Chapter 1. Writing Language Definitions in ASF+SDF

Paul Klint
Jurgen Vinju

2007-10-18 23:00:33 +0200 (Thu, 18 Oct 2007)

Table of Contents

Introduction ...................................................................................................................... 2
  What is the Global Picture? ................................................................. 2
  ASF+SDF ................................................................. 2
  The ASF+SDF Meta-Environment .................................................. 3
  Learning more ...................................................................................... 3
Preparations .................................................................................................................. 3
  Anatomy of an ASF+SDF specification ........................................ 3
  Anatomy of a single module .......................................................... 4
  An example: Booleans ........................................................................ 4
The toy language Pico .............................................................. 13
Define the syntax for Pico .......................................................... 13
  languages/pico/syntax/Types ...................................................... 14
  languages/pico/syntax/Identifiers ................................................ 14
  languages/pico/syntax/Pico .......................................................... 15
  Using the Pico syntax in the Meta-Environment ......................... 16
  Take home points ............................................................................ 18
  Recap: symbols .............................................................................. 18
  Recap: syntax rules ........................................................................ 19
Define a typechecker for Pico ..................................................... 20
  languages/pico/check/Type-environments .................................. 21
  languages/pico/check/Pico .......................................................... 21
  Using the Pico typechecker in The Meta-Environment ............... 25
  Take home points ............................................................................ 29
Defining an evaluator for Pico ..................................................... 29
  languages/pico/run/Values .......................................................... 30
  languages/pico/run/Value-environments .................................. 31
  languages/pico/run/Pico .............................................................. 31
  Using the Pico evaluator in the Meta-EnvironmentOpen ............ 34
  Take home points ............................................................................ 35
Defining a compiler for Pico ....................................................... 35
  languages/pico/compile/AssemblyLanguage .................................. 36
  languages/pico/compile/NextLabel ............................................ 37
  languages/pico/compile/Pico .......................................................... 38
  Using the Pico compiler in the Meta-Environment ...................... 42
  Take home points ............................................................................ 42
Intermezzo: traversal functions .................................................. 42
  Simple trees ...................................................................................... 43
  Counting tree nodes (classical solution) ....................................... 44
  Counting tree nodes (using an accumulator) ................................. 45
  Increment and modify tree leaves (using a transformer) .................. 46
  Increment with variable amount and modify tree leaves (using transformer) 46
Warning

This document is work in progress. See ToDo section.

Introduction

You are interested in using ASF+SDF to define various aspects of a programming language or domain-specific language but you do now want to wade through all the details in the manual (The Language Specification Formalism ASF+SDF)? In that case, this article may be for you. We take the toy language Pico as starting point and walk you through its syntax, typechecking, formatting, execution and more. We do this while assuming zero knowledge of ASF+SDF or The ASF+SDF Meta-Environment.

What is the Global Picture?

ASF+SDF can be used to define various aspects of programming languages and The ASF+SDF Meta-Environment can be used to edit and run these specifications.

ASF+SDF

The goal of ASF+SDF is to define the syntax (form) and semantics (meaning) of programming languages and domain-specific languages. The Syntax Definition Formalism (SDF) is used to define syntactic aspects including:

- Lexical syntax (keywords, comments, string constants, white space, ...).
- Context-free syntax (declarations, statements, ...).

The Algebraic Specification Formalism (ASF) is used to define semantic aspects such as:

- Type checking (are the variables that are used declared and are they used in a type-correct way?).
- Formatting (display the original program using user-defined rules for indentation and formatting).
- Fact extraction (extract all procedure calls or all declarations and uses of variables).
- Execution (run the program with given input values).

By convention, all these language aspects are located in dedicated subdirectories of a language definition:

- syntax (definitions for the syntax).
- check (definitions for type checking).
- format (definitions for formatting).
- extract (definitions for fact extraction).
• `run` (definitions for running a program).
• `debug` (definitions for debugging a program).

Apart from giving a standard structure to all language definitions, this organization also enables the seamless integration of these aspects in the user-interface of The Meta-Environment.

**The ASF+SDF Meta-Environment**

The goal of The ASF+SDF Meta-Environment (or The Meta-Environment for short) is to provide an Interactive Development Environment for ASF+SDF specifications. It supports interactive editing, checking and execution of ASF+SDF specifications. Behind the scenes, this implies the following tasks:

• Providing a graphical user-interface with editors and various visualization tools.
• Tracking changes to specification modules.
• Parsing and checking specification modules.
• Generating parsers for the syntax modules (SDF) that have been changed.
• Generating rewriter engines for the equations modules (ASF) that have been changed.
• Applying the ASF+SDF specification to programs in the language that is being defined by that specification.

The intended user experience of The Meta-Environment is the seamless automation of all these tasks.

**Learning more**

This article guides you through the various stages of writing language definitions in ASF+SDF. The following documents (all available at [http://www.meta-environment.org](http://www.meta-environment.org)) will help you to learn more:

• The basic concepts of syntax analysis are given in the article "Syntax Analysis".
• The basic concepts of term rewriting are discussed in the article "Term Rewriting".
• "The Syntax Definition Formalism SDF" is the reference manual for the SDF formalism.
• "The Language Specification Formalism ASF+SDF" is the reference manual for the ASF+SDF formalism.
• "Guided tour: Playing with Booleans" is a flash movie that gives you a quick overview of the user-interface of the Meta-Environment.

**Preparations**

The anatomies of a complete ASF+SDF specification as well as that of a single module are needed to understand any language definition.

**Anatomy of an ASF+SDF specification**

An ASF+SDF specification consists of a collection of modules as shown in the following Figure 1.1, "Module structure of an ASF+SDF specification" (p. 4). A module can import other modules and this can be understood as the textual inclusion of the imported modules. In this example, the text of modules B, C and D is literally inserted in module A.
Anatomy of a single module

A single module has the following structure:

```plaintext
module ModuleName
  ImportSection*  
  ExportOrHiddenSection*  
  equations
  ConditionalEquation*  
```

Notes:

1. The name of this module. It may be followed by parameters.
2. Zero or more sections that describe modules to be imported by this module.
3. The grammar elements (such as imports, aliases, sorts, lexical syntax, context-free syntax, priorities or variables) that are visible from the outside (defined by exports) or only inside the module (defined by hiddens).
4. The equations of the module that define the meaning of the grammar elements. Equations come in two flavours. Unconditional:

   ```plaintext
   [<TagId>] <Term₁> = <Term₂>
   ```

   and conditional:

   ```plaintext
   [<TagId>] <Condition₁>, <Condition₂>, ...

   ================

   <Term₁> = <Term₂>
   ```

In the unconditional case, an attempt is made to match Term₁ and if this succeeds it is replaced by Term₂ (after proper replacement of variables that result from the match). In the conditional case, an attempt is made to match Term₁ and if this succeeds the conditions are evaluated. If all conditions are true, Term₁ is replaced by Term₂ (again after proper replacement of variables that may in this case result both from the match and from the evaluation of the conditions.).

An example: Booleans

The Meta-Environment comes with a considerable library of built-in languages and datatypes. We explore here the datatype `basic/Booleans` in the ASF+SDF library.
Modular structure

The module structure of basic/Booleans is shown in Figure 1.2, "Modular structure of basic/Booleans" (p. 5).

Figure 1.2. Modular structure of basic/Booleans

We will now discuss all these modules in turn.

**basic/BoolCon**

Let's start with the Boolean constants:

```plaintext
module basic/BoolCon
exports
sorts BoolCon
context-free syntax
"true" -> BoolCon {cons("true")}
"false" -> BoolCon {cons("false")}

hiddens
context-free start-symbols
BoolCon
```

Notes:

1. The sort of Boolean constants is defined as BoolCon. Sort names should always start with a capital letter.
2. The constants true and false. Such literals should always be quoted. The cons attribute plays only a role when generating external APIs and gives a name to this particular construct. In the current presentation it can be ignored.
3. We add a start symbol (i.e., the syntactic notion from with all strings in this language are derived) for BoolCon. The net effect is that we can indeed parse these constants. Note that we hide this start symbol so that it can not proliferate to other modules (and cause undesired ambiguities).

**basic/Whitespace**

The module Whitespace defines what the spaces and newline are:
module basic/Whitespace
exports
  lexical syntax
  \s\t\n\r -> LAYOUT {cons("whitespace")}

context-free restrictions
LAYOUT? -/- [\s\t\n\r]

Notes:

1. A regular expression that defines space (\s), tabulation (\t), newline (\n) and carriage return (\r) as LAYOUT characters. LAYOUT is a predefined name that defines the layout characters that may optionally appear between the symbols in the context-free grammar (e.g., between the keyword if and the test in an if-statement).

2. Lexical syntax tends to become highly ambiguous, e.g., are two spaces one layout symbol or two consecutive ones? Context-free restrictions impose restrictions that resolve this. Here, optional layout (LAYOUT?) may not be followed by a layout character. In other words, the longest possible sequence of layout characters should be considered as one LAYOUT symbol.

basic/Comments

The comment conventions are defined as follows:

module basic/Comments
imports basic/Whitespace
exports
  lexical syntax
  "%%" line:-[\n]* "\n" -> LAYOUT
    {cons("line"),
     category("Comment")}  
  "%" content:-[\%\n]+ "%" -> LAYOUT
    {cons("nested"),
     category("Comment")}  

context-free restrictions
LAYOUT? -/- [\%]

Notes:

1. Defines a line-based comment that starts with \% and ends at the end of the line. The category attribute defines the highlighting category to be used while editing texts that contain this comment.

