Language Theory and Semantics

12

Key Topics

Alphabets
Grammars and Parse Trees
Axiomatic semantics
Operational semantics
Denotational semantics
Lambda calculus
Lattices and partial orders
Complete partial orders
Fixpoint theory

12.1 Introduction

There are two key parts to any programming language, and these are its syntax and semantics. The syntax is the grammar of the language and a program needs to be syntactically correct with respect to its grammar. The semantics of the language is deeper, and determines the meaning of what has been written by the programmer.

The difference between syntax and semantics may be illustrated by an example in a natural language. A sentence may be syntactically correct but semantically meaningless, or a sentence may have semantic meaning but be syntactically incorrect. For example, consider the sentence:

"I will go to Dublin yesterday"

G. O'Regan, *Guide to Discrete Mathematics*, Texts in Computer Science,
DOI 10.1007/978-3-319-44561-8_12

Then this sentence is syntactically valid but semantically meaningless. Similarly, if a speaker utters the sentence "Me Dublin yesterday" we would deduce that the speaker had visited Dublin the previous day even though the sentence is syntactically incorrect.

The semantics of a programming language determines what a syntactically valid program will compute. A programming language is therefore given by:

$$\text{Programming Language} = \text{Syntax} + \text{Semantics}$$

Many programming languages have been developed over the last 60 years including Plankalkül which was developed by Zuse in the 1940s; Fortran developed by IBM in the 1950s; Cobol was developed by a committee in the late 1950s; Algol 60 and Algol 68 were developed by an international committee in the 1960s; Pascal was developed by Wirth in the early 1970s; Ada was developed for the US military in the late 1970s; the C language was developed by Richie and Thompson at Bell Labs in the 1970s; C++ was developed by Stroustrup at Bell Labs in the early 1980s; and Java developed by Gosling at Sun Microsystems in the mid-1990s. A short description of a selection of programming languages in use is in [1].

A programming language needs to have a well-defined syntax and semantics, and the compiler preserves the semantics of the language. Compilers are programs that translate a program that is written in some programming language into another form. It involves syntax analysis and parsing to check the syntactic validity of the program; semantic analysis to determine what the program should do; optimization to improve the speed and performance; and code generation in some target language.

Alphabets are a fundamental building block in language theory, as words and language are generated from alphabets. They are discussed in the next section.

12.2 Alphabets and Words

An *alphabet* is a finite non-empty set A, and the elements of A are called letters. For example, consider the set A which consists of the letters a to z.

Words are finite strings of letters, and a set of words is generated from the alphabet. For example, the alphabet $A = \{a, b\}$ generates the following set of words:

$$\{\varepsilon, a, b, aa, ab, bb, ba, aaa, bbb, \ldots\}$$

Each word consists of an ordered list of one or more letters and the set of words of length two consists of all ordered lists of two letters[1]. It is given by

[1] ε denotes the empty word.

$$A^2 = \{aa, ab, bb, ba\}$$

Similarly, the set of words of length three is given by

$$A^3 = \{aaa, aab, abb, aba, baa, bab, bbb, bba\}$$

The set of all words over the alphabet A is given by the positive closure A^+, and it is defined by

Given any two words $w_1 = a_1a_2\ldots a_k$ and $w_2 = b_1b_2\ldots b_r$ then the concatenation of w_1 and w_2 is given by

$$w = w_1w_2 = a_1a_2\ldots a_kb_1b_2\ldots b_r$$

The empty word is a word of length zero and is denoted by ε. Clearly, $\varepsilon w = w\varepsilon = w$ for all w and so ε is the identity element under the concatenation operation. A^0 is used to denote the set containing the empty word $\{\varepsilon\}$, and the closure A^* ($=A^+ \cup \{\varepsilon\}$) denotes the infinite set of all words over A (including empty words). It is defined as:

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

The mathematical structure (A^*, ^, ε) forms a monoid,[2] where ^ is the concatenation operator for words and the identity element is ε. The length of a word w is denoted by $|w|$ and the length of the empty word is zero: i.e., $|\varepsilon| = 0$.

A subset L of A^* is termed a formal language over A. Given two languages L_1, L_2 then the concatenation (or product) of L_1 and L_2 is defined by

$$L_1L_2 = \{w | w = w_1w_2 \qquad \text{where } w_1 \in L_1 \text{ and } w_2 \in L_2\}$$

The positive closure of L and the closure of L may also be defined as

$$L^+ = \bigcup_{n=1}^{\infty} L^n \quad L^* = \bigcup_{n=0}^{\infty} L^n$$

12.3 Grammars

A formal grammar describes the syntax of a language, and we distinguish between *concrete* and *abstract syntax*. Concrete syntax describes the external appearance of programs as seen by the programmer, whereas abstract syntax aims to describe the

[2]Recall from chapter 6 that a monoid (M, *, *e*) is a structure that is closed and associative under the binary operation "*", and it has an identity element "*e*".

Fig. 12.1 Noah chomsky. Courtesy of Duncan rawlinson

essential structure of programs rather than the external form. In other words, abstract syntax aims to give the components of each language structure while leaving out the representation details (e.g., syntactic sugar). Backus Naur Form (BNF) notation is often used to specify the concrete syntax of a language. A grammar consists of

- A finite set of terminal symbols
- A finite set of nonterminal symbols
- A set of production rules
- A start symbol

A formal grammar generates a formal language, which is a set of finite length sequences of symbols created by applying the production rules of the grammar. The application of a production rule involves replacing symbols at the left-hand side of the rule with the symbols on the right-hand side of the rule. The formal language then consists of all words consisting of terminal symbols that are reached by a derivation (i.e., the application of production rules) starting from the start symbol of the grammar.

