The SIMPL Family of Programming Languages and Compilers*

by
Victor R. Basili

Abstract

The SIMPL family is a set of structured programming languages and compilers under development at the University of Maryland. The term "family" implies that the languages contain some basic common features, such as a common subset of data types and control structures. Each of the languages in the SIMPL family is built as an extension to a base language. The compiler for the base language is written in the base language itself, and the compiler for each new language is an extension to the base compiler.

---

*This research was supported in part by the Office of Naval Research, Mathematics Branch, under Grant N00014-67-A-0239-0021 (NR-044-431).

I. Introduction

The purpose of this paper is to discuss an effective approach for designing and implementing a family of programming languages as that approach has been used in developing the SIMPL family of structured programming languages and compilers at the University of Maryland. The term "family" here represents a set of languages which contain some basic common features, such as a common subset of data types, control structures, etc. Present members of the SIMPL family family consist of the base typeless compiler-writing language SIMPL-X [1], the typed (integer, string and character), transportable compiler-writing language SIMPL-T [2], the mathematically-oriented language SIMPL-R [3], and a systems implementation language for the PDP-11, SIMPL-XI [4]. There are several other languages currently under development, including SIMPL-D, a language that provides the user with a facility for defining new data structures, and SIMPL-G, a graph algorithmic language which is the redesign of GRAAL [5], a programming language for use in the solution of graph problems primarily arising in applications. There are also several languages that have been proposed as additions to the family, such as SIMPL-RD, which is meant to be a transportable numerical applications language for very large data problems, SIMPL-S, a general systems language, a language for use in the development of the IPACS system [6] for pattern recognition problems, and a language for use in artificial intelligence problems.

The rest of this paper is divided into four sections. The next section contains a motivation for the family of programming languages

approach. Section three presents the description of the core language and compiler features of the family, and Section four discusses the members of the family.

II. Motivation

The general approach to solving a problem arising from a particular application area has been to write a solution for the problem using the terminology pertinent to that application area. In this way the problem solver is using the natural language of the problem area to work out a solution to his particular problem. This concept of a natural language has long been used by problem solvers in all disciplines. In mathematics, for example, there are a variety of languages available. The algebraist, in particular, uses specialized primitives, such as set-theoretic operations, etc., for expressing the relevant theorems and proofs that go to make up the problems and solutions of the area.

For problem areas whose solutions are algorithmic in nature, the algorithm is usually represented using the appropriate terminology, in this case basic data elements, operations, control and data structures, etc., associated with the area. If this problem requires a computation for which a computer is used, the algorithm is then encoded into some general-purpose programming language, and this encoded algorithm is then executed on the computer. If this language is not designed for algorithms in the problem area, and the transformation from the "solution" algorithm to the "execution" algorithm is great, then problems arise. To start with the development of the "execution" algorithm requires an extra effort which

might even duplicate the original "solution" algorithm effort. There may be little relation between the two algorithms, and the programming language may actually disguise the underlying algorithm. This encoding may thus introduce errors and will certainly make the algorithm difficult to read. The results might even be worse if the problem solver bypasses the "solution" algorithm and works out only the execution algorithm in order to save time. The programming language is not of aid in the "creative stage" of developing the algorithm, and some more insightful solutions to the problem may be lost.

There has been a growing emphasis on developing problem area oriented programming languages. There has been an ever-increasing number of high-level languages that involve the solutions of set-theoretic problems [7], combinatorial problems [8], artificial intelligence-based problems [9,10], graph algorithm problems [4,11]. The existence of a particular application area oriented programming language provides the solution to the above problems. If it contains the proper primitives, it can be an aid in the problem solving. It can produce highly readable algorithms since they can be expressed in the pertinent terminology of the problem area, generating a good, formally defined reference language for the area.

There are, however, several drawbacks to this approach to problem solving. First, the development of the design of a new programming language and the corresponding implementation of its compiler are fraught with many problems, especially if the language either contains constructs novel to language development or combines constructs that have not necessarily appeared together before. Also, it is not always obvious what operators

and data structures best express the problem solution for any given application area. Secondly, this approach can cause a proliferation of languages and compilers whose intersection is empty.

One possible approach that minimizes some of these drawbacks is the development of a family of programming languages and compilers. The basic idea behind the family is that all languages in the family contain a core design which consists of a minimal set of common language features. These features define the base language for which all other languages in the family are extensions. This also guarantees a common base design for the compilers. It does require that the family design be extremely modular and permit easy access to subsets of the language and individual segments of the compiler.

The major benefits to this approach are first that several application area languages can be developed with a single base structure, minimizing both the proliferation of different languages and the corresponding compiler development effort. Secondly, since many of the constructs for various application areas either overlap directly or are similar in design, the benefits derived from one research effort are either directly usable or of valuable help in the development of other research efforts. Thirdly, the basic extendable design lends itself quite easily to implementing different operators and data structures with minimal effort. In this way the specific set of primitives for the specialized language may be experimented with in an attempt to find those primitives that best express the solutions to problems presented by the application area. Fourth, the processes of language design and compiler implementation have been broken down into small well-defined

steps. This makes each process easier to accomplish.

It should be noted that this family approach is in contrast to the design of a single language and compiler that would encompass the needs of a whole range of application areas similar to the approach taken by a language like PL/1.

The second major problem is the proliferation of compiler development efforts. Since the set of languages forms a family, there are several approaches that may be taken which minimize the compiler development effort.

One approach is to develop an extensible language [12] and build the family of languages out of this extensible base language. The discussion of extensible languages can be broken into two parts. One is the set of languages with a powerful data definitional facility as defined in SIMULA 67 [13], or as recommended by Liskov and Zilles [14] or as is being proposed in SIMPL-D. Here the new language is actually a system built out of the base language by incorporating new data types and data structures into the language. No new compiler is built. This type of extension may cover a large variety of "new languages" and there is a minimal of effort involved. On the other hand, the execution of programs in this "new language" would probably not be as efficient as it would be if the new data types and structures were built into the compiler directly; the types of extensions are limited, i.e., one cannot extend the syntax or set of control structures, for example, and the base language must be fairly powerful and possibly more general than is necessary for the new application.

