Chapter 3
Foundations of Tree- and Graph-Based Abstract Syntax

RICHARD PAIGE.[1]

Abstract A software language can be regarded as a set of structured elements with some associated meaning. A language's *syntax* defines its elements and their structure. We may speak of string, tree, and graph languages – to convey the nature of the elements' structure. One may distinguish two forms of syntax: *concrete* versus *abstract syntax*. The former is tailored towards processing (reading, writing, editing) by humans who are language users; the latter is tailored towards processing (parsing, analyzing, transforming, generating) by programs that are authored by language implementers. In this chapter, we cover the foundations of abstract syntax. This includes the notion of *conformance* of terms (trees) or models (graphs) to *signatures* or *metamodels*. The proposed notations for signatures and metamodels correspond to proper software languages in themselves, giving rise to a *metametalevel* that we develop as well. We defer implementation aspects of abstract syntax, coverage of concrete syntax, and semantics of languages to later chapters.

[1] The software language engineering community aims to integrate more specialized communities. Richard Paige is a modelware "stronghold"; he has contributed to pretty much everything modelware, for example, model merging and composition [6, 1], model evolution [9], model to text and vice versa [10, 5], and visual syntax [7]. Richard Paige definitely advances community integration in his work, as exemplified by his tutorial on metamodeling for grammar researchers [8] or his Twitter persona.

R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7_3

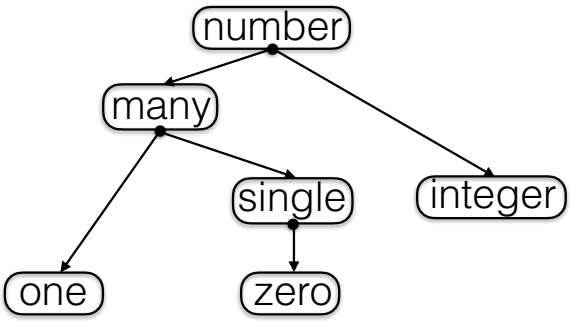

Fig. 3.1 The binary number "10" as a tree.

3.1 Tree-Based Abstract Syntax

The abstract syntax of a language is concerned with the tree- or graph-like structure of language elements. Abstract syntax definitions define languages as sets of trees or graphs. We need a *membership test* to decide whether a given tree or graph is actually an element of the language of interest. To this end, we also speak of conformance of a tree or a graph to a syntax definition, such as a signature or a metamodel. In this section, we focus on tree-based abstract syntax. In the next section, we focus on graph-based abstract syntax.

3.1.1 Trees versus Terms

Let us define the abstract syntax of binary numbers in the sense of a dedicated language: *BNL* (Binary Number Language). We begin by exercising the abstract syntactical representation of numbers; see Fig. 3.1 for a tree-based representation. That is, we use node-labeled rose trees, i.e., trees with any number of subtrees; symbols serve as infos of nodes. The symbols used in a such a tree classify the subtrees at hand. Each symbol has an associated arity (i.e., number of children). The following symbols (with the arity given next to each symbol) are used in the figure:

- *number/2*: The two children are the integer and fractional parts.
- *integer/0*: In fact, there is no fractional part.
- *many/2*: The first child is a digit; the second child is a sequence of digits.
- *single/1*: The child is a digit.
- *one/0*: The digit "1".
- *zero/0*: The digit "0".

For most of this book, we will prefer term-based representations of such trees, but we assume that it is obvious how to go back and forth between the term- and tree-based views. The term-based counterpart to Fig. 3.1 is shown below.

Illustration 3.1 (The binary number "10" as a term)

Term resource languages/BNL/samples/10.term

```
number(
  many(
    one,
    single(
      zero)),
  integer).
```

That is, a simple prefix notation is used here for terms. A function symbol serves as the prefix (e.g., number or many) and the arguments (say, subterms or subtrees) are enclosed in parentheses. Indentation is used here as a visual hint at the tree-like structure.

3.1.2 A Basic Signature Notation

One could use informal text to define (abstract) syntax. We prefer more formal means. We use (many-sorted algebraic) *signatures* [11, 4] for defining tree (term) languages. We propose a specific notation here: *BSL* (*B*asic *S*ignature *L*anguage). This is a "fabricated" language of this book, but similar notations are part of established languages for formal specification, algebraic specification, term rewriting, and functional programming. Let us apply BSL to BNL as follows.

Illustration 3.2 (Abstract syntax of BNL)

BSL resource languages/BNL/as.bsl

```
symbol number: bits × rest → number ; // A binary number
symbol single: bit → bits ; // A single bit
symbol many: bit × bits → bits ; // More than one bit
symbol zero: → bit ; // The zero bit
symbol one: → bit ; // The nonzero bit
symbol integer: → rest ; // An integer number
symbol rational: bits → rest ; // A rational number
```

A (BSL-based) many-sorted algebraic *signature* is a list of types of function symbols. Each *type* (or *profile*) consists of the function symbol, a list of argument sorts (say, argument types), and a result sort (say, the result type). A function symbol with zero argument sorts is also called a constant symbol. Sorts are not separately declared; they are implicitly introduced, as they appear in types of function symbols.

