Human-Usable Textual Notation for ArkM3

Author:
Jelle Slowack

University of Antwerp

Supervisor:
Bart Meyers
Advisor:
Hans Vangheluwe

Abstract

The use of models becomes increasingly important, especially for the development of large complex software systems. These (meta-)models can be defined in both a graphically or textually way. Each modelling tool has its own features, advantages and disadvantages. The aim of this thesis concerns the development of a Human-Usable Textual Notation (HUTN) for ArkM3. ArkM3 is an executable meta-meta-model of AToMPM, which is the successor of AToM^3. The HUTN supports deep meta-modelling (multiple meta-levels) and provides the syntax to define models in terms of classes, associations, actions and constraints. This textual modelling language is a neutral and independent language. This means that the basic syntax of the language does not allow other language snippets. The HUTN has its own syntax, which could be extended or modified by defining new grammar rules. In this way HUTN is extensible and gives the user an opportunity to define domain specific languages.

June 28, 2013

Acknowledgement

First of all, I would like to thank my supervisor Bart Meyers for the patience and time he spent with me. I also would like to thank Prof. Hans Vangheluwe for the opportunity and the pleasant, interesting meetings we had. Next, I thank Simon Van Mierlo, it was always nice to work with him. Finally, i would like to thank my parents and two brothers, because they are always supporting me.

Contents

0. Introduction

In software engineering, the use of models becomes increasingly important, especially for the development of large complex software systems. Some of the advantages of using models are for instance: analysis of a system or code generation. Another technique is model transformations. Models are also very useful as documentation of a system. Thus there are many occasions where models are used. Today, users can define their own models (and metamodels) both graphically and textually using different tools. Examples of graphical tools are MetaEdit+ [1] and AToM^3 [2]. In a graphical tool, such as AToM^3, it is not only possible to draw models that conform to a particular meta-model. Users can also perform transformations on these models. On the other hand there are textual modelling tools e.g., metaDepth [3] and Kermeta [4], which have other advantages.

Krahn et al. [5] denoted several benefits of using a textual language instead of graphical language. Some of these textual language advantages are:

- **Information content**: graphical languages usually need more space (on your screen or a sheet of paper), than textual languages to display models;
- **Description of constraints**: actions and constraints on models are very hard or even impossible to describe graphically. Almost every graphical model contains textual snippets to describe, for instance, boolean expressions;
- **Platform and tool independency**: graphical language users have to draw models in the respective tool. They cannot modify or read the model without that particular graphical tool. Textual language users can modify and read their text in any text editor.

The advantages for the developer are for example the use of existing editors. He does not have to write a whole new editor, but can write a plug-in for Eclipse for his new textual language for highlighting en code completion. Krahn et al. [5] also suggested that *speed of creation* of textual models is superior to graphical models, but this is not always the case (using a specific meta-model tool). For example, drawing a petri net model in a petri net tool is faster than writing it textually. Graphical models are also useful to get a first impression. But a textual modelling language is a good alternative to graphical languages.

The aim of this thesis concerns the development of a textual modelling language for ArkM3[1] or in other words the development of a human textual notation (HUTN) for ArkM3. ArkM3 is described in the thesis of Xiaoxi Dong [6]

[1] ArkM3 = AToMPM reusable kernel Meta-MetaModel

as the meta-meta-model used in the AToMPM project, the successor of AToM^3. Using this textual notation for ArkM3, users can create meta-models in terms of classes and associations. Subsequently users can define actions and constraints on these models.

This thesis is divided in three parts. The first part consists of related work. The second part explains the ArkM3 design and describes the various modifications that has been made. The third part defines the syntax and implementation of the HUTN.

Part I
Related Work

1. Compiler-Compiler comparison

Developing a textual notation, implies the need for a compiler or parser which is capable of interpreting our notation. Today there are several tools or modules which facilitate this task. The most important type of these tools are compiler-compilers (or parser generators). The selection of the right compiler-compiler for this thesis depends on the conclusion of the paper: Compiler-Compiler Comparison with Python Support [7]. In this report various compiler-compilers were compared. The generation of Python is a requirement, because ArkM3 is also written in Python. This section explains some parsing concepts and also contains several (modified) abstracts of the comparison report. At the end, the chosen compiler-compiler (i.e. PLY [8]) will be illustrated. The complete comparison report can be found in Appendix A.

1.1. Introduction to lexers and parsers

> "To translate a program from one language into another, a compiler must first pull it apart and understand its structure and meaning, then put it together in a different way."
>
> Andrew W. Appel (Lexical Analysis) [9]

A compiler usually consists of a lexer and parser. The role of the lexer is to read the input characters of the source and break this stream of characters into tokens (i.e. tokenize). Ulman et al. [10] used three distinct terms to describe the process of tokenization:

- A *token* consists of a token name and an optional attribute value. The name of the token is an abstract symbol representing a kind of lexical unit (e.g. keyword, identifier) and these token names are used as the input symbols for the parser;
- A *pattern* is a description of the form that lexemes of a token may take;
- A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexer as an instance of that token.

Suppose there is an expression `total = 4 + 8.2` and our lexer has the following lexer rules:

```
1 Identifier = "[a-zA-Z]"
2 AssignmentOperator = "="
3 PlusOperator = "+"
```

```
4  Integer = "\d+"
5  Real = "\d*\.\d+"
```

On the left-hand side the token names are defined and the right-hand side consists of regular expressions which define the pattern of our token. When the lexer takes the previous expression as input, then the following token stream will be generated:

```
1  Identifier AssignmentOperator Integer PlusOperator Real
```

These tokens correspond to certain lexemes of the input stream, namely:

- `Identifier` corresponds to `total`;
- `AssignmentOperator` corresponds to =;
- `Integer` corresponds to `4`;
- `PlusOperator` corresponds to +;
- `Real` corresponds to `8.2`.

The generated token stream of the lexer is the input for a parser, which has mainly two functions:

1. *conformance checking*: it checks if the token stream conforms to the definition of the language (i.e. the syntax);
2. *building a parse tree*: this tree represents the syntactic structure of a token stream.

The syntax of a language is described using a grammar which defines the structure of the language by means of grammar rules (i.e. productions). A grammar supporting the structure of previous expression (`total = 4 + 8`) can be of the form:

```
1  assignment  ->  Identifier AssignmentOperator expr
2  expr        ->  number PlusOperator number
3  number      ->  Integer | Real
```

This example provides three productions: `assignment`, `expr` and `number`. A production starts with the name of the production on the left-hand side (e.g. `assignment`). The right-hand side is the specification of the production, which has zero or more terminals and/or non-terminals. A terminal refers to a token type (e.g. `Identifier`) and a non-terminal refers to a production rule that has the same name (e.g. `number`).

1.1.1. Tree Construction

A parser uses these grammars and his productions to perform conformance checking and to build up a parse tree (see Figure 1). The parse tree could be seen as the path of the parser through the grammar (in-order traversal).

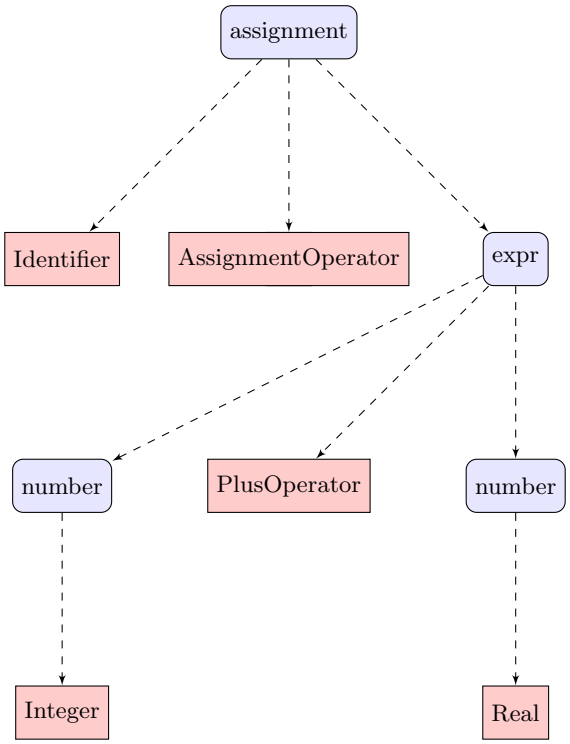

Figure 1: Parse tree for `total = 4 + 8.2` using the previous grammar

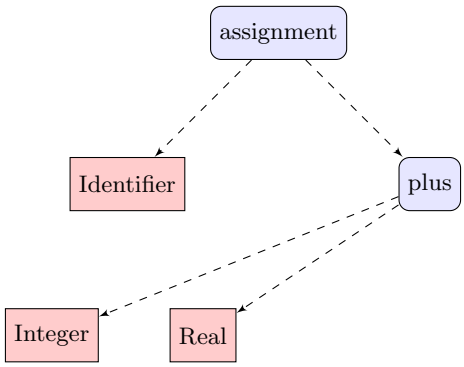

Figure 2: AST for `total = 4 + 8.2` using the previous grammar

As mentioned in the previous paragraph a parser generates a parse tree or in other words a syntax tree (because it represents the syntax). There are two kinds of syntax trees, namely a concrete syntax tree (CST) and an abstract syntax tree (AST). A CST is a normal parse tree that conforms exactly with the syntactic structure of the grammar. An AST is more an abstract representation of the syntactic structure, i.e. it does not represent every detail that appears in the real syntax. Figure 1 is the CST for the expression `total = 4 + 8.2` and Figure 2 is the AST.

1.1.2. Types of Parsers

There are essentially two types of parsing[11]:

Top-down parsers

Top-down parsers generate a parse tree by starting at the root of the tree (the start symbol), expanding the tree by applying productions in a depth-first manner. A top-down parse corresponds to a pre-order traversal of the parse tree. The weakness of top-down parsing is its predictiveness, since parsers have to predict the production that is to be matched. LL parsers are examples of top-down parsers. The L stands for "Left to right" as the parser reads the input from left to right and the L stands for Leftmost derivation, this means the leftmost non-terminal[2] is always derived. An example of a LL parser is ANTLR [12];

Bottom-up parsers

Bottom-up parsers generate a parse tree by starting at the tree's leaves and working toward its root. This technique is more powerful because the predictiveness is eliminated. These parsers only select a production if the entire right-hand

[2]Terminal symbols describe the input, while nonterminal symbols describe the tree structure behind the input.

side matches. A bottom-up parse corresponds to a post-order traversal of the parse tree. The most common bottom-up parsers are the shift-reduce parsers. The parser shifts symbols on to the parse stack and reduces a string of symbols located at the top of the stack to one of the grammar's non-terminals. The following example[3], in Listing 1 and Listing 2, explains the shift and reduce actions.

Listing 1: Sentence Grammar

```
1  Sentence   := NounPhrase VerbPhrase
2  NounPhrase := Art Noun
3  VerbPhrase := Verb | Adverb Verb
4  Art        := 'the' | 'a'
5  Verb       := 'jumps' | 'sings'
6  Noun       := 'dog' | 'cat'
```

Listing 2: Bottom-up parsing using shift and reduce on input: "the dog jumps"

```
1  Stack                  Input Sequence
2  ()                     (the dog jumps)
3  (the)                  (dog jumps)      SHIFT word on to stack
4  (Art)                  (dog jumps)      REDUCE using grammar
       rule
5  (Art dog)              (jumps)          SHIFT..
6  (Art Noun)             (jumps)          REDUCE..
7  (NounPhrase)           (jumps)          REDUCE
8  (NounPhrase jumps)     ()               SHIFT
9  (NounPhrase Verb)      ()               REDUCE
10 (NounPhrase VerbPhrase) ()              REDUCE
11 (Sentence)             ()               SUCCESS
```

LR parsers are examples of bottom-up parsers. The L stands for "Left to right" as the parser reads the input from left to right and the R stands for Rightmost derivation, this means the rightmost nonterminal is always derived. There are different types of LR parsers[11]:

- **SLR** parsers or Simple LR parsers have the simplest implementation. They do not have to scan through the possible reductions, because there is at most one reduction. Python Lex-Yacc [8] supports SLR grammars;
- **LALR** parsers are the intermediate form. They have smaller parse tables, because they reduce the amount of reductions. LALR parsers can handle more languages than SLR parsers and they are very efficient. Examples of LALR parsers are the GOLD Parser [13] and PLY [8];
- **LR** parsers are the most powerful parsers. They can parse a larger set of languages compared with LALR and SLR, however they have much bigger

[3]Example adapted from http://en.wikipedia.org/wiki/Bottom-up_parsing

parse tables which results in low efficiency. An example of a LR parser is Wisent [14].

Top-down (LL) vs. Bottom-up (LR)

A big difference between LL and LR grammars is that an LL grammar requires:

- **eliminating left recursion**: LL can't handle left recursion, because it will recurse forever due to the leftmost derivation;

Listing 3: Expression grammar using left recursion

```
1  Expr := Expr + Number
2    | Expr - Number
3    | Number
```

Listing 4: Expression grammar without left recursion

```
1  Expr := Number Expr_2
2  Expr_2 :=  + Number Expr_2? | - Number Expr_2?
```

- **factoring common prefixes**: this is required when two or more grammar rule choices share a common prefix string. This is necessary because LL parsing requires selecting an alternative based on a fixed number of input tokens. The `if-then-else`-grammar is a famous example that shares a common prefix string.

Listing 5: if-then-else grammar

```
1  S := if expr then S else S
2     | if expr then S
3     | other
```

Listing 6: The left-factored form of the if-then-else grammar

```
1  S := if expr then S E | other
2  E := (else S)?
```

LR parsing can handle a larger range of languages than LL parsing. Figure 3 shows the expressiveness of the several grammar classes. The number (0,1,k) between parentheses denotes the lookahead. This is the amount of tokens that the parser can look ahead at the next tokens in order to decide what to do.

1.2. Compiler-Compiler

Writing a compiler from scratch is a lot of work, because the programmer has to write the grammar and the code for the parser and lexer. We can address this problem by using compiler-compilers. These tools use a formal language description to generate a compiler. In this description the user defines the tokens and productions. In this way, we do not have to write the basic code for the

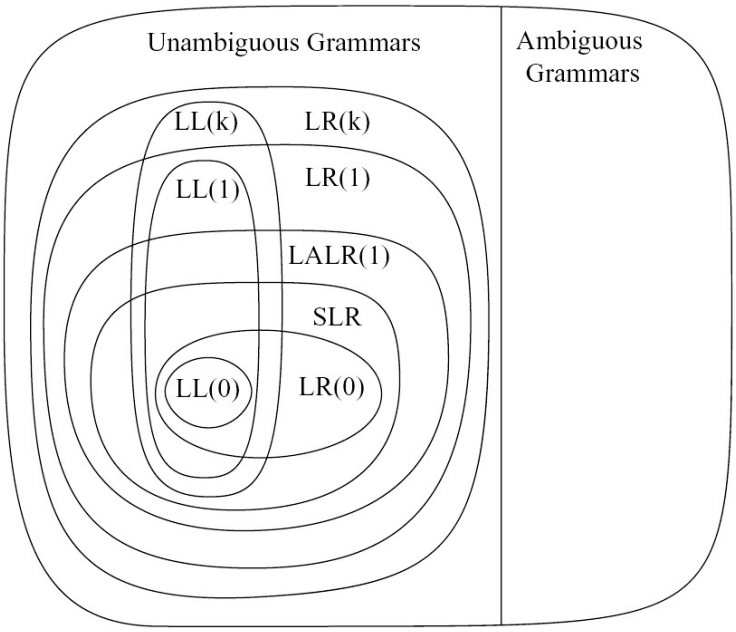

Figure 3: A hierarchy of grammar classes[9]

parser and lexer. The most common form of a compiler-compiler is a parser generator. The output of a parser generator is the source code of the parser (and lexer). The input is a formal description that is defined in a grammar. The generated parser is able to parse an input file according to the syntax defined in the grammar. A schematic overview can be found in Figure 4.

1.2.1. Island Grammars

Certain compiler-compilers support the concept of *island grammars*. An island grammar is used to parse a part of an input stream using an alternative grammar (by invoking another parser). For instance, a HTML parser is used to parse HTML documents which conform to its HTML grammar. If this HTML parser encounters a PHP expression within the HTML document. It invokes a PHP parser, which parses this PHP expression using its PHP grammar. This example descibes a PHP island grammar.

The island grammar concept has a lot of advantages. The first one is separation of concerns. There are two grammar files that are responsible for their own expressions, i.e. HTML and PHP expression. Two smaller grammer files instead of one (big) grammar file not only improves the readibility, it is also more efficient and faster. The number of grammer rules of the parser are reduced, which implies smaller parser tables. Another advantage is the reuse of the parsers. They could be used as a standalone parser or in other parsers and/or projects.

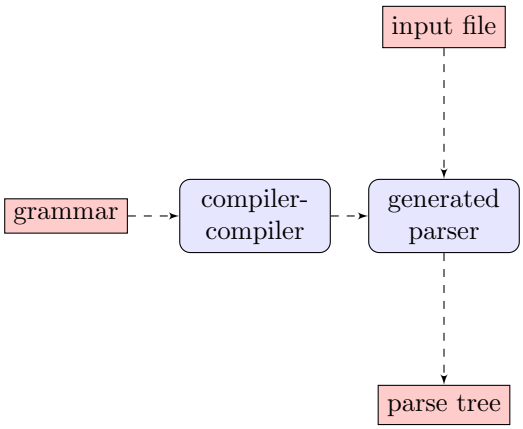

Figure 4: Schematic overview of a parser generator

1.2.2. PLY: Python Lex-Yacc

An example of a parser generator is PLY [8]. Python-Lex-Yacc is a Python implementation of the popular parser generator lex and yacc. PLY consists of two seperate modules, `lex.py` and `yacc.py`.

The lexer

The lex module is the lexer, which uses regular expressions to define the different patterns to match tokens. This module will produce a sequence of `LexTokens t`, where `t.type` conforms to the token name and `t.value` to the lexeme. There are two ways of specifying the patterns of tokens. For simple tokens the regular expression is defined in a string variable. But if some kind of action needs to be performed, then a token rule can be defined as a function where the documentation string contains the regular expression (i.e. the pattern of the token), see Listing 7. The name (without the prefix `t_`) of tokens defined in a string variable should conform to a name in the `token` list. This is similar for function-defined tokens, except that the user is able to change token name by assigning a new value to `t.type`.

The parser

The parser is defined using the yacc module. Each parser rule is specified as a function and prefixed with `p_`. The first function is the start symbol of the grammar and in each function the docstring contains the specification of the rule. If the abstract in Listing 8 was the complete grammar, then the start symbol would be `expression` and three grammar rules are reachable for this start symbol, i.e. the plus, minus and term expression. The statements after the docstring are the actions of that rule. The argument `p` is a tuple containing the values of each symbol in the rule, see Listing 8.

Listing 7: Abstract of a lexer in PLY

```
# List of token names.   This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'MULT',
   'DIV',
)

# lexer rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_MULT   = r'\*'
t_DIV  = r'/'

# lexer rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

1.3. Conclusion of Comparison Report

The comparison report compares three compiler-compilers in detail, namely SableCC3 [15], ANTLRv3 [16] and PLY [8]. The conclusion of the report is that PLY is chosen in favor of ANTLRv3 and SableCC3. Good support for both Python and island grammars made the difference. Table 5 provides a good overview of the conclusion.

Listing 8: Abstract of parser in PLY

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term MULT factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIV factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor(p):
    'factor : NUMBER'
    p[0] = p[1]
```

	SableCC3	ANTLRv3	PLY
Python support	++ (unstable)	++++ (stable)	+++++ (Python library)
Tree construction	+++ (rewrite rules)	+++++ (rewrite rules, operators)	+ (create own tree)
Visitor pattern	+++++ (extended visitor pattern)	++ (create own visitor)	+ (create own visitor)
Elegance	+++ (walkers, separation action vs. grammar)	+++ (grammar actions, tree grammars, predicates)	+++ (precedence rules, encapsulation, parser library)
Tools	+ (Python tools)	+++ (ANTLRWorks)	++ (debugger, Python tools)
Documentation	+ (no Python documentation)	+++++ (internet,books)	+++ (internet)
Island Grammars	++++ (lexer states)	+++ (it is possible, but dirty)	+++++ (lexer state, call other parser class)
Scalability	+++ LALR(1)	+++ LL(k)	++++ LALR(1)

Figure 5: Conclusion Compiler-Compiler Comparison

2. Textual Meta-Modelling

This section will handle existing textual meta modelling tools and languages.

2.1. Epsilon Object Language

The Epsilon Object Language or EOL [17] is one of the modelling languages provided by Epsilon[4] [18]. EOL is meta-model independent and builds on OCL[5] [19], which is a query and constraint language for MOF models like UML [20]. OCL is widely used in different model management tools, but it has some limitations [21], inter alia:

- OCL does not support CRUD operations on models (except read), therefore the modification of models is impossible;
- No support for statement sequencing, leading to complex statements which are hard to understand;
- Only one single model accessible concurrently.

The aim of EOL is to overcome the limitations of OCL. Statement sequencing is supported by separating the different statements using the "`;`" symbol and grouping of statements is achieved using the "{" and "}" delimiters. Another issue of OCL is its meta-model dependency, because it is only compatible with MOF models. EOL is not bound to a specific meta-model and is able to manage models from various technologies (MOF, EMF and UML). It adresses the need for a common infrastructure for model management by providing three key facilities:

1. Navigation of models, e.g. query operations like OCL;
2. Modification of models, e.g. adding an element to an existing model;
3. Accesing multiple models at the same time, which implies the user is able to define operations between multiple models (e.g. transformation, merging, etc.).

2.1.1. User-Defined Operations

There are two ways of defining operations in EOL, viz. context and context-less operations. In the first case, the context-type of an operation is explicitly specified. The advantage of context-typed operations is that it is possible to call this operations on instances of the type as if the operation was defined in the corresponding type. An example of an operation on a context-type is given in Listing 9. On the contrary, there are context-less operations. These operations are less readable and the context is given via arguments, see Listing 10 for an example.

[4] Epsilon = Extensible Platform of Integrated Languages for mOdel maNagement
[5] OCL = Object Constrain Language

Listing 9: Context-Defined Operation [17]

```
1.add1().add2().println();

operation Integer add1() : Integer {
 return self + 1;
}

operation Integer add2() : Integer {
 return self + 2;
}
```

Listing 10: Context-Less Operation [17]

```
add2(add1(1)).println();

operation add1(base : Integer) : Integer {
 return base + 1;
}

operation add2(base : Integer) : Integer {
 return base + 2;
}
```

Operations in EOL support three annotations of which two are executable and another simple annotation. The two executable annotations are pre and post: these annotations are used to define the resp. pre- and post-conditions of the current operation. The last and simple annotation is the cached annotation, which is used for parameter-less operations. When the cached notation is specified, the body of the operation will only be called once and the result of this operation will be cached. All successive calls use the cache to get the result of the operation. An example of pre- and post-conditions can be found in Listing 11, whereas an example of the cached annotation is illustrated in Listing 12. This example illustrate an implementation of Fibonacci numbers in EOL. When the cached annotation is specified, the body operation is called only 16 times (from `Integer` 0 to 15). Without caching the body would be called 1973 times.

2.1.2. Types

The type system of EOL is inspired by the OCL types. An overview can be found in Figure 6. `Any` type is the superclass of all types and conforms to the `OclAny` type in OCL. EOL types can be divided in four categories:

Listing 11: Pre- and post-conditions [17]

```
1.add(2);
1.add(-1);

$pre i > 0
$post _result > self
operation Integer add(i : Integer) : Integer {
 return self + i;
}
```

Listing 12: Cached annotation for parameter-less operations [17]

```
15.fibonacci().println();

@cached
operation Integer fibonacci() : Integer {
 if (self = 1 or self = 0) {
  return 1;
 }
 else {
  return (self-1).fibonacci() + (self-2).fibonacci();
 }
}
```

- Primitive types, i.e. `String`, `Integer`, `Real` and `Boolean`;
- Collection types and Map type, i.e. `Sequence`, `Bag`, `Set`, `OrderedSet` and `Map`;
- Native types;
- Model element types.

The table below gives an overview of the collection types.

Type	Ordered	Unique
`Bag`	no	no
`Set`	no	yes
`Sequence`	yes	no
`OrderedSet`	yes	yes

2.1.3. Native Typing

Native Typing is the provision for a user to create objects of the underlying programming environment. This useful for users who need to implement funtionality that is not supported in EOL. For instance, in a Java implementation

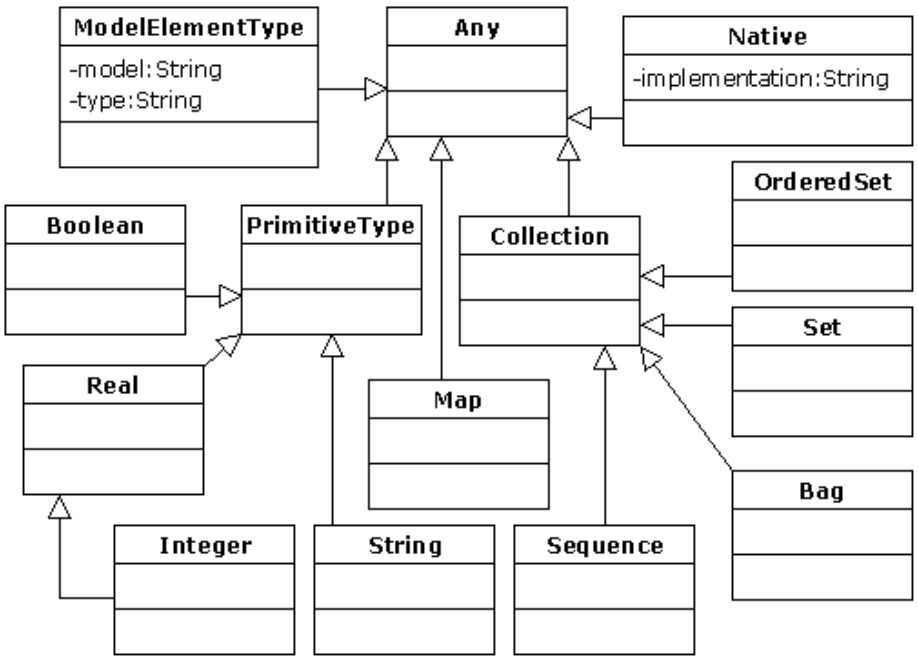

Figure 6: Type system of EOL from [17]

of EOL, users can create instances of existing Java classes via its identifier. The `implementation` attribute in the `Native` type class contains this identifier. An example of a native type is given in Listing 13.

Listing 13: Native type for the java.io.File class

```
1 var file = new Native("java.io.File")("myfile.txt");
2 file.absolutePath.println();
```

The fourth category is Model Element Types. These types are used to access specific types of a model. The textual syntax for model element types is performed via the `!` operator. For instance, `UML!Class` will return the `Class` type in the `UML` model. If there are conflicting type names, then users are able to define the absolute path separated by a `::` symbol. For example, the full path for the same UML class is `UML!Foundation::Core::Class`.

2.1.4. Type Operations

Users of EOL can perform built-in operations on all instances of these types. For instance, it is possible to get all instances of a certain Model Element Type by calling the operation `allInstances()`. For example `UML!Class.allInstances();` returns a set that contains all elements of type `Class`. Each type in EOL has its own operations and all these operations are described in the Epsilon Book [17]. Because all types inherit from `Any` type, they all have the operations of `Any`

type. For example, the operation `isTypeOf(type : Type)` which returns true if the current object is of the given type.

2.1.5. Expressions and Statements

In order to navigate properties of the various models and invoke operations on them, the same operator as in OCL is used: `.` or `->`. Arithmetical operators (i.e. `+`, `-`, `*` and `/`), comparison operators (`=`, `<>`, `>`, `>`, `<=` and `>=`) and logical operators (`and`, `or`, `not`, `implies` and `xor`) are also supported.

Declaration of variables is done by using the keyword `var` and users are able to explicitly type the variable. If no type is specified, the type of the variable is assumed to be of `Any` type. In case of non-primitive types, the initial value of the variable must be specified. The default value for primitive types like Integer, Boolean, String and Real are equal to respectively 0, false, "" and 0.0. An example is given in Listing 14

Listing 14: Variable declarations

```
1 var i : Integer = 5;
2 var c : new UML!Class;
3 var s : String = ``hello world''
4 var s2 : String
```

The statements in EOL are the one which are available in the most other classical programming languages. EOL provides following statements: if-else, switch, case, while, for, break, breakAll, continue and a throw statement. A `breakAll;` statement is used to break out of all loops (while and for). A throw statement is useful if a user wants to throw a exception (e.g. Java Exception) and afterwards catch this exception in his Java program. Another aspect that should be mentioned is the fact that there are two built-in loop variables `hasMore` and `loopCount`. These variables are accessible in each loop. The former `hasMore` variable is a boolean which defines if there are more elements in the current collection that will be executed in the loop. Variable `loopCount` is an integer used to count the number of loops and is set to one before the loop starts. Each iteration `loopCount` is incremented by one. Listing 15 is an example which contains various statements.

