MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation: Information and Communication Technologies (ICT)

APDL : Acquire Process Description Language

Author:
Sylvain Julmy

Under the direction of:
Pierre-André Mudry
Institut Systèmes Industriels

External expert:
Charles Papon

Lausanne, HES-SO//Master, June 29, 2017

Information

Contact Information

Author: Sylvain Julmy
Graduate student - Computer Science
University of Applied Sciences Western Switzerland (HES-SO)

Phone: +41 79 949 10 34

Email: sylvain.julmy@gmail.com

Declaration of Honor

I hereby solemnly declare that I have personally and independently prepared this paper. All quotations in the text have been marked as such, and the paper or considerable parts of it have not previously been subject to any examination or assessment.

Sylvain Julmy

Signature

Location, date

Validation

Accepted by the HES-SO // Master (Switzerland, Lausanne) on a proposal from:

Prof. Pierre-André Mudry Thesis project advisor	Prof. Pierre Pompili Departement director
Signature	*Signature*
Location, date	*Location, date*

Contents

CHAPTER 1

Introduction

The Internet of Things (IoT) is one of the most prolific domains in computer science nowadays. There are connected objects everywhere and the amount of data that we receive from them is growing day after day. Everybody wants to measure some environment variables using a simple and cheap device like an Arduino or a Raspberry Pi.

On another hand, people who want to use those devices are often not familiar with the hardware and/or the software parts of an IoT project. For example, a chemistry engineer could be interested to measure some values in the air, but he is absolutely not in the IoT domain.

1.1 Context

Imagine an electrical engineer who needs to create a system which is able to recover data from multiple sensors, like external temperature and barometric pressure. With that data, the engineer needs to compute a new pressure value with the temperature and compare it to the one given by the barometric pressure sensor using graph plot.

The problem for this electrical engineer is the software part. He doesn't know how to send data through the network, recover them and store them into a database for a future plot. On another hand, a software engineer would not know the hardware part.

IoT programming is at some point a merge of some specific domains like Software, Hardware, Telecommunication and Design. The figure 1.1 illustrates this concept of multi-domain project. Are involved in this concept:

- Hardware.
- Software.
- Design.
- Telecommunication.

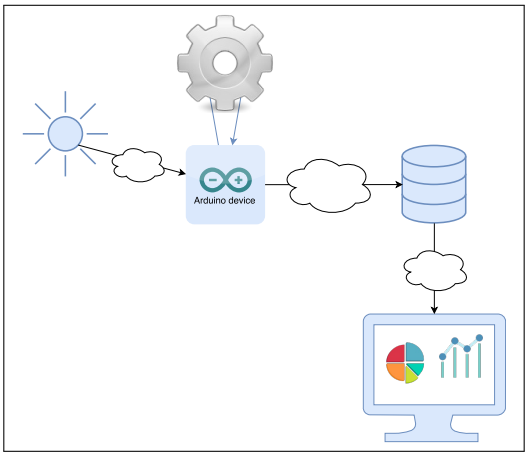


Figure 1.1: Visualisation of the involved domains in the APDL ecosystem : hardware, software, design and telecommunication. And sometimes, a person doesn't know any part of this.

1.2 Problem Statement

According to the previous example we could say that not everybody could develop a project like this from scratch. Maybe some people have the whole knowledge to do this by themselves but not a lambda person.

The major issue with such project is the people diversity. Bringing together people from various professions creates several difficulties : team management, various programming knowledge, different project's perception, and many more.

1.3 Solution Statement

The Acquire and Process Description Language (APDL) ecosystem goal is to provide a simple way to describe such a pipeline. From the sampling of the data from a sensor and its transformation to the display of the result of charts through storing them into a database or send them to a specific server.

1.4 Outline of this thesis

We have set ourselves the challenge of providing a simple way to design specific IoT projects. Before setting out to tackle it, we firstly look at the state of the art in Domain specific language (DSL) development and IoT challenges. Some work has been done in the field of Domain specific frameworks and languages for IoT.

The main technical chapters includes chapter 4, 5, 6. In chapter 4, we present the design of the APDL DSL. We set out the domain of interest and explore the development process of the language. Chapter 5 presents the implementation of the designed language and illustrates the compilation process used by APDL. In chapter 6 we present the validation of the implemented compiler and the generated output based on a set of APDL programs designed for this purpose. We also mention a testing technique called property-based testing which has simplified the parser tests. Finally, in chapter 7, we present the work done about the APDL ecosystem and its utilisation through an example realised from scratch.

CHAPTER 2

Domain specific language

2.1 Introduction

A DSL is a programming language whose goal is to provide a way to solve specific domain problems. For example, Matlab is a domain specific language created to simplify matrix calculus.

What we call "domain" is a set of problems related together in the same scope. For Matlab, those domains could be a matrix multiplication or the resolution of an equation system. Those two operations are related to numerical mathematics and that is the "domain" of Matlab.

This chapter introduces and discusses the question : "Why use a DSL ?" by explaining the advantages that a DSL could bring for domain-oriented problems. Then we present the difference between an internal and an external DSL and the advantages and disadvantages for both of them. Finally, we introduce the implementation of a DSL using the Scala programming language and why Scala is a great tool for the development of DSL.

2.2 Why use a DSL ?

DSL exists since the beginning of computing according to [VKV00], COBOL, Lisp and Fortran have been created for solving problems in a specific area like business processing, numeric computations or symbolic processing. With time, language has become more and more generalist but DSL has been created to simplify some domain specific problems.

Using a DSL instead of a programming language brings advantages as well as disadvantages.

Advantages:

- A high potential of usability and reliability according to [TK10].
- A great learning curve because a DSL is, in general, easier to learn than a general programming language [MHS05].
- An expert of the DSL could modify the application even without knowing how to program.

Disadvantages [VKV00]:

- The cost to develop a DSL : production, user training and maintenance .
- A non-expert user may find hard to modify an existing program.
- In general, a DSL is less efficient than general programming language, because programming languages own some very good compilers with a lot of optimisations.
- The need to learn a new language, even if it's a simple one.
- A DSL doesn't have any tool support (other than a compiler of course) [MHS05].

2.3 Internal and External Domain Specific Language

It exists two types of DSL : internal DSL, also call Domain specific embedded language (DSEL) and external DSL. Understanding the difference, advantages and disadvantages of each of those is crucial in order to create an appropriate solution to the domain specific problems.

The main difference between this two concepts appears at the compilation time, the figure 2.1 shows the compilation of an internal DSL. The input and the output of the compiler are in the same language. Then, the figure 2.2 shows the compilation of an external DSL. This time, we compile the code into another language, called the host language.

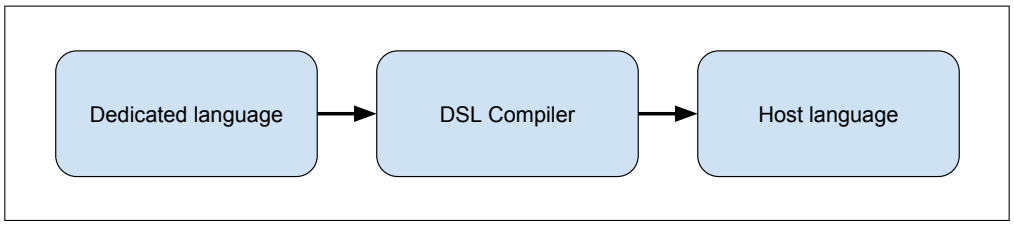


Figure 2.1: Compilation process of an internal Domain specific language. The input and output languages are the same. In general, we could see the input language as a library (with or without syntactic sugar) for more general programming language.

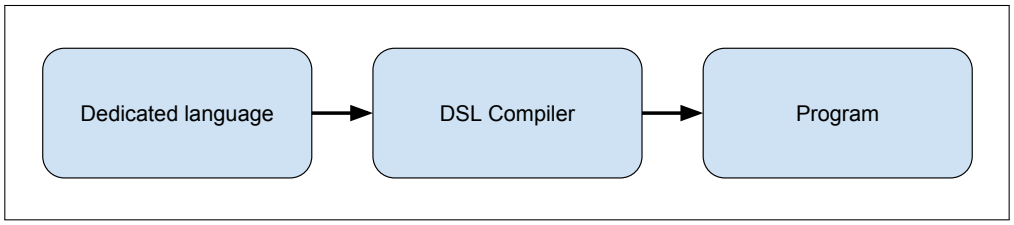


Figure 2.2: Compilation process of an external Domain specific language. The input and output languages aren't the same. This process is closer to a compiler for general language.

2.3.1 Internal Domain Specific Language

As shown in the figure 2.1, the dedicated language and the output language are the same. Internal DSL could be separated into two categories : Deep embedded Domain specific language (Deep embedded DSL) and Shallow embedded Domain specific language (Shallow embedded DSL).

Conceptually, a Shallow embedded DSL captures the semantic notions of the data's domain in various data types and manipulates those data in a fixed way. Alternately, a Deep embedded DSL goes beyond the captures of semantic notions by capturing the operations themselves. As an example for understanding those two concepts we look at listing 2.1. It's a small imaginary DSL used for temperature manipulation between Kelvin and Celsius. We compute the sum of three temperatures, and we want to print it in Celsius.

```
val t1 = Celsius(100)
val t2 = Celsius(50)
val t3 = Kelvin(30)
val t4 = t1 + t2 + t3
println(t4 in Celsius)
```

Listing 2.1: Example of the simple Temperature DSL. We simply want to compute the sum of three temperatures, in Celsius and Kelvin, and finally want to print the result.

Now we are going to compare the simple implementation of this DSL for both Deep embedded DSL and Shallow embedded DSL. The implementation of the Shallow embedded DSL is shown in listing 2.2 and for the Deep embedded DSL in listing 2.3.

```
sealed trait Temperature {
  def +(that : Temperature) = Add(this,that)
  def in(c : Celsius.type) = InCelsius(this)
}
case class Celsius(temp : Double) extends Temperature
case class Kelvin(temp : Double) extends Temperature
case class Add(t1 : Temperature, t2 : Temperature) extends Temperature
case class InCelsius(temperature: Temperature) extends Temperature
```

Listing 2.2: Implementation of the simple Temperature Shallow embedded DSL with Scala, the dedicated language is represented using an AST, the values are never modified.

In the case of a Deep embedded DSL, we are directly using the primitives of the host language. In this case, we use classes to represent a temperature. When we want to combine the temperatures together, we create a new object and we apply the transformation to the existing values as illustrate in the figure 2.4.

```
t1: Celsius = Celsius(100.0)
t2: Celsius = Celsius(100.0)
t3: Kelvin = Kelvin(100.0)
t4: Add = Add(Add(Celsius(100.0),Celsius(100.0)),Kelvin(100.0))

InCelsius(Add(Add(Celsius(100.0),Celsius(100.0)),Kelvin(100.0)))
```

Figure 2.3: Result of the small Temperature program from listing 2.3. The result represents the way to think about a Shallow embedded DSL. We capture the semantic of the computation.

With the Shallow embedded DSL, we save the AST of the computation and we never touch the values. The figure 2.3 shows the output for the Deep embedded DSL. The object is evolving over time.

In general, internal DSL are closely related to the host language and some of them are very good in this work. Languages that offer operator overloading, parametric polymorphism and functional mechanisms give a good user experience and developer satisfaction as well.

```scala
import scala.language.implicitConversions

implicit def C2K(celsius: Celsius): Kelvin = Kelvin(celsius.temp + 273.15)
implicit def K2C(kelvin: Kelvin): Celsius = Celsius(kelvin.temp - 273.15)
implicit def T2K(temperature: Temperature): Kelvin = temperature match {
  case celsius: Celsius => C2K(celsius)
  case kelvin: Kelvin => kelvin
}
implicit def T2C(temperature: Temperature): Celsius = temperature match {
  case kelvin: Kelvin => K2C(kelvin)
  case celsius: Celsius => celsius
}

sealed abstract class Temperature(val tempInCelsius: Double) {
  def in(c: Celsius.type) = T2C(this)
  def in(k: Kelvin.type) = T2K(this)
  def +(that: Temperature): Temperature
}
case class Celsius(temp: Double) extends Temperature(temp) {
  override def +(that: Temperature): Temperature = add(that)

  private def add(that: Temperature): Celsius = {
    Celsius(T2C(that).temp + temp)
  }
}
case class Kelvin(temp: Double) extends Temperature(temp - 273.15) {
  override def +(that: Temperature): Temperature = add(that)

  private def add(that: Temperature): Kelvin = {
    Kelvin(T2K(that).temp + temp)
  }
}
```

Listing 2.3: Implementation of the simple Temperature Deep embedded DSL with Scala, the dedicated language is represented directly by the host language primitives and the data are directly manipulated.

2.3.2 External Domain Specific Language

Unlike internal DSL, an external DSL is not built on top of a host language. This kind of DSL is completely independent from any general programming language or another DSL.

Developing an external DSL needs additional work :

- A parser in order to recover the information from the source code.
- A code generator.
- Anything we want from other programming languages.

An external DSL is very close to a programming language, except that a DSL is clearly simpler than a general programming language like C or Java.

2.3.3 External vs Internal

External and internal DSL are very different from each other. Understanding differences, strength and weaknesses is crucial for further DSL development.

```
t1: Celsius = Celsius(100.0)
t2: Celsius = Celsius(100.0)
t3: Kelvin = Kelvin(100.0)
t4: Temperature = Celsius(26.850000000000023)

Celsius(26.850000000000023)
```

Figure 2.4: Result of the small Temperature program from listing 2.3. The result represents the way to think about a Deep embedded DSL. We capture the operations and we manipulate the information.

External DSL

External DSL comes with the advantage that the DSL designers may define any possible syntax without any limitation[SZ09]. It could be textual, graphical or even audiovisual (vocal command). Thus, because an external DSL is not bound to a specific host language or platform, the DSL could be used in any way and might be exported to additional target language or platform through transformation [SZ09]. Finally, it's almost impossible to accidentally use a feature that is not part of the DSL [SZ09]. For example, developing an internal DSL is kind of developing a library. We have access to all the language's features and any other library. Problems may occur when we try to use the DSl with a feature not treated by the DSL and that might create unexpected behaviour.

On the other hand, an external DSL does not own any tool or support. There is no compiler, no parser, no error checking, and so on. Everything as to be done with the language creation.

Internal DSL

An internal DSL always comes with a host language, and implies that the DSL designer and user have access to the features of the host language [SZ09]. Moreover, all the tools related to the host language are accessible. That includes : compiler, parser, debugger, interpreters, code analysis tools and so on [SZ09].

Another advantage of the internal DSL is the support they have from programming language. Some programming languages have a very good support by default for the development and usage of DSL. For example, Scala and Ruby give some syntactic sugar which gives a pleasant way of development to the user and designer.

In contrast to an external DSL, the user of the internal DSL may use some features by accident and, at the runtime, the occurring errors could be difficult to understand and not managed by the DSL designer.

2.4 Implementation of a Domain Specific Language with Scala

The Scala programming language has some great advantages over the other standard programming languages like Java or C. This section will cover the implementation of external and internal DSL with Scala as well as some examples of implementation. Finally, we will introduce the Lightweight Modular Staging (LMS) approach for internal and external DSL.

In order to illustrate the implementation of an external and internal DSL with Scala, we will create a simple language which represents a calculator. The DSL will be able to parse a simple computation on natural numbers like $1 + 2 * 4$ and will compute the result and display it.

2.4.1 External Domain Specific Language

External DSL could be written in any programming language, from C to Java via Haskell. The reason we use Scala is to be consistent with the internal DSL development. One of the goals of this thesis is to explore the LMS approach (available in section 2.4.3) for DSL development and this is a Scala oriented approach.

Implementing an external DSL is quite the same as implementing a general programming language. The difference is that it's simpler because the DSL is smaller than a programming language. In order to develop an external DSL we need the following components :

- Parser.
- AST representation.
- Code generator or Interpreter.

The parsing part could be achieved with different libraries or framework. We can also create or own tokenizer and parser from scratch. The advantage of that technic is that we can optimise the parser and specialise it for the exact syntax of our DSL.

A big part of the parsing library in Scala is built using the parser combinator method[OSV16]s. Basically, a parser combinator is a higher-order function (HOF) that accepts some parsers in input and returns a parser. The parser is a function that accepts a string in input and returns a structure in output. The combination of several of those HOF creates a parser which is more and more complex.

Parser combinator in Scala is very similar to the Extended Backus-Naur Form (EBNF) syntax. Listing 2.4 shows the parser for our little calculator DSL.

The structures returned by a parser can be represented by a set of Scala classes and traits. Case classes and sealed traits are perfect for such a representation. The listing 2.5 shows the implementation of our simple calculator AST. For example, the computation $2 + 1 - 2$ is represented by `Sub(Add(2,1),2)`.

```
class SimpleCalculatorParser extends RegexParsers with PackratParsers {
  override protected val whiteSpace: Regex = "[ \t\r\f\n]+".r

  override def skipWhitespace: Boolean = true

  lazy val simpleCalculatorProgram: PackratParser[Expr] = expr

  lazy val expr: PackratParser[Expr] = term

  lazy val term: PackratParser[Expr] = {
    term ~ "+" ~ product ^^ { case (l ~ _ ~ r) => Add(l, r) } |
      term ~ "-" ~ product ^^ { case (l ~ _ ~ r) => Sub(l, r) } |
      product
  }

  lazy val product: PackratParser[Expr] = {
    product ~ "*" ~ operand ^^ { case (l ~ _ ~ r) => Mul(l, r) } |
      product ~ "/" ~ operand ^^ { case (l ~ _ ~ r) => Div(l, r) } |
      operand
  }

  lazy val operand: PackratParser[Expr] = number | "(" ~> expr <~ ")"
  lazy val number: PackratParser[Expr] = "[0-9]+".r ^^ { str => Number(str.toInt) }
}
```

Listing 2.4: Implementation of the simple calculator parser. The different parsers are combined, and they produce the AST for each expression separately before generating the whole structure.

```
sealed trait Expr
case class Number(value: Int) extends Expr
case class Add(left: Expr, right: Expr) extends Expr
case class Mul(left: Expr, right: Expr) extends Expr
case class Sub(left: Expr, right: Expr) extends Expr
case class Div(left: Expr, right: Expr) extends Expr
```

Listing 2.5: AST representation of the simple calculator DSL. Scala case classes and sealed traits are very useful for this kind of representation.

