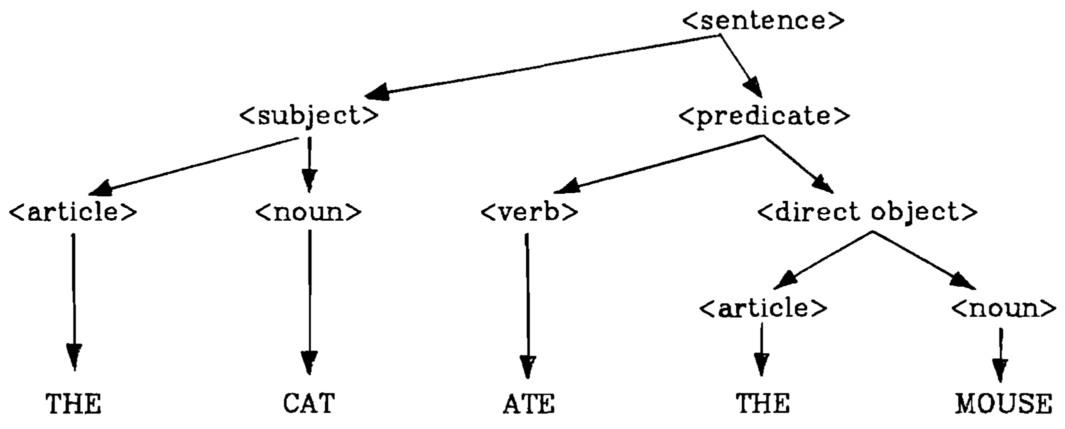

Figure 3. Parse tree

One of them is called sentence symbol or start symbol. This symbol is used as a name of all sentences in a language. In our example this symbol is <sentence>, in programming languages it is often usually the symbol <program>. The sentence itself is composed of basic words (THE, CAT, ATE, THE, MOUSE), which are called *terminal symbols*. The parse tree of the sentence also indicate that the <sentence> is composed of a subject followed by a <predicate>, the <subject> is composed of an <article> followed by a <noun>, etc. We can write *such rules* in the following form:

<sentence> → <subject> <predicate>

<subject> → <article> <noun>

<predicate> → <verb> <direct object>

<direct object> → <article> <noun>

<article> → THE

<noun> → CAT

<noun> → MOUSE

<verb> → ATE

From this point of view we can agree that grammar consists of the

1. definition of the set of non-terminal symbols,
2. definition of the set of terminal symbols,
3. definition of the set of rules, and
4. determination of the sentence symbol from the set of non-terminal symbol.

Once we have a grammar, the rules of the grammar can be used to *derive* or *produce* a sentence by the following scheme.

We start with the sentence symbol and find a rule with this sentence symbol to the left of → and rewrite it as the string to the right of →. In our example

<sentence> → <subject> <predicate>

Thus we are replacing a non-terminal symbol by one of the strings of which it may be composed, e.g., with the string on the right hand side of the rule with nonterminal in question on the left hand side of it. Repeating this process yield to the *derivation* like:

<sentence>	=>	<subject> <predicate>
	=>	<article> <noun> <predicate>
	=>	THE <noun> <predicate>
	=>	THE CAT <predicate>
	=>	THE CAT <verb> <direct object>
	=>	THE CAT ATE <direct object>
	=>	THE CAT ATE <article> <noun>
	=>	THE CAT ATE THE <noun>
	=>	THE CAT ATE THE MOUSE

Note that at each step of the derivation one can replace any non-terminal symbol. The terminal symbols are not replaced.

The grammar in this example describes one sentence of English only. The purpose of a grammar is to describe all sentences of a language with a reasonable number of symbols and rules. It is even possible to describe languages with an infinite number of sentences. Let us have a grammar with the following rules:

<number> → <digit> <number>

<digit> → 0

<digit> → 1

and with the non-terminal symbols <digit> and <number>. The sentence symbol is <number>. The terminal symbols are 0 and 1. This grammar describes an infinite set of binary numbers.