2.1.6. Constraints

It is possible to define constraints on existing models using EOL in terms of operations, but it does not provide textual syntax for contexts and invariants like in OCL. The Epsilon language responsible for this part is the Epsilon Validation Language[22], which is built atop of EOL. EVL is one of the seven task-specific languages that are provided by Epsilon.

Listing 15: Various statements in EOL

```
1  for (i in Sequence{1..10}){
2   if (i = 1){ continue; }
3   switch (i) {
4    case "1" : "1".println();
5    case "2" : "2".println();
6    case default : "default".println();
7   }
8   if (loopCount = 6){
9    break;
10  }
11 }
```

2.2. metaDepth

The Meta-Object Facility [23] is seen as the standard framework for the creation of models and meta-models. The MOF standard has been adapted by several tools and frameworks and the most known framework is the Eclipse Modelling Framework [24]. One of the characteristics is that MOF is designed as a four-layered architecture, where each element in a layer is an instance of one element at the above layer. This approach implies limitations, because there is only one instance-of relationship between two successive layers (meta-levels). metaDepth [3], another meta-modelling framework, addresses this problem by permitting an arbitrary number of meta-levels, i.e. deep meta-modelling.

2.2.1. Deep Meta-Modelling

In [25] an example is given where three meta-levels are squeezed into two meta-levels when deep meta-modelling is not applied. The example considers the type object pattern [26] which defines that a type of an object should also be an object. The given example contains *Products* and *ProductTypes*. In a two-level meta-model the classes `Product` and `ProductType` are defined in the upper meta-level. Whereas the instances of these classes, resp. `Book` and `mobyDick`, are located in the lower meta-level, see Figure 7. This solution has two drawbacks. First of all the user has to manually maintain the links between `Product` and `ProductType`. Secondly the user just want to create a `ProductType` instance like `Book` or `CD`, but now he also has to define the artificial instance `Product`. The three-level meta-model approach solves these two problems. There are no links to maintain and there is artificial `Product` instance, instead there are only `Book` and `CD` instances available.

2.2.2. Potency

Note that the figure of the deep meta-modelling approach presents an `@` followed by a number. This indicates the potency value on an element (model, class or attribute). Potency is used to control indirect instances in two or

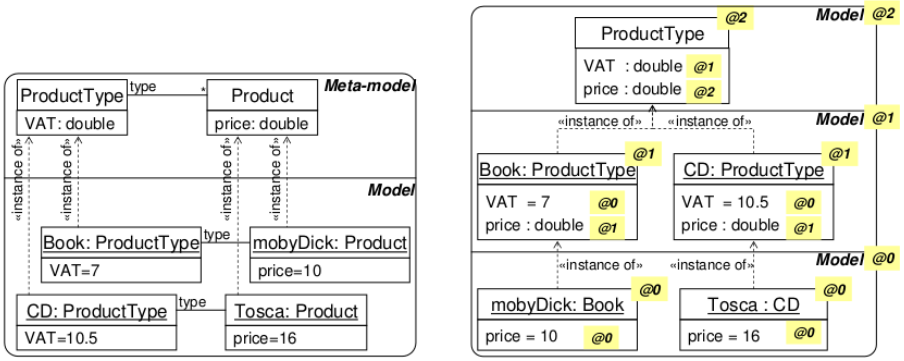

Figure 7: Classical vs. Deep Meta-Modelling [3]

more meta-levels below. For each meta-level downwards, the potency value will decrease by one. If a potency value is greater or equal to zero, the corresponding entity can be instantiated.

2.2.3. Linguistic vs. Ontological

Another remark in the previous example is that the middle meta-level have both class and object facets, i.e. a `Book` is an instance of `ProductType` and can be instantiated (e.g. `mobyDick`) in the lower meta-level. Because the element is not only a class but also an object, the term *clabject* was introduced. Elements of the different meta-levels are all *linguistic* instances of *clabject*, whereas the elements within the domain (e.g. `mobyDick` is a `Book`) define the *ontological* instantiation, see Figure 8.

2.2.4. Textual syntax

There is a textual syntax for building models, and the metaDepth framework is also integrated with Epsilon. Due to this integration, the user can describe actions and constraints using EOL [17] (or Java) expressions. Constraints can be specified inside or outside a clabject and is always surrounded by two `$` symbols: `minVat` and `minPrice` are examples of constraints in Listing 16. Actions are defined by entering the EOL execution mode of metaDepth, see Listing 17. Listing 16 provides the textual syntax of Figure 7. `Store` has a potency of 2, which means it can instantiated in the subsequently two meta-levels. All elements inside `Store` have the same potency 2 as his container `Store`, except `VAT` has a potency equal to 1. `ProductType` has also two constraints (i.e. `minVat` and `minPrice`).

2.2.5. Strict vs. Extensible Meta-Modelling

metaDepth can work in two ontological instantiation modes, viz. *strict* and *extensible*. The strict case is the classic meta-modelling, where the (above)

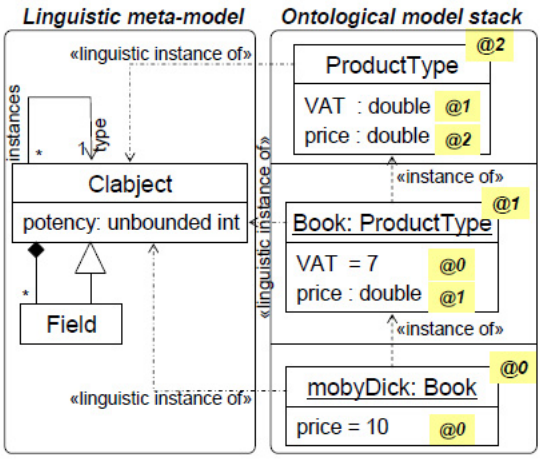

Figure 8: Linguistic vs. Ontological instantiation [3]

meta-model defines all the language properties for the model (i.e. the instance of the meta-model). It is not possible to add new elements or types to the instantiation. In other words: an instance conforms strictly to his meta-model. Strict elements are marked with the `strict` keyword. The extensible case provides the possibility to extend an instance with new elements or types. Extensible elements are marked with the `ext` keyword. If no keyword is defined, the element is extensible. An example using these keywords is given in Listing 18 and Listing 19.

Listing 16: Textual syntax for ProductType example [3]

```
Model Store@2 {
 Node ProductType {
  VAT@1 : double = 7.5;
  price : double = 10;
  minVat@1 : $self.VAT>0$
  minPrice@2 : $self.price>0$
 }
}
Store Library{
 ProductType Book { VAT = 7; }
}
Library MyLibrary{
 Book mobyDick { price = 10;}
}
```

Listing 17: EOL action generating 2000 books [27]

```
# EOL
for (i in Sequence{1..2000}) {
  var b : new Book;
  b.println();
}
#
```

2.3. Kermeta

Kermeta could be seen as an extension of the EMOF standard [23] (cf. Arkm3 Object Package is also based on the EMOF standard). It adds constraints and operational semantics (actions) to the standard. Analogous to ArkM3, actions and constraints are added to the classes in Kermeta. Kermeta is used within and dependent on the Eclipse[6] environment and the minimum requirement is Eclipse 3.6.x, which means Kermeta is editor-dependent.

Kermeta is aspect-oriented. The aspect-oriented paradigm aims to increase the modularity of a system by separating funtional units (aspects). The `require` operator in Kermeta is a key feature for supporting aspect-oriented design. This operator allows extension of a metamodel in sense of properties, operations, constraints and even the addition of new classes. Using the `require` operator it is possible to import different operational semantics from different functional units.

Another nice feature of the `require` operator is that it can be used to im-

[6]Eclipse: `http://www.eclipse.org/`

Listing 18: A meta-model for class and object diagrams [3]

```
strict Model ClassDiagram@2 {
 ext Node Class{
  isAbstract@1 : boolean = false;
  in : Class[*];
  out : Class[*];
  noAbsObjects : $self.isAbstract=false$
 }
 ext Edge Assoc(Class.out,Class.in);
}
```

Listing 19: A class and object diagram [3]

```
ClassDiagram Zoo {
 Class Person {
  name : String {id};
  pet : Animal[*] {out};
 }
 Class Animal {
  kind : String {id};
  owner : Person[1..*] {in};
 }
 Assoc hasPet (Person.pet, Animal.owner) {
  since : int;
 }
}
Zoo myZoo {
 Person p{ name = "Juan"; }
 Animal a{ kind = "monkey"; }
 hasPet(p,a){since = 2010;}
}
```

port metamodels. For instance, a user can graphically create a metamodel using the Eclipse Modelling Framework [24], thereafter he can import his created ecore metamodel in his Kermeta program. This exploits the advantages of both graphical and textual models. Besides importing Ecore files, it is also possible to import OCL [19] constraints.

Jézéquel et al. [28] illustrated the use of the `require` operator in Kermeta. They build an integrated environment for the Logo language. This language is used to navigate a virtual turtle on a board that draws figures when its pen is down[7]. The following example is taken from the paper of Jézéquel et al. [28]. Listing 20 shows an example of a Logo program. Figure 9 on page 30 shows the metamodel of the Logo language developed in the Eclipse modelling Framework. Listing 21 is the OCL file that defines a constraint on the `ProcCall` class and a constraint on the `ProcDeclaration` class. Listing 22 is the Kermeta program that includes the Ecore metamodel and the OCL constraints on that model.

As mentioned before, there is an alternative way. Both the metamodel and the contraints can also be defined in Kermeta. The metamodel in Kermeta for the corresponding Ecore metamodel can be found in Appendix B.1 and the constraints in Kermeta corresponding to the OCL constraints can be found in Appendix B.2.

Listing 20: Logo square program [28]

```
1  # definition of the square procedure
2  TO square :size
3    REPEAT 4 [
4      FORWARD :size
5      RIGHT 90
6    ]
7  END
8
9  #clear screen
10 CLEAR
11
12 #draw a square
13 PENDOWN
14 squar(50)
15 PENUP
```

Listing 21: OCL constraint on the Logo meta-model (StaticSemantics.ocl) [28]

```
1  package kmLogo::ASM
2
```

[7] Logo language on wikipedia: `http://en.wikipedia.org/wiki/Logo_(programming_language)`

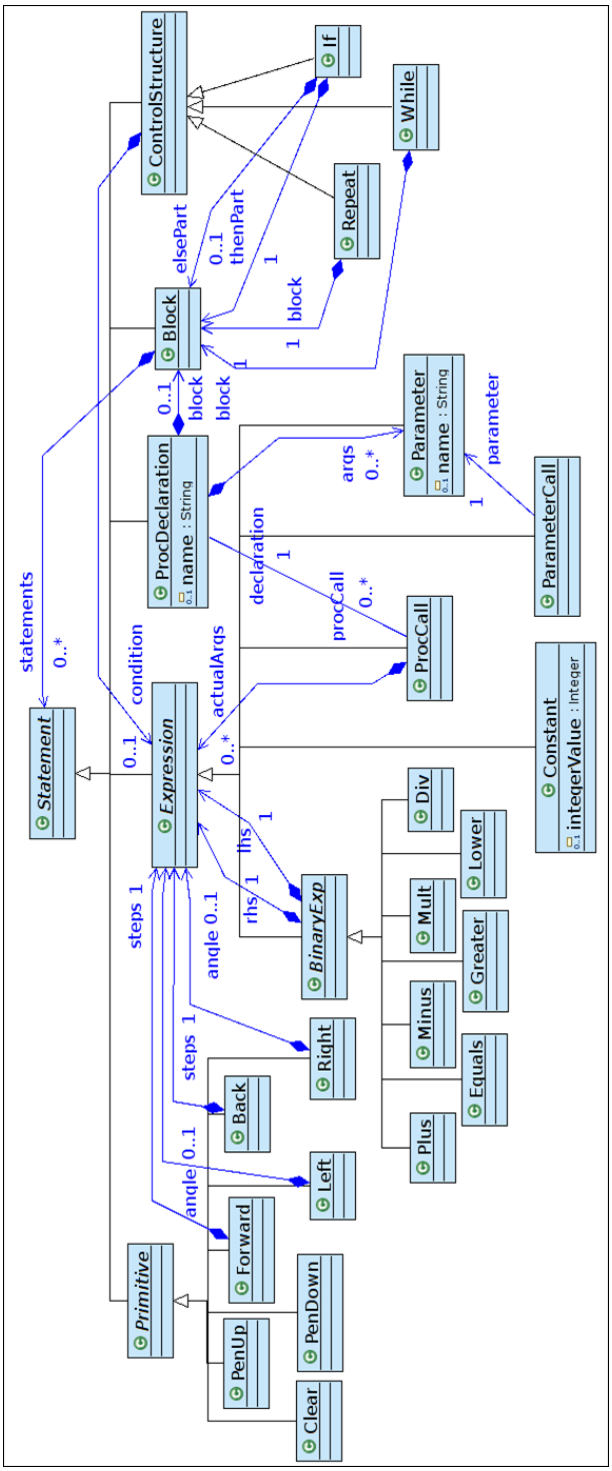

Figure 9: Logo metamodel using Ecore (ASMLogo.ecore) [28]

```
3  context ProcCall
4  inv same_number_of_formals_and_actuals :
5    actualArgs->size() = declaration.args->size()
6
7  context ProcDeclaration
8  inv unique_names_for_formal_arguments :
9    args->forAll(a1, a2 | a1.name = a2.name implies a1 = a2)
10
11 endpackage
```

Listing 22: Excerpt of the Kermeta program [28]

```
1  package kmLogo;
2  require "ASMLogo.ecore"
3  require "StaticSemantics.ocl"
4  [..]
5  class Main {
6    operation Main(): Void is do
7    // load a Logo program and check constraints on it
8    // then run it
9    end
10 }
```

The previous example shows how static semantics, like the metamodel and his constraints, are handled. Moving on to operational semantics, firstly the runtime model and its operations have to be created. Secondly the operational semantics of the Logo language have to be defined and finally the interpreter has to be implemented. The logo runtime metamodel is in this case a Turtle that can move and draw figures when his pen is down. The metamodel is defined in Figure 10. This metamodel is imported in a Kermeta file (LogoVMSemantics.kmt) to implement operations like `rotate`, see Listing 23. Next we need to attach this to Logo language, i.e. bind the abstract syntax of the Logo language to the operations of the runtime model. For example when a Logo language user writes the command `FORWARD`, the turtle has to move forward. An excerpt of the operational Logo semantics is described in Listing 24. Each statement has an operation `eval` which performs the appropriate actions. Lastly the interpreter uses this `eval` command to start the execution of the Logo program. This file imports four files:

1. ASMLogo.ecore is the metamodel of the Logo language;
2. StaticSemantics.ocl are the constraints on this metamodel written in OCL;
3. LogoVMSemantics.kmt is the Kermeta file that requires VMLogo.ecore, which is the runtime metamodel (Turtle), and the operations defined for this metamodel;
4. OperationalSemantics.kmt are operational semantics on the Logo language.

An excerpt of this file can be found in Listing 25, more details about this example and further information about Kermeta can be found in the paper Model Driven Language Engineering with Kermeta [28]. The characteristics of Kermeta are mainly the weaving of semantics into metamodels. Metamodels can be defined in Ecore or Kermeta. Static semantics can be defined in OCL or Kermeta. OCL support is implemented with model transformation from the AST of OCL to the AST of Kermeta. Operational semantics are defined in Kermeta.

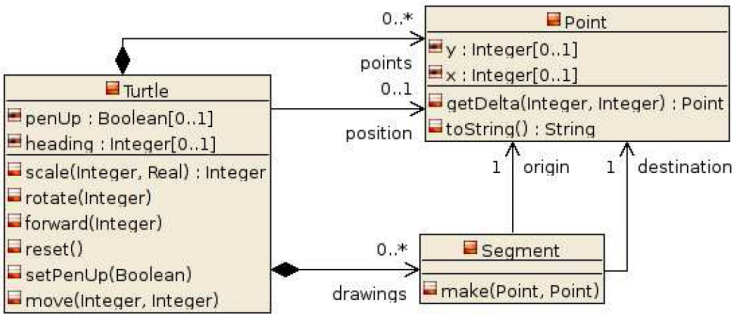

Figure 10: Logo runtime metamodel (VMLogo.ecore) [28]

Listing 23: Excerpt of the runtime model operations in Kermeta (LogoVMSemantics.kmt) [28]

```
1  package kmLogo;
2  require "VMLogo.ecore"
3  [..]
4  package VM{
5    aspect class Turtle{
6      operation rotate(angle : Integer) is do
7        heading := (heading + angle).mod(360)
8      end
9    }
10   [..]
11 }
```

Listing 24: Excerpt of the Logo operational semantics (OperationalSemantics.kmt) [28]

```
1  package kmLogo;
2  require "ASMLogo.ecore"
3  require "LogoVMSemantics.kmt"
4  [..]
5  package ASM{
6    aspect class Forward{
7      method eval(context : Context): Integer is do
8        context.turtle.forward(steps.eval(context))
9        result := voidtype
10     end
```

```
11    }
12    aspect class PenDown{
13      method eval(context : Context): Integer is do
14        context.turtle.setPenUp(false)
15        result := voidtype
16      end
17    }
18    [...]
19  }
```

Listing 25: Excerpt of the Logo interpreter [28]

```
1  package kmLogo;
2  require "ASMLogo.ecore"
3  require "StaticSemantics.ocl"
4  require "LogoVMSemantics.kmt"
5  require "OperationalSemantics.kmt"
6  [..]
7  class Main {
8    operation Main(): Void is do
9      var rep : EMFRepository init EMFRepository.new
10     var logoProgram : ASMLogo :: Block
11     // load logoProgram from its XMI file
12     logoProgram ?= rep.getResource("Square.xmi").one
13     // Create a new Context containing the Logo VM
14     var context : LogoVMSemantics::Context init LogoVMSemantics
           ::Context.new
15     // now execute the logoProgram
16     logoProgram.eval(context)
17   end
18 end
```

2.4. Conclusion of Textual Meta-Modelling Tools

EOL allows CRUD operations on models and complete the shortcomings of OCL. One of its features are user-defined operations, which could be context-typed (Listing 9) or context-less (Listing 10). A context-typed operation can be seen as attaching a method on a certain context or model. This is similar like in Kermeta, where static semantics (constraints) can be defined using OCL like in EOL. Operational semantics are defined by adding Kermeta operations to classes in the metamodel. In Kermeta the keyword `aspect` is used to define a certain context. The term comes from aspect oriented programming, whereas the semantics are weaved in to a certain model. metaDepth uses EOL for its operational semantics and OCL for its constraints. metaDepth also supports potency, to control instantiation of model elements in multiple meta-levels, which is not supported by EOL or Kermeta. Another feature of metaDepth are Clabjects. Each element in a model is a linguistic instance of `Clabject`.

Part II
ArkM3 Design

AToMPM [6, 29] (A Tool for Multi-Paradigm Modelling) is modelling environment under development, where every element is modelled using the appropriate formalism. The metamodel architecture of AToMPM is a two-dimensional architecture, a logical and physical dimension, as described in Xiaoxi Dong's thesis [6], see Figure 11 adapted from [6]. The logical dimension defines the abstract syntax of models and conforms respectively to the M1, M2 and M3 layer in MOF. The phyisical dimension is the internal representation of the meta-modelling tool (i.e. Himesis Graph).

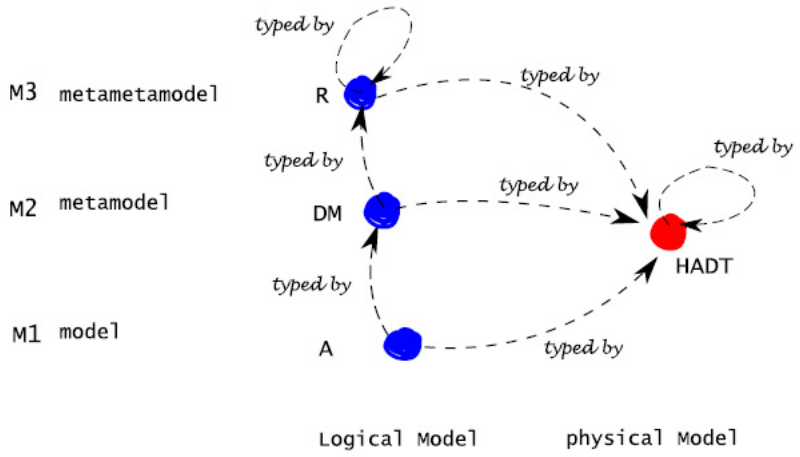

Figure 11: AToMPM two-dimensional architecture [6]

ArkM3 is an executable, self-described, general-purposed meta-metamodel and is the root of the logical dimension of AToMPM. Where AToM^3 had the Entity Relationship formalism as its meta-metamodel, AToMPM has ArkM3. This part will handle ArkM3 and will point out some changes that have been made. These adjustments are mainly made by Bart Meyers, Simon Van Mierlo or Jelle Slowack. The structure of this section corresponds to the structure of ArkM3 shown in Figure 12, namely the Object Package, the Action Language Package and the Data Type and Data Value Package.

3. Object Package

The object package is based on the OMG's standard EMOF [23] and consequently it supports an object-oriented definition of metamodels. Conforming

to this standard, the object package contains the definitions of elements like Package, Class, Association and Composition.

3.1. Original Design

Figure 13 describes the original UML diagram of the ArkM3 Object Package. In ArkM3 everything is a subclass of `Element` (datavalues, types, constraints, etc.). The most important characteristic of elements is that they can have actions and/or constraints. This implies that all elements in ArkM3 can have actions or constraints. The UML diagram consists of three specific elements, namely `NamedElement`, `TypedElement` and `MultiplicityElement`. These classes represent an element consisting of a name, type or multiplicity respectively. Next there is a `Package` class which can contain multiple elements. Packages are used to group (related) ArkM3 elements, creating an hierarchy. The `Type` class and his subclasses are discussed in section 4. The class `Class` inherits from `Type` and has properties which represent his attributes. Finally the `Composition` class is a subclass of `Association` which is a subclass of `Class` and `MultiplicityElement`. In this way a composition and association can have attributes and are able to derive and specialise themselves. The source and destination of an association corresponds to `isFrom` and `isTo`. The in- and outwards cardinalities of an association are properties.

3.2. Modifications

Figure 14 illustrates the modified UML diagram of the ArkM3 Object Package. In this paragraph all modifications with respect to the Object package are listed:

- Only elements of class type `ActionableElement` can have actions or constraints. In the original design all elements were allowed to have actions and constraints;
- A `Type` is no longer a subclass of `NamedElement`, because only a custom type should be a named element. See Section 4;
- Owned elements of a package are no longer instances of type `Element`. Instead an element in a package should be a `NamedElement`. The main

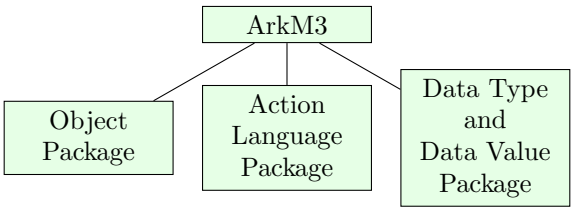

Figure 12: ArkM3 contains three packages

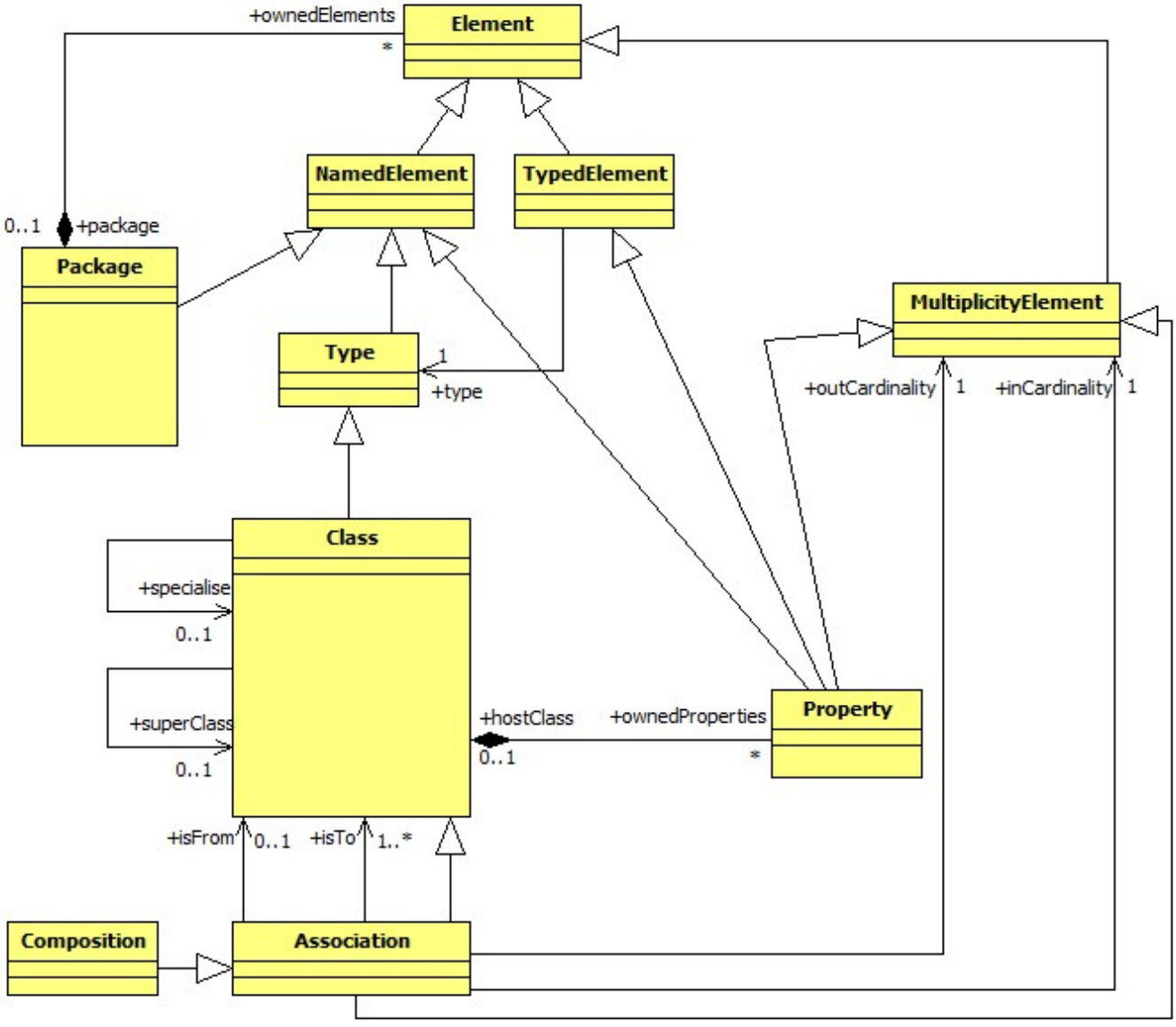

Figure 13: Original meta-model of Object Package

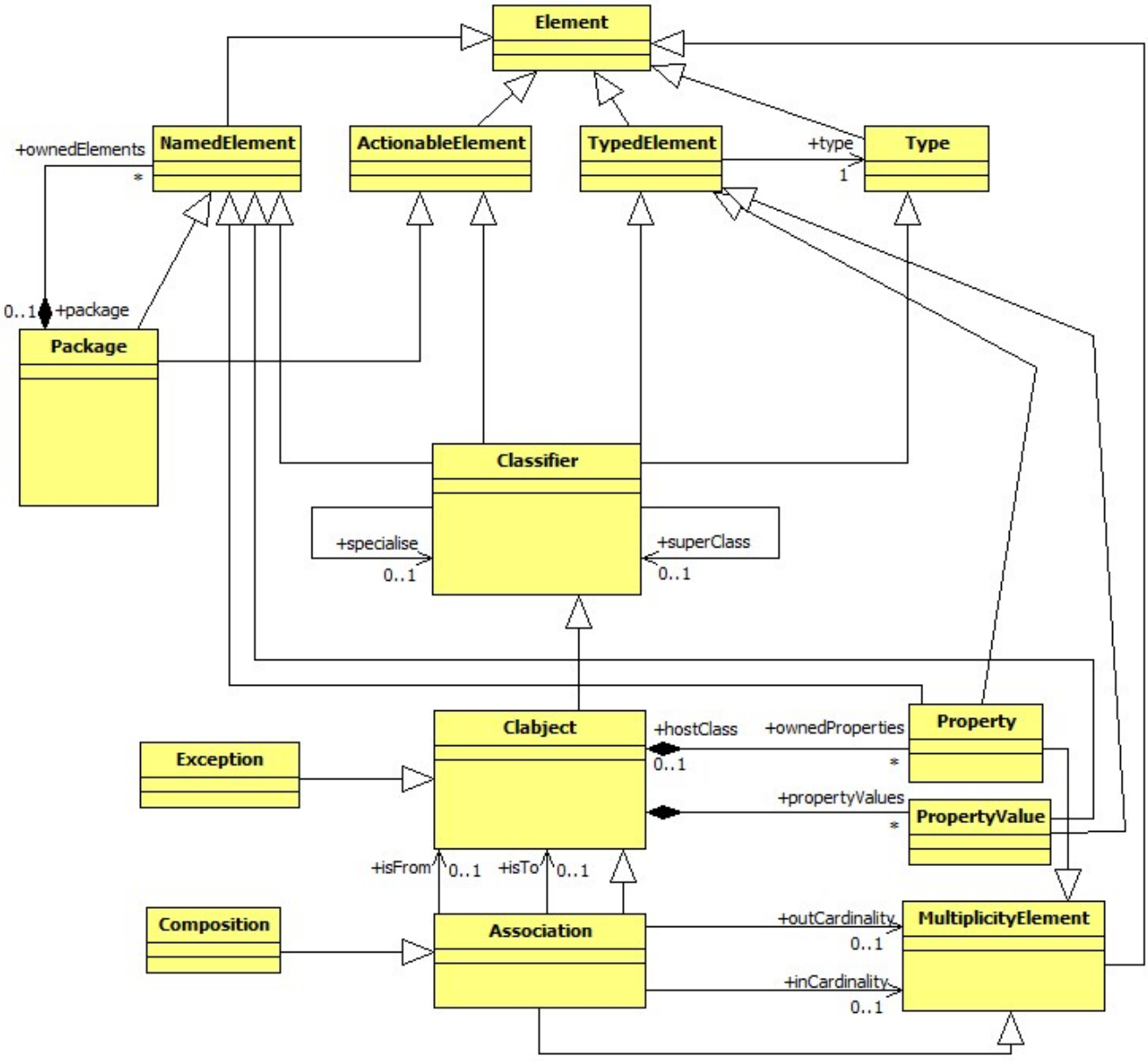

Figure 14: New meta-model of Object Package

reason is that elements within a package can now be referenced. An advantage of allowing only named elements in a package is that these elements can now be saved in a `MappingValue`. This increases the performance for looking up certain elements in a package. Before this modification, one has to manually loop over the elements to search a particular element;

- The orginal `Class` is split up in to two classes: `Classifier` and `Clabject`. This is influenced by the clabjects in metaDepth, because the original `Class` has both class and object facets. Thus the name `Class` is not correct and is replaced by `Clabject`. Typical class functionality is moved to the superclass `Classifier`;
- The concept of a `PropertyValue` is introduced. Properties define the type and default value of a property within a clabject. PropertyValues are the actual values or instances of a particular property. The type of a PropertyValue should conform to its property. An example of a property is an integer attribute `i` in metamodel *myInteger* which has a default value 0. In this case the PropertyValue could be the instantiation of that attribute `i` (e.g. `i=9`) in model *myIntegerInstance* which is an instance of *myInteger*;
- Multiplicities of `inCardinality` and `outCardinality` are changed from `1` to `0..1`. The reason for this change is that an abstract association can have no in- and/or out cardinality.
- As hyperedges are not supported, the multiplicity is adjusted for `isTo` from `1..*` to `0..1` (lower bound 1 is changed to 0 due to the abstract association issue described in the previous item).
- An `Exception` class is added, which is a subclass of `Clabject`. In this way, exceptions can have properties, actions and constraints.

