1 Domain Specific Languages

In this project a domain-specific language (short DSL) has been developed to simplify the expression of mapping between two models. In this section a short introduction to the field of domain-specific languages will be given.

1.1 Introduction

DSLs are an important part of the toolkit that is information technology. Users who dare not write a program in any general programming language (GPL) are happily writing DSL code in their Spreadsheets. Mathematicians have found their place in the world of programming by using software such as MATLAB or Mathematica. Even for the advanced programmer it makes a big difference whether he is forced to use a GPLs like C for any small problems or if he can side step the GPL and use a DSL such as regular expressions or for example one of the unix mini languages (sed, grep, ...).

A domain-specific language is defined in contrast of general programming languages (GPLs). While the exact definitions vary across the literature, there are a few key concepts, that are commonly held to identify a DSL. A DSL is a codified language having as its target a single kind of problem (the domain). As a consequence a DSL has the form of a very specific notation and a limited expressiveness [4]. A domain language can, but does not necessarily have to, be executable. A DSL can be used to extract data, process data, format output, model a situation, etc.

Before initiating the development of a DSL, the advantages and disadvantages should be considered. The main goal of introducing a DSL is to arrive at a better fit between the logic as written in the program code and the logic in the minds of the domain experts. Would a DSL be a true mapping to a domain it should be easily understandable and even writable by those experts. (In praxis this fit is seldomly achieved.) Additionally, as the domain logic is not bogged down by the details of the process logic, it will require less lines of code and hence arrive at greater robustness.

On the other side, a DSL induces a good deal of costs [6]. On the one hand there is the cost for the development of the DSL and on the other the cost for training users of the DSL. Even after the initial development and training the DSL will induce costs through maintenance and extensions. (In contrast to any GPL the user group of DSLs is mostly very small. As such changes the cost of improving the language is not distributed as broadly as with a GPL.) A list of all the advantages and drawbacks is listed in table 1.

1.2 Phases of DSL development

The development of a DSL follows four phases: Decision, Analysis, Design, Implementation. The first phase is nothing but the question whether the creation of a DSL is feasible and the decision to do so. (For decision patterns please look into [7].)

The analysis phase comes close to the field of domain engineering. The domain and its borders have to be identified and the knowledge has to be aggregated. As a prerequisite for designing a DSL it is necessary to have extensive knowledge of the domain.

Advantages	Disadvantages
✓readable by domain experts	✗costs of implementation and design
✓(possibly) writable by domain experts	✗difficulty of finding the right scope
✓self-documenting	✗requires additional training
✓shorter, hence less bugs	✗language can suffer from isolation
✓enable conservation of domain knowledge	
✓validation at domain level	

Table 1: DSL Advantages and Disadvantages

The understanding of legacy systems has a particular important role. To reach an understanding of the domain three options are available. Either an informal approach is used, that is the domain is simply described. Or in a formal approach, the domain is researched with a domain analysis methodology. Or, finally, the domain knowledge is mined from existing code bases.

In the design phase, the knowledge is formalized to create notions and operations. Those are used to design the language. The steps taken differ according to whether a new language is invented or an existing one is exploited. In the former case there are nearly no constraints on the format of the language other than the expectancy of familiarity by the users. In the latter, the DSL needs to follow the existing language. In table 2 the common design patterns are listed.

Language exploitation	
Piggyback	The DSL is implemented in an existing language. (e.g. as a collection of interwoven objects)
Specialization	The existing language is restricted for use as a DSL.
Extension	The existing language is extended with libraries, overloading of operators, etc.
Language invention	

Table 2: DSL design patterns [7]

In the implementation the chosen DSL is realized in code. Depending on the choice in the design phase one of several implementation patterns can be used. See table 3. Still most DSLs end up as libraries of functions and objects [10].

1.3 Choice

In this project an informal analysis was chosen, followed by language invention and the development of the compiler. Three legacy systems have been analyzed for the creation of the DSL (see chapter Related Work).

Language invention was chosen for the design phase. The language evolved in parallel with new insights into the domain and problem. It was also planned from the beginning

Language exploitation	
Macro processing	Macros are added to the base language.
Source-to-source transformation	dsl language is transformed to base language.
Pipeline	A dsl is successively translated till the base language is reached.
Lexical processing	Only lexical scanning, no parsing.
Embedding	DSL notions and operations are directly implemented as constructs in a GPL.
Extensible compiler/interpreter	The compiler of the base language is extended.
Commercial Off-The-Shell (COTS)	An existing solution is sought and applied.
Language invention	
Interpreter	DSL statements are executed in a step-by-step fashion without the generation of intermediate code.
Compiler	The DSL is decoded into base language and then executed.