Now let us proceed to the specification of semantics. As we have stated above, semantics consists of a specification of the meaning of a sentence of the language. Here we shall consider a translational approach of the semantics specification.

The simplest method of the translational semantics specification is an association of semantic actions with each syntactic construct. These semantic actions, often called semantic routines (Gries 1971), output actions (Aho and Ullman 1977), etc., may involve the computation of values for internal variables of the translator, the invocation of some procedure to perform particular operation, etc.

To formalize this way of semantics specification we can use notion of the *translational grammar*. (transformational grammar).

The translational grammar is a natural extension of a grammar. Having a grammar, we obtain a translational grammar in the following way:

- we define the set of output symbols, and
- we permit output symbols to appear on the right hand side of grammar rules.

The following example shows the translational grammar which describes the translation of expression from infix notation to postfix notation.

This grammar have the following six rules:

<expression> → <expression> + <term> ADD

<expression> → <term>

<term> → <term> * <factor> MPY

<term> → <factor>

<factor> → (<expression>)

<factor> → a A

Note that these rules are composed of:

non-terminals:	<expression>, <term>, <factor>,
terminals:	+, *, (,), a
output symbols:	ADD, MPY, A.

The sentence symbol is <expression>.

Similar to the earlier grammar, we can generate sentence via a derivation.

In our example of the translational grammar we can write among others the following translational derivation:

```
<expression>   =>   <expression> + <term> ADD

               =>   <term> + <term> ADD

               =>   <factor> + <term> ADD

               =>   a A + <term> ADD

               =>   a A + <factor> ADD

               =>   a A + a A ADD
```

The resulting string is a mixture of input and output symbols. We obtain the input string by dropping the output symbols. In our case the input string is a + a. Similarly, to obtain output string we must drop the input symbols. Therefore the output string is A A ADD.

It means that the translation of the input string, a + a is the output string, A A ADD. In this way we may obtain output string corresponding to each well formed input string.

The translational grammar describes translation often called string-to-string translation. It is a special case of more general formal systems for description of string-to-string translation like for example syntax directed translation schemes or pair grammars.

Use of this types of formalisms to describe process of translation have some limitation. For example, let us suppose that output symbol represents an invocation of some procedure. From the discussion above it follows that this procedure has no parameters. This shortcoming may be eliminated using attributed translational grammars (or attributed transformational grammars) (Lewis, Rosenkrantz and Stearns 1976).

An attributed translational grammar is translational grammar each symbol of which (non-terminal terminal, or output symbols) has a certain number (it may be zero) of attributes. The determination of the attribute values is realized by semantic rules which are associated with the translational grammar rules.

The attribute translation grammar describes an *attributed translation*. It means that with an *attributed input string* is associated an *attributed output string*. For example consider the input string

a(X) + a(Y) * a(Z)

vwhere a, +, * are input symbols and X, Y, Z are values of the attributes of the a's. The following output string may be associated with that input:

MPY (Y,Z,R1) ADD(X,R1,R2),

where MPY and ADD are output symbols and X, Y, Z, R1 and R2 are values of their attributes.

Note that ADD (a,b,c) can be interpreted as the instruction to a computer to add a to b and store the result in c, for example.

Attributed translational grammars are formal systems for translational semantics specification with respect to syntax specification using context free grammars. Other formal methods to describe translational semantics are discussed in Riedewald (1978).

3. STRUCTURE OF A TRANSLATOR

3.1 Compilers and Interpreters

A *translator* is a program which processes a *source text* written in a *source language* with a procedure known as compilation or interpretation. Therefore we can divide translators into two main groups:

1. compilers, and
2. interpreters.

Figure 4. shows the principle of the compiler.

