EXTENSIBLE LANGUAGES:
A POTENTIAL USER'S POINT OF VIEW

J.J. Duby
IBM France

(Ask not what you can do for extensible languages,
ask what extensible languages can do for you.)

The purpose of this paper is to describe a computer user's concern about what extensible languages will do for him, and how they will do it. It may in some respects sound a little demagogic, by considering users requirements as priority requirements. However, as Lenin said, "facts are stubborn", and if users have been used to certain performance characteristics of usual programming languages, they will want to get at least the same performance from extensible languages, and in fact much more - otherwise why switch?

Semantic range:

The semantic range of the majority of the extensible languages implemented to-day stretches more upwards than downwards. This is very unfortunate, since one of the most interesting capabilities of extensible languages would be to enable the user to support new devices as well as new data types, and such a capability requires downward extensibility[2] giving access to machine language and system functions.

For instance, users now want to be able to have access to assembly language through high level languages, and several extensions of ALGOL 60 and PL/I have been defined and implemented to fulfil this need. Very naturally they will expect that extensible languages do the same. Will indeed the semantic range of extensible languages encompass that of assembly language?

PL/I was extended from its original definition to support teleprocessing by the addition of one file attribute and one condition, providing a link to the QTAM Message Control Program of the Operating System: this extension required a new version of the compiler, but achieving the same result by mere language extension techniques would be a significant mark for an extensible language. Another target would be to design extensions to run in a paged environment, enabling, for instance, to force object modules to be loaded in the same page, or to block a page in real memory during some portion of the execution.

Downward extensibility however raises a fundamental contradiction in the concept of an extensible language: if the base language or the extension mechanism contains machine dependent features, the language is not universal; if neither does, it is not downwards extensible. The only solution to this contradiction would be the long awaited, but yet to come, universal semantics description language.

Obsolete programming languages such as FORTRAN and COBOL are still widely used. Among the most commonly mentionned reasons, one hears: precise syntax diagnostics, efficient object code, possibility to link separately compiled modules. It is not obvious that the extensible languages will have the same popular characteristics.

Syntax diagnostics:

It is unfortunately recursively unsolvable to determine whether an arbitrary context free grammar is ambiguous when the terminal alphabet contains more than one symbol [4]. Besides, the syntactic definition method by successive and possibly independent extensions makes it difficult for the extension designers to check for possible ambiguities without knowing the previous extensions.

For instance, the designers of the macro extensions MOD and NORM defined below following Schuman and Jorrand's notation [6] and using PL/I terminology, had better not ignore one another:

```
macro MOD  --> '|PRIM|' means 'sum(abs(PRIM))'
macro NORM --> '||PRIM||' means 'sqrt(sum(PRIM*PRIM))'
```

However, this difficulty can be overcome by using parallel parsers for the derived language, or by restricting its syntax.

Object code efficiency:

The question of optimised object code may possibly be left aside for experimental programming languages. However, when one approaches the limits of the power of computers, or when one designs a language and a compiler to be used in a production environment, optimisation is a must. The importance of this user requirement is best exemplified by the poor acceptance of early PL/I compilers by FORTRAN and COBOL users. Unfortunately, the known techniques of program optimisation do not seem to be all readily applicable to extensible languages.

One can classify the different optimising techniques now in use in three large families:

i. Local optimisation of the code generated and register assignment, taking into account the context in which the code is being produced.

ii. Global optimisation, operating on register assignment and deleting, displacing and replacing instructions taking into account the control flow of the program.

iii. Syntax directed optimisation, which takes into account the pattern of occurrence of operators in the syntactic tree of the program to avoid unnecessary computations.

Local optimisation can be easily performed in the case of extensible languages [2].

The elementary operations performed by global optimisation (instruction displacement and substitution) can also be easily adapted to extensible languages since they are performed at some intermediate language level; however, the flow analysis [1] which governs those operations will be impacted by the introduction of extended control structures, and strength reduction of instructions operating on induction variables [5] will be affected by the introduction of extended data types: communication of information by the extension designer to the optimiser would probably be useful in these two areas.

It will be not only useful, but absolutely necessary, if one wants to perform syntax-directed optimisation. Furthermore, the relative importance of syntax-directed optimisation with respect to other optimising techniques increases with the semantic density of the language: the more powerful the operators are, the more necessary syntax-directed optimisation is. As an example, suppose one introduces APL operators into ALGOL 68 [7]; then if A and B are two 10x10 matrices, the elaboration of

(1 2 1 2) ⌿ A∘.×B

will lead to the computation of a 10x10x10x10 array. Only syntax-directed optimisation can detect, by top down scanning of the syntax tree, that only 100 components of the outer product out of 10,000 are necessary. This kind of optimising information must be transmitted by the extension designer, in a suitable language [3]. Furthermore, a good syntax-directed optimisation requires that all combinations of all operators be considered as candidates for special casing, which is impossible if the extensions are defined independently from one another.

Linkage of separately compiled programs:

The problem of linking two separately compiled programs written in an extensible language is not made trivial by the mere fact that all extensions are translated into the same derived language: representations of data structures must also be compatible. And it is not obvious that a same data structure introduced in two different extensions will have two identical representations. To take one simple example,

$$(a_{i}^{j})_{\substack{i=1,n \\ j=1,p}}$$ can be defined as

$$((a_{i}^{j})_{i=1,n}^{j=1,p})$$ as well as $$((a_{i}^{j})_{i}^{j=1,p})_{i=1,n}$$

This kind of phenomenon explains why FORTRAN and PL/I programs could not be linked together, even though compilers obeyed the same linkage conventions. Obviously, the facility to

introduce new data structures in extensible languages will make the problem frequently arise. Storing data structure representation information in the object code, and using this information to automatically generate at linkage edition time mapping procedures will probably be difficult, and the object time efficiency will probably suffer, however clever the solution is.

Conclusion:

The problems that were raised in this paper have to do with every area of extensible languages:

- Specification of the base language and extension mechanism: is it possible to combine accessibility to machine code and transferability?

- Formulation of syntactic extensions: is it possible to define a new extension ignoring all the previous ones?

- Formulation of semantic extensions: is it possible to define semantic extensions that will produce good object code?

- Specification of the object language: is it possible to insure compatibility of separately defined data structures?

More research will probably be necessary to solve those problems. May extensible language designers be convinced that solutions to those problems are of utmost importance for the crowd of computer users.

BIBLIOGRAPHY:

[1] ALLEN, F.E., Control flow analysis, SIGPLAN Notices, July 1970.

[2] DICKMAN, B.N., ETC : An extensible macro based compiler, SJCC 1971.

[3] ELSON, M., and RAKE, S., Code Generation for large language compilers, IBM Systems Journal, Vol. 9, Nr. 3, 1970.

[4] FLOYD, R.W., On ambiguity in phrase structure languages, Communications of the ACM, October 1962.

[5] LOWRY, E.S. and MEDLOCK, C.W., Object code optimisation, Communications of the ACM, January 1969.

[6] SCHUMAN, S.A., and JORRAND, P., Definition mechanism in extensible programming languages, FJCC 1970.

[7] van WIJNGAARDEN, A., Private communication.