2. Define a comment that is contained within a single line between \% and \%.

3. Again a follow restriction that forces layout followed by a comment to be included in that comment.

basic/Booleans

After these preparations, we can now discuss the syntax part of basic/Booleans:

module basic/Booleans
imports basic/BoolCon
exports
sorts Boolean 2

context-free syntax
   -> Boolean
   {cons("constant")}

lhs:Boolean "|" rhs:Boolean -> Boolean
   {left, cons("or")}

lhs:Boolean "&" rhs:Boolean -> Boolean
   {left, cons("and")}

"not" "(" Boolean ")" -> Boolean
   {cons("not")}

"(" Boolean ")" -> Boolean
   {bracket, cons("bracket")}

context-free priorities
   Boolean "&" Boolean -> Boolean >
   Boolean "|" Boolean -> Boolean

hiddens
context-free start-symbols
   Boolean 8

imports basic/Comments 3

variables
   "Bool" [0-9]* -> Boolean 8

Notes:

1. Import the Boolean constants true and false defined earlier.
2. The sort Boolean will represent Boolean expressions.
3. Each Boolean constant is also a Boolean expression. This is called an injection rule or a chain rule.
4. The infix operators for Boolean or (|) and Boolean and (&). Both are left-associative and this is indicated by the attribute left. Also note that the arguments get explicit names (lhs and rhs) and that a constructor is defined (or, respectively, and). This information is only relevant for generated APIs that are accessed by external tools.
5. The prefix operator not.
6. The parentheses ( and ) may be used as brackets in Boolean expressions.
7. & has a higher priority than |. The expression Bool & Bool | Bool will thus be interpreted as (Bool & Bool) | Bool.
8. Define a (hidden) start symbol for Boolean expressions.
9. Import basic/Comments for the benefit of writing equations for Booleans (see below). Observe the subtlety that this is a hidden import: comments are only available locally in this module and are not exported. By strictly adhering to this convention, low-level comment conventions cannot pollute higher level modules and interfere with comment conventions defined at that level.
10. Declares Boolean variables like Bool, Bool1, Bool2, Bool123, Bool", Bool", and Bool1'.
Having covered all syntactic aspects of the Booleans, we can now turn our attention to the equations:

\[
\begin{align*}
[B1] & \quad \text{true} \mid \text{Bool} = \text{true} \quad \text{(1)} \\
[B2] & \quad \text{false} \mid \text{Bool} = \text{Bool} \\
[B3] & \quad \text{true} \& \text{Bool} = \text{Bool} \quad \text{(2)} \\
[B4] & \quad \text{false} \& \text{Bool} = \text{false} \\
[B5] & \quad \text{not (false)} = \text{true} \quad \text{(3)} \\
[B6] & \quad \text{not (true)} = \text{false}
\end{align*}
\]

Notes:

1. Meaning of the \( \mid \) operator, in other words: with these two rules the \( \mid \) operator can be eliminated from any Boolean expression.
3. Meaning of \text{not} operator.

The syntax of ASF+SDF equations is not fixed but depends on the syntax rules. This can be seen by making the fixed ASF+SDF syntax \textbf{bold} and the syntax specific for Booleans \textit{italic}:

\[
\begin{align*}
[B1] & \quad \textbf{true} \mid \text{Bool} = \text{true} \\
[B2] & \quad \textbf{false} \mid \text{Bool} = \text{Bool} \\
[B3] & \quad \textbf{true} \& \text{Bool} = \text{Bool} \\
[B4] & \quad \textbf{false} \& \text{Bool} = \text{false} \\
[B5] & \quad \textit{not (false)} = \text{true} \\
[B6] & \quad \textit{not (true)} = \text{false}
\end{align*}
\]

This mixture of syntaxes will become even more apparent when we discuss the Pico definitions later.

**Reducing a Boolean expression**

The Boolean term \( \text{not (true} \& \text{not (false} \mid \text{true}) \) should reduce to \text{true} (check this for yourself before looking at Figure 1.3, “Reducing a Boolean term.” (p. 9)).
Figure 1.3. Reducing a Boolean term.

```
not( true & not( false | true ) )
```

[B2] \( \text{false | Bool} = \text{Bool} \)

```
not( true & not( true ) )
```

[B6] \( \text{not (true)} = \text{false} \)

```
not( true & false )
```

[B3] \( \text{true & Bool} = \text{Bool} \)

```
not( false )
```

[B5] \( \text{not (false)} = \text{true} \)

Running the Boolean example in The Meta-Environment

The flash movie "Guided tour: Playing with Booleans" gives you a quick overview of the user-interface of The Meta-Environment. Here we give you only some quick hints to get started. After installation, the The Meta-Environment is available as the command: `asfsdf-meta`. Typing this command results in the screen shown in Figure 1.4, “Initial screen of The Meta-Environment” (p. 9).

Figure 1.4. Initial screen of The Meta-Environment
Opening the module `basic/Booleans` is achieved as follows:

- Press the menu Module and select the button Open....
- An Open Module dialog appears that asks for the directory to Look in.
- Select the entry ASF+SDF library and a list of choices appears.
- Select the subdirectory `basic` and select `Booleans.sdf` in the list of modules that is provided. The progress so far, is shown in Figure 1.5, “Selecting `basic/Booleans.sdf`” (p. 10).
- Now pushing the Open Module button will open `basic/Booleans` and all its imported modules. The result is shown in Figure 1.6, “After opening `basic/Booleans`” (p. 11).

**Figure 1.5. Selecting `basic/Booleans.sdf`**
Now that the Booleans have been opened, you may want to inspect their definition yourself. There are two ways to achieve this:

- Right click (= click with the right-most mouse button) on Booleans in the Modules pane on the left, or
- Right click on the oval labelled Booleans in the Import Graph pane on the right.

In both cases, a hierarchical pop-up menu will appear:

- Select the menu entry Editing.... A new pop-up menu will appear.
- Select the menu entry Syntax, and the syntax part of module Booleans will appear in an editor pane. The result is shown in Figure 1.7, “Opening the syntax of module Booleans” (p. 12).
After these initial steps, viewing the equations and opening and reducing a Boolean term will require similar interactions with The Meta-Environment.

**Important**

There is no substitute for trying this out yourself.

**Take home points**

The example of the Booleans illustrates the following points that are also valid for more complex examples:

- Each module *defines a language*: in this case the language of Booleans. In other contexts one can also speak about the datatype of the Booleans. We will use language and datatype as synonyms. Other examples of languages are integers, stacks, C, and Java. We treat them all in a uniform fashion.

- We can use the language definition for Booleans (and by implication for any language) to:
  - Create a syntax-directed editor for the Boolean language and create Boolean terms. In the context of, for instance, Java, it would be more common to say: create a syntax-directed editor for Java and create Java programs.
  - Apply the equations to this term and reduce it to a *normal form* (= a term that is not further reducible).
  - Import it in another module; this makes the Boolean language available for the importing module.
The toy language Pico

The toy language Pico has a single purpose in life: being so simple that specifications of every possible language aspect are so simple that they fit on a few pages. It can be summarized as follows:

- There are two types: natural numbers and strings.
- Variables have to be declared.
- Statements are assignment, if-then-else and while-do.
- Expressions may contain naturals, strings, variables, addition (+), subtraction (−) and concatenation (||).
- The operators + and − have operands of type natural and their result is natural.
- The operator || has operands of type string and its results is also of type string.
- Tests in if-then-else statement and while-statement should be of type natural.

Let's look at a simple Pico program that computes the factorial function:

```
begin declare input : natural, output : natural, repnr : natural, rep : natural;
input := 14;
output := 1;
while input - 1 do
rep := output;
repnr := input;
while repnr - 1 do
output := output + rep;
repnr := repnr - 1
od;
input := input - 1
od
end
```

Notes:

1. Pico programs do not have input/output statements, so we use variables for that purpose.
2. Pico has no multiplication operator so we have to simulate it with repeated addition (yes, simplicity comes at a price!).

Define the syntax for Pico

The import structure of the syntax definition of Pico is shown in Figure 1.8, “Import structure of Pico syntax” (p. 14). The modules basic/NatCon, basic/StrCon and basic/Whitespace are reused from the ASF+SDF library. The modules languages/pico/syntax/Identifiers, languages/pico/syntax/Types and languages/pico/syntax/Pico are specified for Pico and are now discussed in more detail.
Variables can be declared in Pico programs with one of two types: "natural number" or "string". This is defined as follows:

```plaintext
module languages/pico/syntax/Types
exports
  sorts TYPE
  context-free syntax
    "natural"  -> TYPE
    "string"   -> TYPE
    "nil-type" -> TYPE
```

Notes:

1. TYPE is the sort of possible types in Pico programs.
2. The constants natural and string represent types as they can be declared in a Pico program.
3. The constant nil-type is used for handling error cases.

Identifiers are used for the names of variables in Pico programs. They are defined as follows:

```plaintext
module languages/pico/syntax/Identifiers
exports
  sorts PICO-ID
  lexical syntax
    [a-z] [a-z0-9]* -> PICO-ID
  context-free restrictions
    PICO-ID -=- [a-z0-9]
```

Notes:

1. PICO-ID is the sort of identifiers in Pico programs.
PICO-ID is defined using a regular expression that contains the following elements:

- \([a-z]\): a character class that ranges over all lower case letters.
- \([a-z0-9]\): a character class that ranges over all lower case letters and over all digits.
- \([a-z0-9]\)*: zero or more repetitions of the character class \([a-z0-9]\). In other cases the postfix + operator can be used that defines one or more repetitions of the preceding construct.

The overall effect of this definition is that Pico identifiers start with a lower case letter that can be followed by lower case letters or by digits.

PICO-ID defines the longest match for Pico identifiers.

### languages/pico/syntax/Pico

After these preparations we can present the syntax for Pico:

```java
module languages/pico/syntax/Pico

imports languages/pico/syntax/Identifiers
imports languages/pico/syntax/Types
imports basic/NatCon
imports basic/StrCon

hiddens

  context-free start-symbols

    PROGRAM

exports

sorts PROGRAM DECLS ID-TYPE STATEMENT EXP

context-free syntax

"begin" DECLS {STATEMENT";" }* "end" -> PROGRAM

"declare" (ID-TYPE "," )*";" -> DECLS

PICO-ID ":=" TYPE -> ID-TYPE

context-free syntax

PICO-ID ":=" EXP -> STATEMENT

"if" EXP "then" {STATEMENT";" }* 
  "else" {STATEMENT";" }* 
"fi" -> STATEMENT

"while" EXP "do" 
  {STATEMENT ";" }* 
"od" -> STATEMENT

context-free syntax

PICO-ID -> EXP

NatCon -> EXP

StrCon -> EXP

EXP "+" EXP -> EXP (left)
```

1. context-free start-symbols
2. sorts PROGRAM DECLS ID-TYPE STATEMENT EXP
3. context-free syntax
4. "begin" DECLS {STATEMENT";" }* "end" -> PROGRAM
5. PICO-ID ":=" EXP -> STATEMENT
6. "if" EXP "then" {STATEMENT";" }* 
  "else" {STATEMENT";" }* 
"fi" -> STATEMENT
7. PICO-ID -> EXP
8. NatCon -> EXP
9. StrCon -> EXP
10. EXP "+" EXP -> EXP (left)
Writing Language Definitions in ASF+SDF

\[
\begin{align*}
\text{EXP "-" EXP} & \rightarrow \text{EXP (left)} \\
\text{EXP "|" EXP} & \rightarrow \text{EXP (left)} \\
"(" \text{EXP ")"} & \rightarrow \text{EXP (bracket)}
\end{align*}
\]

context-free priorities

\[
\begin{align*}
\text{EXP "|" EXP} & \rightarrow \text{EXP >} \\
\text{EXP "-" EXP} & \rightarrow \text{EXP >} \\
\text{EXP "+" EXP} & \rightarrow \text{EXP}
\end{align*}
\]

Notes:

1. PROGRAM is the start symbol for this grammar, i.e., each Pico program is derived from it.
2. The sorts PROGRAM, DECLS, ID-TYPE, STATEMENT and EXP are declared here and are used (in addition to the sorts declared in the imported modules) to define the Pico grammar.
3. This first context-free syntax section declares the top level structure of a Pico program.
4. The rule for PROGRAM contains the list construct \{(STATEMENT ; )\}*. It describes zero or more statements separated by a semicolon (;).
5. This section declares the syntax for statements.
6. This final context-free syntax section declares expression syntax.
7. These three rules define that Pico identifiers, natural constants and string constants may occur in expressions.
8. The syntax of the operators +, − and | | is defined. Observe that all three are left-associative. This implies that an expression like 1+2+3 is considered to be of the form (1+2)+3 and that the other interpretation 1+ (2+3) is rejected.
9. ( and ) can be used as brackets in expressions.
10. Priorities define the relative ordering of operators and are used to disambiguate text when more interpretations are possible. The higher the priority, the stronger the binding. The expression 1−2+3 will thus be interpreted as (1−2)+3.