4. Data Type and Data Value Package

In order to define actions and constraints, there has to be a notion of values. In ArkM3 data types and data values are separated (Type Object pattern [26]). Figure 15 gives an example of an IntegerValue 3.

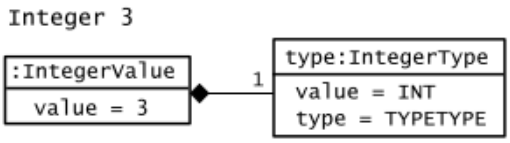

Figure 15: IntegerValue example from [6]

4.1. Original Design

This section will focus on the DataType package. The structure of this package is modified in order to allow a strongly typed language which can perform static type checking and where a user can write type expressions, see Section 6.2.1. ArkM3 provides the following primitive types (and values): `IntegerType`, `BooleanType`, `StringType`, `RealType` and `TupleType`. Besides the primitive types, there are collection types like a `SequenceType`, `SetType` and `MappingType`. The latter is a map or dictionary. All elements in a sequence are ordered and not unique, whereas the elements in a set are unordered and unique. Finally, there is also `AnyType` which is used when the type of a value cannot be determined. See Figure 16 for an overview of all the types in ArkM3.

4.2. Modifications

The original design is modified due to the fact that support for type expressions is added and the strongly typed language should be maintained. A type expression can define a sequence type which consists of integers. Another type expression can define a sequence which can consist of floats and strings. A mapping type can map an integer to any type. To allow these type expressions, the design needs to be adjusted. Figure 17 illustrates the modified UML diagram of the Data Type package. The first adjustment is that `AnyType` is the superclass for all data types. The reason for this change is that the `AnyType` will match all types and corresponds as the base type like the `Any` type in EOL or the `OclAny` type in OCL. The primitive types remained almost unchanged, except for the `RealType` which is renamed to `FloatType`. Another change is the `TupleType` being no primitive anymore, because it can contain multiple values which, in turn, have a type. Especially for type expressions, a `TupleType` must know the various types of its elements in order to check if a certain value corresponds with the types within the `TupleType`. Types like a set and a sequence only have one base type (e.g. set of integers). Therefore `SequenceType` and `SetType` have the `SingleBaseType` as a superclass. This class represents types with only one base type and all elements of sequence or set should conform to the base type. This base type can be a union of types. For instance, the base type of a set of integers and strings is a union type which defines the union of integers and strings. Another collection type is the `MappingType` which has a key- and valuetype.

Additionally there was a need for some extra types:

- `UnionType` provides the user the ability to make a union of types. For example a sequence which accepts integers and floats;
- `CustomType` is used for custom type declarations. It links a user-defined type name to a type expression. For instance the custom type *IntFloatList*

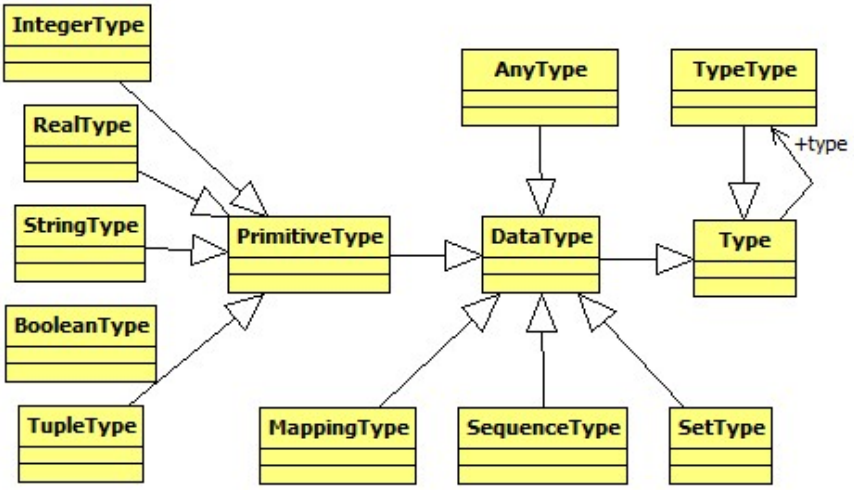

Figure 16: Original metamodel of DataType

is a user-defined type, which represents a sequence which can contain integers and floats. The name of this custom type is *IntFloatList* and the type expression is a sequence type, where the base type is a union of integer and float;

- `ImmutableType` defines that the value cannot be changed;
- `VoidType` is used when there is no value (i.e. void);
- `TypeReference` is used to refer to a model element (like Model Element Type in EOL). For example a package or a clabject (within a package). For instance, if there is a `Tiger` class within a `zoo` package, then a type reference is of the form `zoo.Tiger`. This can be used to create instances of this clabject `Tiger`.

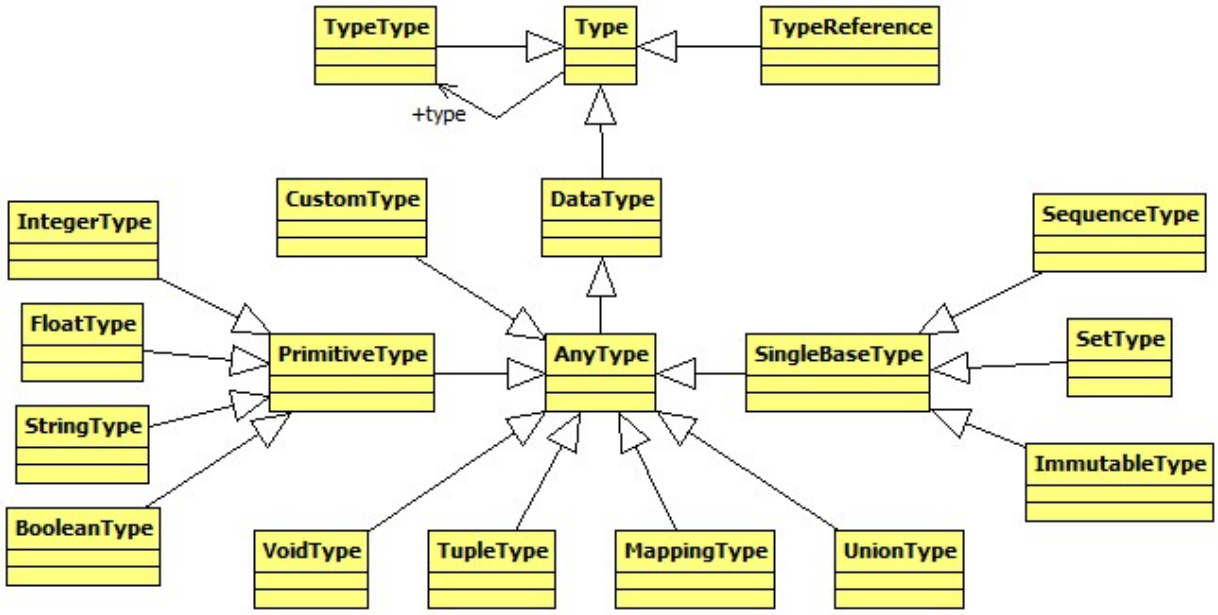

Figure 17: Modified meta-model of DataType

5. Action Language Package

The Action Language Package is the third package of ArkM3. The aim of this package is to provide a general-purpose action language. Like EOL is the action language of metaDepth, the Action Language Package defines the action language for ArkM3. The (modified) Action Language Package provides three main elements: constraints, actions and functions. Function are actually actions that can have parameters and a return value. These elements are all defined using the HUTN syntax and conform to the ArkM3 design. It is not possible to import OCL files like in Kermeta or to write OCL constraints like in metaDepth. The Action Language Package has its own design and syntax to describe the static and operational semantics.

EOL provides two types of operations: context-less and context-defined. The Action Language Package supports also user-defined operations that are functions or actions. These functions or actions can be added to model elements like, for instance, clabjects or associations. This operations are context-defined. It is also possible to define operations without a context, i.e. context-less. There operations are defined in a package.

5.1. Original Design

The package is divided in the following components:

- **Action and Constraint**: they both have a name and type. An action represents the operational behaviour of a model, while a constraint checks the semantics of a model. The latter always returns a boolean value;
- **Statements**: the possible statements are listed in the table below;

Statement	Attributes	Implementation
Expressions	expr, ... expr	traverse the content in order
Conditional	expr, stat1, stat2	if expr is true execute stat1 otherwise stat2.
Loop	var, init, term, stat	initialise var, execute stat and update var, iterate until term expression is true
Declaration	id, type	binds an identifier id to a certain type
Return	expr	return expr's value or reference to obj referred by expr

Table 1: Statements table adapted from [6]

- **Expressions and Operators**: an expression can be an identifier, a call expression or a reference. An operator is a subclass of the `Expression` class and following operators are provided:
  - Arithmetic operators, like `Plus` or `Minus`;
  - Boolean operators, like `And` or `Not`;
  - Type conversion operators, like `ToInt` or `ToBool`;
  - Comparison operators, like `GreaterThan` or `LessThan`;
  - Collection operators, like `Length` or `Append`;
  - Semantic operators, like `AllInstance` or `Filter`;
  - Model manipulation operators, like `Create` or `Read`.

  The complete UML diagram for the available operators can be found in Figure 18.

5.2. Modifications

The single modification within this component is the addition of a `Function` class. This class is a subclass of the existing `Action` class and has some important differences compared to actions. The first one is that a function can have a parameter list. In this way a user can provide arguments and perform action code on it, but they both have the concept of `self` if they are not declared in a package, i.e. context-defined. Secondly, functions have return values. Another difference between functions and actions (or constraints), is that functions are not separately stored in a list like in the `ActionableElement` attribute `ownedActions`. They are stored like clabject attributes, in properties or in property values:

- A *function declaration* is **stored as** an instance of a `Property`. A function declaration defines the name of a function together with its parameter types and return type. The name of the `Property` instance is the function name and the type of the property is a `MappingType`, which maps the parameter types to the return type. The key type of the MappingType is a TupleType containing the parameter types (in order). The value type of the MappingType conforms to the return type;
- On the other hand, the *function definition* (which defines the body of a function) is **stored in** a `PropertyValue` instance. The name of the property value is equal to the name of the function. The value of the property value is a `Function` instance. The signature of the function in the PropertyValue conforms to the signature defined in the Property. This means that parameter type and return type of the the function in the PropertyValue conforms to the Property according to the Liskov substitution principle.

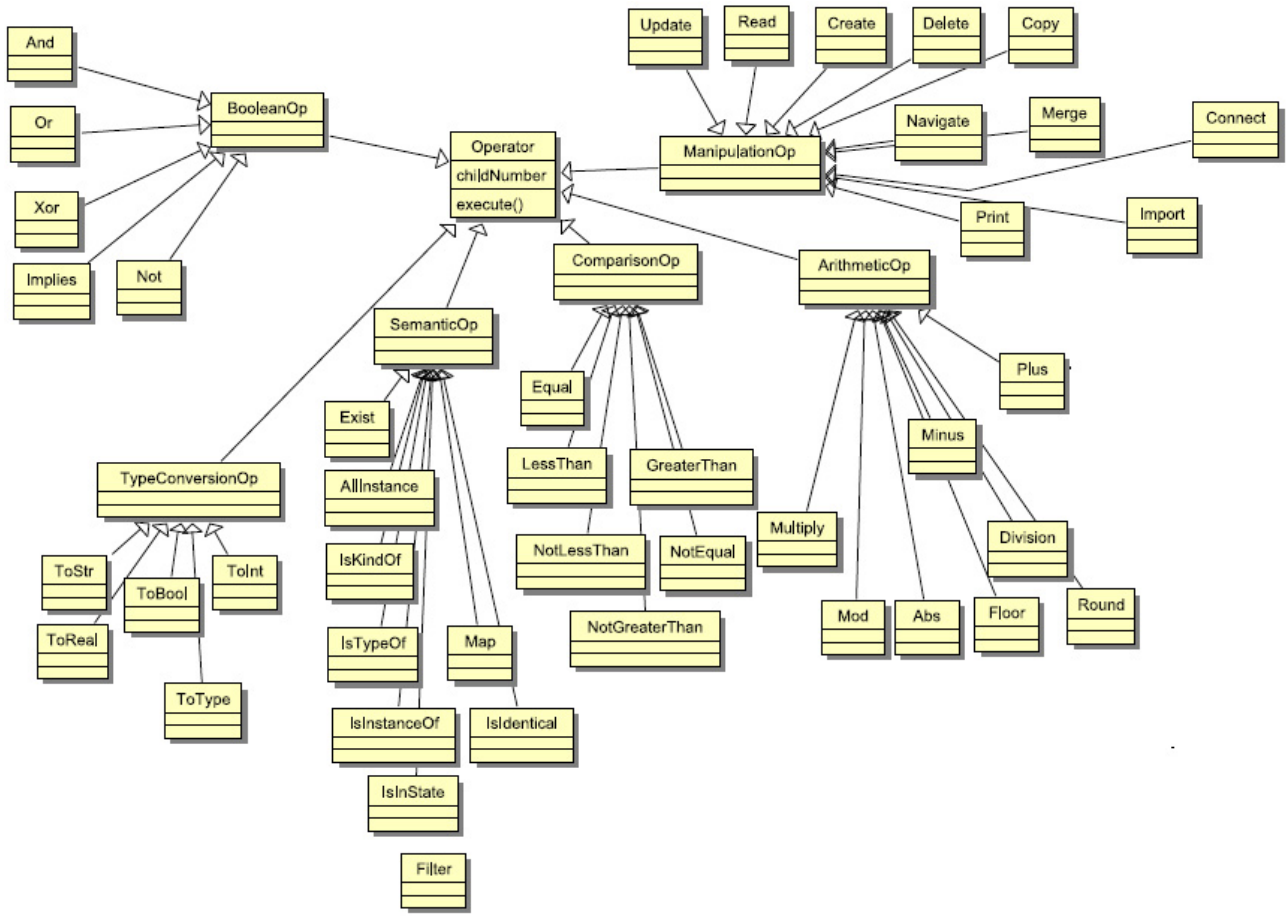

Figure 18: Operators in ArkM3 adapted from [6]

Note that a function declaration is stored in the same way as an attribute declaration with type `MappingType`.

Part III
HUTN

6. Syntax of HUTN for ArkM3

The Human Usable Textual Notation (HUTN) for ArkM3 is a Python-like language. We tried to let the syntax resemble Python, e.g. indentation levels are used to determine scope. In contrast to Python however the HUTN is statically typed: each variable has a type and it is possible to cast a variable to another type. This section will demonstrate all the features using various code examples. The organisation of this section corresponds to the structure of the ArkM3 Design. First of all there is the Object Language, next the Data Type and Data Value package is going to be discussed. Finally, the syntax of the Action Language Package is described.

6.1. Object Language

As discussed in Section II, the object package is based on EMOF. Thereby the HUTN for ArkM3 consists of concepts like packages, classes, instances and associations. The basic elements of the Object Language in the HUTN are:

- A **package** which is used to group model elements together;
- A **class**;
- An **instance**, i.e. instance of a class, association, composition, another instance;
- An **association** or **composition**.

A class and instance are clabjects in the ArkM3 design. The distinction between these two elements is their type. The type of a class is `arkm3.object.Clabject`, whereas the type of an instance is the clabject of its class. For instance, the class `Tiger` is a clabject with type `arkm3.object.Clabject`. The instance, `myTiger` is a clabject with type `Tiger`, which is a clabject.

HUTN (ArkM3)	metaDepth
package	Model
class/instance	Node
association	Edge

Table 2: Basic elements: HUTN vs metaDepth

The table above links the corresponding elements of the HUTN for ArkM3 to the elements available in metaDepth. ArkM3 also supports an arbitrary number of meta-levels, because a user can define multiple ontological instances. A Model and a package differ in some aspects. First of all, a package can contain other packages (they can be nested). A Model cannot be nested. Second, a Model represents a metalevel and a package does not. Instances of Model elements are defined in a metalevel below. Thus a model cannot contain instances of its elements, this elements can only be instantiated in the model or metalevel below. In ArkM3, a package can contain classes and instances of these classes. The Nodes and Edges of metaDepth conform to the classes/instances and associations of ArkM3, except that the latter does not support potency.

All elements of the HUTN are defined inside a package. A package is defined using the keyword `package` followed by the identifier of the package. A **package** can contain following elements:

- Another package;
- A class, instance, association or composition;
- An action or constraint;
- A function definition.

Actions are the operational semantics for a particular element. They have a name and a body (i.e. the action code) which contains statements and expressions. An action is defined using the keyword `action` followed by its identifier and a colon `:`. The action code is written on the next line and is indented to define a new scope. Constraints are the static semantics for the current element. Constraints also have a name and a body, the latter should always return a boolean value. To define a constraint one has to use the `constraint` keyword instead of the `action` keyword.

A **class**, an **instance**, a **composition** or **association** can contain the following elements:

- An action or constraint;
- An attribute declaration;
- An attribute definition.

Besides actions and constraints, users can declare and define attributes. The declaration of an attribute specifies a name, a corresponding type and an optional default value. This default value can be an expressions, in this case it is a derived attribute. A definition will define an attribute by assigning a value to the definition. In ArkM3, functions are treated in the same way as attributes. They actually are attributes (i.e. properties), see Section 5.2. A function declaration is an attribute declaration, where the type is a `MappingType` containing

the parameter list as key type and the return type of the function as value type of the map. A function definition is in an attribute definition, where the value of the attribute is the function definition. An example of a function declaration and definition is given in Listing **??**.

Listing 26: Function declaration and definition

```
1  # function declaration
2  {int, int : bool} sumIsOdd
3  # function definition
4  sumIsOdd = {int a, int b: bool} :
5    if (a+b) % 2 == 0:
6      return False
7    else:
8      return True
```

In Listing 27 three classes are defined which reside in a `test` package. The `Animal` class contains an attribute declaration which represents the age of the animal and has a default value zero. `Mammal` is defined as an abstract class and last but not least the `Dog` class inherits from the `Animal` and `Mammal` class (multiple inheritance). The `Dog` class consists of an action `gettingOlder` and a constraint `c`. `Lassie` is an instanceof `Dog`.

Listing 27: Classes in the HUTN for ArkM3

```
1  package be.ac.ua.test
2    class Animal
3      int age = 0
4
5    abstract class Mammal
6
7    class Dog(Animal,Mammal)
8      action gettingOlder:
9        if(self.age < 99):
10         self.age = self.age + 1
11     constraint c :
12       return self.age >= 0
13
14   Dog Lassie
15     age = 6
```

Note that a package statement (e.g.`be.ac.ua.test`) can be defined in one line using a `.` symbol to concatenate sequential package. Another way of defining a package is to use different scopes (i.e. indentation). In Listing 28 two alternatives for the previous package statement are given.

Listing 28: Alternative package definitions

```
1  #alternative 1
2  package be.ac.ua
3    package ua.test
```

```
4
5  #alternative 2
6  package be
7    package ac
8      package ua
9        package test
10         [..]
```

The last example illustrated the syntax for classes and packages, the next example will present the syntax for associations and compositions. The key features are:

- Actions, constraints and attributes;
- Multiple inheritance;
- Role names;
- Abstract, ordered and unique associations or compositions.

Listing 29 illustrates several associations and compositions. Attributes, actions and constraints are not illustrated, because their textual syntax is analogous to definition of attributes, actions and constraints in classes. The optional keywords `abstract`, `unique`, `ordered` should always be written at the beginning of the line. Multiplicities and role names are optional, as well as the keywords `abstract`, `unique` and `ordered`. If no role name is specified `from` and `to` will be used for navigation, e.g. `PersonGotLegs.from`. Otherwise a role name is defined, then this will be used for navigation, e.g. `DogOwnership.owner`. Associations, like the `DogOwnerShip` association, can also have a multiplicity, which should be interpreted as a lower and upper limit for the particular association. Hyperedges are not supported, hence an association is always from exactly one class type to another. Figure 19 shows the corresponding UML diagram for the associations and composition. Note that inheritance, uniqueness and the multiplicity of the `DogOwnerShip` association is not shown in the figure.

Listing 29: Associations in the HUTN for ArkM3

```
1  package be.ac.ua.test
2    [..]
3    abstract association Ownership
4
5    unique association DogOwnerShip<0..*>(Ownership)
6      owner:from Person<1..1>
7      owned:to Dog<0..*>
8
9    composition PersonGotLegs
10     from Person<1..1>
11     to Leg<0..2>
```

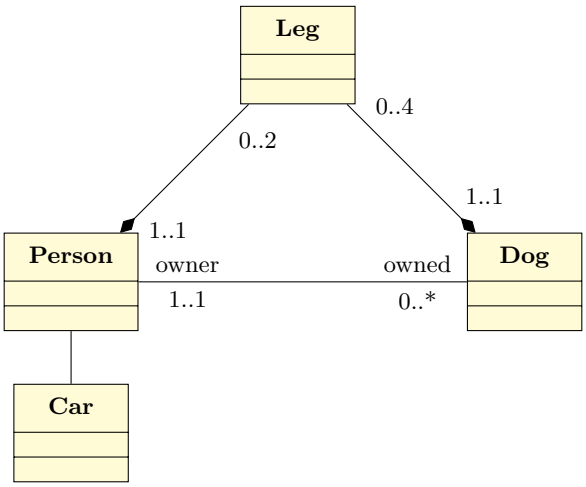

Figure 19: Corresponding associations and leg compositions

```
12
13    composition DogGotLegs
14      from Dog<1..1>
15      to Leg<0..4>
16
17    association CarOwnership(Ownership)
18      from Person
19      to Car
20
21    Person John
22    Dog Lassie
23
24    DogOwnership do
25      owner = John
26      owned = Lassie
```

6.2. Data Type and Data Value

The HUTN for ArkM3 provides the following primitive types: `float`, `int`, `bool` and `string`. The minimum and maximum values for these types are equal to the minimum and maximum for the corresponding Python values. The available collection types are:

- A sequence, where elements are ordered and not unique;
- A set, where elements are unique, but not ordered;
- A map, which maps a key to a certain value (where the key is a primitive type);

- A tuple, which consists of a number of values separated by a comma (like Python tuples). It is an immutable type, where individual items of a tuple cannot be changed.

Besides primitive and collection types, there are also two special types. The first one is the `any` type which will accept all types, secondly the `void` type is the type of the void value.

6.2.1. Type Expressions

Types are defined using type expressions, which allows a user to make combinations of different types. The features of type expressions are:

- Union types, representing a collection of two or more types. A union type accepts all the types defined in its collection.
  E.g. `int | string | float`, accepts integers, strings and floats;
- Set types, are defined using curly brackets.
  E.g. {`int`}, is the definition of a set containing integers;
- Sequence types, are defined using square brackets. They also have an optional parameter which defines the maximum size of the sequence.
  E.g. `[string 8]`, describes a sequence of maximum 8 strings;
- Mapping types, are defined using curly brackets and a colon to seperate the key from the value.
  E.g. {`int:string`}, is a type which maps integers to strings;
- Tuple types, are defined using a comma `,` to separate the different types.
  E.g. `int , int , int`, defines a tuple type containing 3 integers;
- Brackets, to group types together.
  E.g. {`(int | float) : string`}, describes a type which maps integers or floats to strings;
- Immutable types, are types describing values that cannot be altered.
  E.g. `immutable int`, is the definition of a constant integer.

Note that the language user is able to use and define custom types in type expressions, see Section 6.3.1.

Another remark is that one can refer to a type in a package, these are called type references. These type references can also be used in type expressions. For instance, a type expression defining a sequence of the previous `Dog` class (defined in package `be.ac.ua.test`) is written as `[be.ac.ua.test.Dog]`.

6.3. Action Language

The Action Language is part of the HUTN for ArkM3 and is used for executable expressions or statements. The action language is used in case of assigning a value to an attribute or when action body is started which is marked by a colon **:** (e.g. body of a function). It is also possible to write action files and use the action parser, without the notion of the object language.

6.3.1. Statements

The HUTN supports all statements described in the ArkM3 meta-model. In addition it also supports type declarations, exceptions and type casting. Type declarations allow users to define their own custom types and exceptions allow the users to raise and catch exceptions. Type casting is used to cast a value to a certain type.

6.3.2. Declarations

There are two types of declarations:

1. **Variable declarations** that bind a variable to a certain type. It is also possible to immediately assign a value to a variable, see Listing 30 for an example;

Listing 30: Variable declarations

```
# variable declarations
{string} mySet
[int | float] numbers

# variable declarations with assignment
int i = 55
string j = "Hello there!"
[int] list = [1,2,3]
{int} m = {1,4,8,1000,5}
(int,float) j = (1, 2.0)
{int:string} d = {1:"hello"}

#assignments
mySet = {"one","two","three"}
numbers = [1, 2, 3, 4.5, 6.999, 100]
```

2. **Type declarations** that allow the user to define custom types, see Listing 31.

6.3.3. While

A `while` loop is the first loop statement in the HUTN. The loop stops if the loop condition returns false. An example, which increments an integer until it has reached 10, is given below:

Listing 31: Type declarations

```
# type declarations
type StringOrInt = String | Int
type SpecialList = [StringOrInt]

# variable declaration using the custom type
SpecialList my_special_list = [1, "hello"]
```

```
int i = 0
while(i < 10):
  i = i+1
```

6.3.4. For

The `for` loop is the second loop statement in the HUTN and is an iterator-based for loop, which means it will iterate a collection (i.e. iterable).

```
[int] list = [1,2,3,4,5,6]
for i in list:
  print i
```

6.3.5. Conditions

Conditional statements are expressed using the if-elif-else structure:

```
string j = "hello"
if(j == "hello"):
  print "hello world!"
elif(j == ""):
  print "nothing here.."
else:
  print "goodbye world!"
```

6.3.6. Return

The return statement can return a value of any type or even no value.

```
string b = "Jos"
if b == "hello":
  return
else:
  return True and b == "Jos"
```

6.3.7. Comments

It is also possible to add comments. HUTN supports single-line and block comments:

```
#single line comment
float f = 1.5 + 3 * 4  #another single-line comment

/*
 block comment
*/
```

6.3.8. Exceptions

Exceptions are very similar to the exceptions of Python, especially the structure to catch exeptions `try-except-else-finally`. The `else` and `finally` clause are optional. The latter is always executed at the end, while the `else` clause is only executed if no exception occured. Exceptions in ArkM3 are classes (or actually clabjects), in this way an exception inherits all the properties of a class. For instance, it is possible to specify exceptions consisting of attributes and constraints. Listing 32 gives a small example of the exceptions.

