CGTM #100
David Gries
September 10, 1967

COMPILER IMPLEMENTATION SYSTEM

This is the first of a series of CGTM reports which will describe and document the compiler implementation system project. It expresses the overall goals of the project.

1. The subsystem.

When implementing any compiler much of the work consists of imbedding the compiler in the machine operating system. Somebody must learn all the details of the system in order to program the linkages with the system correctly. Many times the system itself must be changed, simply because the system writers did not allow for the addition of compilers.

Part of the proposed compiler-compiler project will be to design and implement a small subsystem to handle these operating system linkages automatically. With a few control cards, or control commands, it should be possible to add a new compiler or delete or change an existing one. It should be easy to indicate that certain runtime subroutines must be present with each object program exectuion. It should be possible to keep on secondary storage any number of compilers and source programs both in symbolic form and as translated programs ready to be executed. The compiler writer will thus be relieved as much as possible from having to know details about the main operating system.

The subsystem will be imbedded in OS 360 and later probably in TORTOS. No specific details about it have been worked out yet.

2. Compiler implementation system.

This term denotes the language (actually the set of languages) in which one writes a compiler. Most compilers are written in the assembly language of some machine. That is, the syntax and semantics are expressed in the assembly language. The assembly language program representing the compiler is then assembled, producing the machine language version of the compiler. Thus, the assembler should be called a "compiler-assembler", since it is used to assemble compilers. One does not call it a "compiler-assembler" because the assembly language is so close to the machine language that the two are considered to be equivalent, or the same. Unfortunately, such a compiler is very unwieldy; it takes a long time to write, it is not readable by most people, it is not a very natural description of the source language in question, and consequently is hard to change. Also, the compiler has then been written for one particular machine only.

What we want to do, then, is to design and implement a language for writing compilers, just as ALGOL and FORTRAN have been designed and implemented for programming scientific algorithms. We want a compiler-compiler and not a compiler-assembler.

Our compiler writing system will be based mainly on Feldman's FSL system ([1], [2]). The following introductory discussion is taken partly from Feldman's article [2].

The problem is to develop a single program which can act as a translator for a large class of compilers. To solve this so-called

compiler-compiler problem, one must find appropriate formalizations of the syntax and semantics of formal languages.

The formalization of semantics for some language, L , will involve representing the meanings of statements of L in terms of an appropriate meta-language. The meanings of statements in the meta-language are assumed to be known. As we saw above, one example of a semantic meta-language (and also a syntax meta-language) is the assembly language of a computer. We will develop a more natural one.

Suppose now that the whole system has been implemented. When a compiler for some language, L, is required, the following steps are taken. First the formal syntax of L, expressed in a syntactic meta-language, is fed into the syntax loader. This program builds tables which will control the recognition and parsing of programs in the language L. Then the semantics of L, written in a semantic meta-language, is fed into the Semantic Loader. This program builds another table, this one containing a description of the meaning of statements in L. Finally, everything to the left of the double line in Figure 1 is discarded, leaving a compiler for L.

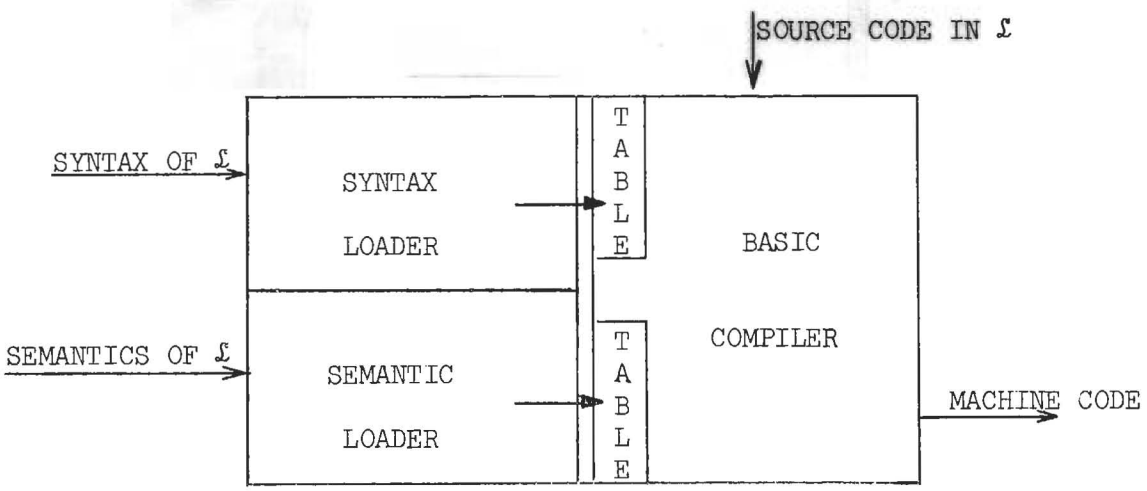


Fig. 1. A compiler-compiler

The resulting compiler is a table-driven translator based on a recognizer using a single pushdown stack. Each element in this stack consists of two machine words -- one for a syntactic construct and the other holding the semantics of that construct. When a particular construct is recognized, its semantic word and the semantic table determine what actions the translator will take.