Using the Pico syntax in the Meta-Environment

Now we will create a tiny Pico program:

- Right click on the module syntax/Pico (since we want to create a term using that syntax). A pop-up menu appears.

- Select the menu entry Editing.... A new pop-up menu will appear.

- Select the menu entry Term and a dialog window will appear to select the desired term. You may select an existing file or create a new one. We do the latter: type in pico-trial.trm. The result is a new editor pane labelled with the name of the file.

- Now start typing the following (syntactically correct) Pico program:

\[
\begin{align*}
\text{begin} \\
\text{declare} \\
\quad \text{x : natural}; \\
\quad \text{x := "abc"} \\
\text{end}
\end{align*}
\]

The result is shown in the Figure 1.9, “First program after typing it in.” (p. 17).
Figure 1.9. First program after typing it in.

Next, goto to the File menu and select the Save entry (or, alternatively, type Ctrl-S). The result is that the file is saved and parsed. As a result, all keywords will be highlighted as shown in Figure 1.10, “First program after saving it.” (p. 18).
Take home points

The syntax of Pico illustrates the following points:

- All modules for a syntax definition reside in a subdirectory named `syntax`.

- The main module of the syntax definition has the same name as the language (with an uppercase, since all module names start with an uppercase letter).

- The modules `languages/pico/syntax/Identifiers`, `languages/pico/syntax/Types` and `languages/pico/syntax/Pico` define (together with the modules they import) the syntax of the Pico language.

- This syntax can be used to:
  - Generate a parser that can parse Pico programs.
  - Generate a syntax-directed editor for Pico programs (including keyword highlighting).
  - Generate a parser that can parse equations containing fragments of Pico programs. This is similar to the use of different syntaxes in the definition of the `Booleans` described in the section called “basic/Booleans” (p. 6) and is used for program analysis and transformation.

Recap: symbols

In the syntax rules we have seen so far, many different symbols are used. Elementary symbols are:

- Literal strings like "begin" or "+".
• Sort names like PROGRAM and STATEMENT. Note that sorts are usually called *non-terminals*.

• Character classes like `[a-z]`. Character classes can be combined using the following operators:
  • `~`: complement.
  • `/`: difference.
  • `/\`: intersection.
  • `\/`: union.

Complex symbols are:

• Repetition:
  • `S*`: zero or more times `S`.
  • `S+`: one or more times `S`.
  • `{S1 S2}*`: zero or more times `S1` *separated by* `S2`.
  • `{S1 S2}+`: one or more times `S1` *separated by* `S2`.

Repetition is best understood by the following examples (assuming the rule "a" -> A):

• A accepts a.
• A+ accepts a, and aa, and ....
• `{A ";"}+` accepts a, and a;a, and a;a; and ... but not: a;a; a;.

Slightly different patterns can be defined as well (using grouping and alternative as defined below):

• `(A ";")+` accepts a, and a;a, and a;a; a, and ... but not a; a; a.
• `(A ";"?)+` accepts a, and a a, and a; a, and a; a; and ...

• Grouping: `(S1 S2 ...)` is identical to `S1 S2 ....`

• Optional: `S?`: zero or one occurrence of `S`.

• Alternative: `S | T`: an `S` or a `T`.

• Tuple: `<S, T>`: shorthand for "<" S "," T ">".

• Parameterized sort: `S[[ P1, P2 ]]`

**Recap: syntax rules**

The general form of a grammar rule is:

```
S1 S2 ... Sn -> S0 Attributes
```

Where `S1, ...` are symbols and `Attributes` is a (possibly empty) list of attributes. Attributes are used to define associativity (e.g. `left`), constructor functions (`cons`), and the like.

Lexical syntax and context-free syntax are similar but between the symbols defined in a context-free rule, optional layout symbols may occur in the input text. A context-free rule (like the one above) is equivalent to:

```
S1 LAYOUT? S2 LAYOUT? ... LAYOUT? Sn -> S0 Attributes
```
Define a typechecker for Pico

For most programming and application languages, it is not enough that programs are syntactically correct, i.e., that they strictly conform to the syntax rules of the language. In many cases, extra requirements are imposed on programs such as:

• Variables have to be declared before they can be used.

• The operands of operators have to be of certain types.

• Procedures can only be called with parameters that correspond in number and type with the formal parameters with the procedure has been declared.

These extra requirements are usually called type constraints, typechecking rules, or static semantics. We illustrate these requirements for the case of Pico.

The typechecking rules for the Pico language are very simple:

• The only types are natural and string.

• All variables should be declared before use.

• Left-hand side and right-hand side of an assignment statement should have equal type.

• The test in while-statement and if-statement should be natural.

• Operands of + and − should be natural; their result is also natural.

• Operands of || should be string; the result is also string.

The task of a typechecker for Pico is to assert that a given Pico program complies with the above rules. The typechecker can be seen as a transformation from a Pico program to an error report.

The import structure of the Pico typechecker is shown in Figure 1.11, “Import structure of Pico typechecker” (p. 20).

Figure 1.11. Import structure of Pico typechecker
languages/pico/check/Type-environments

The purpose of type environments is to maintain a mapping between identifiers and their type. This is done as follows:

```haskell
module languages/pico/check/Type-environments

imports languages/pico/syntax/Identifiers
imports containers/Table[PICO-ID TYPE]
exports
  sorts TENV TYPE
aliases
    Table[[PICO-ID,TYPE]] -> TENV

Notes:

1. Import the library module containers/Table. It is a parameterized module with first parameter denoting the sort of the keys in the table and the second parameter denoting the sort of the values to be stored in the table. After this import, the sort Table[[PICO-ID,TYPE]] (sorry about the double square brackets!) can be used as any other sort in the specification.

2. It is pragmatic to give a more descriptive name to Table[[PICO-ID,TYPE]] and we use the alias construct for this. From now on, TENV is an alias (= an abbreviation) for Table[[PICO-ID,TYPE]].

For convenience, we list the functions of Tables here:

```haskell
module containers/Table[Key Value]
...
context-free syntax
  "not-in-table"       -> Value  {constructor}
  "new-table"          -> Table[[Key,Value]]
lookup(Table[[Key,Value]], Key)  -> Value
store(Table[[Key,Value]], Key,   Value) -> Table[[Key,Value]]
delete(Table[[Key,Value]], Key)  -> Table[[Key,Value]]
element(Table[[Key,Value]], Key) -> Boolean
keys(Table[[Key,Value]])      -> List[[Key]]
values(Table[[Key,Value]])    -> List[[Value]]
```

In the case of Type-environments, the formal parameter Key is bound to PICO-ID and Value is bound to TYPE.

languages/pico/check/Pico

The central idea of the Pico typechecker is to visit all language constructs in a given Pico program while maintaining a type environment that maps identifiers to their declared type. Whenever an identifier is used, the type correctness of that use in the given context is checked against its declared type that is given by the type environment. An error message is generated when any violation of the type rules is detected. The following type checker is realistic in the following sense:

- It discovers all errors.
- It generates a message for each error.
- The error message contains the source code location of the Pico construct that violates the type rules.
- The type checker can be directly embedded in and used from The Meta-Environment.
Syntax part

First consider the syntax part of the Pico typechecker:

```haskell
module languages/pico/check/Pico

imports basic/Booleans

imports basic/Errors

imports languages/pico/syntax/Pico

imports languages/pico/check/Type-environments

imports utilities/PosInfo[EXP]

imports utilities/PosInfo[PICO-ID]

imports utilities/PosInfo[PROGRAM]

exports

context-free syntax

"tcp"(PROGRAM) -> {Error ","}*

hiddens

context-free syntax

"tcd"(DECLS) -> TENV

"tcits"({ID-TYPE ","}*, TENV) -> TENV

"tcit"(ID-TYPE, TENV) -> TENV

"tcs"({STATEMENT ";"}*, TENV) -> {Error ","}*

"tcst"(STATEMENT, TENV) -> {Error ","}*

"tce"(EXP, TYPE, TENV) -> {Error ","}*

context-free start-symbols

Summary PROGRAM {Error ","}*

variables

"Message" -> StrCon

"Error*" [0-9\']* -> {Error ","}*

"Decls" [0-9\']* -> DECLS

"Exp" [0-9\']* -> EXP

"Id" [0-9\']* -> PICO-ID

"Id-type*" [0-9]"* -> {ID-TYPE ","}*

"Nat-con*" [0-9]\'\]* -> NatCon

"Series" [0-9]\'\]* -> {STATEMENT ";"}+

"Stat" [0-9]\'\]* -> STATEMENT

"Stat*" [0-9]\'\]* -> {STATEMENT ";"}*

"Str-con" [0-9\']* -> StrCon

"Tenv" [0-9\']* -> TENV

"Type" [0-9\']* -> TYPE

"Program" [0-9\']* -> PROGRAM
```

Notes:

1. The library module `basic/Errors` describes an error format that is ubiquitous in The Meta-Environment. There are four categories of errors: info, warning, error and fatal. Each error contains a text that describes the error and a list of subjects where the error occurred. Each subject consists of a description and actual source code locations. All error messages detected by some tool are collected in a summary that contains an identifier for the tool, an identifier for this error summary and a list of errors. In the next subsection, various examples will be shown.

2. The library module `utilities/PosInfo` defines functions to get the source code location of any part of the input program.
tcp is the main type check function for complete Pico programs. Several auxiliary functions are defined that are used in the definition of tcp. Note how some of these functions take a TENV (type environment) as one of their arguments and return a possibly modified TENV as result. Other functions only use the TENV argument and directly produce error messages when necessary.

**Equations part**

Now let's turn our attention to the equations part of the Pico typechecker:

```plaintext
equations

[Main] start(PROGRAM, Program) =
    start(Summary,
        summary("pico-check",
            get-filename(get-location(Program)),
            [tcp(Program)]))

equations

[Tc1] tcp(begin Decls Series end) = tcs(Series, tcd(Decls))

equations

[Tc2] tcd(declare Id-type*;) = tcits(Id-type*, new-table)

equations

[Tc3a] tcits(Id:Type, Id-type*, Tenv) =
    tcits(Id-type*, tcit(Id:Type, Tenv))

[Tc3b] tcits(,Tenv) = Tenv

equations

[Tc4a] lookup(Tenv, Id) == not-in-table
    tcit(Id:Type, Tenv) = store(Tenv, Id, Type)

[default] tcit(Id:Type, Tenv) = Tenv

equations

[Tc5a] tcs(Stat ; Stat*, Tenv) =
    tcst(Stat,Tenv), tcs(Stat*,Tenv)

[Tc5b] tcs(,Tenv) =

equations

[Tc6a] not-in-table == lookup(Tenv, Id)
    tcst(Id := Exp, Tenv) =
```
error("Variable not declared",
    [localized("Id", get-location(Id))])

[default] tcst(Id := Exp, Tenv) =
  tce(Exp, lookup(Tenv, Id), Tenv)

[Tc6b] tcst(if Exp then Series1 else Series2 fi, Tenv) =
  tce(Exp, natural, Tenv),
  tcs(Series1, Tenv), tcs(Series2, Tenv)

[Tc6c] tcst(while Exp do Series od, Tenv) =
  tce(Exp, natural, Tenv), tcs(Series, Tenv)
equations

[default] tce(Exp, natural, Tenv) =
  error("Expression should be of type natural",
    [localized("Expression", get-location(Exp))])
[default] tce(Exp, string, Tenv) =
  error("Expression should be of type string",
    [localized("Expression", get-location(Exp))])

[Tc7a] tce(Id, Type, Tenv) =
  when Type == lookup(Tenv, Id)
[Tc7b] tce(Nat-con, natural, Tenv) =
[Tc7c] tce(Str-con, string, Tenv) =
[Tc7d] tce(Exp1 || Exp2, string, Tenv) =
  tce(Exp1, string, Tenv), tce(Exp2, string, Tenv)
[Tc7d] tce(Exp1 + Exp2, natural, Tenv) =
  tce(Exp1, natural, Tenv), tce(Exp2, natural, Tenv)
[Tc7d] tce(Exp1 - Exp2, natural, Tenv) =
  tce(Exp1, natural, Tenv), tce(Exp2, natural, Tenv)

Notes:

The first equation of this module is probably also the most intimidating one. Its role is similar
to the main function in C programs or the main method in Java programs; it is an entry
point for execution and indicates with which function execution starts. Here, we want to
establish a connection between the ordinary ASF+SDF functions defined in a module and
the calling context of The Meta-Environment. Typically, you are editing a Pico program
and on the push of a button you want the typecheck function to be applied to it. The left-
hand side start(PROGRAM, Program) defines that execution operates on something
of the sort PROGRAM. That "something" is given as value of the variable Program. The
right-hand side start(Summary, summary("pico-check",get-filename(get-
location(Program)), [tcp(Program)]) defines in a similar fashion that the result
sort of execution is a Summary and a recipe how to compute that result.

Type checking a complete program amounts to typechecking its declarations (this yields a type
environment) and then checking its statements in that environment.

To type check declarations create a new type environment (new-table) and visit all identifier-
type pairs in the declaration.

To check a list of identifier-type pairs, we have to visit each pair in the list. We use list matching
to achieve this: in the left-hand side tcits(Id:Type, Id-type*, Tenv) the first argument
of tcits (the list of identifier-type pairs) is decomposed into three values that are assigned to
variables. The first pair is decomposed into an identifier (Id) and a type (Type). The remainder
of the list is assigned to the variable Id-type* (note that we use the naming convention that
variables that have lists of zero or more elements as value end on * and, similarly, variables that have one or more elements as value end on +). In the right-hand side of this equation, the contribution of the first pair to the type-environment is computed (using tcit) and the remainder of the list of identifier-type pairs is checked recursively.

The left-hand side tcits(,Tenv) looks like a typo but it is not. An empty list of identifier-type pairs is really denoted by the empty text!

An identifier that does not yet occur in the type environment is stored in the type environment together with its type.

Again, list matching to decompose a list of statements into a first statement and a list of remaining statements. Next, check the first statement and the remaining statements.

Check an assignment statement and discover that the identifier on the left-hand side of the assignment is not declared. Return an error.

Otherwise, check the type of the expression on the right-hand side of the assignment. The declared type of the variable on the left-hand side and the derived type of the expression on the right-hand side should be the same. Note that the [default] label in front of this equation characterizes it as a default equation that is applied after all other equations have been tried. Default equations serve to catch all cases that are not handled by other equations.

Checking if-statements and while-statements amounts to checking that the test is of type natural.

Two default equations that generate an error message when the expected type and the actual type of an expression are unequal.

When the declared type of an identifier is equal to the expected type, we generate an empty list of errors.

Similarly, natural constants and string constants satisfy the expected types natural, respectively, string.

For operators, the operands are checked separately.

### Using the Pico typechecker in The Meta-Environment

Restart The Meta-Environment and do the following:

- Open the Module menu and select Open....

- In the dialog select ASF+SDF Library to look for modules.

- Select successively the directories languages, pico, check, and then finally the module Pico.sdf. This is the top level module of the Pico typechecker.

- After some processing the screen results shown in Figure 1.12, “After opening languages/pico/check/Pico” (p. 26).

The display of the import graph can be changed in various ways:

- Left click and drag on the display to move to other parts of the import graph.

- Right click on the display to adjust the size of the graph to let it fit on the display.
Now select languages/pico/check/Pico (in the Modules panel on the left or in the import graph) and open our previously created Pico program pico-trial.trm over it (by right clicking on it, following the pop-up menus, and opening pico-trial.trm). Click somewhere in pico-trial.trm and observe that several term editing menus have appeared. Of particular interest is the menu labelled Pico, see Figure 1.13, “Opening trial term over module languages/pico/check/Pico” (p. 27).

**Important**

The version of The meta-Environment produced to create these screen dumps gives two messages "constructor used more than one". Ignore them.
The final step is now to open the Pico menu and to select the check button. The effect is that the trial Pico program will be type checked and that an error message will be displayed. Recall that the typecheck function \( \text{tcp} \) is invoked with the current Pico program as argument and is then reduced. See Figure 1.14, “After pushing the check button in the Pico menu” (p. 28).
Figure 1.14. After pushing the check button in the Pico menu

The error message is more informative than it seems: unfolding it reveals the location of the error. Just click on the message to highlight the error in the source code. That is shown in Figure 1.15, “Click on the error message to highlight its source” (p. 29).
Writing Language Definitions in ASF+SDF

Figure 1.15. Click on the error message to highlight its source

Take home points

The Pico typechecker illustrates the following points:

• ASF+SDF can be used to define a type checker.

• ASF+SDF provides support for error messages and source code locations.

• All modules for a typechecker reside in a subdirectory named check.

• A typechecker can be integrated with The Meta-Environment. It is activated via the check button.

• Error messages are linked back to locations in the source text.

Defining an evaluator for Pico

In addition to syntax rules and typechecking rules, most languages also need evaluation rules that determine how a program is to be executed. Execution amounts to taking a program and all its input and computing the program's result by a step-by-step execution of the statements in the program.

The evaluation rules for the Pico language are simple:

• Variables of type natural are initialized to 0.

• Variables of type string are initialized to the empty string.

• A variable evaluates to its current value.

• The variable on the left-hand side of an assignment statement gets as value the value that results from evaluating the expression on the right-hand side of the assignment.
• If the test in an if-statement or while-statement evaluates to 0, this is interpreted as \textit{false}.

• Conversely, if the test in an if-statement or while-statement evaluates to a value unequal to 0, this is interpreted as \textit{true}.

• The statements in a list of statements are evaluated in sequential order.

The task of the Pico evaluator is to reduce a Pico program to the output it generates, in this case a value environment. The Pico evaluator can be seen as a transformation from a Pico program to its output.

The import structure of the Pico evaluator is shown in Figure 1.16, “Import structure of Pico evaluator” (p. 30).

\textbf{Figure 1.16. Import structure of Pico evaluator}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{import_structure.png}
\end{figure}

\texttt{languages/pico/run/Values}

The sort \texttt{VALUE} is simply a container for integer and string constants and is defined as follows:

\begin{verbatim}
module languages/pico/run/Values
imports basic/Integers basic/StrCon
exports sorts VALUE
context-free syntax
    Integer    -> VALUE 1
    StrCon     -> VALUE
    "nil-value" -> VALUE 2
\end{verbatim}
Notes:

1. It would be better to use here `IntCon` rather than `Integer`. This is a historic relict in the specification.
2. The constant `nil-value` denotes error values.

languages/pico/run/Value-environments

The purpose of value environments is to maintain a mapping between identifiers and their current value. This is done as follows:

```plaintext
module languages/pico/run/Value-environments
imports languages/pico/syntax/Identifiers
imports languages/pico/run/Values
imports containers/Table[PICO-ID VALUE]
exports
  sorts VENV
  aliases
    Table[[PICO-ID, VALUE]] -> VENV
```

After the discussion of Type-environments in the section called “languages/pico/check/Type-environments” (p. 21) this definition should be easy to follow.

languages/pico/run/Pico

The central idea of the Pico evaluator is to first visit the declarations and initialise the declared variables. Next, the statements are visited one-by-one and their effect on the value environment is computed. The final value environment is then returned as the result of evaluation.

Syntax part

The syntax part of the Pico evaluator looks as follows:

```plaintext
module languages/pico/run/Pico
imports languages/pico/syntax/Pico
imports languages/pico/run/Value-environments
imports basic/Strings
exports
  context-free syntax
    "evp"(PROGRAM)   -> VENV
  context-free syntax
    "evd"(DECLS)     -> VENV
    "evits"((ID-TYPE ",",)*) -> VENV
    "evs"({STATEMENT ";"*, VENV) -> VENV
    "evst"(STATEMENT, VENV) -> VENV
    "eve"(EXP, VENV)  -> VALUE

hiddens
  imports basic/Comments
  context-free start-symbols
    VALUE-ENV PROGRAM
```
Having seen the syntax part of the Pico typechecker there are no surprises here. The top level function `evp` maps programs to value environments and needs some auxiliary functions to achieve this.

**Equations part**

The equations part of the Pico evaluator:

```plaintext
equations

[Main] start(PROGRAM, Program) =
    start(VENV, evp(Program))  ①

equations

[Ev1]  evp(begin Decls Series end) =
    evs(Series, evd(Decls))

[Ev2]  evd(declare Id-type*;) = evits(Id-type*)

[Ev3a] evits(Id:natural, Id-type*) =
    store(evits(Id-type*), Id, 0)  ②

[Ev3b] evits(Id:string, Id-type*) =
    store(evits(Id-type*), Id, "")

[Ev3c] evits() = []

[Ev4a] Venv' := evst(Stat, Venv),
    Venv'' := evs(Stat*, Venv')
    ------------------------------------------- ③
evs(Stat ; Stat*, Venv) = Venv''

[Ev4b] evs( , Venv) = Venv

[Ev5a] evst(Id := Exp, Venv) =
    store(Venv, Id, eve(Exp, Venv))

[Ev5b]                    eve(Exp, Venv) != 0
    ------------------------------------------- ④
evst(if Exp then Series1 else Series2 fi, Venv) =
```
Writing Language Definitions in ASF+SDF

\[
evs(Series1, Venv)
\]

\[33\]

\[
[Ev5c] \quad \text{eve}(\text{Exp}, Venv) == 0
\]

\[
\text{evst}(\text{if Exp then Series1 else Series2 fi}, Venv) = \evs(Series2, Venv)
\]

\[5\]

\[
[Ev5d] \quad \text{eve}(\text{Exp}, Venv) == 0
\]

\[
\text{evst}(\text{while Exp do Series od}, Venv) = Venv
\]

\[6\]

\[
[Ev5e] \quad \text{eve}(\text{Exp}, Venv) != 0, \ Venv' := \evs(Series, Venv)
\]

\[
\text{evst}(\text{while Exp do Series od}, Venv) =
\]
\[
\text{evst}(\text{while Exp do Series od}, Venv) = \evst(\text{while Exp do Series od}, Venv')
\]

\[7\]

\[
[Ev6a] \quad \text{eve}(\text{Id}, Venv) = \text{lookup}(Venv, \text{Id})
\]

\[8\]

\[
[Ev6b] \quad \text{eve}(\text{Nat-con, Venv}) = \text{Nat-con}
\]

\[9\]

\[
[Ev6c] \quad \text{eve}(\text{Str-con, Venv}) = \text{Str-con}
\]

\[10\]

\[
[Ev6d] \quad \text{Nat1} := \text{eve}(\text{Exp1, Venv}),
\]
\[
\text{Nat2} := \text{eve}(\text{Exp2, Venv})
\]
\[
\text{eve}(\text{Exp1 + Exp2, Venv}) = \text{Nat1 + Nat2}
\]

\[11\]

\[
[Ev6e] \quad \text{Nat1} := \text{eve}(\text{Exp1, Venv}),
\]
\[
\text{Nat2} := \text{eve}(\text{Exp2, Venv})
\]
\[
\text{eve}(\text{Exp1 - Exp2, Venv}) = \text{Nat1} -/ \text{Nat2}
\]

\[12\]

\[
[Ev6f] \quad \text{Str1} := \text{eve}(\text{Exp1, Venv}),
\]
\[
\text{Str2} := \text{eve}(\text{Exp2, Venv}),
\]
\[
\text{Str3} := \text{concat}(\text{Str1, Str2})
\]
\[
\text{eve}(\text{Exp1 || Exp2, Venv}) = \text{Str3}
\]

\[13\]

\[
[\text{default-Ev6}] \quad \text{eve}(\text{Exp, Venv}) = \text{nil-value}
\]

Notes:

1. A start equation that connects the functions defined here to The Meta-Environment: given a program, a value environment is returned and this is computed by applying the function evp to the program.
2. Here declared variables are initialized.
3. Evaluation of a series of statements. Observe how the value environment Venv' that is modified by executing the first statement is used to evaluate the remaining statements. In this way, the assignments made by the first statement become available for the remaining statements.
4. Evaluation of if statement; the true case.
5. Evaluation of if statement; the false case.
6. Evaluation of while statement; the false case.
7. Evaluation of while statement; the true case. Note that the body of the while statement is evaluated and that the resulting value environment Venv' is used to evaluate the next iteration of the while statement.
A variable evaluates to its current value.

Constants evaluate to themselves.

Operators are evaluated by first evaluating their arguments and then applying the relevant operator to them. Observe that the + in the Pico expression Exp1 + Exp2 is the plus operator as defined by Pico. The + operator in Nat1 + Nat2 is the addition on integers as defined in basic/Integers. So we reuse the definition of addition on integers to define addition in Pico.

Now apply the evaluator function evp to our Pico factorial example:

```plaintext
evp({
    begin declare input : natural,
    output : natural,
    repnr: natural,
    rep: natural;
    input := 14;
    output := 1;
    while input - 1 do
        rep := output;
        repnr := input;
        while repnr - 1 do
            output := output + rep;
            repnr := repnr - 1
        od;
        input := input - 1
    od
    end
})
```

The result is:

```
[input,1>,
<repnr,1>,
<output,87178291200>,
<rep,43589145600>]
```

**Using the Pico evaluator in the Meta-Environment**

To evaluate a Pico program perform the following steps:

- Open the module `languages/pico/run/Pico.sdf` from the ASF+SDF Library.
- Open the Pico program `fac.trm` over this module.
- Open the Pico menu and to select the run button. The program is evaluated and the result is shown in a new editor pane called `run.out`. This may take a while!

The result is shown in Figure 1.17, “After running the factorial program.” (p. 35).

**Important**

Here again, more warnings appear. They can be ignored.
Take home points

The Pico evaluator has revealed the following points:

- ASF+SDF can be used to define an evaluator.
- All modules for an evaluator reside in a subdirectory named `run`.
- An evaluator can be integrated in The Meta-Environment. It is activated via the run button.

Defining a compiler for Pico

A compiler transforms a program in some higher level language (in this case Pico) to a lower level, in most cases assembly language. We will first define the assembly language and then define the transformation rules from Pico to assembly language. The overall import structure is shown in Figure 1.18, “Import structure Pico compiler” (p. 36)
languages/pico/compile/AssemblyLanguage

Now we define an assembly language for a stack-based CPU:

```plaintext
module languages/pico/compile/AssemblyLanguage

imports basic/Integers basic/Strings
imports languages/pico/syntax/Identifiers

exports
  sorts Label Instr
  lexical syntax
    [a-z0-9]+   -> Label
  context-free syntax
    "dclnat" PICO-ID  -> Instr
    "dclstr" PICO-ID  -> Instr
    "push" NatCon     -> Instr
    "push" StrCon     -> Instr
    "rvalue" PICO-ID  -> Instr
    "lvalue" PICO-ID  -> Instr
    "assign"          -> Instr
    "add"             -> Instr
    "sub"             -> Instr
    "conc"            -> Instr
    "label" Label     -> Instr
    "goto" Label      -> Instr
    "gotrue" Label    -> Instr
    "gofalse" Label   -> Instr
    "noop"            -> Instr

```

Figure 1.18. Import structure Pico compiler
sorts Instrs
context-free syntax
(Instr",")+ \rightarrow Instrs

Notes:
1. Define an instruction label.
2. Directives to allocate a variable of type natural or string.
3. Push a natural constant or a string constant on the stack.
4. Push the value of a variable on the stack.
5. Push the name of a variable on the stack.
6. Assign to a variable. The top entries on the stack are the value to be assigned and the name of the variable. Both are removed after executing this instruction.
7. The three operators for addition, subtraction and concatenation. They expect two values on the stack and replace them by the result of the operation.
8. Declare a label. This can be a target of one of the goto instructions.
9. Goto statements. The unconditional jump (goto) just continues execution at the instruction with the given label. The conditional jumps first inspect the top of the stack, pop it, and perform a jump depending on the value they found at the top of the stack.
A dummy instruction.
A list of instructions separated by semicolons.

languages/pico/compile/NextLabel

During code generation for if-statement and while-statement, the need will arise to generate new labels to describe the control flow that is implied by these statements. The function nextlabel defined below describes this.

The syntax part looks as follows:

module languages/pico/compile/NextLabel
imports languages/pico/compile/AssemblyLanguage
exports
context-free syntax
   "nextlabel" "(" Label ")" \rightarrow Label
hiddens
lexical variables
   "Char+" [0-9]* \rightarrow [a-z0-9]+

Notes:
1. The function nextlabel takes a Label as argument and creates a new, unique, label.
2. The definition uses a so-called lexical variable. Ordinary variables range over the subtrees (or subterms, if you prefer) of a term. In some cases it is necessary to be able to even inspect the contents of the lexical entities that form the leaves of a tree. Examples are identifiers, string constants and sometimes even the layout. Here we want to inspect the textual content of labels and define a lexical variable that ranges over the same lexical syntax as labels.

The equations part:
equations
This single equation decomposes a given label into a list of characters. Next, it creates a new label consisting of the original list of characters extended with a single character \( x \). Note that for every lexical sort (here: \( \text{Label} \)) there exits an automatically generated constructor function (here: \( \text{label} \)) that can be used to access the characters of a lexical value or to construct a new one. The former happens at the left-hand side of the above equation, the latter on the right-hand side. Also observe that this is completely type safe and that only syntactically correct lexical values can be constructed in this way.

**languages/pico/compile/Pico**

The goal of the Pico compiler is to translate a Pico program into an equivalent assembly language program.

**Syntax part**

The syntax part of the Pico compiler:

```plaintext
module languages/pico/compile/Pico

imports languages/pico/syntax/Pico
imports languages/pico/compile/AssemblyLanguage
imports languages/pico/compile/NextLabel

exports
context-free syntax
  trp( PROGRAM ) -> Instrs

hiddens
context-free start-symbols
  PROGRAM Instrs

context-free syntax
  trd(DECLS) -> {Instr ;"}+
  trits({ID-TYPE ","}*) -> {Instr ;"}+
  trs({STATEMENT ;"}*, Label) -> <{Instr ;"}+, Label>
  trstat(STATEMENT, Label) -> <{Instr ;"}+, Label>
  tre(EXP) -> {Instr ;"}+

hiddens
variables
  "Decls"[0-9\']* -> DECLS
  "Exp"[0-9\']* -> EXP
  "Id"[0-9]* -> PICO-ID
  "Id-type*"[0-9]* -> {ID-TYPE ","}*
  "Nat-con"[0-9\']* -> NatCon
  "Series"[0-9\']* -> {STATEMENT ;"}+
  "Stat"[0-9\']* -> STATEMENT
  "Stat*"[0-9\']* -> {STATEMENT ;"}*
  "Str-con"[0-9\']* -> StrCon
  "Str"[0-9\']* -> String
  "Instr*"[0-9\']* -> {Instr ;"}+
  "Label*"[0-9\']* -> Label
  "Program" -> PROGRAM
```
Notes:

1. The main compiler function translates a Pico program into a sequence of instructions.
2. Auxiliary compiler functions. Observe that some of these functions have the output sort <(Instr ";"), Label>. This a tuple with as first component a list of instructions and as second components a label. The general idea is that those functions that generate new labels (for instance, to compile an if-statement) return an instruction sequence and the last label they have generated.

Equations part

The equations of the Pico compiler look as follows:

```plaintext
equations
[main] start(PROGRAM, Program) =
    start(Instrs, trp(Program))

equations
[Tr1] Instr*1 := trd(Decls),
     <Instr*2, Label> := trs(Series, x)
    trp(begin Decls Series end) = Instr*1; Instr*2

[Tr2] trd(declare Id-type*;) = trits(Id-type*)

[Tr3a] trits(ID:natural, Id-type*) =
     dclnat Id;
     trits(Id-type*)

[Tr3b] trits(ID:string, Id-type*) =
     dclstr Id;
     trits(Id-type*)

[Tr3c] trits() = noop

[Tr4a] <Instr*1, Label'> := trst(Stat, Label),
     <Instr*2, Label''> := trs(Stat*, Label')
    trs(Stat ; Stat*, Label) =
     < Instr*1;
     Instr*2
     ,
     Label'' >

[Tr4b] trs(, Label) = <noop, Label>

[Tr5a] Instr* := tre(Exp)
    trst(Id := Exp, Label) =
     < lvalue Id;
     Instr*;
     assign
     ,
```
Label >

[Tr5b] Instr* := tre(Exp),
<Instr*1, Label'> := trs(Series1, Label),
<Instr*2, Label''> := trs(Series2, Label'),
Label1 := nextlabel(Label''),
Label2 := nextlabel(Label1)
===================================================
trst(if Exp then Series1 else Series2 fi, Label) =
< Instr*;
gofalse Label1;
Instr*1;
goto Label2;
label Label1;
Instr*2;
label Label2
,
Label2 >

[Tr5c] Instr*1 := tre(Exp),
<Instr*2, Label'> := trs(Series, Label),
Label1 := nextlabel(Label'),
Label2 := nextlabel(Label1)
=======================================
trst(while Exp do Series od, Label) =
< label Label1;
Instr*1;
gofalse Label2;
Instr*2;
goto Label1;
label Label2
,
Label2 >

[Tr6a] tre(Nat-con) = push Nat-con

[Tr6b] tre(Str-con) = push Str-con

[Tr6c] tre(Id) = rvalue Id

[Tr6d] Instr*1 := tre(Exp1), Instr*2 := tre(Exp2)
===================================================
tre(Exp1 + Exp2) = Instr*1; Instr*2; add

[Tr6e] Instr*1 := tre(Exp1), Instr*2 := tre(Exp2)
==========================================
tre(Exp1 - Exp2) = Instr*1; Instr*2; sub

[Tr6f] Instr*1 := tre(Exp1), Instr*2 := tre(Exp2)
==========================================
tre(Exp1 || Exp2) = Instr*1; Instr*2; conc

Notes:

1. The ubiquitous start equation that establishes that trees of sort PROGRAM are being transformed into a result of sort Instrs and this result is computed by the function trp.
A Pico program is compiled by first compiling its declarations and then its instructions. The resulting instruction sequences are then concatenated and returned as result. Note that the translation of the series part of the program starts using the label $x$ (the single $x$ character).

Compile the declarations.

Compile an (id, type) pair. Depending on the declared type, we generate a `dclnat` or `dclstr` instruction.

Compile a series of statements by first compiling the first one and then the rest. Observe the careful propagation of the label information.

An empty series of statements is translated into a `noop` instruction.

Compile an assignment statement.

Compile an if-statement. Labels are created to mark the instructions generated for the else branch and, respectively, the instructions following the code generated for the if statement.

Compile a while-statement. Two new label are generated to mark the instructions generated for the while statement and, respectively, the instructions following the code generated for the while loop.

Constants are compiled into appropriate push instructions.

Compilation of expressions. First compile both arguments (when the generated code is executed it leaves the two argument values on the stack) and then generate the appropriate operator instruction that will replace the two values on the stack by the result of the operator.

Recall the Pico example that computes factorial and let's apply the function `trp` to it:

```
trp(
begin declare input : natural,
     output : natural,
     repnr: natural,
     rep: natural;
     input := 14;
     output := 1;
     while input - 1 do
       rep := output;
       repnr := input;
       while repnr - 1 do
         output := output + rep;
         repnr := repnr - 1
       od;
     input := input - 1
   od
end
)
```

The result is the following assembly language program:

```
dclnat input;
dclnat output;
dclnat repnr;
dclnat rep;
noop; lvalue input;
push 14;
assign ;
lvalue output;
push 1;
assign ;
label xxxx;
rvalue input; push 1; sub;
gofalse xxxx;
lvalue rep;
```
Using the Pico compiler in the Meta-Environment

TBD

Caution

compile does not generate a Pico menu entry but we need to use Reduce instead.

Take home points

The Pico compiler illustrates the following issues:

• ASF+SDF can be used to define a compiler.

• ASF+SDF allows the decomposition of lexical values into characters and the construction of syntactically correct new lexical values.

• Three languages are involved in the definition of the Pico compiler: ASF+SDF (the specification language), Pico (source language), and AssemblyLanguage (target language).

• The compiler can be integrated in The Meta-Environment and is activated via the compile button.

Intermezzo: traversal functions

As we have seen in the preceding examples, functions like typechecking, evaluation and compilation recursively visit all nodes in the parse tree of a program and perform their task at each node, being checking, evaluation or code generation. In these cases it is unavoidable that we need to define an equation for each possible language construct that has to be processed. The number of equations will then be at least equal to the number of grammar rules in the syntax of the language. In the case of real languages, like Java or COBOL, this amounts to hundreds and hundreds of syntax rules and by implication of hundreds of equations as well.
There are, fortunately, also many other applications where something interesting has to be done at only a few nodes. Examples are:

- Computing program metrics like counting the number of identifiers, goto statements, McCabe complexity and the like.
- Extracting function calls.

ASF+SDF provides traversal functions that automate these cases. There are two important aspects of traversal functions:

- The kind of traversal:
  - accumulate a value during the traversal, this is indicated by: $\text{traversal(accum)}$. A typical example is a function that counts statements.
  - transform the tree during the traversal, this is indicated by: $\text{traversal(trafo)}$. A typical example is a function that replaces certain statements.
  - accumulate and transform at the same time, this is indicated by: $\text{traversal(accum,trafo)}$.

- The order of traversal:
  - top-down versus bottom-up: In Figure 1.19, “Top-down versus bottom-up” (p. 43) the difference is shown: a top-down traversal starts with the root node and then traverses the root's children from the left to right, and so on. A bottom-up traversal, however, starts with the left-most leave of the tree and works its way up in the tree towards the root which is visited last.
  - break (stop at the first node where something can be done) or continue after visiting a node.

**Figure 1.19. Top-down versus bottom-up**

![Top-down versus bottom-up diagram]

**Simple trees**

Consider the following very simple language of trees:

```plaintext
module Tree-syntax
imports Naturals
exports
sorts TREE
context-free syntax
NAT -> TREE
```
Writing Language Definitions in ASF+SDF

\[
\begin{align*}
\text{f(TREE, TREE)} & \rightarrow \text{TREE} \\
\text{g(TREE, TREE)} & \rightarrow \text{TREE} \\
\text{h(TREE, TREE)} & \rightarrow \text{TREE}
\end{align*}
\]

variables

- \text{“N”[0-9]*} \rightarrow \text{NAT}
- \text{“T”[0-9]*} \rightarrow \text{TREE}

Notes:

1. The leaves of a tree are natural numbers.
2. The symbols \( f \), \( g \) and \( h \) can be used to construct composite trees.
3. The variables \( N \) and \( T \) (both possibly followed by digits) are defined for the convenient use in the following examples.

An example of a tree defined in this way is \( f(g(1,2),3) \). Graphically, it looks as in Figure 1.20, “Example \( f(g(1,2),3) \) as tree” (p. 44).

**Figure 1.20. Example \( f(g(1,2),3) \) as tree**

![Tree diagram](image)

**Counting tree nodes (classical solution)**

The classical solution for counting the nodes in a tree is as follows:

```plaintext
module Tree-cnt
imports Tree-syntax
exports
context-free syntax

\[
\begin{align*}
\text{cnt(TREE)} & \rightarrow \text{NAT} \quad \text{1}
\end{align*}
\]

equations

\[
\begin{align*}
\text{1} & \quad \text{cnt}(N) = 1 \quad \text{2} \\
\text{2} & \quad \text{cnt}(f(T_1,T_2)) = 1+\text{cnt}(T_1)+\text{cnt}(T_2) \quad \text{3} \\
\text{3} & \quad \text{cnt}(g(T_1,T_2)) = 1+\text{cnt}(T_1)+\text{cnt}(T_2) \\
\text{4} & \quad \text{cnt}(h(T_1,T_2)) = 1+\text{cnt}(T_1)+\text{cnt}(T_2)
\end{align*}
\]

Notes:

1. The function \( \text{cnt} \) visits all nodes in a tree and returns the number of nodes.
2. Visiting a leaf (a number) counts for one.
3. Visiting a non-leaf counts for one (this node) plus the counts for the left and the right subtree. An equation is needed for every constructor in the tree language.

Counting the nodes of our previous example is achieved as follows:
The steps to arrive at this result are shown in Figure 1.21, “Reducing \( cnt(f(g(1, 2), 3)) \)” (p. 45).

**Figure 1.21. Reducing \( cnt(f(g(1, 2), 3)) \)**

![Tree diagram showing the reduction process]

**Counting tree nodes (using an accumulator)**

Let’s now reformulate the node counting example using traversal functions:

```plaintext
module Tree-cnt
imports Tree-syntax
exports
  context-free syntax
    cnt(TREE, NAT) -> NAT {traversal(accu, 1, bottom-up, continue)}

equations
[1] cnt(T, N) = N + 1
```

Notes:

1. Define `cnt` as a bottom-up accumulator that continues after each matching node. The second argument is the accumulator used for counting. Observe that in this example, it is arbitrary whether we do a bottom-up or a top-down traversal.

2. A single equation is needed to count any node (both leaves and complex nodes). The first argument `T` matches any tree, and the accumulator value `N` is incremented for it. Given this incremented value, the following nodes of the tree will be visited and counted.

