A Comparative Study of Formalisms for Programming Language Definition.

A thesis presented in partial fulfilment
of the requirements for the degree of
Master of Science in
Computer Science at
Massey University.

Ian Joseph Thompson

January 1975

Abstract

This study looks at a number of methods for defining the full syntax and semantics of computer programming languages. The syntax, especially the nature of context-dependent conditions in it, is first examined, then some extensions of context-free grammars are compared to see to what extent they can encompass the full context-conditions of typical programming languages. It is found that several syntax extensions are inadequate in this regard, and that the ability to calculate complicated functions and conditions, and to eventually delete the values of such functions, is needed. This ability may be obtained either by allowing unrestricted rules and meta-variables in the phrase-structure, or by associating mathematical functions either with individual production rules or with the whole context-free structure, to transform it into an 'abstract syntax'.

Since the form of a definition of a programming language semantics depends critically on how one conceives "meaning", five main types of semantics are considered: these are called 'natural', 'propositional', 'functional', and 'structural' semantics, as well as a semantics based on string rewriting rules. The five types are compared for their success in defining the semantics of computing languages, of the example Algol-like language ALEX in particular. Among other conclusions, it is found that the semantics of structures and computations on structures is the only type sufficiently comprehensive, precise, and readable.

Acknowledgements

In the preparation of this thesis, I wish particularly to thank Lloyd Thomas and Professor Tate for many valuable discussions, suggestions, and criticisms, and also to thank Bob Doran for help in his obtaining important source materials.

<u>Table of Contents</u>

1 The problems of programming language definition

Defining a computing language is generally done in two stages:

1) syntax : defining as rigorously as possible the set of all possible programs of the language, together with their formal structures and substructures.
2) semantics : associating with each such program its meaning, so that the effects of executing the program with its data are as rigorously defined as possible.

Details of definitions of these two stages will be discussed in the following two chapters (although the exact demarcation between the stages has varied for different people; I shall discuss this further in section 2.2).

The aim is to be able to define a significantly large language, including both stages of definition, and to this end there are several criteria for comparing the different systems examined later.

1) Scope of the definition method

   Is it applicable to all features of all programming languages, or are there some features that can be encompassed either not at all, only with great difficulty, or at the cost of breaking up a neat system?

2) Elegance

   A general aim is for a definition as readable, concise, and 'transparent' as possible. A readable definition should be understandable even with only a short initiation into the details of the formalism ; it should not be written in a wholly foreign language. A definition should also transparently follow the language being defined; this means that small changes in the language should require only small changes in the formal definition. Concerning conciseness, one should distinguish between the method and its application to specific languages: a very simple method will generally lead to a very complicated definition.

3) Rigour

Syntax definitions should define, ideally, all and only the programs in the language, and assign correct formal structures to valid programs. They should avoid overlapping, incompatible, ambiguous, and/or missing specifications. Similarly with semantics. Note, however, that it is occasionally desireable to leave certain parts of a standard definition either completely open, or to deliberately give only a partial definition of them. For example, the details of real arithmetic, beyond certain basic conditions, may be postponed beyond the standard definition; and in any case the effect of merging parallel operations should intentionally be left undetermined.

4) Formalisation

The formalists' ideal is that a definition should say everything that can be said about all programs in the language, and in such a manner that mechanical statements can be made about the program without either using human understanding at this point, or running it on a computer with specific data. Such statements, for example, could concern the mechanical design of implementations, or the mechanical proofs of correctness, equivalence, etc., of programs in the language.

Chapter 2 looks at the syntactic, and chapter 3 the semantic, components of definitions of programming languages, and in the appendices I have used those methods which are sufficiently powerful for the definition of "ALEX". ALEX is the name which henceforth I give to a certain subset of Algol 60; it does not include arrays, for-loops, conditional expressions, or designational expressions, but it does include mixed-mode arithmetic, procedures, functions, call-by-name and call-by-value parameters, "goto" and "if" statements, and the implicit declaration of labels.