# 2
# A simple programming language and its implementation

*J.A. Bergstra*
*J. Heering*
*P. Klint*

In this chapter we give a complete definition of the toy language PICO by specifying the parsing, typechecking and execution of PICO programs.

## 2.1 Introduction

Is it possible to give a complete algebraic specification of a simple programming language by specifying all the processes needed to run programs in that language? This chapter attempts to answer this question by developing a specification of the toy language PICO. This specification describes in detail all necessary steps from entering a PICO program in its textual form to computing its value. Since we are not particularly interested in the language PICO itself, an attempt will be made to make the specification as independent as possible from specific details related to PICO. This is achieved by specifying general techniques for defining and implementing programming languages such as, e.g., the generation of scanners and parsers, and to specialize these for the case of PICO. In general, one should distinguish the following two issues:

- How is a language defined?
- How can we obtain an implementation for it from this definition?

The ultimate goal is to obtain a simple and intuitively appealing language definition that can be transformed automatically into an implementation. This goal will not be reached in this chapter. Instead, we will define a language by giving an algebraic specification of its implementation.

Given a program in the form of a text string, how can we determine that this string is an acceptable program and how do we run this program? We will consider the following steps during the processing of programs:

1. **Lexical scanning**: remove layout from the input string and try to decompose it into a sequence of tokens describing keywords, identifiers, and constants.
2. **Parsing**: determine whether or not this sequence of lexical tokens forms a syntactically legal program.
3. **Construction of abstract syntax trees**: construct a tree representation of the program that captures its essential structure but suppresses syntactic details such as derivation chains and the precise concrete representation of each language construct.
4. **Typechecking**: check that the abstract syntax tree conforms to certain language-specific constraints, e.g., all variables that are used in the program should also occur in a declaration, and the use of variables should be consistent with their declared type (static semantics).
5. **Evaluation**: execute the program by interpreting its abstract syntax tree (dynamic semantics).

It turns out that, in the case of PICO, the specification of steps (1) , (2) and (3) is more complex than that of steps (4) and (5). Therefore, we present the specification of PICO in two stages. First, a system is specified which only defines steps (4) and (5): this covers typechecking and evaluation of programs in the form of abstract syntax trees. Next, a second system is specified which extends the first one with steps (1), (2) and (3): this covers scanning, parsing, typechecking and evaluation of programs in their concrete form.

The organization of this chapter is centered around the presentation of these two PICO systems. In Section 2.2 an informal definition of the language PICO is presented. The elementary data types needed later on are specified in Section 2.3. The first PICO system (based on a tree representation of programs) is presented in Section 2.4. The next problem to be addressed is how scanning and parsing can be specified algebraically. In

Section 2.5 the notion of a completely parameterized context-free parser is introduced. This parameterized parser is used in Section 2.6, where the second PICO system (based on a concrete representation of programs) is presented. Section 2.7 concludes this chapter with an assessment of these two specifications.

## 2.2 Informal definition of the language PICO

The language PICO is extremely simple. It is essentially the language of while-programs. A program consists of declarations followed by statements. All variables occurring in the statements have to be declared to be either of type natural number or of type string. Statements may be assignment statements, if-statements and while-statements. Expressions may be a single identifier, addition or subtraction of natural numbers, or concatenation of strings.

At the lexical level, PICO programs consist of a sequence of lexical items separated by layout. Lexical items are keywords, identifiers, constants for natural numbers and strings, and punctuation marks. The lexical grammar for PICO is:

```
<lexical-stream>  ::= <lexical-item> <lexical-stream> |
                      <lexical-item>
<lexical-item>    ::= <optional-layout>
                      (<keyword-or-id> |
                       <natural-constant> |
                       <string-constant> |
                       <literal>)
<optional-layout> ::= <layout> | <empty>
<keyword-or-id>   ::= 'begin' | 'end' | 'declare' |
                      'natural' | 'string' | 'if' |
                      'then' | 'else' | 'fi' |
                      'while' | 'do' | 'od' | <id>
<id>              ::= <letter> <id-chars>
<id-chars>        ::= <id-char> <id-chars> | <empty>
<id-char>         ::= <letter> | <digit>
<natural-constant>::= <digit> <digits>
<digits>          ::= <digit> <digits> | <empty>
<string-constant> ::= <quote> <string-tail>
<string-tail>     ::= <any-char-but-quote>
                        <string-tail> | <quote>
<quote>           ::= '"'
<any-char-but-quote>
                  ::= <letter> | <digit> |
```

```
                    <literal> | <layout>
<literal>       ::= '(' | ')' | '+' | '-' | ';' |
                    ',' | '||' | ':' | ':='
<letter>        ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
<digit>         ::= '0' | ... | '9'
<layout>        ::= ' ' | <newline> | <tab>
```

Here, <newline> and <tab> are assumed to be primitive notions corresponding to, respectively, the newline character and the tabulation. The concrete syntax of PICO is:

```
<pico-program> ::= 'begin' <decls> <series> 'end'
<decls>        ::= 'declare' <id-type-list> ';'
<id-type-list> ::= <id> ':' <type>
                      (<empty> | ',' <id-type-list>)
<type>         ::= 'natural' | 'string'
<series>       ::= <empty> |
                   <stat> (<empty> | ';' <series>)
<stat>         ::=  <assign> | <if> | <while>
<assign>       ::= <id> ':=' <exp>
<if>           ::= 'if' <exp> 'then' <series>
                                 'else' <series> 'fi'
<while>        ::= 'while' <exp> 'do' <series> 'od'
<exp>          ::=  <primary> |
                    <plus> | <minus> | <conc>
<primary>      ::= <id> | <natural-constant> |
                   <string-constant> | '(' <exp> ')'
<plus>         ::= <exp> '+' <primary>
<minus>        ::= <exp> '-' <primary>
<conc>         ::= <exp> '||' <primary>
<empty>        ::= ''
```

The non-terminals <id>, <natural-constant> and <string-constant> are defined in the lexical grammar given above and represent identifiers, natural number constants and string constants respectively.
There are two overall static semantic constraints on programs:

- All identifiers occurring in a program should have been declared and their use should be compatible with their declaration. More precisely, this means that all <id>s occurring in an <assign> or a <primary> should have been declared, i.e., should occur in some <id-type> in the <id-type-list> of the <decls>-part of the PICO-program, and that the type of <id>s should be compatible with the expressions in which they occur.

- The <exp> occurring in an <if>- or <while>-statement should be of type natural number.

A type can be given to an <exp> or <primary> as follows:

- if a <primary> consists of an <id>, that <id> should have been declared and the type of the <primary> is the same as the type of the <id> in its declaration;
- a <primary> consisting of a <natural-constant> has type natural number;
- a <primary> consisting of a <string-constant> has type string;
- an <exp> consisting of a single <primary> has the same type as that <primary>;
- an <exp> consisting of a <plus> or <minus> has type natural number;
- an <exp> consisting of a <conc> has type string.

Given this notion of types of <exp> and <primary>, the static semantic constraints can be formulated in more detail:

- the arguments of the operators + and - should be of type natural number;
- the arguments of the operator | | should be of type string;
- the <id> and <exp> that occur in an <assign> should have the same type;
- the <exp>s that occur in <if> and <while> should have type natural number.

The rules for the evaluation of PICO programs (dynamic semantics) are straightforward except that

- natural number variables are initialized with value 0;
- string variables are initialized with "" (empty string);
- the <exp> in an <if> or <while> is assumed to be true if its value is not equal to 0.

## 2.3 Elementary data types

As a prerequisite for the specification of the two PICO systems in following sections of this chapter, we first give some specifications for elementary data types:

**Booleans** (2.3.1): truth values `true` and `false` with functions `and`, `or`, and `not`.

**Naturals** (2.3.2): natural numbers with constants `0`, `1` and `10` and functions `succ` (successor), `add` (addition), `sub` (subtraction), `mul`

(multiplication), `exp` (exponentiation), `eq` (equality of natural numbers), `less` (less than), `lesseq` (less than or equal), `greater` (greater than) and `greatereq` (greater than or equal).

**Characters** (2.3.3): the alphabet consists of constants for letters, digits, and punctuation marks. The functions `eq` (equality of characters), `ord` (ordinal number of character in the alphabet), `is-letter` (is character a letter?), `is-upper` (is character an upper case letter?), `is-lower` (is character a lower case letter?) and `is-digit` (is character a digit?) are defined on them.

**Sequences** (2.3.4): linear lists of items. Sequences are parameterized with the data type of the items. The only constant is `null`, the empty sequence. The functions `eq` (equality of sequences), `seq` (combine item with sequence), `conc` (concatenate two sequences) and `conv-to-seq` (convert an item to a sequence) are defined for sequences.

**Strings** (2.3.5): sequences of characters, i.e., sequences with items bound by characters. The only constant is `null-string`, the empty string. The following functions are defined: `eq` (equality of strings), `seq` (combine character with a string), `conc` (concatenate two strings), `string` (convert a character to a string) and `str-to-nat` (convert a string to a natural number).

**Tables** (2.3.6): mapping from strings to entries, where entries are a parameter. The only constant is `null-table`, the empty table. The functions `table` (add new entry to table), `lookup` (searches for an entry in a table), and `delete` (deletes an entry from a table) are defined for tables.

**Types** (2.3.7): the types that may occur during typechecking of PICO programs are natural number and string. These two possibilities are represented by the constants `natural-type` and `string-type`. The constant `error-type` represents erroneous types. The only function on types is `eq` (equality of types).

**Values** (2.3.8): the values that may occur during the execution of PICO programs are natural number and string. These are represented by the constant `error-value` (values that are the result of an erroneous computation) and the functions `pico-value` (injection from natural numbers and strings to PICO values) and `eq` (equality of PICO values).

### 2.3.1 Booleans

Booleans are truth values `true` and `false` with functions `and`, `or` and `not`.

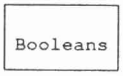


*Fig. 2.1:*
*Structure diagram for Booleans*

```
module Booleans
begin
  exports
    begin
      sorts BOOL
      functions
        true :                -> BOOL
        false:                -> BOOL
        or    : BOOL # BOOL   -> BOOL
        and   : BOOL # BOOL   -> BOOL
        not   : BOOL          -> BOOL
    end

  equations

  [1]  or(true, true)     = true
  [2]  or(true, false)    = true
  [3]  or(false, true)    = true
  [4]  or(false, false)   = false
  [5]  and(true, true)    = true
  [6]  and(true, false)   = false
  [7]  and(false, true)   = false
  [8]  and(false, false)  = false
  [9]  not(true)          = false
  [10] not(false)         = true

end Booleans
```

### 2.3.2 Naturals

Natural numbers are defined here with constants `0`, `1` and `10` and functions `succ` (successor), `add` (addition), `sub` (subtraction), `mul` (multiplication), `exp` (exponentiation), `eq` (equality), `less` (less than), `lesseq` (less than or equal), `greater` (greater than) and `greatereq` (greater than or equal).

The equations for the constants `1` and `10` are not very satisfactory. Clearly, a mechanism is needed for defining a shorthand notation for *all* constants. In Section 2.3.5 this subject is discussed in connection with string constants.

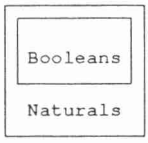


*Fig. 2.2:*
*Structure diagram for natural numbers*

```
module Naturals
begin
  exports
    begin
      sorts NAT
      functions
        0          :                -> NAT
        1          :                -> NAT
        10         :                -> NAT
        succ       : NAT            -> NAT
        add        : NAT # NAT      -> NAT
        sub        : NAT # NAT      -> NAT
        mul        : NAT # NAT      -> NAT
        exp        : NAT # NAT      -> NAT
        eq         : NAT # NAT      -> BOOL
        less       : NAT # NAT      -> BOOL
        lesseq     : NAT # NAT      -> BOOL
        greater    : NAT # NAT      -> BOOL
        greatereq: NAT # NAT        -> BOOL
    end

  imports Booleans

  variables
    x, y :-> NAT

  equations

  [11] 1                            = succ(0)
  [12] 10                           = succ(succ(succ(succ(
                                        succ(succ(succ(succ(
                                          succ(succ(0)))))))))
  [13] add(x, 0)                    = x
  [14] add(x, succ(y))              = succ(add(x, y))

  [15] sub(0, x)                    = 0
```

```
    [16] sub(x, 0)                  = x
    [17] sub(succ(x), succ(y))      = sub(x, y)

    [18] mul(x, 0)                  = 0
    [19] mul(x, succ(y))            = add(x, mul(x, y))

    [20] exp(x, 0)                  = 1
    [21] exp(x, succ(y))            = mul(x, exp(x, y))

    [22] eq(0, 0)                   = true
    [23] eq(succ(x),0)              = false
    [24] eq(succ(x), succ(y))       = eq(x, y)
    [25] eq(0, succ(x))             = false

    [26] less(x, 0)                 = false
    [27] less(0, succ(x))           = true
    [28] less(succ(x), succ(y))     = less(x, y)

    [29] lesseq(x, y)               = or(less(x,y),eq(x,y))

    [30] greater(x, y)              = not(lesseq(x, y))

    [31] greatereq(x, y)            = or(greater(x, y),
                                         eq(x, y))
end Naturals
```

### 2.3.3 Characters

The alphabet of characters consists of constants for letters, digits, and punctuation marks. The functions `eq` (equality), `ord` (ordinal number of character in the alphabet), `is-letter` (is character a letter?), `is-upper` (is character an upper case letter?), `is-lower` (is character a lower case letter?) and `is-digit` (is character a digit?) are defined on them.