Table 3: DSL implementation patterns [7]

to hand the user a very simple language closely mapped to the domain. Language exploitation with its lesser development cost was discarded in favor of more freedom in the design of the language.

In the implementation phase it was chosen to build a compiler. The reason being that the product of the DSL, a data structure created from a template, was easier to build with a compiler (one step translation) than with an interpreter (stepwise translation).

An alternative would have been to exploit the host language, python, using the piggyback approach and implement the DSL via embedding. This approach would require less effort in compiler development, but would restrict the syntax mostly to encapsulated python objects and function calls. An impediment on the reading and writing of the user.

As part of the chosen approach a compiler had to be developed. As this is a very time consuming and possibly complex task, the next section is devoted to this topic.

2 Compiler

In the scope of the project a custom language had to be created including a compiler [8]. In the next section a compiler will be introduced. An overview can be found in figure 1. Subsequent tools will be introduced that simplify the construction of a lexer (alternatively called tokenizer) and parser.

2.1 Compiler end to end

Input string The input to a compile is a basic string of some kind (numbers, letters, bits).

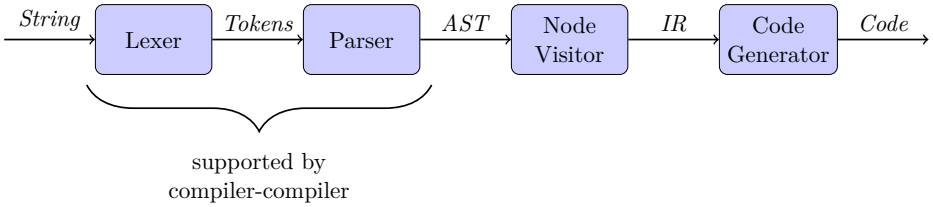

Figure 1: Compiler End-to-end

Listing 1: input string

```
foo = 5 + 3
```

Lexer As a first step the string is read and smallest units (tokens) are identified. Definitions of the tokens are most commonly expressed in form of regular expressions. A token consists at least of a name and an argument. The argument is the string matching the token specification

Listing 2: Tokenizer rules

```
NAME := 'a-z'
EQ := '='
NUM := '-?[0-9]+'
PLUS := '+'
WS := ' '
```

Listing 3: Tokenizer output

```
NAME EQ NAME NUM PLUS NUM
```

Match & Derive

The basic constructs of the compiler are the lexer and parser. The lexer expresses smallest units (lexemes or tokens) in the form of regular expressions and the parser expresses the possible combination of tokens.
Together they can either fit an existing string, so called "matching". Or they can be used to construct such a string, that is called "derivation".

Parser In the next step the tokens are organized into a tree according to a grammar, a structured set of rules.

Listing 4: Parser rules

```
<root> ::= <assignment> | <expr>
<assignment> ::= NAME EQ <expr>
<expr> ::= NUM | NUM PLUS <expr>
```

Listing 5: Parser output: Parse tree

```
assignment
+-- var
    + -- NAME('foo')
+-- EQ('=')
+-- expr
    +-- NUM('5')
    +-- PLUS('+')
    +-- NUM('3')
```

Grammar in Backus-Naur Form

Grammar is conventionally expressed in the BNF or one of it's derivatives (Extended BNF or Augmented BNF) [1].
A rule in BNF is segmented into a left-hand side and a right-hand side separated by ":=". The rule expresses that the single symbol on the left-hand side can be rewritten by the symbols and terminals on the right-hand side. Symbols or non-terminals are written with pointy parentheses ("<" and ">"). Terminals are either tokens which are written with all upper case letters or literals, that is strings, which are surrounded by quotes.

```
<two_words> ::= WORD ',' WORD
```

The rule above matches two words. (The token WORD is not shown.) To make a rule more general the "|" allows a rule to have multiple options.

```
<sentence> ::= <words> <mark>
<words> ::= WORD <words> | WORD
<mark> ::= '.' | '?' | '!'
```

This rule can match any number of words, followed by a dot, exclamation mark or question mark. All together the BNF allows the specification of any context free grammar. The Extended BNF cannot express any rule that the BNF cannot express. However the EBNF syntax includes more structures which allow a shorter expression of most grammars [9].

Abstract Syntrax Tree During the parsing an abstract representation of the parsing tree is created. In contrast to the parsing tree it does not contain all the details of the string.

Listing 6: Abstract Syntax Tree

```
assignment
+-- variable('foo')
+-- add('add')
    +-- Integer(5)
```

```
    +— Integer(3)
```

Node Visitor The AST has to be walked at least once for the generation of an output. The node visitor defines what operation are executed on every node on the tree.