A construct that appears on the left-hand side of a production rule is termed a *nonterminal*, whereas a construct that only appears on the right-hand side of a production rule is termed a *terminal.* The set of nonterminals N is disjoint from the set of terminals A.

The theory of the syntax of programming languages is well established, and programming languages have a well-defined grammar that allows syntactically valid programs to be derived from the grammars.

Chomsky[3] (Fig. 12.1) was a famous linguist who classified a number of different types of grammar that occur.

[3]Chomsky made important contributions to linguistics and the theory of grammars. He is more widely known today as a critic of United States foreign policy.

Table 12.1 Chomsky hierarchy of grammars

Grammar type	Description
Type 0 grammar	Type 0-grammars include all formal grammars. They have production rules of the form $\alpha \rightarrow \beta$, where α and β are strings of terminals and nonterminals. They generate all languages that can be recognized by a Turing machine (discussed in Chap. 7)
Type 1 grammar (context sensitive)	These grammars generate the context-sensitive languages. They have production rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ where A is a nonterminal and α, β and γ are strings of terminals and nonterminals. They generate all languages that can be recognized by a linear bounded automaton[a]
Type 2 grammar (context free)	These grammars generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$, where A is a nonterminal and γ is a string of terminals and nonterminals. These languages are recognized by a pushdown automaton[b] and are used to define the syntax of most programming languages
Type 3 grammar (regular grammars)	These grammars generate the regular languages (or regular expressions). These are defined by rules of the form $A \rightarrow a$ or $A \rightarrow aB$, where A and B are nonterminals and a is a single terminal. A finite state automaton recognizes these languages (discussed in Chap. 7), and regular expressions are used to define the lexical structure of programming languages

[a]A linear bounded automaton is a restricted form of a nondeterministic Turing machine in which a limited finite portion of the tape (a function of the length of the input) may be accessed
[b]A pushdown automaton is a finite automaton that can make use of a stack containing data, and it is discussed in Chap. 7

The Chomsky hierarchy (Table 12.1) consists of four levels including regular grammars; context free grammars; context sensitive grammars and unrestricted grammars. The grammars are distinguished by the production rules, which determine the type of language that is generated.

Regular grammars are used to generate the words that may appear in a programming language. This includes the identifiers (e.g., names for variables, functions and procedures); special symbols (e.g., addition, multiplication, etc.); and the reserved words of the language.

A rewriting system for context free grammars is a finite relation between N and $(A \cup N)^*$: i.e., a subset of $N \times (A \cup N)^*$: A production rule <N> $\rightarrow$ w is one element of this relation, and is an ordered pair (<N>, w) where w is a word consisting of zero or more terminal and nonterminal letters. This production rule means that <N> may be replaced by w.

12.3.1 Backus Naur Form

Backus Naur Form[4] (BNF) provides an elegant means of specifying the syntax of programming languages. It was originally employed to define the grammar for the Algol-60 programming language [2], and a variant was used by Wirth to specify the syntax of the Pascal programming language. BNF is widely used and accepted today as the way to specify the syntax of programming languages.

[4]Backus Naur Form is named after John Backus and Peter Naur. It was created as part of the design of the Algol 60 programming language, and is used to define the syntax rules of the language.

BNF specifications essentially describe the external appearance of programs as seen by the programmer. The grammar of a context-free grammar may then be input into a parser (e.g., Yacc), and the parser is used to determine if a program is syntactically correct or not.

A BNF specification consists of a set of production rules with each production rule describing the form of a class of language elements such expressions, statements and so on. A production rule is of the form

<symbol> ::= <expression with symbols>

where <symbol> is a *nonterminal*, and the expression consists of sequence of terminal and nonterminal symbols. A construct that has alternate forms appears more than once, and this is expressed by sequences separated by the vertical bar "|" (which indicates a choice). In other word, there is more than one possible substitution for the symbol on the left-hand side of the rule. Symbols that never appear on the left-hand side of a production rule are called *terminals*.

The following example defines the syntax of various statements in a sample programming language:

<loop statement> ::= <while loop> | <for loop>
<while loop> ::= while (<condition>) <statement>
<for loop> ::= for (<expression>) <statement>
<statement> ::= <assignment statement> | <loop statement>
<assignment statement> ::= <variable> := <expression>

This is a partial definition of the syntax of various statements in the language. It includes various nonterminals such as (<loop statement>, <while loop> and so on. The terminals include 'while', 'for', ':=', '("and")'. The production rules for <condition> and <expression> are not included.

The grammar of a context-free language (e.g. LL(1), LL(k), LR(1), LR(k)) grammar expressed in BNF notation) may be translated by a parser into a parse table. The parse table may then be employed to determine whether a particular program is valid with respect to its grammar.

Example 12.1 (Context-free grammar) The example considered is that of parenthesis matching in which there are two terminal symbols and one nonterminal symbol

$$\begin{aligned} S &\rightarrow SS \\ S &\rightarrow (S) \\ S &\rightarrow () \end{aligned}$$

Then by starting with S and applying the rules we can construct

$$S \rightarrow SS \rightarrow (S)S \rightarrow (())S \rightarrow (())()$$

Example 12.2 (Context-free grammar) The example considered is that of expressions in a programming language. The definition is ambiguous as there is more than one derivation tree for some expressions (e.g., there are two parse trees for the expression 5 × 3 + 1 discussed below).