Three observations can be made about this specification. First, one may notice that the absence of constants for the natural numbers forces us to write equations of the form

```
ord(char-3)      = succ(ord(char-2))
```

instead of the more natural form

```
ord(char-3)      = 3
```

Secondly, it is clear that some abbreviation mechanism is needed for specifications that contain many constants as is the case here. At the expense of additional complexity of the specification, this could have been

achieved by defining characters in two stages: first, a basic alphabet is defined which consists only of lower case letters and a hyphen; next, this basic alphabet is used to generate all constants for the full alphabet. Names of constants are then only allowed to contain symbols from the basic alphabet, i.e., `char-upper-case-a` instead of `char-A`. Thirdly, one may dispute the utility of including low level modules such as this specification of the characters. In larger specifications it is common practice to specify only the high-level modules completely, and to limit the specification of the low-level modules to their signature.

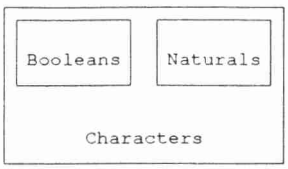


*Fig. 2.3:*
*Structure diagram for characters*

```
module Characters
begin
  exports
    begin
      sorts     CHAR
      functions
        eq         : CHAR # CHAR   -> BOOL
        is-upper   : CHAR          -> BOOL
        is-lower   : CHAR          -> BOOL
        is-letter  : CHAR          -> BOOL
        is-digit   : CHAR          -> BOOL
        ord        : CHAR          -> NAT

        char-0     :               -> CHAR
        char-1     :               -> CHAR
        char-2     :               -> CHAR
        char-3     :               -> CHAR
        char-4     :               -> CHAR
        char-5     :               -> CHAR
        char-6     :               -> CHAR
        char-7     :               -> CHAR
        char-8     :               -> CHAR
        char-9     :               -> CHAR

        char-ht    :               -> CHAR -- tab --
```

```
char-nl   :                -> CHAR -- nl --
char-space:                -> CHAR -- space --
char-quote:                -> CHAR -- " --
char-lpar :                -> CHAR -- ( --
char-rpar :                -> CHAR -- ) --
char-times:                -> CHAR -- * --
char-plus :                -> CHAR -- + --
char-comma:                -> CHAR -- , --
char-minus:                -> CHAR -- - --
char-point:                -> CHAR -- . --
char-slash:                -> CHAR -- / --
char-bar  :                -> CHAR -- | --
char-equal:                -> CHAR -- = --
char-colon:                -> CHAR -- : --
char-semi :                -> CHAR -- ; --

char-A    :                -> CHAR
char-B    :                -> CHAR
char-C    :                -> CHAR
char-D    :                -> CHAR
char-E    :                -> CHAR
char-F    :                -> CHAR
char-G    :                -> CHAR
char-H    :                -> CHAR
char-I    :                -> CHAR
char-J    :                -> CHAR
char-K    :                -> CHAR
char-L    :                -> CHAR
char-M    :                -> CHAR
char-N    :                -> CHAR
char-O    :                -> CHAR
char-P    :                -> CHAR
char-Q    :                -> CHAR
char-R    :                -> CHAR
char-S    :                -> CHAR
char-T    :                -> CHAR
char-U    :                -> CHAR
char-V    :                -> CHAR
char-W    :                -> CHAR
char-X    :                -> CHAR
char-Y    :                -> CHAR
char-Z    :                -> CHAR

char-a    :                -> CHAR
char-b    :                -> CHAR
```

```
      char-c    :                -> CHAR
      char-d    :                -> CHAR
      char-e    :                -> CHAR
      char-f    :                -> CHAR
      char-g    :                -> CHAR
      char-h    :                -> CHAR
      char-i    :                -> CHAR
      char-j    :                -> CHAR
      char-k    :                -> CHAR
      char-l    :                -> CHAR
      char-m    :                -> CHAR
      char-n    :                -> CHAR
      char-o    :                -> CHAR
      char-p    :                -> CHAR
      char-q    :                -> CHAR
      char-r    :                -> CHAR
      char-s    :                -> CHAR
      char-t    :                -> CHAR
      char-u    :                -> CHAR
      char-v    :                -> CHAR
      char-w    :                -> CHAR
      char-x    :                -> CHAR
      char-y    :                -> CHAR
      char-z    :                -> CHAR

  end

imports Booleans, Naturals

variables
  c, c1, c2 :-> CHAR

equations

 [32] ord(char-0)      = 0
 [33] ord(char-1)      = succ(ord(char-0))
 [34] ord(char-2)      = succ(ord(char-1))
 [35] ord(char-3)      = succ(ord(char-2))
 [36] ord(char-4)      = succ(ord(char-3))
 [37] ord(char-5)      = succ(ord(char-4))
 [38] ord(char-6)      = succ(ord(char-5))
 [39] ord(char-7)      = succ(ord(char-6))
 [40] ord(char-8)      = succ(ord(char-7))
 [41] ord(char-9)      = succ(ord(char-8))
```

```
[42] ord(char-ht)     = succ(ord(char-9))
[43] ord(char-nl)     = succ(ord(char-ht))
[44] ord(char-space)  = succ(ord(char-nl))
[45] ord(char-quote)  = succ(ord(char-space))
[46] ord(char-lpar)   = succ(ord(char-quote))
[47] ord(char-rpar)   = succ(ord(char-lpar))
[48] ord(char-times)  = succ(ord(char-rpar))
[49] ord(char-plus)   = succ(ord(char-times))
[50] ord(char-comma)  = succ(ord(char-plus))
[51] ord(char-minus)  = succ(ord(char-comma))
[52] ord(char-point)  = succ(ord(char-minus))
[53] ord(char-slash)  = succ(ord(char-point))
[54] ord(char-bar)    = succ(ord(char-slash))
[55] ord(char-equal)  = succ(ord(char-bar))
[56] ord(char-colon)  = succ(ord(char-equal))
[57] ord(char-semi)   = succ(ord(char-colon))


[58] ord(char-A)      = succ(ord(char-semi))
[59] ord(char-B)      = succ(ord(char-A))
[60] ord(char-C)      = succ(ord(char-B))
[61] ord(char-D)      = succ(ord(char-C))
[62] ord(char-E)      = succ(ord(char-D))
[63] ord(char-F)      = succ(ord(char-E))
[64] ord(char-G)      = succ(ord(char-F))
[65] ord(char-H)      = succ(ord(char-G))
[66] ord(char-I)      = succ(ord(char-H))
[67] ord(char-J)      = succ(ord(char-I))
[68] ord(char-K)      = succ(ord(char-J))
[69] ord(char-L)      = succ(ord(char-K))
[70] ord(char-M)      = succ(ord(char-L))
[71] ord(char-N)      = succ(ord(char-M))
[72] ord(char-O)      = succ(ord(char-N))
[73] ord(char-P)      = succ(ord(char-O))
[74] ord(char-Q)      = succ(ord(char-P))
[75] ord(char-R)      = succ(ord(char-Q))
[76] ord(char-S)      = succ(ord(char-R))
[77] ord(char-T)      = succ(ord(char-S))
[78] ord(char-U)      = succ(ord(char-T))
[79] ord(char-V)      = succ(ord(char-U))
[80] ord(char-W)      = succ(ord(char-V))
[81] ord(char-X)      = succ(ord(char-W))
[82] ord(char-Y)      = succ(ord(char-X))
[83] ord(char-Z)      = succ(ord(char-Y))
```

```
    [84] ord(char-a)      = succ(ord(char-Z))
    [85] ord(char-b)      = succ(ord(char-a))
    [86] ord(char-c)      = succ(ord(char-b))
    [87] ord(char-d)      = succ(ord(char-c))
    [88] ord(char-e)      = succ(ord(char-d))
    [89] ord(char-f)      = succ(ord(char-e))
    [90] ord(char-g)      = succ(ord(char-f))
    [91] ord(char-h)      = succ(ord(char-g))
    [92] ord(char-i)      = succ(ord(char-h))
    [93] ord(char-j)      = succ(ord(char-i))
    [94] ord(char-k)      = succ(ord(char-j))
    [95] ord(char-l)      = succ(ord(char-k))
    [96] ord(char-m)      = succ(ord(char-l))
    [97] ord(char-n)      = succ(ord(char-m))
    [98] ord(char-o)      = succ(ord(char-n))
    [99] ord(char-p)      = succ(ord(char-o))
   [100] ord(char-q)      = succ(ord(char-p))
   [101] ord(char-r)      = succ(ord(char-q))
   [102] ord(char-s)      = succ(ord(char-r))
   [103] ord(char-t)      = succ(ord(char-s))
   [104] ord(char-u)      = succ(ord(char-t))
   [105] ord(char-v)      = succ(ord(char-u))
   [106] ord(char-w)      = succ(ord(char-v))
   [107] ord(char-x)      = succ(ord(char-w))
   [108] ord(char-y)      = succ(ord(char-x))
   [109] ord(char-z)      = succ(ord(char-y))

   [110] eq(c1, c2)    = eq(ord(c1), ord(c2))

   [111] is-upper(c)   = and(
                         greatereq(ord(c),ord(char-A)),
                         lesseq(ord(c),ord(char-Z)))

   [112] is-lower(c)   = and(
                         greatereq(ord(c),ord(char-a)),
                         lesseq(ord(c),ord(char-z)))

   [113] is-digit(c)   = and(
                         greatereq(ord(c),ord(char-0)),
                         lesseq(ord(c),ord(char-9)))

   [114] is-letter(c) = or(is-upper(c), is-lower(c))

end Characters
```

### 2.3.4 Sequences

Sequences are linear lists of items; they are parameterized with the data type of the items. The only constant is `null`, the empty sequence. The following functions are defined: `eq` (equality), `seq` (combine item with sequence), `conc` (concatenate two sequences) and `conv-to-seq` (convert an item to a sequence).

Note that the function `eq` in this module is overloaded.

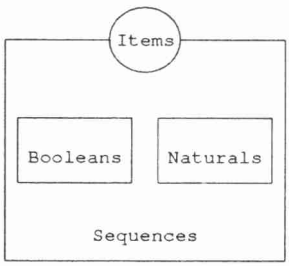


*Fig. 2.4:*
*Structure diagram for sequences*

```
module Sequences
begin

  parameters Items
    begin
      sorts  ITEM
      functions
        eq: ITEM # ITEM -> BOOL
    end Items

  exports
    begin
      sorts     SEQ
      functions
        null        :              -> SEQ
        seq         : ITEM # SEQ   -> SEQ
        conc        : SEQ # SEQ    -> SEQ
        length      : SEQ          -> NAT
        eq          : SEQ # SEQ    -> BOOL
        conv-to-seq: ITEM          -> SEQ
    end

  imports Booleans, Naturals
```

```
    variables
      s, s1, s2 :-> SEQ
      it, it1, it2 :-> ITEM

    equations

    [115] conc(s, null)          = s
    [116] conc(null, s)          = s
    [117] conc(seq(it, s1), s2)  = seq(it, conc(s1, s2))

    [118] eq(s, s)               = true
    [119] eq(s1, s2)             = eq(s2, s1)
    [120] eq(null, seq(it, s))   = false
    [121] eq(seq(it1,s1), seq(it2,s2))
                                 = and(eq(it1,it2),
                                       eq(s1,s2))

    [122] length(null)           = 0
    [123] length(seq(it,s))      = succ(length(s))

    [124] conv-to-seq(it)        = seq(it, null)

  end Sequences
```

### 2.3.5 Strings

Strings are sequences of characters, i.e., `Sequences` with `Items` bound to `Characters`. The only constant is `null-string`, the empty string. The functions `eq` (equality), `seq` (combine character with string), `conc` (concatenate two strings), `string` (convert character to string) and `str-to-nat` (convert string to natural number) are defined for strings.