Now we need to compute the result of the expression represented by the AST. We will just recursively walk into our AST, compute the leaf and finally obtain the result of our expression. The listing 2.6 shows the interpreter for the simple calculator DSL. In a case of code generation, instead of returning the evaluation of the AST, we return to the set of operations that are necessary to obtain the same result on a different platform or programming language.

2.4.2 Internal Domain Specific Language

Scala is a great language for developing internal DSL. The language itself offers some great features and syntactic sugar to create a user-friendly syntax. Those advantages are the following [Kri13]:

- Curried function.
- Implicit function, class and parameters.
- Omitting parenthesis.

```
sealed trait Expr {
  def eval: Int = this match {
    case Number(value) => value
    case Add(left, right) => left.eval + right.eval
    case Mul(left, right) => left.eval * right.eval
    case Sub(left, right) => left.eval - right.eval
    case Div(left, right) => left.eval / right.eval
  }
}
```

Listing 2.6: Interpreter of the simple calculator DSL, each expression is evaluated recursively. In a case of code generation, we don't return a result of the expression, but the successive operations to get this result on another platform.

- No end statement symbol (";").
- Infix operator syntax for methods.
- Operator overloading.
- HOF.
- By-name parameters evaluation.
- Traits.
- Definition of new type.

The listing 2.7 shows the implementation of our simple calculator using an internal representation. The usage of the internal DSL is different from the external one, but the semantic is the same.

The listing 2.7 is also showing some features of Scala in action. The implicit definition `int2ScInt` is used to convert standard Scala Int into our special kind of Int named ScInt. We create new methods named "+", "-", and so on and we use them with the infix notation : `2 + 3` is the same as `2.+(3)`. By the way, we are also using methods from the standard Scala library, in this case `println`, with our DSL.

Finally, we could define new types with Scala. Listing 2.7 is not showing a great example, but we can define type for everything. For example, a set of integer may be seen as a function $Int \rightarrow Boolean$ so, in Scala, we could define the set type with :

```
type Set[A] = A => Boolean
```

Declaring new types is useful for the DSL designer as well as for the user because the written code becomes clearer.

2.4.3 The LMS approach

LMS is a Scala library, based on Scala-virtualized [Rom+12], which provides a runtime code generation approach [RO10]. The library as been created for providing a way to create high-level DSL and use staging to optimize the code generated by the compiler.

The basic idea of LMS is to represent two types of computation :

```scala
object Main extends SimpleCalculator {
  val a : int = 2
  val b : int = 3
  println(a + b - 3)
}

trait SimpleCalculator extends App {
  type int = ScInt

  sealed trait ScType {
    def +(that: ScType): ScType = (this, that) match {
      case (ScInt(a), ScInt(b)) => ScInt(a + b)
    }

    def -(that: ScType): ScType = (this, that) match {
      case (ScInt(a), ScInt(b)) => ScInt(a + b)
    }

    def *(that: ScType): ScType = (this, that) match {
      case (ScInt(a), ScInt(b)) => ScInt(a + b)
    }

    def /(that: ScType): ScType = (this, that) match {
      case (ScInt(a), ScInt(b)) => ScInt(a + b)
    }

    override def toString: String = this match {
      case ScInt(value) => s"$value"
    }
  }
  case class ScInt(value: Int) extends ScType

  implicit def Int2ScInt(int: Int) : ScInt = ScInt(int)
}
```

Listing 2.7: Implementation of the simple calculator DSL. The DSL is directly used in the code and the implementation is showing some advantages of using Scala.

- the computations done at the compile time, noted A
- the computations done at the runtime, noted $Rep[A]$

where A is the type of the computation.

In order to illustrate this concept, look at the listing 2.5. The idea is quite the same, we want to call three times the function `doSomething`. Now, we are going to generate the corresponding scala code for each of those loops.

Loop A:

The code `0 until 3 : Range` is fully known at the compile time. So the compiler doesn't have to produce a loop and produces the following :

```scala
doSomething(0)
doSomething(1)
doSomething(2)
```

Loop $Rep[A]$:

```
def doSomething(i : Int) : Rep[Unit] = {
  if(i > 5)
    println(i)
  else
    println(i + 1)
}

// A
for(i <- 0 until 3 : Range) {
  doSomething(i)
}

// Rep[A]
for(i <- 0 until 3 : Rep[Range]) {
  doSomething(i)
}
```

Figure 2.5: A represents the value computed at the compile time and $Rep[A]$ represents the value computed at the runtime.

The code `0 until 3 : Rep[Range]` is not completely known at the compile time. So the compiler will produce a loop in accordance with the semantic of the operation.

```
var x = 0
while(x < 3) {
  val i = x
  doSomething(i)
  x += 1
}
```

LMS code generation is easy to implement, but does not offer control on the generation. The difference between A and $Rep[A]$ is clearly identifiable at this stage. A is interpreted immediately by Scala as native code and the output doesn't have any trail of A. On the other hand, the compiler doesn't know what $Rep[A]$ means, and uses the virtualisation to transform $Rep[A]$ into an AST of case classes and sealed traits. Finally, the case classes are transformed into source code by the *emitNode* function.

For an example, we take the code from listing 2.5. The contents of the *doSomething* function could be virtualised [Rom+12] into

```
__IfThenElse(__Greater(i,5), __Println(i), __Println(Add(i,1)))
```

then, the *emitNode* function use the pattern matching to generate C source code :

```
override def emitNode(sym: Sym[Any], rhs: Def[Any]) = rhs match {
  // ...
  case __IfThenElse(c,a,b) => emitValDef(sym, s"if(${quote(c)}) {${quote(a)}} else
    ↪  {${quote(b)}}")
  case __Greater(a,b) => emitValDef(sym, s"${quote(a)} > ${quote(b)}")
  case __Println(a) => emitValDef(sym, s"""printf("%d",${quote(a)})""")
  case Add(a,b) => emitValDef(sym, s"${quote(a)} + ${quote(b)}")
  case _ => super.emitNode(sym, rhs)
}
```

2.5 Summary

In this chapter we saw what is a DSL, its advantages, disadvantages and different kinds of DSL. External DSL could be seen as a programming language with lesser features and domain-specific oriented. Internal DSL own two branches : Deep embedded DSL and Shallow embedded DSL, each of those is designed and built differently but also in a host language. Finally, we studied and illustrated the implementation of any kind of DSL using the Scala programming and explained why Scala is a great language to build DSL by showing various libraries to create DSL.

CHAPTER 3

Internet of Things Development

3.1 Introduction

Internet of Things has become a huge area of research and development over the past few years. With the emergence of data science and libraries for big data analysis like Hadoop and Spark, the scientist wants more and more data to analyse.

Recovering environment data with embedded hardware is present from the start of the sixties [Com17]. Unlike to standard computer systems, embedded systems and IoT systems suffer of several challenges.

This chapter introduces the challenge of the IoT development, from the device itself to the software implementation. Then, we will set out the development of IoT projects using general programming languages, framework for embedded development and DSL for IoT systems.

3.2 Internet of Things challenges

3.2.1 Heterogeneity

As the first challenge for the system development in IoT, we should mention the disparity of the devices. The heterogeneity of the objects does not allow them to interact [Gon+14]. In order to connect them, we need a common interface like a central server or a common description language.

There are various devices and development frameworks in the world of embedded hardware: Arduino, Raspberry PI, Mbed, SPL, Simba, Teensy, and so on. Each of them with specific properties and specific programming language.

3.2.2 Optimisation

One big limitation for IoT system development is the power management. It directly affects the algorithms we can use and means that the solutions we create would always be the power-optimised one[SN16]. In general, the power management can be done by the operating, which is a luxury to have in most parts of the available devices.

We have to mention in this context a technique named Dynamic Power Management (DPM). The idea is to shut down the devices when they aren't busy and turn them on when they need to receive/send data. In fact, it is a kind of a replacement for the Operating System (OS). But using such a technique could cause latency and additional development.

3.2.3 Connectivity

In order to communicate together, and as previously announced in the section 3.2.1, IoT devices need to send information through a standard interface. Developing such interface requires experience and knowledge in software development and programming language as well as a wide knowledge in the domain of sensor devices and objects to interconnect [Gon+14].

3.2.4 More Challenges

Many more challenges exist in the area of IoT development, they will not be dealt with in this thesis because they are away from the main process of simplification we want to bring with APDL. The following list mentions other challenges that may be interesting to be dealt with in the future.

- Engineering unit: Fahrenheit/Celsius, foot/metre, and so on.
- Various network protocols: the device is connected with various protocols like ZigBee, MQTT, and so one.
- Reliability, the device could be instantly checked and have some error handling application level.
- Data description, we need to annotate the produced data (raw data are a problem).
- Various security problems:
  - Physical security.
  - Data exchange security.
  - Cloud storage security.
  - Updates and patches.
- Data curation and data brokering: a device may produce a huge amount of data and we need to deal with.
- Data description: metadata to describe raw data.

3.3 Programming languages for the Internet of Things

Using a programming language to create IoT systems offer full control to the developer in terms of device design, conception, management and development. On the other hand, the developer has to do almost everything by himself. As mentioned in 3.2.2, the great majority of the embedded devices does not support an OS.

3.3.1 General Programming Languages

Use of a general programming language for IoT development offers a lot to the user; he can design and implement almost anything he wants. Some languages have incorporated a library and/or functions in order to access the hardware directly so anyone could develop an IoT device.

C and C++ are both a good choice for such a project. Compilation targets are available for almost any platform and offer great speed and memory management. The very big advantage is as they are compiled languages, they don't need a runtime management and compiled binaries are very small for a device with a limited memory. From another point of view, the programmer needs to do almost everything by himself: memory management, pointer arithmetic error handling and, if we use C, there is no standard library for data structure and algorithm.

Python is a language to mention as well, it has been created in 1990 by Guido van Rossum [Ros95]. Instead of C and C++, Python needs a virtual machine in order to run and that is not acceptable for a small device. If the device we use owns an OS, Python is one of the best choices for the development. The language owns a great community with many tutorials and libraries for the IoT programming and especially with the Raspberry PI. We could mention too that a tool named Cython[Beh+11] is able to compile, after some work, a Python code to a C executable, which removes the interpreter and virtual machine needed at the origin. Another idea is to run Python without any OS[Edg15].

Java is also a great choice if the development of the system involves various devices with different platforms and when an OS is available. When we use C or C++, we need to clearly identify the hardware. When we have multiple devices with various architectures, we have to code platform-specific code for each of them and that involves poor scaleability and reutilisability. Java offers to run on any platform if an OS and a Java Virtual Machine (JVM) is available.

There are a lot more programming languages for IoT development: Javascript, Go, Rust, Parasail, B#, Assembly, and so on. We mentioned C, C++, Java and Python in order to expose the advantages and disadvantages to use a general programming language for IoT systems.

3.3.2 Framework for embedded development

Over the past years, multiple frameworks for embedded systems appeared and grew up. That kind of framework simplifies a lot the development process for embedded and IoT systems. We will explore in particular two of them: Arduino and Mbed.

Arduino is an open-source platform based on easy-to-use hardware and software[Ard17a]. Such an electronic card is capable of reading input like sensors or buttons, as well as generating output through serial or LED. In order to use the hardware, a programming language named Arduino as well and based on Wiring[Ard17a] is provided. This language is a kind of a subset of C but with the addition of classes.

Mbed is an open-source operating system based on 32-bit ARM Cortex-M micro-controllers[TW12]. Mbed has been specially designed for IoT device design and development. The language used by the developer is C++ with the addition of the Mbed SDK[TW12].

3.4 Domain Specific Languages for the Internet of Things

Research and work has been done during the past years on the DSL for the IoT. As presented in section 3.2, IoT projects may involve a lot of different aspects, technologies and knowledge. From hardware specification to network protocols, to software design and implementation, we are not going to present all the DSL for the IoT. We present the ones that have been chosen for the proximity with this thesis purpose.

Node-red is a Visual Domain Specific Modeling Language (VDSML), developed with Node.js and Javascript whose goal is to wire together hardware devices, API and online services[IBM17]. A flow based editor is provided in the browser and the user is invited to create nodes, wire them, deploy and run the projects with a maximum of simplicity[Sal+15]. The runtime engine is built on top of Node.js and makes use of its event-driven and non-blocking model. This makes it ideal to run on small hardware such as Arduino or Raspberry[IBM17]. By the way, Node-red is by far the most well-known and used DSL in the area of IoT.

DiaSuite is a tool suite, designed by the Inria, that uses a software design approach to drive the development process[Ber+14]. The tool is focused on the Sense/Compute/Control (SCC) architecture. The SCC architecture is a pattern involved in a system like: building automation, application monitoring, robotics, autonomic computing[TMD09]. The tool suite includes a domain-specific design language, a compiler to Java code, a simulator and a deployment framework.

DSL-4-IoT is an Editor-Designer based on a high level Visual Programming Language (VPL), established on the class of VDSML[Sal+15]. The metamodel is established on a formal representation and an abstract syntax. The runtime execution of the generated files is running on top of the open-source project OpenHAB[EK15]. The tool relies on a declarative DSL used to graphically represent an IoT system.

ArduinoML is a modelling language use to describe the Arduino Uno micro-controller[MCB14]. The project has grown with the participation of several students from the University of Nice-Sophia Antipolis in France. ArduinoML is just a concept, its implementation is written with several different languages such as Haskell, Python, Pharo, Scala and many others[Moo17].

3.5 Summary

In this chapter we have presented some of the challenges of the IoT world. Heterogeneity, power optimisation and connectivity are huge problems. Then we presented some general programming languages available for the IoT development as well as some frameworks that simplify the design and implementation of embedded devices. Finally, we have seen various DSL who tried to resolve challenges and problems that are encountered by the IoT projects.

The various DSL we presented are solving some of the challenges of the IoT. It's not easy to solve all the challenges of the IoT world, but those DSL offer a simple way to deal with some of them.

CHAPTER 4

Design

4.1 Introduction

This chapter will present the design of the APDL DSL. Firstly, we set out the domain of interest and some kind of a generalisation for the embedded code we produce. The domain of interest regroups all the relevant information related to the DSL we create. This chapter will aslo present some general parts of the embedded systems we want to generate. As a reminder, we want to generate the code for some devices that are going to recover some data through sensors, apply transformation on it and send them through a communication network like a serial port.

The second part is about the fragmentation of APDL. We are going to present why APDL is fragmented into multiple DSL or languages and explain the advantages and disadvantages of this method. We would also introduce the PlatformIO system for embedded devices.

Finally, we present the final design chosen for APDL DSL and the explanation on the choice of making an external DSL.

4.2 Domain of interest

According to [Deu+98], the development phase of a DSL requires a thorough understanding of the underlying domain. The recommendation is to follow the next seven points[Deu+98].

- Identify problem domain of interest.
- Gather all relevant knowledge in this domain.
- Cluster this knowledge in a handful of semantic notions and operations on them.
- Construct a library that implements the semantic notions.
- Design a DSL that concisely describes applications in the domain.
- Design and implement a compiler that translates DSL programs to a sequence of library calls.
- Write DSL programs for all desired applications and compile them.

The point “Identify problem domain of interest” is treated in chapter 3. In the next section, we will gather the relevant information on our specific domain.

4.3 Generalisation of the Embedded Framework

“What do we want ?”

That’s the first questions to ask when we start analysing the domain’s problems. As an answer, we suggest the following :

“We want to simply design a system, depending on a specific device, that is able of recovering some input data, transform and send them through a network.”

Note that the previous sentence doesn't care about what's happening to the transferred data, including handling, storage and visualisation. This part is treated in chapter 7.