<expr> ::= <numeral> | (<expr>)
| (<expr> <operator> <expr>)
<operator> ::= + | - | × | /
<digit> ::= 0 | 1 | .... | 9
<numeral>::=<digit> |<digit><numeral>

Example 12.3 (Regular Grammar) The definition of an identifier in most programming languages is similar to

<identifier> ::= <let> <letdig>
<letdig> ::= <let> | <dig> | ε
<letdig> ::= <let> <letdig> | <dig> <letdig>
<let> ::= *a* | *b* | *c* | .... | *z*
<dig> ::= 0 | 1 | .... | 9

12.3.2 Parse Trees and Derivations

Let A and N be the terminal and nonterminal alphabet of a rewriting system and let $\langle X \rangle \rightarrow w$ be a production. Let x be a word in $(A \cup N)^*$ with $x = u \langle X \rangle v$ for some words $u, v \in (A \cup N)^*$. Then x is said to directly yield uwv and this is written as $x \Rightarrow uwv$.

This single substitution ($\Rightarrow$) can be extended by a finite number of productions ($\Rightarrow^*$), and this gives the set of words that can be obtained from a given word. This derivation is achieved by applying several production rules (one production rule is applied at a time) in the grammar.

That is, given $x, y \in (A \cup N)^*$ then x yields y (or y is a derivation of x) if $x = y$, or there exists a sequence of words $w_1, w_2, \ldots, w_n \in (A \cup N)^*$ such that $x = w_1$, $y = w_n$ and $w_i \Rightarrow w_{i+1}$ for $1 \leq i \leq n - 1$. This is written as $x \Rightarrow^* y$.

The expression grammar presented in Example 12.2 is ambiguous, and this means that an expression such as 5 × 3 + 1 has more than one interpretation. (Figures 12.2 and 12.3). It is not clear from the grammar whether multiplication is performed first and then addition, or whether addition is performed first and then multiplication.

The first parse tree is given in Fig. 12.2, and the interpretation of the first parse tree is that multiplication is performed first and then addition (this is the normal interpretation of such expressions in programming languages as multiplication is a higher precedence operator than addition).

The interpretation of the second parse tree is that addition is performed first and then multiplication (Fig. 12.3). It may seem a little strange that one expression has

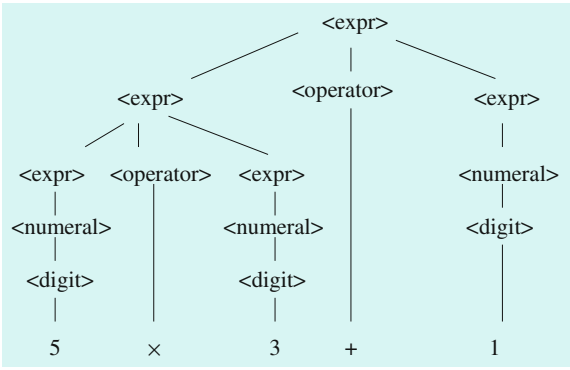

Fig. 12.2 Parse tree $5 \times 3 + 1$

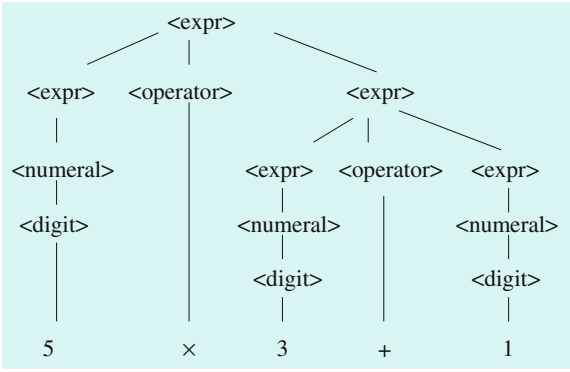

Fig. 12.3 Parse tree $5 \times 3 + 1$

two parse trees and it shows that the grammar is ambiguous. This means that there is a choice for the compiler in evaluating the expression, and the compiler needs to assign the right meaning to the expression. For the expression grammar one solution would be for the language designer to alter the definition of the grammar to remove the ambiguity.

12.4 Programming Language Semantics

The formal semantics of a programming language is concerned with defining the actual meaning of a language. Language semantics is deeper than syntax, and the theory of the syntax of programming languages is well established. A programmer writes a program according to the rules of the language. The compiler first checks the program for syntactic correctness: i.e., it determines whether the program written is valid according to the rules of the grammar of the language. If the program is syntactically correct, then the compiler determines the meaning of what has been written and generates the corresponding machine code.[5]

[5]Of course, what the programmer has written may not be what the programmer had intended.

Table 12.2 Programming language semantics

Approach	Description
Axiomatic semantics	This involves giving meaning to phrases of the language using logical axioms It employs *pre-* and *post-condition assertion*s to specify what happens when the statement executes. The relationship between the initial assertion and the final assertion essentially gives the semantics of the code
Operational semantics	This approach describes how a valid program is interpreted as sequences of computational steps. These sequences then define the meaning of the program An abstract machine (SECD machine) may be defined to give meaning to phrases, and this is done by describing the transitions they induce on states of the machine
Denotational semantics	This approach provides meaning to programs in terms of mathematical objects such as integers, tuples and functions Each phrase in the language is translated into a mathematical object that is the *denotation* of the phrase

The compiler must preserve the semantics of the language: i.e., the semantics are not defined by the compiler, but rather the function of the compiler is to preserve the semantics of the language. Therefore, there is a need to have an unambiguous definition of the meaning of the language independently of the compiler, and the meaning is then preserved by the compiler.