In the case of the data type string there is an urgent need for a short hand notation for string constants. The PICO specification would become unreadable without it. We will therefore use an, *ad hoc*, convenient notation for string constants to denote the terms generated by the module `Strings`, e.g., the term

```
seq(char-a, seq(char-b, null-string))
```

will be written as

```
"ab"
```

The empty string, i.e., the constant `null-string`, will be written as `""`. A *general* abbreviation scheme is indispensable for obtaining readable specifications containing constants for natural numbers and strings, sets, lists, etc. We have not attempted to solve this problem in ASF. Rather, a

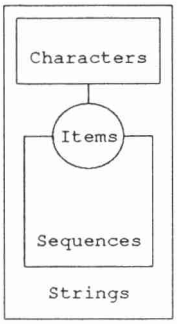


*Fig. 2.5:*
*Structure diagram for strings*

separate, specialized, formalism was designed (SDF: Syntax Definition Formalism, see Chapter 6) which provides means to introduce general user-defined syntactic notations in (algebraic) specifications.

```
module Strings
begin
  exports
    begin
      functions
        str-to-nat : STRING -> NAT
    end

  imports Sequences
                { renamed by
                    [ SEQ -> STRING,
                      null -> null-string,
                      conv-to-seq -> string]
                  Items bound by
                    [ ITEM -> CHAR,
                      eq -> eq]
                  to Characters }

  variables
    c :-> CHAR
    str :-> STRING

  equations

  [125] str-to-nat(seq(c, str))
          = add(mul(ord(c), exp(10,length(str))),
```

```
                    str-to-nat(str))
  [126] str-to-nat(null-string) = 0

end Strings
```

### 2.3.6 Tables

Tables are mappings from strings to entries, where entries are a parameter. The only constant is `null-table`, the empty table. The following functions are defined on tables: `table` (add new entry to table), `lookup` (searches for an entry in a table) and `delete` (deletes an entry from a table).

Note that adding a pair (`name`, `error-entry`) to a table has the somewhat strange, but harmless, effect that

```
lookup(name, table(name, error-entry, tbl1))
   = <true, error-entry>
```

and that

```
lookup(name, null-table)
   = <false, error-entry>
```

Only in the first case `name` occurs in the table, but except for the `true`/`false` flag, the same value is delivered.

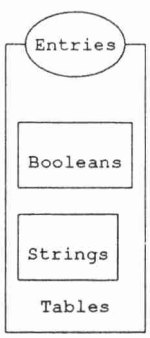


*Fig. 2.6:*
*Structure diagram for tables*

```
module Tables
begin

  parameters Entries
    begin
      sorts     ENTRY
      functions
        error-entry:  -> ENTRY
    end Entries

  exports
    begin
      sorts TABLE
      functions
        null-table:                              -> TABLE
        table     : STRING # ENTRY # TABLE       -> TABLE
        lookup    : STRING # TABLE               -> BOOL #
                                                    ENTRY
        delete    : STRING # TABLE               -> TABLE
    end

  imports Booleans, Strings

  variables
    name, name1, name2 :-> STRING
    e :-> ENTRY
    tbl :-> TABLE

  equations

  [127] lookup(name, null-table)
          = <false, error-entry>

  [128] lookup(name1, table(name2, e, tbl))
          = if(eq(name1, name2),
               <true, e>,
               lookup(name1, tbl))
  [129] delete(name, null-table)
          = null-table
  [130] delete(name1, table(name2, e, tbl))
          = if(eq(name1, name2),
               delete(name1, tbl),
               table(name2, e, delete(name1, tbl)))
end Tables
```

### 2.3.7 Types

The data type `PICO-types` defines the allowed types in PICO programs, i.e., natural numbers and strings. An additional `error-type` is introduced for describing typing errors.

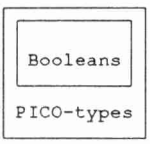


*Fig. 2.7:*
*Structure diagram for types*

```
module PICO-types
begin
  exports
    begin
      sorts PICO-TYPE
      functions
        natural-type:                          -> PICO-TYPE
        string-type :                          -> PICO-TYPE
        error-type  :                          -> PICO-TYPE
        eq          : PICO-TYPE #
                                  PICO-TYPE -> BOOL
    end

  imports Booleans

  variables
    x, y :-> PICO-TYPE

  equations

  [131] eq(x, x)                          = true
  [132] eq(x, y)                          = eq(y, x)
  [133] eq(natural-type, string-type)     = false
  [134] eq(natural-type,  error-type)     = false
  [135] eq(string-type,  error-type)      = false

end PICO-types
```

### 2.3.8 Values

The data type `PICO-values` defines the allowed values as they may occur during the execution of PICO programs, i.e., natural numbers and strings. An additional `error-value` is introduced for describing values that are the result of evaluating erroneous programs. Note that there is no natural number or string corresponding to `error-value`. Two conversion functions are defined for converting `Naturals` and `Strings` to `PICO-values`.

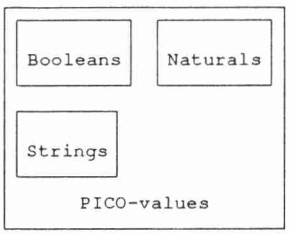


*Fig. 2.8:*
*Structure diagram for values*

```
module PICO-values
begin
  exports
    begin
      sorts PICO-VALUE
      functions
        error-value:                          -> PICO-VALUE
        pico-value : NAT                      -> PICO-VALUE
        pico-value : STRING                   -> PICO-VALUE
        eq         : PICO-VALUE #
                                 PICO-VALUE -> BOOL
    end

  imports Booleans, Naturals, Strings

  variables
    x, y :-> PICO-VALUE
    nat, nat1, nat2 :-> NAT
    str, str1, str2 :-> STRING

  equations

  [136] eq(x, x) = true
  [137] eq(x, y) = eq(y, x)
```

```
    [138] eq(pico-value(nat1), pico-value(nat2))
            = eq(nat1, nat2)
    [139] eq(pico-value(nat), pico-value(str))
            = false
    [140] eq(pico-value(nat), error-value)
            = false
    [141] eq(pico-value(str1), pico-value(str2))
            = eq(str1, str2)
    [142] eq(pico-value(str), error-value)
            = false

end PICO-values
```

## 2.4 First system: representing programs as trees

On top of the foundation constructed in the previous section we can now specify a first PICO system which deals with typechecking and evaluation of PICO programs. In this first system, it is assumed that PICO programs exist in the form of an abstract syntax tree. Here, we do not address the question of how abstract syntax trees are constructed for a given program in concrete form. This problem is the subject of Sections 2.5 and 2.6 of this chapter.

The constituents of the first PICO system are:

**Abstract syntax** (2.4.1): defines the abstract tree structure underlying the concrete (textual) representation of programs.

**Typechecking** (2.4.2): defines the process of checking the static semantic constraints on programs (in the form of abstract syntax trees).

**Evaluation** (2.4.3): defines the evaluation of programs (in the form of abstract syntax trees).

**PICO system 1** (2.4.4): combines the above components into a whole by defining a function `run-pico-abstree` which typechecks and evaluates a given abstract syntax tree.

### 2.4.1 Abstract syntax

The abstract syntax defines all legal syntax trees for PICO programs. This definition closely follows the definition of the concrete syntax of PICO as given in Section 2.2. The possible syntactic domains are defined by the sorts `PICO-PROGRAM` (complete PICO programs), `DECLS` (the declarations part of a PICO program), `SERIES` (a series of statements), `STATEMENT` (a

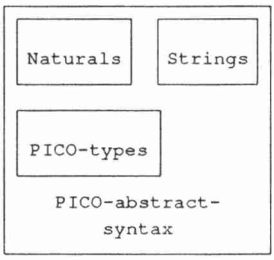


*Fig. 2.9:*
*Structure diagram for abstract syntax*

single statement), EXP (an expression) and ID (an identifier). The well-formed syntax trees are defined on these domains by means of a number of free constructor functions.

```
module PICO-abstract-syntax
begin
  exports
    begin
      sorts PICO-PROGRAM, DECLS, SERIES, STATEMENT,
            EXP, ID
      functions
        abs-pico-program: DECLS # SERIES    ->
                                            PICO-PROGRAM
        abs-decls : ID # PICO-TYPE # DECLS -> DECLS
        abs-empty-decls:                    -> DECLS
        abs-series: STATEMENT # SERIES      -> SERIES
        abs-empty-series:                   -> SERIES
        abs-assign: ID  # EXP               -> STATEMENT
        abs-if    : EXP # SERIES # SERIES   -> STATEMENT
        abs-while : EXP # SERIES            -> STATEMENT
        abs-plus  : EXP # EXP               -> EXP
        abs-minus : EXP # EXP               -> EXP
        abs-conc  : EXP # EXP               -> EXP
        abs-var   : ID                      -> EXP
        abs-natural-constant: NAT           -> EXP
        abs-string-constant: STRING         -> EXP
        abs-id    : STRING                  -> ID
    end

  imports Naturals, Strings, PICO-types

end PICO-abstract-syntax
```

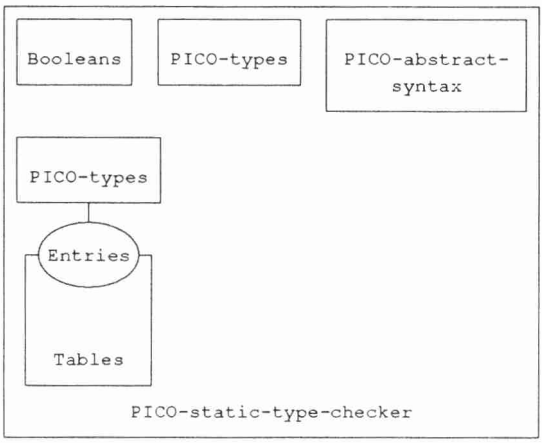


*Fig. 2.10: Structure diagram for typechecker*

### 2.4.2 Typechecking

Now we specify the checking of static semantic constraints on PICO programs as defined informally in Section 2.2. The principal function is `check` which operates on an abstract PICO program and checks whether this program is in accordance with the static semantic constraints. For each construct in the abstract syntax tree these constraints are expressed as transformations on a type environment. Type environments are defined as a combination of `Tables` and `PICO-types`. Checking the declaration section of a PICO program amounts to constructing a type environment, and checking the statement section amounts to checking each statement for conformity with a given type environment.

```
module PICO-static-type-checker
begin
  exports
    begin
      functions
        check: PICO-PROGRAM          -> BOOL
        check: DECLS # TYPE-ENV      -> BOOL # TYPE-ENV
        check: SERIES # TYPE-ENV     -> BOOL
        check: STATEMENT # TYPE-ENV -> BOOL
    end

  imports Booleans, PICO-types, PICO-abstract-syntax,
```

```
          Tables
               { renamed by
                   [ TABLE -> TYPE-ENV,
                     null-table -> null-type-env ]
                 Entries bound by
                   [ ENTRY -> PICO-TYPE,
                     error-entry -> error-type ]
                 to PICO-types }
functions
  type-of-exp: EXP # TYPE-ENV      -> PICO-TYPE

variables
  dec :-> DECLS
  ser, ser1, ser2 :-> SERIES
  stat :-> STATEMENT
  name :-> STRING
  nat :-> NAT
  typ :-> PICO-TYPE
  str :-> STRING
  x, x1, x2 :-> EXP
  env, env1, env2 :-> TYPE-ENV
  b1, b2, found :-> BOOL

equations

[143] check(abs-pico-program(dec, ser))
      = and(b1, b2)
        when <b1, env1> = check(dec, null-type-env),
             b2         = check(ser, env1)

[144] check(abs-decls(abs-id(name), typ, dec), env)
      = check(dec, table(name, typ, env))
[145] check(abs-empty-decls, env)
      = < true, env >

[146] check(abs-series(stat, ser), env)
      = and(check(stat, env), check(ser, env))

[147] check(abs-empty-series, env)
      = true

[148] check(abs-assign(abs-id(name), x), env)
      = and(found, eq(typ, type-of-exp(x, env)))
        when <found, typ> = lookup(name, env)
```

```
[149] check(abs-if(x, ser1, ser2), env)
      = and(eq(type-of-exp(x,env),natural-type),
            and(check(ser1, env), check(ser2, env)))

[150] check(abs-while(x, ser), env)
      = and(eq(type-of-exp(x, env), natural-type),
            check(ser, env))

[151] type-of-exp(abs-plus(x1, x2), env)
      = if(and(eq(type-of-exp(x1,env),natural-type),
               eq(type-of-exp(x2,env),natural-type)),
           natural-type,
           error-type)

[152] type-of-exp(abs-minus(x1, x2), env)
      = if(and(eq(type-of-exp(x1,env),natural-type),
               eq(type-of-exp(x2,env),natural-type)),
           natural-type,
           error-type)

[153] type-of-exp(abs-conc(x1, x2), env)
      = if(and(eq(type-of-exp(x1,env),string-type),
               eq(type-of-exp(x2,env),string-type)),
           string-type,
           error-type)

[154] type-of-exp(abs-natural-constant(nat), env)
      = natural-type
[155] type-of-exp(abs-string-constant(str), env)
      = string-type
[156] type-of-exp(abs-var(abs-id(name)), env)
      = if(found, typ, error-type)
        when <found, typ> = lookup(name, env)

end PICO-static-type-checker
```

### 2.4.3 Evaluation

In this section the evaluation of PICO programs is defined. The evaluation of programs is defined by defining the evaluation of each kind of construct that may appear in the abstract syntax tree. Evaluation is expressed as a transformation on value environments which describe the values of the variables in the program. Value environments are defined as combinations of `Tables` and `PICO-values`.