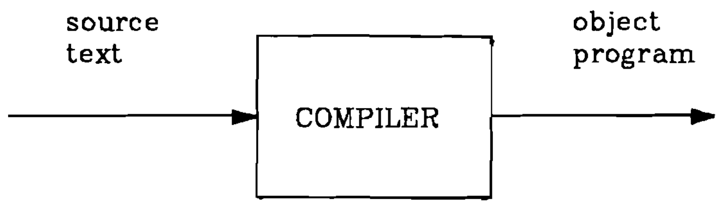

Figure 4. Principle of compiler

The compiler reads the source text and translates (transforms) it into an equivalent object program written in the object language. The object language is usually specific to a particular machine or a particular class of machines. Hence during the *compile time* the compiler produces an object program. The actual execution (or running) of the object program occurs at a *run- time*.

This means that the running of the object program, e.g., reading of input data, execution and producing an output data, is activity performed at another time than at the compile time.

Figure 5 shows the principle of the interpreter.

The interpreter reads the source text, analyzes it and performs the operations prescribed by the statements of the source text. This activity is called an interpretation of the source program.

During the interpretation of the source text input data are read and output data are produced immediately during interpretation. No object program is produced by the interpreter.

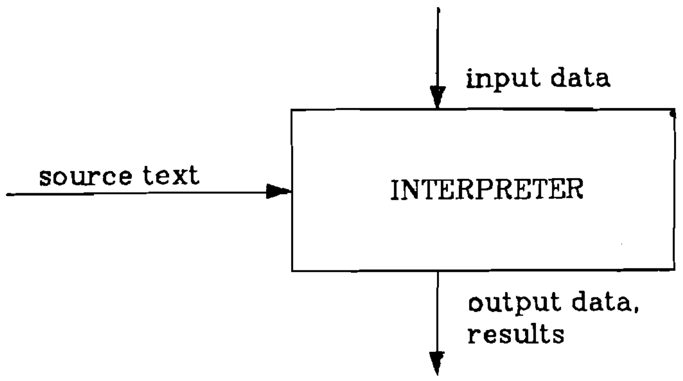

Figure 5. Principle of interpreter

In the context of developing nonprocedural interfaces we may develop either compilers or interpreters. In the case of using an interpreter as a nonprocedural interface, the corresponding operations or procedures are performed during processing of a user input. In order to repeat these operation later, the input text must be interpreted again.If the original input text is not stored the user must type this input text again. In contrast to the interpreter, the compiler produces an object program which can be run immediately and also may be stored for further use. The object program is composed of instructions or statements for calling individual procedures or programs of the library of procedural programs.

3.2 Basic Structure of Compilers and Interpreters

Existing compilers and interpreters display a wide variety of internal structures. Despite this variety there are many common features.

The compiling process (performed by compiler) can be divided into four logical steps:

1. lexical analysis,
2. syntax analysis,
3. semantics analysis, and
4. object code generation.

We can consider the structure of the compiler according to this classification as in Figure 6.

Lexical analyzer is the simplest part of the compiler. It reads *characters* of the source text and creates basic *symbols* like numbers, names, reserved words, delimiters, etc. At the same time lexical analyzer skips characters which are meaningless for the further processing, for example spaces, line characters, comments, etc.

The further role of the lexical analyzer is a transformation of the source text symbols into an internal machine form. The source text symbols are then represented by integer numbers, generally in binary form.

This is generally done to simplify further analysis of the source text by permitting subsequent parts of the compiler not work with strings of characters of variable length but with integer having a constant length.

The output of the lexical analyzer is a string of symbols. This *string of symbols* is the input to the *syntax analyzer*. This analyzer checks to see if the input text is written correctly according to the syntactic rules. At the same time the syntactic structure is determined and is usually outputed from the syntax analyzer in the form of a *derivation tree*.

During the *semantic analysis* the semantic correctness is determined. In addition the input to the object program generator is prepared.

An object program generator creates the output of the compilation process(eman object program.