The specification and implementation of the syntax and semantic meta-languages will be the content of future notes. Before elaborating more on the structure of a compiler (Fig. 1 describes only a one-pass compiler), we give a few definite examples of the types of constructs one might have in the syntax and semantic meta-languages. This should give the reader a more concrete idea of just what is going on.

Suppose our syntax meta-language was essentially Backus Normal Form (which it won't be) and that we were writing a compiler for AIGOL. One syntax statement might be

<Declaration> ::= real <identifier> (1)

The meaning would be as follows: If "real <identifier>" is recognized at the top of the stack, then execute routine number 1 of the semantic meta-language, and replace "real <identifier>" in the stack by "<Declaration>". Semantic routine 1 might be

(1) Comment Enter the identifier appearing in the semantic word of the top stack element into the symbol table SYMB, as a real variable, assigning it the object program address given by the variable STORLOC;

ENTER (SYMB; STACK 1, REAL, STORLOC) ;

Comment increase the storage location pointer STORLOC;

STORLOC ← STORLOC + 1;

Similarly, the meaning of the syntax meta-language statement

<Term> ::= <Term>*<Factor> (2)

would be : if " <Term>* <Factor> " is recognized in the stack, replace it by <Term> and execute semantic routine 2. Routine 2 might be

(2) Comment generate the code for multiplication, the operands being specified by the semantic words of the first and third stack elements. Store a semantic description of the result in the first stack element;

CODE (STACK1 * STACK 3) → STACK1;

The above is by no means an exact description of constructions of the proposed syntax and semantic neta-languages. It is intended only to give a concrete exemple of the type of constructions that will be used.

In each case, a syntactic construct was recognized and reduced, and a corresponding semantic subroutine was executed to evaluate the construction. This is the basic idea behind the compiler-compiler. The syntax meta-language lets one describe in a natural way how a source program is to be parsed; the semantic meta-language has the constructs necessary to evaluate the recognized constructs easily.

The only trouble with the above description is that we are only able to write one-pass compilers. If any object code optimization is

to be done, or if a language is so complicated that a one-pass compiler is not sufficient, then the compiler-compiler would be of no use. Our system will therefore allow a much more complicated structure.

Before going into details about the structure of a compiler, there are three concepts which must be explained. They will occur throughout these notes, and are easily confused. When talking about translators or compilers, there are two separate phases in getting any source program to run: compile time - the time during which the source program is being compiled or translated and runtime - the actual execution of the translated source program (which will be referred to as the object program). When discussing compiler-compilers, a third phase is introduced - the compilation of a compiler itself. This we will call metacompile time. These three concepts tend to be confusing, and we state them again: metacompile time - the translation or compilation of a compiler; compile time - the translation or compilation of a source program by a compiler; runtime - the execution of a translated source program.

3. Structure of a compiler written in our system.

3.1 Passes

A compiler may consist of any number of passes. These may be interpreted as "core loads", which will be executed in an order which is fixed at metacompile time. Of course, at compile time it will be possible to skip certain passes, depending on the source program (for instance if a source program has an error in it, the pass which actually does the translating may be skipped). It is not necessary to have more than one pass in a compiler; one pass compilers may be written if desired. The determination of the number of passes in any compiler depends on the language to be translated and the particular way in which the compiler is going to be used. A compiler which is going to be used mostly by students should probably be a one pass compiler, since it must translate programs as fast as possible. A compiler which is to produce very efficient object programs must necessarily have more than one pass.

As an example, consider the following three pass compiler for ALGOL. Pass 1 scans the source program and produces a "symbol table", ordered by block structure of all declared identifiers and labels and a list of constants in the program. Pass 2 makes a complete syntax check of the source program, using the symbol table. If the program is correct, control is given to pass 3, which generates the object program.

