Technical Report CS74016-R

FIRST REPORT ON EPOS: ASPECTS OF GENERALITY AND EFFICIENCY IN PROGRAMMING LANGUAGE IMPLEMENTATION

Johannes J. Martin

November 1974

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061

ASPECTS OF GENERALITY AND EFFICIENCY IN PROGRAMMING LANGUAGE IMPLEMENTATION

Johannes J. Martin

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Va. 24061

1. Introduction

Composing and documenting voluminous programs is a difficult and time consuming task despite the multitude of existing programming languages. It is still the exception that a programmer finds a language that has all the properties he likes to have to his disposal for a given project. Further, many programmers show a very understandable reluctance toward learning a new notation since mastering all its inevitable pitfalls and idiosyncrasies frequently takes more time than programmers deem justifiable.

In this context, extensible languages are a promising branch on the tree of programming systems. These languages are supposed to permit the user to define his own notation and, hence, to create a language, custom tailored to his problem. The many advantages of such a system have been extensively discussed and praised by others [N71, W71, I70, S67, L66]. Only two points shall be added:

1) With the propagation of extensible systems, language design can be expected to become a new style of programming that will decisively improve the reliability and transparency of large programs.

2) For the academic world, extensible languages will make language design a better discipline to teach and especially to practice.

However, the extraordinary flexibility offered by the concept of extensible languages causes its own special problems. Used unwisely, it leads to slow compilations and inefficient code. These problems have been considered by some workers who offer the following measures as solutions:

1) restricting extension mechanisms to the very competent programmer by e.g. only providing a changeable (extensible) compiler rather than linguistic extension features [S71],

2) complementing the extension mechanisms by restriction mechanisms that permit the user to disable existing features and, hence, to freeze a language at some state of extension [W71, S70].

A different method, discussed by van Gils [G71], is derived from design principles of ALGOL 68. Although not specifically introduced for the sake of efficiency, this method seems to be most attractive for the purpose. By restricting extensions to certain non-insidious types, it not only helps efficiency but also promotes simplicity of rules and rather systematic error handling.

The generality/efficiency trade-off is the underlying theme for the following discussion of some design aspects of the programming system EPOS (Eine Programmiersprache Ohne Semantik), currently under development in the Computer Science Department of VPI&SU. This project has been greatly influenced by the works of van Gils [G71], Jorrand [J71], and by ALGOL 68 [L71, W69]. It is assumed that the reader is familiar with these publications as well as with the basic problems of compiler design.

2. On Defining Programming Languages

2.1 Conventional languages

The designer of a conventional programming language usually defines

1) a set of data objects, operations for these objects and control constructs, and

2) a suitable notation for the above components and rules for their assembly.

The notation rules form the grammar of the language; the data objects, operations and control constructs form the semantics.

To the author's knowledge, the grammars of programming languages ever formalized are context-free and, thus, definable by the Backus-Naur-Formalism (BNF). The precise description of the semantics can be accomplished by means of some hypothetical computer. The connection between the grammar and the hypothetical computer may then be established by a translation schema that maps syntactic elements of the grammar into sequences of instructions of the hypothetical computer (There are a number of translation schemata described in the literature, a good summary is found in Aho and Ullman [A73] who also furnish a comprehensive listing of the primary literature.).

Thus, we can define a conventional programming language as a triple

(1) $$L_c = (G, C, T)$$

where

G is a (context-free) grammar,

C a computer and

T a translation schema.

Being well known, methods for formalizing G, C and T will not be further discussed here.

2.2 Extensible languages

Extensible languages have been looked on as relatively small conventional base languages with additional mechanisms for specifying extensions. It seems, therefore, that the method of language definition for conventional languages, outlined above, should easily be applicable to extensible systems. A closer analysis reveals, however, that this is not true.

From the three components of the above definition only the hypothetical computer is equally useful for both, conventional and extensible languages as the basis for the description of the semantics whereas (i) the grammar and (ii) the translation schema cannot simply be transferred.