We are going to take two examples, one with the Arduino Framework and one with the Mbed SDK, and try to generalise the concept between each other. The listing 4.1 shows the Arduino code, and the listing 4.2 is showing the Mbed ones. The two codes have the same goal :

- Recover two analogical inputs, the temperature and the luminosity.
- Transform the temperature with a function.
- Send the data each second.

```
#include "Timer.h"

Timer timer;

int pinLum = 1;
int pinTemp = 0;

float decode_temperature(int x){
  int B = 3975;
  float resistance = ((((float)(1023 - x)) * 1000) / x);
  float temperature = ((1 / ((log((resistance / 1000)) / B) + (1 / 298.15))) - 273.15);
  return temperature;
}

void sendLum(){
  int data = analogRead(pinLum);
  byte * b = (byte *) &data;
  Serial.write(b,4);
}

void sendTemperature(){
  int rawData = analogRead(pinTemp);
  float data = decode_temperature(rawData);
  byte * b = (byte *) &data;
  Serial.write(b,4);
}

void loop() {
  timer.update();
}

void setup() {
  Serial.begin(9600);
  timer.every(1000,sendLum);
  timer.every(1000,sendTemperature);
}
```

Listing 4.1: Implementation of a simple data recovering device with the Arduino Framework. The sampling is achieved using a Timer and callbacks. The callback's methods are sending the data through a serial network.

```
#include <mbed.h>

Ticker ticker;

Serial pc(USBTX, USBRX);

AnalogIn rawTemp(PC_1);
AnalogIn lum(PC_3);

 // Transform tf
float tf (int x){
  int B = 3975;
  float resistance = ((((float)(1023 - x)) * 1000) / x);
  float temperature = ((1 / ((log((resistance / 1000)) / B) + (1 / 298.15))) - 273.15);
  return temperature;
}

void sendLum(){
  float data = lum.read();
  uint8_t * b = (uint8_t *) &data;
  unsigned int size = 0;
  while(size < sizeof(b)) {
      pc.putc(b[size++]);
  }
}

void sendTemperature(){
  float data = rawTemp.read();
  uint8_t * b = (uint8_t *) &data;
  unsigned int size = 0;
  while(size < sizeof(b)) {
    pc.putc(b[size++]);
  }
}

int main(void) {
  pc.baud(9600);
  ticker.attach(&sendLum,1.0);
  ticker.attach(&sendTemperature,1.0);
  while(1);
  return 0;
}
```

Listing 4.2: Implementation of a simple data recovering device with the Mbed Framework. The sampling is achieved using a Ticker and callbacks. The callback's methods are sending the data through a serial network.

The two codes 4.1 and 4.2 are very close to each other. We can recognise some key elements :

An input is represented by all the elements necessary to recover the data from a sensor through specific pin. For Arduino, it's the code `analogRead(pinId)` which provides the data from the sensor. For Mbed, it's the object `AnalogIn` with the method `read()`. An input could be represented as a function $f : () \rightarrow A$where A is the type of the data we get when using the input.

The sampling indicates, in the software level, when the device needs to recover the sensor data. The two main kinds of sampling are "by update" and "by frequencies". The sampling by update is presented in listing 4.3, the device sends the data only if it has changed. In the sampling by frequencies showed in listing 4.4, the device sends the data on each specified interval.

```
void setup(){
  timer.every(1000,byFrequences);
}

void byFrequences(){
  // Get data
  float data = tf(analogRead(1));
  // As a byte array…
  byte * b = (byte *) &data;
  // Send data
  Serial.write(b,4);
}
```

Listing 4.3: Sampling by update implemented with an Arduino. The device sends the value in each time interval.

```
void setup(){
  timer.every(1000,serial_lum);
}

void byUpdate(){
  // Get data
  float data = tf(tf(analogRead(1)));
  if(data != last_serial_temp2) {
    // As a byte array…
    byte * b = (byte *) &data;
    // Send data
    Serial.write(b,4);
  }
  last_serial_temp2 = data;
}
```

Listing 4.4: Sampling by update implemented with an Arduino. Before sending the value, we test if it changes or not.

A transformation is a function $f : A \to B$ with an input and an output, and is independent from the languages or framework. The only important part of a transformation is its semantic. Depending on the language, a transformation is not implemented in the same way but owns the same properties. This concept gives the opportunity to design a higher level language and then compile it into an embedded one.

A **device** is the hardware in which we are going to run the previous elements (inputs, transformations, and so on). The device depends on the hardware and the software, sometimes it comes with a framework like mentioned in 3.3.2.

The last generalisation we made is called the "Acquire and Process lifecycle". The lifecycle is represented by two steps in the life of an embedded device : the setup and the loop. The setup is the first operation made by the device, like configuration, calibration, and so on. The loop is the set of operations executed permanentely until we stop the device.

4.4 Multiple Domain Specific Languages

According to 4.2, we are going to cluster the acquired knowledge in a handful of semantic notions with operations on them and construct a library with a language that implements those notions [Deu+98].

From the user's point of view, we want to describe several devices that gather inputs, apply some transformations and send them through a network with a specific sampling. Those notions are translated into the following operations :

- Create a device.
- Link some inputs with a device.
- Create a transformation function.
- Link an input with a transformation.
- Send an input through a network with a sampling.
- Specify the kind of sampling.

The problem is that we can't specify everything with a unique DSL. The transformation's implementation is not part of the devices and input specifications. That's why we would have multiple DSL in our ecosystem. For example, if we want to add a way to specify the storage and the visualisation of the gathered data, doing it inside the declarative DSL could create some inconsistency and behaviour misunderstanding.

4.5 PlatformIO

PlatformIO[IDV17] is an open source ecosystem for IoT development. It provides a cross-platform IDE and unifies debuggers with remote unit testing and firmware update[IDV17].

Such a build system is made for the kind of project we are doing with APDL. Each board supported by the PlatformIO is represented by a unique identifier and the build systems provide an automatic downloading of libraries and dependencies through a configuration file.

PlatformIO is used as a back end for APDL, so only the PlatformIO supported boards are available with APDL.

4.6 Design of the APDL's Domain Specific Languages

The goal of APDL is to provide a simple way of declaring sensors oriented systems. The design language needs to be simple and easy to read. Some highly verbose languages are not easy to read for a human. We want to quickly understand the purpose of a declared system.

The listing 4.5 is showing all the concepts discussed in 4.4. We can easily understand the purpose of the system : gather two inputs, one on pin 1 and one on pin 0. Then we transform one and send both through a serial network. One with a sampling of 1 second and the other is sent when it changes.

```
@device arduino1 {
    id = uno
    framework = arduino
    @input rawTemp analogInput 1
    @input lum analogInput 0
    @input temp tf rawTemp
    @serial lum each 1 s
    @serial temp update
}
```

Listing 4.5: Declaration of a device using the APDL DSL. Everything is purely declarative, we never indicate how to do it.

The device declaration owns several information, each point indicates the corresponding APDL code too :

- An identifier for the device, after the `@device` keyword.
- A key-value pair, named `id`, which is the corresponding PlatformIO id of the boards[IDV17].
- A key-value pair, named `framework`, which is corresponding to the development framework of the board, also available on PlatformIO[IDV17].
- Some inputs, each one specified with an identifier, the type of the input and the arguments for the type.
- Some serial, each one specified with an identifier and the sampling method and values.

4.7 The APDL-Transform Domain Specific Language

The APDL TransForm (TF) language is another DSL whose goal is to provide a high-level way of coding a transformation. As explained in 4.3, the advantage to design a high-level language for transformation is the capability of using a simple language and then to compile it to the desired platform.

The syntax of the TF is quite similar to Scala and gives the traditional concept for the user. The listing 4.6 shows an example of a transformation. This transformation is used to convert the resistance of the sensor in Celsius.

```
@define transform def tf (x:int) -> float {
    val B : int = 3975
    val resistance : float = (float)(1023 - x) * 1000 / x
    val temperature : float = 1 / (log(resistance/1000) / B + 1 / 298.15) - 273.15
    return temperature
}
```

Listing 4.6: Implementation of a transformation function using the TF language. The syntax is quite similar to Scala and gives the opportunity to write a high level code for embedded platform.

The TF language is fully integrated with APDL, so we can modify the language and extend it as much as we want. For the moment, the language offers the following concept :

- Addition, subtraction, multiplication and division.
- C-like type casting.
- Variable and constant declarations.
- Array creating and access.
- Function definition and call.
- Boolean expression.
- Comparison operators.
- Loops.
- Flow-control structure : If-then-else, return, break and continue.
- C-like type hierarchy.

Basically, all the implementation that comes after `@define transform` is the declaration of a function with TF. An introduction to the TF language is available in appendix A.

4.8 Extending the User Possibilities

In the section 4.7 we introduce the keyword `@define`. This keyword provides to the user the possibility to define a new entity when designing a APDL project. There are three different possible kinds of definitions the user can create : transformations, inputs and components. We already saw the transformation definition in section 4.7.

4.8.1 Defining New Inputs

An input entity represents the way to recover the sensor information on a device. For example, we would take the `analogInput` seen in listing 4.5. An analogical input is present on a lot of platforms. For our example, we would take in consideration only the Arduino platform[Ard17b] and the Mbed platform[mbe].

The analogical input is present in both frameworks, but its implementation and usage for the user are quite different in both. In order to cover all kinds of implementations for an input concept, we need to provide a general way to define inputs. Such a way is shown in listing 4.7.

```
@define input analogInput pin:str {
    @gen mbed {
        global = "AnalogIn @id(@pin);"
        setup = ""
        loop = ""
        expr = "@id.read()"
        type = float
    }
    @gen arduino {
        global = ""
        setup = ""
        loop = ""
        expr = "analogRead(@pin)"
        type = int
    }
}
```

Listing 4.7: Definition of an analogical input using APDL. The generic part for any kind of frameworks is present between the "input" identifier and the opening brackets. We could define the identifier for the input and some arguments. Then we could write any code that would be generated for the specified platform at the compile time.

Listing 4.7 also introduces the **@gen** keyword. This keyword allows the user to define how to use the inputs in a specified framework by giving some information in terms of key-value pairs : global, setup, loop, expr and type. When we generalise an embedded device implementation, we are bound to those concepts as shown in listings 4.8 and 4.9.

The "setup" and "loop" values are corresponding to their eponym discussed in the section 4.3, the "global" value represents what all the users want to put in the global scope, like global variables, function definitions or constant. Finally, the "expr" and "type" values indicate to the compile how to recover the input value and its type.

```
/* Global */

void setup(){
  /* Setup */
}

void loop(){
  /* loop */

  /* type */ data = /* expr */ ;
}
```

Listing 4.8: Generalisation of an embedded device lifecycle with the Arduino framework. The framework already provides the "loop" and "setup" function. This example also shows the result of the input definition for Arduino.

```
#include <mbed.h>

/* Global */

int main(void){
  /* Setup */

  while(1){
    /* Loop */

    /* type */ data = /* expr */ ;
  }
}
```

Listing 4.9: Generalisation of an embedded device lifecycle with the Mbed framework. The framework does not provide the "loop" and "setup" abstraction like Arduino does, so we need to simulate them. This example also shows the result of the input definition for Arduino.

The strings present in the values for the **@gen** definition are a kind of interpolated strings and have access to the parameters through the macro **@parameterIdentifier**. APDL also provides the macro **@id** which is a unique identifier generated by the compiler.

Now, admitting we generate the "analogInput" definition for Mbed, we would obtain the result shown in listing 4.10. We simply generate the defined input with the information about the generation given by the user.

```
#include <mbed.h>

AnalogIn x_1(A1);

int main(void){

  while(1){
    float data = x_1.read() ;
  }
}
```

Listing 4.10: Generation of an input definition for the Mbed framework. All we do is just replacing the arguments for the definition and then generate the code at the specified points showed in listing 4.8. We admit that the "pin" parameter value is equal to 1. The **@id** value is generated by the compiler.

By using this kind of definition and generation, we can write any kind of inputs on any platforms. By offering a general description for the generation, the user could also define some inputs that require a library. In this case, the user can include the library by using the global value.

4.8.2 Defining New Components

We could go further than the input's definition mechanism by offering to the user the possibility to define components. A component could be seen as a black box, with some inputs and an output. An example of a component design with APDL is shown in listing 4.11.

```
@define component simpleOperator op:str {
    @in x:int y:int
    @out int
    @gen mbed {
        global = ""
        setup = ""
        loop = ""
        expr = "@x @op @y"
    }
    @gen arduino {
        global = ""
        setup = ""
        loop = ""
        expr = "@x @op @y"
    }
}
```

Listing 4.11: Definition of a component with APDL. A component's definition is quite similar to an input definition except that the output type is not depending on the framework. A component definition is a kind of macro system for APDL. Another difference is the generation. An input is used only for recovering the input and a component is generated as a function.

At the generation time, a component is described as a function $f : (A_0, A_1, ...A_n) \rightarrow B$ where $A_0, ..., A_n$,are the parameter's types of the functions and B is the return type.

Using a component is almost the same as using an input. We just have to specify the type of the component and give the exact number of arguments. The arguments are, in order, the device parameters and then the input parameters :

```
@input componentOutput simpleOperator + input1 input2
```

The generated code is shown in listing 4.12.

When we use this implementation and the definition of the component defined in listing 4.11, we obtain the code from listing 4.12. All the parameters are interpolated at the compile time.

```
int component_simpleOperator_tempLum(int x,int y) {
  return x + y;
}
```

Listing 4.12: Generated code from the component defined in listing 4.11. A component is represented as a function, and it is parameterised with its arguments.

4.9 Summary

In this chapter, we have presented the design process and choices on the multiple APDL's DSLs. We set out the APDL's domain of interest by introducing the development process of a DSL by [Deu+98]. Then we suggested a generalisation in section 4.3 for any sensor-oriented projects, independently from any platforms, by introducing the concept of : input, devices, transformation, sampling and the APDL lifecycle with the steps called "setup" and "loop".

After, we discovered in section 4.4 why we need multiple DSL for APDL and argued that implementing everything with a single DSL could create some inconsistency and behaviour misunderstanding.

Finally, we set up the design of all the DSL and concepts created with APDL : the APDL DSL, the TF language for transformation and the possibility of defining new inputs in section 4.8.1 and new components in section 4.8.2.

CHAPTER 5

Implementation

5.1 Introduction

In chapter 2.4 we have introduced the implementation of DSL using the Scala programming language and why Scala is a great choice for DSL development.

This chapter deal with the implementation of the APDL DSL compiler by showing and explaining the different part of the compilation process. We will se the lexical and semantic analysis, the construction of the AST and finally the code generation.

Some specific part of the implementation are more complex so additional information are given.

5.2 Compilation Process

The whole APDL compilation process is illustrated in figure 5.1. The compilation process is separated into fours phases. Each phase receives an input and returns an output to the next phases, except for the first and last ones. The four phases are :

- The lexical analyser receives the APDL source code and returns a first AST.
- The semantic analyser receive the AST form the lexical analyser and return a symbol table and a second AST, refined and more complete.
- The code generator receive the symbol table and an AST and generate the corresponding code in a file.
- Code formatting, reformat the code inside the generated files.

The code formatting phase is not very important, it just provides a way to the user to inspect the generated code and possibly find issues or errors.

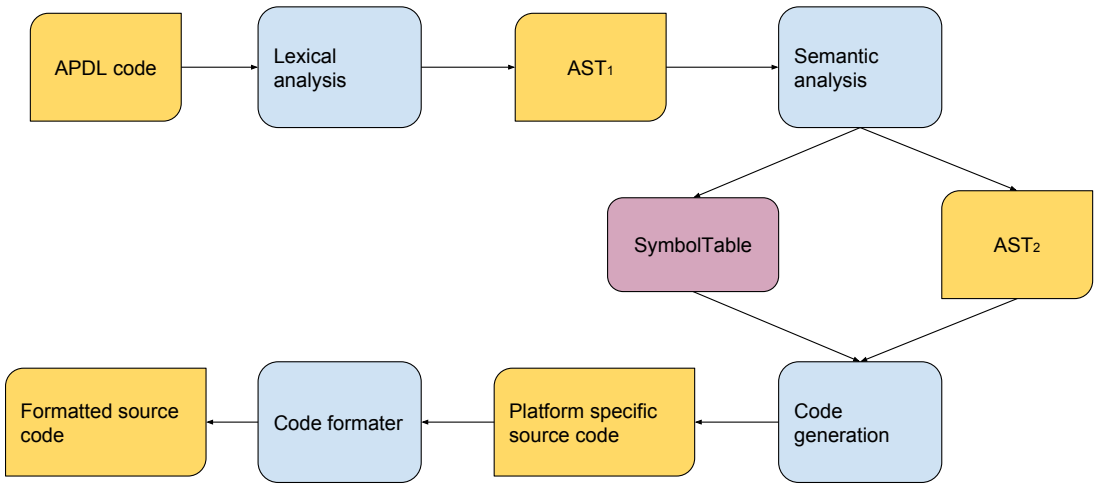


Figure 5.1: The APDL compilation process, the source code is parsed into a first AST before transforming into a second one and a symbol table by the semantic analyser. Then, the code generator uses the AST and the symbol table to generate a raw platform specific source code. Finally, a code formatter like Astyle[Dav], ends the process in order to conveniently read the generated code.

5.3 Lexical Analysis

The lexical analysis implementation has been achieved using the Scala parser-combinator library[OSV16]. As explained in 4.4, we have fragmented APDL into multiple DSL. That means we have to parse the TF and the main languages.

So we have two parsers for our DSL, a first one called "TransformDSLParser" and the second one called "MainParser". In fact, the "MainParser" has been separated into two more parsers, one for the user definition and the second for the rest. But they could be seen as one.

The syntax of the TF language is shown in listing 5.1. The syntax is close to some popular programming languages like Java, C or Scala. We could note the absence of an "end-of-statement" symbol by opposition to C-like languages.

The syntax for defining new components, inputs or transformations is available in listing 5.2, the one for the main DSL in listing 5.3.

The implementation of the parsers has been achieved by using a specific parser from the Scala parser-combinator library. Precisely, we have used a parser technique called the packrat parser[For06]. Parsing grammar with packrat parser is guaranteed to operate in a linear time through the use of memoization[For06]. Packrat parser also allows the usage of left recursion, which is significant with the type of grammar we used for defining the syntax of our APDL.