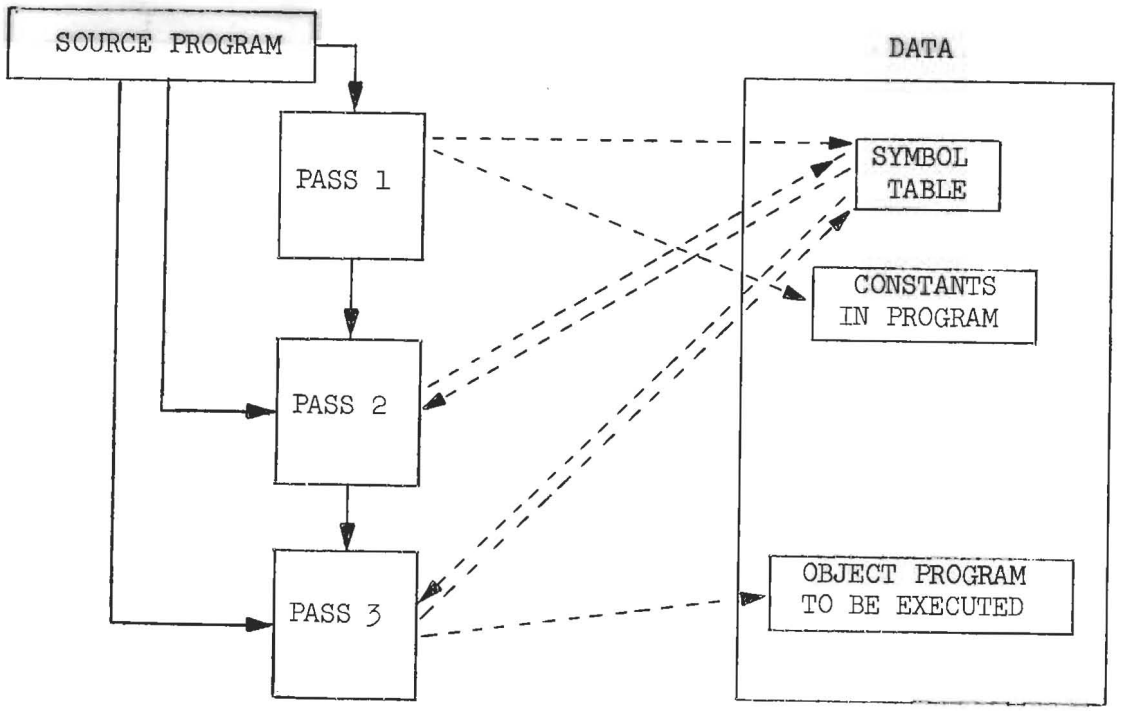


Fig. 2. Three pass compiler

The above diagram indicates that some data must be global to some or all of the passes, and possible also global to the object program. The language must allow this through some kind of global declaration.

3.2 Phase

Each pass consists of one or more phases. These phases are executed, in an order fixed at metacompile time, in the following manner. Phase 1 gets the next symbol from the source program and works with it until it has done all that it possibly can with it. It then turns the symbol over to phase 2, which works with it. And so on. When the last phase is finished, control is returned to phase1, which gets the next source program symbol. The process is then repeated. Of course, through the aid of a simple switch, it will be possible to "turn off" or "turn on" any phase at any time.

Why such a structure?

1) It should be relatively easy to implement.

2) In order to make the construction of a compiler simpler and cleaner, one may want to have different passes to check or translate different constructs of the language. Much bookkeeping and compile time is saved by calling these passes phases, thereby keeping both in memory at the same time, and just switching back and forth between them depending on the source program.

3) It is sometimes necessary to check certain constructs in more than one way. For instance, subscripts of array elements have to be checked for syntax errors, but in certain instances (if object program optimization is to be done), it may be necessary to check these subscripts for linearity in the loop variable of the surrounding FOR loop. Both can be done in one pass, but the overall idea of the syntax checker becomes clouded. It is better to have two passes. An even better solution would be to call the syntax checker phase 1, the linearity checker phase 2, and have phase 1 connect the linearity checker when a "[" is read. The two logically separate functions are then still separated, but the source program itself must only be read once and the two phases can share routines and data storage, saving time. Figure 3 contains a diagram of this.

The statement "SCAN", therefore, when executed in a phase, has the following meaning.