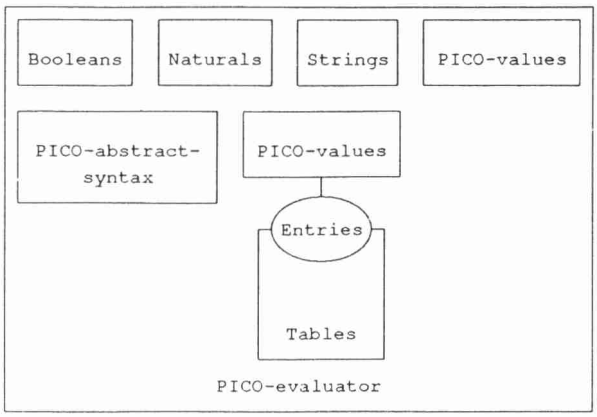


*Fig. 2.11: Structure diagram for evaluator*

There is a technical problem here, since the evaluation of programs need not terminate. These non-terminating computations are reflected in the initial model of this specification by undesirable values of sort `PICO-VALUE` which are unequal to any of the standard values of this sort. More precisely: module `PICO-evaluator` is not *sufficiently complete* with respect to module `PICO-values` (see [GH78]). It is possible to eliminate these values by introducing auxiliary sorts and by adding conditions to all equations of the evaluator. In that way, the evaluation of each language construct will only be defined if the evaluation of all its components terminates. In the specification as presented here, this additional complexity has been avoided at the expense of introducing such undesirable values.

```
module PICO-evaluator
begin
  exports
    begin
      functions
        eval: PICO-PROGRAM           -> VALUE-ENV
        eval: DECLS # VALUE-ENV      -> VALUE-ENV
        eval: SERIES # VALUE-ENV     -> VALUE-ENV
        eval: STATEMENT # VALUE-ENV -> VALUE-ENV
        eval: EXP # VALUE-ENV        -> PICO-VALUE
    end

  imports Booleans, Naturals, Strings,
          PICO-values, PICO-abstract-syntax,
```

```
        Tables
              { renamed by
                  [ TABLE -> VALUE-ENV,
                    null-table -> null-value-env]
                Entries bound by
                  [ ENTRY -> PICO-VALUE,
                    error-entry -> error-value]
                to PICO-values }
functions
  append-statement: SERIES # STATEMENT -> SERIES

variables
  dec :-> DECLS
  ser, ser1, ser2 :-> SERIES
  stm, stm1, stm2 :-> STATEMENT
  name :-> STRING
  nat, nat1, nat2 :-> NAT
  val :-> PICO-VALUE
  str, str1, str2 :-> STRING
  x, x1, x2 :-> EXP
  env :-> VALUE-ENV
  found :-> BOOL

equations

[157] eval(abs-pico-program(dec, ser))
        = eval(ser, eval(dec, null-value-env))

[158] eval(abs-decls(abs-id(name), natural-type, dec),
        env)
        = eval(dec, table(name, pico-value(0), env))
[159] eval(abs-decls(abs-id(name), string-type, dec),
        env)
        = eval(dec,
           table(name,pico-value(null-string), env))
[160] eval(abs-empty-decls, env)
        = env

[161] eval(abs-series(stm, ser), env)
        = eval(ser, eval(stm, env))

[162] eval(abs-empty-series, env)
        = env

[163] eval(abs-assign(abs-id(name), x), env)
```

```
            = table(name, eval(x, env), env)

   [164] eval(abs-if(x, ser1, ser2), env)
            = if(eq(eval(x,env), pico-value(0)),
                 eval(ser2, env),
                 eval(ser1, env))

   [165] eval(abs-while(x, ser), env)
            = if(eq(eval(x,env), pico-value(0)),
                 env,
                 eval(append-statement(ser,
                                       abs-while(x,ser)),
                      env))

   [166] eval(abs-plus(x1, x2), env)
            = pico-value(add(nat1,nat2))
              when pico-value(nat1) = eval(x1, env),
                   pico-value(nat2) = eval(x2, env)

   [167] eval(abs-minus(x1, x2), env)
            = pico-value(sub(nat1,nat2))
              when pico-value(nat1) = eval(x1, env),
                   pico-value(nat2) = eval(x2, env)

   [168] eval(abs-conc(x1, x2), env)
            = pico-value(conc(str1, str2))
              when pico-value(str1) = eval(x1, env),
                   pico-value(str2) = eval(x2, env)

   [169] eval(abs-natural-constant(nat), env)
            = pico-value(nat)

   [170] eval(abs-string-constant(str), env)
            = pico-value(str)

   [171] eval(abs-var(abs-id(name)), env)
            = val
              when <found, val> = lookup(name, env)

   [172] append-statement(abs-empty-series, stm)
            = abs-series(stm, abs-empty-series)
   [173] append-statement(abs-series(stm1, ser), stm2)
            = abs-series(stm1,
                         append-statement(ser, stm2))
end PICO-evaluator
```

### 2.4.4 The first PICO system

The previous definitions of abstract syntax, typechecking and evaluation are now combined in a single "system". This system defines the function `run-pico-abstree` which specifies how one can "run" a given PICO program (in the form of an abstract syntax tree). Running a PICO program amounts to typechecking and evaluating it. If one of these steps fails, the result of running the program is the constant `error-value`.

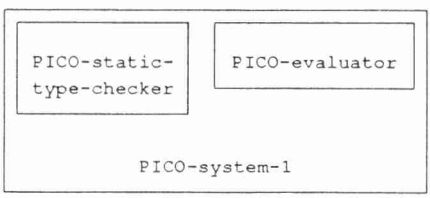


*Fig. 2.12:*
*Structure diagram for first PICO system*

```
module PICO-system-1
begin
  exports
    begin
      functions
        run-pico-abstree: PICO-PROGRAM -> PICO-VALUE
    end

  imports PICO-static-type-checker, PICO-evaluator

  functions
    run-pico-abstree' : PICO-PROGRAM -> PICO-VALUE

  variables
    abs-prog :-> PICO-PROGRAM
    has-output :-> BOOL
    v :-> PICO-VALUE
  equations
  [174] run-pico-abstree(abs-prog)
          = if(check(abs-prog),
               run-pico-abstree'(abs-prog),
               error-value)
  [175] run-pico-abstree'(abs-prog)
          = if(has-output, v, error-value)
            when <has-output, v> =
                 lookup("output", eval(abs-prog))
end PICO-system-1
```

## 2.5 Context-free parsing

In the previous section we have concentrated on the problem of specifying typechecking and evaluation of programs in the form of abstract syntax trees. The problem of converting a given program from its textual form into an abstract syntax tree was left undiscussed. This will be the subject of the current and the next section. Our approach will be to specify a general parser generator and to use this parser generator to obtain a parser for PICO. One could object that this aproach is much too general. It would actually be much simpler just to specify a parser for PICO. However, the exercise of specifying a general parser generator provides several opportunities to experiment with parameterization and modularization as will become clear shortly.

First we address the problem of how a context-free grammar can be specified within the algebraic framework and how the parsing process is to be described. A syntactic definition of a language can globally be subdivided in definitions for:

**lexical syntax**: which defines the tokens of the language, i.e., keywords, identifiers, punctuation marks, etc.

**context-free syntax**: which defines the concrete form of programs, i.e. the sequences of tokens that constitute a legal program.

**abstract syntax**: which defines the abstract tree structure underlying the concrete (textual) form of programs. As we have seen before, all further operations on programs can be defined as operations on the abstract syntax tree (see Sections 2.4.2 and 2.4.3).

In this section, we will define a parser (`Context-free-parser`, see Section 2.5.4) which is parameterized with a lexical scanner and a grammar describing the concrete syntax and the construction rules for abstract syntax trees. The parsing problem is decomposed as follows:

- Lexical analysis is delegated to a `Scanner` (a parameter of `Context-free-parser`), which transforms an input string into a sequence of lexical tokens (2.5.1). A token is a pair of strings: the first describes the lexical category of the token, the second gives the string value of the token.
- Abstract syntax trees are represented by the data type `Atrees`. Rules for the construction of abstract syntax trees are part of the grammar for a given language. The essential function is `build`, which specifies for each non-terminal how certain (named) components of the syntax rule have to be combined into an abstract syntax tree (2.5.2).

- BNF patterns (2.5.3) are introduced to allow the description of arbitrary context-free grammars. The main functions and operators introduced are `t` (indicates a terminal in the grammar), `n` (indicates a non-terminal), `+` (sequential composition of components of a grammar rule), and `|` (alternation). A grammar constructed by means of these operators can later be bound to the parameter `Syntax` of `Context-free-parser`.
- Actual parsing is described in `Context-free-parser` (2.5.4). This module has four parameters of which two are inherited from imported modules. The parameters `Scanner` and `Syntax` define the interface with the lexical scanner and with the concrete syntax and abstract syntax. `Context-free-parser` imports `BNF-patterns` (inheriting the unbound parameter `Non-terminals`) and `Atree-environments` (inheriting the unbound parameter `Operators`). `Context-free-parser` describes a parser which is driven by the BNF operators occurring in `Syntax`. We require that `Syntax` represents an unambiguous grammar.

In the next section we will apply `Context-free-parser` in order to obtain a scanner and parser for PICO.

### 2.5.1 Interface with lexical scanner

Lexical analysis transforms an input string into a sequence of lexical tokens. A token is a pair of strings: the first describes the lexical category of the token, the second gives the string value of the token, e.g., `token("identifier", "xyz")` or `token("natural-constant", "35")`. In this section, the data types `Tokens` and `Token-sequences` are defined.

```
module Tokens
begin
  exports
    begin
      sorts TOKEN
      functions
        token: STRING # STRING  -> TOKEN
        eq   : TOKEN # TOKEN    -> BOOL
    end

  imports Booleans, Strings
```

```
  variables
    s1, s2, s3, s4 :-> STRING

  equations

  [176] eq(token(s1, s2), token(s3, s4))
          = and(eq(s1, s3), eq(s2, s4))
end Tokens

module Token-sequences
begin
  imports Sequences
              { renamed by
                  [ SEQ -> TOKEN-SEQUENCE,
                    null -> null-token-sequence ]
                Items bound by
                  [ ITEM -> TOKEN,
                    eq -> eq ]
                to Tokens }
end Token-sequences
```

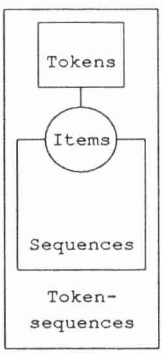


*Fig. 2.13:*
*Structure diagram for token sequences*

### 2.5.2 Interface with abstract syntax tree constructor

Abstract syntax trees are defined by the data type `Atrees`. Abstract syntax trees are essentially labeled trees whose nodes consist of an operator, indicating the construction operator of the node, and zero or more abstract syntax trees as sons. `Atrees` has one parameter `Operators`, which defines the interface to the set of operators for constructing abstract syntax trees.

Conversion functions are defined for the common cases that the leaves of the abstract syntax tree consist of `Strings`, `Naturals` or `Tokens`.