As an example for the packrat algorithm, we would take the grammar of the "additiveExpr" :

```
additiveExpr ::= additiveExpr "+" multiplicativeExpr
               | additiveExpr "-" multiplicativeExpr
               | multiplicativeExpr
```

```
/* Types */
returnType ::= "void" | tfType
tfType ::= arrayType | primitiveType
arrayType ::= "[" "]" tfType
primitiveType ::= "bool" | numericType
numericType ::= integralType | floatingPointType
integralType ::= "int" | "long" | "char" | "short" | "byte"
floatingPointType ::= "float" | "double"

/* Expressions */
constantExpr ::= logicalOrExpr
logicalOrExpr ::= logicalOrExpr "||" logicalAndExpr | logicalAndExpr
logicalAndExpr ::= logicalAndExpr "&&" logicalEqExpr | logicalEqExpr
logicalEqExpr ::= logicalEqExpr "==" logicalRelationalExpr | logicalEqExpr "!="
  ↪  logicalRelationalExpr | logicalRelationalExpr
logicalRelationalExpr ::= logicalRelationalExpr ">" additiveExpr | logicalRelationalExpr "<"
  ↪  additiveExpr | logicalRelationalExpr ">=" additiveExpr | logicalRelationalExpr "<="
  ↪  additiveExpr | additiveExpr
additiveExpr ::= additiveExpr "+" multiplicativeExpr | additiveExpr "-" multiplicativeExpr |
  ↪  multiplicativeExpr
multiplicativeExpr ::= multiplicativeExpr "*" castExpr | multiplicativeExpr "/" castExpr |
  ↪  castExpr
castExpr ::= "(" primitiveType ")" castExpr | notExpr
notExpr ::= "!" notExpr | postfixExpr
postfixExpr ::= functionCall | arrayAccess | primaryExpr
functionCall ::= identifier "(" functionArgs ")"
arrayAccess ::= identifier ("[" constantExpr "]")+
primaryExpr ::= atom | symbol | literal | "(" constantExpr ")"
functionArgs ::= (constantExpr ("," constantExpr)*)?
atom ::= "true" | "false"
symbol ::= identifier
literal ::= [+-]?([0-9]+([.][0-9]*)?|[.][0-9]+)(E[0-9]+)?
identifier ::= [a-zA-Z_][a-zA-Z0-9_]*
assignExpr ::= postfixExpr "=" assignExpr | logicalOrExpr

/* Statement */
statement ::= block | selectionStatement | loops | jump | declaration | exprStatement
block ::= "{" (statement)* ")"
selectionStatement ::= ifThenElse | ifThen
loops ::= while | doWhile
jump ::= "break" | "continue" | "return" constantExpr
declaration ::= newVar | newArray | newVal | functionDeclaration
exprStatement ::= constantExpr
ifThenElse ::= "if" "(" constantExpr ")" statement "else" statement
ifThen ::= "if" "(" constantExpr ")" statement
while ::= "while" "(" constantExpr ")" statement
doWhile ::= "do" statement "while" "(" constantExpr ")"
newVar ::= "var" identifier ":" tfType "=" constantExpr | "var" identifier ":" tfType
newArray ::= "var" identifier ":" arrayType "=" arrayInit | "var" identifier ":" arrayType
newVal ::= "val" identifier ":" tfType "=" constantExpr
functionDeclaration ::= "def" functionHeader functionBody
arrayInit ::= "[" literal "]" | "{" constantExpr ("," constantExpr)* "}"
functionHeader ::= identifier "(" (arg ("," arg)*)? ")" "->" returnType
functionBody ::= block
arg ::= identifier ":" tfType
```

Listing 5.1: EBNF syntax of the Transform APDL Language. The syntax is similar to C and Java, except that there is no need to specify an "end-of-statement".

```
defines ::= (define)*
define ::= "@define" (defineComponent | defineInput | defineTransform)
defineComponent ::= "component" identifier parameters "{" defineComponentBody "}"
defineInput ::= "input" identifier parameters "{" gens "}"
defineTransform ::= "transform" functionDeclaration
parameters ::= (parameter)*
parameter ::= identifier ":" type
defineComponentBody ::= inputs output gens
gens ::= (gen)*
gen ::= "@gen" identifier "{" genBody "}"
inputs ::= "@in" parameters
output ::= "@out" type
type ::= "str" | "id" | stdType
stdType ::= "int" | "long" | "char" | "short" | "byte" | "float" | "double"
genBody ::= global setup loop expr genType
global ::= "global" "=" '"' literalString '"'
setup ::= "setup" "=" '"' literalString '"'
loop ::= "loop" "=" '"' literalString '"'
expr ::= "expr" "=" '"' literalString '"'
genType ::= "type" "=" stdType
literalString ::= (.|[^\"])*
```

Listing 5.2: EBNF syntax for defining new components, inputs and transformations. The EBNF nodes that aren't in this syntax are in the syntax from listing 5.1.

```
program ::= (projectName | apdlDevice | define)+
projectName ::= "project_name" "=" '"' literalString '"'
apdlDevice ::= "@device" identifier "{" (keyValue | apdlInput | apdlSerial)+ "}"
apdlInput ::= "@input" identifier identifier apdlParameters
apdlSerial ::= "@serial" identfier apdlSampling
apdlParameters ::= (apdlParameter)*
apdlParameter ::= "[^ \t\f\n\r{}@]+"
apdlSampling ::= apdlSamplingUpdate | apdlSamplingTimer
apdlSamplingUpdate ::= "update"
apdlSamplingTimer ::= "each" '[0-9]+' timeUnit
timeUnit ::= "ns" | "ms" | "s" | "m" | "h" | "d"
```

Listing 5.3: The full EBNF syntax for the APDL DSL. Any language accepted by this grammar is a valid, syntactically speaking, APDL program. The EBNF nodes that aren't in this syntax are in the syntax from listing 5.2.

If we use a standard algorithm in order to implement this grammar, we would have something like

```
def tfAdditiveExpr: Parser[Expr] = {
  tfAdditiveExpr ~ ("+" ~> tfMultiplicativeExpr) ^^ {
    case (l ~ r) => Add(l, r)
  } |
  tfAdditiveExpr ~ ("-" ~> tfMultiplicativeExpr) ^^ {
    case (l ~ r) => Sub(l, r)
  } |
  tfMultiplicativeExpr
}
```

The problem with this implementation is the infinite recursive call which happens, because the first rule of the token is the token itself. Using packrat parser solves this problem by enabling the left recursion as well as using memoization through lazy evaluation :

```
lazy val tfAdditiveExpr: PackratParser[Expr] = {
  tfAdditiveExpr ~ ("+" ~> tfMultiplicativeExpr) ^^ {
    case (l ~ r) => Add(l, r)
  } |
  tfAdditiveExpr ~ ("-" ~> tfMultiplicativeExpr) ^^ {
    case (l ~ r) => Sub(l, r)
  } |
  tfMultiplicativeExpr
}
```

This time, it's a lazy value so we don't have an immediate infinite recursion. So the packrat algorithm can operate and enables the left recursion.

During the lexical analysis, the parsers are producing the AST of the program. In Scala, the AST is designed and built by using case classes and sealed traits. We are not going to present the whole implementation of the AST, all the code is available in appendix B. We will just present the implementation for the expression of the TF Language, the code is shown in listing 5.4.

The full set of syntactical diagrams is available in appendix C. We also need to mention that the parser is not performing any verification of the input's semantics, that's the semantics analyser role.

```scala
sealed trait ApdlAst

sealed trait Expr extends ApdlAst
// Arithmetic expression
case class Add(left: Expr, right: Expr) extends Expr
case class Mul(left: Expr, right: Expr) extends Expr
case class Sub(left: Expr, right: Expr) extends Expr
case class Div(left: Expr, right: Expr) extends Expr
case class Cast(tfTyp: TfPrimitivesTyp, expr: Expr) extends Expr
case class Literal(value: String) extends Expr
case class Symbol(name: String) extends Expr
case class FunctionCall(funcName: String, args: List[Expr]) extends Expr
case class ArrayAccess(array: Expr, field: Expr) extends Expr
case class VarAssignement(target: Expr, value: Expr) extends Expr

// Boolean expression
case class True() extends Expr
case class False() extends Expr
case class Or(left: Expr, right: Expr) extends Expr
case class And(left: Expr, right: Expr) extends Expr
case class Not(expr: Expr) extends Expr
case class Greater(left: Expr, right: Expr) extends Expr
case class Smaller(left: Expr, right: Expr) extends Expr
case class GreaterEquals(left: Expr, right: Expr) extends Expr
case class SmallerEquals(left: Expr, right: Expr) extends Expr
case class Equals(left: Expr, right: Expr) extends Expr
case class NotEquals(left: Expr, right: Expr) extends Expr
```

Listing 5.4: Implementation of the APDL abstract syntax tree using case classes and sealed traits. The usage of inheritance is very powerful.

5.4 Semantic Analysis

The semantic analyser role is to construct the symbol table of the APDL compiler. The symbol table is just a map where the key is the identifier of a definition or a variable inside a device, and the value is the AST of the corresponding definition or user implementation.

The implementation of the symbol table is available in listing 5.5. The semantic analyser builds the table, especially the input, by recursively walking through the AST and generating the corresponding values for the symbol table.

The AST generated by the parser is very simple and is just a translation of the program into traits and case classes. If we take the input

```
@device arduino1 {
    id = uno
    framework = arduino
    @input rawTemp analogInput 1
    @input temp tf rawTemp
    @input temp2 tf temp
    @serial temp2 update
}
```

we saw that the *temp*2 value is built with the *temp* value, which is built with the *rawTemp* value. So we could define a graph G of inputs : $G = rawRemp \rightarrow temp \rightarrow temp2$, where $\rightarrow$ means “is input of”.

```
sealed trait SymbolTableElement

case class Component(identifier: String,
                     outputType: ApdlType,
                     parameters: List[Parameter]) extends SymbolTableElement
case class Transform(functionDecl: FunctionDecl) extends SymbolTableElement

sealed trait Input extends SymbolTableElement {
  def getDefinition: ApdlDefine = this match {
    case InputDefault(_, definition, _, _) => definition
    case InputTransformed(_, definition, _) => definition
    case InputComponented(_, definition, _) => definition
  }
}

case class InputDefault(identifier: String,
                        definition: ApdlDefineInput,
                        args: List[String],
                        expr: String) extends Input
case class InputTransformed(identifier: String,
                            definition: ApdlDefineTransform,
                            input: Input) extends Input
case class InputComponented(identifier: String,
                            definition: ApdlDefineComponent,
                            inputs: List[Input]) extends Input
```

Listing 5.5: Implementation of the symbol table accepted values using sealed trait and case classes. Basically, it's just a map with a few methods and a typed value for correctness.

The previous graph is linear and could be just seen as a list. When component comes into play, we could have multiple inputs for a same node of the graph and we obtain a tree. Note that defining components with multiple outputs is not yet permitted by APDL but is planned for the future.

In order to construct the whole recursive inclusion of inputs, we need, at first, to generate the inputs located in the leaves of the tree. Then, we generate the nodes that are already known inputs and we repeat that until we got the full inputs set done. The algorithm is described in Algorithm1.

A Scala implementation of the algorithm 1 is available in appendix D. The implementation is recursive and also processes some verification on the types of the data and on the presence of the identifier in the symbol table.

We also need to note that there is mostly no type system, type verification or semantic verification of the code at the compile time. It requires a lot of work to build a strong verification system, and the time allowed for this work is too short. The user needs to be extremely worried about his code, and the errors that he encounters could have been found before the runtime. Implementing a good type system and more verification is planned for the future.

Algorithm 1 Recursive inclusion construction algorithms. The Scala implementation is available in appendix D.

List of non-leaf inputs : is
Symbol table : s
repeat
  $(generableInput, nonGenerableInputs) \leftarrow is.partition(isGenerable)$
  for $i \in is$ **do**
    if $i : Transform$ **then**
      $sourceInput \leftarrow s.get$(first argument of i)
      $key \leftarrow$ Identifier of i
      $value \leftarrow Transform$(AST of the function declaration, $sourceInput$)
      $s.put(key, value)$
    end if
    if $i : Component$ **then**
      $args \leftarrow$ Input identifier of i
      $sourceInputs \leftarrow s.get(args)$
      $key \leftarrow$ Identifier of i
      $value \leftarrow ComponentedInput(key,$ Definition AST of $i, sourceInputs)$
      $s.put(key, value)$
    end if
  end for
until is is empty

5.5 Code Generation

There are multiple parts in the code generation process :

- Generating the project folders and the script for launching the application.
- Generating the PlatformIO project for a device.
- Generating the code for a specific device, including inputs, serial sends, transformation functions and components.

The project and device generation is quite simple and does not require a lot of explanations. Basically, it's just a set of new folders and PlatformIO ".ini" files for each device. A bash script is also produced in order to launch the whole application.

The code generation is a lot more interesting. Firstly, we have seen in chapter 3.3.2 ,that a lot of various frameworks are available. Each framework has its vision on how to implement a device. For example, Arduino offers two methods, *loop* and *setup*, that offers the eponym functionality to the user. Mbed does not have this, and we have to simulate them.

For this purpose, we have developed an abstraction for the code generation. The code to generate is the same for both the Mbed and Arduino frameworks, but the location to generate it differs from one to the other. Both listings 5.6 and 5.7 for Arduino, respectively Mbed, show both implementations.

```
override def close(): Unit = {
    pw.append {
      s"""
          |#include <stdbool.h>
          |${if (generateTimer) s"""#include "Timer.h""""}
          |
          |${if (generateTimer) s"Timer t;"}
          |
          |// Global definition
          |${global.toString}
          |
          |// Function definition
          |${function.toString}
          |
          |void loop() {
          |  ${if (generateTimer) s"t.update();"}
          |  ${loop.toString}
          |}
          |
          |void setup() {
          |  ${if (generateSerial) s"Serial.begin(9600);"}
          |  ${setup.toString}
          |}
          |
        """.stripMargin
    }
    pw.flush()
    pw.close()
    formatSource()
  }
}
```

Listing 5.6: Implementation of the Mbed code generation location. The code is generated by using Stringwriter. The compiler accumulates the generated code in buffer until the end of the compilation. At the end, the compiler interpolates the string with the contents of his buffers.

```
override def close(): Unit = {
    pw.append {
      s"""
          |#include <stdbool.h>
          |#include <mbed.h>
          |
          |${if (generateTimer) s"Ticker ticker;"}
          |
          |${if (generateSerial) s"Serial pc(USBTX, USBRX);"}
          |
          |// Global definition
          |${global.toString}
          |
          |// Function definition
          |${function.toString}
          |
          |int main(void) {
          |  ${if (generateSerial) s"pc.baud(4800);"}
          |  // Setup
          |  ${setup.toString}
          |
          |  // Loop
          |  while(1) {
          |    ${loop.toString}
          |  }
          |  return 0;
          |}
          |
       """.stripMargin
    }
    pw.flush()
    pw.close()
    formatSource()
  }
```

Listing 5.7: Implementation of the Mbed code generation location. The code is generated by using Stringwriter. The compiler accumulate the generated code in buffer until the end of the compilation. At the end, the compiler interpolate the string with the contents of his buffers.

Another improvement we can do about the code generation is the management of the frameworks specific objects. For example the Timer/Ticker class in Mbed and Arduino. Arduino allow only ten callback per Timer object and for the moment there is no verification about it. The general improvement is about creating classes that manage the behaviour of the framework specific object we need and make an abstraction for those.

The code generation for the TF language is also simple to implements. The use of sealed traits and case classes allow us to use recursive methods in order to generate an appropriate C source code. As an example, listing 5.8 showed the implementation of the code generation only for the expressions from the TF language. The code is recursively generated by using pattern matching on the AST tokens.

```
// ...
def apply(apdlAst: ApdlAst): String = apdlAst match {
  case e: Expr => e match {
    case Add(left, right) => s"(${apply(left)} + ${apply(right)})"
    case Mul(left, right) => s"(${apply(left)} * ${apply(right)})"
    case Sub(left, right) => s"(${apply(left)} - ${apply(right)})"
    case Div(left, right) => s"(${apply(left)} / ${apply(right)})"
    // ...
  }
case t : TfRetType => t match {...}
// ...
```

Listing 5.8: Implementation, in Scala, of the code generation for TF language expressions. Case classes allow the heavy use of pattern matching and recursive call to compute the whole AST.

The APDL compiler do not really act as a compiler for the moment. It acts like a code translator from APDL to C/C++ with some adjustment in function of the chosen framework for a device.

The last point to mention is about the scope identifier collision. If a user defines a new component like the following.

```
@define component loopCounter {
    @in
    @out int
    @gen arduino {
        global = "int a;"
        setup = "a = 0"
        loop = "a = a + 1;"
        expr = "a"
  }
}
```

The user absolutely don't know if the identifier *a* or *loopCounter* is already defined by another definition or by APDL itself. The lack of such a verification system is not yet implemented in APDL and, as well as other features, is planned for the future.

5.6 Summary

This chapter has presented the actual implementation of the APDL compiler. The main idea to remind is the compilation process illustrated in figure 5.1. The raw source code is firstly parsing into a simple AST before generating into a symbol table. Finally, the code generator uses the symbol table and the remaining AST for the code generation.

We also note that the compiler is by far not finish and a lot of major features and improvements has to be done before having a real product. Some of those features are :

- Type verification during the compilation time.
- A better type system in general.
- Verification about the scope collision in new definition.
- Some modules which simplify the management of framework specific entities.

This is a non-exhaustive list and APDL probably owns several more issues and possible improvements.

Finally, we present the skeleton process for the code generation by showing a small code(5.8) that generate the C code for the expression.

CHAPTER 6

Validation and Results

6.1 Introduction

This part will present the tests, and results obtained with APDL. We will introduce a technique called property-based testing, in which we don't care about the inputs of the test, but about the properties of an entity. Then, we will introduce the implementation of the inputs generators for the tests execution.

Afterwards, we will talk about the validation of APDL towards the other DSL mentioned in chapter 3. We will also indicate why it is hard to validate a DSL in specific area.