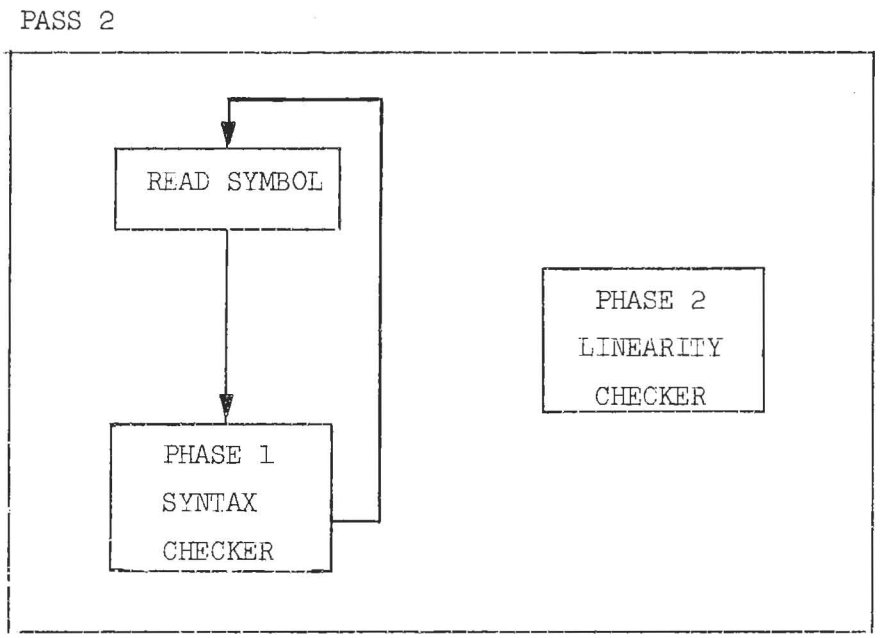
> Execute the next phase in the sequence (if "on"). If there are no more to execute, read the next symbol of the source program and execute the first phase in the sequence.

3.3 Syntax and semantic sublanguages.

Each phase may consist of two parts - a syntax program (written in the above mentioned syntax meta-language) and a semantic program (written in the semantic meta-language).

It will be necessary to allow phases which have no syntax program. For instance, a phase which just prints out error messages from a packed list does not need a syntax program - just a semantic part.

There are many techniques for parsing sentences of programming languages. The syntax meta-language will include constructs for implementing Feldman's production language (see [1], [2] and transition matrices (see [3])). Later, if time permits, other techniques, such as Wirth's precedence techniques [4], will be implemented. This will allow us to use the system as a practical tool for teaching compiler techniques.