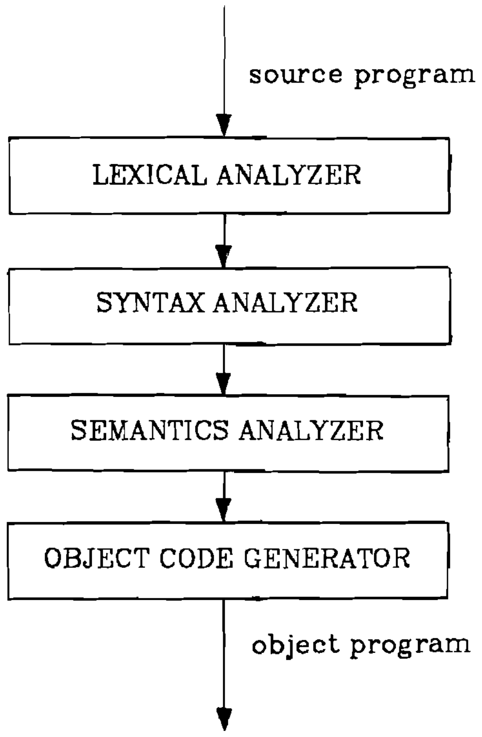

Figure 6. Structure of the compiler.

Similar to the compiling process we can divide process of the interpretation into four parts:

1. lexical analysis,
2. syntax analysis,
3. semantics analysis, and
4. interpretation.

We can consider the structure of the interpreter as shown in Figure 7.

The first three parts of the interpreter are essentially the same as those in the compiler. The fourth part, the interpretation part, is the only different part of the interpreter.

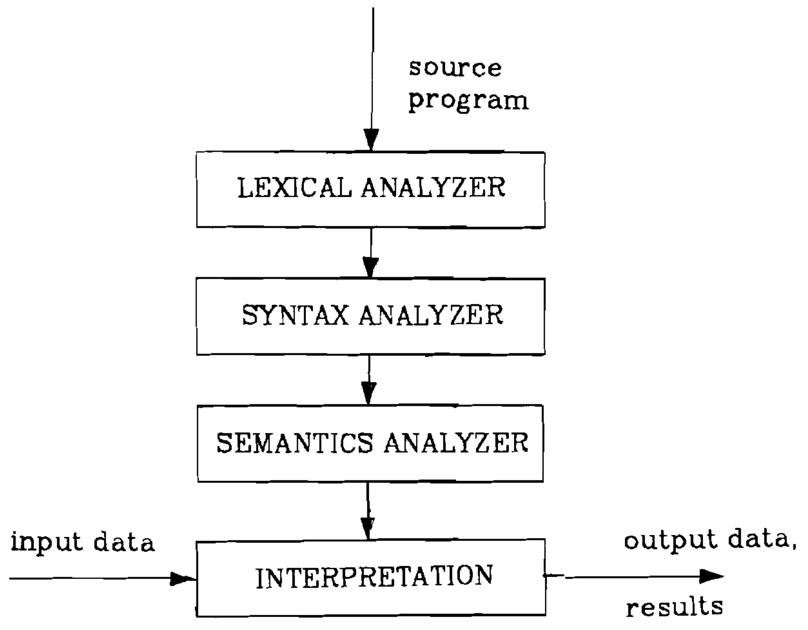

Figure 7. Structure of the interpreter

3.3 Batch and Conversational Compilers and Interpreters

Another subdivision of translators is into

1. batch, or
2. interactive

ones.

Batch translators are used in computer systems where a user has indirect access to a computer. In such a system user can neither influence the way of computation or interpretation nor intervene in case when error occurs. This means that the whole program must be prepared before starting translation.

In operating systems which enable direct access to the computer an interactive translator can be used. In such systems user can intervene during translation or can directly correct errors if they occur using a terminal.

Now we shall discuss the structures of interactive compilers and interpreters.

The principle of interactive interpreter can be described as follows:

The source text is read from a terminal, checked on syntactic and semantic correctness, and then is either immediately interpreted or stored and interpreted just after the user's request. In the latter case the user can request interpretation of one statement, part of the stored text, or the whole input text.

The interactive interpreter is composed usually of two main parts:

1. control part, and
2. interpretative part.

The control part of the interactive interpreter has the following basic functions:

1. to perform reading of the input text,
2. to ensure checking of the correctness of input text using the interpretative part,
3. to store the input text or ensure direct interpretation of the input text, and
4. to ensure interpretation of stored text.