Listing 32: Exceptions in the HUTN

```
package atompmTest.testExceptions
 exception MyException
  int d = 1
  string a

 exception MyException2(MyException)
  int e = 2
  string a

 action a:
     int bla

  try:
    bla = 7
    raise MyException()
  except MyException as d:
    bla = 8
  except (MyException2,MyException) as e:
    bla = 9
  except:
    bla = 10
  else:
    bla = 11
  finally:
    bla = 12
```

6.3.9. Basic Operators

The following operators are supported:

- Boolean operators, like `==, !=, <, >, >=, <=, and, or, not`;
- Arithmetic operators, like `+, -, *, /, %`;
- Index operators and slice operators, an example is given below.

```
1  [int] list
2  list.append(1)
3  list.append(2)
4  list.append(3)
5  list.append(4)
6  print list[0] # prints 1
7
8  list = list[:2] # slice of list = [1,2,3]
9  list.append(5)
10
11 [[int]] doubleList
12 doubleList.append(list)
13 print list[0] # prints list 1,2,3
14 print list[0][0] # prints 1
15 print list[0][1] # prints 2
16 print list[0,1] # this is the same as previous line
```

Each type has its own operations, this paragraph illustrates the available operations for a string, set {}, sequence `[]` and map {`:`}. Optional parameters are in italic. The operations are influenced by the existing Python operations[8].

String operations:

- `int string.count(string sub)`
  Count the number of times a substring occurred in a string;
- `int string.find(string sub)`
  Find a substring and return the index;
- `int string.rfind(string sub)`
  Same operation as `find`, but from right to left;
- `string string.replace(string old, string new ,` *`int limit`*`)`
  Replaces a substring by a new string and has an optional limit;
- `string string.strip(`*`string sub`*`)`
  Return a copy of string, where leading and trailing sub string is removed;
- `string string.lstrip(`*`string sub`*`)`
  Return a copy of string, where leading sub string is removed;

[8]Python Types documentation: `http://docs.Python.org/library/stdtypes.html`

- `string string.rstrip(`*`string sub`*`)`
  Return a copy of string, where trailing sub string is removed;
- `[string] string.split(string sep ,` *`int limit`*`)`
  Return a sequence of the words of a splitted string, where sep is the separator;
- `[string] string.rsplit(string sep ,` *`int limit`*`)`
  Same operation as `strip`, but starting from the right;
- `string string.lower()`
  Return a copy of string where all letters are converted to lower case;
- `string string.upper()`
  Return a copy of string where all letters are converted to upper case;
- `string string.swapcase()`
  Return a copy of string where all letters are swapped from lower to upper case and vice versa;
- `string string.title()`
  Return a copy of string where each first letter of a word is upper case;
- `string string.capitalize()`
  Return a copy of string where the first letter of the string is converted to upper case.
- `int seq.size()`
  Return the size of the string;

Set operations:

- `void set.add(any e)`
  Add an element to a set;
- `void set.remove(any e)`
  Remove an element of a set;
- {`any`} `set.union(set e)`
  Return a copy of the union of the two sets;
- {`any`} `set.difference(set e)`
  Return a copy of the difference of the two sets;
- {`any`} `set.symm_diff(set e)`
  Return a copy of the symmetric difference of the two sets;
- {`any`} `set.intersect(set e)`
  Return a copy of the intersection of the two sets;

- `void set.union_update(set e)`
  Update the current set to the union of the two sets;
- `void set.difference_update(set e)`
  Update the current set to the difference of the two sets;
- `void set.symm_diff_update(set e)`
  Update the current set to the symmetric difference of the two sets;
- `void set.intersect_update(set e)`
  Update the current set to the intersection of the two sets;
- `bool set.contains(any e)`
  Return a boolean indicating if a given element i is in the current set;
- `bool set.issubset(set s)`
  Return a boolean indicating if a given set is a subset of the current set;
- `bool set.issuperset(set s)`
  Return a boolean indicating if a given set is a superset of the current set;
- `int set.size()`
  Return the size of the set.

Sequence operations:

- `void seq.append(any e)`
  Append an element to the sequence;
- `void seq.extend(seq e)`
  Extend the sequence with another sequence;
- `void seq.insert(int index, any e)`
  Insert an element at a given index position;
- `void seq.delete(any e)`
  Delete an element in the sequence;
- `void seq.delete_index(int i)`
  Delete an element in the sequence using an index;
- `any seq.pop()`
  Pop the first element of a sequence;
- `int seq.count(any e)`
  Count the amount of times an element occurs in the sequence;
- `any seq.get(int i)`
  Get an element at index i;

- `int seq.size()`
  Return the size of the sequence;
- `bool seq.contains(any e)`
  Return a boolean indicating if a given element occurs in the sequence;
- `void seq.reverse()`
  Reverse the sequence;
- `void seq.sort()`
  Sort the sequence.

Map operations:

- {any} `map.get_keys()`
  Return a set containing the keys of the map;
- {any} `map.keys()`
  Return a set containing the keys of the map;
- {any} `map.get_values()`
  Return a set containing the values of the map;
- {any} `map.values()`
  Return a set containing the values of the map;
- `any map.get(any key)`
  Return an element of the map using a key;
- `void map.set(any key, any value)`
  Set an element of the map.
- `int seq.size()`
  Return the size of the map;

A small example using a couple of sequence functions:

```
1 [any] l  =[1,2,[5,3]]
2
3 l.append(99)
4 l.remove(1)
5 l.extend([1,2])
6 l.insert(0,808)
```

Other basic functions are:

- `print` to print out expressions in the console.
  Usage: `print "hello";`
- `floor`, `ceil` to round a number down.
  Usage: `floor(1.9);`

- `abs` to get the positive number of a value.
  Usage: `abs(-1.8);`
- `round` to round a number.
  Usage: `round(1.7);`
- `in` returns True if a certain value is in the collection (set, map, sequence).
  E.g. `1 in [1,2,3]` returns True;
- `len` to get the length of a collection.
  Usage: `len([1,2,3]])`.

6.3.10. Type Casting

Another feature of the HUTN is type casting. This particularly useful to cast an instance to one of its subclasses. For instance, the meta-model below defines some animals and a zoo.

```
1  package animalpark
2    class Animal
3
4    class Zoo
5      [Animal] animals
6
7    class Bear
8
9    class Tiger
10     int special_attribute = 0
```

Suppose one wants to loop over the animals and print the `special_attribute` of the `Tiger` class. Then casting is needed to cast an instance of animal to an instance of `Tiger`. Otherwise the attribute cannot be accessed. An example of type casting is given below:

```
1  Zoo myZoo
2  myZoo.animals.append(new Tiger)
3
4  for animal in myZoo.animals:
5   # type casting
6   Tiger t = (Tiger) animal
7   print t.special_attribute
```

6.3.11. Model Manipulation Operators

Last but not least the model manipulation operators are handled. First of all, the CRUD operations are discussed. To create instances, the keyword `new` is used. Each time a new instance is created, it is added to the current package. Reading, updating or navigating a model element is done by defining the path separated by dots. Finally the `delete` keyword is used to delete an element in

the current package. For instance, `animalpark.Zoo` refers to the `Zoo` class in package `animalpark`. An example of creating 100 tigers and deleting 1 tiger using the previous zoo example is given below:

```
1  package test
2    action tigerpark
3      animalpark.Zoo myZoo = new animalpark.Zoo
4      int i = 0
5      while i < 100:
6        animalpark.Tiger t = new animalpark.Tiger
7        myZoo.animals.append(t)
8        i = i + 1
9
10     delete myZoo.animals.pop()
```

Using the meta-model of ArkM3, it is possible to get an attribute using a string:

```
1      myZoo.get_PropertyValue("animals").get_value()
```

Other semantic model operators which are supported in HUTN:

- `elem.get_all_instances()` gets all instances of an element type;
- `elem1.is_kind(elem2)` checks if an element is an instance of an element or one of it subclasses;
- `elem1.is_instance(elem2)` checks if an element is an instance of an element;
- `elem1.is_identical(elem2)` checks if two elements are identical.

6.4. Interface of HUTN class

The `HUTN` class has four important functions and they all return the ArkM3 object that is parsed:

1. `parse(string)`
   parses a string which conforms to the HUTN;
2. `parseFile(filename)`
   parses a HUTN file;
3. `parseAction(string)`
   parses a string which conforms to the Action language of the HUTN;
4. `parseActionFile(filename)`
   parses an Action language file;
5. `compileFile(filename)`
   parses a HUTN file and generates Python code.

7. Implementation of HUTN

This section will explain the implementation of the HUTN. The programming language that is used is Python and after comparison of different compiler-compilers [7] PLY [8] was used to construct the parser and lexer.

7.1. Code Structure

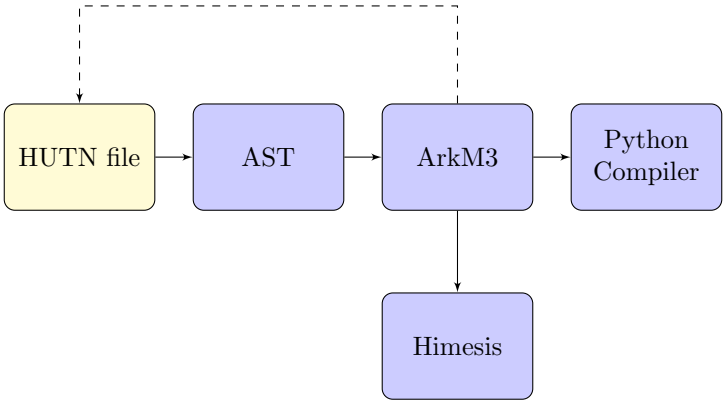

Figure 20: HUTN Overview

Figure 20 gives a good overview of what happens when a HUTN file is parsed. The process will be briefly described in a number of steps:

1. An input file is parsed using PLY to construct an Abstract Syntax Tree. Syntax errors and illegal characters will be recognized in this process;
2. The AST is visited by an AST Visitor. This visitor will visit all the nodes of the AST, detects type errors and generates the ArkM3 structure. It has two functions:
   i. Populating the metaverse by using the ArkM3 interface which is responsible for CRUD operations on the metaverse. Elements in the metaverse are the elements of the ArkM3 Object package: packages, classes, associations, compositions, instances, actions, contraints and properties;
   ii. The elements belonging to the Action Language, i.e. the body of actions, constraints or functions, are mapped to the corresponding ArkM3 classes.
3. Every ArkM3 class has a Himesis representation (developed by Simon Van Mierlo);
4. The execution of the models is achieved by mapping the ArkM3 structure to executable Python code.

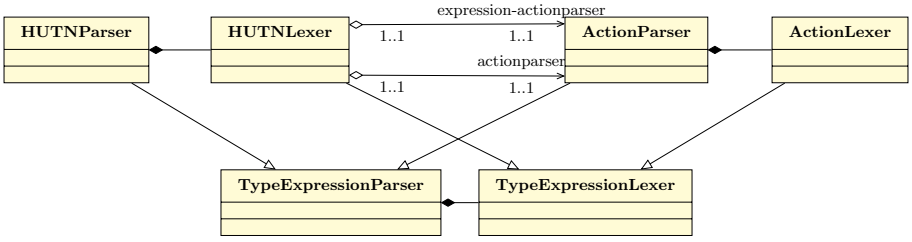

Figure 21: Parser and lexer inheritance

The dashed lines indicates pretty printing. Using a visitor on the ArkM3 tree, the appropriate HUTN input file could be generated. This is useful for both testing purposes as for the transformation of graphically defined ArkM3 models (which have no textual notation). At this moment there is only pretty printing from the ArkM3 tree to the HUTN, but in the future it is useful to generate Javascript files too. Because Javascript is already used in the current version of AToMPM and users who use the HUTN could check if their HUTN file corresponds to the Javascript file.

7.2. Lexing and Parsing of HUTN File

For this project PLY is used as parser-generator, which was already introduced in Section 1.2.2. This section will describe some noteworthy aspects of the parsing process.

7.2.1. Parser Inheritance

Section 1.2.2 discussed the definition of a parser and lexer class using PLY. The class diagram for the HUTN parser and lexer is given in Figure 21. The separation of the parsers (and lexers) corresponds to three different packages of ArkM3, see Section II. There are three parsers, i.e. `HUTNParser`, `ActionParser` and `TypeExpressionParser`. They correspond respectively to the Object Language, Action Language and Data Type package. The `TypeExpressionParser` is the superclass of the other two parsers as these two parsers need to parse type expression. Due to inheritance, one should note that a parser can override methods of its superclass. This means overriding grammar rules that can change the grammar and the behaviour of the parser.

Note that parser inheritance is not the same as island grammars (see Section 1.2.1). Parser inheritance implies a single parser with inherited grammar rules, which eventually can be overridden by the subclass parser. The concept of island grammars defines two (or more) separate parsers.

7.2.2. Island Grammars

The HUTNLexer in Figure 21 has two attributes, namely `expression_actionparser` and `actionparser`. The former is an instance of `ActionParser`, which only allows single-line expressions. This parser is used for the

assignment of attributes. The latter allows all Action Language constructs and is the normal instance of the `ActionParser`. In contrast to the `TypeExpressionParser`, which is inherited by the `HUTNParser`, the two action parsers are attributes of the `HUTNLexer`. The action parsers are attributes, because they are used as island grammars and are invoked when an action language construct occurs. This occurrence is noticed by the `HUTNLexer` using lexer states.

Lexer States

PLY provides a feature where lexers can have different states. Lexer states can change the initial lexing behaviour, i.e. each state can have its own tokens and lexer rules. In this way, a certain token can, for instance, trigger a different kind of lexing. There are two types of lexer states: inclusive and exclusive. An inclusive state adds additional tokens and rules to the lexer. An exclusive state completely overrides the default behaviour. The lexer will only apply rules and tokens defined for that state. Exclusive lexer states are used within the `HUTNLexer` and the basic steps of the lexer are:

1. The `HUTNLexer` starts in the normal lexer state;
2. If the `HUTNLexer` identifies the start of action code, then the lexer will change to an exclusive state named `actioncode`;
3. When the `HUTNLexer` is in the exclusive `actioncode` state, the lexer only uses rules defined for this state. These rules should invoke the appropriate `ActionParser`;
4. When the `ActionParser` is finished, the `HUTNLexer` state returns to the normal state and the lexer continues.

Identify Start of Action Code (step 2)

There are two cases when the Action Language could occur. Firstly, the body of an action, constraint or function. Secondly, the assignment of a value/-expression to an attribute. In the first situation, the Action Language body is always preceded by a colon. So when the lexer matches a colon token, it can change his state to `actioncode`. There is only one remark, the colon must be placed at the end of the current line (ignoring whitespaces at the end). Listing 33 is an extract of the HUTNLexer where the first function matches the lexemes with a colon at the end of the line and changes the state to `actioncode`. The second function describes the normal behaviour (if the colon is not at the end of the line).

Listing 33: Change lexer state for action body

```
1      def t_COLON(self,t):
2          r':(?=\s*$)'
3          self.change_state('actioncode')
4          return t
5
```

```
6       def t_COLON_normal(self,t):
7           r'\:'
8           t.type = 'COLON'
9           return t
```

The second case was the assignment of a value to an attribute. In this case the Action Language is preceded by an equals operator. When the lexer matches the equals operator it should change the state, except when it is followed by a colon on the same line. This is due to function definitions, where the equals operator is followed by the parameter types and then the colon with function body (e.g. `f = int i:  void:`).

Listing 34: Change lexer state for action assignment

```
1       def t_EQ(self, t):
2           r'=(?!.*\:\s*$)'
3           self.change_state('actioncode')
4           return t
5
6       def t_EQ_normal(self,t):
7           r'='
8           t.type = "EQ"
9           return t
```

Invoke the Appropriate ActionParser (step 3-4)

When the lexer state is changed to the exclusive state `actioncode`, only lexer rules for this state will be reachable. The `HUTNLexer` needs two lexer rules in accordance with the two different `ActionParser` instances. One rule for the single-line expression `ActionParser` (attribute assignment) and another rule for the normal `ActionParser` (action, constraint, function). Listing 35 defines the two lexer rules. Remark that each lexer rule has a prefix `actioncode`. The prefix indicates that these functions are lexer rules for the exclusive state `actioncode`.

The first function is the lexer rule for the single-line expression `Action-Parser`, i.e. `t_actioncode_expressionaction`. The aim of this function is to parse an expression of an attribute assignment. It should invoke the appropriate `ActionParser` and parse the expression using this parser, which will return an AST. This AST is assigned to the token value. In this way, the `HUTNParser` does not have to create the AST of the expression.

On line 2 in Listing 35 the pattern of the token is defined as a regular expression. This regular expression matches all characters (except new line) after the assignment symbol "=". The type of the token is set to `ACTION_ISLAND-_GRAMMAR`, which will be used later in a parser rule of the `HUTNParser`. The lexeme `g.value`, which represents the matched expression, is passed to the `ActionParser` that will return the AST. The AST is assigned to token value. Finally, the lexer state is changed back to its normal `INITIAL` state. Note

Listing 35: Invoke the Appropriate ActionParser

```
1       def t_actioncode_expressionaction(self, g):
2           r'(?<==).*'
3           g.type = 'ACTION_ISLAND_GRAMMAR'
4
5           self.eap.clear()
6           parsed = self.eap.parse(g.value.lstrip(),lineno=g.lexer
                .lineno)
7           g.value = parsed
8           self.change_state('INITIAL')
9
10          return g
11
12      def t_actioncode_normalaction(self, g):
13          r'(?<=\:)\s*'
14          g.type = 'ACTION_ISLAND_GRAMMAR'
15          self.ap.clear()
16
17          # search start of action code
18          lexeme = g.lexer.lexdata[g.lexpos:]
19          stripped_lexeme = lexeme.lstrip("\t").lstrip(" ")
20          pos = g.lexpos + (len(lexeme) - len(stripped_lexeme))
21          # parse
22          parsed = self.ap.parse(stripped_lexeme,lineno=g.lexer.
                lineno)
23          g.value = parsed
24          # if multiple scopes were ended in action parser
25          self.lexer.lineno = self.ap.clex.lexer.lineno - 1
26          if self.ap.clex.latest_endscope != -1:
27                  self.lexer.lexpos = pos + self.ap.clex.
                        latest_endscope.lexpos
28          else:
29                  self.lexer.lexpos = self.ap.clex.lexer.lexpos -
                         1
30          self.change_state('INITIAL')
31
32          return g
```

that, in order to maintain the line numbers between the different parsers, the line number of the current `HUTNLexer` is passed on to the `ActionParser`. Another remark is that the leading whitespace characters of the lexeme, i.e. `g.value`, are stripped before parsing using `lstrip`.

The second function is the lexer rule for the normal `ActionParser`, i.e. `t_actioncode_normalaction`. The aim of this rule is to parse Action Language constructs used within constraints, actions and functions (statements, expressions, etc.). The implementation of this function is approximately similar to the first function. This time, the regular expression only matches the start position of the action code language, i.e. right after the colon symbol "`:`". The `ActionParser` starts parsing from this point until the end of the action code. The starting point must be a new line character, since the `ActionParser` uses the scope, i.e. indentation size, of the action to determine the end of the action code. Due to the new line character, only leading tabs and spaces are stripped before parsing. The last difference corresponding to the previous rule is that the lexer position of the `HUTNLexer` should be changed to the position of the `ActionLexer`.

The two lexer rules in `HUTNLexer` produce a token of type `ACTION_ISLAND_GRAMMAR`. This token type is used within the `HUTNParser`. Listing 36 shows two parser rules where the `ACTION_ISLAND_GRAMMAR` token is used. The value of this token is the AST produced by the `ActionParser`.

Listing 36: Parser rules in `HUTNParser` for attribute assignment and action

```
1     def p_attribute_instance(self, p):
2         'attribute_instance : ID EQ ACTION_ISLAND_GRAMMAR'
3         p[0] = AttributeInstance(name=p[1], value=p[3], lineno=
              p.lineno(1))
4
5     def p_action_language_action(self, p):
6         'action_language_term_action : ACTION ID COLON
              ACTION_ISLAND_GRAMMAR'
7         p[0] = Action(p[2], p[4], p.lineno(1))
```

7.2.3. Tab or indent handling in PLY

A Python-like language uses indents to define the different scopes instead of curly brackets. The word indent is used instead of tab, because a tab is a character and the indentation of scope consists of tabs and spaces. The recognition of the beginning and ending of a scope is done in the lexer. The lexer class has two attributes: a scope and tab_width. The latter defines how many spaces a tab is worth, in this case it is four. In other words, four spaces are equal to one tab for the lexer. Scopes are defined by calculating the indentation size. For instance, if the previous line had an indentation of four spaces and the current line has an indentation of 6 spaces, then a new scope is defined.

The lexer has an attribute `scope`, which is a stack, and stores the previous indent sizes (i.e. the previous scopes). When the lexer recognises a new line, it starts calculating the indentation size by counting the tabs and spaces. If the size of the current indent is calculated (remember that a tab is the equivalent of four spaces), then it checks the previous scope. Now there are three possibilities:

1. The current scope has an indentation which is the same as the current indentation, this means the lexer is still in the same scope;
2. The current scope has an indentation which is smaller than the current indentation, this means the previous scope is closed;
3. The current scope has an indentation which is greater than the current indentation, this means a new scope has started.

If a new scope is started, the lexer emits a `BEGIN_SCOPE` token to identify the new scope and the new indentation size is pushed on to the scope stack. Afterwards the lexer moves on.

If the previous scope is closed, the previous indentation is popped from the scope stack and the lexer emits an `END_SCOPE` token. Now the lexer needs to check if the indentation which is now on top of the stack is not greater than the current indentation. If this is not the case, the lexer can move on. Otherwise this means that another scope is also closed and the lexer needs to emit another `END_SCOPE` token. In this case, PLY causes trouble. A lexer cannot emit two tokens in the same lexer rule. In order to circumvent this problem, the lexer's position in the current input text is changed to the previous position. Now the lexer will emit one `END_SCOPE` token and will return to the same lexer rule. The same rule will then recognise that a scope is closed and start all over again until all scopes are closed. The lexer rule described in this paragraph is given in Listing 37. The regular of expression of this lexer rule seems a little bit odd, due to the different symbols for a new line in operating systems. A new line symbol in Windows is `\r\n`, in UNIX this is `\n` and sometimes `\r` is used as a new line. The regular expression `((\r\n) | \n | \r)` covers these three types. The second part of the expression `[\t\\]` matches tabs and spaces.

Listing 37: Lexer rule for handling indents

```
1       def t_NLTAB(self, t):
2         r'(( (\r\n) | \n | \r )[\t\\ ]*)+'
3
4         # the regular expression above matches the beginning of
              a newline
5         # count the indentation size
6         t.lexer.lineno += self._count_lines(t.value)
7         tabsize = self._count_tabsize(t.value)
8         t.type = 'NL'
9         empty = self.scope.empty()
10
```

```
11          # scope indentation < current
12          if((empty and tabsize > 0) or (not(empty) and self.
                scope.peek() < tabsize)):
13              self.scope.push(tabsize)
14              t.type = 'BEGIN_SCOPE'
15
16          # scope indentation > current
17          elif(not(empty) and self.scope.peek() > tabsize):
18              self.scope.pop()
19              # if multiple scopes are ended, go back and
                    generate a new END_SCOPE token
20              if(not(self.scope.empty()) and self.scope.peek() >
                    tabsize):
21                  self.helper_rollback_one_token(t)
22              t.type = 'END_SCOPE'
23          return t
```

7.2.4. Type Expression Parser

In this section the grammar of the type expression parser is discussed. The grammar definition of the type expression parser using EBNF-like syntax is given below.

```
1  typeExpression      :=  ['immutable'] typeExpressionSpec
2  typeExpressionSpec  :=  primitiveTypeExpr | compositeTypeExpr
3  primitiveTypeExpr   :=  IntegerType | RealType | BooleanType |
       StringType | AnyType | VoidType | typeName
4  compositeTypeExpr   :=  UnionType | SetType | SequenceType |
       MappingType | ProductType | BracketedType
5  UnionType           :=  typeExpression '|' typeExpression [...]
6  SetType             :=  '{' typeExpression '}'
7  SequenceType        :=  '[' typeExpression [size] ']'
8  MappingType         :=  typeExpression ':' typeExpression
9  ProductType         :=  typeExpression '*' typeExpression [...]
10 BracketedType       :=  '(' typeExpression ')'
```

In this grammar typeName can be a custom type or the name of a metamodel. Listing 38 describes a part of the Type Expression parser. The grammar rules of the parser are defined in the docstring of the methods. Each method accepts a single argument p which represents a sequence containing the values of each grammar symbol in the corresponding rule. Listing 38 illustrates an excerpt of the Type Expression Parser.

Listing 38: Excerpt of the Type Expression Parser

```
1  def p_type_Expression(self,p):
2  'typeExpression : IMMUTABLE typeExpression'
3  #  p[0]             p[1]         p[2]
4  p[0] = TypeImmutable(p[2])
```

```
5
6  def p_type_ExpressionSpec(self,p):
7  '''typeExpression : primitiveTypeExpr
8  | compositeTypeExpr'''
9  p[0] = p[1]
10
11 def p_type_compositeTypeExpr(self,p):
12 '''compositeTypeExpr : SetType
13 | UnionType
14 | SequenceType
15 | MappingType
16 | ProductType
17 | BracketedType'''
18 p[0] = p[1]
19
20 def p_type_UnionType(self,p):
21 'UnionType : typeExpression VERTICAL_BAR typeExpression'
22 p[0] = TypeUnion(p[1],p[3],p.lineno(1))
23
24 def p_type_SetType(self,p):
25 'SetType :  L_ACCOL typeExpression R_ACCOL'
26 p[0] = TypeSet(p[2],p.lineno(1))
27
28 def p_type_SequenceTypeLimit(self,p):
29 'SequenceType : L_SQ_BRACKET typeExpression INT R_SQ_BRACKET '
30 p[0] = TypeSeq(p[2],size=p[3],lineno=p.lineno(1))
31
32 def p_type_SequenceType(self,p):
33 'SequenceType : L_SQ_BRACKET typeExpression R_SQ_BRACKET '
34 p[0] = TypeSeq(p[2],lineno=p.lineno(1))
35
36   [..]
```

7.3. Abstract Syntax Tree Visitor

The `HUTNParser`[9] builds an AST which is visited by the `AST_Visitor`. This visitor is influenced by the Python built-in ast.NodeVisitor, see Listing 39. First, the Python visitor is introduced and in the next section the `AST_Visitor` is discussed.

7.3.1. The Python Built-in ast.NodeVisitor

The NodeVisitor class has two methods: `visit` and `generic_visit`. The `visit` method takes the class name of the node and calls the method: `visit_classname`, where "classname" is the name of the node's class. The

[9] The `ActionParser` and `TypeExpressionParser` also build an AST

Listing 39: Python ast.NodeVisitor

```
class NodeVisitor(object):
  def visit(self, node):
    """Visit a node."""
    method = 'visit_' + node.__class__.__name__
    visitor = getattr(self, method, self.generic_visit)
    return visitor(node)

  def generic_visit(self, node):
    """Called if no explicit visitor function exists for a node
        ."""
    for field, value in iter_fields(node):
      if isinstance(value, list):
        for item in value:
          if isinstance(item, AST):
            self.visit(item)
    elif isinstance(value, AST):
     self.visit(value)
```

`generic_visit` method is called when the class does not provide a method for the appropriate class name. One drawback of this implementation is that it does not consider inheritance, this implies that one has to create a method for each subclass. However, it is possible to extend the `generic_visit` method to eliminate this drawback. The `generic_visit` method can search for methods for one of the node's subclasses.

An example of an ast.NodeVisitor

Assume there is visitor `v` and `v` visits the node root, which is an instance of the class Root. Then the visitor attempts to call method `visit_Root(node)`. If this method is not implemented in the visitor, then the `generic_visit` method will be executed on this node. For example, if the AST looks like Figure 22 and our visitor is the one defined in Listing 40, then the output will be: `visited root 1 999`.