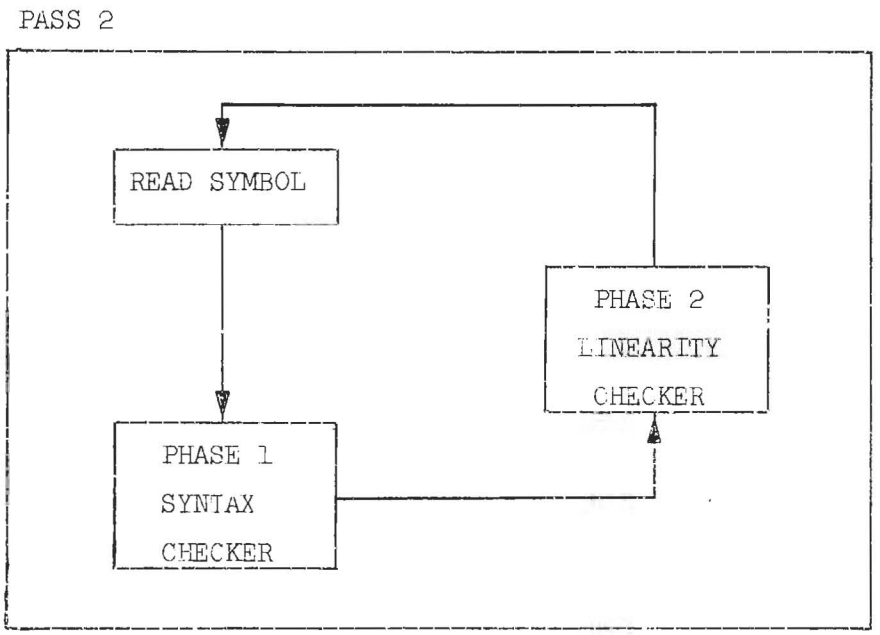


Fig. 3. Pass 2 at different points during compile time

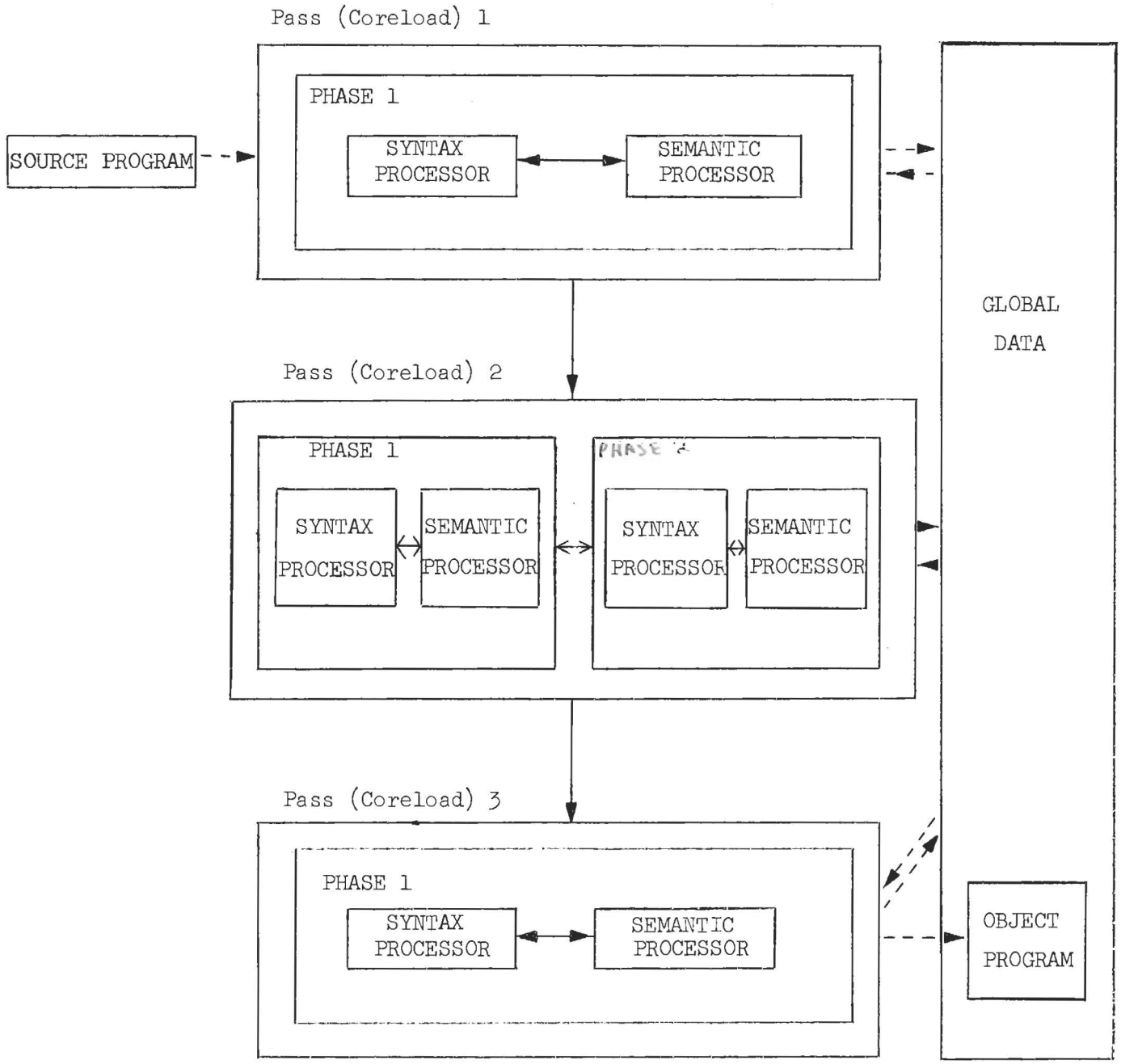


Fig. 4. Three pass compiler

4. Other uses for the compiler-compiler.

It should be possible to use the system for any algorithm which is dependent on evaluating constructs of a language which have been recognized. For instance, an on-line editor could be coded in the compiler implementation language. Its function is just to make up calls to subroutines which delete, insert and change files depending on the input commands. These commands must be parsed, however, in order to interpret them. If the semantic meta-language has the necessary constructs, it can be used. We would be happy to hear of other ideas for the use of the compiler-compiler, so that we could try to include the necessary constructs in the semantic meta-language.

REFERENCES

[1] Feldman, Jerome A., "A Formal Semantics for Computer Oriented Languages," Thesis, Carnegie Institute of Technology (May, 1964).

[2] _____, "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler." Comm. of the ACM 9 (January 1966), 3-9.

[3] Gries, D., "The use of transition matrices in compiling", Technical Report CS 57, Computer Science Department, Stanford University (March 17, 1967).

[4] Wirth, N.,and Weber, H., "Euler: a generalization of ALGOL, and its formal definition: part I" Comm. of the ACM 9 (Jan 1966), 13 - 25.