Figure 8 shows the structure of an interactive interpreter.

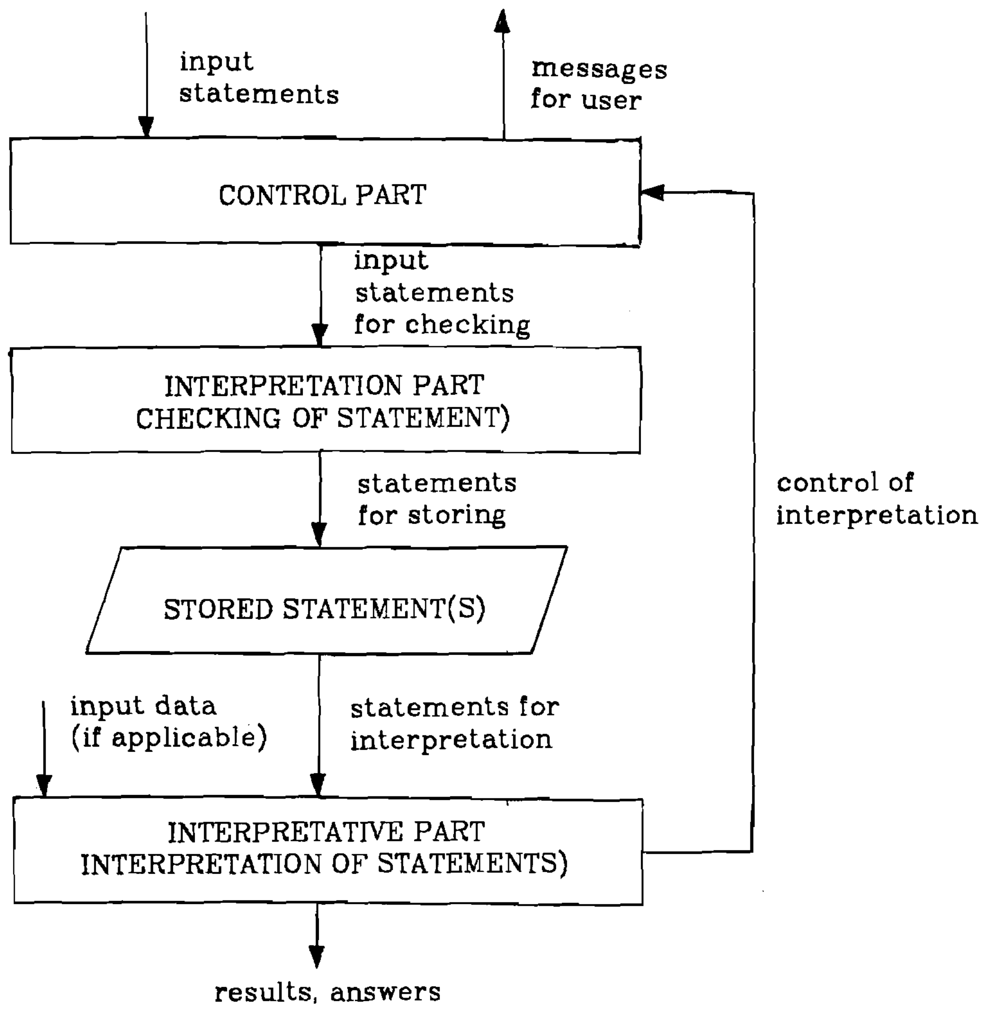

Figure 8. Structure of an interactive interpreter

The interpretative part of the interactive interpreter has two functions:

1. checking of statements, e.g., to perform the lexical, syntax and semantics analysis, and
2. immediate interpretation of the checked statement or interpretation of the stored statements.

The reason for performing both functions by the interpretative part of the interactive interpreter is, that lexical, syntax and semantics analysis are parts of the interpretation of the statement.