Listing 7: Pseudo Code Node visitor

```
visit_assignment(node, symbol_table):
    name = node.children[0]
    arg = visit_function(node.children[1])
    symbol_table[name] = arg

visit_add(node):
    return node.children[1] + node.children[2]
```

Symbol Table A symbol table keeps track of all symbols that are assigned values. In a nested environment it differentiates between the symbol's value in the different levels of nesting.

Listing 8: Symbol table contents

```
>>> symbol_table = Dictionary()
... visit_assignment(ast, symbol_table)
... print symbol_table
... "{'foo': 8}"
```

Output generation The final step is the output generation. Would the aim be to generate program code, this phase would consist of finding the right byte codes. Instead the goal here is to generate a simple string:

Listing 9: Pseudo code template rendering

```
>>> template = "I believe ${name} foo is exactly '${value}'"
... render(template, name=symbol_table.keys()[0], value=symbol_table.values()[0]
... "I believe foo is exactly '8'"
```

Types The basic categorization of data is based on a type hierarchy. For example values could be differentiated into numbers and strings, unsigned and signed numbers, hexadecimal and binary numbers etc. To structure basic functions operating over a type (e.g. identifying the length of a string), a common approach is to visibly associate them with the type. Either by implementing as a part of a library that shares the name with the type. (The "string" library in Erlang) Or by implementing the type as an object and the methods as methods of this object. (In this example the type system is avoided and instead the calculation is done in the node visitor.

To sum it up. The input is received as a string. It is then converted to a sequence of tokens. These tokens are structured based on a grammar. The structure is cleaned to become an abstract syntax tree.

The AST is walked to execute the calculations. Variables appearing in the tree are stored in a symbol table (in the simplest case a dictionary). Finally an output is generated via a rendered template.

2.2 Compiler construction tools

In the compiler the development of the tokenizer and parser require the most effort. As such they have received a great deal of attention. So called compiler-compilers, also called parser generators, are supposed to ease the parser and lexer construction.

The most famous compiler-compilers are the combination of the YACC and LEX tools. YACC (*Yet Another Compiler Compiler*) is a LALR parser generator written by Stephen C. Johnson at AT&T in 1970. LEX (*LEXical Analyzer*) is a tokenizer written by Mike Lesk and Eric Schmidt in 1975 and is used in combination with YACC.

As the compiler had to interface with pyang in this project only compiler-compilers producing compilers in python are relevant. Even among those a great number of parsers generators have already been developed [11]. Among the many three generators stand out. PLY and Pyparsing are recommended in the advanced python literature (e.g. in [3]). ANTLR is a java parser generator which can generate parsers for python that are beyond the typical capabilities of a parser generator.

LL & LR parsers

Parsers conventionally parse input from left to right (following the basic pattern of the English language which program code inherited from). Iteratively more and more of the input string is put on a stack which is rewritten based on a set of rules. Parsers differ in their direction of derivation.
Left deriving (LL) parsers first match the most left substring. As such they start at the beginning of the string and in top-down manner match more and more of the string till either a parse tree can be constructed from the whole string or an error has been encountered. Top-down LL parsers in the form or recursive descent parsers can be constructed by hand. Care has to be taken in the construction of the grammar. Rules that recurse on themselves before reading additional input lead to endless loops.

```
# endless loop
<rule> ::= <rule> 'b' | 'a'

# fixed grammar
<rule> ::= 'a' <rule2>
<rule2> ::= 'b' <rule2> | 'b'
```

Right deriving (LR) parsers always match the biggest substring of the string

from right to left before reading more input. This process of bottom-up parsing is notoriously hard to implement hence the early adaptation of the compiler-compiler Yacc.
Comparing LL with LR parsers it has been established, that LL parsers are simpler to build, but less expressive than LR parsers. LR parsers do not have to deal with left recursion, but are more difficult to debug. However a multitude of compiler-compilers exist, which ease the process of developing a LL or LR parser enormously. In this case the choice of using an LL or a LR parser depends on the ease of the available tools [5].

	LL parser	**LR parser**
Construction	Easy	Hard
Left-recursion	No	Yes
Debugging	Easy	Hard

In the following the compiler-compilers named above will be introduced. The goal is to show how a lexer and parser can be implemented for the toy grammar in listing 10.

Listing 10: "A toy grammar"

```
<root> := <assign> | <expr>
<assign> := ID '=' <expr>
<expr> := NUM OP <expr> | NUM

ID = '[a-zA-Z][a-z_0-9]*'
NUM = '[0-9]+'
```

2.2.1 PLY

PLY stands for "Python Lex-Yacc" and this is exactly what it is. It tries to mimic the functionality of lex and yacc as close as possible while maintaining its own syntax. It was conceived by David Beazley in 2001 for an introductory course to compilers [2].