The construction process for abstract syntax trees as described in Section 2.5.4 uses the notion of environments of abstract syntax trees, i.e., tables which map strings onto abstract syntax trees. This notion is realized by the data type `Atree-environments`. Note that the parameter `Operators` of `Atrees` is inherited by `Atree-environments`.

```
module Atrees
begin

  parameters
    Operators
    begin
      sorts OPERATOR
    end Operators

  exports
    begin
      sorts ATREE
      functions
        error-atree  :                                -> ATREE
        null-atree   :                                -> ATREE
        atree        : OPERATOR                       -> ATREE
        atree        : OPERATOR # ATREE               -> ATREE
        atree        : OPERATOR # ATREE # ATREE
                                                      -> ATREE
        atree        : OPERATOR # ATREE #
                                  ATREE # ATREE       -> ATREE
        string-atree : STRING                         -> ATREE
        natural-atree: NAT                            -> ATREE
        lexical-atree: TOKEN                          -> ATREE
    end

  imports Booleans, Naturals, Strings, Tokens

end Atrees

module Atree-environments
begin
  exports
    begin
      functions
```

```
        _ ^ _ : STRING # ATREE-ENV -> ATREE
    end

  imports Tables
        { renamed by
            [ TABLE -> ATREE-ENV,
              null-table -> null-atree-env]
          Entries bound by
            [ ENTRY -> ATREE,
              error-entry -> error-atree]
            to Atrees }
  variables
    s :-> STRING
    e :-> ATREE-ENV
    f :-> BOOL
    v :-> ATREE

  equations

  [177] s ^ e    = v
                   when <f, v> = lookup(s, e)

end Atree-environments
```

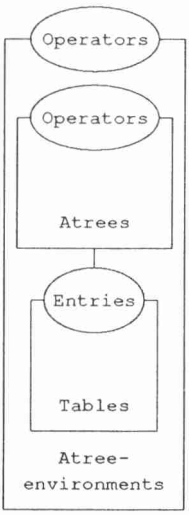


*Fig. 2.14:*
*Structure diagram for atree-environments*

### 2.5.3 BNF patterns

BNF patterns are introduced to allow the description of arbitrary context-free grammars. The main functions and operators introduced are `t` (indicates a terminal in the grammar), `n` (indicates a non-terminal), `lexical` (indicates a lexical item), `+` (sequential composition of components of a grammar rule), and `|` (alternation). The functions `t`, `n` and `lexical` have two variants: the variant with one argument indicates respectively a terminal, non-terminal or lexical item; the variant with two arguments also associates a name with the syntactic notion. These names can later be used to refer to the abstract syntax tree which is the result of parsing the given syntactic notion. An actual grammar constructed with these operators can be bound to the parameter `Syntax` of `Context-free-parser`. Examples of grammars using this notation are the lexical syntax (2.6.1.2) and concrete syntax (2.6.2.2) of PICO.

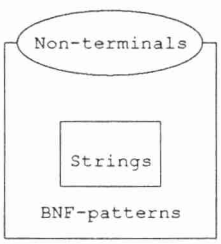


*Fig. 2.15:*
*Structure diagram for BNF patterns*

```
module BNF-patterns
begin

  parameters
    Non-terminals
    begin
      sorts NON-TERMINAL
    end Non-terminals

  exports
    begin
      sorts PATTERN
      functions

        _ + _    : PATTERN # PATTERN      -> PATTERN
        _ | _    : PATTERN # PATTERN      -> PATTERN
```

```
        t       : STRING                  -> PATTERN
        t       : STRING # STRING         -> PATTERN
        n       : NON-TERMINAL            -> PATTERN
        n       : NON-TERMINAL # STRING   -> PATTERN
        lexical : STRING                  -> PATTERN
        lexical : STRING # STRING         -> PATTERN
        null-pattern:                     -> PATTERN
    end

  imports Strings

end BNF-patterns
```

### 2.5.4 Context-free parser

`Context-free-parser` describes the actual parsing process. It has four parameters of which two are inherited from imported modules. Parameter `Scanner` defines the interface with the lexical scanner, i.e., the function `scan` which converts input strings to `Token-sequences`. Parameter `Syntax` defines the interface with the rules of the syntax (function `rule`) and with the rules for constructing abstract syntax trees (function `build`). `Context-free-parser` imports `BNF-patterns` (inheriting the unbound parameter `Non-terminals`, which defines the interface with the set of non-terminals of the syntax) and `Atree-environments` (inheriting the unbound parameter `Operators`, which defines the interface with the set of construction operators for the abstract syntax).

`Context-free-parser` describes a parser for the language described by the syntax rules. The equations in `Context-free-parser` describe for each type of BNF operator the conditions under which (a part of) the input `Token-sequence` is acceptable. The BNF operator `n` (non-terminal) uses the function `rule` from parameter `Syntax` to associate a pattern with a non-terminal. Acceptance of a part of the input is expressed by constructing an `Atree-environment` consisting of named `Atrees`. Acceptance of a non-terminal is expressed by the function `build` from `Syntax` for that non-terminal.

We require that the syntax is unambiguous. This simplifies the definition of `Context-free-parser` considerably: in the definition given below only *one* abstract syntax tree has to be constructed instead of a *set* of abstract syntax trees as would be necessary in the case of an ambiguous grammar.

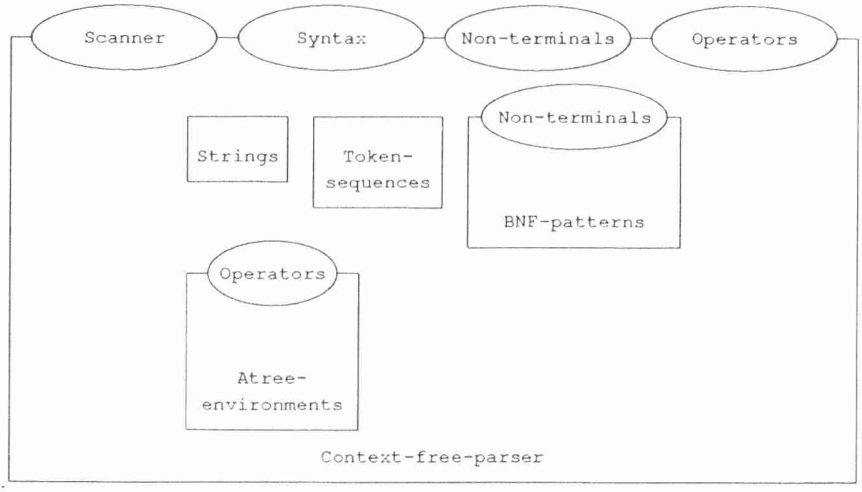


*Fig. 2.16: Structure diagram for context-free parser*

```
module Context-free-parser
begin
  parameters
    Scanner
    begin
      functions
        scan: STRING -> TOKEN-SEQUENCE
    end Scanner,

    Syntax
    begin
      functions
        rule : NON-TERMINAL              -> PATTERN
        build: NON-TERMINAL # ATREE-ENV -> ATREE
    end Syntax

  exports
    begin
      functions
        parse : NON-TERMINAL # STRING    -> ATREE
    end

  imports Strings, Token-sequences,
          BNF-patterns, Atree-environments
```

```
functions
  parse-rule: NON-TERMINAL # TOKEN-SEQUENCE
                -> BOOL # ATREE # TOKEN-SEQUENCE
  parse-pat : PATTERN # TOKEN-SEQUENCE # ATREE-ENV
                -> BOOL # ATREE-ENV # TOKEN-SEQUENCE

variables
  x :-> NON-TERMINAL
  p, p1, p2 :-> PATTERN
  env, env1, env2 :-> ATREE-ENV
  atree, atree1, atree2 :-> ATREE
  s, tail, tail1, tail2 :-> TOKEN-SEQUENCE
  id, val, str, lextype :-> STRING
  r, r1, r2 :-> BOOL

equations
[178] parse(x, str)
        = if(and(r, eq(tail, null-token-sequence)),
             atree,
             error-atree)
          when
            <r,atree,tail> = parse-rule(x,scan(str))

[179] parse-rule(x, s)
        = if(r, < true, build(x, env), tail >,
             < false, error-atree, tail >)
          when <r, env, tail> =
            parse-pat(rule(x), s, null-atree-env)

[180] parse-pat(null-pattern, s, env)
        = <true, env, s>

[181] parse-pat(p1 + p2, s, env1)
        = if(r, parse-pat(p2, tail, env2),
             < false, env2, tail >)
          when <r,env2,tail> = parse-pat(p1,s,env1)

[182] parse-pat(p1 | p2, s, env)
        = if(not(r1),
             < r2, env2, tail2 >,
             if(not(r2),
                < r1, env1, tail1 >,
                < false, env, s >))
          when <r1,env1,tail1> = parse-pat(p1,s,env),
               <r2,env2,tail2> = parse-pat(p2,s,env)
```

```
[183] parse-pat(n(x), s, env)
        = <r, env, tail>
          when <r, atree, tail> = parse-rule(x, s)

[184] parse-pat(n(x,id), s, env)
        = if(r,
             < true, table(id, atree, env), tail >,
             < false, env, tail >)
          when <r, atree, tail> = parse-rule(x, s)

[185] parse-pat(t(str),
                seq(token(lextype, val), s),
                env)
        = if(and(eq(str, val),
                 or(eq(lextype, "keyword"),
                    eq(lextype, "literal"))),
             < true, env, s>,
             < false, env, s> )

[186] parse-pat(t(str), null-token-sequence, env)
        = if(eq(str, null-string),
             <true, env, null-token-sequence>,
             <false, env, null-token-sequence>)

[187] parse-pat(t(str, id),
                seq(token(lextype, val), s),
                env)
        = if(and(eq(str, val),
                 or(eq(lextype, "keyword"),
                    eq(lextype, "literal"))),
             < true,
               table(id,
                     lexical-atree(token(lextype,
                                         str)),
                     env),
               s >,
             < false, env, s > )

[188] parse-pat(t(str, id), null-token-sequence, env)
        = if(eq(str, null-string),
             < true,
               table(id,
                     lexical-atree(token("literal",
                                         "")),
                     env),
```

```
                null-token-sequence>,
              < false, env, null-token-sequence >)

[189] parse-pat(lexical(str),
                seq(token(lextype,val),s),
                env)
        = if(eq(lextype, str),
             < true, env, s >,
             < false, env, s > )

[190] parse-pat(lexical(str),
                null-token-sequence,
                env)
        = < false, env, null-token-sequence >

[191] parse-pat(lexical(str,id),
                seq(token(lextype, val), s),
                env)
        = if(eq(lextype, str),
             < true,
               table(id,
                     lexical-atree(token(lextype,
                                         val)),
                     env),
               s >,
             < false, env, s> )

[192] parse-pat(lexical(str,id),
                null-token-sequence,
                env)
        = <false, env, null-token-sequence>

end Context-free-parser
```

## 2.6 Second system: representing programs as strings

After the preparations in the previous section, the following steps are still needed to obtain a specification of PICO in which not only typechecking and evaluation, but also parsing and construction of abstract syntax trees are covered:

**Lexical syntax** (2.6.1): The lexical syntax of PICO has to be specified. This is done by constructing a lexical scanner on the basis of

`Context-free-parser` as defined in the previous section.

**Abstract and concrete syntax** (2.6.2): The concrete syntax of PICO and the rules for the construction of abstract syntax trees have to be specified. This is accomplished by a *second* application of `Context-free-parser`.

**PICO system 2** (2.6.3): The above components have to be combined with the first PICO system (2.4.4) into a second PICO system. The first system already covered typechecking and evaluation of programs in the form of abstract syntax trees. This second system adds the functionality of converting programs as text to programs as abstract syntax trees.

### 2.6.1 Lexical syntax

The lexical syntax describes the lexical tokens that may occur in a PICO program. We construct a lexical scanner for PICO by means of `Context-free-parser`:

- A character-level scanner is defined (2.6.1.1). This character-level scanner distinguishes characters according to their character types, i.e., letter, digit, layout, etc.
- The lexical syntax for PICO and the construction rules for lexical tokens are defined (2.6.1.2). This amounts to defining the syntactic form of identifiers, strings, etc. and to defining the result for each case, e.g., parsing the non-terminal `natural-constant` of the lexical syntax will have as result `token("natural-constant", x)`, where `x` is the string representation of the natural number constant.
- A lexical scanner for PICO is obtained by combining the results of the previous two steps with `Context-free-parser` (2.6.1.3).