Finally, we are going to discuss about the novelties suggested by APDL like as the generalisation of the frameworks for embedded development.

6.2 Testing the parser

A big part of testing is the generation of concrete inputs. It's almost impossible for a single human to create any kind of possible inputs for a parser. That's why, for the project purpose, we are using a technique called property-based testing.

Property-based testing is a kind of testing that allows the user to specify assertions about logical properties which a test function should fulfil. In our case, what the APDL parser should parse and what it should not parse.

ScalaCheck [Ric] is a library that offers property-based testing and also allows the user to create his own generators. For APDL, we have created a set of generators, which are capable of creating any kind of possible valid APDL projects. The full implementation of those generators is available in appendix E.

We will just take an example in order to understand what we have done to test the parser. The following generator is able to generate an expression for the TF language :

```
def genExpr: Gen[Expr] = genExprInner(maxExprSize)

private def genExprInner(depth: Int): Gen[Expr] = {
  if (depth == 0) genExprTerminal
  else {
    val nextDepth = depth - 1
    Gen.oneOf(
      genAdd(nextDepth),genMul(nextDepth),genDiv(nextDepth),
      genSub(nextDepth),genCast(nextDepth),genOr(nextDepth),
      genAdd(nextDepth),genNot(nextDepth),
      genSmaller(nextDepth),genSmallerEquals(nextDepth),
      genEquals(nextDepth),genNotEquals(nextDepth),
      genGreater(nextDepth),genGreaterEquals(nextDepth),
      genFunctionCall(nextDepth),genSymbol,
      genLiteral,genTrue,genFalse
    )
  }
}
```

In order to prevent the framework of generating huge expression which would cause stack overflow, we use a parameter to indicate the maximal depth for expression.

All of the functions contained in the `Gen.oneOf` methods are generators too and are mostly the same as

```
def genAdd(depth: Int): Gen[Add] =
    for {
      e1 <- genExprInner(depth)
      e2 <- genExprInner(depth)
    } yield Add(e1, e2)
```

Once the generators are defined, we can implement a code generator that is producing APDL source code. The generated code allows us to parse it with the APDL parser and finally, we test if the two resulting AST are equals. The code generator for APDL is available in appendix F.

When we have both of those components, we could write tests about logical assertion. Listing 6.1 shows a property which asserts that any generated expression has to be equal to the parsed, and generated code.

```
behavior of "The TransformApdlParser parser"

it should "correctly parse any well formed expr" in {
  val gen = new ApdlExprGenerators(5)
  val codeGen: TransformApdlBackendGenerators = new TransformApdlBackendGenerators {}

  check {
    forAll(gen.genExpr) { e =>
      val code = codeGen.toApdlCode(e)
      val ast = parse(code,tfExpr)
      ast == e
    }
  }
}
```

Listing 6.1: Property of an APDL expression. We assert that for all generated expressions, both of the AST have to be equal.

We can't assume that our parser is correct even with those tests. It's possible that we have made the same error twice, so the test will pass but the parser isn't correct.

We could mention that a test has been created for the preprocessor. APDL allows the user to separate his projects into multiple files and use the `@include` keyword in a C style. The include commands are just replaced by the contents of the specified file. The implementation of this test is available in appendix G.

6.3 Testing the Code Generation

Testing the code generation is harder than testing the parser and consumes much more time. Because we generate the code for different platforms, we have to write many tests cases, and we can't use the generators discussed in section 6.2. We don't have a parser for C source code that can create an APDL AST.

The only test we can make without consuming too much time is to write some APDL source code, generate it and assert that the generated code is working on the expected goal. The sections 6.3.1 to 6.3.3 present various APDL source codes and their corresponding generated outputs, each test indicates if the semantics of the APDL source is respected in the output. Each output has been runned on the specified device and the semantics is respected.

6.3.1 Test A

Input

```
@device A {
    id = uno
    framework = arduino
    @input t analogInput 1
    @serial t update
}
```

Output

```
#include "Timer.h"

Timer t;

int last_serial_t;

void serial_t() {
    // Get data
    int data = analogRead(1);

    if(data != last_serial_t) {
        char buffer[1024];
        sprintf(buffer,"t : %d", (int)data);
        //sprintf(buffer,"t : %d", data);
        Serial.println(buffer);
    }
    last_serial_t = data;
}

void loop(){
    t.update();
}

void setup() {
    Serial.begin(9600);
    last_serial_t = analogRead(1);
    t.every(1000,serial_t);
}
```

Semantic is respected : yes

6.3.2 Test B

Input

```
@device B {
    id = disco_f429zi
    framework = mbed
    @input t analogInput 1
    @input t1 tf t
    @input t2 tf t
    @serial t2 each 2 s
}

@define transform def tf (x:int) -> float {
    val B : int = 3975
    val resistance : float = (float)(1023 - x) * 1000 / x
    val temperature : float = 1 / (log(resistance/1000) / B + 1 / 298.15) - 273.15
    return temperature
}
```

Output

```
#include <mbed.h>

Ticker ticker;

Serial pc(USBTX, USBRX);

AnalogIn temp_2(1);

float tf (int x) {
    int B = 3975;
    float resistance = ((((float)(1023 - x)) * 1000) / x);
    float temperature = ((1 / ((log((resistance / 1000)) / B) + (1 / 298.15))) - 273.15);
    return temperature;
}

void serial_t2() {
    // Get data
    float data = tf(temp_2.read());
    pc.printf("t2 : %f",data);
}

int main(void) {
    pc.baud(9600);
    // Setup
    ticker.attach(&serial_t2,2.0);

    // Loop
    while(1) {
    }

    return 0;
}
```

Semantic is respected : yes

6.3.3 Test C

Input

```
@device arduino1 {
    id = uno
    framework = arduino
    @input rawTemp analogInput 1
    @input lum analogInput 0
    @input temp tf rawTemp
    @input temp2 tf temp
    @input lumTemp simpleOperator + lum temp
    @input tempLum simpleOperator - temp lum
    @input lc loopCounter
    @serial lum each 1 s
    @serial temp each 1 s
    @serial temp2 update
    @serial lumTemp update
    @serial lc update
}

@define transform def tf (x:int) -> float {
    val B : int = 3975
    val resistance : float = (float)(1023 - x) * 1000 / x
    val temperature : float = 1 / (log(resistance/1000) / B + 1 / 298.15) - 273.15
    return temperature
}

@define component simpleOperator op:str {
    @in x:int y:int
    @out int
    @gen mbed {
        global = ""
        setup = ""
        loop = ""
        expr = "@x @op @y"
    }
    @gen arduino {
        global = ""
        setup = ""
        loop = ""
        expr = "@x @op @y"
    }
}

@define component loopCounter {
    @in
    @out int
    @gen arduino {
        global = "int a;"
        setup = "a = 0;"
        loop = "a = a + 1;"
        expr = "a"
  }
}
```

Output

```
#include ”Timer.h”

Timer t;

int a;
float last_serial_temp2;
int last_serial_lumTemp;
int last_serial_lc;

// Transform tf
float tf (int x) {
    int B = 3975;
    float resistance = ((((float)(1023 - x)) * 1000) / x);
    float temperature = ((1 / ((log((resistance / 1000)) / B) + (1 / 298.15))) - 273.15);
    return temperature;
}
// End transform tf

// Component simpleOperator
int component_simpleOperator_lumTemp(int x,int y) {
    return x + y;
}
// End of simpleOperator

// Component simpleOperator
int component_simpleOperator_tempLum(int x,int y) {
    return x - y;
}
// End of simpleOperator

// Component loopCounter
int component_loopCounter_lc() {
    return a;
}
// End of loopCounter

void serial_lum() {
    // Get data
    int data = analogRead(0);
    char buffer[1024];
    sprintf(buffer,”lum : %d”, (int)data);
    //sprintf(buffer,”lum : %d”, data);
    Serial.println(buffer);
}

void serial_temp() {
    // Get data
    float data = tf(analogRead(1));
    char buffer[1024];
    sprintf(buffer,”temp : %d”, (int)data);
    //sprintf(buffer,”temp : %f”, data);
    Serial.println(buffer);
}

void serial_temp2() {
    // Get data
    float data = tf(tf(analogRead(1)));

    if(data != last_serial_temp2) {
        char buffer[1024];
        sprintf(buffer,”temp2 : %d”, (int)data);
        //sprintf(buffer,”temp2 : %f”, data);
        Serial.println(buffer);
```

```
    }

    last_serial_temp2 = data;
}

void serial_lumTemp() {
    // Get data
    int data = component_simpleOperator_lumTemp(analogRead(0),tf(analogRead(1)));

    if(data != last_serial_lumTemp) {
        char buffer[1024];
        sprintf(buffer,"lumTemp : %d", (int)data);
        //sprintf(buffer,"lumTemp : %d", data);
        Serial.println(buffer);
    }

    last_serial_lumTemp = data;
}

void serial_lc() {
    // Get data
    int data = component_loopCounter_lc();

    if(data != last_serial_lc) {
        char buffer[1024];
        sprintf(buffer,"lc : %d", (int)data);
        //sprintf(buffer,"lc : %d", data);
        Serial.println(buffer);
    }

    last_serial_lc = data;
}

void loop() {
    t.update();
    a = a + 1;
}

void setup() {
    Serial.begin(9600);
    a = 0
        t.every(1000,serial_lum);
    t.every(1000,serial_temp);
    last_serial_temp2 = tf(tf(analogRead(1)));
    t.every(1000,serial_temp2);
    last_serial_lumTemp = component_simpleOperator_lumTemp(analogRead(0),tf(analogRead(1)));
    t.every(1000,serial_lumTemp);
    last_serial_lc = component_loopCounter_lc();
    t.every(1000,serial_lc);
}
```

Semantic is respected : yes

6.4 Results

Due to the time taken at the beginning of the project for trying to use LMS in order to implement APDL, we didn't have enough time left for a proper validation of the results. However, we have created multiple APDL projects using the DSL and have obtained some good results.

6.4.1 Code Generation

The code generation is working for Arduino and for Mbed. PlatformIO's projects are well generated and also used for library dependencies. PlatformIO enables us to specify several libraries and the tool downloads them itself.

The results about the code generation are also part of the testing on the same subject discussed in sections 6.3. In order to obtain some verifiable results, we have to use APDL and collect feedback.

6.4.2 APDL Versus the Rest of the World

Comparing APDL with the DSL analysed in chapter 3 isn't trivial. We haven't found a DSL which is trying to do the same thing as ours. We are probably the only one who are working on this kind of DSL which is targeting low-level device for a specific kind of work.

Between all the mentioned DSL in chapter 3, Node-Red[IBM17] is the most famous and used one. In contrast to APDL, Node-red is acting on a high-level entity like Twitter, Email or MQTT messages. There are no notions of low-level or embedded devices. Some extensions exist for connecting Raspberry Pi with Node-red, but that's not the priority of the developers.

An idea to validate APDL would be to create a set of representative projects to implement with various kinds of DSL for IoT and to compare the results. The issue with this validation is the time. The necessary time to find people who already know the other technologies is huge. In addition, as mentioned in chapter 2, the purpose of a DSL is to be domain-specific. Comparing multiple DSL with different purposes is probably not a good idea.

The last way to validate APDL is to use it over the time and get the feedback from the user in order to improve it. This brings us to one of the key purposes of a DSL. The biggest advantage of developing a DSL is also its biggest disadvantage : the domain-specific orientation. The domain-specific constraint is very painful when comes time to validate the result against another similar purpose product.

6.5 APDL Contribution to the Internet of Things

APDL is, by far, not finish yet. A lot of work needs to be done to round off and improve the concept, but the foundations are here. With APDL, we have shown that it is possible to design a very high definition language for very low-level purpose. One step further, the language is purely declarative except for the definition of new entities. A declarative language is, in general, simpler to read than an imperative one.

Another concept is the generalisation of the embedded devices. Conceptually, any IoT device and associated framework could be generalised into an entity which contains the following properties:

- An initialisation phase, which is the set of operations executed once, and called “setup”.
- A repeated loop phase, which is the set of operations executed at each interval of times, and called “loop”.
- Some input functions or entities, which are gathering the environment values into the software part.
- Some transformation functions, which are manipulating the input values.
- Some components, which contain what we can’t generalise for any frameworks or devices.

Even if APDL is not yet ready for the production, the established concept could bring novelty to the IoT world. The improvement of APDL or even the development of new languages, frameworks or DSL could lead to a new era of progress in IoT development.

6.6 Summary

We have presented the validation and the results obtained with the APDL DSL. We introduced the property-based testing techniques used to validate the parsers. We generated random AST which are converted into APDL code, before parsed by the parser and transformed into a new AST.

Testing the code generation is a lot trickier and time consuming, a set of tests have been created in order to obtain some kind of a validation.

Finally, we discussed about the novelties brought by APDL into the IoT field. We saw that even if APDL is not ready for the production, the concept known from the development phase is relevant for the future development of the IoT.

CHAPTER 7

The Acquire and Process Description Language Ecosystem

7.1 Introduction

The APDL ecosystem is the work done around the compiler in order to validate the project. We will show the implementation of the serial handler and some example, so the user will be able to create his own handler if he wants.

Then, we will show the storage and the visualisation of the data. The technologies used for this part are recent and completely free.

Finally, we will implement, from scratch, an APDL system and show how to launch the generated entity one by one.

7.2 The Serial Handler

The APDL DSL don't go further than the serial network. The system only receives, process and sends the data through a serial with a certain format. After that, APDL don't know what happens to the data.

A very simple serial handler has been developed using the Python language. Python offers a library called "pyserial" which simplify the handling of serial connections. The listing 7.1 shows the very simple implementation of a serial handler using Python.

```
import serial
arduino = serial.Serial('/dev/ttyACM0', 9600, bytesize=8, timeout=1)

data = arduino.readLine()
asciiData = data.decode('ascii')

# format of data = "topic : value"
array = asciiData.split(":")
topic: str = array[0]
value: str = array[1]

# replace the \n from the device
value = value.replace("\n","",1)
value = value.replace("\r","",1)
```

Listing 7.1: Simple serial handler implementation in Python. Pyserial simplifies the recovering process and the Python standard library offers some functions to decode the data.

Once the data has been recovered, the user can do anything he wants with it. For example, the APDL handler implemented as part of this project is sending the data to an InfluxDB database[Inf].

APDL offers the possibility to automaticaly generate an handler implementation in python with the corresponding identifier. This possibility is activated by using the option "–generate-handler" with the APDL compiler. Otherwise, the user could define his own handler.

Listing 7.2 shows an implementation of an APDL handler. The implementation is very small and allow the user to completly modify as much as he wants.

```
import serial

from datetime import datetime
from influxdb import InfluxDBClient


def main():
    arduino = serial.Serial('/dev/ttyACM0', 9600, bytesize=8, timeout=1)

    user = 'root'
    password = 'root'
    dbname = 'apdl-default'

    client = InfluxDBClient('localhost', 8086, user, password, dbname)
    client.drop_database(dbname)
    print("Create database")
    client.create_database(dbname)

    # The format is
    # topic : value

    while True:
        data = arduino.readline()
        asciiData: str = data.decode('ascii')
        try:
            array = asciiData.split(":")
            topic: str = array[0]
            value: str = array[1]
            value = value.replace("\n", "", 1)
            value = value.replace("\r", "", 1)
            value = value.replace(" ", "")
            topic = topic.replace(" ", "")
            if "temp" not in topic:
                continue
            print("Send to influxdb == " + topic + " : " + value)
            json = [
                {
                    "measurement": topic,
                    "fields": {
                        "value": int(value)
                    }
                }
            ]
            client.write_points(json)
        except IndexError:
            continue
        except serial.serialutil.SerialException:
            continue

if __name__ == "__main__":
    main()
```

Listing 7.2: Example of an APDL Handler which is also working with InfluxDB. The data are recovered from the serial connection and send to InfluxDB.

7.3 Data Storage and Visualisation

The APDL compiler also produce a Docker-compose configuration file. This configuration file allows a simple ecosystem generation by using Docker. In order to store the data acquired by the device and to visualise them, we used two products : InfluxDB and Grafana.

InfluxDB[Inf] is a real-time database focused on time series. The data we aquire through APDL could be seen as time series, so InfluxDB is perfect for this kind of project.

Grafana[Gra] is an open plaform for data analytics, monitoring and visualisation. The tool is provided with an automatic connection to InfluxDB. Figure 7.1 shows Grafana in action, we can see a graph representation of the temperature values recovered by an APDL device.