A program's syntax[6] gives no information to the meaning of the program, and therefore there is a need to supplement the syntactic description of the language with a formal unambiguous definition of its semantics.

It is possible to utter syntactically correct but semantically meaningless sentences in a natural language. Similarly, it is possible to write syntactically correct programs that behave in quite a different way from the intention of the programmer.

The formal semantics of a language is given by a mathematical model that describes the possible computations described by the language. There are three main approaches to programming language semantics namely axiomatic semantics, operational semantics and denotational semantics (Table 12.2):

There are several applications of programming language semantics including language design, program verification, compiler writing and language standardization. The three main approaches to semantics are described in more detail below.

12.4.1 Axiomatic Semantics

Axiomatic semantics gives meaning to phrases of the language by describing the logical axioms that apply to them. It was developed by C.A.R. Hoare[7] in a famous paper "*An axiomatic basis for computer programming*" [3]. His axiomatic theory consists of *syntactic elements*, *axioms* and *rules of inference*.

[6]There are attribute (or affix) grammars that extend the syntactic description of the language with supplementary elements covering the semantics. The process of adding semantics to the syntactic description is termed decoration.

[7]Hoare was influenced by earlier work by Floyd on assigning meanings to programs using flowcharts [4].

The well-formed formulae that are of interest in axiomatic semantics are pre–post assertion formulae of the form $P\{a\}Q$, where a is an instruction in the language and P and Q are assertions: i.e., properties of the program objects that may be true or false.

An *assertion* is essentially a predicate that may be true in some states and false in other states. For example, the assertion $(x - y > 5)$ is true in the state in which the values of x and y are 7 and 1, respectively, and false in the state where x and y have values 4 and 2.

The pre- and post-condition assertions are employed to specify what happens when the statement executes. The relationship between the initial assertion and the final assertion gives the semantics of the code statement. The *pre*-and *post-condition* assertions are of the form

$$P\{a\}Q$$

The precondition P is a predicate (input assertion), and the postcondition Q is a predicate (output assertion). The braces separate the assertions from the program fragment. The well-formed formula $P\{a\}Q$ is itself a predicate that is either true or false.

This notation expresses the *partial correctness*[8] of a with respect to P and Q, and its meaning is that if statement a is executed in a state in which the predicate P is true and execution terminates, then it will result in a state in which assertion Q is satisfied.

The axiomatic semantics approach is described in more detail in [5], and the axiomatic semantics of a selection of statements is presented below.

- **Skip**

The skip statement does nothing and whatever condition is true on entry to the command is true on exit from the command. It's meaning is given by:

$$P\{skip\}P$$

- **Assignment**

The meaning of the assignment statement is given by the axiom

$$P_e^x\{x := e\}P$$

The meaning of the assignment statement is that P will be true after execution of the assignment statement if and only if the predicate P_e^x with the value of x replaced

[8]Total correctness is expressed using $\{P\}a\{Q\}$ amd program fragment a is totally correct for precondition P and postcondition Q if and only if whenever a is executed in any state in which P is satisfied then execution terminates, and the resulting state satisfies Q.

by e in P is true before execution (since x will contain the value of e after execution).

The notation P_e^x denotes the expression obtained by substituting e for all free occurrences of x in P.

- **Compound**

The meaning of the conditional command is:

$$\frac{P\{S_1\}Q, Q\{S_2\}R}{P\{S_1; S_2\}R}$$

The compound statement involves the execution of S_1 followed by the execution of S_2. The meaning of the compound statement is that R will be true after the execution of the compound statement S_1; S_2 provided that P is true, if it is established that Q will be true after the execution of S_1 provided that P is true, and that R is true after the execution of S_2 provided Q is true.

There needs to be at least one rule associated with every construct in the language in order to give its axiomatic semantics. The semantics of other programming language statements such as the 'while' statement and the 'if' statement are described in [5].

12.4.2 Operational Semantics

The operational semantics definition is similar to an interpreter, where the semantics of a language are expressed by a mechanism that makes it possible to determine the effect of any program in the language. The meaning of a program is given by the evaluation history that an interpreter produces when it interprets the program. The interpreter may be close to an executable programming language or it may be a mathematical language.

The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. The evaluation history then defines the meaning of the program, and this is a sequence of internal interpreter configurations.

One early use of operational semantics was the work done by John McCarthy in the late 1950s on the semantics of LISP in terms of the lambda calculus. The use of lambda calculus allows the meaning of a program to be expressed using a mathematical interpreter, and this offers precision through the use of mathematics.

The meaning of a program may be given in terms of a hypothetical or virtual machine that performs the set of actions that corresponds to the program. An abstract machine (SECD machine[9]) may be defined to give meaning to phrases in

[9]This virtual stack based machine was originally designed by Peter Landin to evaluate lambda calculus expressions, and it has since been used as a target for several compilers.

the language, and this is done by describing the transitions that they induce on states of the machine.

Operational semantics give an intuitive description of the programming language being studied, and its descriptions are close to real programs. It can play a useful role as a testing tool during the design of new languages, as it is relatively easy to design an interpreter to execute the description of example programs. This allows the effects of new languages or new language features to be simulated and studied through actual execution of the semantic descriptions prior to writing a compiler for the language. Another words, operational semantics can play a role in rapid prototyping during language design, and to get early feedback on the suitability of the language.