The second and more general type of extensibility is characterized by languages such as ELF [15] and IMP [16]. Here the user may define new syntactic and control structures and a new compiler can be generated. This added extensibility, however, requires more effort and knowledge (of such things as syntactic and semantic structure) than the previous type of extensibility. The major problem here is that the base language and compiler are usually large and powerful and tend to make the new language more powerful and its compiler less efficient and larger than what was needed for their design. The initial process of designing the base language and building its compiler is also a very major effort.

Another approach to developing a family of languages is the translator-writing system approach [17]. For all practical purposes, a TWS is usually limited to generating compilers for what in effort amounts to a family of languages as defined here (ignoring syntax) or else it becomes too inefficient. In many ways a TWS is a more flexible approach than the extensible language approach since it allows the user more freedom with the language design so the design can be tailored more precisely to the particular application area and not contain any unneeded language constructs. It generates a separate compiler which can be smaller and hence more efficient. Depending, however, on the particular TWS, it is usually more difficult for the user to develop his compiler and the ability to incorporate new data structures or types is quite limited.

Another approach, the one discussed here, is to start with a base language and a base compiler, building each new language in the family as an extension to the base language and each new compiler in the family as

an extension to the base compiler. This approach is similar to that suggested in [18].

The major benefits involved here are that each new language has its own compiler. The appropriate kind of error analysis can be built into it. If the base language is simple enough, the user gets no more overhead in his new language than is necessary for the particular application. The compiler need contain no extra extensibility support features. Since it is hand-coded, the compiler can be more efficient than a TWS generated compiler and the code generated can be more efficient. There is a greater amount of freedom in the language design within the limits of the family. Lastly, if each new language is implemented in a stepwise fashion, the implementation is made easier, and there is always some basic language and compiler to work with.

On the other hand, it does require more effort and knowledge than any of the other approaches to get a compiler up and running. How much effort, however, is the crucial point. Our experience with the SIMPL family seems to demonstrate that this approach can be accomplished with less effort than might be expected if the base language and compiler are properly defined.

The basic idea behind this core approach is the bootstrapping of each of the languages and compilers in the family. The essence of this bootstrap process is now described. Let SIMPL-A and SIMPL-B represent any two languages in the family such that SIMPL-B is to be an extension of SIMPL-A. The new language SIMPL-B is then defined as:

SIMPL-B = SIMPL-A U {new features}

so that SIMPL-A is just a subset of SIMPL-B. The SIMPL-B compiler is then built out of the SIMPL-A compiler (which is assumed to be written in SIMPL-A) using a bootstrapping technique which tries to minimize the amount of old code changed versus the amount of new code added. We will distinguish between two different types of extensions to the SIMPL-A compiler--those which are modifications to the code as it exists in the SIMPL-A compiler and those which involve the writing of new independent modules necessary to support the new features of the extended language. In general, attempts were made both in the original compiler design and in the style in which the extensions were done to force the majority of the work to be done in the form of writing independent modules or modifying code, where necessary, in as small and as well-defined a number of existing modules as possible.

The bootstrap process may have two goals: to build a SIMPL-B compiler written in SIMPL-A or written in SIMPL-B. The steps for both of these goals are represented in Figure 1--the first two steps yielding a compiler for SIMPL-B written in SIMPL-A, steps 3 to 5 taking that compiler and generating a compiler written in SIMPL-B.

Before discussing the actual bootstrap process, a brief explanation of the notation in Figure 1 will be given.

$$\frac{P}{L}$$

denotes the source version of program P written in language L and

denotes the executable form of P/L for some machine. C(L) represents the compiler for language L ; P/L mod X represents the modification or updating of a program P/L by some set of fixes or rewrites X to P/L written in L ; X ∪ Y represents the union of programs X and Y into one system; and X → Y represents the running of program X through program Y .

The general procedure then consists of first updating the SIMPL-A compiler to accept the SIMPL-B language. This process usually consists of writing a set of new program segments to handle the new features of B and then modifying the old compiler so that it recognizes these new features by possibly adding some extra cases to a case statement, some words to a table, and/or some flags to a descriptor. Step 2 involves the running of this new SIMPL-B compiler written in SIMPL-A through the SIMPL-A compiler producing an executable SIMPL-B compiler. The process stops here if the goal was a compiler for SIMPL-B written in SIMPL-A.

If, however, the goal was a compiler for SIMPL-B written in SIMPL-B, Step 3 calls for a rewrite of the SIMPL-B compiler in SIMPL-B. This rewrite involves taking those new features of SIMPL-B which would make the compiler work more effectively, easier to read, write or extend, more transportable or efficient and using them to rewrite the appropriate modules of the compiler. Step 4 calls for running the new compiler through the old one producing an executable SIMPL-B compiler written in SIMPL-B. Step 5 calls for the running of the new compiler through itself which would generate a

1. $$\left(\frac{\text{C(SIMPL-A)}}{\text{SIMPL-A}} \ \underline{\text{mod}} \ \left\{\frac{\text{B-fix}}{\text{SIMPL-A}}\right\}\right) \cup \left\{\frac{\text{B-routines}}{\text{SIMPL-A}}\right\} = \frac{\text{C(SIMPL-B)}}{\text{SIMPL-A}}$$

2. $$\frac{\text{C(SIMPL-B)}}{\text{SIMPL-A}} \rightarrow \boxed{\frac{\text{C(SIMPL-A)}}{\text{SIMPL-A}}} = \boxed{\frac{\text{C(SIMPL-B)}}{\text{SIMPL-A}}}$$

3. $$\frac{\text{C(SIMPL-B)}}{\text{SIMPL-A}} \ \underline{\text{mod}} \ \frac{\text{B-rewrite}}{\text{SIMPL-B}} = \frac{\text{C(SIMPL-B)}}{\text{SIMPL-B}}$$

4. $$\frac{\text{C(SIMPL-B)}}{\text{SIMPL-B}} \rightarrow \boxed{\frac{\text{C(SIMPL-B)}}{\text{SIMPL-A}}} = \boxed{\frac{\text{C(SIMPL-B)}}{\text{SIMPL-B}}}$$

5. $$\frac{\text{C(SIMPL-B)}}{\text{SIMPL-B}} \rightarrow \boxed{\frac{\text{C(SIMPL-B)}}{\text{SIMPL-B}}} = \boxed{\frac{\text{C(SIMPL-B)}}{\text{SIMPL-B}}}$$ 1

The bootstrapping of a compiler for SIMPL-B
from a compiler for SIMPL-A

Figure 1

better version of the compiler if the rewrites from Step 3 were of any benefit in the compiling of SIMPL-B programs.