Counting the nodes of the example
Writing Language Definitions in ASF+SDF

\[
\text{cnt} \left( f \left( g \left( f(1,2), 3 \right), \\
g \left( g(4,5), 6 \right) \right), 0 \right)
\]
gives:

11

**Increment and modify tree leaves (using a transformer)**

Now we switch to a simple transformation example: given a tree construct a new one that is identical except that all values at the leaves are incremented by one. This is achieved as follows:

```plaintext
module Tree-inc
imports Tree-syntax
exports
context-free syntax
  inc(TREE) -> TREE {traversal(trafo, bottom-up, continue)}

equations
[1] inc(N) = N + 1
```

Notes:

1. The transformer \textit{inc} will compute the desired tree. Observe the type of this function: it transforms trees to trees.
2. The single equation that performs the replacement. Observe that this equation only matches at nodes that contain an integer (this is due to the \textit{N} argument on the left-hand side). When the equation is applicable it replaces the given integer leaf by an incremented one.

The example:

```
inc( f( g( f(1,2), 3 ), \\
g( g(4,5), 6 )))
```
gives:

```
f( g( f(2,3), 4 ), \\
g( g(5,6), 7 ))
```

**Increment with variable amount and modify tree leaves (using transformer)**

Continuing the incrementing example, we now want to make the \textit{amount} of the increment variable.

```plaintext
module Tree-incp
imports Tree-syntax
exports
context-free syntax
  inc(TREE, NAT) -> TREE {traversal(trafo, bottom-up, continue)}

equations
[1] inc(N, 1) = N + 1
```

Notes:

1. The transformer \textit{inc} will compute the desired tree. Observe the type of this function: it transforms trees to trees.
2. The single equation that performs the replacement. Observe that this equation only matches at nodes that contain an integer (this is due to the \textit{N} argument on the left-hand side). When the equation is applicable it replaces the given integer leaf by an incremented one.

The example:

```
inc( f( g( f(1,2), 3 ), \\
g( g(4,5), 6 )))
```
gives:

```
f( g( f(2,3), 4 ), \\
g( g(5,6), 7 ))
```
We define a new \textit{inc} function with a second argument that represents the amount of the increment.

The equation has to be adjusted as well. \(N_1\) matches an node (as before) and \(N_2\) represents the desired increment. Application of this equation replaces the current leaf by its value increment with \(N_2\).

Example:

\[
\text{inc}( f( g( f(1,2), 3 ),
      g( g(4,5), 6 )),
      7 )
\]

Result:

\[
f( g( f( 8, 9), 10),
     g( g(11,12), 13))
\]

**Tree replacement**

Performing replacements in trees is a frequently occurring operation. Typically, certain values in a tree have to be replaced by other ones. These values can be simple or complex. One can distinguish different forms of replacement:

- **Deep replacement** replaces only occurrences close to the leaves of the tree.
- **Shallow replacement** replaces only occurrences close to the root.
- **Full replacement** replaces all occurrences in a tree.

These forms of replacement form a nice test case for traversal functions and we will discuss each in detail. The task at hand will be: perform a deep/shallow/full replacement of all function symbols \(g\) by a new function symbol \(i\).

**Deep replacement**

Deep replacement only replaces occurrences close to the leaves of the tree. This suggests a bottom-up approach where on each path from the root of the tree to each leaf, at most one replacement may occur. This is exactly what the break attribute of traversal function achieves.

```
module Tree-drepl
imports Tree-syntax
exports context-free syntax
   i(TREE, TREE)   -> TREE 1
   drepl(TREE)      -> TREE (traversal(trafo,
                                bottom-up,
                                break)) 2
equations
[1] drepl(g(T1, T2)) = i(T1, T2) 3
```

Notes:

1. Add the additional tree constructor \(i\) to make examples slightly more interesting.
The function `drepl` is a bottom-up transformer that stops at the first match on each path from root to leaf.

The equation that performs the actual replacement: the subtree \( g(T_1, T_2) \) is replaced by \( i(T_1, T_2) \).

An example:

```plaintext
drepl( f( g( f(1,2), 3 ) ,
       g( g(4,5), 6 ) ) )
```

results in:

```plaintext
f( i( f(1,2), 3 ),
   g( i(4,5), 6 ) )
```

Indeed, only the deepest occurrences of \( g \) have been replaced.

**Shallow replacement**

Shallow replacement only replaces occurrences close to the root of the tree. This suggests a top-down approach where on each path from the root of the tree to each leaf, at most one replacement may occur. This is exactly what the `break` attribute of traversal function achieves.

```plaintext
module Tree-srepl
imports Tree-syntax
exports
context-free syntax
i(TREE, TREE)   -> TREE
srepl(TREE)      -> TREE {traversal(trafo,
                              top-down,
                              break)}
equations
[1] srepl(g(T1, T2)) = i(T1, T2)
```

Notes:

1. The function `srepl` is a top-down transformer that stops at the first match on each path from root to leaf.

The example:

```plaintext
srepl( f( g( f(1,2), 3 ) ,
       g( g(4,5), 6 ) ) )
```

results in:

```plaintext
f( i( f(1,2), 3 ),
   i( g(4,5), 6 ) )
```

As expected, only the outermost occurrences of \( g \) have been replaced.

**Full replacement**

Full replacement replaces all occurrences in the tree. In this case, there is no difference between a top-down or a bottom-up approach. The `continue` attribute of traversal function achieves traversal of all tree nodes.

```plaintext
module Tree-frepl
```
imports Tree-syntax
exports
category-free syntax
i( TREE, TREE ) \rightarrow TREE
frepl( TREE ) \rightarrow TREE \{ traversal(\text{trafo}, \top-down, \text{continue}) \}
equations
[1] frepl( g( T1, T2 )) = i( T1, T2 )

The example

frepl( f( g( f(1,2), 3 ),
g( g(4,5), 6 ) ) )

results in:

f( i( f(1,2), 3 ),
i( i(4,5), 6 ) )

Indeed, all occurrences of \textit{g} have now been replaced.

**A real example: COBOL transformation**

Now let’s turn our attention to a real example of traversal functions. The problem is that COBOL75 has two forms of if-statement:

- "IF" Expr "THEN" Stats "END-IF"?
- "IF" Expr "THEN" Stats "ELSE" Stats "END-IF"?

In other words, COBOL75 has an if-then statement and an if-then-else statement and both end on an optional keyword \textit{END-IF}. This leads to the well known dangling else problem that may cause ambiguities in nested if statements. Given

\textbf{IF} expr1 \textbf{THEN} IF expr2 \textbf{THEN} stats1 ELSE stats2

Is this an if-then statement with an if-then-else statement as then branch as suggested by the following layout:

\textbf{IF} expr1 \textbf{THEN}
\textbf{IF} expr2 \textbf{THEN}
\hspace{1em} stats1
\hspace{1em} ELSE
\hspace{1em} stats2

Or is it the other way around, an if-then-else statement with an if-then statement as then branch:

\textbf{IF} expr1 \textbf{THEN}
\textbf{IF} expr2 \textbf{THEN}
\hspace{1em} stats1
\hspace{1em} ELSE
\hspace{1em} stats2

This is utterly confusing and error-prone, therefore it is a good idea to add \textit{END-IF} to all if-statement to indicate that either

\textbf{IF} expr1 \textbf{THEN}
\textbf{IF} expr2 \textbf{THEN}
Writing Language Definitions in ASF+SDF