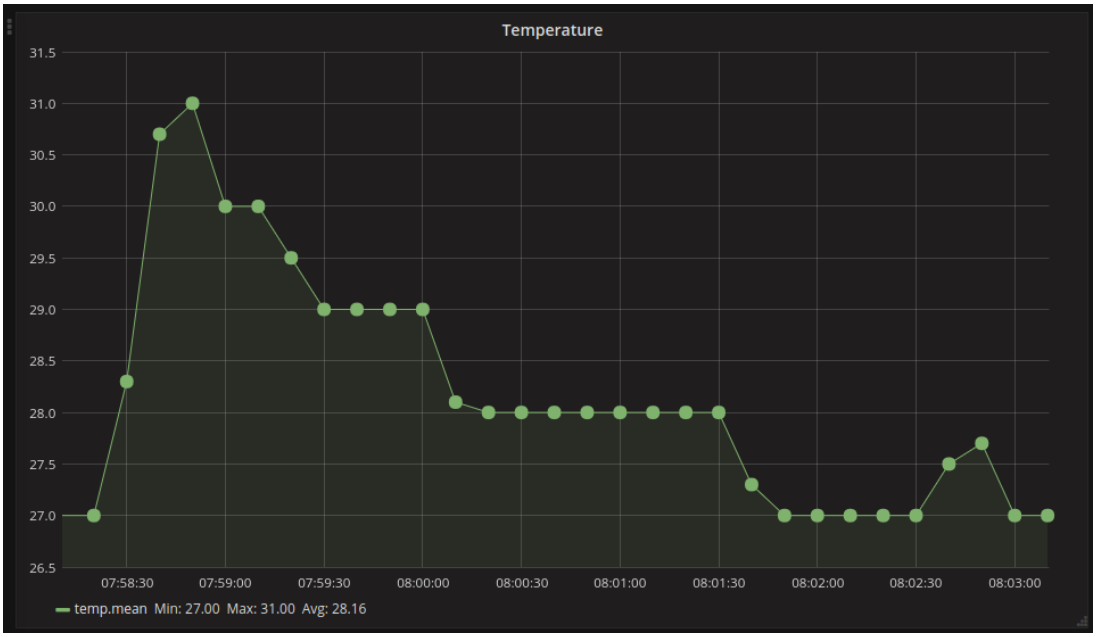


Figure 7.1: Visualisation of the temperature acquired by an APDL device. The visualisation is done by using Grafana, a free plaform for data analytics, monitoring and visualisation.

Unfortunately, current APDL compiler does not provide an immediate way of visualising the data. When we launch the project for the first time, the Grafana image is pull by docker and the user needs to configure the visualisation by himself.

7.4 A Complete Example

As an example, we will show a complete development of an APDL project from the start to the end. The goal is to build a simple system to visualise the temperature of the room. The chosen device is the Arduino UNO [Sto14]. The figure 7.2 shows the installation of the device with a temperature sensor.

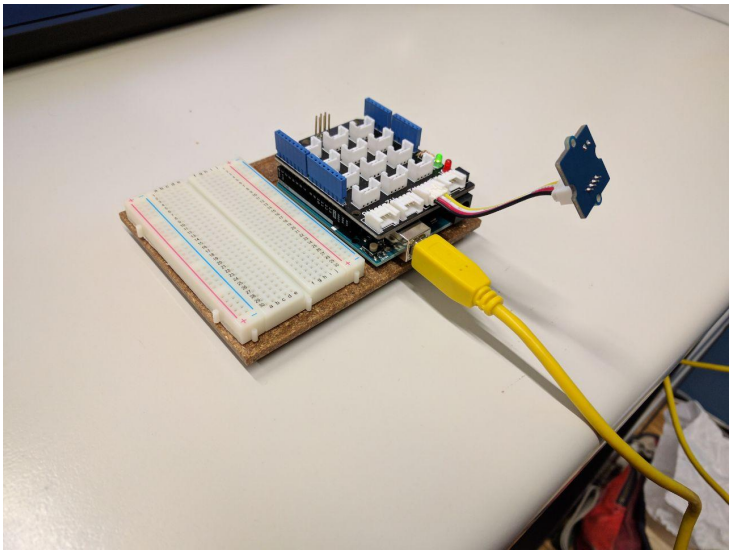


Figure 7.2: Device installation of the complete example. We just use a temperature sensor connected to the pin “A1” of the Arduino.

Now we will write the code of our corresponding system. The listing 7.3 shows the implementation, with APDL, of the project. Firstly, we define the name of the project and includes a file named “apdl_component.apdl”. This file, available in appendix H, provides the analog and digital inputs definition for Mbed and Arduino. Then, we can declare our device with the corresponding inputs and serials. Finally, we define a transformation function named “tf”. The temperature sensor on the arduino give back the value of the resistance. So we need to compute the temperature with this value.

```
project_name = "Example"

@include "apdl_component.apdl"

@device arduino1 {
    id = uno
    framework = arduino
    @input t analogInput 1
    @input temp tf t
    @serial temp each 1 s
}

@define transform def tf (x:int) -> float {
    val B : int = 3975
    val resistance : float = (float)(1023 - x) * 1000 / x
    val temperature : float = 1 / (log(resistance/1000) / B + 1 / 298.15) - 273.15
    return temperature
}
```

Listing 7.3: APDL implementation of the complete example. We declare a device with the id “uno” and the framework “arduino” then we import the file “apdl_component.apdl” which provides some inputs. Finally, we define the transformation function “tf” using the TF language, as well as the inputs and the serial.

Once we are compiling the project, the compiler generates a directory with all the necessary entities to launch the project. The generated bash script executes the following commands :

```
# upload the code to the device
platformio run -t upload
# launch the docker images for influxdb and grafana
docker-compose up
# launch the serial handler and start sending the data to influxdb
python handler.py
```

Now, we can visualise the temperature of the room with Grafana. For that, we open a browser and write "localhost:3000". Now we can look at the room's temperature. First, we have to configure a data source with a direct connection to influxdb. The identifier and password for the administator access are "root". Finally, we have to create a Dashboard, select the corresponding points in influxdb and that's all.

Further work is planned in order to fully automated the APDL compilation process so that the user has the minimal set of action to do.

7.5 Summary

The work done around the APDL compiler is simple and easely reproducible. We saw the implementation of the APDL's handler and how the user can modify it for personal purposes.

We also documented the storage and visualisation of the data. Both used technologies, InfluxDB and Grafana, are free and very easy to use even for people whose knowledge is basic.

Finally, we saw a complete example of an APDL project implemented from scratch and the its results. Not all the part are fully automated but further work are planned to do so.

CHAPTER 8

Conclusion

8.1 Summary

Looking back to the challenge laid down at the beginning of this project, we have shown with concrete examples how we can generalise the concept of IoT and embedded programming. The cases we have investigated, generalisations of embedded programming, DSL for the development of specific IoT projects and the conception of an entire ecosystem based on APDL, are promising and could lead to a new era of simplification and development for the IoT world.

In chapter 2 and 3, we set our work's foundations. The design and development of a DSL is complex and the specific domain of investigation, IoT and embedded devices, is wide and involves different technologies.

Chapter 4 deal with the design of the APDL DSL and explain the choices made during its construction, especially the fragmentation of APDL into multiple DSL. We also showed the design of the TF language and argued why we need an additional programming language to describe transformation. Finally, we suggested the proposition of generalising the development of sensor-oriented projects. We introduced the concepts of inputs, devices, transformations, samplings and APDL lifecycle.

In chapter 5, we illustrated and explained the implementation of the APDL compiler. The compilation process shown in figure 5.1 decrypt perfectly the modelling of the compiler. This chapter also notes that the compiler is by far not finish and some first-class features and improvements have to be done before running APDL into the production.

Chapter 6 try to validate the work done so far. Firstly, we introduced the concept of property-based testing and its application to the parser testing and validation. In a second time, we presented the generators used to generate almost any kind of valid program for APDL. We also tried to present why it's hard to validate a DSL against another technology and suggested a last way to validate APDL. At the end, we presented the contribution of APDL to the IoT world, and the properties hold by the generalisation of any IoT devices.

Finally, chapter 7 explains the technological choices made in order to use the APDL DSL inside a bigger system. Grafana and InfluxDB are both free and easy to use for any people. At the end, we presented a complete realisation of a simple IoT project from scratch and proved that it's possible to simplify the development process of IoT devices.

In the following sections, we will present the future of the projects, and the possible improvements and features to implements in APDL. We will end this thesis by the acknowledgement for all the people involved in this work and with a personal statement.

8.2 Further Work

As said before, the APDL compiler is not ready for the production but the foundations for future work are ready. APDL is an open source project, and its usage is not limited in any way. Contributions are welcome from anybody. The following list shows the improvements and features that could be added to APDL :

- Verify the types during the compilation and a better type system in general.

- Verify and prevents variable name collision in the same scope.
- Allow the definition of a new framework with the APDL language in order to prevent compiler modification.
- Implement a standard library and a library manager for APDL in APDL.
- Validate the compiler by using APDL in larger projects and get feedback from the users.

8.3 Acknowledgement

I would like to sincerely thank the following people, without whom this project would have been impossible to realise. Their works, support and advice have been crucial during those four months of hard work.

- Professor Dr. Pierre-André Mudry, the initiator and supervisor of this thesis, for supporting me and providing the key advice to complete this project.
- My mother, Manuela Julmy, for the support and advice she gave me during all this period.
- My father, Roland Julmy, for having reviewed this document and encouraging me.
- My uncle, Marcel Julmy, for having reviewed this document and encouraging me.
- To all the people or organisation I forgot, many thanks !

8.4 Personal statement

I have spent the last four months on this thesis, and working alone most of the time could be really hard. As mentioned previously, the support I received during this time was crucial for the success of this thesis, but also for me not to be discouraged by the complexity of this task.

The results are promising and I am satisfied of the obtained results. I believe that APDL could be used by different companies to simplify some internal process. I see an application of this work in the growing fields of Data Science and Machine Learning, in which the systems need more and more data to analyse.

Finally, working on this thesis open me to several domains and technologies that I didn't know that much. I took the opportunity to improve my knowledge in several fields such as embedded hardware and software, compiler construction or IoT concepts. I enjoyed a lot doing this thesis and hope that all the accumulate knowledge is going to be useful for any interested people.

Fribourg, the 29th of June of 2017

Sylvain Marcel Julmy

Bibliography

[Ard17a] Arduino. *Arduino introduction*. 2017.

[Ard17b] Arduino Software. *Arduino - Open Source Products for Electronic Projects*. 2017.

[Beh+11] S Behnel et al. "Cython: The Best of Both Worlds". In: *Comput. Sci. Eng.* 13.2 (2011), pp. 31–39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118.

[Ber+14] Benjamin Bertran et al. "DiaSuite: A tool suite to develop Sense/Compute/Control applications". In: *Sci. Comput. Program.* 79.4 (2014), pp. 39–51. ISSN: 01676423. DOI: 10.1016/j.scico.2012.04.001.

[Com17] Wikipedia Community. *D-17B*. 2017.

[Dav] Tal Davidson. "Artistic style Detector". In: *online* ().

[Deu+98] Arie Van Deursen et al. "Little Languages: Little Maintenance". In: *J. Softw. Maint.* 10.2 (1998), pp. 75–92. ISSN: 1040-550X. DOI: 10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5.

[Edg15] Jake Edge. *Python without an operating system*. 2015. URL: https://lwn.net/Articles/641244/.

[EK15] T Eichstadt-Engelen and K Kreuzer. *openHAB: Automatisiertes Heim* -. shortcuts ptie.˜1. entwickler.Press, 2015. ISBN: 9783868025590.

[For06] Bryan Ford. "Packrat Parsing: Simple, Powerful, Lazy, Linear Time". In: *Icfp* (2006), p. 12. ISSN: 03621340. DOI: 10.1145/581478.581483.

[Gon+14] Cristian Gonzalez Garcia et al. "Midgar: Generation of heterogeneous objects interconnecting applications. A Domain Specific Language proposal for Internet of Things scenarios". In: *Comput. Networks* 64.May 2014 (2014), pp. 143–158. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2014.02.010.

[Gra] Grafana Labs. *Grafana - The open platform for analytics and monitoring*.

[IBM17] Dave Conway-Jones IBM Emerging Technology Nick O'Leary. *Node-RED*. 2017. URL: http://nodered.org/.

[IDV17] Kravets Ivan, Kyrychuk Dmytro, and Koval Valerii. *An open source ecosystem for IoT development ů PlatformIO*. 2017.

[Inf] Inc. InfluxData. *InfluxData (InfluxDB) | Time Series Database Monitoring Metrics & Events*.

[Kri13] Filip Krikava. *Domain specific languages and Scala*. Presentation given to Rivier Scala / Clojure User Group. 2013.

[mbe] ARM mbed. *Mbed*.

[MCB14] Sébastien Mosser, Philippe Collet, and Mireille Blay-Fornarino. "Exploiting the Internet of Things to teach domain-specific languages and modeling the arduinoML project". In: *CEUR Workshop Proc.* 1346 (2014), pp. 45–54. ISSN: 16130073.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892.

[Moo17] Sebastien Mooser. *Arduino-ML-kernel*. 2017.

[OSV16] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: Updated for Scala 2.12*. 3rd. USA: Artima Incorporation, 2016. ISBN: 0981531687, 9780981531687.

[Ric] Rickard Nilsson. *ScalaCheck*.

[RO10] Tiark Rompf and Martin Odersky. "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *SIGPLAN Not.* 46.2 (2010), pp. 127–136. ISSN: 0362-1340. DOI: 10.1145/1942788.1868314.

[Rom+12] Tiark Rompf et al. "Scala-Virtualized: linguistic reuse for deep embeddings". In: *Higher-Order Symb. Comput.* 25.1 (2012), pp. 165–207. ISSN: 1573-0557. DOI: 10.1007/s10990-013-9096-9.

[Ros95] Guido Rossum. *Python Reference Manual*. Tech. rep. Amsterdam, The Netherlands, The Netherlands, 1995.

[Sal+15] A. Salihbegovic et al. "Design of a domain specific language and IDE for Internet of things applications". In: *2015 38th Int. Conv. Inf. Commun. Technol. Electron. Microelectron. MIPRO 2015 - Proc.* May (2015), pp. 996–1001. DOI: 10.1109/MIPRO.2015.7160420.

[SN16] Manfred Sneps-Sneppe and Dmitry Namiot. "On web-based domain-specific language for Internet of Things". In: *Int. Congr. Ultra Mod. Telecommun. Control Syst. Work.* 2016-Janua (2016), pp. 287–292. ISSN: 2157023X. DOI: 10.1109/ICUMT.2015.7382444.

[Sto14] Arduino Store. *Arduino UNO Rev3*. 2014.

[SZ09] Mark Strembeck and Uwe Zdun. "An approach for the systematic development of domain-specific languages". In: *Softw. - Pract. Exp.* 39.15 (2009), pp. 1253–1292. ISSN: 00380644. DOI: 10.1002/spe.936.

[TK10] Juha-Pekka Tolvanen and Steven Kelly. "Integrating models with domain-specific modeling languages". In: *Proc. 10th Work. Domain-Specific Model. - DSM '10* (2010), p. 1. DOI: 10.1145/2060329.2060354.

[TMD09] R N Taylor, N Medvidovic, and E M Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[TW12] Rob Toulson and Tim Wilmshurst. *Fast and Effective Embedded Systems Design: Applying the ARM Mbed*. 1st. Newton, MA, USA: Newnes, 2012.

[VKV00] A. Van Deursen, Paul Klint, and Joost Visser. "Domain-specific languages: an annotated bibliography". In: *ACM Sigplan Not.* 35.June (2000), pp. 26–36. ISSN: 03621340. DOI: 10.1145/352029.352035.

List of Figures

List of Listings

Appendices

Appendix A

APDL Language Introduction

The syntax of the APDL language is very simple, and the goal of this document is to provide a small introduction to APDL.

Creating a project

A project is always identified by a name through the `project_name` key-value pair. We could also fragment the project into multiple files and include them with the `@include` keyword. So, an empty project looks like :

```
project_name = "Example"

@include "apdl_component.apdl"
@include "other_file.apdl"
```

We also need to note that the comments aren't supported for the moment.

Defining a device

A device is the key entity of an APDL project, and it is defined by using the `@device` keyword with an identifier. The identifier has to be unique and don't containing any special character. The full grammar of APDL is available in appendix C.

When we define a device, we have to specify an "id" and a "framework" :

```
@device example {
  id = Uno
  framework = arduino
}
```

For the moment, only “arduino” and “mbed” are supported for the framework value. The “id” value corresponds to the one available with PlatformIO. [1]. In addition to these two values, we could define inputs with the `@input` keyword and serial with the `@serial` keyword.

A serial is always defined in the same way :

```
@serial inputIdentifier samplingType
```

The “inputIdentifier” corresponds to the identifier of a previously defined input in the same device. The “samplingType” corresponds to the kind of sampling to use and its potential value. For the moment, there are two kinds of sampling :

- by update : `@serial inputIdentifier update`.
- by frequency : `@serial inputIdentifier each n timeUnit`, where n is an integer and *timeUnit* is a time unit.

The time units are corresponding like the following map :

- `ns` : nanoseconds
- `ms` : milliseconds
- `s` : seconds
- `m` : minutes
- `h` : hours
- `d` : days

The definition of an input for a device is described in the following sections.

Defining new inputs

Defining new inputs is available through the `@define input` keyword. The input needs a unique identifier and a parameter list. The list could be empty and the parameter is written with the pattern `identifier : type`. The type is the following :

- `str` : a unique string, without space.
- `id` : an identifier.
- The other types : int, float, long, bool, double, char, byte and short, have the same behaviour as in the C language.

Once the input signature has been defined, we have to specify the `@gen` part. This part owns five key-value pairs that are always in the same order and are always required by the compiler. Except of the “type” value, the others are literal strings that are going to be generated inside the source code of the target. So the code written in those values are specific to the target language or framework. For example, the analogue input for arduino and mbed is defined with the following :

[1] `http://platformio.org/boards`

```
@define input analogInput pin:str {
    @gen mbed {
        global = "AnalogIn @id(@pin);"
        setup = ""
        loop = ""
        expr = "@id.read()"
        type = float
    }
    @gen arduino {
        global = ""
        setup = ""
        loop = ""
        expr = "analogRead(@pin)"
        type = int
    }
}
```

The four fields are :

- "global" : printed inside the global definition.
- "setup" : printed inside the setup definition.
- "loop" : printed inside the loop definition.
- "expr" : the expression inside the target source code which provides the value.