An abstract syntax abstracts from representational details that surface in a concrete syntax. That is, an abstract syntax definition does not necessarily cover all notational variations and details that may just be part of the concrete syntax for language users' convenience. In a very limited manner, such abstraction can be observed even in the trivial signature for BNL. That is, no separator between integer and fractional parts is expressed. We will consider more interesting forms of abstraction later, when we study concrete syntax (Chapter 6).

The intended interpretation of a signature is the set of terms that can be built recursively from the function symbols with the constants as base cases. Of course, such term construction must obey the types of the symbols.

Simply because we can switch back and forth between trees and terms, as illustrated above, we can also view a signature as a tree grammar [3], i.e., a grammar that defines (generates) a set of trees.

3.1.3 Abstract Syntax Trees

When terms are generated by a signature meant for abstract syntax definition, then we refer to these terms also as abstract syntax trees (ASTs). Given a signature and a term, there is a simple way to show that the term is indeed a valid AST with regard to the signature, i.e., a term "generated" by the signature. This is the case if each function symbol of the term is declared by the signature and it is used with the declared arity (i.e., number of subterms), and each subterm is built from a function symbol, with the result sort agreeing with the argument sort of the position in which the subterm occurs, as illustrated for binary numbers and their syntax below.

Illustration 3.3 (A term generated by a signature)
The term of Illustration 3.1 is a term generated by the signature of Illustration 3.2 in accordance with the following evidence:

• *number(*	symbol number: bits × rest → number ;
– *many(*	symbol many: bit × bits → bits ;
· *one,*	symbol one: → bit ;
· *single(*	symbol single: bit → bits ;
· *zero)),*	symbol zero: → bit ;
– *integer).*	symbol integer : → rest ;

Exercise 3.1 (BNL with negative numbers) [Basic level]
Extend the signature of Illustration 3.2 to enable negative binary numbers.

Exercise 3.2 (Redundancy in an abstract syntax) [Intermediate level]
The obvious solution to Exercise 3.1 would enable a redundant representation of "0" (i.e., zero) with a positive and a negative zero. Further, the initial abstract syntax definition in Illustration 3.2 permits a form of redundancy. That is, bit sequences with leading zeros before "." or bit sequences with trailing zeros after "." can be represented, for example, "0010" instead of just "10". Define an abstract syntax that avoids both forms of redundancy.

3.1.4 An Extended Signature Notation

Consider again the signature in Illustration 3.2. There are issues of conciseness. That is, optionality of the fractional part is encoded by the function symbols integer and rational, subject to an "auxiliary" sort rest. Sequences of bits are encoded by the function symbols single and many, subject to an "auxiliary" sort bits. These are recurring idioms which can be expressed more concisely in an extended signature notation. We propose a specific notation here: *ESL* (*E*xtended *S*ignature *L*anguage). This is a "fabricated" language of this book, but the notation is again inspired by established languages for algebraic specification, term rewriting, and functional programming. We begin by exercising a more concise representation of numbers. Let us apply ESL to BNL as follows.

Illustration 3.4 (More concise abstract syntactical representation of "10")

Term resource languages/BNL/ESL/samples/10.term

```
([one, zero], []).
```

We use standard notation for tuples ("$(\cdots)$") and lists ("$[\cdots]$"). Numbers are pairs of integer and fractional parts; both parts are simply sequences of bits, as captured by the following signature.

Illustration 3.5 (More concise abstract syntax of BNL)

ESL resource languages/BNL/ESL/as.esl

```
type number = bit+ × bit* ;
symbol zero: → bit ;
symbol one: → bit ;
```

These are all the constructs of ESL:

- symbol declarations as in BSL;

- type declarations to define (say, alias) types in terms of other types;
- list types $t*$ and $t+$ for a given type t;
- optional types $t?$ for a given type t;
- tuple types $t_1 \times \ldots \times t_n$ for given types $t_1, \ldots, t_n$;
- primitive types:
  - boolean;
  - integer;
  - float;
  - string;
  - term ("all conceivable terms"; see Definition 3.3).

3.1.5 Illustrative Examples of Signatures

We define the tree-based abstract syntax of a few more languages here. We revisit ("fabricated") languages that were introduced in Chapter 1.

3.1.5.1 Syntax of Simple Expressions

Let us define the abstract syntax of the expression language BTL.