One disadvantage of the operational approach is that the meaning of the language is understood in terms of execution: i.e., in terms of interpreter configurations, rather than in an explicit *machine independent specification*. An operational description is just one way to execute programs. Another disadvantage is that the interpreters for non-trivial languages often tend to be large and complex.

A more detailed account of operational semantics is in [6, 7].

12.4.3 Denotational Semantics

Denotational semantics [7] expresses the semantics of a programming language by a translation schema that associates a meaning (denotation) with each program in the language. It maps a program directly to its meaning, and it was originally called mathematical semantics as it provides meaning to programs in terms of mathematical values such as integers, tuples and functions. That is, the meaning of a program is a mathematical object, and an interpreter is not employed. Instead, a valuation function is employed to map a program directly to its meaning, and the denotational description of a programming language is given by a set of *meaning functions* M associated with the constructs of the language (Fig. 12.4).

Each meaning function is of the form $M_T : T \rightarrow D_T$, where T is some construct in the language and D_T is some semantic domain. Many of the meaning functions will be "higher-order": i.e., functions that yield functions as results. The signature of the meaning function is from syntactic domains (i.e., T) to semantic domains (i.e., D_T). A valuation map $V_T : T \rightarrow \mathbf{B}$ may be employed to check the static semantics prior to giving a meaning of the language construct.[10]

A denotational definition is more abstract than an operational definition. It does not specify the computational steps and its exclusive focus is on the programs to the exclusion of the state and other data elements. The state is less visible in denotational specifications.

It was developed by Christopher Strachey and Dana Scott at the Programming Research Group at Oxford, England in the mid-1960s, and their approach to

[10]This is similar to what a compiler does in that if errors are found during the compilation phase, the compiler halts and displays the errors and does not continue with code generation.

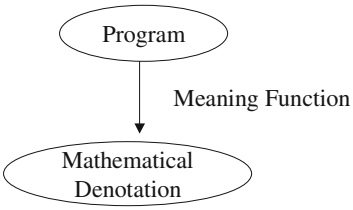

Fig. 12.4 Denotational semantics

semantics is known as the Scott-Strachey approach [8]. It provided a mathematical foundation for the semantics of programming languages.

Dana Scott's contributions included the formulation of domain theory, and this allowed programs containing recursive functions and loops to be given a precise semantics. Each phrase in the language is translated into a mathematical object that is the *denotation* of the phrase. Denotational Semantics has been applied to language design and implementation.

12.5 Lambda Calculus

Functions (discussed in Chap. 2) are an essential part of mathematics, and they play a key role in specifying the semantics of programming language constructs. We discussed partial and total functions in Chap. 2, and a function was defined as a special type of relation, and simple finite functions may be defined as an explicit set of pairs: e.g.,

$$f \triangleq \{(a, 1), (b, 2), (c, 3)\}$$

However, for more complex functions there is a need to define the function more abstractly, rather than listing all of its member pairs. This may be done in a similar manner to set comprehension, where a set is defined in terms of a characteristic property of its members.

Functions may be defined (by comprehension) through a powerful abstract notation known as lambda calculus. This notation was introduced by Alonzo Church in the 1930s to study computability, and lambda calculus provides an abstract framework for describing mathematical functions and their evaluation. It may be used to study function definition, function application, parameter passing and recursion.

Any computable function can be expressed and evaluated using lambda calculus or Turing machines, as these are equivalent formalisms. Lambda calculus uses a small set of transformation rules, and these include

- Alpha-conversion rule (α-conversion)[11]

[11]This essentially expresses that the names of bound variables is unimportant.

- Beta-reduction rule (β-reduction)[12]
- Eta-conversion (η-conversion)[13]

Every expression in the **λ**-calculus stands for a function with a single argument. The argument of the function is itself a function with a single argument, and so on. The definition of a function is anonymous in the calculus. For example, the function that adds one to its argument is usually defined as $f(x) = x + 1$. However, in **λ**-calculus the function is defined as

$$succ \triangleq \lambda x \cdot x + 1$$

The name of the formal argument x is irrelevant and an equivalent definition of the function is $\lambda z \cdot z + 1$. The evaluation of a function f with respect to an argument (e.g. 3) is usually expressed by $f(3)$. In **λ**-calculus this would be written as $(\lambda x \cdot x + 1)\,3$, and this evaluates to $3 + 1 = 4$. Function application is *left associative*: i.e., $f\,x\,y = (f\,x)\,y$. A function of two variables is expressed in lambda calculus as a function of one argument, which returns a function of one argument. This is known as *currying*: e.g., the function $f(x, y) = x + y$ is written as $\lambda x \cdot \lambda y \cdot x + y$. This is often abbreviated to $\lambda x\,y \cdot x + y$.λ-calculus is a simple mathematical system, and its syntax is defined as follows:

```
<exp> ::= <identifier>          |
          λ <identifier>.<exp>  |  --abstraction
          <exp> <exp>           |  --application
          ( <exp> )
```

λ-Calculus's four lines of syntax plus *conversion* rules, are sufficient to define Booleans, integers, data structures and computations on them. It inspired Lisp and modern functional programming languages. The original calculus was untyped, but typed lambda calculi have been introduced in recent years. The typed lambda calculus allows the sets to which the function arguments apply to be specified. For example, the definition of the *plus* function is given as:

$$plus \triangleq \lambda a, b : \mathbb{N} \cdot a + b$$

The lambda calculus makes it possible to express properties of the function without reference to members of the base sets on which the function operates. It allows functional operations such as function composition to be applied, and one key benefit is that the calculus provides powerful support for higher order functions. This is important in the expression of the denotational semantics of the constructs of programming languages.