(i) For conventional languages, it is a priori possible to define a complete context-free grammar that permits the linear structure of the input text to be transformed into the hierarchical structure of the derivation tree. This transformation prepares

the final translation into object code by isolating and ordering the translatable phrases.

For extensible languages, only the grammar of the base language can be completely described whereas, for possible extensions, only the class of permissible grammars can a priori be determined.

(ii) A quite analogous argument holds for the translation schema.

Therefore, extensible languages founded on higher level base languages are 5-tuples

(2) $$L_{eb} = (G_b, H, C, T_b, U)$$

where

G_b is the grammar of the base language,

H the class of grammars that may be used for extensions,

C a hypothetical computer,

T_b the translation schema of the base language,

U the class of translation schemata that may be used for extensions.

The class H of permissible grammars decisively influences the complexity of the compiler. It is well known that parsers for arbitrary context-free grammars are much more complex and take more time and space than a parser for, say, operator precedence grammars. In section 3, we will discuss criteria for selecting a grammar class in more detail.

Note: One might suspect that the classes H and U should not be mentioned in the definition because both seem to be implicitly defined by the extension mechanisms contained in the base language. However, there are two different classes of grammars as well as of translations:
1) the class of grammars (translations) that can be specified,
2) the class of grammars (translations) that can be handled by the parser (code generator)
H and U represent the classes that can be handled.

2.3 Significance of the base language

We will now consider a simplification of the definition schema for extensible languages. The very nature of extensible languages makes it possible to use the extension mechanism not only for adding new features to the language but also for introducing new notations for existing (base language) features. Thus, if a user does not like the way declarations, operations or control structures are denoted in the base language he may tie some or all of these features to his own notations (i.e. redefine the features by means of the extension mechanisms).

Because of this flexibility, the usual criteria for sound language design like conciseness and natural or traditional denotation are of very little importance for base language design. Thus, pursuing other qualities like simplicity, transparency, generality and adaptability, we may simplify and generalize extensible systems by uniting the definition of the base language with the definition of the hypothetical computer i.e. by declaring the instruction set of the hypothetical computer to be the base language of the extensible system.

Note: In order to make the definition of declaration statements possible, the instruction set of the hypothetical computer must, of course, contain instructions for storage allocation and rather sophisticated bookkeeping.

Because of the unification of the base language and the hypothetical computer the definition schema again becomes a triple:

(3) $$L_e = (H, C, U)$$

where

H is a class of grammars,

C a hypothetical computer and

U a class of translation schemata.

The general structure of the system is depicted in fig. 1. The general compiling algorithms update and interrogate a data base that contains the specific information about the language used and the program processed.