Illustration 3.6 (Abstract syntax of BTL)

BSL resource languages/BTL/as.bsl

```
symbol true : → expr ; // The Boolean "true"
symbol false : → expr ; // The Boolean "false"
symbol zero : → expr ; // The natural number zero
symbol succ : expr → expr ; // Successor of a natural number
symbol pred : expr → expr ; // Predecessor of a natural number
symbol iszero : expr → expr ; // Test for a number to be zero
symbol if : expr × expr × expr → expr ; // Conditional
```

It may be interesting to reflect on the conceivable differences between abstract and concrete syntax. In particular, a concrete syntax may favor "mixfix" syntax "*if* ... *then* ... *else* ..." for the conditional form. In an abstract syntax, we use prefix notation universally.

3.1.5.2 Syntax of Simple Imperative Programs

Let us define the abstract syntax of the imperative programming language BIPL.

Illustration 3.7 (Abstract syntax of BIPL)

ESL resource languages/BIPL/as.esl

```
// Statements
symbol skip : → stmt ;
symbol assign : string × expr → stmt ;
symbol seq : stmt × stmt → stmt ;
symbol if : expr × stmt × stmt → stmt ;
symbol while : expr × stmt → stmt ;

// Expressions
symbol intconst : integer → expr ;
symbol var : string → expr ;
symbol unary : uop × expr → expr ;
symbol binary : bop × expr × expr → expr ;

// Unary operators
symbol negate : → uop ;
symbol not : → uop ;

// Binary operators
symbol or : → bop ;
symbol and : → bop ;
symbol lt : → bop ;
symbol leq : → bop ;
symbol eq : → bop ;
symbol geq : → bop ;
symbol gt : → bop ;
symbol add : → bop ;
symbol sub : → bop ;
symbol mul : → bop ;
```

Thus, there are symbols for the empty statement (“skip”), assignment, if-then-else, while-loops, and sequences of statements. There are symbols for expression forms and operator symbols. We use the primitive type integer for integer literals in the abstract syntax. We use the primitive type string for variable names in the abstract syntax.

3.1.5.3 Syntax of Simple Functional Programs

Let us define the abstract syntax of the functional programming language BFPL.

Illustration 3.8 (Abstract syntax of BFPL)

ESL resource languages/BFPL/as.esl

```
// Program = typed functions + main expression
type program = functions × expr ;
type functions = function* ;
type function = string × funsig × fundef ;
type funsig = simpletype* × simpletype ;
type fundef = string* × expr ;

// Simple types
symbol inttype : → simpletype ;
symbol booltype : → simpletype ;

// Expressions
symbol intconst : integer → expr ;
symbol boolconst : boolean → expr ;
symbol arg : string → expr ;
symbol if : expr × expr × expr → expr ;
symbol unary : uop × expr → expr ;
symbol binary : bop × expr × expr → expr ;
symbol apply : string × expr* → expr ;

// Unary and binary operators
...
```

3.1.5.4 Syntax of Finite State Machines

Let us define the abstract syntax of the domain-specific modeling language FSML.

Illustration 3.9 (Abstract syntax of FSML)

ESL resource languages/FSML/as.esl

```
type fsm = state* ;
type state = initial × stateid × transition* ;
type initial = boolean ;
type transition = event × action? × stateid ;
type stateid = string ;
type event = string ;
type action = string ;
```

Because FSMs have such a simple structure, we can define the abstract syntax solely in terms of lists, tuples, optional elements, and primitive types – without introducing any FSML-specific function symbols. We represent the familiar turnstile example according to this signature as follows.

Illustration 3.10 (Abstract syntactical representation of a turnstile FSM)

Term resource languages/FSML/sample.term

```
[
  (true,locked,[
   (ticket,[collect],unlocked),
   (pass,[alarm],exception)]),

  (false,unlocked,[
    (ticket,[eject],unlocked),
    (pass,[],locked)]),

  (false,exception,[
    (ticket,[eject],exception),
    (pass,[],exception),
    (mute,[],exception),
    (release,[],locked)])
].
```

3.1.6 Languages as Sets of Terms

We may complement the informal explanation of tree-based abstract syntax, given so far, with formal definitions.

Definition 3.1 ((Many-sorted algebraic) signature) *A signature Σ is a triple $\langle F, S, P\rangle$, where F is a finite set of function symbols, S is a finite set of sorts, and P is a finite set of types of function symbols ("profiles") as a subset of $F \times S^* \times S$. There are no distinct types $\langle f_1, a_1, s_1\rangle$, $\langle f_2, a_2, s_2\rangle \in P$ with $f_1 = f_2$. For any $\langle c, \langle\rangle, s\rangle \in P$ (i.e., a type with the empty sequence of argument sorts), we say that c is a constant symbol.*

Definition 3.2 (Terms of a sort) *Given a signature $\Sigma = \langle F, S, P\rangle$, the set of terms of a given sort $s \in S$, also denoted by Φ_s, is defined as the smallest set closed under these rules:*