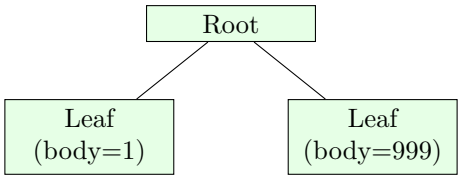

Figure 22: Simple Tree Example

Listing 40: Example of a Visitor

```
1  class Root(object):
2    def __init__(self,l,r):
3      self.left = l
4      self.right = r
5
6  class Leaf(object):
7    def __init__(self,x)
8      self.body = x
9
10 class Visitor(ast.NodeVisitor):
11     def visit_Root(self,node):
12       print "visited root"
13       a = self.visit(node.left)
14       b = self.visit(node.right)
15       print a, b
16     def visit_Leaf(self,node):
17       return node.body
18     [..]
```

7.3.2. The HUTN AST_Visitor

The AST Visitor differs from the Python built-in visitor, see Listing 41. The AST Visitor consists of two visit methods. The first method `visit` is a public visit method and resets all the attributes of the visitor. The second one `_visit` conforms to the method of the built-in NodeVisitor. Another difference between the AST_Visitor and the built-in NodeVisitor is the `generic_visit` method. This method throws an exception in the AST_Visitor, because it is only called when a particular type of the AST is not implemented.

The task of the AST Visitor is to map the AST tree to ArkM3 classes. Section 41 describes two functions. Firstly, the visitor maps the elements of the Action Language to the corresponding ArkM3 classes. Secondly, the ArkM3 interface is used to perform CRUD operations on the metaverse. An example of the former, where an if-else condition statement is mapped on to an ArkM3 class, is given in Listing 42. This method returns a `ConditionStmt` instance, which is an ArkM3 class. The ArkM3 interface has four functions corresponding to the CRUD operations. The first one, is the `create` function which has three parameters: `location`, `type` and `params`. The `create` functions creates an element at a given location in the metaverse, where the type of the element together with the parameters to construct the type are given. An example of a visit method that uses the ArkM3 interface is given in Listing 43. This listing illustrates the creation of packages and its elements. The variable `nameList` contains a list with the consecutive package names, where the first package name is the parent of the second package name and so on. For instance, a package `sub` is created within package `super` (syntax is `package super.sub`), then

Listing 41: AST_Visitor

```
1  class AST_Visitor(object):
2      def visit(self, node, debug=False):
3          self.clear()
4          return self._visit(node, debug)
5
6      def _visit(self, node, debug=False):
7          """Visit a node."""
8          if(debug):
9              self.logger.debug("Visiting %s" % node.__class__)
10         method = 'visit_' + node.__class__.__name__
11         self.history.append(node.__class__.__name__)
12         visitor = getattr(self, method, self.generic_visit)
13         return visitor(node)
14
15     def generic_visit(self, node):
16         self.logger.debug("Not implemented %s" % node.__class__
               )
17         self.logger.debug("Node: %s"% node)
18         self.logger.debug("History:")
19         for x in self.history:
20             self.logger.debug("%s"%x)
21         raise Exception("Visitor not implemented error for type
               : %s %s %s" % (node.__class__, node, self.history))
```

Listing 42: The visit method in the AST Visitor for a Condition Statement

```
1         def visit_ConditionStatement(self, node):
2           prevInsideStmt = self.helper_PreStmt()
3           self._symboltable_push()
4           ifbody = self.helper_SequenceValueVisitor(node.getIf())
5           self._symboltable_pop()
6           if(node.getElse() is None):
7               return self.helper_PostStmt(prevInsideStmt,
                    ConditionStmt(self._visit(node.getTest()),
                    ifbody, SequenceValue()))
8           else:
9               if isinstance(node.getElse(), ConditionStatement):
10                  self._symboltable_push()
11                  elsebody = self.helper_SequenceValueVisitor([
                        node.getElse()])
12                  self._symboltable_pop()
13              else:
14                  self._symboltable_push()
15                  elsebody = self.helper_SequenceValueVisitor(
                        node.getElse())
16                  self._symboltable_pop()
17              return self.helper_PostStmt(prevInsideStmt,
                    ConditionStmt(self._visit(node.getTest()),
                    ifbody, elsebody))
```

the first element of the list is `super` and the last element is `sub`. When all necessary packages of the `sub` package are created, in this case it is only the `super` package, all package elements are visited. Finally the visitor returns the package using the `read` function of the ArkM3 interface. This function reads an element of the metaverse using its location (separated by dots).

One last remark, concerning the AST Visitor, is that the AST Visitor uses a separate Type Expression Visitor for visiting types and type expressions.

7.4. Compiler to Python

In order to execute actions, constraints or functions, a `PythonCompileVisitor` is used. This visitor works in the same way as the `AST_Visitor`, except that it transforms ArkM3 elements to Python files.

7.4.1. Mapping from ArkM3 to Python

Each ArkM3 Package is mapped on to a Python package. Each Clabject, Association and Composition is a separate Python file within the Python package of its ArkM3 Package. The Python file contains its Actions, Constrains and Functions as Python functions. There is also a function within the file which create an instance of the corresponding ArkM3 element. Actions, Constraints

Listing 43: The visit method in the AST Visitor for a Package

```
1       def visit_Package(self, node):
2          # set current location to package
3          nameList = node.getName()
4          # self.current_scope.set(nameList)
5          # add corresponding packages to metaverse
6          location = ""
7          if self.current_scope.peek() != "":
8              location = "." + self.current_scope.peek()
9          for pname in nameList:
10             self.current_scope.push(pname)
11             self.metaverse.create(location[1:], ArkM3.PACKAGE,
                   pname)
12             location += "." + pname
13
14         # visit package contents
15         for i in node.getChildren():
16             self._visit(i)
17
18         for pname in nameList:
19             self.current_scope.pop()
20
21         location = []
22         if self.current_scope.peek() != "": location.append(
               self.current_scope.peek())
23         location.append(node.getName()[0])
24         return self.metaverse.read(".".join(location))._object
```

and Functions within a Package are written in a Python file within the corresponding Python package, where the name of the file is equal to the name of the package.

7.4.2. Execute

Actions, Constraints and Function can be executed. They all have a method `execute` which uses the execute method of the `PythonCompileVisitor`, see Listing 44. This method will link the ArkM3 element, i.e. `Action`, `Constraint` or `Function`, to the generated Python function. If the Python code is not yet generated, the `PythonCompileVisitor` will first generate the necessary Python code.

The execute function will first check if the ArkM3 element has the attribute `_hostElement`. This refers to the parent element. For instance: a package or a clabject. Then it will check if the `_execute` attribute is available. This is the function that is linked to the generated Python function. If it is available, then it is already linked. Otherwise we need to check if the generated Python files are present. This is done in the `try`-block. When the import fails, the Python files will be generated using the visitor and the appropriate Python function is linked to the `_execute` attribute.

Listing 44: Execute method of the PythonCompileVisitor

```
1     def execute(self, element, *args):
2         if not hasattr(element, "_hostelement") or element.
              _hostelement == None: raise Exception("Host Element
              is not set")
3         if not hasattr(element, "_execute"):
4             modulelist = self._parent_module.split(".")
5             modulelist.extend(element._hostelement.get_location
                  ().get_value().split("."))
6             from arkm3.Object import Package
7             if isinstance(element._hostelement,Package):
8                 modulelist.append(element._hostelement.get_name
                      ().get_value())
9             modulelist.append(element.get_name().get_value())
10
11            module = ".".join(modulelist[:-1])
12            fnames = modulelist[1:]
13            # import python files if available
14            try:
15                i = __import__(module,globals(),locals())
16                for attr in fnames:
17                    reload(i)
18                    i = getattr(i,attr)
19                element._execute = i
20            except:
21                # generate python files and import
```

```
22                 element._hostelement.accept(self)
23                 i = __import__(module,globals(),locals())
24                 for attr in fnames:
25                     reload(i)
26                     i = getattr(i,attr)
27                 element._execute = i
28         if isinstance(element._hostelement,Package):
29             return element._execute(*args)
30         else:
31             return element._execute(element._hostelement,*args)
```

8. Tests

During the development of the HUTN. Different aspects need to be tested to ensure the system is stable. The typical steps of test execution are described in [30]:

0. (a) determine the coverage goal, (b) unit testing;
1. Establish that the implementation under test is minimally operational by exercising the interfaces between it parts;
2. Execute the test suite and fix tests which do not pass;
3. Use a coverage tool to instrument the implementation under test;
4. Develop additional tests to exercise uncovered code;
5. Stop testing when the coverage goal is met and all tests pass.

There are different test strategies, but they should all combine responsibility-based test cases and implementation-based test cases. The former defines test cases from the view of responsibilities of a class. The latter looks at the implementation to provide test cases. Code coverage analysis is an example of implementation-based testing. Another example of implementation-based testing are test cases that trigger and test exceptions. The drawback of implementation-based testing is that coverage metrics is a guideline and does not imply the absence of faults. That is why they should be combined with responsibility-based test cases, which are more focused on the functionality, requirements and the contract of the class.

During testing there are also two kinds of intentions to test. The first one is fault-directed testing, which means tests should reveal faults through failures. A fault is a piece of missing or incorrect code, whereas a failure is the manifested inability of a program to function well. The second one is conformance-direct testing that demonstrates conformance to the requirements. These two intentions does not necessarily exclude each other. For instance, there can be a test case which is both fault- and conformance-directed.

8.1. Testing the HUTN

Python provides a built-in testing framework. It defines test cases that are used to group different tests together. These test cases are executed within a test suite. The test suite for HUTN consists of following test cases:

- ExceptionTest: tests the Exceptions of `ArkM3`;
- ParserTest: consists of tests that check if the input is correctly parsed;
- PrettyPrinterTest: tests if the Pretty Printer returns the correct output. PrettyPrintTest also consists of automated generated test cases;

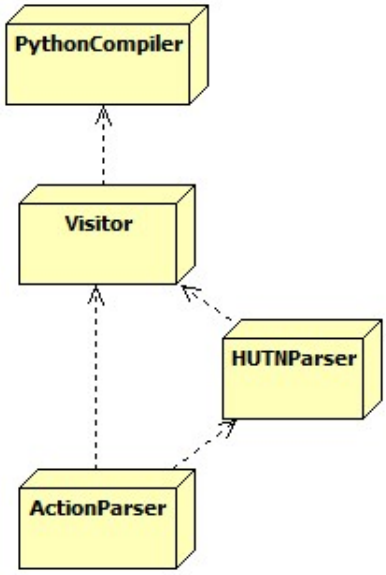

Figure 23: Dependencies bottom-up integration

- ActionVisitorTest: tests each construct of the Action Language using the `ActionParser` after they has been visited by the AST_Visitor. There are tests that trigger exceptions, in case of incorrect input. Other tests compare the generated ArkM3 objects and test if they are equal or not equal;
- VisitorIntegrationTest: checks if all the tests of the ActionVisitorTest are still running for the `HUTNParser`;
- VisitorTest: tests each construct of the Object Language using the `HUTN-Parser`;
- PythonCompileVisitorIntegrationTest: checks if all the tests of the ActionVisitorTest and VisitorTest are still running;
- PythonCompileVisitorTest: tests the execution of Actions, Constraints and Functions.

The code coverage for these tests is checked using the Python Code Coverage module [31].

8.2. Test pattern: Bottom-up Integration

The testing pattern, Bottom-up Integration, is applied during this project. This pattern is described in [32] and the intent is:

> "Demonstrate stability by adding components to the system under test in uses dependency order, beginning with components having the fewest dependencies" - Robert V. Binder

Using this pattern the components with the least number of dependencies are tested first. If all tests pass, then the components on the next higher level are tested. This process repeats until the root is reached and the entire system is exercised. The test model for this pattern is a dependency tree and the dependencies of the parsers are shown in Figure 23. First the Action Parser was tested, then the HUTN (or object) parser and finally the Python Compiler.

8.3. Examples of Different Test Types

In this paragraph, some examples are given of different test types.

- Trigger exception test, this test asserts that a certain exception is throwed. This useful to test the error detection of the parsers. The listing below parses a file, where a variable is used without its declaration. The parser should raise an Exception;

```
1      def testDeclarationNotFound(self):
2          with self.assertRaises(Exception) as cm:
3              self.parse("../hutn/test/files/action/
                   declarationnotfound.arkm3")
4          the_exception = cm.exception
5          self.assertTrue(str(the_exception).endswith("No
               declaration found for i"))
```

- Comparing abstract syntax trees and test whether they are equal or not equal. The listing below gives an example of an AST equal test;

```
1   def testDeclarationScope(self):
2          """
3          if True:
4              int i
5          else:
6              bool i
7          """
8          d = self.parse("../hutn/test/files/action/
               declarationscope.arkm3")
9          a = Action()
10         d1 = DeclarationStatement(
11                                                 IntegerType(),
12                                                 Identifier(
                                                       StringValue
                                                       ("i")))
13         d2 = DeclarationStatement(
14                                                 BooleanType(),
```

```
15                                                  Identifier(
                                                        StringValue
                                                        ("i")))
16          a.add_statement(
17                    ConditionStmt(test=self._literal(True),
18                                  ifbody=SequenceValue([d1]),
19                                  elsebody=SequenceValue([d2
                                        ])
20                                  )
21                    )
22
23          self.assertTrue(d.equals(a,debug=True))
```

- Oracles are equal or not equal. This used to test if the correct output was given. The code example below illustrates a test for the PrettyPrinter, where the input should be equal to the output;

```
1       def testPrintVisitor(self):
2           r = self.parse('../hutn/test/files/prettyprint/
                testType.arkm3', self.dp)
3           r.accept(self._pv)
4           oracle = file("../hutn/test/files/prettyprint/
                testType.arkm3", "r").read()
5           self.assertEqual(oracle, str(self._pv))
```

- Rerun the visitor tests for the Python Compiler.

```
1       def parse(self, fileName):
2           action = VisitorRegressionActionTest.parse(self,
                fileName)
3           pyfile = fileName.rsplit("/",1)[1].split(".")[0]+
                ".py"
4           cv = PythonCompileVisitor(filename = self.dir + os.
                sep + pyfile)
5           action.accept(cv)
6           cv.write()
7           self.runPython(cmd="Action()",pyfile = self.dir +
                os.sep + pyfile)
8           return action
9
10      def runPython(self, cmd = "", pyfile = None):
11          try:
12              f = open(pyfile)
13              string = f.read()
14              string += os.linesep + cmd
15              code = compile(string, '<string>', 'exec')
16              ns = {}
17              exec code in ns
```

Name	Statements	Missed Statements	Cover
Nodes.py	747	85	88,6%
DataTypeLexer.py	48	12	75%
DataTypeParser.py	62	6	90,3%
ActionLexer.py	160	13	91,9%
ActionParser.py	281	24	91,5%
HUTN.py	100	1	99%
HUTNLexer.py	170	17	90%
HUTNParser.py	174	9	94,8%
TypeMatcher.py	79	11	86,1%
TypeVisitor.py	125	18	85,6%
Visitor.py	867	166	80,9%
TOTAL	2813	362	87,1%

Table 3: Coverage for the HUTN parsers

```
18          except Exception as e:
19              traceback.print_exc(file=sys.stdout)
20              raise e
```

8.4. Coverage

The coverage of the three HUTN parsers and its visitors is given in Table 3. The coverage is not yet complete and some additional tests need to be added. The `Nodes.py` file contains the classes of the AST. This file is not completely covered because the equals function is not executed. The `TypeMatcher` is the class that checks if certain types conform to each other according to the Liskov substitution principle. All the other class files are classes that were already discussed in the previous section.

9. Extending the HUTN

There are various motivations for extending the HUTN. A first one is to add elements to the ArkM3 Action language for your particular models. For example, the addition of new timed operators like "crosses from below" or "crosses from above". These temporal operators return `True` if a certain variable crosses a certain value from below or above. To add this functionality not only the ArkM3 metamodel should be extended, but there should also be a textual notation which define the operators. Another motivation is to add syntactic sugar for custom models and build a small domain specific language by defining custom syntax. It is possible to extend different components of the HUTN (lexers parsers, visitors). In the next paragraph the different phases will be revealed to extend the HUTN. The example of the two operators ("crosses from below" and "crosses from above") will be used as a common thread. This are interesting operators, because they are timed. They not only look to their current value, but also at their previous value.

9.1. Extend the ArkM3 metamodel

In this step, the ArkM3 metamodel is going to be extended. The new operators need to be defined. These classes should extend the right ArkM3 classes. Because an operator is an action, the right class is located in the ArkM3 Action module, i.e. `ComparisonOp`. The `ComparisonOp` class is used as superclass for the `CrossesFromBelow` and `CrossesFromAbove` class, see Listing 45. Note that these classes extend the `ComparisonOp` class with an `self._id` attribute. This a unique identifier for each timed operator and is necessary to store and retrieve the previous value of the operator.

Listing 45: Two new classes extending the original ArkM3 metamodel

```
1  class ComparisonOp(BinaryOperator):
2
3      def __init__(self, child1 = None, child2 = None, container
           = None, init_id = None):
4
5          BinaryOperator.__init__(self, child1, child2, container
               , init_id)
6          self.set_type(BooleanType())
7
8  class CrossesFromBelow(Action.ComparisonOp):
9      def __init__(self, child1=None, child2=None, name=None,
           container=None, init_id=None):
10         Action.ComparisonOp.__init__(self, child1=child1,
               child2=child2, container=container, init_id=init_id)
11         self._id = name
12
13 class CrossesFromAbove(Action.ComparisonOp):
```

```
14      def __init__(self, child1=None, child2=None, name=None,
            container=None, init_id=None):
15          Action.ComparisonOp.__init__(self, child1=child1,
                child2=child2, container=container, init_id=init_id)
16          self._id = name
```

9.2. Extend the parsers and lexers

Next, the parser and lexer need to be extended. In the HUTN, there are two parsers. The HUTN parser and his corresponding lexer which is used to parse ArkM3 Object statements and on the other hand the Action parser and corresponding lexer which is used to parse ArkM3 Action statements. They are both extendable, but in case of operators only the Action parser and lexer need to be extended.

Just like the original parser and lexer, the extended version is written in Python using PLY. It is recommended that the original parser and lexer are super classes of the extended version. In this way all original properties are inherited. The textual notation for the `CrossesFromBelow` and `CrossesFromAbove` operator will be denoted as respectively `>!` and `<!`. These tokens are defined in the lexer. The extended parser contains the grammar rules. These rules map the expressions to intermediate AST nodes, which will be visited by the extended visitor in the next phase.

Listing 46: Extended parser and lexer

```
1  class Extended_Lexer(ActionLexer):
2    tokens = ActionLexer.tokens + ['CROSSES_BELOW', '
         CROSSES_ABOVE', 'INSTATE']
3
4    def t_CROSSES_BELOW(self, t):
5      r'\>!'
6      return t
7
8    def t_CROSSES_ABOVE(self, t):
9      r'\<!'
10     return t
11
12
13 class Extended_Parser(ActionParser):
14
15   precedence = (('nonassoc', 'CROSSES_BELOW', 'CROSSES_ABOVE')
         ,) + ActionParser.precedence
16
17   def p_expr_crosses_below(self, p):
18     'expr : expr CROSSES_BELOW expr'
19     p[0] = ASTCrossesFromBelow(p[1], p[3], p.lineno)
20
21   def p_expr_crosses_above(self, p):
```

```
22      'expr : expr CROSSES_ABOVE expr'
23      p[0] = ASTCrossesFromAbove(p[1], p[3], p.lineno)
24
25
26  class ASTCrossesFromBelow(ASTBinaryOperator):
27      def __repr__(self):
28          return "%s >! %s" % (repr(self.lhs), repr(self.rhs))
29
30  class ASTCrossesFromAbove(ASTBinaryOperator):
31      def __repr__(self):
32          return "%s <! %s" % (repr(self.lhs), repr(self.rhs))
```

9.3. Extend the visitors

HUTN contains two visitors: a TypeVisitor that handles type expressions and a general AST_Visitor that actually visits all the other nodes in the AST (Action and Object). These two visitors are both extendable and the TypeVisitor is called when the AST_Visitor encounters an TypeExpression node in the tree. For the current example it suffices to extend the AST_Visitor. Note that it is also required to have the original visitor as a superclass of the extended visitor and that for each node the visitor calls the method `visit_` appended by the node's classname. The extended visitor needs to visit the two AST classes, i.e. `ASTCrossesFromBelow` and `ASTCrossesFromAbove`, see Listing 47.

Listing 47: Extended visitor

```
1   class Extended_Visitor(AST_Visitor):
2     def __init__(self, debug=None, typevisitor=None,
          pythonvisitor=None):
3       AST_Visitor.__init__(self, debug=debug, typevisitor=
            typevisitor, pythonvisitor=pythonvisitor)
4       self.crosses = set()
5
6     def _generate_crosses_id(self):
7       id_length = 6
8       id = self.metaverse._id_generator(id_length)
9       while id in self.crosses:
10          id_length += 1
11          id = self.metaverse._id_generator(id_length)
12      self.crosses.add(id)
13      return id
14
15    def visit_ASTCrossesFromBelow(self, node):
16      id = StringValue(self._generate_crosses_id())
17      return CrossesFromBelow(self._visit(node.getLhs()), self.
            _visit(node.getRhs()), id)
18
19    def visit_ASTCrossesFromAbove(self, node):
```

```
20      id = StringValue(self._generate_crosses_id())
21      return CrossesFromAbove(self._visit(node.getLhs()), self.
            _visit(node.getRhs()), id)
```

9.4. Extend the ArkM3-to-Python compiler

This is also a visitor, i.e. PythonCompileVisitor, which maps each ArkM3 class to Python code. Listing 48 illustrates the Python implementation of the timed operators.

Listing 48: Extended Python compile visitor

```
1  class Extended_Python_Visitor(PythonCompileVisitor):
2      def visit_CrossesFromBelow(self, node):
3          self._add_abs_import("hutn.extended.ExtendedParser")
4          self._write_chars("P_CrossesFrom.get_instance(%s).
               checkBelow("%node._id)
5          self.visit(node._child[0])
6          self._write_chars(",")
7          self.visit(node._child[1])
8          self._write_chars(")")
9
10     def visit_CrossesFromAbove(self, node):
11         self._add_abs_import("hutn.extended.ExtendedParser")
12         self._write_chars("P_CrossesFrom.get_instance(%s).
               checkAbove("%node._id)
13         self.visit(node._child[0])
14         self._write_chars(",")
15         self.visit(node._child[1])
16         self._write_chars(")")
17
18 class P_CrossesFrom(object):
19     instances = dict()
20
21     @staticmethod
22     def get_instance(instance_id):
23         cls = P_CrossesFrom
24         if cls.instances.has_key(instance_id):
25             return cls.instances[instance_id]
26         else:
27             i = cls()
28             cls.instances[instance_id] = i
29             return i
30
31     def __init__(self):
32         self.value = VoidValue()
33         self.prev = VoidValue()
34
35     def checkBelow(self, value, value2):
```

```
36          if self.prev == VoidValue():
37              self.prev = value
38              self.value = value2
39              return False
40          elif self.prev <= self.value and value > self.value:
41              self.prev = value
42              return True
43          self.prev = value
44          return False
45
46      def checkAbove(self, value, value2):
47          if self.prev == VoidValue():
48              self.prev = value
49              self.value = value2
50              return False
51          elif self.prev >= self.value and value < self.value:
52              self.prev = value
53              return True
54          self.prev = value
55          return False
```

9.5. Instantiate the extended HUTN

As a final step, the HUTN needs to know it uses extended classes. HUTN is the class which contains all the different parsers. In this step, the HUTN is instantiated with three arguments (i.e. the classnames) specifying the extended components (Extended_Parser, Extended_Visitor, Extended_Python_Visitor and Extended_Lexer), see Listing 49. This modification allows the textual notation to read and evaluate these new operators, see Listing 50. The complete extended HUTN can be found in Appendix C.

It should be noted that the constructor of HUTN supports more parameters to extend corresponding components. The possible parameters are listed in Table 4. This example showed a way to extend the ArkM3 metamodel. But in the extended AST visitor, it is also possible to create elements of your own metamodel using your own syntax that is defined in your extended parser. In this way you can create your own domain specific languages.

Listing 49: Extended HUTN Instance

```
1   hutn = HUTN(actionparser=Extended_Parser, visitor=
        Extended_Visitor, actionlexer=Extended_Lexer,
        pythonvisitor=Extended_Python_Visitor)
2   hutn.parseFile('files/testCrossesFrom.txt')
```

Parameter	Default parameter	Function
`hutnlexer`	HUTNLexer	Definition of tokens for the Object language (lexer).
`hutnparser`	HUTNParser	Definition of Object language grammar (parser).
`actionlexer`	ActionLexer	Definition of tokens for the Action language (lexer).
`actionparser`	ActionParser	Definition of Action language grammar (parser).
`visitor`	AST_Visitor	Definition of general AST visitor which creates the arkm3 structure.
`typevisitor`	AST_TypeVisitor	Definition of the type visitor who is responsible for the arkm3 types.
`pythonvisitor`	PythonCompileVisitor	Definition of the visitor for Python execution

Table 4: Parameters for extending the HUTN

Listing 50: testCrossesFrom.txt: Simple crosses from below/above example

```
1  package atompmTest
2    action a:
3     [int] temps = [10,20,30,40,50]
4     for i in temps:
5      if i >! 30:
6       print "crosses from above "+string(i)
7     temps.reverse()
8     for i in temps:
9      if i <! 30:
10      print "crosses from below "+string(i)
```

Part IV
Conclusion

10. Conclusion & Future Work

Textual meta-modelling tools get more and more important. They have become a good alternative for the graphical meta-modelling tools. AToMPM is still under development, but this Human Usable Textual Notation can be used to create your own ArkM3 modes. this thesis represents a first proposal for the HUTN. The HUTN provides three parsers: a `TypeExpressionParser`, an `ActionParser` and a `HUTNParser`. Each parser can be used separately. One can use the `TypeExpressionParser` to parse type expressions or the `ActionParser` to parse action code. The `HUTNParser` is used to parse HUTN files or strings. These three parser can be extended or even be replaced by other parsers. In this way users are able to modify the HUTN and create their own DSLs. The Python Visitor Compiler of the HUTN, which is responsible for evaluating the action code, could also be extended or replaced.

The HUTN provides all mandatory concepts compared to other textual meta-modelling tools. Like in EOL, CRUD operations are allowed. Users can write actions, constraints or functions that are attached to a class (context-defined) or in package (context-less). Like in metaDepth, all model elements in the HUTN are linguistic instances of `Clabject`. It is also possible to have multiple meta-levels in your model. Listing 51 illustrates this using the the ProductType example of the metaDepth paper, see Section 2.2.1.

Listing 51: Multiple meta-levels: ProductType example in HUTN

```
1  package product
2   class ProducType
3    float VAT
4    float price
5
6   ProductType Book
7    VAT = 7
8
9   ProductType CD
10   VAT = 1.5
11
12  Book mobyDick
13   price = 10
14
15  CD TheSuburbs
16   price = 16
```

In the future, a Domain Specific Language could be developed using the HUTN interface that is described in Section 9 Extending the HUTN. An extension that allows to import OCL constraints in the HUTN, like the `require` mechanism in Kermata, could be a possibility. Potencies, like in metaDepth, can be added to ArkM3 and the HUTN. There is also an issue concerning types, the primitive types should be singleton types. But at the moment this is impossible, because a singleton node has no Himesis presentation. Some other future work is a pretty printer that prints ArkM3 to Javascript, because AToMPM is written in Javascript. An editor for the HUTN with syntax highlighting and early error detection is also a nice feature for in the future.

11. Bibliography

References

[1] MetaCase, MetaEdit+.
URL http://www.metacase.com/products.html

[2] MSDL, AtoM3.
URL http://atom3.cs.mcgill.ca/

[3] J. de Lara, E. Guerra, MetaDepth.
URL http://astreo.ii.uam.es/~jlara/metaDepth/

[4] Triskell, Kermeta.
URL http://www.kermeta.org/

[5] H. Krahn, B. Rumpe, M. Schindler, Text-based Modeling (2007).

[6] X. Dong, Ark, the Metamodelling Kernel for Domain Specific Modelling (2010).

[7] J. Slowack, Comparison of compiler-compilers with Python support (2012).

[8] David M. Beazley, Python Lex-Yacc (2011).
URL http://www.dabeaz.com/ply/

[9] A. W. Appel, Modern compiler implementation in Java, 1997.

[10] J. D. Ullman, R. Sethi, M. S. Lam, A. V. Aho, Compilers: Principles, Techniques, & Tools - 2nd edition, 2007.

[11] C. N. Fischer, R. K. Cytron, J. Richard J. LeBlanc, Crafting a Compiler, 2010.

[12] T. Parr, The Definitive Antlr Reference: Building Domain-Specific Languages, 2007.

[13] D. Cook, GOLD Parsing System.
URL http://goldparser.org/

[14] J. Voss, Wisent: a Python parser generator.
URL http://www.seehuhn.de/pages/wisent

[15] SableCC, SableCC Documentation Page.
URL http://sablecc.org/wiki/DocumentationPage

[16] Terence Parr, ANTLRv3 (2012).
URL http://www.antlr3.org/

[17] D. Kolovos, L. Rose, R. Paige, The Epsilon Book: The Epsilon Object Language, 2012.