It is not always of any value to have a SIMPL-B compiler written in SIMPL-B. Two places where it might be of benefit are if writing the SIMPL-B compiler in SIMPL-A required the use of some assembly code to access machine features not otherwise accessible (e.g., writing the SIMPL-R compiler in

SIMPL-T requires some assembly code to access the floating point hardware) or there are features in SIMPL-B which enhance the compiler writing (e.g., writing the SIMPL-T compiler in SIMPL-T permits the use of strings).

It is worth noting that Steps 1 and 2 and Steps 3 to 5 may be performed iteratively, making small numbers of modifications at a time to help with the ease of writing future modifications. The drawback to this iterative process is that reproducing the generation of the SIMPL-B compiler from the SIMPL-A compiler becomes a more complex step. This complexity can be a serious drawback in transporting the family of languages onto another machine.

III. Core of the Languages and Compilers

There are certain aspects of all the languages and compilers in the SIMPL family that are consistent. This section discusses these common features and their motivation. It contains essentially the definition of the first member of the family SIMPL-X and the second member of the family SIMPL-T. (An example SIMPL-X and SIMPL-T program may be found in the Appendix.)

The major motivation behind the design of the language SIMPL-X was that it was to be used as the base language for the family of SIMPL languages. Therefore the goals set for this language were fivefold. It should be as simple as possible since the features of the language would be common to all the members of the family. It should be designed within some generalized framework so that it can be easily extended. Like all languages it should be well-defined and consistent in the specification of its syntax and semantics. It should contain facilities for writing compilers since its compiler and that of its extensions would be written in itself. Lastly, it should be reasonably transportable so that programs

written in it and its extensions, including their compilers, can be moved to a variety of machines with minimal effort.

The three major design criteria for the compiler are its extensibility, its transportability, and its ability to produce relatively good object code. Extensibility is important in order to be able to build new compilers out of the present one for the various extensions of the languages. The design needs to be transportable so that all the languages and their compilers could be moved onto a variety of machines. Since the compilers for each member of the family are either self-compilers or written in the base language, the compilers need to generate fairly good object code.

To these ends, the salient language features of SIMPL-X are

1) Every program consists of a sequence of procedures which can access a set of global variables, parameters, or local variables.

2) The statements are the assignment, if-then-else, while, case, call, exit and return statements. There are compound statements in the language, but there is no block structure.

3) There is easy communication between separately compiled programs by means of external references and entry points.

4) There is an integer-type variable. Associated with this variable is an extensive set of operations which include arithmetic, relational, logical, shift, bit and partword operations.

5) There is a one-dimensional array data structure.

6) Procedures and functions may be recursive. Only scalars and structures may be passed as parameters. Scalars are passed by value or reference and structures are passed by reference.

7) There is a simple set of read and write stream I/O commands.

8) The syntax and semantics of the language are relatively simple, consistent, and uncluttered.

An overriding philosophy behind the design of the entire family was to keep the languages as simple as possible, thus the motivation for the name. The idea was to begin with a minimal, skeletal design and add the features for each new language at later stages in the development, only after these features could be studied, modeled, tested out, discussed with members of the community (in this case the members of the Computer Science Department of the University of Maryland), and deemed appropriate for the particular application the language was to address.

In keeping with this philosophy, it was decided that the syntax and semantics of the languages should be as uncluttered and consistent as possible. Syntactically this took the guise of eliminating syntactic redundancy and using consistent formats for similar types of features. Semantically, an attempt was made at minimizing the number of default options, forbidding special cases and keeping a consistent philosophy about the interpretation of similar language features.

Only the integer data type was included in the base language, SIMPL-X. In order to make the language powerful enough to write its compiler, however, an extensive set of operators were placed in the language. These included arithmetic (+, -, *, /), relational (=, $\neq$, $\leq$, <, $\geq$, >), logical (and, or, not), bit (bitand, bitor, bitexclusive or bitcomplement), shift (left and right), and partword operators.

The only data structure in the base language is the one-dimensional array.

Because this was the core design for the family, an attempt was made to keep the number of language features to the bare minimum without limiting

the power of the language. Several common language features were either eliminated from consideration or simplified. There is no block structure at this base level although there are compound statements. The language is procedure oriented and a program is composed of a set of segments (procedures or functions) which may access any of a set of globals, locals or parameters. All segments must be declared globally in a program, i.e., a procedure or function may not be declared within another procedure or function. However, any segment or data item may be labeled as an entry point and any program may contain references to externally declared segments and data. The calling conventions are also minimal with only scalars (by value or reference) and structures (by reference) passable as parameters; segments may not be passed as parameters. All these base restrictions make the runtime environment simple from the user's and compiler's point of view.

The requirement that each of these languages and their compilers would be extended also contributed to the motivation for simplicity. Each of the compilers would require modifications and partial rewrites in order to create a new compiler for the next language on the tree. Therefore everything must be done in the design to support the writing of correct and readable code and to allow for modifying parts of the compiler without affecting other parts. To this end, SIMPL-X was designed for writing programs that conform to the standards of structured programming and modular design. Thus, the statement structure consists of a minimal set of standard structured control statements. There are no goto's in the language but there are controlled exits from loops.