- *If c is a constant symbol of sort $s \in S$, i.e., $\langle c, \langle\rangle, s\rangle \in P$, then $c \in \Phi_s$.*
- *If $\langle f, \langle s_1, \ldots, s_n\rangle, s\rangle \in P$, $n > 0$, s, s_1, $\ldots$, $s_n \in S$, $t_1 \in \Phi_{s_1}$, $\ldots$, $t_n \in \Phi_{s_n}$, then $f(t_1, \ldots, t_n) \in \Phi_s$.*

3.1.7 Conformance to a Signature

We will now set out the concept of conformance to decide whether a given term is actually an element of a certain sort for a given signature. To this end, we assume a possibly infinite set F^U of candidate function symbols, and we define a set $\Sigma(F^U)$ of all terms that can be built from F^U so that we can refer to $\Sigma(F^U)$ as the universe on which to define conformance. As the terms in $\Sigma(F^U)$ are constructed "before" distinguishing any sorts, we also call them pre-terms.

Definition 3.3 (Pre-terms) *Given a set F^U of candidate function symbols, the set of all pre-terms, also denoted by $\Sigma(F^U)$, is defined as the smallest set closed under these rules:*

- $F^U \subset \Sigma(F^U)$.
- *For all $f \in F^U, n > 0$, if $t_1, \ldots, t_n \in \Sigma(F^U)$, then $f(t_1, \ldots, t_n) \in \Sigma(F^U)$.*

Conformance is easily defined in an algorithmic manner.

Definition 3.4 (Conformance of a pre-term to a signature) *Given a set F^U of candidate function symbols, a pre-term $t \in \Sigma(F^U)$ and a signature $\Sigma = \langle F, S, P \rangle$ with $F \subseteq F^U$, we say that t is of sort $s \in S$ and conforms to Σ, also denoted by $\Sigma \vdash t : s$, if:*

- *$t \in F^U$ and $\langle t, \langle\rangle, s\rangle \in P$, or*
- *t is of the form $f(t_1, \ldots, t_n)$ such that*
  - *$f \in F^U$, and*
  - *$t_1, \ldots, t_n \in \Sigma(F^U)$, and*
  - *$\langle t, \langle s_1, \ldots, s_n\rangle, s\rangle \in P$ for some $s_1, \ldots, s_n \in S$, and*
  - *$\Sigma \vdash t_1 : s_1, \ldots, \Sigma \vdash t_n : s_n$.*

Operationally, given a pre-term, its sort is the result sort of the outermost function symbol, while the sorts of the subterms must be checked recursively to ensure that they are equal to the argument sorts of the function symbol.

3.2 Graph-Based Abstract Syntax

Many if not most software languages involve conceptual references in that one may want to refer in one position of a compound element to another position. Thus, one may need to model *reference relationships*. Tree-based abstract syntax is limited

in this respect, as language elements are simply decomposed into parts in a tree-like manner; references need to be encoded and resolved programmatically. Graph-based abstract syntax distinguishes whole-part and referencing relationships. In this section, we use a simple *metamodeling* notation for defining graph languages.

3.2.1 Trees versus Graphs

We illustrate graph-based abstract syntax here with the Buddy Language (BL) for modeling persons with their names and buddy relationships. This is a "fabricated" language of this book, but it can be viewed as a simple variant of the popular example *FOAF* ("friends of a friend"). We begin with a tree-based variant of the Buddy Language, as illustrated below.

Illustration 3.11 (Two buddies as a term)

Term resource languages/BL/samples/small-world.term

```
[ (joe, [bill]), (bill, [joe]) ].
```

Joe's buddy is Bill. Bill's buddy is Joe. Thus, the idea is that persons' names function as "ids" of persons. We use these ids in the term-based representation to refer to buddies. Arguably, names are not necessarily unique. Thus, in practice, we should use other means of identification, for example, social security numbers, but we will keep things simple here. Thus, the tree-based abstract syntax of the Buddy Language may be defined as follows.

Illustration 3.12 (Tree-based abstract syntax of BL)

ESL resource languages/BL/as.esl

```
type world = person* ;
type person = string × string? ;
```

The expected meaning of the names as acting as references is not captured by the signature. We may be able to define a separate analysis that checks names for consistent use, but such an analysis is not prescribed by the signature and not standardized as part of conformance.

Once we use a graph-based syntax we can model references explicitly. This is demonstrated now for a graph-based variant of the Buddy Language. The following illustration shows a graph rather than a tree for Joe and Bill's buddy relationships.