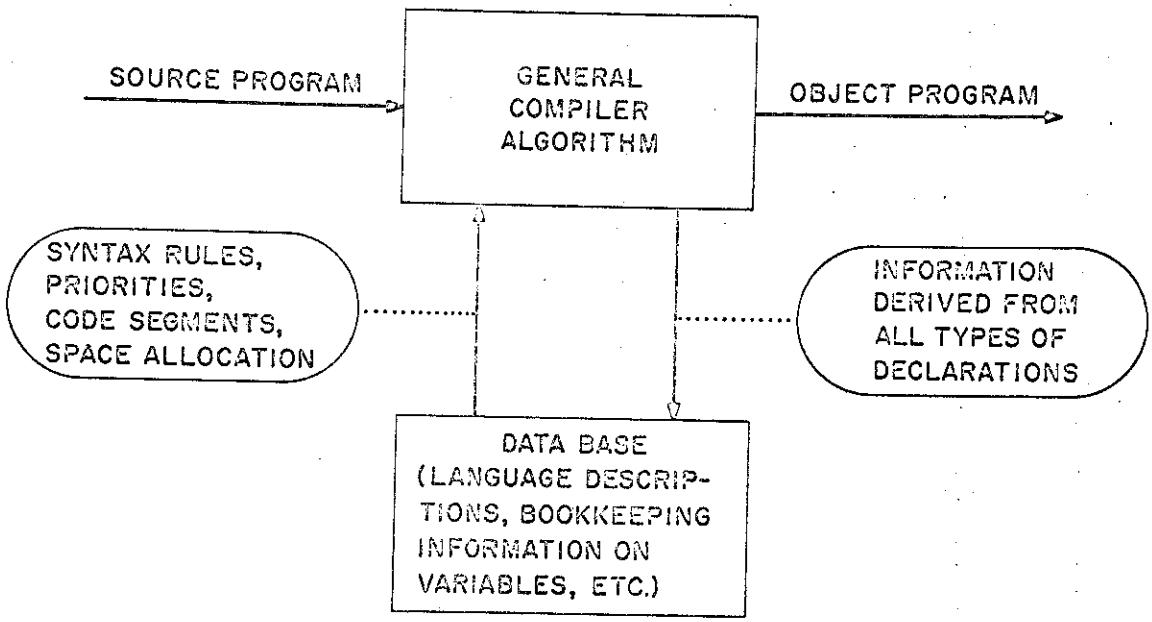

FIGURE 1 Overall structure of the compiler

Besides simplicity and transparency, the concept offers the programmer the opportunity to resort at any time to the low level base of the system and, hence, insure that critical code is implemented in the most efficient manner. Further, new operating system functions can be made available by adding the necessary instructions to the instruction set of the hypothetical computer.

3. The Choice of the Grammatical Class

We should now discuss the criteria for choosing the classes for the grammar and the translation schema and describe the hypothetical computer and the data base. Because of the limited space we shall, however, be content with an analysis of the first problem, i.e. choosing the grammar class. An in depth discussion of the other design criteria, developed in the context of the EPOS system, will be the subject of a report available in the near future.

The grammatical class chosen determines the simplicity of the parsing algorithm needed as well as the notational flexibility of the system. Since flexibility and simplicity are both desirable but, unfortunately, conflicting goals a compromise must be found.

Of the two extreme classes of grammars, (i) the regular grammars and (ii) all context-free grammars, neither is acceptable.

(i) Regular grammars do not permit nesting to be denoted and, therefore, are totally insufficient.

(ii) The class of all context-free grammars is not recommended for theoretical and practical reasons.

The theoretical problem is in ambiguity. It is not always possible to detect potential ambiguities by analysing the grammar rules added. Hence, we can possibly never tell whether a grammar is free of ambiguities. Perhaps, this problem is not too severe as there are rather simple responses to ambiguities later detected. For instance, one of the parses could be taken at random and a message issued. Nevertheless the problem should be avoided if it can be done without sacrificing essential advantages.

More serious is the fact that even the best general parsing algorithms (e.g. Earley's [E70]) are rather slow because of their high overhead, also for grammars that can be parsed in a time proportional to n. Moreover, some unambiguous grammars require proportional to n^2 parsing steps which is intolerable if long input sentences are considered.

Finally, the fact that parsing overhead increases with the size of the grammar makes general context-free grammars an undesirable class: actual grammars of extensible languages may grow unpredictably.

3.1 Operator precedence grammars

Van Gils [G71] suggested the use of (i) operator precedence grammars as a class whose precedence relation is defined by left and right priorities for finding the skeletal parse (i.e. the structure of the parse tree) and (ii) mode resolution with coercion for modifying and labelling the parse tree.

Since we feel that the simplicity of operator precedence parsing is very attractive but that its limitations are too severe, we will suggest two additions (target modes and environment parameters) that do not complicate the parser noticeably but enhance the flexibility considerably.

3.2 Target modes

Consider the grammar G_a

$V_T = (a|b|v|,|;)$ $\quad V_N = (S'|S|D|E)$

$S' \rightarrow S;$

$S \rightarrow D|S,D \qquad D \rightarrow a\ E|b\ E \qquad E \rightarrow v|E,v$

and the sentence

a v,v,v,b v,v;

This resembles the problem which occurs with the extended form of the collateral identity declaration in ALGOL 68:

(4) $\quad$ int i, j, k, real x, y;

G_a--or the grammar that describes (4)--is not an operator precedence grammar since the relation between a and b on the one hand and , on the other is not unique. As a consequence operator precedence parsing cannot detect the right hand side of the handle that starts with a--or int in (4) respectively.