To help insure the ease of extension to the language, general operational semantic models of the base language were defined in VDL [19] and HGL [20]. A basic scheme for language design was broken into seven extensible categories: a set of data types, a set of operations on those types, a set of data structures, a set of control structures and consistent, modular and well-defined syntax, semantics and translation scheme. An attempt was made to define clean, efficient interfaces between each of the categories and the subelements within them. For example, several global decisions, such as the enforcement of strong typing, were made. The base language features were then designed within this overall structure. This permitted a kind of top-down view of the semantics of the languages even though the family was built in a bottom-up way.

Because of the need to write compilers in the language, several special features were included, such as the ability to read and write files and a set of high-level primitives for generating object code. These, together with the features given earlier, make the language quite effective in writing compilers.

Another important motivation was to make the languages and their compilers as transportable as possible. It was felt that the ability to write transportable software and move the family of languages onto other machines was well worth whatever effort was expended in its behalf. The standard conflicts between efficiency and machine independence arose, however, and a variety of decisions were made in order to achieve a reasonable balance. Since SIMPL-X is typeless (only integer data type), the language is word-size dependent. This problem becomes especially acute

when text manipulation applications are involved, such as the writing of a compiler, and the various computers store a different number of characters per word. This problem can be minimized if character strings are introduced into the language. There are also several operators, such as the bit, partword, and shift operators, which are highly word-size dependent and therefore not machine independent. These operators were put in for efficiency, and their bad effect can be minimized if they are sparingly and carefully used.

It was primarily the problems of transportability that led to the design of the first extension to SIMPL-X, SIMPL-T, and allowed us a first test of the extensibility of the base language and compiler.

The salient language features of SIMPL-T are

1) All the features of SIMPL-X.

2) String and character data types. Strings are of variable length with a declared maximum. The range of characters is the full set of ASCII characters.

3) A set of string operators which include concatenation, the substring operator, an operator to find an occurrence of a substring of a string, and the relational operators defined on the lexicographical ordering.

4) There is a character case statement.

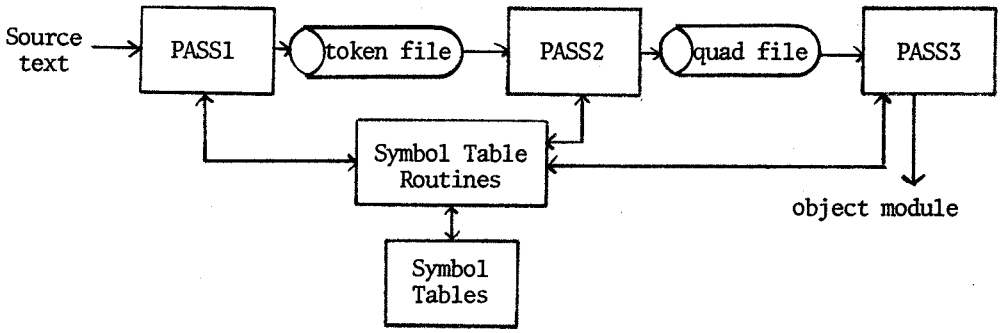
5) Strong typing is imposed and there are intrinsic functions that convert between data types.

6) For each data type, the only data structures are arrays.

7) There is the facility for interfacing with other languages.

The inclusion of strings and characters data types in SIMPL-T permits the writing of transportable text manipulation programs. Thus, transportable software can be written in SIMPL-T, though it need not be.

As stated earlier, the core compiler design is motivated by the goals of extensibility, transportability and generating efficient code. It consists of three separate passes and a set of symbol table routines. Pass 1 is basically a high-powered scanner which reads source text, outputs a token file and builds the symbol table. The symbol table is always referenced through a set of symbol table routines. Pass 2 is the parser. It handles a reduced grammar since the declarations were already processed by Pass 1. The parser is syntax-directed, and it outputs a file of high-level quadruples (quads). The third pass is the code generator which processes the quad file and outputs an object module.



The extensibility of the compiler is achieved primarily through the modular and structured design of the compiler which minimizes the problems of modification by requiring that changes be made in a small and well-defined number of places. In particular, the scanner is essentially modular enough that most extensions can easily be made by adding new cases or modifying old cases of the relevant case statements. New keywords or operators can simply be added to the appropriate tables and new tokens can be generated. Symbol table descriptors have several empty bits that can be used as needed for

the extensions.

The parser is syntax-directed and can handle a variety of different parsing schemes in a hierarchical fashion. The grammar can be partitioned into subgrammars in which the start symbol of one subgrammar is a terminal symbol in one or more of the others. In the compiler for SIMPL-X there are two major such modules. The top module uses a sort of SLR(1) characteristic state diagram, encoded as a case statement, which parses everything down to the statement level, excluding such things as expressions and argument lists. A second module uses an operator precedence scheme to parse expressions. In the top module expressions are essentially handled as terminal symbols which effect a class on the expression handler. There is also a separate module for argument lists.

The quads output by the parser are high level and yet general enough to permit the addition of several new control structures without introducing new quads and thus without effecting the code generator at all.

For example, the quads generated for a while statement of the form:

while <expression> do <statementlist> end

take the form:

while
quads for the <expression>
whiletest
quads for the <statement list>
endwhile

Now assume that we wish to extend the language to include a for statement of the form:

for <var> := $<expr>_1$ to $<expr>_2$ by $<expr>_3$ do <statement list> end

Then the compiler may be extended by including the following changes. The

scanner needs to accept the for symbol as a reserved word. This is accomplished by adding for to the symbol table as a reserved word. The parser must be made to recognize the for symbol and take a new action. This involves adding a new set of independent states to the characteristic state diagram which in practice amounts to adding a set of cases to the relevant case statement in order to generate the appropriate set of quads, which in this case might define the semantics of the for statement as follows:

```
*   quads for <expr>1
*   quads for <expr>2
*   quads for <expr>3
*   quad for <var> := <expr>1
    while
    quads for the expression <var> ≤ <expr>3
    whiletest
    quads for <statement list>
*   quad for <var> := <var> + <expr>2
    endwhile
```

The marked quads distinguish those that must be generated in addition to the standard while statement quads. These use only standard quad operators already defined. In this case the code generator need not be touched.

