Introduction

Any notation for giving instructions is a programming language. Arithmetic notation is a programming language; so is Pascal. The input data format for an applications program is also a programming language. The person who uses an applications program thinks of its input commands as a language, just like the program's implementor thought of Pascal when he used it to implement the applications program. The person who wrote the Pascal compiler had a similar view about the language used for coding the compiler. This series of languages and viewpoints terminates at the physical machine, where code is converted into action.

A programming language has three main characteristics:

1. *Syntax*: the appearance and structure of its sentences.
2. *Semantics*: the assignment of meanings to the sentences. Mathematicians use meanings like numbers and functions, programmers favor machine actions, musicians prefer audible tones, and so on.
3. *Pragmatics*: the usability of the language. This includes the possible areas of application of the language, its ease of implementation and use, and the language's success in fulfilling its stated goals.

Syntax, semantics, and pragmatics are features of every computer program. Let's consider an applications program once again. It is a *processor* for its input language, and it has two main parts. The first part, the input checker module (the *parser*), reads the input and verifies that it has the proper syntax. The second part, the evaluation module, evaluates the input to its corresponding output, and in doing so, defines the input's semantics. How the system is implemented and used are pragmatics issues.

These characteristics also apply to a general purpose language like Pascal. An interpreter for Pascal also has a parser and an evaluation module. A pragmatics issue is that the interpretation of programs is slow, so we might prefer a compiler instead. A Pascal compiler transforms its input program into a fast-running, equivalent version in machine language.

The compiler presents some deeper semantic questions. In the case of the interpreter, the semantics of a Pascal program is defined entirely by the interpreter. But a compiler does not define the meaning— it *preserves* the meaning of the Pascal program in the machine language program that it constructs. The semantics of Pascal is an issue independent of any particular compiler or computer. The point is driven home when we implement Pascal compilers on two different machines. The two different compilers preserve the same semantics of Pascal. Rigorous definitions of the syntax and semantics of Pascal are required to verify that a compiler is correctly implemented.

The area of syntax specification has been thoroughly studied, and Backus-Naur form (BNF) is widely used for defining syntax. One of reasons the area is so well developed is that a close correspondence exists between a language's BNF definition and its parser: the definition dictates how to build the parser. Indeed, a parser generator system maps a BNF definition to a guaranteed correct parser. In addition, a BNF definition provides valuable documentation that can be used by a programmer with minimal training.

Semantics definition methods are also valuable to implementors and programmers, for they provide:

1. A precise standard for a computer implementation. The standard guarantees that the language is implemented exactly the same on all machines.
2. Useful user documentation. A trained programmer can read a formal semantics definition and use it as a reference to answer subtle questions about the language.
3. A tool for design and analysis. Typically, systems are implemented before their designers study pragmatics. This is because few tools exist for testing and analyzing a language. Just as syntax definitions can be modified and made error-free so that fast parsers result, semantic definitions can be written and tuned to suggest efficient, elegant implementations.
4. Input to a compiler generator. A compiler generator maps a semantics definition to a guaranteed correct implementation for the language. The generator reduces systems development to systems specification and frees the programmer from the most mundane and error prone aspects of implementation.

Unfortunately, the semantics area is not as well developed as the syntax area. This is for two reasons. First, semantic features are much more difficult to define and describe. (In fact, BNF's utility is enhanced because those syntactic aspects that it *cannot* describe are pushed into the semantics area! The dividing line between the two areas is not fixed.) Second, a standard method for writing semantics is still evolving. One of the aims of this book is to advocate one promising method.

METHODS FOR SEMANTICS SPECIFICATION

Programmers naturally take the meaning of a program to be the actions that a machine takes upon it. The first versions of programming language semantics used machines and their actions as their foundation.

The *operational semantics* method uses an interpreter to define a language. The meaning of a program in the language is the evaluation history that the interpreter produces when it interprets the program. The evaluation history is a sequence of internal interpreter configurations.