Illustration 3.13 (Two buddies as a graph)

Graph resource languages/BL/samples/small-world.graph

```
0 & { class : world,
  persons : [
    1 & { class : person, name : 'joe', buddy : [ #2 ] },
    2 & { class : person, name : 'bill', buddy : [ #1 ] } ] }.
```

Thus, a graph is essentially a container of potentially labeled sub-graphs with base cases for references (such as "#2") or primitive values (such as "joe"). In the example, the complete graph models a "world"; there are two subgraphs for persons. List brackets "$[\cdots]$" are used here to deal with optionality of buddies. (In general, list brackets may also be used to deal with list cardinalities, i.e., "+" and "*".) In the example, the references to buddies are optional. A (sub-) graph is made referable by assigning an id to it, as expressed by the notation "1 & ..." above. The labels for sub-graphs (such as "name" or "buddy") can be thought of as selectors for those sub-graphs. We use a special label "class" to record the type of a sub-graph. Types are to be described eventually by a metamodel.

3.2.2 Languages as Sets of Graphs

Let us define the set of (resolvable) pre-graphs, i.e., all as yet "untyped" graphs, akin to the pre-terms of Definition 3.3 for tree-based abstract syntax. For simplicity, the definition does not cover primitive values and lists.

Definition 3.5 (Pre-graphs) *Given sets L^U and R^U, referred to as (universes of) labels (for sub-graphs) and ids (for referencing), the set of all pre-graphs, also denoted by $M(L^U, R^U)$, is defined as the smallest set closed under this rule:*

- *If $r \in R^U$ and*
  $g_1, \ldots, g_n \in M(L^U, R^U) \cup R^U$ and
  distinct $l_1, \ldots, l_n \in L^U$,
  then $\langle r, \{\langle l_1, g_1\rangle, \ldots, \langle l_n, g_n\rangle\}\rangle \in M(L^U, R^U)$.

The component r of the pre-graph is referred to as its id.

Thus, pre-graphs are essentially sets of labeled pre-graphs with an id for the collection. There is the special case in which a sub-pre-graph is not a collection (not even an empty collection), but it is a reference, which is why the g_i may also be drawn from R^U in the definition.

We should impose an important constraint on pre-graphs: they should be resolvable, such that assigned ids are distinct, and, for each id used as a reference, there

should be a sub-pre-graph with that assigned id. Let us first define sets of sub-pre-graphs and references of a given pre-graph which we need to refer to when formulating the constraint described above.

Definition 3.6 (Sub-pre-graphs and pre-graph references) *Given a pre-graph $g \in M(L^U, R^U)$, the multi-set of its sub-pre-graphs, also denoted by $SPG(g)$, is defined as the smallest multi-set closed under these rules:*

- $g \in SPG(g)$.
- *If $\langle r, \{\langle l_1, g_1\rangle, \ldots, \langle l_n, g_n\rangle\}\rangle \in SPG(g)$ for appropriate r, $l_1, \ldots, l_n$, and $g_1, \ldots, g_n$, then $SPG(g_i) \subset SPG(g)$ for $1 \leq i \leq n$ and g_i is a pre-graph (and not a reference).*

The set of pre-graph references, also denoted by $PGR(g)$, is defined as the smallest set closed under this rule:

- *If $\langle r, \{\langle l_1, g_1\rangle, \ldots, \langle l_n, g_n\rangle\}\rangle \in SPG(g)$ for appropriate r, $l_1, \ldots, l_n$, and $g_1, \ldots, g_n$, then $g_i \in PGR(g)$ for $1 \leq i \leq n$ and g_i is a reference (and not a pre-graph).*

We use a multi-set rather than a plain set when gathering sub-pre-graphs because, in this manner, we can "observe" identical sub-pre-graphs (with also the same id); see the following definition.

Definition 3.7 (Resolvable pre-graph) *A pre-graph $g \in M(L^U, R^U)$ is said to be resolvable if the following conditions hold:*

- *For all distinct $g_1, g_2 \in SPG(g)$, the ids of g_1 and g_2 are distinct.*
- *$SPG(g)$ is a set (rather than a proper multi-set).*
- *For all $r \in PGR(g)$, there exists a $g' \in SPG(g)$ such that its id equals r.*

The first condition condition requires that all sub-pre-graphs have a distinct id. Additionally, the second rules out (completely) identical sub-pre-graphs. The third condition requires that each reference used equals the id of one sub-pre-graph.

3.2.3 A Metamodeling Notation

On top of this basic formal model, we can define metamodels for describing sets of graphs in the same way as signatures describe sets of terms.

We propose a specific notation here: *MML* (MetaModeling Language). This is a "fabricated" language of this book, but the notation is inspired by established metamodeling frameworks such as *EMF*'s metamodeling language *Ecore*[2]. Let us

[2] https://eclipse.org/modeling/emf/

reapproach the Buddy Language; see the following metamodel for graph-based as opposed to tree-based abstract syntax.