It should be noted that this particular form of extension to the language does not necessarily take advantage of the better code that can be generated for a for statement. To do that, new for quads can be defined and a fix can be made to the code generator to process the for statement more efficiently.

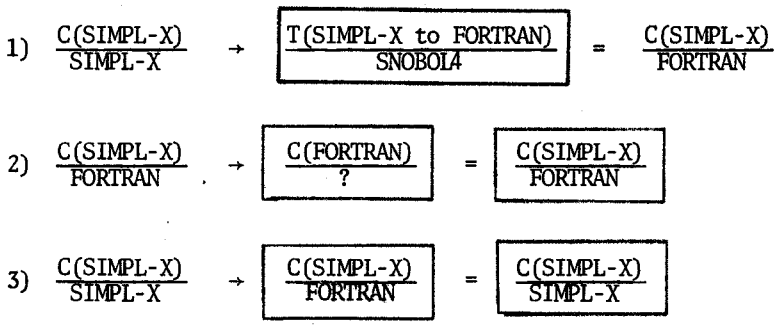
Suppose on the other hand, a new data type is being added to the language. This lower-level extension would require the addition of some new quads and various modifications to the code generator. A look at the cost of adding a new data type is given in the next section when the addition of reals (SIMPL-R) to SIMPL-T is discussed.

The goal of transportability for the compiler is also supported by its modular and structured design. To transport the compiler onto a new machine essentially requires the rewritting of the code generator along with the appropriate set of system library routines. The quads are very high level and machine independent and therefore transcend the specific idiosyncracies of any particular instruction set. Thus, if one neglects for the moment the problem of word size, everything up to code generation is at least theoretically machine independent. In fact, the basic top-down design for the existing code generator is modular enough that for a large number of machine architectures, many of the code generation routines may be used and only the appropriate set of actual instruction generation and register allocation routines need be rewritten for the new machine.

The main problem with transportability arises when it comes to word size. This problem can be minimized by using variable length character strings to represent textual data, letting all bit strings of information be packaged into multiples of some "transportable" word size (16 bits in this case) and using machine dependent operators sparingly and carefully. The last two techniques were used in the SIMPL-X compiler; all three were used in the SIMPL-T compiler. It is the SIMPL-T compiler which achieves the high level of transportability required by the system, and is being used as the base language which is presently being transported onto a CDC 6600 and an IBM 360. The SIMPL-X compiler is less transportable when being moved onto a machine which stores a different number of characters per word. There would be similar types of problems to transporting FORTRAN programs which do text manipulation. However, it is worth noting that the

SIMPL-X compiler was transported to a PDP-10 (which has the same word size as the UNIVAC 1108) even though the resulting compiler was rather unorthodox since it used UNIVAC 1108 character encoding for its own internal character definition.

The compiler for SIMPL-X was written in SIMPL-X. In order to get this first compiler in the family running, an initial bootstrap process was devised which involved the writing of a SIMPL-X-to-FORTRAN translator in SNOBOL4. This particular bootstrap process is outlined in Figure 2, where $T(L_1 \text{ to } L_2)$ denotes a program to translate from language L_1 to language L_2. The SIMPL-X compiler written in SIMPL-X was then run through this translator producing a SIMPL-X compiler written in FORTRAN. The SIMPL-X compiler written in SIMPL-X is then run through the executable form of SIMPL-X compiler written in FORTRAN producing an executable SIMPL-X compiler written in SIMPL-X. The motivation for this particular form of bootstrap as well as further details of its implementation and use can be found in [21].



The Initial Bootstrap

Figure 2

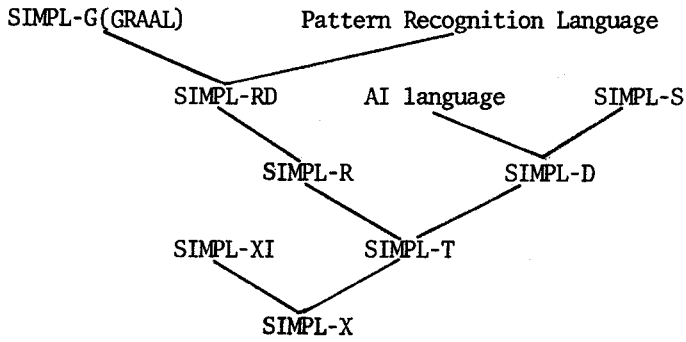
The SIMPL-T compiler was bootstrapped from SIMPL-X using the basic process outlined in Figure 1. Although the basic overall SIMPL-X compiler design was kept the same, some redesign was done at lower levels of the design so that this new compiler would benefit from the experience we gained in writing the transportable, extendable compiler for SIMPL-X. Because of this partial redesign effort, the SIMPL-T compiler is not a good example of the effort involved in bootstrapping a new compiler. A better view of this process might be gained from analyzing the extension of SIMPL-T to SIMPL-R which is discussed in the next section.

Because SIMPL-T and the SIMPL-T compiler would be the transportable base language and compiler for the family, the SIMPL-X-to-FORTRAN translator was modified to a SIMPL-T-to-FORTRAN translator. This supplied a transportable bootstrap for the family of languages.

The goal of producing relatively good object code is accomplishable on two levels--through an optimizer and through an efficient code generator. The separation of passes two and three of the compiler and the high-level nature of the quads permit the inclusion of a separate, transportable code optimization pass on the quads. Since the code generator is not transportable, the development of a generator for efficient code depends on the particular implementation. The code generator for the UNIVAC 1108 is organized in a top-down manner which permits a large amount of optimization of code and does generate fairly efficient code itself.

IV. Members of the SIMPL Family

Each new language in the family is designed as an extension to the base languages, SIMPL-X or SIMPL-T. The compiler for each new member of the family is an extension to the compiler of another member of the family. The tree structure representation of the family is given in Figure 3. Essentially the root of each subtree is a subset of the languages representing nodes on that subtree. Thus, SIMPL-T is an extension to SIMPL-X; SIMPL-R and SIMPL-D are extensions to SIMPL-T, etc.