The following excerpt from a real application achieves this:

```plaintext
module End-If-Trafo
imports Cobol
exports
context-free syntax
  addEndIf(Program) -> Program {traversal(trafo, continue, top-down)}
variables
  "Stats"[0-9]*      -> StatsOptIfNotClosed
  "Expr"[0-9]*       -> L-exp
  "OptThen"[0-9]*    -> OptThen
equations
[1] addEndIf(IF Expr OptThen Stats) = IF Expr OptThen Stats END-IF
[2] addEndIf(IF Expr OptThen Stats1 ELSE Stats2) = IF Expr OptThen Stats1 ELSE Stats2 END-IF

Notes:
[1] Import the COBOL grammar. It is huge and consists of hundreds of syntax rules.
[2] The transformer addEndIf will perform the desired transformation.
[3] Two equations define the transformation for an if-then, respectively, an if-then-else statement
    with missing END-IF keyword.

It must be stressed that, due to the nesting of if-statements, such a kind of transformation can never be
done by a tool that is based on lexical analysis alone. The enormous savings due to the use of traversal
functions should also be stressed: only two equations suffice to define the required transformation.
Compare this with the hundreds of equations that would have been needed without traversal functions.

A funny Pico typechecker

To conclude these illustrations of the use of traversal functions, we reconsider the problem of Pico
typechecking and present a completely alternative solution that does not use type environments. Given
a Pico program, the overall approach is to remove those part that are type correct and to repeat this
process as long as possible. A type correct program will hence be reduced to an empty program, while
a type-incorrect program will be reduced to a program that precisely contains the incorrect statements. The replacement strategy is as follows:

- Replace all constants by their type, e.g., 3 becomes \texttt{type(natural)}.
- Replace all variables by their declared type, e.g., \texttt{x + 3} becomes \texttt{type(natural) + type(natural)}.
- Simplify type correct expressions, e.g., \texttt{type(natural) + type(natural)} becomes \texttt{type(natural)}.
- Remove all type correct statements, e.g., \texttt{type(natural) := type(natural)} is removed.

Consider the erroneous program:

\begin{verbatim}
begin
  declare x : natural,
   y : natural,
   s : string;
  x := 10; s := "abc";
  if x then
    x := x + 1
  else
    s := x + 2
  fi;
  y := x + 2;
end
\end{verbatim}

(take a second to spot the error) will thus reduce to:

\begin{verbatim}
begin
  declare;
  type(string) := type(natural);
end
\end{verbatim}

This style of type checking leads to descriptive messages concerning the cause of errors and is an alternative to the style where explicit error locations have to be taken into account in the specification.

The "funny" Pico typechecker is defined as follows:

\begin{verbatim}
module Pico-typecheck
imports Pico-syntax
exports context-free syntax
  type(TYPE)                  -> ID
  replace(STATS, ID-TYPE)    -> STATS {traversal(trafo,
                                        bottom-up,
                                        break)}
  replace(EXP     , ID-TYPE)  -> EXP   {traversal(trafo,
                                        bottom-up,
                                        break)}

variables ...

equations
[0] begin declare Id-type, Decl*; Stat* end =
    begin declare Decl*; replace(Stat*, Id-type) end
[1] replace(Id     , Id : Type) = type(Type)
\end{verbatim}
Writing Language Definitions in ASF+SDF

[2] replace(Nat-con, Id : Type) = type(natural)
[3] replace(Str-con, Id : Type) = type(string)

[4] type(string) || type(string)  = type(string)
[5] type(natural) + type(natural) = type(natural)
[6] type(natural) - type(natural) = type(natural)

[7] Stat*1; if type(natural) then Stat*3 fi ; Stat*4
   = Stat*1; Stat*2; Stat*3; Stat*4

[8] Stat*1; while type(natural) do Stat*2 od; Stat*3
   = Stat*1; Stat*2; Stat*3

[9] Stat*1; type(Type) := type(Type); Stat*2
   = Stat*1; Stat*2

Notes:

First we have to extend the Pico syntax, to allow type indications in the code. Here, we allow at
every position where an identifier may occur a type indication of the form type(natural)
or type(string).

The traversal function replace will replace a specific identifier (as defined by its second
argument) in a Pico program. Since programs are composed of subtrees of various sorts (e.g.,
PROGRAM, DECL, STAT, EXP, ...) the first argument of replace may range over all these sorts.
Each variant that is used in the equations has to be declared here.

Visit each variable declaration and use replace to replace the variable by its type in the body
of the program.

Replace constants and variables by their type.

Replace type-correct expressions by their type.

Remove type-correct expressions and statements.

Take home points

This intermezzo introduces the following points:

• For tree traversals that have to do work at every node, it is unavoidable to write many equations to
describe the traversal: at least one equation is needed for every rule in the grammar of the language
in question. Examples are typechecking, evaluation and code generation.

• Other applications only have to do work at a few types of nodes. This is exemplified by various
forms of metrics calculation and fact extraction.

• Traversal functions automate such applications: only equations are needed for the cases where real
work has to be done at a node; the other cases are handled automatically.

• An accumulator is a traversal function that computes a certain value during the traversal of the tree.

• A transformer is a traversal function that transform the tree during the traversal.

• An accumulating transformer combines these functionalities.

• Traversal functions can traverse the tree from root to leaves (top-down) or from leaves to root
(bottom-up).

• Traversal functions can visit all nodes on each path from root to tree (continuous) or only the first
matching node (break).
Define a fact extractor for Pico

The goal of a fact extractor is to extract information from source files that can be used by later analysis tools. Fact extraction is tightly coupled to the requirements of the analysis tools and can differ widely per application. Here we show a simple fact extractor that extracts:

- A control flow graph.
- A statement histogram.

The import structure is shown in Figure 1.22, “Import structure Pico fact extractor” (p. 53).

**Figure 1.22. Import structure Pico fact extractor**

Two imported modules require explanation, but we do not go into too much details.

The module utilities/Rstores defines rstores, relation stores, that are used to maintain mapping between a typed variable and a set or relation. Rstores are used to collect the extracted facts and can be used later on by analysis tools. Typical operations are:

- **create-store**: create a new rstore.
- **declare**: define a new variable with its type and initial value.
- **set**: assign a value to a variable in the rstore.

The module utilities/Parsing defines parsing and unparsing functions for values of specific sorts. The only function that we will use is:

- **unparse-to-string**: convert an arbitrary value (like an identifier, expression, or statement in a Pico program) to a string value. This conversion is necessary since sets and relations can only contain basic values like integers and strings and no complex values like fragments from parse trees.

**languages/pico/extract/Pico**

The syntax definition of the Pico fact extractor:

```plaintext
module languages/pico/extract/Pico
imports utilities/RStores
imports languages/pico/syntax/Pico
imports basic/Integers
imports utilities/Parsing[PICO-ID]
imports utilities/Parsing[STATEMENT]
```
Writing Language Definitions in ASF+SDF

imports utilities/Parsing[EXP]
imports utilities/PosInfo[STATEMENT]
imports utilities/PosInfo[EXP]

hiddens
context-free syntax
  controlFlow(PROGRAM, RStore)        -> RStore
  statementHistogram(PROGRAM, RStore) -> RStore

context-free syntax
  countStatements(PROGRAM, RStore)    -> RStore
  countStatements(STATEMENT, RStore)  -> RStore
  cflow((STATEMENT ";"*)                -> <RElem,RElem,RElem>

context-free start-symbols
PROGRAM RStore RElem

variables
  "Program" [0-9]* -> PROGRAM
  "Decls" [0-9]*   -> DECLS
  "Stat" [0-9]*    -> STATEMENT
  "Stat*" [0-9]*   -> (STATEMENT ";"*)
  "Stat+" [0-9]*   -> (STATEMENT ";"+)
  "Exp" [0-9]*     -> EXP
  "Id" [0-9]*      -> PICO-ID
  "Entry" [0-9]*   -> RElem
  "Exit" [0-9]*    -> RElem
  "Rel" [0-9]*     -> RElem
  "Control" [0-9]* -> RElem

variables
  "Store"[0-9]*    -> RStore {strict}
  "Int" [0-9]*     -> Integer {strict}

Notes:

1. The modules utilities/Parsing and utilities/PosInfo are instantiated for all relevant sorts.
2. Given a Pico program and an rstore, the function controlFlow adds a controlflow relation to the given rstore.
3. Similarly, function statementHistogram adds a histogram relation to the rstore.
4. These two functions are implemented using auxiliary traversal functions.
5. Variables can be annotated with the attributes wild and strict. They do not change the meaning of variables but help in enforcing their correct usage:
   - Wild variables may only be used once in an equation to receive a value (as a result of matching or evaluating a condition) but that value may not be user later on in the equation. Wild variables serve to match an arbitrary part of the input that is not re-used later. Typical error case: using the same variable to match two arbitrary parts of the input term and by doing so erroneously enforcing that both arbitrary parts are identical.
• Strict variables receive a value and that value may be used precisely once. Typical error case: using some intermediate variables to represent a global state and incorrectly passing the latest version of the global state to another function.

The equations for the Pico fact extractor:

```plaintext
equations

[main] Store1 := create-store(),
    Store2 := statementHistogram(Program, Store1),
    Store3 := controlFlow(Program, Store2)

start(PROGRAM, Program) = start(RStore, Store3)

equations

[hist] statementHistogram(Program, Store) = countStatements(Program,
            declare(Store,
                StatementHistogram,
                rel[str,int]))

equations

[cS1] countStatements(Id := Exp, Store) = inc(Store, StatementHistogram, "Assignment")

[cS2] countStatements(if Exp
    then Stat*1
    else Stat*2 fi,
    Store) =
    inc(Store, StatementHistogram, "Conditional")

[cS3] countStatements(while Exp do Stat* od, Store) =
    inc(Store, StatementHistogram, "Loop")

equations

[cfg] Store1 := declare(Store,
            ControlFlow,
            rel[<str,loc>,<str,loc>]),
            <Entry, Rel, Exit> := cflow(Stat*)

controlFlow(begin Decls Stat* end, Store) =
    set(Store1, ControlFlow, Rel)

equations

[cfg-1]
    <Entry1, Rel1, Exit1> := cflow(Stat),
    <Entry2, Rel2, Exit2> := cflow(Stat+)

cflow(Stat ; Stat+) =
    <
    Entry1,
```
union(Rel1, union(Rel2, product(Exit1, Entry2))), Exit2
>
[cfg-2] cflow() = <{}, {}, {}>
equations
[cfg-3]
<Entry, Rel, Exit> := cflow(Stat*),
Control := <unparse-to-string(Exp),
get-location(Exp)>
==================================
cflow(while Exp do Stat* od) =
<
{Control},
union(product({Control}, Entry),
union(Rel, product(Exit, {Control}))),
{Control}
>
[cfg-4]
<Entry1, Rel1, Exit1> := cflow(Stat*1),
<Entry2, Rel2, Exit2> := cflow(Stat*2),
Control := <unparse-to-string(Exp),
get-location(Exp)>
==================================
cflow(if Exp then Stat*1 else Stat*2 fi) =
<
{Control},
union(product({Control}, Entry1),
union(product((Control), Entry2),
union(Rel1, Rel2))),
union(Exit1, Exit2)
>
[default-cfg]
Control := <unparse-to-string(Stat),
get-location(Stat)>
==================================
cflow(Stat) = <{Control}, {}, {Control}>

Notes:
1. A start equation that links the extractor to the Meta-Environment: it creates a new rstore, fills it with the two defined extractions, and returns it as result.
2. Add a new variable StatementHistogram of type rel[<str, int>] to the store and apply the auxiliary function countStatements to collect the desired facts.
3. Perform the counting for three statement categories: assignments, conditionals and loops. For each category a tuple of the form <Category, N> is added to the relation. The function inc (imported via the module RStores) adds the value of N for the relevant category (or creates a new tuple if it did not yet exist).
4. The function controlFlow declares a new variable ControlFlow of type rel[<str, loc>, <str, loc>] and applies the auxiliary function cflow to collect the facts.
5. The function cflow returns a triple: the entry points, the internal control flow, and the exit points of the investigated language construct. In the case of a statement sequence, the exits of the first statement and the entries of the following statements have to be connected via a Cartesian
product to create all possible control flow transfers between them. The internal relations Rel1 and Rel2 are added to this and the appropriate triple is returned.

A slightly more complex construction in order to cater for all control flows in a while loop. Observe that the control expression is the entry as well as the exit of the while-statement. Internally, the exits of the body of the loop have to be connected with the control expression. Here again, some plumbing is necessary to collect all the possible control flow from control expression, via then or else branch to the exits of the statement.

**Using the Pico extractor in the Meta-Environment**

To run a fact extraction on a Pico program perform the following steps:

- Open the module `languages/pico/extract/Pico.sdf` from the ASF+SDF Library.
- Open the term `fac.trm` over this module (you may need to copy it first from this article or from the SDF Library in the examples directory).
- Open the Pico menu and select the entry extract.

After some delay the extraction is complete (check, for instance, the progress messages at the bottom of the window). The result is an rstore that we can inspect with the fact browser that is available in the left pane under Facts. For the various variables in the store different visualisations are available (they depend on the actual type of each variable). The effect of selecting the graph visualization for the ControlFlow variable is shown in Figure 1.23, “Graph display of ControlFlow” (p. 57).

**Figure 1.23. Graph display of ControlFlow**

Another example is the visualization of the StatementHistogram variable as bar chart shown in Figure 1.24, “Bar chart display of StatementHistogram” (p. 58).
Take home points

The Pico extractor illustrates the following points:

- Traversal functions are very useful for writing relatively short extraction functions.
- Rstores can be used to represent extracted facts.
- Fact extract can be integrated in The Meta-Environment and is activated via the extract button.
- The Meta-Environment supports various visualizations for rstores.

To Do

- Complete the compiler section.