For example, the location of those fields are like the following for Arduino :

```
// Global definition

void setup() {
  // Setup
}

void loop() {
  // Loop

  // Expr and type
  /* type */ data = /* expr */;
}
```

or for Mbed :

```
// Global definition

int main() {
  // Setup

  while(1) {
    // Loop

    // Expr and type
    /* type */ data = /* expr */;
  }

}
```

We need to be careful when defining global variable, because the identifier collisions are not verified.

To use a defined input, we just write the type of the input after its identifier and provide the required number of arguments after the type :

```
@input temp analogInput 1
```

Defining new components

Defining new components is almost the same as defining new inputs, except that the **@gen** block don't have a "type" key-value pair but two additional information for the components itself, the input's identifier and type, and the output's type :

```
@define component simpleOperator op:str {
    @in x:int y:int
    @out int
    @gen mbed {
        global = ""
        setup = ""
        loop = ""
        expr = "@x @op @y"
    }
    @gen arduino {
        global = ""
        setup = ""
        loop = ""
        expr = "@x @op @y"
    }
}

@define component loopCounter {
    @in
    @out int
    @gen arduino {
        global = "int a;"
        setup = "a = 0;"
        loop = "a = a + 1;"
        expr = "a"
  }
}
```

A component is generated as a function for the moment, so we can't have a component with multiple outputs.

The usage of a component is almost the same as for the input, but the arguments of the component come before the inputs arguments :

```
@input componentedTemp simpleOperator + inputA inputB
```

Defining new transformations

Defining new transformations is quite different from the definition of new inputs or components. To define new transformations, we use an additional DSL called TF, for TransForm Language. The TF language is very similar to Scala and acts as a high-level language.

The complete grammar of the TF language is available in appendix C. The major differences with Scala are :

- The types are written with lowercase letters.
- The types are obligatory, there is no type inference.
- The function definition output type is given after `->` instead of `:`.
- Type casting is available with the C syntax : `(int)3.5`
- The `return` keyword is required for a non-void function.

Here is an example of the definition of a transformation function :

```
@define transform def tf (x:int) -> float {
    val B : int = 3975
    val resistance : float = (float)(1023 - x) * 1000 / x
    val temperature : float = 1 / (log(resistance/1000) / B + 1 / 298.15) - 273.15
    return temperature
}
```

Appendix B

APDL Abstract Syntax Tree Implementation

MainParsers.scala

```
package apdl.parser

import apdl.ApdlParserException

import scala.Function.tupled
import scala.util.matching.Regex

class MainParsers extends DefineParsers {

  override protected val whiteSpace: Regex = "[ \t\r\f\n]+".r

  override def skipWhitespace: Boolean = true

  val ws: Regex = whiteSpace

  def program: Parser[ApdlProject] = {
    def process(xs: List[Object]): ApdlProject = {
      val projectName: String = xs.find(_.isInstanceOf[String]) match {
        case Some(value) => value.asInstanceOf[String]
        case None => throw new ApdlParserException("No project name specifying")
      }

      val devices: List[ApdlDevice] =
        ↪  xs.filter(_.isInstanceOf[ApdlDevice]).map(_.asInstanceOf[ApdlDevice])

      val defineInputs: List[ApdlDefineInput] = xs
        .filter(_.isInstanceOf[ApdlDefineInput])
        .map(_.asInstanceOf[ApdlDefineInput])

      val defineComponents: List[ApdlDefineComponent] = xs
        .filter(_.isInstanceOf[ApdlDefineComponent])
        .map(_.asInstanceOf[ApdlDefineComponent])

      val defineTransforms: List[ApdlDefineTransform] = xs
        .filter(_.isInstanceOf[ApdlDefineTransform])
        .map(_.asInstanceOf[ApdlDefineTransform])

      ApdlProject(projectName, devices, defineInputs, defineComponents, defineTransforms)
```

```
    }

    rep1(projectName | apdlDevice | apdlDefine) ^^ {
      xs =>
        process(xs)
    }
  }

  def projectName: Parser[String] = "project_name" ~ "=" ~ "\"" ~> literalString <~ "\"" ^^ {
    ↪  str => str }

  def keyValue: Parser[(String, String)] = identifier ~ "=" ~ identifier ^^ { case (k ~ _ ~
    ↪  v) => (k, v) }

  def apdlInput: Parser[ApdlInput] = "@input" ~> identifier ~ identifier ~ apdlParameters ^^
    ↪  {
    case (name ~ typ ~ params) => ApdlInput(name, typ, params)
  }

  def apdlParameters: Parser[List[String]] = rep(apdlParameter)

  def apdlParameter: Parser[String] = "[^ \t\f\n\r{}@]+".r ^^ { str => str }

  def apdlSerial: Parser[ApdlSerial] = "@serial" ~> identifier ~ apdlSampling ^^ {
    case (ident ~ sampling) => ApdlSerial(ident, sampling)
  }

  def apdlSampling: Parser[ApdlSampling] = apdlSamplingUpdate | apdlSamplingTimer

  def apdlSamplingUpdate: Parser[ApdlSamplingUpdate.type] = "update" ^^ { _ =>
    ↪  ApdlSamplingUpdate }

  def apdlSamplingTimer: Parser[ApdlSamplingTimer] = "each" ~> "[0-9]+".r ~ timeUnit ^^ {
    ↪  case (value ~ tu) => ApdlSamplingTimer(value.toInt, tu) }

  def timeUnit: Parser[ApdlTimeUnit] = {
    "ns" ^^ { _ => ApdlTimeUnit.ns } |
      "ms" ^^ { _ => ApdlTimeUnit.ms } |
      "s" ^^ { _ => ApdlTimeUnit.s } |
      "m" ^^ { _ => ApdlTimeUnit.m } |
      "h" ^^ { _ => ApdlTimeUnit.h } |
      "d" ^^ { _ => ApdlTimeUnit.d }
  }

  def apdlDevice: Parser[ApdlDevice] = {

    def process(ident: String, xs: List[Object]): ApdlDevice = {
      val inputs = xs.filter(_.isInstanceOf[ApdlInput]).map(_.asInstanceOf[ApdlInput])
      val serials = xs.filter(_.isInstanceOf[ApdlSerial]).map(_.asInstanceOf[ApdlSerial])
      val keyValues = xs.filter(_.isInstanceOf[(String,
        ↪  String)]).map(_.asInstanceOf[(String, String)])
      val framework = (keyValues find tupled((k, _) => k == "framework")).getOrElse(throw new
        ↪  ApdlParserException(s"No framework specify for $ident"))._2
      val id = (keyValues find tupled((k, _) => k == "id")).getOrElse(throw new
        ↪  ApdlParserException(s"No id specify for $ident"))._2
      val parameters = (keyValues filter tupled((k, _) => k != "id" && k !=
        ↪  "framework")).toMap
      ApdlDevice(ident, id, framework, inputs, serials, parameters)
    }

    "@device" ~> identifier ~ (lb ~> rep1(keyValue | apdlInput | apdlSerial) <~ rb) ^^ { case
      ↪  (ident ~ xs) => process(ident, xs) }
  }
}

case class ApdlProject(
```

```
                        name: String,
                        devices: List[ApdlDevice],
                        defineInputs: List[ApdlDefineInput],
                        defineComponents: List[ApdlDefineComponent],
                        defineTransforms: List[ApdlDefineTransform]
                      )

case class ApdlDevice(name: String, id: String, framework: String, inputs: List[ApdlInput],
  ↪  serials: List[ApdlSerial], additionalParameters: Map[String, String])

case class ApdlInput(identifier: String, defineInputIdentifier: String, args: List[String])

case class ApdlSerial(inputName: String, sampling: ApdlSampling)

sealed trait ApdlSampling
case object ApdlSamplingUpdate extends ApdlSampling
case class ApdlSamplingTimer(value: Int, timeUnit: ApdlTimeUnit) extends ApdlSampling {

  def ms: Int = timeUnit match {
    case ApdlTimeUnit.ns => value / 1000
    case ApdlTimeUnit.ms => value
    case ApdlTimeUnit.s => value * 1000
    case ApdlTimeUnit.m => value * 1000 * 60
    case ApdlTimeUnit.h => value * 1000 * 60 * 60
    case ApdlTimeUnit.d => value * 1000 * 60 * 60 * 24
  }

  def s:Int = timeUnit match {
    case ApdlTimeUnit.ns => value / 1000000
    case ApdlTimeUnit.ms => value / 1000
    case ApdlTimeUnit.s => value
    case ApdlTimeUnit.m => value * 60
    case ApdlTimeUnit.h => value * 60 * 60
    case ApdlTimeUnit.d => value * 60 * 60 * 24
  }
}

sealed trait ApdlTimeUnit
object ApdlTimeUnit {
  case object ns extends ApdlTimeUnit
  case object ms extends ApdlTimeUnit
  case object s extends ApdlTimeUnit
  case object m extends ApdlTimeUnit
  case object h extends ApdlTimeUnit
  case object d extends ApdlTimeUnit

  def values: Seq[ApdlTimeUnit] = Seq(ns, ms, s, m, h, d)
}
```

DefineParsers.scala

```
package apdl.parser

import apdl.ApdlCodeGenerationException
import apdl.parser.ApdlType.{Id, Str}

import scala.util.matching.Regex
import scala.util.parsing.combinator.{PackratParsers, RegexParsers}

class DefineParsers extends TransformDslParser with RegexParsers with PackratParsers {
```

```
override protected val whiteSpace: Regex = "[ \t\r\f\n]+".r
override def skipWhitespace: Boolean = true

lazy val defines: PackratParser[List[ApdlDefine]] = rep(apdlDefine)
lazy val apdlDefine: PackratParser[ApdlDefine] = "@define" ~> (defineComponent |
  ↪  defineInput | defineTransform)

lazy val defineComponent: PackratParser[ApdlDefineComponent] = {
  "component" ~> identifier ~ parameters ~ lb ~ defineComponentBody ~ rb ^^ {
    case (i ~ params ~ _ ~ ((in, out, gs)) ~ _) => ApdlDefineComponent(i, params, in, out,
      ↪  gs)
  }
}

lazy val defineComponentBody: PackratParser[(Inputs, Output, Map[String, Gen])] = {
  inputs ~ output ~ gens ^^ { case (i ~ o ~ g) => (i, o, g) }
}

lazy val inputs: PackratParser[Inputs] = "@in" ~> parameters ^^ { ps => Inputs(ps) }
lazy val output: PackratParser[Output] = "@out" ~> apdlType ^^ { typ => Output(typ) }
lazy val gens: PackratParser[Map[String, Gen]] = rep(gen) ^^ { gs => gs.toMap }
lazy val gen: PackratParser[(String, Gen)] = "@gen" ~> identifier ~ (lb ~> genBody) <~ rb
  ↪  ^^ {
  case (i ~ b) => i -> b
}

lazy val genBody: PackratParser[Gen] = {
  global ~ setup ~ loop ~ expr ~ (genType?) ^^ { case (g ~ s ~ l ~ e ~ t) => Gen(g, s, l,
    ↪  e, t) }
}

lazy val global: PackratParser[String] = "global" ~ "=" ~ "\"" ~> literalString <~ "\"" ^^
  ↪  { str => str }
lazy val setup: PackratParser[String] = "setup" ~ "=" ~ "\"" ~> literalString <~ "\"" ^^ {
  ↪  str => str }
lazy val loop: PackratParser[String] = "loop" ~ "=" ~ "\"" ~> literalString <~ "\"" ^^ {
  ↪  str => str }
lazy val expr: PackratParser[String] = "expr" ~ "=" ~ "\"" ~> literalString <~ "\"" ^^ {
  ↪  str => str }
lazy val literalString: PackratParser[String] = """(\\.|[^\\"])*""".r ^^ { str => str }
lazy val genType : PackratParser[ApdlType] = "type" ~ "=" ~> apdlStdType

lazy val defineInput: PackratParser[ApdlDefineInput] = "input" ~> identifier ~ parameters
  ↪  ~ (lb ~> gens <~ rb) ^^ {
  case (defId ~ defParams ~ defGens) => ApdlDefineInput(defId, defParams, defGens)
}

lazy val apdlType: PackratParser[ApdlType] = str | id | apdlStdType
lazy val str: PackratParser[ApdlType.Str.type] = "str" ^^^ ApdlType.Str
lazy val id: PackratParser[ApdlType.Id.type] = "id" ^^^ ApdlType.Id
lazy val apdlStdType: PackratParser[ApdlType] = {
  "int" ^^^ ApdlType.Int |
    "float" ^^^ ApdlType.Float |
    "double" ^^^ ApdlType.Double |
    "char" ^^^ ApdlType.Char |
    "byte" ^^^ ApdlType.Byte |
    "short" ^^^ ApdlType.Short |
    "bool" ^^^ ApdlType.Bool |
    "long" ^^^ ApdlType.Long
}

lazy val parameters: PackratParser[List[Parameter]] = rep(parameter)
lazy val parameter: PackratParser[Parameter] = identifier ~ (":" ~> apdlType) ^^ { case (i
  ↪  ~ t) => Parameter(i, t) }

lazy val number: PackratParser[String] = "[-+]?[0-9]+.?[0-9]*".r ^^ { str => str }
```

```
  lazy val defineTransform: PackratParser[ApdlDefineTransform] = {
    "transform" ~> tfFunctionDeclaration ^^ { f => ApdlDefineTransform(f) }
  }
}

case class Inputs(parameters: List[Parameter])
case class Output(outputType: ApdlType)
case class Gen(global: String, setup: String, loop: String, expr: String, typ :
 ↪  Option[ApdlType])

sealed trait ApdlDefine {
  def identifier: String = this match {
    case ApdlDefineInput(name, _, _) => name
    case ApdlDefineComponent(name, _, _, _, _) => name
    case ApdlDefineTransform(functionDecl) => functionDecl.header.identifier
  }
}
case class ApdlDefineInput(name: String, parameters: List[Parameter], gens: Map[String,
 ↪  Gen]) extends ApdlDefine
case class ApdlDefineComponent(name: String, parameters: List[Parameter], inputs: Inputs,
 ↪  outputType: Output, gens: Map[String, Gen]) extends ApdlDefine
case class ApdlDefineTransform(functionDecl: FunctionDecl) extends ApdlDefine

case class Parameter(id: String, typ: ApdlType)

sealed trait ApdlType {
  override def toString: String = this match {
    case Str => "str"
    case Id => "id"
    case ApdlType.Int => "int"
    case ApdlType.Float => "float"
    case ApdlType.Double => "double"
    case ApdlType.Long => "long"
    case ApdlType.Byte => "byte"
    case ApdlType.Short => "short"
    case ApdlType.Bool => "bool"
    case ApdlType.Char => "char"
  }
}

object ApdlType {
  // any string : _AS)D SA)D,...
  case object Str extends ApdlType
  // any valid identifier : _id, asAdASDsa, id_ad_ASDS_ad
  case object Id extends ApdlType
  // standard types
  case object Int extends ApdlType
  case object Float extends ApdlType
  case object Long extends ApdlType
  case object Bool extends ApdlType
  case object Double extends ApdlType
  case object Short extends ApdlType
  case object Char extends ApdlType
  case object Byte extends ApdlType

  def values: Seq[ApdlType] = Seq(Str, Id, Int, Float, Long, Bool, Double, Short, Char, Byte)
  def stdValues: Seq[ApdlType] = Seq(Int, Float, Long, Bool, Double, Short, Char, Byte)
}

object DefineUtils {
  implicit class Defines(defines: List[ApdlDefine]) {
    def defineFromString(stringIdentifier: String): ApdlDefine = {
      defines
        .find(_.identifier == stringIdentifier)
        .getOrElse(throw new ApdlCodeGenerationException(s"Unknow definition input :
         ↪  $stringIdentifier"))
```

```
    }
  }
}
```