SIMPL Family

Figure 3

We will now give a brief overview of the languages and compilers in the family other than SIMPL-X and SIMPL-T.

SIMPL-R

SIMPL-R was developed in order to handle standard mathematical-type problems of the type handled by FORTRAN. It is essentially an extension to SIMPL-T which includes the real data type. Along with real constants and

variables it incorporates several new operators into the language including the real arithmetic operators (+, -, *, /, **), the relationals on reals, and some special operators such as absolute value, sign, etc.

The strong typing of SIMPL-T was carried into SIMPL-R but integers were considered to be a subset of the reals and an integer coerces to real when in a binary operation with a real operand. A complete library of special functions, sin, cos, etc., was also included. Reals like all other data types may be stored in typed arrays.

The SIMPL-R compiler was bootstrapped from SIMPL-T using Steps 1 and 2 for the process defined in Figure 1. This yielded a SIMPL-R compiler written in SIMPL-T. Some detail of the modifications required and the time involved in completing the process will now be given because it is felt that the results are interesting.

Basically, the changes made to the SIMPL-T compiler can be summarized as follows:

The symbol table was modified (mod) by adding to it several new keywords and intrinsic names and some new flags were set in existing fields. A new hash procedure for real constants was added to the set of symbol table routines (U).

Pass 1 was modified by including the recognition of special dot operators, such as .ABS. for absolute value, and the exponential operator (**) ; the modification of the constant scanner to recognize real constants; and an assortment of special cases. Most of the modifications involved a fix (mod) and the addition (U) of some new routines to handle the new case.

Pass 2 modifications involved mainly the expression parser although some modification had to be made to the procedure call parser because of coercion of integer to real.

Pass 3 contains eight separately compiled modules and changes had to be made in four. In one it amounted to the changing of a flag. In another it required allocating core for reals, which involved almost duplicating an already existing routine. In yet another it involved a fix to some utility routines to cover reals. The only major modification was in the actual code generator for expressions which involved modifying the old +, -, *, / procs to handle the joint operations of integer and real and adding new procs for the new operators.

The actual construction of the SIMPL-R compiler was performed by someone who was not in anyway involved in the SIMPL family project until that point. He worked between half and full-time on the project for three months and at the end of that time had a working SIMPL-R compiler written in SIMPL-T (plus about 150 lines of assembly code which was used for turning string representations of real constants into real constants, doing constant folding, etc., since SIMPL-T has no facilities for handling reals). The fix-up of the symbol table and Pass 1 took about six weeks. This included the time required to become familiar with the language and the compiler. There was no documentation for the compiler other than a well-commented listing of the compiler itself. The modification of Pass 2 took about two weeks and Pass 3 and the library took about five weeks. It is worth noting that the implementor was a good programmer but was not very well-versed with the 1108 system or assembly code which also slowed him down.

It is not clear whether all extensions will go this well. Certainly the similarity between real and integer data types was of some advantage. However, it is felt that the ease with which the implementor was able to step into the project and find the right places to make the changes and add the routines support the view that a compiler can be designed to be easily extended.

SIMPL-XI

The motivation for SIMPL-XI was to develop a high-level language that would do efficient systems programming on the PDP-11. Since it was decided that an escape to assembly language or calls to assembly language routines should be discouraged, extensions were made to SIMPL-X which enabled the programmer to have access to the full capabilities of the hardware of the machine. Extended facilities for managing the hardware include the ability to address real memory, control I/O devices, process interrupts, issue supervisor call instructions, alter the state vector of the machine and control the virtual memory.

The SIMPL-XI compiler runs on the UNIVAC 1106 and 1108 computers. The development of the SIMPL-XI compiler from the SIMPL-X compiler proceeded as follows. Pass 3 (the code generator) of the SIMPL-X compiler was rewritten to produce a cross-compiler that executes on the UNIVAC machines and produces code for the PDP-11. This generated a SIMPL-X cross-compiler for the PDP-11. Then this SIMPL-X cross-compiler was modified as described in Figure 1, **Steps 1 and 2**, to generate a SIMPL-XI cross-compiler written in SIMPL-X.

This cross-compiler was not bootstrapped to the PDP-11 because the machine only had 8K of memory. However, further development of the PDP-11 system includes another 16K of memory and a disk which should allow enough storage to bootstrap SIMPL-XI onto the PDP-11. The present plan is to redevelop SIMPL-XI as an extension to SIMPL-T and the SIMPL-T compiler to take advantage of the string and character processing as well as the more transportable and extendable features of the SIMPL-T compiler.

This completes the list of languages which are presently fully defined and whose compilers are currently working. We will now discuss briefly those languages which are currently under development or in the planning stages.

SIMPL-D

All the presently defined and implemented languages in the family are thus far limited with respect to data structures, supporting only arrays. Because most of the languages that are to be defined will require more data structuring facilities, SIMPL-D is being designed and implemented both as a test language and as a core design for the data structuring facilities that will be placed in the other languages. The present design includes the (1) definition of records, which are defined as data structures whose subelements are any of the data types, arrays, or other records; (2) the inclusion of a data definitional facility so that new data structures, their accessing mechanisms and operators may be defined by the user; (3) some storage allocation and deallocation primitives; and (4) some restricted form of pointer variables.

These data structuring facilities will either be used explicitly in several of the succeeding languages or expanded to meet their particular needs and probably used in the building of their compilers.

The compiler for SIMPL-D is being built as an extension to the SIMPL-T compiler.

SIMPL-RD

The motivation for SIMPL-RD is to develop a transportable language for dealing with large-scale numerical applications problems that support the relatively machine-independent compactification of data storage. The language would essentially combine all the features of SIMPL-R and SIMPL-D and the compiler would basically be the union of the compilers for SIMPL-R and SIMPL-D.