It is therefore suggested to add the following parsing rule:

For finding the end of its handle, a prefix-like symbol (i.e. a symbol with maximal left priority) may specify the mode (target mode) of the first symbol outside of the handle. If the symbol with the specified mode is preceded by an infix operator (e.g. a comma) the handle ends before the infix symbol. If a target mode is specified, the end of a handle is found if a symbol with a sufficiently low left priority or with the specified (target) mode is encountered. As an example consider the identity declaration above assuming that int and real have the mode mode and also specify mode as a target mode.

Note: In order to properly handle symbols that are not yet declared we assume that they have, by default, maximal left and right priorities and the mode undefined.

Another problem occurs in statements that declare properties of symbols to which priorities have already been assigned like operators in operation declarations of ALGOL 68. Here priority oriented precedence parsing must fail. This problem, too, can be solved by target modes. However, in this case we want the symbol with the specified target mode to be the last element of the current handle rather than the first of the next handle.

In both cases, finding the right end of a handle requires that the mode of a symbol be considered before its priorities.

3.3 Environment parameters

We will use the following simple grammar in order to illustrate the motiviation of the second proposed addition to precedence parsing rules:

$V_T = (a|b|v|.)$ $\quad V_N = (S|A|B)$

$S \rightarrow a\ A|b\ B$ $\quad A \rightarrow v|A\ .\ v$ $\quad B \rightarrow v|v\ .\ B$

This grammar is unambiguous and parseable in linear time. It is not an operator precedence grammar, though, because the relations between $\lessdot$ and $\doteq$ as well as $\doteq$ and $\lessdot$ are not unique. The problem, however, can easily be resolved if the parser is made to remember whether it has previously encountered an a or b. This is accomplished by environment parameters subject to the following rules:

1) A prefix-like symbol P may specify an environment parameter. This parameter is applicable to all elements of the handle that has P as its first constituent.

2) If a symbol P defines a new parameter it automatically disables temporarily (for the remainder of the handle) the parameter that is defined for an outer handle. At the right of the end of the handle the outer parameter is restored.

3) All attributes of a symbol can be made dependent on the current environment parameter (i.e. mode, priorities, target mode, the new environment parameter, the syntactic pattern of the handle governed by the symbol and the code to be generated for the handle). Thus, the same symbol can mean quite different things under the influence of different parameters.

4) The outer most parameter (i.e. the parameter of the whole program) has some predetermined value e.g. zero.

Environment parameters enhance the flexibility without complicating the parsing problem. Moreover, they can be used to improve parsing efficiency and program clarity by giving 'hints' to the compiler and the human reader.

Further, environment parameters permit different sublanguages to be envoked and dismissed within the same program. This feature will be used to distinguish between programs for the hypothetical computer and programs of the extended system. It is

considered for incorporating assembly and microprogramming into the system.

Finally, it is contemplated to direct the lexical scanner (i.e. the rules that govern the forming and separating of symbols) by environment parameters.

3.4 Multiple use of symbols

The problem that some operators are used as both infix and prefix operators can be solved either in the manner suggested by van Gils i.e. by considering the modes of neighboring symbols or by using the following fact:

The right priority of a symbol immediately preceding an infix (a prefix) operator is always (never) maximal. Thus, if a symbol may be both, an infix and a prefix operator, the right priority of the preceding symbol resolves the question.

3.5 Error recovery

One of the compiler problems without a good solution is error handling in general and the continuation of parsing after an error has been detected in particular. The latter problem has a rather systematic solution if operator precedence grammars--pure or with the additions suggested--are used. If all precedence decisions are based on priorities and/or target modes, a unique parse tree is found for every possible string of symbols. Thus, the part of the parser that identifies the structure of the input program never detects errors. All error handling must therefore be done by the following, mode resolving, step. Since the structure of the input is already determined before any error is found the question of parse continuation after error detection never occurs.