[12]This essentially expresses the idea of function application.

[13]This essentially expresses the idea that two functions are equal if and only if they give the same results for all arguments.

12.6 Lattices and Order

This section considers some of the mathematical structures used in the definition of the semantic domains used in denotational semantics. These mathematical structures may also be employed to give a secure foundation for recursion (discussed in Chap. 4), and it is essential that the conditions in which recursion may be used safely be understood.

It is natural to ask when presented with a recursive definition whether it means anything at all, and in some cases the answer is negative. Recursive definitions are a powerful and elegant way of giving the denotational semantics of language constructs. The mathematical structures considered in this section include partial orders, total orders, lattices, complete lattices and complete partial orders.

12.6.1 Partially Ordered Sets

A *partial order* $\leq$ on a set P is a binary relation such that for all $x, y, z \in P$ the following properties hold:

(i) $x \leq x$ (*Reflexivity*)
(ii) $x \leq y$ and $y \leq x \Rightarrow x = y$ (*Anti−symmetry*)
(iii) $x \leq y$ and $y \leq z \Rightarrow x \leq z$ (*Transitivity*)

A set P with an order relation $\leq$ is said to be a *partially ordered* set (Fig. 12.5).

Example 12.4 Consider the power set $\mathbb{P}X$, which consists of all the subsets of the set X with the ordering defined by set inclusion. That is, $A \leq B$ if and only if $A \subseteq B$ then $\subseteq$ is a partial order on $\mathbb{P}X$.

A partially ordered set is a *totally ordered* set (*also called chain*) if for all $x, y \in P$ then either $x \leq y$ or $y \leq x$. That is, any two elements of P are directly comparable.

A partially ordered set P is an *anti-chain* if for any x, y in P then $x \leq y$ only if $x = y$. That is, the only elements in P that are comparable to a particular element are the element itself.

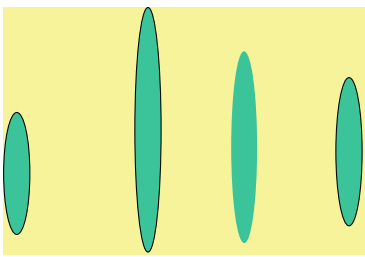

Fig. 12.5 Pictorial representation of a partial order

Maps between Ordered Sets
Let P and Q be partially ordered sets then a map ϕ from P to Q may preserve the order in P and Q. We distinguish between order preserving, order embedding and order isomorphism. These terms are defined as follows:

Order Preserving (or *Monotonic Increasing Function*)
A mapping ϕ: P $\rightarrow$ Q is said to be order preserving if

$$x \leq y \Rightarrow \phi(x) \leq \phi(y)$$

Order Embedding
A mapping ϕ: P $\rightarrow$ Q is said to be an order embedding if

$$x \leq y \text{ in } P \text{ if and only if } \phi(x) \leq \phi(y) \text{in } Q.$$

Order Isomorphism
The mapping ϕ : P $\rightarrow$ Q is an order isomorphism if and only if it is an order embedding mapping onto Q.

Dual of a Partially Ordered Set
The dual of a partially ordered set P (denoted P^{∂}) is a new partially ordered set formed from P where $x \leq y$ holds in P^{∂} if and only if $y \leq x$ holds in P (i.e., P^{∂} is obtained by reversing the order on P).

For each statement about P there is a corresponding statement about P^{∂}. Given any statement Φ about a partially ordered set, then the dual statement Φ^{∂} is obtained by replacing each occurrence of $\leq$ by $\geq$ and vice versa.

Duality Principle
Given that statement Φ is true of a partially ordered set P, then the statement Φ^{∂} is true of P^{∂}.

Maximal and Minimum Elements
Let P be a partially ordered set and let $Q \subseteq P$ then

(i) $a \in Q$ is a *maximal* element of Q if $a \leq x \in Q \Rightarrow a = x$.
(ii) $a \in Q$ is the *greatest (*or *maximum)* element of Q if $a \geq x$ for every $x \in Q$, and in that case we write $a = \max Q$

A *minimal* element of Q and the *least (*or *minimum)* are defined dually by reversing the order. The greatest element (if it exists) is called the top element and is denoted by $\top$. The least element (if it exists) is called the bottom element and is denoted by $\bot$.

Example 12.5 Let X be a set and consider $\mathbb{P}X$ the set of all subsets of X with the ordering defined by set inclusion. The top element $\top$ is given by X, and the bottom element $\bot$ is given by $\varnothing$.

A finite totally ordered set always has top and bottom elements, but an infinite chain need not have.

12.6.2 Lattices

Let P be a partially ordered set and let $S \subseteq P$. An element $x \in P$ is an upper bound of S if $s \leq x$ for all $s \in S$. A lower bound is defined similarly.

The set of all upper bounds for S is denoted by S^u, and the set of all lower bounds for S is denoted by S^l.

$$S^u = \{x \in \mathrm{P} | (\forall s \in \mathrm{S}) s \leq x\}$$
$$S^l = \{x \in \mathrm{P} \mid (\forall s \in \mathrm{S}) s \geq x\}$$

If S^u has a least element x then x is called the *least upper bound* of S. Similarly, if S^l has a greatest element x then x is called the *greatest lower bound* of S.