[18] D. Kolovos, L. Rose, R. Paige, The Epsilon Book, 2012.

[19] OMG, Object Constraint Language.
URL http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL

[20] OMG, Unified Modeling Language.
URL http://www.uml.org/

[21] D. Kolovos, R. Paige, F. A. Polack, The Epsilon Object Language (EOL), 2012.

[22] D. Kolovos, L. Rose, R. Paige, The Epsilon Book: The Epsilon Validation Language, 2012.

[23] OMG, OMG Meta Object Facility (MOF) Core Specification (2011).

[24] Eclipse, Eclipse Modeling Framework Project.
URL http://www.eclipse.org/modeling/emf/?project=emf

[25] J. de Lara, E. Guerra, Deep meta-modelling with METADEPTH.

[26] R. Johnson, B. Woolf, The Type Object Pattern.

[27] J. de Lara, E. Guerra, MetaDepth Documentation.
URL http://astreo.ii.uam.es/~jlara/metaDepth/Documentation.html

[28] J.-M. Jézéquel, O. Barais, F. Fleurey, Model Driven Language Engineering with Kermeta (2010).

[29] R. Mannadiar, H. Vangheluwe, AToMPM: A Tool for Multi-Paradigm Modelling.
URL http://msdl.cs.mcgill.ca/people/raphael/files/poster.pdf

[30] R. V. Binder, Testing Object-Oriented Systems, 2003.

[31] N. Batchelder, Python Code Coverage.
URL http://wiki.python.org/moin/CodeCoverage

[32] R. V. Binder, Testing Object-Oriented Systems: Bottom-up Integration, 2003.

12. Appendix

Appendix A. Comparison

Comparison of compiler-compilers with Python support

Jelle Slowack

Abstract

Writing a compiler from scratch is a lot of work and compiler-compilers or compiler generators address this problem. A compiler-compiler is a tool that uses a formal description of a language to generate a parser. Thereafter this parser could be used to build a compiler. This report focusses on three tools that generate python code or in other words where python code is the target language. The three tools are SableCC, ANTLR and PLY. Different criteria will be used to compare these tools: workflow, tree construction, visitor pattern, elegance, python support, tool support, documentation, island grammars and scalability.

Keywords: compiler-compiler, compiler generator, parser generator, ANTLR, SableCC, PLY, Python

Contents

1. Introduction

1.1. Compilers, Lexers and Parsers

> "To translate a program from one language into another, a compiler must first pull it apart and understand its structure and meaning, then put it together in a different way."
>
> Andrew W. Appel (Lexical Analysis)[1]

A compiler usually consists of a lexer and parser. The lexer pulls an input stream apart by breaking it into tokens. This process of forming tokens from an input stream is called tokenization. An example of tokens that are formed from an input stream that contains following expression `total = 4 + 8.2` is given below:

```
1 Identifier AssignmentOperator Integer PlusOperator Real
```

These tokens correspond to the input stream, namely:

- `Identifier` corresponds to `total`;
- `AssignmentOperator` corresponds to `=`;
- `Integer` corresponds to `4`;
- `PlusOperator` corresponds to `+`;
- `Real` corresponds to `8.2`.

This token stream is passed onto the parser. The parser has mainly two functions:

1. conformance checking: it checks if the token stream conforms to the definition of the language (ie. the syntax);
2. building a parse tree: this tree represents the syntactic structure of a token stream.

In this report, the language definition is described using a context-free grammar, this grammar defines the structure of the language by means of grammar rules (ie. productions). A grammar supporting the structure of previous expression (`total = 4 + 8`) is of the form:

```
1 assignment  ->  Identifier AssignmentOperator expr
2 expr        ->  number PlusOperator number
3 number      ->  Integer | Real
```

The example provides three productions: `assignment`, `expr` and `number`. A production starts with the name of the production on the left-hand side (eg. `assignment`). The right-hand side is the specification of the production. This specification has zero or more terminals and/or nonterminals. A terminal refers to a token type (eg. `Identifier`) and a nonterminal refers to a production rule that has the same name (eg. `number`).

A parser uses these grammars and his productions to do conformance checking and to build up a parse tree. The parse tree could be seen as the path of the parser through the grammar.

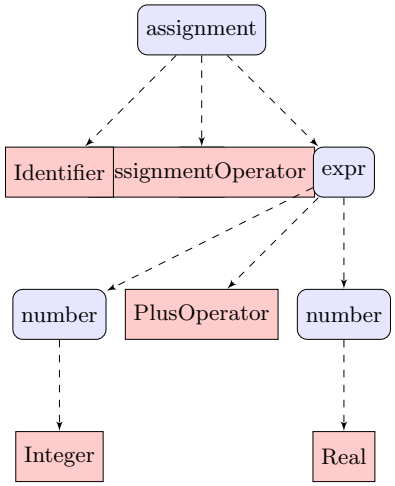

Figure 1: Parse tree for `total = 4 + 8.2` using the previous grammar

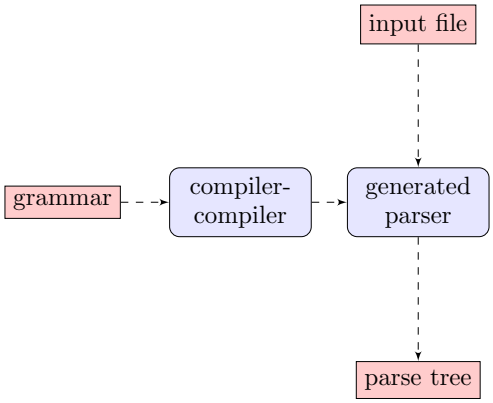

Figure 2: Schematic overview of a parser generator

1.2. Compiler-Compiler

Writing a compiler from scratch is a lot of work, because the programmer has to write the grammar and the code for the parser and lexer. Compiler-compilers or compiler generators address this problem. These tools use a formal language description to generate a compiler. In this description the user defines the tokens and productions. In this way, the programmer does not have to write the basic code for the parser and lexer. The most common form of a compiler-compiler is a parser generator. The output of a parser generator is the source code of the parser (and lexer). The input is a formal description that is defined in a grammar. The generated parser is able to parse an input file according to the syntax defined in the grammar. A schematic overview can be found in Figure 2.

1.3. Tree Construction

As mentioned in the previous paragraphs a parser generates a parse tree or in other words a syntax tree (because it represents the syntax). There are two kinds of syntax trees, namely a

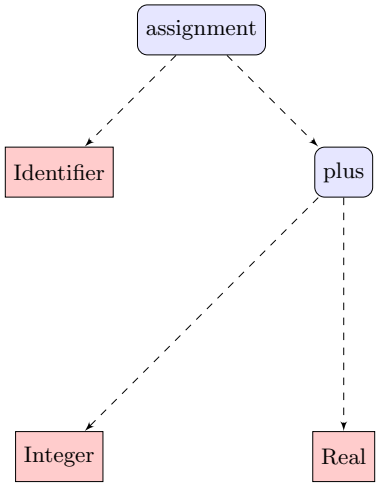

Figure 3: AST for `total = 4 + 8.2` using the previous grammar

concrete syntax tree (CST) and an abstract syntax tree (AST). A CST is a normal parse tree that conforms exactly with the syntactic structure of the grammar. An AST is more an abstract representation of the syntactic structure, ie. it does not represent every detail that appears in the real syntax. Figure 1 is the CST for the expression `total = 4 + 8.2` and Figure 3 is the AST.

1.4. Types of Parsers

There are essentially two types of parsing[2]:

1.4.1. Top-down parsers

Top-down parsers generate a parse tree by starting at the root of the tree (the start symbol), expanding the tree by applying productions in a depth-first manner. A top-down parse corresponds to a preorder traversal of the parse tree. The weakness of top-down parsing is its predictiveness, because parsers have to predict the production that is to be matched. LL parsers are examples of top-down parsers. The L stands for "Left to right" as the parser reads the input from left to right and the L stands for Leftmost derivation, this means the leftmost nonterminal[1] is always derived;

1.4.2. Bottom-up parsers

Bottom-up parsers generate a parse tree by starting at the tree's leaves and working toward its root. This technique is more powerful because the predictiveness is eliminated. These parsers only select a production if the entire right-hand side matches. A bottom-up parse corresponds to a postorder traversal of the parse tree. The most common bottom-up parsers are the shift-reduce parsers. The parser shifts symbols onto the parse stack and reduces a string of symbols located at the top of the stack to one of the grammar's nonterminals. The following example[2], in Listing 1 and Listing 2, explains the shift and reduce actions.

[1] Terminal symbols describe the input, while nonterminal symbols describe the tree structure behind the input.
[2] Example adapted from http://en.wikipedia.org/wiki/Bottom-up_parsing

Listing 1: Sentence Grammar

```
Sentence   := NounPhrase VerbPhrase
NounPhrase := Art Noun
VerbPhrase := Verb | Adverb Verb
Art        := 'the' | 'a'
Verb       := 'jumps' | 'sings'
Noun       := 'dog' | 'cat'
```

Listing 2: Bottom-up parsing using shift and reduce on input: "the dog jumps"

```
Stack                    Input Sequence
()                       (the dog jumps)
(the)                    (dog jumps)       SHIFT word onto stack
(Art)                    (dog jumps)       REDUCE using grammar rule
(Art dog)                (jumps)           SHIFT..
(Art Noun)               (jumps)           REDUCE..
(NounPhrase)             (jumps)           REDUCE
(NounPhrase jumps)       ()                SHIFT
(NounPhrase Verb)        ()                REDUCE
(NounPhrase VerbPhrase)()                  REDUCE
(Sentence)               ()                SUCCESS
```

LR parsers are examples of bottom-up parsers. The L stands for "Left to right" as the parser reads the input from left to right and the R stands for Rightmost derivation, this means the rightmost nonterminal is always derived. There are different types of LR parsers[2]:

- **SLR** parsers or Simple LR parsers have the simplest implementation. They do not have to scan through the possible reductions, because there is at most one reduction;
- **LALR** parsers are the intermediate form. They have smaller parse tables, because they reduce the amount of reductions. LALR parsers can handle more languages than SLR parsers and they are very efficient;
- **LR** parsers are the most powerful parsers. They can parse a larger set of languages compared with LALR and SLR, however they have much bigger parse tables which results in low efficiency.

1.4.3. Top-down (LL) vs. Bottom-up (LR)

A big difference between LL and LR grammars is that an LL grammar requires:

- **eliminating left recursion**: LL can't handle left recursion, because it will recurse forever due to the leftmost derivation;

Listing 3: Expression grammar using left recursion

```
Expr := Expr + Number
      | Expr - Number
      | Number
```

Listing 4: Expression grammar without left recursion

```
Expr := Number Expr_2
Expr_2 :=  + Number Expr_2? | - Number Expr_2?
```

- **factoring common prefixes**: this is required when two or more grammar rule choices share a common prefix string. This is necessary because LL parsing requires selecting an alternative based on a fixed number of input tokens. The `if-then-else`-grammar is a famous example that shares a common prefix string.

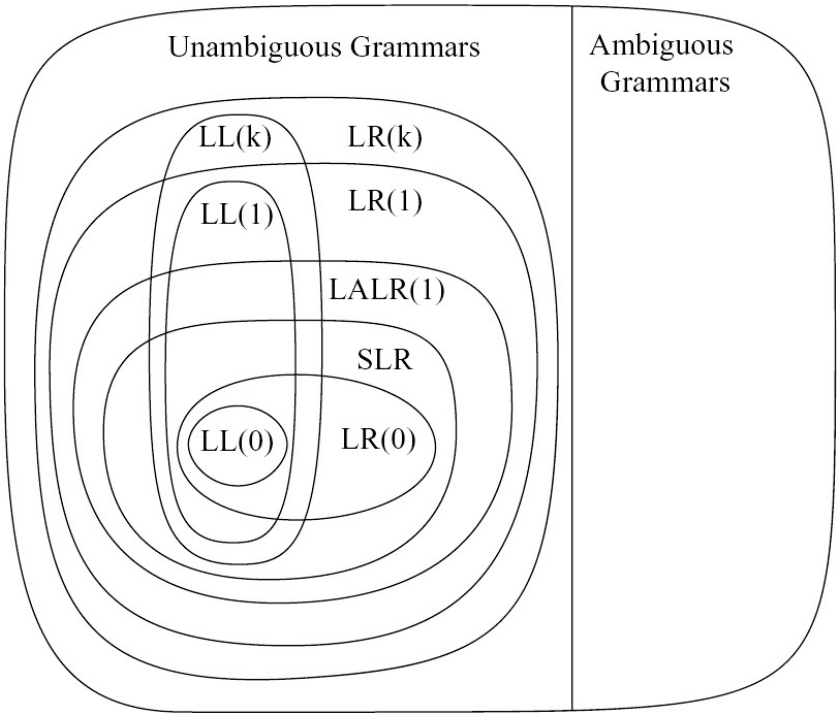

Figure 4: A hierarchy of grammar classes[1]

Listing 5: if-then-else grammar

```
1 S := if expr then S else S
2     | if expr then S
3     | other
```

Listing 6: The left-factored form of the if-then-else grammar

```
1 S := if expr then S E | other
2 E := (else S)?
```

LR parsing can handle a larger range of languages than LL parsing. Figure 4 shows the difference between the several grammar classes. The number (0,1,k) between parentheses denotes the lookahead. This is the amount of tokens that the parser can look ahead at the next tokens in order to decide what to do.

1.5. Island Grammars

There is also a special type of grammar that allows to embed another grammar in itself, this is called island grammars. For instance a HTML grammar should support the embedding of javascript code. In this case there are two grammars: a HTML grammar and a javascript grammar. The javascript grammar is embedded in the HTML grammar.

2. SableCC3

2.1. Workflow

Steps to build a compiler using SableCC3:

1. Create a SableCC grammar containing the lexical definitions and the productions of the language to be compiled. The productions will define the concrete syntax tree and it is possible to modify this tree in an abstract syntax tree via rewrite rules, see section 2.2;
2. Launch SableCC on the grammar file to generate a framework. Use SableCC.altgen[3] (not the normal SableCC version) to generate Python code: eg. `java -jar lib/sablecc.jar -t python grammar.g`. This command will generate the parser, lexer and walkers corresponding to the language defined in the grammar;
3. Create one or more working classes, possibly inheriting from classes generated by SableCC. Write the action code here;
4. Create a main compiler class that activates lexer, parser and working classes.

2.2. Tree construction

This section will handle the construction of a concrete syntax tree and subsequently an abstract syntax tree in SableCC3.

2.2.1. Concrete Syntax Tree

The concrete syntax tree is defined in an EBNF-like syntax in the productions section of a grammar. The CST is explained by means of an example for simple arithmetic expressions such as (1 + 2). This example is adapted from an article written by Nat Pryce[4].

Listing 7: Example Simple Expressions in SableCC3

```
Tokens
  add = '+';
  sub = '-';
  mul = '*';
  div = '/';
  left_paren = '(';
  right_paren = ')';
  number = ['0' .. '9']+;
  whitespace = (' ')*;


Ignored Tokens
  whitespace;


Productions
  expr = {add} [left]:expr add [right]:factor
       | {sub} [left]:expr sub [right]:factor
       | {factor} factor;

  factor= {mul} [left]:factor mul [right]:value
   | {div} [left]:factor div [right]:value
   | {value} value;

  value = {number} number
 | {paren} left_paren expr right_paren;
```

- In this example `expr` is a production that has 3 alternatives;

- An alternative has zero or more elements and each element is either a production name or a token name;
- An alternative is named by placing the name between curly brackets at the beginning of the alternative. If there is no name for an alternative, then SableCC will give it the name of the production prefixed by an 'A'. So when there is more than one alternative, it is obliged to give each alternative a name. Otherwise SableCC will generate two classes with the same name and you will get an error.
  Example: production `factor` has three alternatives named `mul`, `div`, `value`;
- The name of each element is defined inside square brackets followed by a colon and the element itself. The naming of elements is required when there is more than one element of the same type.

When a grammar is finished, it is time for SableCC3 to generate the framework. The generated parser class will automatically build a typed CST, while parsing the input. The nodes in the tree are represented by the `Node` class. Each production is represented by an abstract production class (that inherits from the Node class) and each alternative of a production is represented by a alternative class (that inherits from the production class). [3]

These alternative and production classes have names, SableCC naming rules are as follows:

- The classname of a production:
  Form: `P<production-name-capitalized>`
  Example: `PExpr`
- The classname of an alternative:
  Form: `A<alternative-name-capitalized><production-name-capitalized>`
  Example: `AAddExpr`
  `(alternative:  expr = {add} [left]:expr add [right]:factor )`

The CST for the expression grammar defined above will look like Figure 5 on page A - 11. The CST is quite large for such a simple expression, so this is not efficient for our walker. So the next section will point out the transformation from a CST to an AST.

2.2.2. Abstract Syntax Tree

In SableCC3 it is possible to rewrite the CST (the production rules define the CST) into an abstract syntax tree (AST) using rewrite rules. Rewrite rules are added to the productions to define how the CST is translated into the AST. The AST itself is defined in the Abstract Syntax Tree section. Below is an adjusted example using an AST[4]:

Listing 8: Example Simple Expressions in SableCC3 using an AST

```
1  Tokens
2    add = '+';
3    sub = '-';
4    mul = '*';
5    div = '/';
6    left_paren = '(';
7    right_paren = ')';
8    number = ['0' .. '9']+;
9    whitespace = (' ')*;
```

[3] If python output is generated, then there is no abstract production class. Instead, each alternative inherits directly from the Node class.

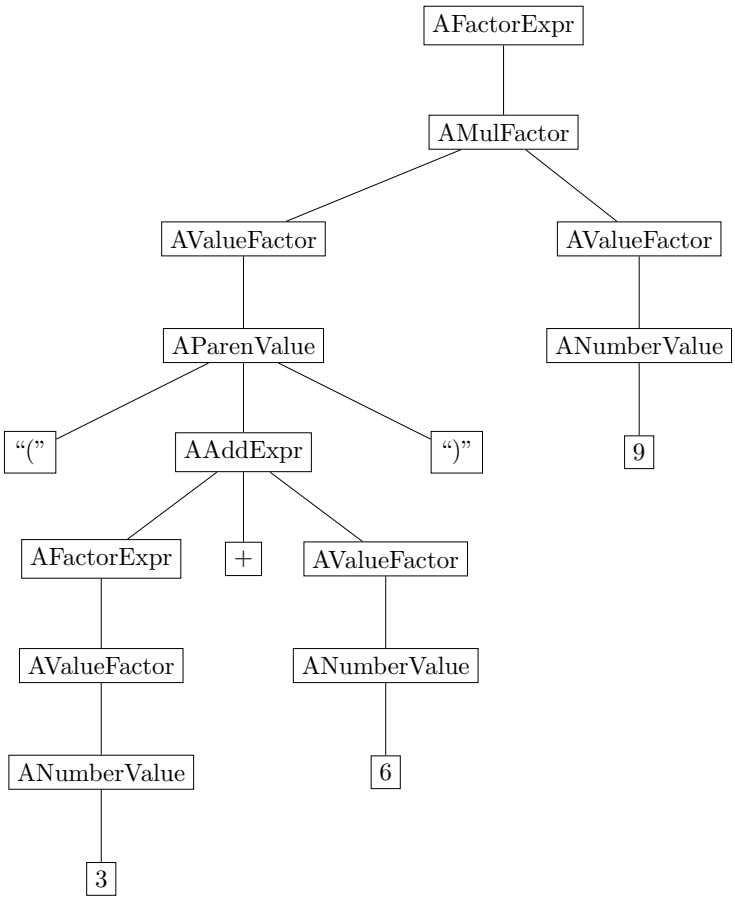

Figure 5: Concrete Syntax Tree for expression: (3+6)*9

```
10
11
12  Ignored Tokens
13    whitespace;
14
15
16  Productions
17    expr {-> expr} =
18        {add} [l]:expr add [r]:factor  {-> New expr.add(l.expr, r.expr)}
19      | {sub} [l]:expr sub [r]:factor  {-> New expr.sub(l.expr, r.expr)}
20      | {factor} factor                {-> factor.expr};
21
22    factor {-> expr} =
23        {mul} [l]:factor mul [r]:value {-> New expr.mul(l.expr, r.expr)}
24      | {div} [l]:factor div [r]:value {-> New expr.div(l.expr, r.expr)}
25      | {value} value                  {-> value.expr};
26
27    value {-> expr} =
28        {number} number                        {-> New expr.number(number)}
29      | {parens} left_paren expr right_paren   {-> expr.expr};
30
31
32    Abstract Syntax Tree
33
34    expr = {add} [left]:expr [right]:expr
35        | {sub} [left]:expr [right]:expr
36        | {mul} [left]:expr [right]:expr
37        | {div} [left]:expr [right]:expr
38        | {number} number;
```

First of all, each production is translated into an abstract `expr` production. This abstract production is defined below in the Abstract Syntax Tree section.
`eg.  factor {-> expr} = [..]`

Note: the concrete and abstract syntax reside in separate namespaces, i.e. the `expr` in curly brackets written in the productions section points to the `expr` production defined in the Abstract Syntax Tree and not to `expr` in the Productions section.

Secondly, the alternatives of a production (defined in the Productions section of a grammar) need to match the alternatives of the abstract `expr` production (defined in the Abstract Syntax Tree section). These rewrite rules are specified between curly brackets at the end of an alternative.

```
1  [..]
2  factor {-> expr} =
3      {mul} [l]:factor mul [r]:value {-> New expr.mul(l.expr, r.expr)}
4  [..]
```

The `New` keyword defines a new node, in the example above an `AMulExpr` node is created and the arguments are the children of this node. In other words, `expr.mul` matches the `mul` alternative in the `expr` production defined in the Abstract Syntax Tree. The arguments `l.expr` and `r.expr` identify the transformed nodes in the abstract syntax tree, more generally `<name of child>.<transformed type>`. It is also possible to ignore an alternative in the Productions section i.e., this alternative will not appear in the AST. To ignore an alternative, his name is simply enclosed in parentheses instead of brackets. In the following example the `number` alternative is ignored, so the AST will not contain any numbers.

Listing 9: Number alternative ignored

```
1  expr =
```

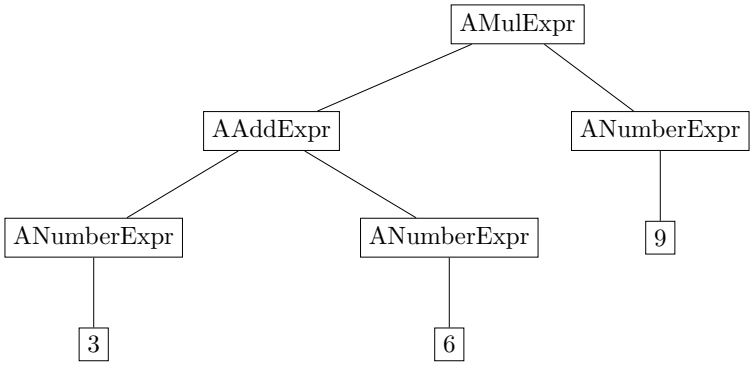

Figure 6: Abstract Syntax Tree for expression: (3+6)*9

```
2        [..]
3        | (number) number;
```

The AST for the same expression we used in Figure 5 can be found in Figure 6 .

2.3. Visitor Pattern

SableCC uses an extended visitor design pattern, that is described in the the thesis "SableCC, An Object-Oriented Compiler Framework" by Etienne Gagnon [5]. It is called "extended", because it is more generic. The visitors are tree-walkers. By default, SableCC provides two tree-walker classes. One that visits the nodes in a normal depth-first traversal (i.e. DepthFirstAdapter). The second class visits the AST nodes in the reverse depth-first traversal (i.e. ReversedDepthFirstAdapter). These walkers have methods that are called just before and after visiting a node while walking in the AST. These methods are named inXxx and outXxx respectively, where Xxx are the types of grammar alternatives (i.e. the types of nodes in the AST). It is easy to extend the class and override this methods. In this way it is possible to execute action code before and after the nodeis visited. This approach isolates action code and tree walking code in their own separate classes. An example of extending the DepthFirstAdapter class can be found in section 2.8.

2.4. Elegance

In this section the elegance and special features in SableCC are discussed.

2.4.1. Walkers

By default, SableCC provides two tree-walker classes. See previous section 2.3 Visitor Pattern.

2.4.2. Separation action versus grammar

In SableCC it is not possible to write action code in the grammar. Action code is written by extending the generated SableCC classes. The advantage of this approach is that it is easier to debug, because the debugging cycle is smaller due to the separation of grammar and action code, see Figure 7. This enables interactive debugging in an IDE. Separating the code from the grammar is cleaner than writing code in the grammar.

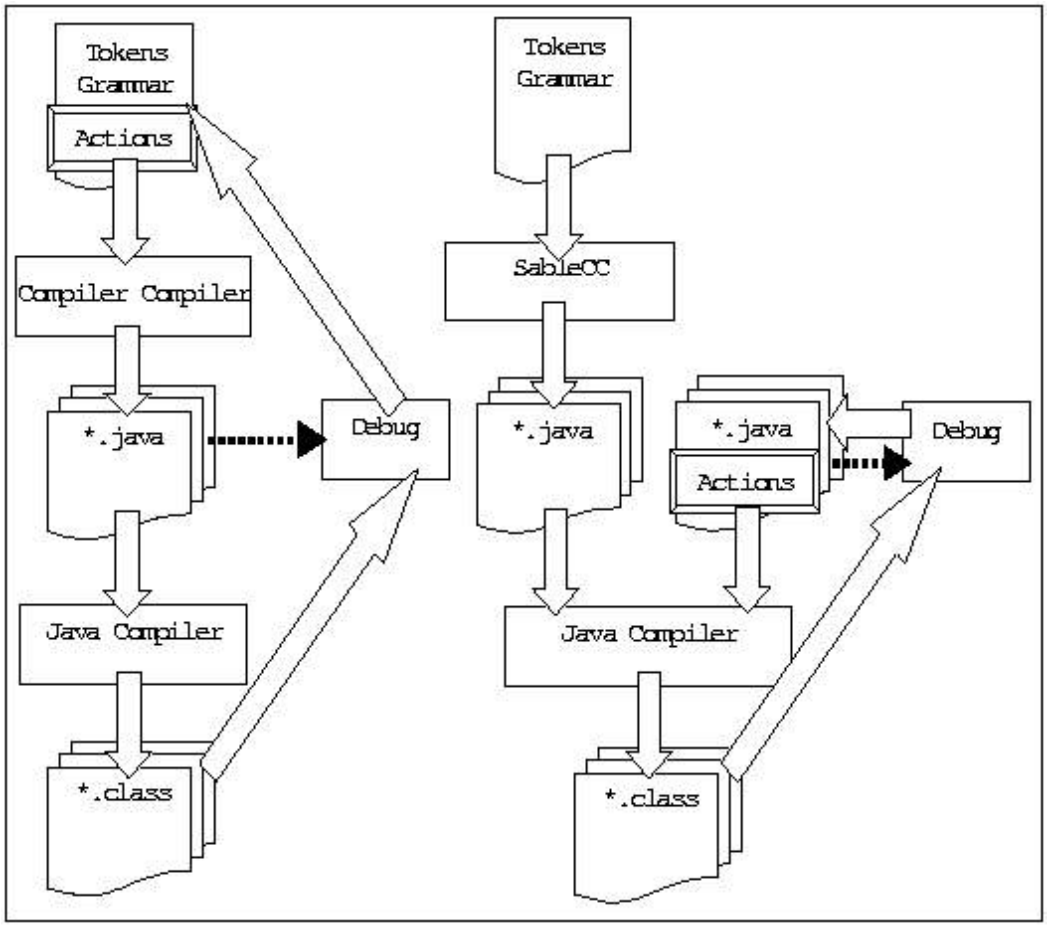

Figure 7: Traditional versus SableCC actions debugging cycle[5]

2.4.3. Lexer States

Each state is associated with a set of tokens. When the lexer is in a state, only the tokens associated with this state are recognized. States can be used for many purposes. For example, they can help detecting a "beginning of line"-state, and recognize some tokens only if they appear at the beginning of a line. Lexer States are also very useful for making island grammars (see 2.8). Of course, if there are states, there must be a mechanism to have transitions from one state to another state. In SableCC, state transitions are triggered by token recognition. Every time a token is recognizing, the lexer applies the transition specified for the current state and the recognized token. States and state transitions are defined in curly brackets before the token.

The purpose of the following example is to recognize when the parser is at the beginning of a line. There are two states: `bol` (beginning of line) and `inline`. The `char` token defines all characters except the line feed (10) and carriage return (13) character, because those two characters start a new line. If the `char` token is recognized and the current state is:

1. `bol`, then state `inline` should be selected, because there is no new line. This is defined by `bol->inline` on line 4.
2. `inline`, then we stay in this state, because there is no new line. This is defined by `inline` on line 4.

The `eol` token defines the characters that start a new line.