Example of a tokenizer in PLY:

Listing 11: PLY Tokenizer

```
from ply import lex

t_ID = r'[a-zA-Z][a-z_0-9]*'
t_PLUS = r'\+'
t_NUM = r'\d+'
t_EQ = r'='
t_ignore = r' '
tokens = ['ID', 'PLUS', 'NUM', 'EQ']

lexer = lex.lex()
```

The tokenizer expects the tokens to be defined as variables or functions (the choice influences the order of tokens to be matched). Line number and position can be accessed via the token.

The respective parser follows:

Listing 12: PLY Yacc

```
from ply import yacc

def p_root(p):
    """ root : assign
             | expr """
    p[0] = p[1]

def p_assign(p):
    """ assign : ID EQ expr """
    p[0] = ('assign', p[1], p[3])

def p_expr_plus(p):
    """ expr : NUM PLUS expr """
    p[0] = ('plus', p[1], p[3])

def p_expr_num(p):
    """ expr : NUM """
    p[0] = p[1]

parser = yacc.yacc()
```

The parser makes use of the Python doc string, the first comment in a Python function or class. PLY has the advantage of being readable to the users of the most widely used YACC & LEX.

2.2.2 Pyparsing

Pyparsing was created by Paul McGuire in 2006 as an alternative to the conventional YACC parsing. The grammar is created by combining multiple parsing objects [11].

Listing 13: Pyparsing Token

```
from pyparsing import Word, alphas, Literal, alphanums, nums

# Tokens
plus = Literal('+').suppress()
equal = Literal('=').suppress()
name = Word( alphas, alphanums + "_" )
number = Word( nums )
```

The literals “+” and “=” are suppressed, so that they do not show up in the final parse result.

Listing 14: Pyparsing Parser

```
from pyparsing import Forward
# Parsing

def fun_plus(x, y, toks):
    return ("plus", toks[0], toks[1])

def fun_assign(toks):
    return ("assign", toks[0], toks[1])
def fun_plus(x, y, toks):
    return ("plus", toks[0], toks[1])

def fun_assign(toks):
    return ("assign", toks[0], toks[1])

expr = Forward() # allows recursion
expr2 = (expr | number)
expr << (number + plus + expr2) # recurse
expr.setParseAction(fun_plus)

assign = (name + equal + expr)
assign.setParseAction(fun_assign)

root = ( assign | expr )

print(root.parseString(string)[0])
print(root.parseString(string2)[0])
```

The parser of the pyparsing is easily used to extract or transform information. Creating an tuple-based AST complicates its structure a bit and makes the functions above necessary.

2.2.3 ANTLR

Terence Parr has first released ANTLR (ANother Tool for Language Recognition) in 1992. It is remarkable for generating a parser from a minimal grammar:

Listing 15: ANTLR grammar

```
grammar foo;

options {
  language = Python;
}

root : assign | expr;
assign : ID '=' expr;
expr : NUM | NUM '+' expr;
```

```
ID : ('a'..'z' | 'A'..'Z')+;
NUM : '0'..'9'+;
WS : ' ' {$channel=HIDDEN;}; // $channel is used to hide whitespaces
```

ANTLR even supports the inclusion of AST construction and of tree walking inside of its grammar. As such it is very attractive framework for developing a DSL. Its main disadvantage is that the generation of a Python parser is still in beta phase. ANTLR's main target is to create Java parsers. Yet many other targets, one of them being Python exist.

2.2.4 Choice

For this project PLY was chosen. Even though the expression of the grammar is more lengthy. Programmers used to compiler construction have definitely seen yacc & lex in their career. The other compiler-compilers are less well-known.

References

[1] Alfred Aho, Monica Lam, Ravi Sethi, and JD Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition, 2007.

[2] David Beazley. Ply (python lex-yacc). `http://www.dabeaz.com/ply/`, July 2014.

[3] David Beazley and Brian K. Jones. *Python Cookbook*. " O'Reilly Media, Inc.", 3 edition, 2013.

[4] Martin Fowler. Domainspecificlanguage. `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html`, August 2014.

[5] Lars Marius Garshol. Bnf and ebnf: What are they and how do they work? `http://www.garshol.priv.no/download/text/bnf.html`, July 2014.

[6] Tomaž Kosar, Pablo A Barrientos, Marjan Mernik, et al. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, 2008.

[7] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[8] Terence Parr. *Language implementation patterns: create your own domain-specific and general programming languages*. Pragmatic Bookshelf, 2009.

[9] EBNF Syntaxt Specification Standard. Ebnf: Iso/iec 14977: 1996 (e). *URL http://www. cl. cam. ac. uk/mgk25/iso-14977. pdf*, 1996.

[10] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[11] wiki. python language parsing. https://wiki.python.org/moin/LanguageParsing, July 2014.