#### *2.6.1.1 Lexical character scanner*

`PICO-lexical-character-scanner` defines the character-level scanner `char-scan` which distinguishes characters according to their character types, i.e., letter, digit, layout and literal, and converts the input string into a `Token-sequence`.

```
module PICO-lexical-character-scanner
begin
  exports
    begin
      functions
        char-scan: STRING -> TOKEN-SEQUENCE
    end
```

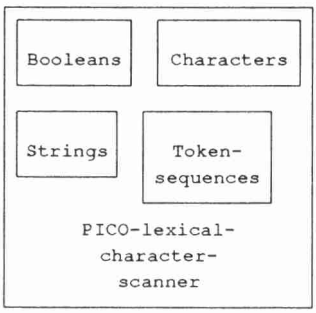


*Fig. 2.17:*
*Structure diagram for character scanner*

```
  imports Booleans, Characters, Strings,
          Token-sequences

  functions
    char-scan1: CHAR -> TOKEN
    is-layout : CHAR -> BOOL

  variables
    c :-> CHAR
    str :-> STRING

  equations
  [193] char-scan(seq(c, str))
           = seq(char-scan1(c), char-scan(str))

  [194] char-scan("")
           = null-token-sequence

  [195] char-scan1(c)
           = if(is-layout(c),
                token("layout", string(c)),
                if(is-letter(c),
                   token("letter",. string(c)),
                   if(is-digit(c),
                      token("digit", string(c)),
                      token("literal",string(c)))))

  [196] is-layout(c)
           = or(eq(c, char-space),
                or(eq(c, char-ht), eq(c, char-nl)))
end PICO-lexical-character-scanner
```

#### *2.6.1.2 Lexical syntax and rules for token construction*

The lexical syntax for PICO and the construction rules for lexical tokens are now defined. This amounts to defining the syntactic form of identifiers, strings, and so on, and to defining the result for each case, e.g., parsing the non-terminal `natural-constant` of the lexical syntax will have as result `token("natural-constant", x)`, where `x` is the string representation of the natural number constant. The following data types are defined:

- `PICO-non-terminals-of-lexical-syntax`: defines the sort `LEX-NON-TERMINAL` and all non-terminals of the lexical syntax.
- `PICO-lex-BNF-patterns`: defines a version of `BNF-patterns` with parameter `Non-terminals` bound to `PICO-non-terminals-of-lexical-syntax`.
- `PICO-atree-operators-of-lexical-syntax`: defines the sort `LEX-OPERATOR` and the operators for constructing abstract syntax trees for the lexical syntax.
- `PICO-lex-atree-environments`: defines a version of `Atree-environments` with parameter `Operators` bound to `PICO-atree-operators-of-lexical-syntax`.

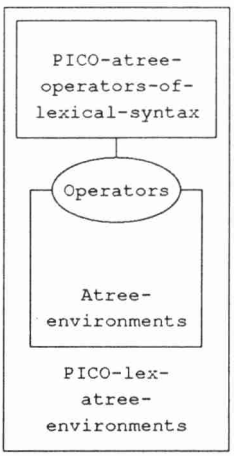


*Fig. 2.18:*
*Structure diagram for (lexical) atree-environments*

- `PICO-lexical-syntax`: defines the lexical syntax for PICO and the rules for token construction. Essentially, the grammar contains for each non-terminal pairs of equations for the functions `rule` (i.e., the actual syntax rule) and `build` (i.e., the construction procedure for abstract syntax trees). Note that all syntax rules with names starting with `non-`

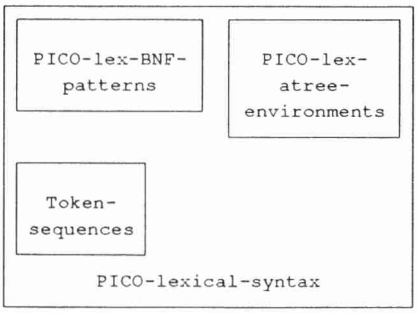


*Fig. 2.19:*
*Structure diagram for lexical syntax*

`empty` do not appear in the original grammar. These rules are artefacts made necessary by limitations in the descriptive power of `BNF-patterns`; most notably, it is impossible to associate different `build` functions with the alternatives in one `rule`.

```
module PICO-non-terminals-of-lexical-syntax
begin
  exports
    begin
      sorts LEX-NON-TERMINAL
      functions
        lexical-stream           : -> LEX-NON-TERMINAL
        non-empty-lexical-stream: -> LEX-NON-TERMINAL
        empty-lexical-stream     : -> LEX-NON-TERMINAL
        lexical-item             : -> LEX-NON-TERMINAL
        optional-layout          : -> LEX-NON-TERMINAL
        keyword-or-ident         : -> LEX-NON-TERMINAL
        ident                    : -> LEX-NON-TERMINAL
        ident-chars              : -> LEX-NON-TERMINAL
        non-empty-ident-chars    : -> LEX-NON-TERMINAL
        ident-char               : -> LEX-NON-TERMINAL
        natural-const            : -> LEX-NON-TERMINAL
        digits                   : -> LEX-NON-TERMINAL
        non-empty-digits         : -> LEX-NON-TERMINAL
        digit                    : -> LEX-NON-TERMINAL
        string-const             : -> LEX-NON-TERMINAL
        string-tail              : -> LEX-NON-TERMINAL
        non-empty-string-tail    : -> LEX-NON-TERMINAL
        quote                    : -> LEX-NON-TERMINAL
        any-char-but-quote       : -> LEX-NON-TERMINAL
        letter                   : -> LEX-NON-TERMINAL
```

```
        layout                      : -> LEX-NON-TERMINAL
        literal                     : -> LEX-NON-TERMINAL
        concat                      : -> LEX-NON-TERMINAL
        assign-or-colon             : -> LEX-NON-TERMINAL
        empty                       : -> LEX-NON-TERMINAL
    end
end PICO-non-terminals-of-lexical-syntax

module PICO-lex-BNF-patterns
begin
  imports BNF-patterns
        { renamed by
            [ PATTERN -> LEX-PATTERN,
              null-pattern -> null-lex-pattern,
              t -> lt ]
          Non-terminals bound by
            [ NON-TERMINAL -> LEX-NON-TERMINAL ]
            to PICO-non-terminals-of-lexical-syntax
        }
end PICO-lex-BNF-patterns

module PICO-atree-operators-of-lexical-syntax
begin
  exports
    begin
      sorts LEX-OPERATOR
      functions
        op-lex-item  :  -> LEX-OPERATOR
        op-lex-stream:  -> LEX-OPERATOR

    end

  imports Booleans

end PICO-atree-operators-of-lexical-syntax

module PICO-lex-atree-environments
begin

  imports Atree-environments
        { renamed by
            [ ATREE -> LEX-ATREE,
              atree -> lex-atree,
              null-atree -> null-lex-atree,
              error-atree -> error-lex-atree,
```

```
            lexical-atree -> lexical-lex-atree,
            ATREE-ENV -> LEX-ATREE-ENV,
            null-atree-env -> null-lex-atree-env ]
        Operators bound by
          [ OPERATOR -> LEX-OPERATOR ]
          to PICO-atree-operators-of-lexical-syntax
    }
end  PICO-lex-atree-environments

module PICO-lexical-syntax
begin

  exports
    begin
      functions
        rule       : LEX-NON-TERMINAL     -> LEX-PATTERN
        build      : LEX-NON-TERMINAL #
                              LEX-ATREE-ENV -> LEX-ATREE
        lex-stream: TOKEN-SEQUENCE        -> LEX-ATREE
        lex-item   : TOKEN                -> LEX-ATREE
    end

  imports PICO-lex-BNF-patterns,
          PICO-lex-atree-environments, Token-sequences

  variables
    env:-> LEX-ATREE-ENV
    l, l1, l2 :-> TOKEN-SEQUENCE
    t, t1, t2 :-> TOKEN
    s, s1, s2 :-> STRING
    d, d1, d2 :-> STRING

  equations

  [197] rule(lexical-stream)
          = n(non-empty-lexical-stream,"ls", |
            n(empty-lexical-stream,"ls")
  [198] build(lexical-stream, env)
          = "ls" ^ env
  [199] rule(non-empty-lexical-stream)
          = n(lexical-item,"t") + n(lexical-stream,"l")
  [200] build(non-empty-lexical-stream, env)
          = lex-atree(op-lex-stream,
                      lex-stream(seq(t, l)))
            when lex-atree(op-lex-item,lex-item(t))
```

```
                    = "t" ^ env,
                lex-atree(op-lex-stream,lex-stream(l))
                    = "l" ^ env
[201] rule(empty-lexical-stream)
        = n(empty)
[202] build(empty-lexical-stream, env)
        = lex-atree(op-lex-stream,
                    lex-stream(null-token-sequence))

[203] rule(lexical-item)
        = n(optional-layout) +
            ( n(keyword-or-ident,"i") |
              n(natural-const,"i") |
              n(string-const,"i") |
               n(literal,"i"))
[204] build(lexical-item, env)
        = "i" ^ env

[205] rule(optional-layout)
        = n(layout) | n(empty)
[206] build(optional-layout, env)
        = null-lex-atree

[207] rule(keyword-or-ident)
        = n(ident,"i")
[208] build(keyword-or-ident, env)
        = if(or(eq(s, "begin"),
             or(eq(s, "end"),
             or(eq(s, "declare"),
             or(eq(s, "natural"),
             or(eq(s, "string"),
             or(eq(s, "if"),
             or(eq(s, "then"),
             or(eq(s, "else"),
             or(eq(s, "fi"),
             or(eq(s, "while"),
             or(eq(s, "do"),
             eq(s, "od")))))))))))),
             lex-atree(op-lex-item,
                       lex-item(token("keyword",
                                      s))),
             lex-atree(op-lex-item,
                       lex-item(token("id", s))))

          when lexical-lex-atree(token("id",s))
```

```
                = "i" ^ env

[209] rule(ident)
         = n(letter,"s1") + n(ident-chars,"s2")
[210] build(ident, env)
         = lexical-lex-atree(token("id",
                                   conc(s1, s2)))
           when string-atree(s1) = "s1" ^ env,
                string-atree(s2) = "s2" ^ env

[211] rule(ident-chars)
         = n(non-empty-ident-chars,"s") |
           n(empty,"s")
[212] build(ident-chars, env)
         = "s" ^ env
[213] rule(non-empty-ident-chars)
         = n(ident-char,"s1") + n(ident-chars,"s2")
[214] build(non-empty-ident-chars, env)
         = string-atree(conc(s1, s2))
           when string-atree(s1) = "s1" ^ env,
                string-atree(s2) = "s2" ^ env
[215] rule(ident-char)
         = n(letter,"x") | n(digit,"x")
[216] build(ident-char, env)
         = "x" ^ env

[217] rule(natural-const)
         = n(digit,"d1") + n(digits,"d2")
[218] build(natural-const, env)
         = lex-atree(op-lex-item,
              lex-item(token("natural-constant",
                             conc(d1, d2))))
           when string-atree(d1) = "d1" ^ env,
                string-atree(d2) = "d2" ^ env

[219] rule(digits)
         = n(non-empty-digits,"d") | n(empty,"d")
[220] build(digits, env)
         = "d" ^ env

[221] rule(non-empty-digits)
         = n(digit,"d1") + n(digits,"d2")
[222] build(non-empty-digits, env)
         = string-atree(conc(d1, d2))
```

```
            when string-atree(d1) = "d1" ^ env,
                 string-atree(d2) = "d2" ^ env

[223] rule(string-const)
         = n(quote) + n(string-tail,"s")
[224] build(string-const, env)
         = lex-atree(op-lex-item,
                lex-item(token("string-constant", s)))
           when string-atree(s) = "s" ^ env

[225] rule(string-tail)
         = n(non-empty-string-tail,"s") | n(quote,"s")
[226] build(string-tail, env)
         = "s" ^ env
[227] rule(non-empty-string-tail)
         = n(any-char-but-quote,"s1") +
           n(string-tail,"s2")
[228] build(non-empty-string-tail, env)
         = string-atree(conc(s1, s2))
           when string-atree(s1) = "s1" ^ env,
                string-atree(s2) = "s2" ^ env

[229] rule(quote)
         = lt(string(char-quote))
[230] build(quote, env)
         = string-atree("")

[231] rule(any-char-but-quote)
         = n(letter,"c") |
           n(digit,"c") |
           n(literal,"c") |
           n(layout,"c")
[232] build(any-char-but-quote, env)
         = "c" ^ env

[233] rule(letter)
         = lexical("letter","s")
[234] build(letter, env)
         = string-atree(s)
           when lexical-lex-atree(token("letter",s))
                   = "s" ^ env

[235] rule(digit)
         = lexical("digit","d")
[236] build(digit, env)
```

```
          = string-atree(d)
            when lexical-lex-atree(token("digit", d))
                    = "d" ^ env
  [237] rule(layout)
          = lexical("layout","s")
  [238] build(layout, env)
          = string-atree(s)
            when lexical-lex-atree(token("layout", s))
                    = "s" ^ env
  [239] rule(literal)
          = lt("(","s") |
            lt(")","s") |
            lt("+","s") |
            lt("-","s") |
            lt(";","s") |
            lt(",","s") |
            n(concat,"s") |
            n(assign-or-colon,"s")
  [240] build(literal, env)
          = "s" ^ env

  [241] rule(concat)
          = lt("|") + lt("|")
  [242] build(concat, env)
          = string-atree("||")

  [243] rule(assign-or-colon)
          = lt(":") + (lt("=","s") | n(empty,"s"))
  [244] build(assign-or-colon, env)
          = if(eq(s, "="),
               string-atree(":="),
               string-atree(":"))
            when string-atree(s) = "s" ^ env

  [245] rule(empty)
          = lt("")
  [246] build(empty, env)
          = string-atree("")

end PICO-lexical-syntax
```