Listing 10: Example Beginning of Line state

```
1 States
2   bol, inline;
3 Tokens
4   {bol->inline, inline} char = [[0 .. 0xffff ] - [10 + 13]];
5   {bol, inline->bol} eol = 10 | 13 | 10 13;
```

2.5. Python support

There is an external module that supports alternative output, developed by Indrek Mandre [3]. The python backend is developed by Fidel Viegas. The problem is that it is built on SableCC3-beta3 and the last update was on 14-11-2004. In 2005, SableCC3.0 was released and meanwhile there is SableCC3.2. So there is no Python support for this version.

Another problem is that the generated parser contains a bug, ie. the read method in PushbackReader is broken. You can fix this by replacing the generated read method by the method provided in Listing 11.

Listing 11: Fixed PushbackReader read method

```
1 class PushbackReader(object):
2   [..]
3   def read (self,l=1): #fixed
4     if ( len(self.__stack) > 0 ):
5       return self.__stack.pop()
6     return self.__reader.read(l) #fixed
```

To conclude, there is Python third-party support for SableCC but it's unreliable (bugs, not maintained,..).

2.6. Tool Support

There is no specific tool support for SableCC3, but once the code is generated you can make use of the tools of the appropriate programming language (debugging, syntax highlighting,..).

2.7. Documentation

Documentation is scarce. There is a thesis [5] and there are some links to tutorials on the official site [6]. For Python generation there is no documentation and there are no examples.

2.8. Island Grammars

Using an example, this section will explain that it is possible to create island grammars in SableCC3. This example will extend the existing SableCC 3 grammar for simple arithmetic expressions by allowing PHP[7] expressions. The evaluation of the PHP expressions is done by another existing PHP4 grammar[8].

In the original simple arithmetic grammar it is possible to write the following expressions:

```
1 + (2 * 3);
55 / 1;
5
```

The goal is to adjust this grammar so that it is also possible to write PHP expressions:

```
1 + (2 * 3);
55 / 1;
<?php echo "bla"; ?>;
5
```

PHP expressions are written within the `<?php .. ?>` tag. The key to island grammars in SableCC are lexer states. In this example there are two lexer states: php and normal. The php state is used to recognize the `<?php .. ?>` tag and the normal state is used for expressions (i.e. the normal behavior of the grammar). The following listing contains the original token list of the simple expressions grammar:

```
1  Tokens
2    l_par = '(';
3    r_par = ')';
4    plus = '+';
5    minus = '-';
6    mult = '*';
7    div = '/';
8    semi = ';';
9
10   blank = blank;
11   number = digit+;
```

First of all there should be two states: php and normal. Secondly three tokens need to be added for recognition of the PHP expression:

1. `php_start`: this is the start tag `<?php` and handles the transition to the php state.
2. `php_body`: this contains the php code.
3. `php_end`: this is the end tag `?>` and handles the transition to the normal state.

Now the modified token list of our grammar looks like this:

```
1  States
2    normal, php;
3
4  Tokens
5    {normal->php, php} php_start = '<?php ';
6    {php} php_body =[all - ['?' + '>']]*;
7    {php->normal, normal} php_end = '?>';
8
9    {normal} l_par = '(';
10   {normal} r_par = ')';
```

```
11    {normal} plus = '+';
12    {normal} minus = '-';
13    {normal} mult = '*';
14    {normal} div = '/';
15    {normal} semi = ';';
16
17    {normal} blank = blank;
18    {normal} number = digit+;
```

Our modified grammar is finished and it is time for SableCC to generate the Python classes (parser, tree, node, walkers). Once this is done, it is important to change the "read" bug in the PushbackReader class as mentioned in Listing 11 on page A - 15. The last step is the creation of the compiler class in Python using the generated classes (Listing 12).

- On line 3: import the parser.py file, this file contains all generated classes.
- On line 5: define that the file to compile is given as an argument, so the compiler script is used in this way:
  `$ python compiler.py test.php`
- On line 9: define the walker class, this class overrides the outAPhpExp method. These methods are explained in section 2.3. In this case it means, that outAPhpExp is called right after the node APhpExp is visited. In this way it is possible to get the body of the PHP expression (`node.getPhpBody()` on line 11) and compile it using a PHP compiler. Because the PHP4 SableCC parser[8] is written in Java and not in Python, the .jar file is executed via commands[9]. In Python it would be easier, because we just had to call a python compiler class.

Listing 12: compiler.py

```
1  import sys
2  import commands
3  from parser import *
4
5  #filename as argument
6  f = sys.argv[1]
7  t = open(f,"r")
8
9  class MyPhpWalker(DepthFirstAdapter):
10   def outAPhpExp(self, node):
11     phpcode = node.getPhpBody().getText().replace('"', '\\"')
12     cmd = 'echo "<?php '+phpcode+'?>" |   ';
13     cmd += 'java -cp php4/lib/php4sablecc.jar::php4/classes/ TestParser'
14     d = commands.getstatusoutput(cmd)
15     pass
16
17 #create parser
18 p = Parser(Lexer(PushbackReader(t)))
19
20 #parse!
21 tree = p.parse()
22
23 #apply the walker.
24 tree.apply(MyPhpWalker())
```

The original simple arithmetic expressions grammar can be found in Appendix A on page A - 37, the modified grammar in Appendix B on page A - 41

2.9. Scalability

SableCC is a LALR(1) parser (bottom-up). LR parsing can handle a larger range of languages than LL parsing.

2.10. SableCC4

Currently Etienne M. Gagnon is working on a complete rewrite of SableCC (i.e. SableCC4) that has many new features:

- Improved lexer engine (additional operators, lookahead, and more).
- Improved parser engine (linear approximate LR(K) parsing, semantic selectors, and more).
- Improved conflict reporting (enabling LR grammar debugging).
- Flexible code generation (to enable back-ends for various languages).
- Improved syntax for CST-AST transformations.

But the date of the release of SableCC4 is not known.

3. ANTLRv3

3.1. Workflow

Steps to build a compiler using ANTLRv3:

1. Open the ANTLRWorks IDE and create a grammar which contains the tokens, parser and lexer rules. Make sure the target language is set to Python. This is discussed in more detail in section 3.2;
2. Generate the parser and lexer code using shortcut CTRL+Shift+G or by clicking in the menu bar. It is also possible to generate your code via command-line: `java -jar antlr-3.1.3.jar grammar.g`;
3. Create a Python file and import the lexer and parser. An example is given in Listing 13;
4. Start building a compiler.

Listing 13: ANTLR example that prints the syntax tree

```
1  import sys
2  import antlr3
3  from testLexer import testLexer
4  from testParser import testParser
5  f = sys.argv[1]
6  t = open(f,"r")
7  char_stream = antlr3.ANTLRInputStream(t)
8  lexer = testLexer(char_stream)
9  tokens = antlr3.CommonTokenStream(lexer)
10 parser = testParser(tokens)
11 try:
12   r = parser.expr()
13   print r.tree.toStringTree()
14 except antlr3.RecognitionException:
15   traceback.print_stack()
```

3.2. Tree construction

3.2.1. Grammar Basics

As in SableCC, this section will also consider a simple grammar that accepts simple arithmetic expressions. Grammars in ANTLR have four important sections:

1. Options, here the target language (Python) of the compiler compiler, the output of the parser (AST) and eventually other options are defined;
2. Tokens contains the list of tokens;
3. Parser rules define the matching rules for the parser (these are like the productions in SableCC) and they start with a lowercase letter. Parser rules can have multiple alternatives and ANTLR takes care of the names. Parser rules may reference literals, parser and lexer rules, but never only literals;
4. Lexer rules define the matching rules for the lexer and they start with an uppercase letter. Lexer rules contain only either literals or references to other lexer rules.

Listing 14: Example simple expressions in ANTLRv3

```
1  grammar SimpleCalc;
2
3  options {
4    language = Python;
5    output=AST;
6  }
```

```
7
8  tokens {
9   PLUS  = '+' ;
10  MINUS = '-' ;
11  MULT = '*' ;
12  DIV = '/' ;
13 }
14
15 /*-PARSER RULES-----------------------------------------------*/
16 expr : term ( ( PLUS | MINUS )  term )* ;
17 term : factor ( ( MULT | DIV ) factor )* ;
18 factor : NUMBER ;
19
20
21 /*-LEXER RULES-----------------------------------------------*/
22 NUMBER : (DIGIT)+ ;
23 fragment DIGIT : '0'..'9' ;
24 WS      : (' '|'\t'|'\n')+;
```

The default behavior of ANTLR is that the lexer rule names are automatically elevated into token status. The only difference between the lexer rules and the token section is that the token section is prioritized over the lexer rule. For instance, suppose there is token defined that only matches the keyword "`import`". Then it would also match the more general lexer rule "`('a'..'z')+`". But due to the prioritization the `import` token will be used and not the lexer rule.

In Listing 14, `DIGIT` is prefixed by the keyword `fragment`. This means that the lexer rule is not elevated into token status. In other words, it cannot be used by a parser rule and is used to improve the readability of the grammar.

3.2.2. Abstract Syntax Tree

ANTLR provides two mechanisms for the creation of Abstract Syntax Trees: via operators or via rewrite rules. There are two kinds of operators:

- `!` excludes a node or subtree;
- `^` makes the node root of subtree.

In the following listing, PLUS or MINUS is the root in each `expr` (parser rule) and DIV or MULT is the root in each `term` (parser rule).

```
1 /*-PARSER RULES-----------------------------------------------*/
2 expr : term ( ( PLUS | MINUS )^  term )* ;
3 term : factor ( ( MULT | DIV )^ factor )* ;
4 factor : NUMBER ;
```

Figure 8 on page A - 21 shows the AST for expression `3*2+5`.

The rewrite rules are something similar like the SableCC rewrite rules. The rewrite syntax is more powerful than the operators, because now it is possible to reorder the elements in a rule. Suppose that variables are also supported in our grammar and tokens NEWLINE and TYPE are added. In this case rewrite rules are used:

```
1  stat : expr NEWLINE          -> expr
2  | TYPE ID '=' expr NEWLINE -> ^('=' ID TYPE expr)
```

This is impossible when using operators, because it is impossible to rearrange the order of elements in a rule.

3.3. Visitor Pattern

It is possible to apply the visitor pattern, but you have to write a tree walker all by yourself[10]. This is not user-friendly because the parser is rather complicated.

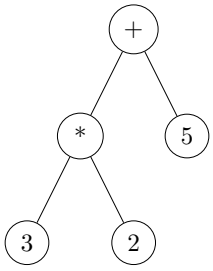

Figure 8: Abstract Syntax Tree for expression: 3*2+5

3.4. Elegance

In this section the elegance and special features in ANTLR are discussed.

3.4.1. Actions

It is possible to add action code in your grammar written in the target language[4]. In our case the target language is Python. These actions are executed when the parsers encounters it. Actions are written between curly brackets. The following example just prints out the number.

```
1 /*-PARSER RULES----------------------------------------------*/
2 expr : term ( ( PLUS | MINUS )  term )* ;
3 term : factor ( ( MULT | DIV ) factor )* ;
4 factor : NUMBER {print $NUMBER.getText()};
```

3.4.2. Syntactic Predicates

A syntactic predicate specifies the validity of applying a rule. Suppose we would write our `expr` rule of Listing 14 in this way:

```
1 expr :  term (PLUS term)*
2         |  term (MINUS term)*;
```

ANTLR would give an error stating that the following alternatives can never be matched. Because ANTLR is a LL parser and alternatives cannot have common prefixes, see "factoring common prefixes" on page A - 7. In this case both alternatives have a common prefix (i.e. `term`). To get rid of this conflict, syntactic predicate are used. ANTLR evaluates the predicate and if the predicate matches his token stream, the alternative of the predicate will be selected. A syntactic predicate is of the form: `(PREDICATE) => RULE`

Listing 15: Modified `expr` rule using a syntactic predicate

```
1 expr : (term PLUS term)=> term (PLUS term)*
2         |  term (MINUS term)*;
```

3.4.3. Semantic Predicates

A semantic predicate is a way to enforce extra (semantic) rules upon grammar actions using plain code.

There are 3 types of semantic predicates (note that RULE is a placeholder for a parse rule):

[4]The target language of ANTLR is defined in the options section of your grammar.

1. **Validating semantic predicates** :
   `RULE {boolean-expression}?`
   If the boolean expression returns false, then a FailedPredicateException is thrown and parsing fails.
2. **Gated semantic predicates** :
   `{boolean-expression}?=> RULE`
   If the boolean expression returns false, then a syntax error is produced and parsing fails. The difference with a validating semantic predicate is that the boolean expression is before the rule (instead of after the rule) and a syntax error is produced (instead of an exception).
3. **Disambiguating semantic predicates** :
   `{boolean-expression}?  RULE`
   If the boolean expression returns false, the rule is ignored.

An example of a disambiguating semantic predicate (line 13) is given below. In this example the parser wants a distinction between low and high numbers.

Listing 16: An example of a disambiguating semantic predicate

```
1  grammar Numbers;
2
3  parse
4     :  atom (',' atom)* EOF
5     ;
6
7  atom
8     :  low  {System.out.println("low  = " + $low.text);}
9     |  high {System.out.println("high = " + $high.text);}
10    ;
11
12 low
13    :  {Integer.valueOf(input.LT(1).getText()) <= 500}? Number
14    ;
15
16 high
17    :  Number
18    ;
19
20 Number
21    :  Digit Digit Digit
22    |  Digit Digit
23    |  Digit
24    ;
25
26 fragment Digit
27    :  '0'..'9'
28    ;
29
30 WhiteSpace
31    :  (' ' | '\t' | '\r' | '\n') {skip();}
32    ;
```

3.5. Python support

Python is a standard target language in ANTLR maintained by Benjamin Niemann[11]. Right now the Python target is only functional up to v3.1.3, while the latest version of ANTLR is v3.4.

3.6. Tool Support

ANTLRWorks[12] provides syntax highlighting and completion on grammar elements. It allows the programmer to enter test fragments to parse. However there is no highlighting/completion for the code inside actions.

The ANTLRWorks debugger for Python is still in early development[11]. Tree parsers are currently not supported, but token parsers should work (with and without AST generation). There is also an Eclipse plugin for ANTLR (ANTLR IDE[13]) that supports some simple syntax highlighting for Python.

3.7. Documentation

The amount of documentation is huge. There is a large website[5], containing lots of information. Most of the documentation is for the target language Java, but that is not a problem because the construction of (tree) grammars in ANTLR is important and that is for each target language (almost) the same[6]. Moreover, there is enough documentation for target language Python:

- Antlr3PythonTarget:
  http://www.antlr.org/wiki/display/ANTLR3/Antlr3PythonTarget
- Python ANTLR Run-Time documentation:
  http://www.antlr.org/wiki/display/ANTLR3/Python+runtime
- Python ANTLR API documentation:
  http://www.antlr.org/api/Python/index.html

Sometimes the example and instructions are outdated. There is also a book[14] about ANTLR.

3.8. Island Grammars

Island grammars are possible in ANTLR. This section will explain it using an example inspired from an existing example[7]. In our existing SimpleCalc grammar, defined in Listing 14, it is not possible to use PHP expressions. A SimplePHP grammar will be embedded into the existing grammar to solve this problem. There is no PHP grammar available at this moment and writing a new PHP grammar in PLY is outside the scope of this report. Thus SimplePHP is just a simple grammar that can parse the `echo` expression. Below is an example of an input file that conforms to the new SimpleCalc grammar, followed by the modified SimpleCalc grammar.

Listing 17: Input file

```
1 1+1
2 <?php echo "test"; ?>
3 1*5
```

Listing 18: Modified SimpleCalc grammar

```
1 grammar SimpleCalc;
2
3 options {
4  language = Python;
```

[5] http://www.antlr.org/wiki/display/ANTLR3/ANTLR+3+Wiki+Home

[6] The names can sometimes be different, eg. org.antlr.runtime.CharStream.EOF in Java vs. antlr3.EOF in Python

[7] Python Island Grammar in http://www.antlr.org/download/examples-v3.tar.gz

```
 output=AST;
}

tokens {
 PLUS  = '+' ;
 MINUS = '-' ;
 MULT = '*' ;
 DIV = '/' ;
}Z

/*-PARSER RULES-----------------------------------------------*/
expr : term ( ( PLUS | MINUS )  term )* ;
term : factor ( ( MULT | DIV ) factor )* ;
factor : NUMBER ;


/*-LEXER RULES-----------------------------------------------*/
NUMBER : (DIGIT)+ ;
fragment DIGIT : '0'..'9' ;
WS      : (' '|'\t'|'\n')+;

PHP : '<?php'
          {
print "enter PHP"
import SimplePHPLexer
import SimplePHPParser
j = SimplePHPLexer.SimplePHPLexer(self.input)
tokens = CommonTokenStream(j)
tokens.discardTokenType(SimplePHPLexer.WS)
p = SimplePHPParser.SimplePHPParser(tokens)
p.start()

$channel = PHP_CHANNEL
          }
         ;
```

The only thing that has changed is that there is a new lexer rule `PHP` at the bottom of the grammar. This new rule contains an action to call the lexer and parser of the second grammar, ie. SimplePHP grammar. If the SimplePHP parser recognizes the PHP endtag `?>`, the grammar will change its token via an action in the EOF_TOKEN. This will stop the SimplePHP parser and will return to our SimpleCalc grammar parser.

Listing 19: Javadoc Grammar

```
grammar SimplePHP;

options {
    language=Python;
}


start  :  echo;

echo   : 'echo "' ID {print $ID.text} '";';

ID      : ('a'..'z'|'A'..'Z')+;

END     : '?>' {self._state.token = EOF_TOKEN}
          {print "exit PHP"}
         ;

WS      : (' '|'\t'|'\n')+
```

```
19          ;
```

Our mini-compiler only uses the SimpleCalc lexer and parser, because the SimplePHP parser is called inside the SimpleCalc grammar. So the python file will look like this:

Listing 20: compiler.py

```
1  import sys
2  import antlr3
3  from SimpleCalcLexer import SimpleCalcLexer
4  from SimpleCalcParser import SimpleCalcParser
5
6  cStream = antlr3.StringStream(open(sys.argv[1]).read())
7  lexer = SimpleCalcLexer(cStream)
8  tStream = antlr3.CommonTokenStream(lexer)
9  parser = SimpleCalcParser(tStream)
10 parser.expr()
```

3.9. Scalability

ANTLR is a LL(*) parser (top-down). The asterisk defines that it the parser is not restricted to look only k tokens ahead to make parse decisions. As we have seen in the introduction, LL parsing can handle less languages than LL parsers, but the generated code of the parser is easier to understand.

3.10. ANTLRv4

Currently Terence Parr is working on a complete rebuild of ANTLRv3[15].

4. PLY

4.1. Workflow

PLY (Python-Lexx-Yacc) consists of two separate modules, lex.py and yacc.py, both of which are found in a Python package called ply. The lex.py (lexer) module is used to break input text into a collection of tokens specified by a collection of regular expression rules. The yacc.py (parser) module is used to recognize language syntax that has been specified in the form of a context free grammar. The two tools are meant to work together. Specifically, lex.py provides an external interface in the form of a token() function that returns the next valid token on the input stream. yacc.py calls this repeatedly to retrieve tokens and invoke grammar rules. The output of yacc.py is often an Abstract Syntax Tree (AST). However, this is entirely up to the user. If desired, yacc.py can also be used to implement simple one-pass compilers.[8]

4.2. Tree construction

As mentioned in the previous section: first step is defining a lexer using the lex.py module and the second step is to define the parser that will output the AST. In this section lexer, parser and tree construction will be explained by means of an adjusted example from the official documentation[16]. This example will compile simple arithmetic expressions.

4.2.1. Lexer

Listing 21 defines our lexer for the simple expressions language.

Listing 21: Simple Arithmetic Expressions Lexer: exprlex.py

```
1  import ply.lex as lex
2
3  # List of token names.   This is always required
4  tokens = (
5      'NUMBER',
6      'PLUS',
7      'MINUS',
8      'MULT',
9      'DIV',
10 )
11
12 # Regular expression rules for simple tokens
13 t_PLUS    = r'\+'
14 t_MINUS   = r'-'
15 t_MULT   = r'\*'
16 t_DIV  = r'/'
17
18 # A regular expression rule with some action code
19 def t_NUMBER(t):
20     r'\d+'
21     t.value = int(t.value)
22     return t
23
24 # Define a rule so we can track line numbers
25 def t_newline(t):
26     r'\n+'
27     t.lexer.lineno += len(t.value)
28
29 # A string containing ignored characters (spaces and tabs)
30 t_ignore  = ' \t'
```

[8]This introduction is adapted from the official site: http://www.dabeaz.com/ply/ply.html

```
31
32 # Error handling rule
33 def t_error(t):
34     print "line",t.lexer.lineno,": illegal character",t.value[0]
35     t.lexer.skip(1)
36
37 # Build the lexer
38 lexer = lex.lex()
```

The `tokens` variable is just a list of our token names, but it does not define them. Each token is specified by a regular expression rule. The name of the declaration that specifies a certain token must have a prefix `t_` followed by the token name. For simple tokens we could write:

```
1  t_DIV  = r'/'
```

Alternatively, if some kind of action code needs to be executed, then a token rule could be specified as a function:

```
1  def t_NUMBER(t):
2      r'\d+'
3      t.value = int(t.value)
4      return t
```

This token rule matches every number that has one or more digits. Its value is converted to an int. There is also an important difference between the use of functions or strings, because functions are matched in order of specification, whereas strings are added after the functions and are sorted by decreasing regular expression length (longer expressions are added first).

Next there are three special functions defined in our lexer:

1. `t_newline(t)` specifies how the lexer should count line numbers;
2. `t_ignore` is a string that contains ignored characters. Alternatively, it is also possible to ignore a certain token by prefixing the token with `t_ignore_` instead of `t_` or a token rule that does not return any value is also ignored;
3. `t_error(t)` specifies how the lexer should handle lexer errors. In the example above the character is skipped and the lexer continues.

There are no reserved words in our lexer, but suppose there were reserved words like IF, THEN and ELSE. Then there should be a single rule to match an identifier and the type of the token should be modified if it is a reserver word. In the Listing below, function `t_ID(t)` checks for reserved words. If it is a reserved word, it changes the token type to the type of the reserved word. Otherwise it will keep its type `ID`.

Listing 22: Reserved words

```
1   reserved = {
2     'if' : 'IF',
3     'then' : 'THEN',
4     'else' : 'ELSE',
5     'while' : 'WHILE',
6   }
7
8   tokens = ['MINUS','PLUS',...,'ID'] + list(reserved.values())
9
10  def t_ID(t):
11      r'[a-zA-Z_][a-zA-Z_0-9]*'
12      t.type = reserved.get(t.value,'ID')    # Check for reserved words
13      return t
```

This approach reduces the number of regular expression rules.

4.2.2. Parser

Listing 23 defines our parser for the simple expressions language.

Listing 23: Simple Arithmetic Expressions Parser: expr.py

```
import ply.yacc as yacc

# Get the token map from the lexer.  This is required.
from exprlex import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term MULT factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIV factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor(p):
    'factor : NUMBER'
    p[0] = p[1]

# Error rule for syntax errors
def p_error(p):
    print "Syntax error in input!"

# Build the parser
parser = yacc.yacc()

while True:
   try:
       s = raw_input('calc > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
   print result
```

This python program will ask for an expression and calculate this expression, this is done in the while-loop. Each grammar rule is defined as a function and prefixed with `p_`. The first function is the start symbol. In each function the docstring contains the CFG specification of the rule. The statements after the string are the actions of that rule. The argument `p` contains the values of each symbol in the rule:

```
def p_expression_plus(p):
    'expression : expression PLUS term'
```

```
    #    ^               ^          ^     ^
    #  p[0]            p[1]       p[2] p[3]

    p[0] = p[1] + p[3]
```

In function `p_expression_plus(p)` our action code is adding our "expression" (`p[1]` to the "term" (`p[3]`). Because a plus expression is similar to a minus, mult and div expression, this could be combined:

```
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
       term       : term MULT factor
                  | term DIV factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

4.2.3. Abstract Syntax Tree

PLY provides no special functions for generating an AST. The programmer should construct it. A very simple way to construct a tree is to propagate a tuple in each grammar rule:

```
def p_expression_plus(p):
      'expression : expression PLUS term'
      p[0] = ('+',p[1],p[3])

def p_expression_minus(p):
      'expression : expression MINUS term'
      p[0] = ('-',p[1],p[3])
...
```

4.3. Visitor Pattern

It is possible, but PLY does not generate a concrete or abstract syntax tree. So first of all you have to create a tree. Secondly, because PLY does not generate trees it certainly does not generate walkers or visitors. The programmer has to write everything on its own.

4.4. Elegance

In this section the elegance and special features in PLY are discussed.

4.4.1. Precedence Rules

The expression grammar in section 4.2.2 has been written to eliminate ambiguity (i.e. the parser does not have multiple options to parse a string), but PLY can handle ambiguity if there are proper precedence rules defined. This means, if the parser has multiple options, then precedence rules define the order of precedence for tokens.

Listing 24 shows our modified ambiguous expression specification. It is ambigous because the parser does not know how to handle for example `1+4*6`. Does it means `(1+4)*6` or `1+(4*6)`?

Listing 24: Ambiguous Expression Grammar

```
expression : expression PLUS expression
           | expression MINUS expression
           | expression MULT expression
           | expression DIV expression
           | NUMBER
```

In PLY, a precedence rule assigns a token to a certain precedence level and associativity. Associativity defines how the parser should handle multiple tokens sharing the same precedence. There are three types of associativity: `left`, `right` and `nonassoc`. Listing 25 defines the precedence rules in PLY. There are two rules and these rules are ordered from lowest to highest precedence (eg. `1+2*3` is parsed as `1+(2*3)`, because MULT has a higher precedence). Each rule has a left associativity (eg. `1+2-3+4` is parsed as `((1+2)-3)+4`. If the associativity of PLUS and MINUS was set to right, then the expression would be parsed as `1+(2-(3+4))`. If it was set to non-associative, then these tokens (PLUS, MINUS) could not be chained and a syntax error is raised for this expression).

Listing 25: Precedence Rules

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MULT', 'DIV'),
)
```

4.4.2. Parser Library and Encapsulation

PLY is a parser library and not a parser generator (like ANTLR and SableCC). It does not generate a specific parser using a grammar as input. It provides a Python parser and lexer class, which are extended by the programmer. Lexer rules are defined in the lexer class, parser rules in the parser class. This leads to an easy creation of parsers for several different grammars and to make instances of that parser at the same time. Because it is written by the programmer, it is easy to understand his own code and he could write the "optimal" parser for his grammar. This is not the case when using parser generators. Here the code for our parser is generated and it is harder to understand. It is even harder to change the generated parser to the "optimal" parser.

4.5. Python support

PLY is completely in Python and it is well maintained (last release 17/2/2011).

4.6. Tool Support

There are no tools, just two python modules (lex and yacc). The `lexx()` and `yacc()` commands have a debug mode that can be enabled using the `debug` flag (`lex.lex(debug=True)`).

4.7. Documentation

The official website[9] offers enough documentation and examples.

[9] http://www.dabeaz.com/ply/

4.8. Island Grammars

Island grammars are possible in PLY. Section 4.2 defines the expression grammar. A modification of this grammar that supports php expressions using another php parser is described in Listing 26 and Listing 27. The PHP token is added to the lexer and a php expression is written between `<?  ..  ?>` tags. In the parser one parse rule is added (`p_expression_php`) that will match the php expressions. The begin and end tag are deleted and the string is passed onto a PHP parser.

Listing 26: Modified Expression Lexer

```
tokens = [
  'NUMBER',
  'PLUS',
  'MINUS',
  'MULT',
  'DIV',
  'PHP',
]

def t_PHP(t):
  r'<\? .* \?>'
  return t

..
```

Listing 27: Modified Expression Parser

```
def p_expression_php(p):
  'expression : PHP'

  # strip '<?' and '?>' tags
  phpstring = str(p[1])
  phpstring = phpstring.replace("<?","")
  phpstring = phpstring.replace("?>","")

  #parse php
  parser = php_parser.PHPParser()
  result = parser.parse(phpstring)
  print result

..
```

4.9. Scalability

In PLY the programmer may choose between LALR(1) (default) or SLR parsing. Andrew Dalke stated in his article[17] that in a certain case PLY is 3.7 times faster than ANTLR. This was expected, because PLY is much lighter. It is actually a Python library and the speed of the parser is more the responsability of the programmer.

5. Other Tools

This section will briefly discuss some tools that did not satisfy the criteria, but are still worth to mention.

5.1. Yapps

Yapps[18] claims that it is designed to be used when regular expressions are not enough and other parser systems are too much. But this report aims to search for a descent (large) parser system. Yapps uses LL(1) parsing, which is less powerful than all the other parsing types discussed in the previous sections.

5.2. PyParsing

The parsing module of PyParsing[19] provides a library of classes that are used to construct the grammar directly in Python. Disadvantages of using PyParsing:

1. there is no difference between lexer and parser;
2. it is ideal for small and simple grammars;
3. indentation-style blocking is discouraged, PyParsing is not meant to parse indentation-style blocks.