Another words, x is the least upper bound of S if

(i) x is an upper bound of S.
(ii) $x \leq y$ for all upper bounds y of S

The least upper bound of S is also called the *supremum* of S denoted (sup S), and the greatest lower bound is also called the infimum of S, and is denoted by inf S.

Join and Meet Operations

The *join* of x and y (denoted by $x \vee y$) is given by $\sup\{x, y\}$ when it exists. The *meet* of x and y (denoted by $x \wedge y$) is given by $\inf\{x, y\}$ when it exists.

The supremum of S is denoted by $\vee$S, and the infimum of S is denoted by $\in S$.

Definition

Let P be a non-empty partially ordered set then

(i) If $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$ then P is called a *lattice*.
(ii) If $\vee S$ and $\wedge S$ exist for all $S \subseteq P$ then P is called a *complete lattice*

Every non-empty finite subset of a lattice has a meet and a join (inductive argument can be used), and every finite lattice is a complete lattice. Further, any complete lattice is bounded—i.e., it has top and bottom elements (Fig. 12.6).

Example 12.6 Let X be a set and consider $\mathbb{P}X$ the set of all subsets of X with the ordering defined by set inclusion. Then $\mathbb{P}X$ is a complete lattice in which

$$\vee \{A_i | i \in \mathrm{I}\} = \cup A_i$$
$$\wedge \{A_i | i \in \mathrm{I}\} = \cap A_i$$

Consider the set of natural numbers $\mathbb{N}$ and consider the usual ordering of $<$. Then $\mathbb{N}$ is a lattice with the join and meet operations defined as

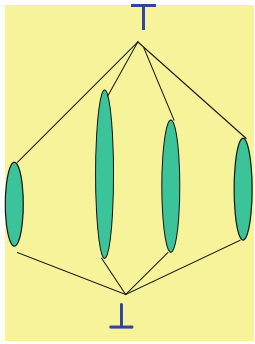

Fig. 12.6 Pictorial representation of a complete lattice

$$x \vee y = \max(\mathrm{x}, \mathrm{y})$$
$$x \wedge y = \min(\mathrm{x}, \mathrm{y})$$

Another possible definition of the meet and join operations are in terms of the greatest common multiple and least common divisor.

$$x \vee y = \mathrm{lcm}(\mathrm{x}, \mathrm{y})$$
$$x \wedge y = \mathrm{gcd}(\mathrm{x}, \mathrm{y})$$

12.6.3 Complete Partial Orders

Let S be a non-empty subset of a partially ordered set P. Then

(i) S is said to be a *directed set* if for every finite subset F of S there exists $z \in S$ such that $z \in F^u$.
(ii) S is said to be *consistent* if for every finite subset F of S there exists $z \in P$ such that $z \in F^u$

A partially ordered set P is a *complete partial order* (CPO) if:

(i) P has a bottom element $\perp$
(ii) $\bigvee D$ exists for each directed subset D of P

The simplest example of a directed set is a chain, and we note that any complete lattice is a complete partial order, and that any finite lattice is a complete lattice.

12.6.4 Recursion

Recursive definitions arise frequently in programs and offer an elegant way to define routines and data types. A recursive routine contains a direct or indirect call to itself, and a recursive data type contains a direct or indirect reference to specimens of the same type. Recursion needs to be used with care, as there is always a danger that the recursive definition may be circular (i.e., defines nothing). It is therefore important to investigate when a recursive definition may be used safely, and to give a mathematical definition of recursion.

The control flow in a recursive routine must contain at least one non-recursive branch since if all possible branches included a recursive form the routine could never terminate. The value of at least one argument in the recursive call is different from the initial value of the formal argument as otherwise the recursive call would result in the same sequence of events and therefore would never terminate.

The mathematical meaning of recursion is defined in terms of *fixed point theory*, which is concerned with determining solutions to equations of the form $x = \tau(x)$, where the function τ is of the form $\tau : X \to X$.

A recursive definition may be interpreted as a fixpoint equation of the form $f = \Phi(f)$; i.e., the fixpoint of a high-level functional Φ that takes a function as an argument. For example, consider the functional Φ defined as follows:

$$\Phi \mathrel{\underline{\Delta}} \lambda f \lambda n \cdot \text{if } n = 0 \text{ then } 1 \text{ else } n^* f(n-1)$$

Then a fixpoint of Φ is a function f such that $f = \Phi(f)$ or another words

$$f = \lambda n \cdot \text{if } n = 0 \text{ then } 1 \text{ else } n^* f(n-1)$$

Clearly, the factorial function is a fixpoint of Φ, and it is the only total function that is a fixpoint. The solution of the equation $f = \Phi(f)$ (where Φ has a fixpoint) is determined as the limit f of the sequence of functions $f_0, f_1, f_2, \ldots$, where the f_i are defined inductively as

$$\begin{aligned} f_0 &\mathrel{\underline{\Delta}} \varnothing \qquad (\text{the empty partial function}) \\ f_i &\mathrel{\underline{\Delta}} \Phi(f_{i-1}) \end{aligned}$$

Each f_i may be viewed as a successive approximation to the true solution f of the fixpoint equation, with each f_i bringing a little more information on the solution than its predecessor f_{i-1}.