Illustration 3.14 (Graph-based abstract syntax of BL)

MML resource languages/BL/gbl.mml

```
class world { part persons : person* ; }
class person {
  value name : string ;
  reference buddy : person? ;
}
datatype string ;
```

That is, a metamodel describes a set of classes with members for values, parts, and references with an associated cardinality ("?" for optional members, "*" and "+" for lists). In the metamodel shown, the persons of a world are modeled as parts, whereas the buddy of a person is modeled as a reference. While it is not demonstrated by the simple metamodel at hand, classes may also be related by inheritance so that a sub-class inherits all members from its super-class. Classes may also be abstract to express the fact that they cannot be instantiated.

Let us also show a metamodel for BL which does not involve references, but it encodes references by means of persons' names, just like in the case of the earlier signature-based model.

Illustration 3.15 (A metamodel for BL without references)

MML resource languages/BL/tbl.mml

```
class world { part persons : person* ; }
class person {
  value name : string ;
  value buddy : string? ;
}
datatype string ;
```

3.2.4 Conformance to a Metamodel

We omit the definition of conformance of a pre-graph to a metamodel; it is relatively straightforward and easy to define and implement; it is similar to conformance of a pre-term to a signature. In particular, each sub-graph would need to conform to its class in terms of the members of the class, as prescribed by the metamodel.

Exercise 3.3 (Pre-graph-to-metamodel conformance) [Intermediate level]
Define pre-graph-to-metamodel conformance.

When graphs conform to a metamodel meant for abstract syntax definition, then we refer to these graphs as abstract syntax graphs (ASGs). We may also say "model" instead of "graph".

Exercise 3.4 (Metamodeling with EMF) [Intermediate level]
Study the Eclipse Modeling Framework (EMF) and define an Ecore-based abstract syntax of BL.

3.2.5 Illustrative Examples of Metamodels

Many software languages involve references conceptually once we take into account the meaning of language elements or the assumed result of checking well-formedness.

3.2.5.1 Syntax of Finite State Machines

The target state ids in FSML's transitions may be viewed as references to states. Accordingly, we may also define a graph-based abstract syntax for FSML as shown below. We begin with a corresponding metamodel, followed by a model (a graph) for the turnstile FSM.

Illustration 3.16 (A metamodel for FSML with references)

MML resource languages/FSML/mm.mml

```
class fsm { part states : state* ; }
class state {
  value initial : boolean ;
  value stateid : string ;
  part transitions : transition* ;
}
class transition {
  value event : string ;
  value action : string? ;
  reference target : state ;
}
datatype boolean ;
datatype string ;
```

Illustration 3.17 (Graph-based representation of a turnstile FSM)

Graph resource languages/FSML/sample.graph

```
{
 class : fsm,
 states : [
  1 & {
   class : state,
   initial : true,
   stateid : 'locked',
   transitions : [
    {
     class : transition,
     event : 'ticket',
     action : ['collect'],
     target : #2
    },
    ...
   ]
  },
  2 & { ...
  },
  3 & { ...
  }
 ]
}.
```

3.2.5.2 Syntax of Simple Functional Programs

Exercise 3.5 (A metamodel for BFPL) [Basic level]
Consider function applications in the functional language BFPL. The name in a function application can be understood as a reference to the corresponding function declaration. Accordingly, devise a graph-based abstract syntax for BFPL by turning the signature of Illustration 3.8 into a metamodel with references for functions used in function applications.

3.3 Context Conditions

The abstract syntax definitions discussed thus far do not capture all the constraints that one would want to assume for the relevant languages. Here are a few examples:

Imperative programs (BIPL) We may require that a program should only use a variable in an expression once the variable has been assigned a value. Also, when operators are applied to subexpressions, the types of the latter should agree with the operand types of the former. These are name binding or typing constraints that are not captured by BIPL's signature.

Functional programs (BFPL) We may require that the function name of a function application can actually be resolved to a function declaration with a suitable type. Again, when operators are applied to subexpressions, the types of the latter should agree with the operand types of the former. These are typing constraints that are not captured by BFPL's signature.

Finite state machines (FSML) We may require, for example, that, in a given state, for a given event, at most one transition is feasible. Also, each state id referenced as a target of a transition must be declared. These are well-formedness constraints that are not captured by FSML's signature. The consistent referencing of (target) states is modeled by FSML's metamodel.

Buddy relationships (BL) We may require that a buddy graph (i.e., a BL world) should not contain any person whose buddy is that person themselves. This is a well-formedness constraint that is not captured by BL's metamodel.