5.3. Spoofax

Spoofax[20] is a very powerful language workbench for developing textual domain-specific languages in Eclipse. The programmer can edit and use his language in Eclipse. Spoofax has a lot of nice features like custom errors, syntax directed editor generation and deploying your own editor as an Eclipse plugin.

The only problem, in the scope of this report, is that the created language must be used within Eclipse. There is no generated parser written in Python. The compiler of your language is actually the Spoofax plugin within Eclipse. This report focusses on an editor-independent compiler.

6. Conclusion

	SableCC3	ANTLRv3	PLY
Python support	++ (unstable)	++++ (stable)	+++++ (python library)
Tree construction	+++ (rewrite rules)	+++++ (rewrite rules, operators)	+ (create own tree)
Visitor pattern	+++++ (extended visitor pattern)	++ (create own visitor)	+ (create own visitor)
Elegance	+++ (walkers, separation action vs. grammar)	+++ (grammar actions, tree grammars, predicates)	+++ (precedence rules, encapsulation, parser library)
Tools	+ (python tools)	+++ (ANTLRWorks)	++ (debugger, python tools)
Documentation	+ (no python documentation)	+++++ (internet,books)	+++ (internet)
Island Grammars	++++ (lexer states)	+++ (it is possible, but dirty)	+++++ (call other parser class)
Scalability	+++ LALR(1)	+++ LL(k)	++++ LALR(1)

The Python generated code in SableCC3 contains bugs and the alternative output package uses a beta version of SableCC. In ANTLRv3 the Python support is better: it uses ANTLRv3.1.3 (the latest version is 3.4) and is very stable. The python support for PLY is excellent, because it is just a Python library.

For tree construction ANTLRv3 does not only offer rewrite rules (like SableCC), but also operators. The visitor pattern is the flagship of SableCC and the amount of work to achieve this in ANTLRv3 is high. Trees in PLY are manually created therefore the visitor pattern in PLY is even harder than in ANTLR (because first a tree class have to be created, before implementing the visitor pattern).

SableCC is elegant by separating the grammar from the action code, ANTLRv3 is elegant by giving so much features in the ANTLRWorks IDE and it also supports predicates and actions inside a grammar. There are many tools available for ANTLRv3, unfortunately the tools for the Python target language is rather limited. SableCC does not provide any tools, but thanks to the separation of grammar and action code language-specific tools (like a debugger) could be used. This is also the case for PLY, additionally there is debug-mode option.

The documentation for SableCC is very scarce and the documentation for the Python target language does not exist. ANTLRv3 has a gold mine of documentation. PLY provides enough basic documentation.

Through lexer states it is possible to make island grammars in SableCC in a very clean way. In ANTLRv3 it is not so clean and action code is used to achieve this. In PLY it is very simple by calling just another parser class.

In general the python target for SableCC cannot be used, because of bugs. ANTLRv3 is the most powerful and easiest tool among these three tools (easy tree construction, mixed grammar containing actions and predicates). Using ANTLR the creation of the compiler is quickly

launched, but when the code is generated and there is a need for walkers visiting the AST, it is not so straightforward anymore. In the end PLY is faster and it can handle island grammars in a very clean way. In PLY all the code is just python code and could be easily debugged. There is more responsability for the programmer. He has to code more, but he also gets more freedom in his code. Another difference is that ANTLR needs a java program to generate the parser and PLY is actually just a python library. In my opinion PLY is the best tool for making a compiler in python.

7. Bibliography

References

[1] A. W. Appel, Modern compiler implementation in Java, 1997.

[2] C. N. Fischer, R. K. Cytron, J. Richard J. LeBlanc, Crafting a Compiler, 2010.

[3] Indrek Mandre, Indrek's SableCC page: Alternative output for SableCC 3 (2004).
URL http://www.mare.ee/indrek/sablecc/#altgen

[4] Nat Pryce, Concrete to Abstract Syntax Transformations with SableCC (2005).
URL http://nat.truemesh.com/archives/000531.html

[5] E. Gagnon, Sablecc, an object-oriented compiler framework (March).

[6] SableCC, SableCC Documentation Page.
URL http://sablecc.org/wiki/DocumentationPage

[7] Rasmus Lerdorf, PHP: Hypertext Preprocessor.
URL http://php.net/

[8] Indrek Mandre, Indrek's SableCC page: PHP 4 grammar for SableCC 3 complete with transformations (2003).
URL http://www.mare.ee/indrek/sablecc/#php4

[9] Python, 35.16. commands Utilities for running commands Python v2.7.2 documentation.
URL http://docs.python.org/library/commands.html

[10] A. Tripp, Manual Tree Walking Is Better Than Tree Grammars (2006).
URL http://www.antlr.org/article/1170602723163/treewalkers.html

[11] B. Niemann, Antlr3PythonTarget - ANTLR 3 - ANTLR Project.
URL http://www.antlr.org/wiki/display/ANTLR3/Antlr3PythonTarget

[12] Jean Bovet, ANTLRWorks: The ANTLR GUI Development Environment.
URL http://www.antlr.org/works/index.html

[13] E. Espina, ANTLR IDE. An eclipse plugin for ANTLR grammars.
URL http://antlrv3ide.sourceforge.net/

[14] T. Parr, The Definitive Antlr Reference: Building Domain-Specific Languages, Pragmatic Bookshelf; 1 edition, 2007.

[15] T. Parr, ANTLR v4 plans - Terence Parr - ANTLR Project.
URL http://www.antlr.org/wiki/display/ admin/ANTLR+v4+plans

[16] David M. Beazley, ply official documentation.
URL http://www.dabeaz.com/ply/ply.html

[17] Andrew Dalke, More ANTLR - Java, and comparisons to PLY and PyParsing (2007).
URL http://www.dalkescientific.com/writings/diary/archive/2007/11/03/antlr_java.html

[18] A. Patel, Parsing with Yapps.
URL http://theory.stanford.edu/ amitp/yapps/

[19] P. McGuire, PyParsing.
URL http://pyparsing.wikispaces.com/

[20] E. V. Lennart C. L. Kats, Spoofax.
URL http://spoofax.org

Appendix A. Original Simple Arithmetic Expressions Grammar

```
1  //
2  // This is a demonstration grammar file with transformation rules for the
3  // new SableCC3 parser generator (rel sablecc-3-beta.3.altgen.20040327)
4  //
5  // Points to remember:
6  //
7  //  * Why CST and AST? Due to limitations of parser technology the
8  //    human described grammar does not represent the "perfect" abstract
9  //    form of the language parsed. To get the AST transformations are done.
10 //    (AST - Abstract Syntax Tree; CST - Concrete Syntax Tree)
11 //
12 //  * The AST section must be complete, nothing from Productions is
13 //    automatically placed there.
14 //
15 //  * In curly braces are things related to AST, eg. we built the AST in
16 //    those things, so everything is built/transformed from the leaves to
17 //    root.
18 //
19 //  * Transformation is divided into two parts - product transformation
20 //    declaration that declares what the alternatives should be
21 //    transformed to and alternatives transformations definitions that
22 //    define how the transform is actually done.
23 //
24 //  * A production can be transformed to multiple elements as seen in the
25 //    'random_x2' rule.
26 //
27 //  * Productions with same structure as in the AST can be directly
28 //    transformed to AST - see the 'textual' rule. What really happens is
29 //    that for productions and alternatives without transform specification
30 //    default transformation rules are generated. This also means that you
31 //    could omit the transform declaration at exp rule.
32 //
33 //  * Lists it seems are represented with brackets '[ elem1, elem2, .. ]' and
34 //    not parenthesis as described in the doc. If element is also a list it
35 //    is automatically expanded and used. Empty list is [].
36 //
37 //  * The output we get from the parser is as described in the AST. We only
38 //    have to work with that. The productions section is no longer used.
39 //
40 //  * When you just want to get rid of a production declare and define
41 //    it and its alternatives as {-> } or with the newer sablecc release
```

```
42  //    you can just leave it without any rules. See the 'separator' rule.
43  //
44  //  * You can't place null-s into lists. When the expression is null
45  //    (either by ?) or directly set in transformation and is later added
46  //    to a list - it is eliminated by SableCC.
47  //
48  //  * With the latest sablecc release '?' and '+' are supported in
49  //    the AST section. They are also enforced from productions.
50  //    You'll see when errors start popping up.
51  //
52  //  * In the product transformation declaration you can similarily use
53  //    renaming in the style 'productname { .. [use_name]:name .. } = ..'
54  //    This can be very useful when using multiple elements of the same type
55  //    at transform. See the 'random_x2' rule for example.
56  //
57  //  Written by Indrek Mandre <indrek (at) mare . ee> in July-August 2003
58  //  Example constructed from the SableCC docs/Kevin Agbakpem and
59  //  Etienne Bergeron e-mail. http://www.mare.ee/indrek/sablecc/
60  //
61
62  Package expression;
63
64  Helpers
65
66      digit = ['0' .. '9'];
67      tab = 9;
68      cr = 13;
69      lf = 10;
70      eol = cr lf | cr | lf;
71
72      blank = (' ' | tab | eol)+;
73
74  Tokens
75      l_par = '(';
76      r_par = ')';
77      plus = '+';
78      minus = '-';
79      mult = '*';
80      div = '/';
81      semi = ';';
82
83      blank = blank;
84      number = digit+;
```

```
85
86      one = 'one';
87      two = 'two';
88      three = 'three';
89
90      random = 'random_digit';
91
92
93  Ignored Tokens
94
95      blank;
96
97  Productions
98
99      grammar           = exp_list                    {-> New grammar ([exp_list.exp])}
100                       ;
101
102     exp_list          {-> exp*} =
103                         {list}    exp_list separator exp {-> [exp_list.exp, exp.exp] }
104                       | {single}  exp                {-> [exp.exp] }
105                       ;
106
107     exp               {-> exp} =
108                         {plus}    exp plus factor    {-> New exp.plus (exp.exp, factor.exp) }
109                       | {minus}   exp minus factor   {-> New exp.minus (exp.exp, factor.exp) }
110                       | {factor}  factor             {-> factor.exp }
111                       ;
112
113     factor            {-> exp} =
114                         {mult}    factor mult term   {-> New exp.mult (factor.exp, term.exp) }
115                       | {div}     factor div term    {-> New exp.div (factor.exp, term.exp) }
116                       | {term}    term               {-> term.exp }
117                       ;
118
119     term              {-> exp} =
120                         {number}  number             {-> New exp.number(number) }
121                       | {exp}     l_par exp r_par    {-> exp.exp }
122                       | {textual} textual+           {-> New exp.textual ([textual]) }
123                       | {random_x2} random_x2
124      {-> New exp.random_x2 (random_x2.ran1, random_x2.ran2) }
125                       ;
126
127     textual           =
```

```
128                         {t1}       one
129                       | {t2}       two
130                       | {t3}       three
131                       ;
132
133     random_x2         {-> [ran1]:random [ran2]:random} =
134                         [ran1]:random [ran2]:random {-> ran1 ran2 }
135                       ;
136
137     separator  {-> } =
138                         {semicolon} semi {-> }
139                       ;
140
141
142 Abstract Syntax Tree
143
144     grammar           = exp+
145                       ;
146
147     exp               =
148                         {plus}     [l]:exp  [r]:exp |
149                         {minus}    [l]:exp  [r]:exp |
150                         {div}      [l]:exp  [r]:exp |
151                         {mult}     [l]:exp  [r]:exp |
152                         {textual}  textual+ |
153                         {random_x2} [r1]:random [r2]:random |
154                         {number}   number
155                       ;
156
157     textual            =
158                         {t1}       one
159                       | {t2}       two
160                       | {t3}       three
161                       ;
162
163 //
164 // A few words about this grammar itself:
165 //  - It is supposed to be a little integer based calculator with a few odd
166 //    extensions to demonstrate sablecc transformations
167 //  - You can use textual words to build up numbers (two one three -> 213)
168 //    I didn't really bother to specify all the decimal textual numbers
169 //  - The random number rule is a bit superficial, it just expects user to
170 //    type 'random_digit random_digit' and produces a two-digit random
```

```
171 //    number.  I didn't figure out any better way to make the multiple
172 //    element transform rule "interesting" ;)
173 //
174 //  Valid expressions:
175 //    (1 + 14  / (3 + 4)) * 14          -> 42
176 //    one + 3 - two                     -> 2
177 //    two one + three                   -> 24
178 //    random_digit random_digit         -> ??
179 //    random_digit random_digit + 1     -> ??
180 //    1 + 3 ; 1 ; 4 + 5                 -> 4; 1; 9
181 //
182 // In the Calculate.java is the implementation of the tree visitor that
183 // calculates the values.
184 //
```

Appendix B. Modified Simple Arithmetic Expressions Grammar

```
1  //
2  // This is a demonstration grammar file with transformation rules for the
3  // new SableCC3 parser generator (rel sablecc-3-beta.3.altgen.20040327)
4  //
5  // Points to remember:
6  //
7  //  * Why CST and AST? Due to limitations of parser technology the
8  //    human described grammar does not represent the "perfect" abstract
9  //    form of the language parsed. To get the AST transformations are done.
10 //    (AST - Abstract Syntax Tree; CST - Concrete Syntax Tree)
11 //
12 //  * The AST section must be complete, nothing from Productions is
13 //    automatically placed there.
14 //
15 //  * In curly braces are things related to AST, eg. we built the AST in
16 //    those things, so everything is built/transformed from the leaves to
17 //    root.
18 //
19 //  * Transformation is divided into two parts - product transformation
20 //    declaration that declares what the alternatives should be
21 //    transformed to and alternatives transformations definitions that
22 //    define how the transform is actually done.
23 //
24 //  * A production can be transformed to multiple elements as seen in the
25 //    'random_x2' rule.
26 //
```

```
27  //  * Productions with same structure as in the AST can be directly
28  //    transformed to AST - see the 'textual' rule. What really happens is
29  //    that for productions and alternatives without transform specification
30  //    default transformation rules are generated. This also means that you
31  //    could omit the transform declaration at exp rule.
32  //
33  //  * Lists it seems are represented with brackets '[ elem1, elem2, .. ]' and
34  //    not parenthesis as described in the doc. If element is also a list it
35  //    is automatically expanded and used. Empty list is [].
36  //
37  //  * The output we get from the parser is as described in the AST. We only
38  //    have to work with that. The productions section is no longer used.
39  //
40  //  * When you just want to get rid of a production declare and define
41  //    it and its alternatives as {-> } or with the newer sablecc release
42  //    you can just leave it without any rules. See the 'separator' rule.
43  //
44  //  * You can't place null-s into lists. When the expression is null
45  //    (either by ?) or directly set in transformation and is later added
46  //    to a list - it is eliminated by SableCC.
47  //
48  //  * With the latest sablecc release '?' and '+' are supported in
49  //    the AST section. They are also enforced from productions.
50  //    You'll see when errors start popping up.
51  //
52  //  * In the product transformation declaration you can similarily use
53  //    renaming in the style 'productname { .. [use_name]:name .. } = ..'
54  //    This can be very useful when using multiple elements of the same type
55  //    at transform. See the 'random_x2' rule for example.
56  //
57  //  Written by Indrek Mandre <indrek (at) mare . ee> in July-August 2003
58  //  Example constructed from the SableCC docs/Kevin Agbakpem and
59  //  Etienne Bergeron e-mail. http://www.mare.ee/indrek/sablecc/
60  //
61
62  Package expression;
63
64  Helpers
65      all = [0 .. 0xffff];
66      digit = ['0' .. '9'];
67      tab = 9;
68      cr = 13;
69      lf = 10;
```

```
70      eol = cr lf | cr | lf;
71
72      blank = (' ' | tab | eol)+;
73
74  States
75      normal,php;
76
77  Tokens
78      {normal->php, php} php_start = '<?php ';
79      {php} php_body =[all - ['?' + '>']]*;
80      {php->normal, normal} php_end = '?>';
81      {normal} l_par = '(';
82      {normal} r_par = ')';
83      {normal} plus = '+';
84      {normal} minus = '-';
85      {normal} mult = '*';
86      {normal} div = '/';
87      {normal} semi = ';';
88
89      {normal} blank = blank;
90      {normal} number = digit+;
91
92      {normal} one = 'one';
93      {normal} two = 'two';
94      {normal} three = 'three';
95
96      {normal} random = 'random_digit';
97
98
99  Ignored Tokens
100
101     blank;
102
103 Productions
104
105     grammar           = exp_list                    {-> New grammar ([exp_list.exp])}
106                       ;
107
108     exp_list          {-> exp*} =
109                         {list}    exp_list separator exp {-> [exp_list.exp, exp.exp] }
110                       | {single}  exp                    {-> [exp.exp] }
111                       ;
112
```

```
113     exp               {-> exp} =
114                         {plus}    exp plus factor   {-> New exp.plus (exp.exp, factor.exp) }
115                       | {minus}   exp minus factor  {-> New exp.minus (exp.exp, factor.exp) }
116                       | {factor}  factor            {-> factor.exp }
117                       | {php}     php_start php_body php_end {-> New exp.php(php_body)}
118                       ;
119
120     factor            {-> exp} =
121                         {mult}    factor mult term  {-> New exp.mult (factor.exp, term.exp) }
122                       | {div}     factor div term   {-> New exp.div (factor.exp, term.exp) }
123                       | {term}    term              {-> term.exp }
124                       ;
125
126     term              {-> exp} =
127                         {number}  number            {-> New exp.number(number) }
128                       | {exp}     l_par exp r_par   {-> exp.exp }
129                       | {textual} textual+          {-> New exp.textual ([textual]) }
130                       | {random_x2} random_x2
131      {-> New exp.random_x2 (random_x2.ran1, random_x2.ran2) }
132                       ;
133
134     textual           =
135                         {t1}      one
136                       | {t2}      two
137                       | {t3}      three
138                       ;
139
140     random_x2         {-> [ran1]:random [ran2]:random} =
141                         [ran1]:random [ran2]:random {-> ran1 ran2 }
142                       ;
143
144     separator  {-> } =
145                         {semicolon} semi {-> }
146                       ;
147
148
149 Abstract Syntax Tree
150
151     grammar           = exp+
152                       ;
153
154     exp               =
155                         {plus}    [l]:exp  [r]:exp |
```

```
156                         {minus}    [l]:exp   [r]:exp  |
157                         {div}      [l]:exp   [r]:exp  |
158                         {mult}     [l]:exp   [r]:exp  |
159                         {textual} textual+ |
160                         {random_x2} [r1]:random [r2]:random |
161                         {number}   number |
162                         {php} php_body
163                       ;
164
165     textual            =
166                         {t1}       one
167                       | {t2}       two
168                       | {t3}       three
169                       ;
170
171 //
172 // A few words about this grammar itself:
173 //  - It is supposed to be a little integer based calculator with a few odd
174 //    extensions to demonstrate sablecc transformations
175 //  - You can use textual words to build up numbers (two one three -> 213)
176 //    I didn't really bother to specify all the decimal textual numbers
177 //  - The random number rule is a bit superficial, it just expects user to
178 //    type 'random_digit random_digit' and produces a two-digit random
179 //    number.  I didn't figure out any better way to make the multiple
180 //    element transform rule "interesting" ;)
181 //
182 //  Valid expressions:
183 //    (1 + 14  / (3 + 4)) * 14          -> 42
184 //    one + 3 - two                     -> 2
185 //    two one + three                   -> 24
186 //    random_digit random_digit         -> ??
187 //    random_digit random_digit + 1     -> ??
188 //    1 + 3 ; 1 ; 4 + 5                 -> 4; 1; 9
189 //
190 // In the Calculate.java is the implementation of the tree visitor that
191 // calculates the values.
192 //
```

Appendix B. Kermeta

Appendix B.1. Metamodel Logo with Kermeta

Example can be found at `http://www.kermeta.org/documents/tutorials/building_dsl_tutorials/logo_tutorial/tutorial_single` in section "3.2. Metamodel Ecore with Kermeta".

```
1  @uri "http://www.kermeta.org/kmLogo"
2  package kmLogo;
3
4  require "kermeta"
5  alias Integer : kermeta::standard::Integer;
6  alias Boolean : kermeta::standard::Boolean;
7  alias String : kermeta::standard::String;
8  package ASM
9  {
10  abstract class Instruction
11  {
12  }
13  abstract class Primitive inherits Instruction
14  {
15  }
16  class Back inherits Primitive
17  {
18   attribute steps : Expression[1..1]
19
20  }
21  class Forward inherits Primitive
22  {
23   attribute steps : Expression[1..1]
24
25  }
26  class Left inherits Primitive
27  {
28   attribute angle : Expression
29
30  }
31  class Right inherits Primitive
32  {
33   attribute angle : Expression
34
35  }
36  class PenDown inherits Primitive
37  {
38  }
39  class PenUp inherits Primitive
40  {
```

```
41  }
42  class Clear inherits Primitive
43  {
44  }
45  abstract class Expression inherits Instruction
46  {
47  }
48  abstract class BinaryExp inherits Expression
49  {
50   attribute lhs : Expression[1..1]
51
52   attribute rhs : Expression[1..1]
53
54  }
55  class Constant inherits Expression
56  {
57   attribute integerValue : Integer
58
59  }
60  class ProcCall inherits Expression
61  {
62   attribute actualArgs : Expression[0..*]
63
64   reference declaration : ProcDeclaration[1..1]#procCall
65
66  }
67  class ProcDeclaration inherits Instruction
68  {
69   attribute name : String
70
71   attribute args : Parameter[0..*]
72
73   attribute block : Block
74
75   reference procCall : ProcCall[0..*]#declaration
76
77  }
78  class Block inherits Instruction
79  {
80   attribute instructions : Instruction[0..*]
81
82  }
83  class If inherits ControlStructure
84  {
85   attribute thenPart : Block[1..1]
86
87   attribute elsePart : Block
```

```
 88
 89  }
 90  class ControlStructure inherits Instruction
 91  {
 92   attribute condition : Expression
 93
 94  }
 95  class Repeat inherits ControlStructure
 96  {
 97   attribute block : Block[1..1]
 98
 99  }
100  class While inherits ControlStructure
101  {
102   attribute block : Block[1..1]
103
104  }
105  class Parameter
106  {
107   attribute name : String
108
109  }
110  class ParameterCall inherits Expression
111  {
112   reference parameter : Parameter[1..1]
113
114  }
115  class Plus inherits BinaryExp
116  {
117  }
118  class Minus inherits BinaryExp
119  {
120  }
121  class Mult inherits BinaryExp
122  {
123  }
124  class Div inherits BinaryExp
125  {
126  }
127  class Equals inherits BinaryExp
128  {
129  }
130  class Greater inherits BinaryExp
131  {
132  }
133  class Lower inherits BinaryExp
134  {
```

```
135   }
136   class LogoProgram
137   {
138    attribute instructions : Instruction[0..*]
139
140   }
141  }
```

Appendix B.2. Logo Constraints with Kermeta

Example can be found at `http://www.kermeta.org/documents/tutorials/building_dsl_tutorials/logo_tutorial/tutorial_single` in section "6.2. Implementation in Kermeta".

```
 1  package kmLogo::ASM;
 2  require kermeta
 3  require "http://www.kermeta.org/kmLogo"
 4
 5  aspect class ProcDeclaration{
 6   /**
 7    * No two formal parameters of a procedure may have the same
        name
 8    */
 9   inv unique_names_for_formal_arguments is
10   do
11    args.forAll{ a1 | args.forAll{ a2 |
12     a1.name.equals(a2.name).implies(a1.equals(a2))}}
13   end
14  }
15
16  aspect class ProcCall{
17   /**
18    * A procedure is called with the same number
19    * of arguments as specified in its declaration
20    */
21   inv same_number_of_formals_and_actuals is do
22    actualArgs.size == declaration.args.size
23   end
24  }
```

Appendix C. Extended Parser and Lexer

```
"""
LEXER
"""
class Extended_Lexer(ActionLexer):
  tokens = ActionLexer.tokens + ['CROSSES_BELOW', '
      CROSSES_ABOVE', 'INSTATE']

  def t_CROSSES_BELOW(self, t):
    r'\>!'
    return t

  def t_CROSSES_ABOVE(self, t):
    r'\<!'
    return t

"""
PARSER
"""
class Extended_Parser(ActionParser):
  precedence = ActionParser.precedence + (('nonassoc', '
      CROSSES_BELOW', 'CROSSES_ABOVE'),)

  def p_expr_crosses_below(self, p):
    'expr : expr CROSSES_BELOW expr'
    p[0] = ASTCrossesFromBelow(p[1], p[3], p.lineno)

  def p_expr_crosses_above(self, p):
    'expr : expr CROSSES_ABOVE expr'
    p[0] = ASTCrossesFromAbove(p[1], p[3], p.lineno)

"""
AST_VISITOR
"""
class Extended_Visitor(AST_Visitor):
  def __init__(self, debug=None, typevisitor=None,
      pythonvisitor=None):
    AST_Visitor.__init__(self, debug=debug, typevisitor=
        typevisitor, pythonvisitor=pythonvisitor)
    self.crosses = set()

  def _generate_crosses_id(self):
    id_length = 6
    id = self.metaverse._id_generator(id_length)
    while id in self.crosses:
        id_length += 1
```

```
42          id = self.metaverse._id_generator(id_length)
43      self.crosses.add(id)
44      return id
45
46    def visit_ASTCrossesFromBelow(self, node):
47      id = StringValue(self._generate_crosses_id())
48      return CrossesFromBelow(self._visit(node.getLhs()), self.
            _visit(node.getRhs()), id)
49
50    def visit_ASTCrossesFromAbove(self, node):
51      id = StringValue(self._generate_crosses_id())
52      return CrossesFromAbove(self._visit(node.getLhs()), self.
            _visit(node.getRhs()), id)
53
54  """
55  AST NODES
56  """
57  class ASTCrossesFromBelow(BinaryOperator):
58      def __repr__(self):
59          return "CrossesFromBelow(%s, %s)" % (repr(self.lhs),
                repr(self.rhs))
60
61  class ASTCrossesFromAbove(BinaryOperator):
62      def __repr__(self):
63          return "CrossesFromAbove(%s, %s)" % (repr(self.lhs),
                repr(self.rhs))
64  """
65  ARKM3 NODES
66  """
67  class CrossesFromBelow(Action.ComparisonOp):
68      def __init__(self, child1=None, child2=None, name=None,
            container=None, init_id=None):
69          Action.ComparisonOp.__init__(self, child1=child1,
                child2=child2, container=container, init_id=init_id)
70          self._id = name
71
72  class CrossesFromAbove(Action.ComparisonOp):
73      def __init__(self, child1=None, child2=None, name=None,
            container=None, init_id=None):
74          Action.ComparisonOp.__init__(self, child1=child1,
                child2=child2, container=container, init_id=init_id)
75          self._id = name
76
77  """
78  PYTHON COMPILE VISITOR
79  """
80  class Extended_Python_Visitor(PythonCompileVisitor):
```

```
81      def visit_CrossesFromBelow(self, node):
82          self._add_abs_import("hutn.extended.ExtendedParser")
83          self._write_chars("P_CrossesFrom.get_instance(%s).
                checkBelow("%node._id)
84          self.visit(node._child[0])
85          self._write_chars(",")
86          self.visit(node._child[1])
87          self._write_chars(")")
88
89      def visit_CrossesFromAbove(self, node):
90          self._add_abs_import("hutn.extended.ExtendedParser")
91          self._write_chars("P_CrossesFrom.get_instance(%s).
                checkAbove("%node._id)
92          self.visit(node._child[0])
93          self._write_chars(",")
94          self.visit(node._child[1])
95          self._write_chars(")")
96
97  # >!
98  class P_CrossesFrom(object):
99      instances = dict()
100
101     @staticmethod
102     def get_instance(instance_id):
103         cls = P_CrossesFrom
104         if cls.instances.has_key(instance_id):
105             return cls.instances[instance_id]
106         else:
107             i = cls()
108             cls.instances[instance_id] = i
109             return i
110
111     def __init__(self):
112         self.value = VoidValue()
113         self.prev = VoidValue()
114
115     def checkBelow(self, value, value2):
116         if self.prev == VoidValue():
117             self.prev = value
118             self.value = value2
119             return False
120         elif self.prev <= self.value and value > self.value:
121             self.prev = value
122             return True
123         self.prev = value
124         return False
125
```

```
126     def checkAbove(self, value, value2):
127         if self.prev == VoidValue():
128             self.prev = value
129             self.value = value2
130             return False
131         elif self.prev >= self.value and value < self.value:
132             self.prev = value
133             return True
134         self.prev = value
135         return False
136 
137 
138 """
139 EXTENDED HUTN
140 """
141 class ExtendedHUTN(HUTN):
142     def __init__(self,path_to_arkm3_mm = "../../../models/arkm3
            /"):
143         HUTN.__init__(self, actionparser=Extended_Parser,
              visitor=Extended_Visitor, actionlexer=Extended_Lexer
              , pythonvisitor=Extended_Python_Visitor,
              path_to_arkm3_mm=path_to_arkm3_mm)
```