SIMPL-G

SIMPL-G will be a graph algorithmic language for describing and implementing graph algorithms of the type primarily arising in applications. It will be the redesign of the GRAAL language by redefining it as an extension to SIMPL-RD and adding several new features not available in the present definition of GRAAL [4]. The language will be built on a set-theoretic model of graph theory which allows for considerable flexibility in the selection of the storage representation for different graph structures. A high-level design of the set and graph aspects of GRAAL as they will appear in SIMPL-G may be found in [22].

SIMPL-S

SIMPL-S is being proposed as an extension to SIMPL-D to serve as a high-level systems programming language. As with SIMPL-XI, its goals will

be to encourage the use of a high-level language for systems programming by giving the user access to the capabilities of the hardware in a machine-independent manner. The data structure definitional facility at the lower level will be used to help define efficient, machine-dependent data structures beneath the high-level algorithms. It will include the expansion of several of the facilities in SIMPL-D, including such features as an address operator which returns the address of a variable and an indirection operator.

The designs of the artificial intelligence language and pattern recognition language have not yet been proposed. However, they will probably be defined as extensions to SIMPL-D and SIMPL-RD, respectively.

In addition to the languages themselves, supporting software is being designed and implemented to facilitate the transportability, extendability, efficiency, and effectiveness of the SIMPL family.

A macro preprocessor has been defined and implemented [23]. The major benefits of this preprocessor are that it aids in the development of machine-independent compilers and programs and allows for experimentation with operators and structures before they are actually implemented in the language. The macro preprocessor permits simple string substitution, compile time variables, macro definitions with several parameters, and compile time assignment, "if", "while", and "call" statements. The macro preprocessor is written in SIMPL-X; it is being rewritten in SIMPL-T.

A machine-independent optimizer [24] has been written in SIMPL-X which handles many of the rudimentary optimizations on the intermediate

code (quadruples). A new, more ambitious, machine-independent optimizer is being planned that will handle the optimization of procedure calls and global variables.

As mentioned earlier, two bootstraps were written. The original bootstrap for SIMPL-X was used to get the base compiler running on the UNIVAC 1108. A second transportable SIMPL-T bootstrap has been written and is being used to transport SIMPL-T onto a CDC 6600 and an IBM 360.

An automatic documentor-indentor is being implemented similar to the one suggested in [25]. The idea is to use the phrase structure of a program to define the structure of a formal documentation for that program. This syntax directed documentor would greatly enhance the ease of documentation and understanding of programs written in SIMPL-T.

V. Conclusion

The primary goals for the SIMPL family were that the languages should be simple, well-defined, easy to extend, transportable, and capable of writing compilers. The goals for the compilers were that they be extendable, transportable, and generate relatively efficient code. It is difficult to quantify our success or failure in achieving these goals. Some comments regarding the project are worth making.

Anyone knowing any programming language can be taught SIMPL in one or two hours.

A good deal of time was spent modeling the various features using operational semantic models [19,20]. It is felt that this modeling was of great benefit in contributing to the simplicity and consistency of the

semantic design. It provided a top-down view of the semantic design of the family that made the bottom-up construction easier since there was a good general framework laid out for the extensions. However, certain aspects of the syntactic design have been much more difficult to organize in this way, and there is some fear of problems in this area, such as not reserving the proper symbols for use in later extensions, etc.

Both SIMPL-X and SIMPL-T have been used quite widely in all aspects of the computer science curriculum at the University of Maryland at College Park. This includes its adoption in introductory courses on programming to undergraduate and graduate courses on programming languages, data structures, compiler writing, systems, certification of programs, and semantic modeling. It is also being used in the curriculum at the University of Maryland at Catonsville. Responses from students on questionnaires have been quite favorable.

SIMPL-T is being used by the Defense Systems Division, Software Engineering Program Transference Group at Sperry Rand as the language for building their translator system.

Extensions to languages in the family have been straightforward due to the simplicity and consistency of the basic design. The evidence concerning extensions to the compiler both for SIMPL-R and what appears to be involved for SIMPL-D is quite reassuring.

The goal of transportability of the languages and the compilers is still to be tested. The only SIMPL compiler currently generating code for another machine is the SIMPL-XI cross-compiler, but this is not a real test of transportability. Efforts, however, are underway to transport SIMPL-T

onto an IBM 360 and a CDC 6600.

With respect to the generation of efficient code, several large programs which were written in FORTRAN V for the UNIVAC 1108 and considered to execute quite efficiently were hand-translated into SIMPL-R. When both sets of programs were run on some randomly-generated sets of data, the unoptimized SIMPL-R programs actually executed about ten percent faster than the optimized FORTRAN V programs.

One test of the reliability of the software was demonstrated by a bug contest that was organized for the SIMPL-T compiler. In order to facilitiate the debugging of the compiler, the compiler was released early and prizes were awarded for bugs found. Only a handful of bugs were found.

There are several people who were involved in the SIMPL family project whose work has not already been acknowledged. Albert J. Turner, who did all of the programming of the compilers for SIMPL-X and SIMPL-T and was involved in most of the design efforts; John McHugh, who extended the SIMPL-T compiler to SIMPL-R; J. Michael Kamrad and A. Bruce Carmichael, who programmed the SIMPL-X and SIMPL-T bootstraps, respectively; Hans Breitenlohner, who wrote the execution time monitor and provided assistance in trying to interface to the 1108; and C. W. Barth, who is programming the automatic indentor documentor.