Typical formalisms used for syntax definition – such as those leveraged in this chapter, but also context-free grammars, as leveraged in the context of concrete syntax definition (Chapter 6) – do not permit the capture of all constraints that one may want to assume for software languages. It is not uncommon to differentiate between *context-free* and *context-sensitive syntax*. The former refers to the more structural part of syntax, as definable by signatures, metamodels, or context-free grammars (Chapter 6). The latter assumes a definition that includes constraints requiring "context sensitivity" in terms of, for example, using names. One can model such typing or well-formedness constraints either by a metaprogram-based analysis (Chapter 5, specifically Section 5.3) or by means of dedicated formalisms, for example, type systems (Chapter 9) and attribute grammars (Chapter 12).

3.4 The Metametalevel

The notations for syntax definition (BSL, ESL, and MML) correspond to proper software languages in themselves. In this section, tree- or graph-based syntaxes of these syntax definition languages are defined. Accordingly, we operate at the *metametalevel*. In this manner, we advise on a representation of syntax definitions and thereby prepare for metaprograms that operate on the representation. For instance, we will eventually be able to approach conformance checking as a simple metaprogramming problem (Section 4.2).

3.4.1 The Signature of Signatures

To facilitate understanding, we look at the basic signature notation first. Its abstract syntax is modeled in the extended signature notation, as specified below.

Specification 3.1 (The ESL signature of BSL signatures)

ESL resource languages/BSL/as.esl

```
type signature = profile* ;
type profile = sym × sort* × sort ;
type sym = string ;
type sort = string ;
```

Thus, any signature can be represented as a term, as illustrated below.

Illustration 3.18 (BNL's signature in abstract syntax)

Term resource languages/BNL/as.term

```
[ (number, [bits, rest], number),
  (single, [bit], bits),
  (many, [bit, bits], bits),
  (zero, [], bit),
  (one, [], bit),
  (integer, [], rest),
  (rational, [bits], rest) ].
```

BSL cannot be described in itself, i.e., there is no BSL signature of BSL signatures because BSL lacks ESL's strings needed for the representation of function symbols and sorts. However, ESL can be described in itself as shown below.

Specification 3.2 (The ESL signature of ESL signatures)

ESL resource languages/ESL/as.esl

```
type signature = decl* ;
symbol type : sort × typeexpr → decl ;
symbol symbol : fsym × typeexpr* × sort → decl ;
symbol boolean : → typeexpr ;
symbol integer : → typeexpr ;
symbol float : → typeexpr ;
symbol string : → typeexpr ;
symbol term : → typeexpr ;
symbol sort : sort → typeexpr ;
symbol star : typeexpr → typeexpr ;
symbol plus : typeexpr → typeexpr ;
symbol option : typeexpr → typeexpr ;
symbol tuple : typeexpr* → typeexpr ;
```

```
type sort = string ;
type fsym = string ;
```

The signature does not capture several constraints that we may want to assume for a signature to be well-formed:

- The function symbols of the declared function types are distinct.
- The names of declared types are distinct.
- There is no name that is declared both as a type and as a sort (i.e., as a result type of a function symbol).

One may also expect a constraint that all referenced type and sort names are actually declared. More strongly, one could require that all sorts are "reachable" from a designated top-level sort and that all sorts are "productive" in that there exist terms of each sort. We do not commit to these extra constraints, however, because we may want to deal with incomplete signatures or modules, which could, for example, reference names (types or sorts) that are not declared in the same file.

3.4.2 The Signature of Metamodels

We can devise a signature for the tree-based abstract syntax of metamodels.

Specification 3.3 (The ESL signature of MML metamodels)

ESL resource languages/MML/as.esl

```
type metamodel = classifier⋆ ;
symbol class : abstract × cname × extends? × member⋆ → classifier ;
symbol datatype : cname → classifier ;
type member = kind × mname × cname × cardinality ;
symbol value : → kind ;
symbol part : → kind ;
symbol reference : → kind ;
symbol one : → cardinality ;
symbol option : → cardinality ;
symbol star : → cardinality ;
symbol plus : → cardinality ;
type abstract = boolean ;
type extends = cname ;
type cname = string ;
type mname = string ;
```

3.4.3 The Metamodel of Metamodels

We can devise a metamodel for the graph-based abstract syntax of metamodels.

Specification 3.4 (The MML metamodel of MML metamodels)

MML resource languages/MML/mm.mml

```
abstract class base { value name : string; }
class metamodel { part classifiers : classifier*; }
abstract class classifier extends base { }
class datatype extends classifier { }
class class extends classifier {
  value abstract : boolean;
  reference super : class?;
  part members : member*;
}
abstract class member extends base { part cardinality : cardinality; }
class value extends member { reference type : datatype; }
class part extends member { reference type : class; }
class reference extends member { reference type : class; }
abstract class cardinality { }
class one extends cardinality { }
class option extends cardinality { }
class star extends cardinality { }
class plus extends cardinality { }
datatype string;
datatype boolean;
```

The metamodel of metamodels can also be represented as a model – in fact, as an instance of itself. This underlines its status as a metametamodel.