The function f_i is defined for one more value than f_{i-1}, and gives the same result for any value for which they are both defined. The definition of the factorial function is thus built up as follows:

$$
\begin{aligned}
&f_0 \,\underline{\Delta}\, \varnothing && \text{(the empty partial function)}\\
&f_1 \,\underline{\Delta}\, \{0 \to 1\}\\
&f_2 \,\underline{\Delta}\, \{0 \to 1, 1 \to 1\}\\
&f_3 \,\underline{\Delta}\, \{0 \to 1, 1 \to 1, 2 \to 2\}\\
&f_4 \,\underline{\Delta}\, \{0 \to 1, 1 \to 1, 2 \to 2, 3 \to 6\}\\
&\;\vdots \qquad\qquad \vdots\\
&\;\vdots
\end{aligned}
$$

For every i, the domain of f_i is the interval 1, 2, … $i - 1$ and $f_i\,(n) = n!$ for any n in this interval. Another word f_i is the factorial function restricted to the interval 1, 2, … $i - 1$. The sequence of f_i may be viewed as successive approximations of the true solution of the fixpoint equation (which is the factorial function), with each f_i bringing defined for one more value than its predecessor f_{i-1}, and defining the same result for any value for which they are both defined.

The candidate fixpoint f_∞ is the limit of the sequence of functions f_i, and is the union of all the elements in the sequence. It may be written as follows:

$$f_\infty \,\underline{\Delta}\, \varnothing \cup \Phi(\varnothing) \cup \Phi(\Phi(\varnothing)) \cup \ldots = \cup_{i:\mathbb{N}} f_i,$$

where the sequence f_i is defined inductively as

$$
\begin{aligned}
&f_0 \,\underline{\Delta}\, \varnothing && \text{(the empty partial function)}\\
&f_{i+1} \,\underline{\Delta}\, f_i \cup \Phi(f_i)
\end{aligned}
$$

This forms a subset chain where each element is a subset of the next, and it follows by induction that

$$f_{i+1} = \cup_{j:0\ldots i} \Phi(f_i)$$

A general technique for solving fixpoint equations of the form $h = \tau(h)$ for some functional τ is to start with the least-defined function $\varnothing$ and iterate with τ. The union of all the functions obtained as successive sequence elements is the fixpoint.

The conditions in which f_∞ is a fixpoint of Φ is the requirement for $\Phi(f_\infty) = f_\infty$. This is equivalent to:

$$
\begin{aligned}
\Phi(\cup_{i:\mathbb{N}} f_i) &= \cup_{i:\mathbb{N}} f_i\\
\Phi(\cup_{i:\mathbb{N}} f_i) &= \cup_{i:\mathbb{N}} \Phi(f_i)
\end{aligned}
$$

A sufficient point for Φ to have a fixpoint is that the property $\Phi(\cup_{i:\mathbb{N}} f_i) = \cup_{i:\mathbb{N}} \Phi(f_i)$ holds for any subset chain f_i.

We discussed recursion earlier in Chap. 4, and a more detailed account on the mathematics of recursion is in Chap. 8 of [7].

12.7 Review Questions

1. Explain the difference between syntax and semantics.
2. Describe the Chomsky hierarchy of grammars and give examples of each type.
3. Show that a grammar may be ambiguous leading to two difference parse trees. What problems does this create and how should it be dealt with?
4. Describe axiomatic semantics, operation semantics and denotational semantics and explain the differences between them.
5. Explain partial orders, lattices and complete partial orders. Give examples of each.
6. Show how the meaning of recursion is defined with fixpoint theory.

12.8 Summary

This chapter considered two key parts to any programming language namely syntax and semantics. The syntax of the language is concerned with the production of grammatically correct programs in the language, whereas the semantics of the language is deeper and is concerned with the meaning of what has been written by the programmer.

There are several approaches to defining the semantics of programming languages, and these include axiomatic, operational and denotational semantics. Axiomatic semantics is concerned with defining properties of the language in terms of axioms; operational semantics is concerned with defining the meaning of the language in terms of an interpreter; and denotational semantics is concerned with defining the meaning of the phrases in a language by the denotation or mathematical meaning of the phrase.

Compilers are programs that translate a program that is written in some programming language into another form. It involves syntax analysis and parsing to check the syntactic validity of the program; semantic analysis to determine what the program should do; optimization to improve the speed and performance of the compiler; and code generation in some target language.

Various mathematical structures including partial orders, total orders, lattices and complete partial orders were considered. These are useful in the definition of the denotational semantics of a language, and in giving a mathematical interpretation of recursion.

References

1. Introduction to the History of Computing. Gerard O'Regan. Springer Verlag. 2016.
2. Report on the Algorithmic Language, ALGOL 60. Edited by P. Naur. Communications of the ACM, 3(5):299–314, 1960.
3. An Axiomatic Basis for Computer Programming. C.A.R. Hoare. Communications of the ACM. 12(10):576–585. 1969.
4. Assigning Meanings to Programs. Robert Floyd. Proceedings of Symposia in Applied Mathematics, (19), 19-32. 1967.
5. Mathematical Approaches to Software Quality. Gerard O' Regan. Springer. 2006.
6. A Structural Approach to Operational Semantics. Gordon Plotkin. Technical Report DAIM FN-19. Computer Science Department. AarhusUniversity, Denmark. 1981.
7. Introduction to the Theory of Programming Languages. Bertrand Meyer. Prentice Hall. 1990.
8. Denotational Semantics. The Scott-Strachey Approach to Programming Language Theory. Joseph Stoy. MIT Press. 1977.