#### *2.6.1.3 Lexical scanner*

A lexical scanner for PICO can now be obtained by combining the modules:

- `PICO-lexical-character-scanner`,
- `PICO-lexical-syntax`,
- `PICO-non-terminals-of-lexical-syntax`, and
- `PICO-atree-operators-of-lexical-syntax`

with `Context-free-parser`.

```
module PICO-lexical-scanner
begin
  exports
    begin
      functions
        lex-scan: STRING -> TOKEN-SEQUENCE
    end

  imports Context-free-parser
        { renamed by
            [ ATREE          -> LEX-ATREE,
              atree          -> lex-atree,
              null-atree     -> null-lex-atree,
              error-atree    -> error-lex-atree,
              lexical-atree  -> lexical-lex-atree,
              ATREE-ENV      -> LEX-ATREE-ENV,
              null-atree-env -> null-lex-atree-env,
              PATTERN        -> LEX-PATTERN,
              null-pattern   -> null-lex-pattern,
              t              -> lt ]
          Scanner bound by
            [ scan -> char-scan ]
            to PICO-lexical-character-scanner
          Syntax bound by
            [ rule -> rule,
              build -> build ]
            to PICO-lexical-syntax
          Non-terminals bound by
            [ NON-TERMINAL -> LEX-NON-TERMINAL ]
            to PICO-non-terminals-of-lexical-syntax
          Operators bound by
            [ OPERATOR -> LEX-OPERATOR ]
            to PICO-atree-operators-of-lexical-syntax
        }
```

```
  variables
    l :-> TOKEN-SEQUENCE
    s :-> STRING

  equations

  [247] lex-scan(s) = l
          when lex-atree(op-lex-stream,lex-stream(l))
                    = parse(lexical-stream, s)

end PICO-lexical-scanner
```

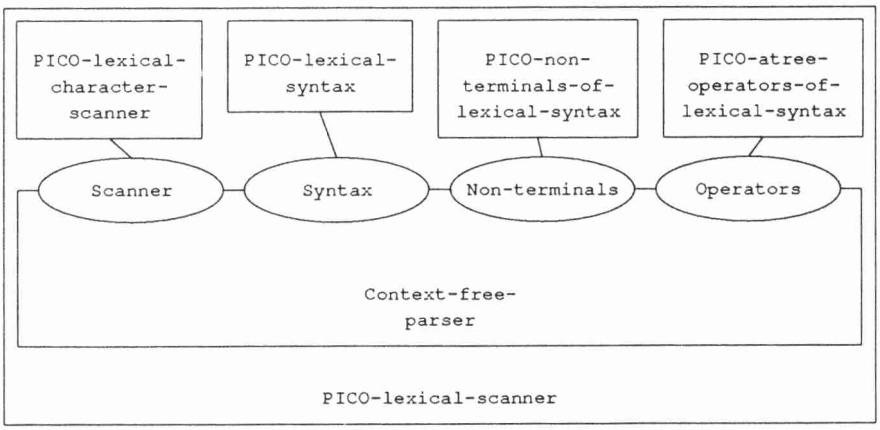


*Fig. 2.20: Structure diagram for lexical scanner*

### 2.6.2 Abstract and concrete syntax

Now we specify the abstract and concrete syntax of PICO. This will result in a specification of a parser that transforms PICO-programs from their concrete form into abstract syntax trees. We proceed as follows:

**Abstract syntax**: The abstract syntax for PICO has already been defined in Section 2.4.1.

**Abstract syntax expressed as general Atree** (2.6.2.1): The abstract syntax of PICO has been defined by means of a fixed signature. The trees constructed during context-free parsing have been expressed as elements of the general data type of `Atrees`. The correspondence between these two representations has to be defined.

**Concrete syntax and tree construction** (2.6.2.2): The concrete syntax and

the rules for constructing abstract syntax trees are defined.

**Parser** (2.6.2.3): The lexical scanner (as defined in the previous section), the concrete syntax and the rules for the construction of abstract syntax trees (both defined in this section) are combined with `Context-free-parser`. In this way we obtain a parser that transforms PICO programs into abstract syntax trees.

### *2.6.2.1 Abstract syntax expressed as Atree*

The abstract syntax for PICO is now defined by means of the data type `Atrees`. This involves the following data types:

- `PICO-atree-operators`: the operators for constructing abstract syntax trees.
- `PICO-atree-environments`: a version of `Atree-environments` with parameter `Operators` bound to `PICO-atree-operators`.

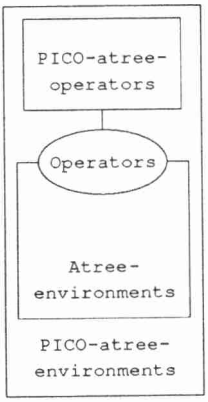


*Fig. 2.21:*
*Structure diagram for atree-environments*

- `PICO-abstract-syntax-as-atrees`: defines the relation between the constructor functions in the PICO abstract syntax (e.g., `abs-if`, `abs-while`, etc.) and their representation as term of `Atrees`.

```
module PICO-atree-operators
begin
  exports
    begin
      sorts PICO-OPERATOR
      functions
```

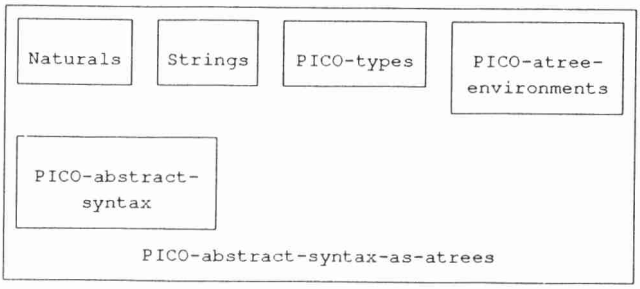


*Fig. 2.22: Structure diagram for abstract-syntax viewed as atrees*

```
        op-pico-program     :     -> PICO-OPERATOR
        op-decls            :     -> PICO-OPERATOR
        op-empty-decls      :     -> PICO-OPERATOR
        op-series           :     -> PICO-OPERATOR
        op-empty-series     :     -> PICO-OPERATOR
        op-assign           :     -> PICO-OPERATOR
        op-if               :     -> PICO-OPERATOR
        op-while            :     -> PICO-OPERATOR
        op-plus             :     -> PICO-OPERATOR
        op-minus            :     -> PICO-OPERATOR
        op-conc             :     -> PICO-OPERATOR
        op-var              :     -> PICO-OPERATOR
        op-natural-constant:      -> PICO-OPERATOR
        op-string-constant  :     -> PICO-OPERATOR
        op-id               :     -> PICO-OPERATOR
        op-natural-type     :     -> PICO-OPERATOR
        op-string-type      :     -> PICO-OPERATOR
    end

  imports Booleans, Naturals

end PICO-atree-operators

module PICO-atree-environments
begin
  imports Atree-environments
        { renamed by
            [ ATREE -> PICO-ATREE,
              atree -> pico-atree,
              null-atree -> null-pico-atree,
              error-atree -> error-pico-atree,
              string-atree -> string-pico-atree,
```

```
                natural-atree -> natural-pico-atree,
                lexical-atree -> lexical-pico-atree,
                ATREE-ENV -> PICO-ATREE-ENV,
                null-atree-env -> null-pico-atree-env ]
            Operators bound by
              [ OPERATOR -> PICO-OPERATOR ]
              to PICO-atree-operators }
end PICO-atree-environments

module PICO-abstract-syntax-as-atrees
begin
  exports
    begin
      functions
        pico-program    : PICO-ATREE -> PICO-PROGRAM
        decls           : PICO-ATREE -> DECLS
        series          : PICO-ATREE -> SERIES
        statement       : PICO-ATREE -> STATEMENT
        exp             : PICO-ATREE -> EXP
        id              : PICO-ATREE -> ID
        pico-type-atree: PICO-TYPE  -> PICO-ATREE
    end

  imports Naturals, Strings, PICO-types,
          PICO-atree-environments,
          PICO-abstract-syntax

  variables
    ds, sr, sr1, sr2, st, i, x, x1, x2 :-> PICO-ATREE
    t :-> PICO-TYPE
    str  :-> STRING
    n :-> NAT
    stat, stat1, stat2 :-> STATEMENT
    ser  :-> SERIES

  equations
  [248] abs-pico-program(decls(ds), series(sr))
          = pico-program(pico-atree(op-pico-program,
                                    ds, sr))
  [249] abs-decls(id(i), t, decls(ds))
          = decls(pico-atree(op-decls, i,
                             pico-type-atree(t),ds))
  [250] abs-empty-decls
          = decls(pico-atree(op-empty-decls))
  [251] abs-series(statement(st), series(sr))
```

```
          = series(pico-atree(op-series, st, sr))
[252] abs-empty-series
          = series(pico-atree(op-empty-series))
[253] abs-assign(id(i), exp(x))
          = statement(pico-atree(op-assign, i, x))
[254] abs-if(exp(x), series(sr1), series(sr2))
          = statement(pico-atree(op-if,x,sr1,sr2))
[255] abs-while(exp(x), series(sr))
          = statement(pico-atree(op-while, x, sr))
[256] abs-plus(exp(x1), exp(x2))
          = exp(pico-atree(op-plus, x1, x2))
[257] abs-minus(exp(x1), exp(x2))
          = exp(pico-atree(op-minus, x1, x2))
[258] abs-conc(exp(x1), exp(x2))
          = exp(pico-atree(op-conc, x1, x2))
[259] abs-var(id(i))
          = exp(pico-atree(op-var, i))
[260] abs-natural-constant(n)
          = exp(pico-atree(op-natural-constant,
                           natural-pico-atree(n)))
[261] abs-string-constant(str)
          = exp(pico-atree(op-string-constant,
                           string-pico-atree(str)))
[262] abs-id(str)
          = id(pico-atree(op-id,
                          string-pico-atree(str)))

end PICO-abstract-syntax-as-atrees
```

#### *2.6.2.2 Concrete syntax and rules for abstract syntax tree construction*

The concrete syntax and the rules for abstract syntax tree construction for PICO are now defined. This involves the following modules:

- `PICO-non-terminals-of-concrete-syntax`: defines the sort `PICO-NON-TERMINAL` and all non-terminals of the concrete syntax.
- `PICO-BNF-patterns`: defines a version of `BNF-patterns` with parameter `Non-terminals` bound to `PICO-non-terminals-of-concrete-syntax`.
- `PICO-concrete-syntax`: defines the concrete syntax for PICO and the rules for abstract syntax tree construction. Essentially the grammar contains for each non-terminal in the concrete syntax pairs of equations for the functions `rule` (i.e., the actual syntax rule) and `build` (i.e., the construction procedure for abstract syntax trees).