3.6 Mode resolution and coercion

The purpose of mode resolution is labeling the parse tree uniquely with addresses to code segments. Mode resolution is a simple bottom-up procedure as long as the modes required at the nodes of the tree match those that are delivered by the roots of their respective subtrees. If the modes do not match, an error has been found or a conversion should be performed (i.e. inserted) by the compiler. The complexity and generality of the procedures that select these conversions is determined (i) by the amount of context that may be considered for making the selection and (ii) by the relation that holds among the modes and indicates whether a conversion exists. In ALGOL 68 the context that may be considered is practically unlimited; the mode structure is hierarchical.

(i) From the standpoint of efficiency it would be very desirable to limit the context sensitivity to the handle processed such that the code for a handle can be generated as soon as the end of that handle is found. Limiting the context considered to just one handle is justified particularly because it does not seem to cost us any essential flexibility.

(ii) Limitations of the hierarchical mode structure have been discussed by Jorrand [J71]. He suggests a scheme that permits an arbitrary graph structure to be used for defining the conversion rules among nodes (classes, as he terms them). His scheme requires that the base language be type-less, a demand which can be met in a very natural way if the base language is represented by the hypothetical computer. It is being investigated if the price that must be paid for the flexibility gained is not too high. In order to determine whether some given mode can be converted into some required mode, a path must be found in the graph that describes the conversion rules. Since the time complexity of path finding algorithms that work on arbitrary graphs can reach k^n (n being the length of the path) the problem of efficiency is very real.

4. Conclusion

Built on a clean definition scheme, the system sketched combines flexibility and efficiency in a balanced manner. It should be pointed out that no time consuming macro facilities but a homogeneous compile algorithm accomodates the extension facilities.

One of the purposes of this paper is to stimulate a discussion on combining considerations of generality and efficiency in a meaningful way, and we shall be grateful for any suggestions or criticism offered.

It is recognized that this paper is too short to do justice to the subject matter. As mentioned before, an in depth discussion on (i) the selection of classes of translations, (ii) the primitives furnished by the hypothetical computer, and (iii) the structure of the data base will be found in a report on the EPOS system.

References

A73 Aho, A. V. and Ullman, J. D., "The theory of parsing, translation, and compiling", Vol. II, Prentice-Hall, Englewood Cliffs, N.J. 1973.

E70 Earley, J., "An efficient context-free parsing algorithm", CACM 13, 2, Feb. 1970, pp. 94-102.

G71 van Gils, T., "Syntactic definition mechanisms,", SIGPLAN Notices, Vol. 6, Number 12, Dec. 1971, pp. 67-74.

I70 Irons, E. T., "Experience with an extensible language", CACM 13, 1, Jan. 1970, pp. 31-40.

J71 Jorrand, P., "Data types and extensible languages", SIGPLAN Notices, Vol. 6, Number 12, Dec. 1971, pp. 75-83.

L66 Leavenworth, B. M., "Syntax macros and extended translation", CACM 9, 11, Nov. 1966, pp. 790-793.

L71 Lindsey, C. H. and van der Meulen, S. G., "Informal introduction to ALGOL 68", North-Holland Publishing Co., Amsterdam, London, 1971.

N71 Notley, M. G., "A model of extensible language systems", SIGPLAN Notices, Vol. 6, Number 12, Dec. 1971, pp. 29-38.

S67 Standish, T. A., "A data definition facility for programming languages", Ph.D. Thesis, Carnegie Institute of Technology, May 1967.

S70 Schuman, S. A., and Jorrand, P., "Definition mechanisms in extensible programming languages", AFIPS Conference Proceedings, Vol. 37, AFIPS Press 1970, pp. 9-20.

S71 Scowen, R. S., "Babel, an application of extensible compilers", SIGPLAN Notices, Vol. 6, Number 12, Dec. 1971, pp. 1-7.

W69 van Wijngaarden, A. (Ed.) et al., "Report on the algorithmic language ALGOL 68, MR.101, Mathematisch Centrum, Amsterdam, Oct. 1969.

W71 Wegbreit, B., "An overview of the ECL programming system", SIGPLAN Notices, Vol. 6, Number 12, Dec. 1971, pp. 26-28.