An interactive interpreter is advantageous namely in the case when we use a particular input text only once. An interactive interpreter does not produce an object program equivalent to the source text. This is a disadvantage in situations when the input text is used many times. An example of the use of one input text more than one time occurs in communication with a data base. In this case it is often desirable to use the same query which can be very complex. In case when we want to use an input text more than once to communicate with computer, it is better to use an interactive compiler producing an object program.

Interactive compilers are often called incremental compilers, because the object program is not created continuously as with batch compilers, but it is produced by increments. The increment is obviously the object program corresponding to one source statement or other simple part of the source text.

The incremental compiler, in contrast to the interactive interpreter, is composed of two parts:

1. control part, and
2. compilation part.

There are following two variants of the incremental compilers:

1. Compilation of the source statement is performed just after its input, provided that this statement is correct.
2. The source text of the statement is stored and compilation is performed just before execution of the statement.

The control part has in the former case the following functions:

a. During the input of the source statement the compilation part is called to perform *checking* and *compilation* of the input statement. At the same time changes in other statements are performed which are evoked by the context relations with the compiled statement. The object program created is stored for further use.
b. When the user wants to execute the object program, the control part ensures it.
c. If the user wish to output the object program, the control part using the increments compiled, creates complete object program. This object program is executable independently on the compiler.

In the latter case, e.g. when incremental compiler compiles each statement just before execution, the control part has the following functions:

a. During the input of the source statement the compilation part is called to check the input statement. If errors do not occur, the source statement is stored.

b. During the execution of the program the control part ensures the compilation of statements which should be executed. Created increments of the object program are stored, and in the case when the statement is executed again the stored object program is used for this execution.

If the user ask to create an object program, the control part using the increments compiled previously creates a complete object program. Such an object program can be executed independently on the compiler.

4. TRANSLATOR WRITING TOOLS AND AIDS

The implementation of translators is becoming easier due to the development of new methods mainly during the last two decades. Nevertheless, the implementation of a good translator is still a nontrivial task. Since 1960's there has been considerable interest to use a computer to reduce effort needed to construct a good translator. The result is that numerous software tools, called compiler-compilers or translator writing systems was developed for the translator implementation. These systems are mainly used for the implementation of translators of programming languages. But very soon was found that the translator implementation tools are useful for the implementation of some parts of a wide variety of application software products as well. Especially they are useful for constructing of the user interfaces of application software systems (Rosenthal 1980).

Johnson (1980) points that usage of available tools for software construction proved to be very successful and has the following advantages:

— The resulting products are produced quickly.
— They are likely to work correctly.
— They are often quite flexible and adaptable as application change.
— Inter-machine portability is often enhanced by using tools.
— Tool usage encourages a natural modularity in the resulting program.
— Using tools constructed by expert programmers get the use expert algorithms.

Further advantage is that user of the tool need not have a detailed knowledge of theoretical disciplines which is the tool based on. Though it must be mentioned that some basic knowledge is helpful.

From the point of view of application software and user interfaces, we can add the following advantages (Rosenthal 1980):

- The code defining the operations of the application is separated from the code defining the user interface. The application operations are defined by a library of procedural subroutines. The user interface is defined by an input language specification.
- The user interface (translator of input language) is generated automatically from the specification of the input language and the programmer is forced therefore to specify this language formally, which assists both design and documentation. The latter is important, since the specification can be read by the user.

Research and development of translator writing tools is still continuing. In spite of many useful and practical results in this field, there is no satisfactory system for construction the translator as a whole. Some projects of this type are in progress as for example the Production-Quality Compiler-Compiler project (Leverett et al. 1980).Translator writing tools for construction of different parts of translator are generally available. There exist constructors for construction of lexical analyzers (Lesk 1979), constructors for syntax analyzers called parser generators (Johnson and Lesk 1978), constructors for syntax analyzers combined with semantic analyzers (Mueller 1977, Koster 1979).

The methods of automatic construction of object code generator are still in research (Graham 1980, Cattell 1980).

Figure 9 is a box diagram of the most of translator writing systems and translators generated by them.