Specification 3.5 (Excerpt of the metametamodel)

Graph resource languages/MML/mm.graph

```
{ class:metamodel,
  classifiers:[
    ( base & { class:class, name:base,
        abstract:true,
        super:[],
        members:[{class:value, name:name, type: #string, cardinality:{class:one}}]}),
    ( metamodel & { class:class, name:metamodel, ...
    ( classifier & { class:class, name:classifier, ...
    ( class & { class:class, name:class, ...
    ( member & { class:class, name:member, ...
    ...
  ] }.
```

Just as in the case of the language of signatures, we may also impose constraints on the language of metamodels, such as that the names of declared classes should be distinct; we omit a discussion of these routine details.

Summary and Outline

Some of the content of this chapter can be summarized in a recipe as follows.

Recipe 3.1 (Authoring an abstract syntax definition).

Syntactic categories *Identify the syntactic categories of the language such as state declarations and transitions in the case of FSML. Assign names to these categories. This identification process may also have been completed as part of a domain analysis (Section 1.3). The assigned names are also referred to as sorts in an algebraic signature or as classes in a metamodel.*

Trees versus graphs *Make a choice as to whether tree- or graph-based abstract syntax should be defined. You may prefer trees if the metaprogramming approach at hand favors trees or if references within the software language artifacts are easily resolved on top of trees.*

Alternatives *Identify the alternatives for each category. (Again, a domain analysis may readily provide such details.) Assign names to these alternatives; these names may be used as function symbols in a signature or as class names in a metamodel.*

Structure *Describe the structure of each alternative in terms of, for example, part-whole relationships or reference relationships, while making appropriate use of cardinalities (optionality, repetition), as supported by the syntax definition formalism at hand.*

Validation *Author samples so that the abstract syntax is exercised. Ultimately, you will want to check that the samples conform to the authored signature or metamodel, as discussed in a later recipe (Recipe 4.1).*

We have explained how trees and graphs may be used for abstract syntactical representation. We have also explained how signatures and metamodels may be used for modeling abstract syntax. The syntax definition formalisms described are inspired by notations used in practice, although some conveniences and expressiveness may be missing. In metamodeling, in particular, one may use a richer formalism with coverage of constraints (e.g., OCL constraints [2]).

In the next chapter, we will discuss the *implementation* of abstract syntax. Afterwards, we will engage in metaprogramming on top of abstract syntax. We are also prepared for metaprograms that process syntax definitions, since we have described the signature of signatures and other such metametalevel definitions in this chapter. A few chapters down the road, we will complement abstract syntax with concrete syntax for the purpose of defining, parsing, and formatting string languages.

References

1. Bézivin, J., Bouzitouna, S., Fabro, M.D.D., Gervais, M., Jouault, F., Kolovos, D.S., Kurtev, I., Paige, R.F.: A canonical scheme for model composition. In: Proc. ECMDA-FA, *LNCS*, vol. 4066, pp. 346–360. Springer (2006)
2. Clark, T., Warmer, J. (eds.): Object Modeling with the OCL, The Rationale behind the Object Constraint Language, *LNCS*, vol. 2263. Springer (2002)
3. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata (2007)
4. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The Syntax Definition Formalism SDF. reference manual. SIGPLAN Not. **24**(11), 43–75 (1989)
5. Herrera, A.S., Willink, E.D., Paige, R.F.: An OCL-based bridge from concrete to abstract syntax. In: Proc. International Workshop on OCL and Textual Modeling, *CEUR Workshop Proceedings*, vol. 1512, pp. 19–34. CEUR-WS.org (2015)
6. Kolovos, D.S., Paige, R.F., Polack, F.: Merging models with the Epsilon Merging Language (EML). In: Proc. MoDELS, *LNCS*, vol. 4199, pp. 215–229. Springer (2006)
7. Kolovos, D.S., Rose, L.M., bin Abid, S., Paige, R.F., Polack, F.A.C., Botterweck, G.: Taming EMF and GMF using model transformation. In: Proc. MODELS, *LNCS*, vol. 6394, pp. 211–225. Springer (2010)
8. Paige, R.F., Kolovos, D.S., Polack, F.A.C.: A tutorial on metamodelling for grammar researchers. Sci. Comput. Program. **96**, 396–416 (2014)
9. Paige, R.F., Matragkas, N.D., Rose, L.M.: Evolving models in model-driven engineering: State-of-the-art and future challenges. J. Syst. Softw. **111**, 272–280 (2016)
10. Rose, L.M., Matragkas, N.D., Kolovos, D.S., Paige, R.F.: A feature model for model-to-text transformation languages. In: Proc. MiSE, pp. 57–63. IEEE (2012)
11. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development (Monographs in Theoretical Computer Science. An EATCS Series). Springer (2011)