Appendix

Example: SIMPL-X Program

```
PROC SORT (INT N, INT ARRAY A)

/* THIS PROCEDURE USES A BUBBLE SORT ALGORITHM TO SORT THE
    ELEMENTS OF ARRAY 'A' INTO ASCENDING ORDER.  THE VALUE
    OF THE PARAMETER 'N' IS THE NUMBER OF ITEMS TO BE SORTED.  */

INT SORTED, /* SWITCH TO INDICATE WHETHER FINISHED */
    LAST,   /* LAST ELEMENT THAT NEEDS TO BE CHECKED */
    I,      /* FOR GOING THROUGH ARRAY */
    SAVE    /* FOR HOLDING VALUES TEMPORARILY */

IF N>1
  THEN /* SORT NEEDED */
    SORTED := 0 /* INDICATE NOT FINISHED */
    LAST := N-1 /* START WITH WHOLE ARRAY */

    WHILE .NOT. SORTED
      DO /* CHECK CURRENT SEQUENCE FOR CORRECTNESS */
        SORTED := 1 /* ASSUME FINISHED */
        I := 1      /* INITIALIZE ELEMENT POINTER */

        WHILE I <= LAST
          DO /* COMPARE ADJACENT ELEMENTS UP TO 'LAST' */
            IF A(I-1) > A(I)
              THEN /* OUT OF ORDER */
                SAVE := A(I)      /* INTERCHANGE */
                A(I) := A(I-1)    /* A(I) AND    */
                A(I-1) := SAVE    /* A(I-1)      */
                SORTED := 0       /* MAY NOT BE FINISHED */
              END
            I := I+1
          END /* LOOP FOR COMPARING ADJACENT ELEMENTS */
        /* A(LAST),..., A(N-1) ARE NOW OK */
        LAST := LAST -1
      END */ LOOP FOR CHECKING CURRENT SEQUENCE */
  END /* IF N>1 */

/* END PROC 'SORT' */
```

Example: SIMPL-T Program

```
/* THIS PROGRAM REPLACES ALL SUBSTRINGS BETWEEN '/*' AND '*/' BY
   BLANKS */

STRING INPUT [80]
INT PTR1, PTR2

PROC REMOVECOMMENTS

WHILE .NOT. EOI
  DO
    READ (INPUT)
    PTR1 := 1           /* INITIALIZE FOR SEARCH */
    WHILE PTR1 <> 0
      DO  /* REMOVE SUBSTRINGS */
        PTR1 := MATCH (INPUT, '/*')
        IF PTR1 <> 0
          THEN /* FOUND BEGINNING */
            PTR2 := MATCH (INPUT, '*/')
            IF PTR2 > PTR1 + 1
              THEN /* FOUND END (AFTER BEGINNING) */
                INPUT [PTR1, PTR2 - PTR1 + 2] := '' /* BLANK IT OUT */
              END
          END
      END
    WRITE (INPUT, SKIP)
  END
/* END PROC 'REMOVECOMMENTS' */

START REMOVECOMMENTS
```

References

1. Basili, V. R., SIMPL-X, A language for writing structured programs, University of Maryland, Computer Science Center, Technical Report TR-223 (1973), 43 pages.

2. Basili, V. R., and Turner, A. J., SIMPL-T, A structured programming language, University of Maryland, Computer Science Center, Computer Note CN-14 (1974), 91 pages.

3. Hamlet, R. G., and Zelkowitz, M. V., SIMPL systems programming in a minicomputer, submitted to IEEE COMPCON '74, Washington, D.C.

4. Rheinboldt, W. C., Basili, V. R., and Mesztenyi, C. K., On a programming language for graph algorithms, BIT 12, 1972, 220-241.

5. McHugh, J., and Basili, V. R., SIMPL-R and its application to large sparse matrix problems, University of Maryland, Computer Science Center, Technical Report TR-310 (to appear).

6. Kanal, Laveen, Interactive pattern analysis and classification systems: A survey and commentary, Proceedings, IEEE, Oct. 1972.

7. Schwartz, J., On Programming: An interim report on the SETL project, New York University, Courant Institute of Mathematical Sciences (1973).

8. Wells, M. B., Elements of Combinatorial Computing, Pergamon Press, Oxford (1970).

9. Feldman, J. A., et al., Recent developments in SAIL - An ALGOL-based language for artificial intelligence, Proceedings FJCC, AFIPS Press, Montvale, N. J., 1193-1202 (1972).

10. Sussman, G. J., et al., MICRO-PLANNER reference manual, MIT Artificial Intelligence Laboratory, Memo No. 57.1 (April 1970).

11. Crespi-Reghizzi, S., and Morpurgo, R., A language for treating graphs, Comm. ACM, 319-323 (May 1970).

12. Schuman, Stephan (ed.), Proceedings of the International Symposium on Extensible Languages, SIGPLAN Notes (Dec. 1971).

13. Dahl, O., Myhrhang, B., and Nygaard, K., The SIMULA 67 common base language, Norwegian Computing Center, Oslo, Publication S-22 (1970).

14. Liskov, B., and Zilles, S., Programming with abstract data types, Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974).

15. Cheatham, T., et al., On the basis for ELF - An extensible language facility, Proceedings FJCC 33, 2, AFIPS Press, Montvale, N. J. (1968).

16. Irons, E. T., Experience with an extensible language, CACM, 31-39, (Jan. 1970).

17. Feldman, J. A., and Gries, D., Translator writing systems, CACM (Feb. 1968).

18. Scowan, R. C., An application of extensible compilers, Proceedings, International Symposium on Extensible Languages, SIGPLAN Notices (Dec. 1971).

19. Lucas, P., et al., Method and notation for the formal definition of programming languages, IBM Laboratory, Vienna, TR 25.087 (1968).

20. Basili, V. R., and Turner, A. J., A hierarchical machine model for the semantics of programming languages, Proceedings, Symposium on High Level Language Computer Architecture, ACM (November 1973).

21. Basili, V. R., and Turner, A. J., A transportable extendable compiler, University of Maryland, Computer Science Center, Technical Report TR-269 (1973); Software - Practice and Experience, Vol. 5, 269-278 (1975).

22. Basili, V. R., Sets and Graphs in GRAAL, Proceedings of the ACM, November 1974, pp. 289-296.

23. Verson, J. A., and Noonan, R. E., A high level macro processor, University of Maryland, Computer Science Center, Technical Report TR-297 (1974).

24. Zelkowitz, M. V., and Bail, W. G., Optimization of structured programs, Software Practices & Experiences 4, 1 (Jan.-March 1974).

25. Mills, H., Syntax directed documentation for PL/360, CACM 13, 4 (April 1970).