The translator consists of a skeleton translator and a set of tables which it uses. The tables contain all the necessary information used by the skeleton translator. These tables are generated by the translator writing system on the base of description of the source language and the specification of the translation from the source language into the object language.

The skeleton translator is composed of some number of universal algorithms. These algorithms are designed to provide different tasks in the translator, e.g., lexical analysis, syntax analysis, code generation, etc.

5. APPLICATION OF TRANSLATOR WRITING METHODS TO WRITING NONPROCEDURAL INTERFACES

The task of the design and implementation of an application software system we propose to divide into some number of subtasks:

- The definition of the input language of the user interface.
- The decision about interface being compiler or interpreter.
- The definition of the internal language of the system.
- The separation of the user interface from the application part of the system.

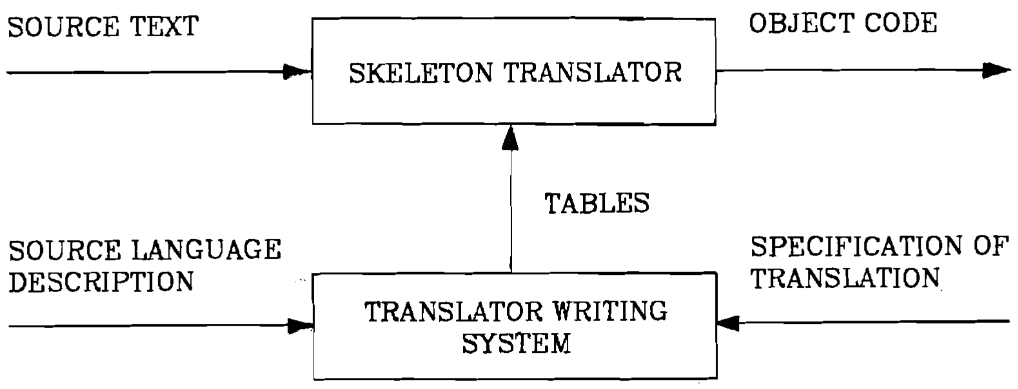
SOURCE TEXT
SKELETON TRANSLATOR
OBJECT CODE
TABLES
SOURCE LANGUAGE DESCRIPTION
TRANSLATOR WRITING SYSTEM
SPECIFICATION OF TRANSLATION

Figure 9. Translator writing system

- The implementation of a library of application programs.
- The definition of the translation.
- The implementation of the user interface.

The definition of the input language is generally the first step of implementation of the application software system. If the decision is, that the input language will be a formal language (see classification above), it is very useful and helpful to use a grammar for the description of this input language syntax. Once a grammar of the input language is written, it is possible to define semantics of each syntactic construct, e.g. meaning of each one. This may be done using natural language, because the formal methods of semantics description are not so simple and easy to understand like grammars for the description of the syntax.

The decision if the system should be either a compiler or an interpreter depends on the demand if one input text should be used many times like a "program". Compiler can produce an object program, which is possible to run more effectively than to process input text again.

Definition of an internal language of the system means the definition of the list of subroutines or subprograms in the application library and the way to call them. In the case of a compiling system this internal language is the object (target) language.

The separation of the user interface from the application part of the system is the principal argument. This separation makes possible to use different approaches for the implementation of the user interface and the application library. Further reasons for such a separation were listed above. But in real situations is sometimes very hard to decide what should be a part of the interface and what should be a part of the application library. For example the check if the datum like 1 August 1981 is meaningful, e.g., the number of a day is less or equal to 31 in August and the year must be inside the time span between 1900 and 2100, can be done by the interface or by the application subroutine for processing dates.

The library of application programs can be written in some programming language using methods as usual in an application area.

The definition of the translation depends on the definition of the input language and on the definition of the internal language of the . system. There is possible to use some formal systems to describe translation. As examples we mentioned above translational grammars and attributed translational grammars. The complexity of translation may vary from trivial cases to very complex ones. According to the complexity of the translation we must select an appropriate method of the translator implementation. This aspect is not discussed here because it needs an additional theoretical background.