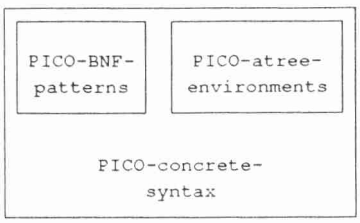


*Fig. 2.23:*
*Structure diagram for concrete syntax*

```
module PICO-non-terminals-of-concrete-syntax
begin
  exports
    begin
      sorts PICO-NON-TERMINAL
      functions
        pico-program      :     -> PICO-NON-TERMINAL
        decls             :     -> PICO-NON-TERMINAL
        empty-decls       :     -> PICO-NON-TERMINAL
        id-type-list      :     -> PICO-NON-TERMINAL
        type              :     -> PICO-NON-TERMINAL
        type-natural      :     -> PICO-NON-TERMINAL
        type-string       :     -> PICO-NON-TERMINAL
        series            :     -> PICO-NON-TERMINAL
        empty-series      :     -> PICO-NON-TERMINAL
        non-empty-series  :     -> PICO-NON-TERMINAL
        stat              :     -> PICO-NON-TERMINAL
        assign            :     -> PICO-NON-TERMINAL
        if                :     -> PICO-NON-TERMINAL
        while             :     -> PICO-NON-TERMINAL
        exp               :     -> PICO-NON-TERMINAL
        primary           :     -> PICO-NON-TERMINAL
        plus              :     -> PICO-NON-TERMINAL
        minus             :     -> PICO-NON-TERMINAL
        conc              :     -> PICO-NON-TERMINAL
        var               :     -> PICO-NON-TERMINAL
        id                :     -> PICO-NON-TERMINAL
        natural-constant  :     -> PICO-NON-TERMINAL
        string-constant   :     -> PICO-NON-TERMINAL
    end
end PICO-non-terminals-of-concrete-syntax
```

```
module PICO-BNF-patterns
begin
  imports BNF-patterns
        { renamed by
            [ PATTERN -> PICO-PATTERN,
              t -> pt,
              lexical -> plexical ]
          Non-terminals bound by
            [ NON-TERMINAL -> PICO-NON-TERMINAL ]
            to PICO-non-terminals-of-concrete-syntax
        }
end PICO-BNF-patterns

module PICO-concrete-syntax
begin
  exports
    begin
      functions
        rule : PICO-NON-TERMINAL    -> PICO-PATTERN
        build: PICO-NON-TERMINAL # PICO-ATREE-ENV
                                    -> PICO-ATREE
    end

  imports PICO-BNF-patterns, PICO-atree-environments

  variables
    env :-> PICO-ATREE-ENV
    str :-> STRING

  equations
  [263] rule(pico-program)
          = pt("begin") + n(decls,"d") +
            n(series,"s") + pt("end")
  [264] build(pico-program, env)
          = pico-atree(op-pico-program,
                       "d"^env, "s"^env)

  [265] rule(decls)
          = pt("declare") + n(id-type-list,"l") +
            pt(";")
  [266] build(decls, env)
          = "l" ^ env

  [267] rule(empty-decls)
          = pt("")
```

```
[268] build(empty-decls, env)
        = pico-atree(op-empty-decls)

[269] rule(id-type-list)
        = n(id,"i") + pt(":") + n(type,"t") +
          ( n(empty-decls,"l") |
            pt(",") + n(id-type-list,"l") )
[270] build(id-type-list, env)
        = pico-atree(op-decls, "i"^env, "t"^env,
                     "l"^env)

[271] rule(type)
        = n(type-natural,"t") |
          n(type-string,"t")
[272] build(type, env)
        = "t" ^ env

[273] rule(type-natural)
        = pt("natural")
[274] build(type-natural, env)
        = pico-atree(op-natural-type)

[275] rule(type-string)
        = pt("string")
[276] build(type-string, env)
        = pico-atree(op-string-type)

[277] rule(series)
        = n(empty-series,"s") |
          n(non-empty-series,"s")
[278] build(series, env)
        = "s" ^ env

[279] rule(empty-series)
        = pt("")
[280] build(empty-series, env)
        = pico-atree(op-empty-series)

[281] rule(non-empty-series)
        = n(stat,"st") +
          ( n(empty-series,"s") |
            pt(";") + n(series,"s") )
[282] build(non-empty-series, env)
        = pico-atree(op-series, "st"^env,
                     "s"^env)
```

```
[283] rule(stat)
        = n(assign,"st") | n(if,"st") |
          n(while,"st")
[284] build(stat, env)
        = "st" ^ env

[285] rule(assign)
        = n(id,"i") + pt(":=") + n(exp,"e")
[286] build(assign,env)
        = pico-atree(op-assign, "i"^env, "e"^env)

[287] rule(if)
        = pt("if") + n(exp,"e") +
          pt("then") + n(series,"s1") +
          pt("else") + n(series,"s2") + pt("fi")
[288] build(if, env)
        = pico-atree(op-if, "e"^env, "s1"^env,
                     "s2"^env)

[289] rule(while)
        = pt("while") + n(exp,"e") +
          pt("do") + n(series,"s") + pt("od")
[290] build(while, env)
        = pico-atree(op-while, "e"^env, "s"^env)

[291] rule(exp)
        = n(primary) |
          n(plus,"e") |
          n(minus,"e") |
          n(conc,"e")
[292] build(exp, env)
        = "e" ^ env

[293] rule(primary)
        = n(var,"e") |
          n(natural-constant,"e") |
          n(string-constant,"e") |
          ( pt("(") + n(exp,"e") + pt(")") )
[294] build(primary, env)
        = "e" ^ env

[295] rule(plus)
        = n(exp,"e1") + pt("+") + n(primary,"e2")
[296] build(plus, env)
        = pico-atree(op-plus, "e1"^env, "e2"^env)
```

```
[297] rule(minus)
        = n(exp,"e1") + pt("-") + n(primary,"e2")
[298] build(minus, env)
        = pico-atree(op-minus, "e1"^env,
                     "e2"^env)

[299] rule(conc)
        = n(exp,"e1") + pt("||") + n(primary,"e2")
[300] build(conc, env)
        = pico-atree(op-conc, "e1"^env, "e2"^env)
[301] rule(var)
        = n(id,"i")
[302] build(var, env)
        = pico-atree(op-var, "i"^env)

[303] rule(id)
        = plexical("id","i")
[304] build(id, env)
        = pico-atree(op-id,
                     string-pico-atree(str))
          when
            lexical-pico-atree(token("id", str))
              = "i" ^ env
[305] rule(natural-constant)
        = plexical("natural-constant","i")

[306] build(natural-constant, env)
        = pico-atree(op-natural-constant,
             natural-pico-atree(str-to-nat(str)))
          when
            lexical-pico-atree(
                 token("natural-constant", str))
              = "i" ^ env

[307] rule(string-constant)
        = plexical("string-constant","s")
[308] build(string-constant, env)
        = pico-atree(op-string-constant,
             string-pico-atree(str))
          when
            lexical-pico-atree(
                 token("string-constant", str))
              = "s" ^ env

end PICO-concrete-syntax
```

#### 2.6.2.3 Parser

A parser for PICO is now obtained by combining `PICO-lexical-scanner` (Section 2.6.1.3), `PICO-concrete-syntax` and `PICO-non-terminals-of-concrete-syntax`, (Section 2.6.2.2), and `PICO-abstract-syntax-as-atrees` (Section 2.6.2.1) with `Context-free-parser`.

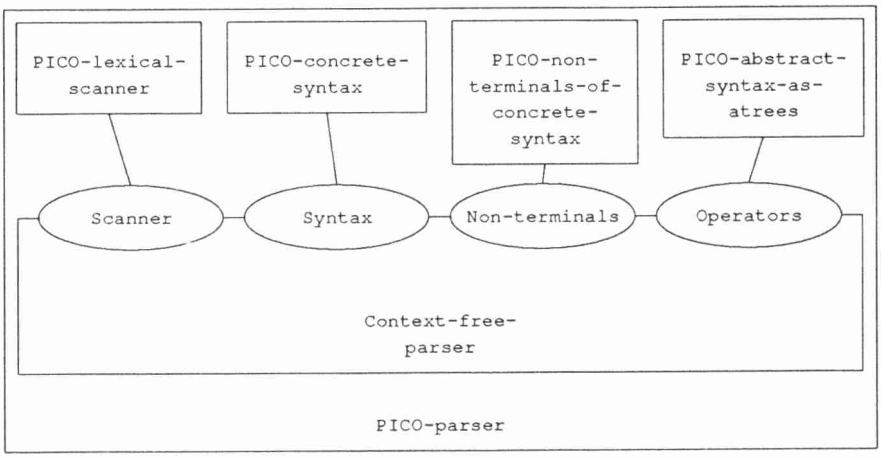


*Fig. 2.24: Structure diagram for parser*

```
module PICO-parser
begin
  exports
    begin
      functions
        parse-and-construct: STRING -> PICO-ATREE
    end

  imports Context-free-parser
        { renamed by
            [ ATREE -> PICO-ATREE,
              atree -> pico-atree,
              null-atree -> null-pico-atree,
              error-atree -> error-pico-atree,
              string-atree -> string-pico-atree,
              natural-atree -> natural-pico-atree,
              lexical-atree -> lexical-pico-atree,
              ATREE-ENV -> PICO-ATREE-ENV,
              null-atree-env -> null-pico-atree-env,
              PATTERN -> PICO-PATTERN,
```

```
            t -> pt,
            lexical -> plexical ]
        Scanner bound by
          [ scan -> lex-scan ]
          to PICO-lexical-scanner
        Syntax bound by
          [ rule -> rule,
            build -> build ]
          to PICO-concrete-syntax
        Non-terminals bound by
          [ NON-TERMINAL -> PICO-NON-TERMINAL ]
          to PICO-non-terminals-of-concrete-syntax
        Operators bound by
          [ OPERATOR -> PICO-OPERATOR ]
          to PICO-abstract-syntax-as-atrees }

  variables
    str :-> STRING

  equations
  [309] parse-and-construct(str)
          = parse(pico-program, str)

end PICO-parser
```

### 2.6.3 The second PICO system

In this final section we combine all previously defined modules to form a second PICO system. The top level function is `run` which converts, if possible, a string into a PICO-value. The following steps are necessary:

- The input string is parsed and converted into an abstract syntax tree using `parse-and-construct` as defined in `PICO-parser`.
- Typechecking and evaluation of the resulting abstract syntax tree have already been specified by the function `run-pico-abstree`, as given in the first PICO system (Section 2.4.4).

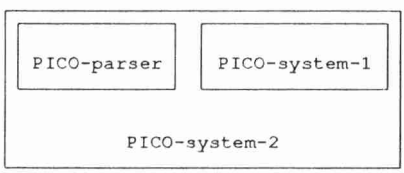


*Fig. 2.25:*
*Structure diagram for second PICO system*

```
module PICO-system-2
begin
  exports
    begin
      functions
        run-pico-string: STRING -> PICO-VALUE
    end

  imports PICO-parser, PICO-system-1

  functions
    run-pico-string': PICO-ATREE -> PICO-VALUE

  variables
    s :-> STRING
    p :-> PICO-ATREE

  equations

  [310] run-pico-string(s)
          = run-pico-string'(parse-and-construct(s))
  [311] run-pico-string'(error-pico-atree)
          =  error-value
  [312] run-pico-string'(p)
          = run-pico-abstree(pico-program(p))

end PICO-system-2
```

## 2.7 Assessment

Assessing the merits of the PICO specifications presented in this chapter we reach the following conclusions:

- The specification of elementary data types is adequate. However, in the specification of characters the ratio between size and content is clearly unacceptable.
- The specification of the first PICO system is adequate: the definition of abstract syntax is intuitively appealing and the definitions of typechecking and evaluation are straightforward.
- The specification of context-free parsing and of the second PICO system using it are less satisfactory. On the positive side, these specifications have merits as interesting excercises in constructing larger, modular

specifications. We feel that a complete algebraic specification of a general parser is also interesting in its own right. On the negative side, one is accustomed to defining grammars by means of concise BNF rules and leaving the process of parsing and tree construction implicit. Compared to such definitions, our explicit formulation of the general problem of scanning, parsing and tree construction looks too complicated.

In all these specifications, there are instances of the phenomenon where one has to specify *all possible cases*, while it would be more natural to specify the *interesting cases only*. Examples are the definition of `eq` on sorts `PICO-TYPE` (Section 2.3.7) and `PICO-VALUE` (Section 2.3.8), the function `check` in `PICO-static-type-checker` (Section 2.4.2), and the function `parse` in module `Context-free-parser` (Section 2.5.4).

These lessons will be reflected in two ways in the remainder of this book. First, the now following language definitions of SMALL (Chapter 3) and POOL (Chapter 4) both take the abstract syntax as their point of departure. All issues related to concrete syntax are ignored in these chapters. Second, we will come back to the problem of specifying syntax in Chapter 6, where SDF (Syntax Definition Formalism) is introduced. This formalism attempts to solve the problems that were identified when assessing the PICO specifications in the current chapter. Yet another definition of PICO (using SDF and a specification style in which *false* and *error* cases are left unspecified) can be found in Chapter 9.