One of the disadvantages of an operational definition is that a language can be understood only in terms of interpreter configurations. No machine-independent definition exists, and a user wanting information about a specific language feature might as well invent a program using the feature and run it on a real machine. Another problem is the interpreter itself: it is represented as an algorithm. If the algorithm is simple and written in an elegant notation, the interpreter can give insight into the language. Unfortunately, interpreters for nontrivial languages are large and complex, and the notation used to write them is often as complex as the language being defined. Operational definitions are still worthy of study because one need only implement the interpreter to implement the language.

The *denotational semantics* method maps a program directly to its meaning, called its *denotation.* The denotation is usually a mathematical value, such as a number or a function. No interpreters are used; a *valuation function* maps a program directly to its meaning.

A denotational definition is more abstract than an operational definition, for it does not

specify computation steps. Its high-level, modular structure makes it especially useful to language designers and users, for the individual parts of a language can be studied without having to examine the entire definition. On the other hand, the implementor of a language is left with more work. The numbers and functions must be represented as objects in a physical machine, and the valuation function must be implemented as the processor. This is an ongoing area of study.

With the *axiomatic semantics* method, the meaning of a program is not explicitly given at all. Instead, *properties* about language constructs are defined. These properties are expressed with axioms and inference rules from symbolic logic. A property about a program is deduced by using the axioms and rules to construct a formal proof of the property. The character of an axiomatic definition is determined by the kind of properties that can be proved. For example, a very simple system may only allow proofs that one program is equal to another, whatever meanings they might have. More complex systems allow proofs about a program's input and output properties.

Axiomatic definitions are more abstract than denotational and operational ones, and the properties proved about a program may not be enough to completely determine the program's meaning. The format is best used to provide preliminary specifications for a language or to give documentation about properties that are of interest to the users of the language.

Each of the three methods of formal semantics definition has a different area of application, and together the three provide a set of tools for language development. Given the task of designing a new programming system, its designers might first supply a list of properties that they wish the system to have. Since a user interacts with the system via an input language, an axiomatic definition is constructed first, defining the input language and how it achieves the desired properties. Next, a denotational semantics is defined to give the meaning of the language. A formal proof is constructed to show that the semantics contains the properties that the axiomatic definition specifies. (The denotational definition is a *model* of the axiomatic system.) Finally, the denotational definition is implemented using an operational definition. These *complementary semantic definitions* of a language support systematic design, development, and implementation.

This book emphasizes the denotational approach. Of the three semantics description methods, denotational semantics is the best format for precisely defining the meaning of a programming language. Possible implementation strategies can be derived from the definition as well. In addition, the study of denotational semantics provides a good foundation for understanding many of the current research areas in semantics and languages. A good number of existing languages, such as ALGOL60, Pascal, and LISP, have been given denotational semantics. The method has also been used to help design and implement languages such as Ada, CHILL, and Lucid.

SUGGESTED READINGS

Surveys of formal semantics: Lucas 1982; Marcotty, Ledgaard, & Bochman 1976; Pagan 1981

Operational semantics: Ollengren 1974; Wegner 1972a, 1972b
Denotational semantics: Gordon 1979; Milne & Strachey 1976; Stoy 1977; Tennent 1976
Axiomatic semantics: Apt 1981; Hoare 1969; Hoare & Wirth 1973
Complementary semantics definitions: deBakker 1980; Donohue 1976; Hoare & Lauer 1974
Languages with denotational semantics definitions: SNOBOL: Tennent 1973
LISP: Gordon 1973, 1975, 1978; Muchnick & Pleban 1982
ALGOL60: Henhapl & Jones 1982; Mosses 1974
Pascal: Andrews & Henhapl 1982; Tennent 1977a
Ada: Bjorner & Oest 1980; Donzeau-Gouge 1980; Kini, Martin, & Stoughton 1982
Lucid: Ashcroft & Wadge 1982
CHILL: Branquart, Louis, & Wodon 1982
Scheme: Muchnick & Pleban 1982