If we use a particular translator writing system to implement user interface, we must describe the translation in terms of the system used. Roughly speaking, the use of translator writing system has an impact on decisions made in almost all stages of an application software system design.

As may be seen from the discussion above, the design and implementation of an application software system is mainly intellectual work. Only the implementation of an application library and user interface is possible to support by software systems like compilers of programming languages for the application library implementation or translator writing systems for the user interface implementation. For the other subtasks of design and implementation of application software system listed above is no efficient way to support them by a computer. They must be supported by the experience of the designer only.

6. CONCLUSION

Understanding of the translator construction for programming languages has increased substantially during the past two decades. Now it is possible to construct automatically reasonably good lexical and syntactic analyzers. Progress has also been made in the area of an automatic construction of semantic analyzers and code generators.

Most of methods used in automatic construction of different parts of translators is based on the notion of a grammar. Therefore the notion of a grammar is basic for the understanding of almost all methods of the translator construction. The methods of the formal description of semantics and methods of the description of a translation process are based on the notion of a grammar.

Despite the fact that translator writing systems was primary included for automatic construction of translators of programming languages, they can be used for implementation of non-procedural user interfaces as well.

At present there exists a large number of software systems, which can be used to make the task of a user interface implementation easier. For example, in Sweden (Carlsson and Guningberg 1980) more than ten software systems for writing interactive interfaces are available.

The results of the last two decades effort in developing the theoretical basis for translator construction and the experiences with the practical translator implementation are collected in many books. Some of them are (Gries 1971, Hopcroft and Ullman 1969, Aho and Ullman 1972, 1973, Lewis, Rosenkrantz and Stearns 1976, Aho and Ullman 1977, Backhouse 1979, Barrett and Couch 1979).

REFERENCES

Aho, A. V. and J.D. Ullman. 1977. Principles of Compiler Design. Reading, Massachusetts: Addison-Wesley.

Aho, A. V. and J. D.Ullman. 1972 and 1973. The Theory of Parsing, Translation and Compiling. (Vol. I. Parsing, Vol. II. Compiling). New Jersey: Englewood Cliffs.

Barrett, W. A. and J. D. Couch. 1979. Compiler Construction: Theory and Practice.

Bochmann, G. V. 1979. Semantic Equivalence of Covering Attribute Grammars. International Journal of Computers and Information Sciences 8(6):523-539.

Cattell, R. G. G. 1980. Automatic Derivation of Code Generators from Machine Description. ACM Trans. on Programming Languages and Systems 2(2):173-190.

Graham, S. L. 1980. Table-Driven Code Generation. Computer 13(8):25-37.

Gries, D. 1971. Compiler Construction for Digital Computers. New York: Wiley.

Hopcroft, J. E. and J. D. Ullman. 1969. Formal Languages and their

Relation to Automata. Reading, Massachusetts: Addison-Wesley.

Johnson, S. C. and M. E. Lesk. 1978. UNIX Time-Sharing System: Language Development Tools. Bell System Technical Journal 57(6):2155-2175.

Lesk, M. E. 1979. LEX—A Lexical Analyzer Generator. UNIX Programmer's Manual 2, Section 20, New Jersey: Murray Hill.

Leveret, B. W. et al. 1980. An Overview of the Production—Quality Compilerompiler Project. Computer 13 (8): 38-49.

Lewis, P. M. II, D. J. Rosenkrantz and R. E. Staerns. 1976. Compiler Design Theory. Reding, Massachusetts: Addison-Wesley.

McCracken, D. D. 1978. The Changing Face of Application Programming. Datamation 24 (November 15):25-30.

Melichar, B. 1981. Nonprocedural Communication between Users and Application Software. RR-81-22. Laxenburg, Austria: International Institute for Applied Systems Ananysis.

Riedewald. 1978.

Rosenthal, D. S. H. 1980. Tools for Constructing User Interfaces. Computer Aided Design 12(5):223 227.

Schneidermann, B. 1978. Improving the Human Factor Aspects of Data Base Interactions. ACM Transactions on Database Systems 3(4):417-439.