TransformDslParser.scala

```
package apdl.parser

import apdl._

import scala.language.postfixOps
import scala.util.matching.Regex
import scala.util.parsing.combinator.{PackratParsers, RegexParsers}

/* Transformater script syntax */
trait TransformDslParser extends RegexParsers with PackratParsers {
  override protected val whiteSpace: Regex = "[ \t\r\f\n]+".r
  override def skipWhitespace: Boolean = true

  // Types
  lazy val tfRetType: PackratParser[TfRetTyp] = tfVoid | tfTyp
  lazy val tfTyp: PackratParser[TfTyp] = tfArrayTyp | tfPrimitivesTyp
  lazy val tfPrimitivesTyp: PackratParser[TfPrimitivesTyp] = tfBooleanTyp | tfNumericTyp
  lazy val tfBooleanTyp: PackratParser[TfBoolean.type] = "bool" ^^ { _ => TfBoolean }
  lazy val tfNumericTyp: PackratParser[TfNumericTyp] = tfIntegralTyp | tfFloatingPointTyp
  lazy val tfIntegralTyp: PackratParser[TfIntegralTyp] = tfInt | tfShort | tfLong | tfByte |
    ↪  tfChar
  lazy val tfInt: PackratParser[TfInt.type] = "int" ^^ { _ => TfInt }
  lazy val tfLong: PackratParser[TfLong.type] = "long" ^^ { _ => TfLong }
  lazy val tfByte: PackratParser[TfByte.type] = "byte" ^^ { _ => TfByte }
  lazy val tfShort: PackratParser[TfShort.type] = "short" ^^ { _ => TfShort }
  lazy val tfChar: PackratParser[TfChar.type] = "char" ^^ { _ => TfChar }
  lazy val tfFloatingPointTyp: PackratParser[TfFloatingPointTyp] = tfFloat | tfDouble
  lazy val tfFloat: PackratParser[TfFloat.type] = "float" ^^ { _ => TfFloat }
  lazy val tfDouble: PackratParser[TfDouble.type] = "double" ^^ { _ => TfDouble }
  lazy val tfArrayTyp: PackratParser[TfArray] = tfTyp <~ "[" ~ "]" ^^ { typ => TfArray(typ) }
  lazy val tfVoid: PackratParser[TfVoid.type] = "void" ^^ { _ => TfVoid }

  // Expressions

  lazy val tfConstantExpr: PackratParser[Expr] = {
    tfLogicalOrExpr
  }

  lazy val tfLogicalOrExpr: PackratParser[Expr] = {
    tfLogicalOrExpr ~ ("||" ~> tfLogicalAndExpr) ^^ { case (l ~ r) => Or(l, r) } |
      tfLogicalAndExpr
  }

  lazy val tfLogicalAndExpr: PackratParser[Expr] = {
    tfLogicalAndExpr ~ ("&&" ~> tfEqualityExpr) ^^ { case (l ~ r) => And(l, r) } |
      tfEqualityExpr
  }

  lazy val tfEqualityExpr: PackratParser[Expr] = {
    tfEqualityExpr ~ ("==" ~> tfRelationalExpr) ^^ { case (l ~ r) => Equals(l, r) } |
      tfEqualityExpr ~ ("!=" ~> tfRelationalExpr) ^^ { case (l ~ r) => NotEquals(l, r) } |
      tfRelationalExpr
  }

  lazy val tfRelationalExpr: PackratParser[Expr] = {
```

```
  tfRelationalExpr ~ (">" ~> tfAdditiveExpr) ^^ { case (l ~ r) => Greater(l, r) } |
    tfRelationalExpr ~ ("<" ~> tfAdditiveExpr) ^^ { case (l ~ r) => Smaller(l, r) } |
    tfRelationalExpr ~ (">=" ~> tfAdditiveExpr) ^^ { case (l ~ r) => GreaterEquals(l, r) } |
    tfRelationalExpr ~ ("<=" ~> tfAdditiveExpr) ^^ { case (l ~ r) => SmallerEquals(l, r) } |
    tfAdditiveExpr
}

lazy val tfAdditiveExpr: PackratParser[Expr] = {
  tfAdditiveExpr ~ ("+" ~> tfMultiplicativeExpr) ^^ { case (l ~ r) => Add(l, r) } |
    tfAdditiveExpr ~ ("-" ~> tfMultiplicativeExpr) ^^ { case (l ~ r) => Sub(l, r) } |
    tfMultiplicativeExpr
}

lazy val tfMultiplicativeExpr: PackratParser[Expr] = {
  tfMultiplicativeExpr ~ ("*" ~> tfCastExpr) ^^ { case (l ~ r) => Mul(l, r) } |
    tfMultiplicativeExpr ~ ("/" ~> tfCastExpr) ^^ { case (l ~ r) => Div(l, r) } |
    tfCastExpr
}

lazy val tfCastExpr: PackratParser[Expr] = {
  (lp ~> tfPrimitivesTyp <~ rp) ~ tfCastExpr ^^ { case (t ~ expr) => Cast(t, expr) } |
    tfNotExpr
}

lazy val tfNotExpr: PackratParser[Expr] = {
  "!" ~> tfNotExpr ^^ { e => Not(e) } |
    tfPostfixExpr
}

lazy val tfPostfixExpr: PackratParser[Expr] = tfFunctionCall | tfArrayAccess |
 ↪  tfPrimaryExpr

lazy val tfPrimaryExpr: PackratParser[Expr] = {
  tfAtom | tfSymbol | tfLiteral | lp ~> tfExpr <~ rp
}

lazy val tfArrayAccess: PackratParser[Expr] = {
  identifier ~ rep1("[" ~> tfExpr <~ "]") ^^ { case (id ~ expr) =>
    expr.tail.foldLeft(ArrayAccess(Symbol(id), expr.head))((acc, elt) => ArrayAccess(acc,
     ↪  elt))
  }
}

lazy val tfAtom: PackratParser[Expr] = {
  "true" ^^ { _ => True() } |
    "false" ^^ { _ => False() }
}

lazy val tfExpr: PackratParser[Expr] = {
  tfAssignExpr
}

lazy val tfAssignExpr: PackratParser[Expr] = {
  tfPostfixExpr ~ ("=" ~> tfAssignExpr) ^^ { case (l ~ r) => VarAssignement(l, r) } |
    tfLogicalOrExpr
}

lazy val tfSymbol: PackratParser[Symbol] = {
  identifier ^^ { x => Symbol(x) }
}

lazy val tfLiteral: PackratParser[Literal] = {
  """[+-]?([0-9]+([.][0-9]*)?|[.][0-9]+)(E[0-9]+)?""".r ^^ Literal
}

lazy val identifier: PackratParser[String] = "[a-zA-Z_][a-zA-Z0-9_]*".r ^^ { str => str }
```

```
lazy val tfFunctionCall: PackratParser[FunctionCall] =
  identifier ~ lp ~ tfFunctionCallArg ~ rp ^^ {
    case (id ~ _ ~ args ~ _) => FunctionCall(id, args)
  }

lazy val tfFunctionCallArg: PackratParser[List[Expr]] = {
  (tfExpr ?) ~ rep("," ~> tfExpr) ^^ { case (a ~ as) =>
    a match {
      case Some(value) => value :: as
      case None => List()
    }
  }
}

lazy val tfArgs: PackratParser[List[TypedIdentifier]] = tfArg ~ rep("," ~> tfArg) ^^ {
  case (a ~ as) => a :: as
}

lazy val tfArg: PackratParser[TypedIdentifier] = {
  identifier ~ ":" ~ tfTyp ^^ { case (id ~ _ ~ typ) => TypedIdentifier(id, typ) }
}

lazy val tfVarAssign: PackratParser[VarAssignement] = {
  identifier ~ "=" ~ tfConstantExpr ^^ { case (id ~ _ ~ expr) => VarAssignement(Symbol(id),
    ↪  expr) }
}

lazy val tfBlock: PackratParser[Block] = lb ~> tfStatements <~ rb ^^ { statements =>
 ↪  Block(statements) }

lazy val tfStatements: PackratParser[List[Statement]] = rep(tfStatement)

lazy val tfStatement: PackratParser[Statement] = {
  tfBlock | tfSelectionStatement | tfLoop | tfJump | tfDeclaration | tfExprStatement
}

lazy val tfExprStatement: PackratParser[ExpressionStatement] = tfExpr ^^ { expr =>
 ↪  ExpressionStatement(expr) }

lazy val tfLoop: PackratParser[Statement] = tfWhile | tfDoWhile
lazy val tfWhile: PackratParser[While] = "while" ~ lp ~ tfExpr ~ rp ~ tfStatement ^^ {
  case (_ ~ _ ~ cond ~ _ ~ statement) => While(cond, statement)
}
lazy val tfDoWhile: PackratParser[DoWhile] = "do" ~ tfStatement ~ "while" ~ lp ~ tfExpr ~
 ↪  rp ^^ {
  case (_ ~ statement ~ _ ~ _ ~ cond ~ _) => DoWhile(cond, statement)
}

lazy val tfDeclaration: PackratParser[Declaration] = {
  tfNewVal | tfNewArray | tfNewVar | tfFunctionDeclaration
}

lazy val tfFunctionDeclaration: PackratParser[FunctionDecl] = {
  "def" ~> tfFunctionHeader ~ tfFunctionBody ^^ { case (h ~ b) => FunctionDecl(h, b) }
}

lazy val tfFunctionHeader: Parser[FunctionHeader] =
  identifier ~ lp ~ tfArgs ~ rp ~ "->" ~ tfRetType ^^ {
    case (id ~ _ ~ parameters ~ _ ~ "->" ~ ret_type) => FunctionHeader(ret_type, id,
      ↪  parameters)
  } |
    identifier ~ (lp ~ rp ~ "->" ~> tfRetType) ^^ { case (id ~ typ) => FunctionHeader(typ,
      ↪  id, List()) }
lazy val tfFunctionBody: PackratParser[FunctionBody] = tfBlock ^^ { b => FunctionBody(b) }
```

```
  lazy val tfNewArray: PackratParser[NewArray] = {
    "var" ~ tfSymbol ~ ":" ~ tfArrayTyp ~ "=" ~ tfArrayInit ^^ {
      case (_ ~ id ~ _ ~ typ ~ _ ~ init) => NewArray(id, typ, init)
    } |
      "var" ~ tfSymbol ~ ":" ~ tfArrayTyp ^^ { _ => throw new
       ↪  ApdlParserException("Uninitialised array") }
  }

  lazy val tfNewVar: PackratParser[NewVar] = {
    "var" ~ tfSymbol ~ ":" ~ tfPrimitivesTyp ~ "=" ~ tfExpr ^^ {
      case (_ ~ id ~ _ ~ typ ~ _ ~ init) => NewVar(id, typ, Some(init))
    } |
      "var" ~ tfSymbol ~ ":" ~ tfPrimitivesTyp ^^ {
        case (_ ~ id ~ _ ~ typ) => NewVar(id, typ, None)
      }
  }

  lazy val tfArrayInit: PackratParser[ArrayInit] = {
    "[" ~> tfLiteral <~ "]" ^^ ArrayInitCapacity |
      "{" ~> tfExpr ~ rep("," ~> tfExpr) <~ "}" ^^ { case (e ~ es) =>
        ArrayInitValue(e :: es)
      }
  }

  lazy val tfNewVal: PackratParser[NewVal] = {
    "val" ~ tfSymbol ~ ":" ~ tfPrimitivesTyp ~ "=" ~ tfExpr ^^ {
      case (_ ~ id ~ _ ~ typ ~ _ ~ init) => NewVal(id, typ, init)
    }
  }

  lazy val tfSelectionStatement: PackratParser[Statement] = tfIfThenElse | tfIfThen
  lazy val tfIfThenElse: PackratParser[IfThenElse] = "if" ~ lp ~ tfExpr ~ rp ~ tfStatement ~
   ↪  "else" ~ tfStatement ^^ {
    case (_ ~ _ ~ cond ~ _ ~ trueBranch ~ _ ~ falseBranch) => IfThenElse(cond, trueBranch,
     ↪  falseBranch)
  }
  lazy val tfIfThen: PackratParser[IfThen] = "if" ~ lp ~ tfExpr ~ rp ~ tfStatement ^^ {
    case (_ ~ _ ~ cond ~ _ ~ statement) => IfThen(cond, statement)
  }

  lazy val tfJump: PackratParser[Statement] = {
    tfBreak | tfContinue | tfReturn
  }

  lazy val tfReturn: PackratParser[Return] = "return" ~> tfExpr ^^ { expr => Return(expr) }
  lazy val tfBreak: PackratParser[Break] = "break" ^^ { _ => Break() }
  lazy val tfContinue: PackratParser[Continue] = "continue" ^^ { _ => Continue() }

  lazy val lp = "("
  lazy val rp = ")"
  lazy val lb = "{"
  lazy val rb = "}"
}
```

TransformDslAst.scala

```
package apdl.parser

import apdl.ApdlProjectException

// TODO : refactoring
```

```
sealed trait ApdlAst
sealed trait Expr extends ApdlAst

case class Add(left: Expr, right: Expr) extends Expr
case class Mul(left: Expr, right: Expr) extends Expr
case class Sub(left: Expr, right: Expr) extends Expr
case class Div(left: Expr, right: Expr) extends Expr
case class Cast(tfTyp: TfPrimitivesTyp, expr: Expr) extends Expr
case class Literal(value: String) extends Expr
case class Symbol(name: String) extends Expr
case class FunctionCall(funcName: String, args: List[Expr]) extends Expr
case class ArrayAccess(array: Expr, field: Expr) extends Expr
case class VarAssignement(target: Expr, value: Expr) extends Expr
// Bool
case class True() extends Expr
case class False() extends Expr
case class Or(left: Expr, right: Expr) extends Expr
case class And(left: Expr, right: Expr) extends Expr
case class Not(expr: Expr) extends Expr
case class Greater(left: Expr, right: Expr) extends Expr
case class Smaller(left: Expr, right: Expr) extends Expr
case class GreaterEquals(left: Expr, right: Expr) extends Expr
case class SmallerEquals(left: Expr, right: Expr) extends Expr
case class Equals(left: Expr, right: Expr) extends Expr
case class NotEquals(left: Expr, right: Expr) extends Expr

sealed trait TfRetTyp extends ApdlAst {
  def asApdlType: ApdlType = this match {
    case TfBoolean => ApdlType.Bool
    case TfInt => ApdlType.Int
    case TfLong => ApdlType.Long
    case TfByte => ApdlType.Byte
    case TfShort => ApdlType.Short
    case TfChar => ApdlType.Char
    case TfDouble => ApdlType.Double
    case TfFloat => ApdlType.Float
    case t => throw new ApdlProjectException(s"$t can't be convert to an ApdlType")
  }
}

sealed trait TfTyp extends TfRetTyp
sealed trait TfPrimitivesTyp extends TfTyp
sealed trait TfNumericTyp extends TfPrimitivesTyp
sealed trait TfIntegralTyp extends TfNumericTyp
sealed trait TfFloatingPointTyp extends TfNumericTyp
case object TfBoolean extends TfPrimitivesTyp
case object TfInt extends TfIntegralTyp
case object TfLong extends TfIntegralTyp
case object TfByte extends TfIntegralTyp
case object TfShort extends TfIntegralTyp
case object TfChar extends TfIntegralTyp
case object TfDouble extends TfFloatingPointTyp
case object TfFloat extends TfFloatingPointTyp
case object TfVoid extends TfRetTyp
case class TfArray(typ: TfTyp) extends TfTyp

case class TypedIdentifier(name: String, typ: TfTyp) extends ApdlAst

sealed trait Statement extends ApdlAst
case class While(cond: Expr, statement: Statement) extends Statement
case class DoWhile(cond: Expr, statement: Statement) extends Statement
case class IfThenElse(cond: Expr, trueBranch: Statement, falseBranch: Statement) extends
  ↪  Statement
case class IfThen(cond: Expr, ifTrue: Statement) extends Statement
case class Return(expr: Expr) extends Statement
```

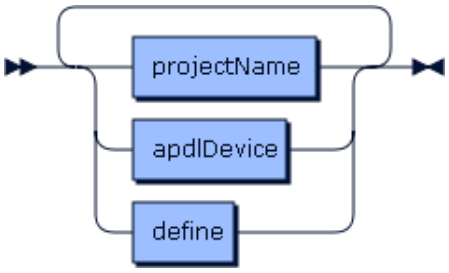
```
case class Break() extends Statement
case class Continue() extends Statement
case class Block(statements: List[Statement]) extends Statement
case class ExpressionStatement(expression: Expr) extends Statement

sealed trait Declaration extends Statement
case class FunctionDecl(header: FunctionHeader, body: FunctionBody) extends Declaration
case class FunctionHeader(resultType: TfRetTyp, identifier: String, parameters:
  ↪  List[TypedIdentifier])
case class FunctionBody(body: Block)

case class NewVal(symbol: Symbol, typ: TfTyp, init: Expr) extends Declaration
case class NewVar(symbol: Symbol, typ: TfTyp, init: Option[Expr]) extends Declaration
case class NewArray(symbol: Symbol, typ: TfArray, init: ArrayInit) extends Declaration

sealed trait ArrayInit extends ApdlAst
case class ArrayInitValue(values: List[Expr]) extends ArrayInit
case class ArrayInitCapacity(capacity: Literal) extends ArrayInit

// Companion object

object Add {
  def apply(l: AnyVal, r: AnyVal): Add = new Add(Literal(l.toString), Literal(r.toString))

  def apply(l: String, r: AnyVal): Add = new Add(Symbol(l), Literal(r.toString))

  def apply(l: AnyVal, r: String): Add = new Add(Literal(l.toString), Symbol(r))

  def apply(l: Expr, r: AnyVal): Add = new Add(l, Literal(r.toString))

  def apply(l: AnyVal, r: Expr): Add = new Add(Literal(l.toString), r)

  def apply(l: Expr, r: String): Add = new Add(l, Symbol(r))

  def apply(l: String, r: Expr): Add = new Add(Symbol(l), r)

  def apply(l: String, r: String): Add = new Add(Symbol(l), Symbol(r))
}

object Mul {
  def apply(l: AnyVal, r: AnyVal): Mul = new Mul(Literal(l.toString), Literal(r.toString))

  def apply(l: String, r: AnyVal): Mul = new Mul(Symbol(l), Literal(r.toString))

  def apply(l: AnyVal, r: String): Mul = new Mul(Literal(l.toString), Symbol(r))

  def apply(l: Expr, r: AnyVal): Mul = new Mul(l, Literal(r.toString))

  def apply(l: AnyVal, r: Expr): Mul = new Mul(Literal(l.toString), r)

  def apply(l: Expr, r: String): Mul = new Mul(l, Symbol(r))

  def apply(l: String, r: Expr): Mul = new Mul(Symbol(l), r)

  def apply(l: String, r: String): Mul = new Mul(Symbol(l), Symbol(r))
}

object Sub {
  def apply(l: AnyVal, r: AnyVal): Sub = new Sub(Literal(l.toString), Literal(r.toString))

  def apply(l: String, r: AnyVal): Sub = new Sub(Symbol(l), Literal(r.toString))

  def apply(l: AnyVal, r: String): Sub = new Sub(Literal(l.toString), Symbol(r))

  def apply(l: Expr, r: AnyVal): Sub = new Sub(l, Literal(r.toString))
```

```
  def apply(l: AnyVal, r: Expr): Sub = new Sub(Literal(l.toString), r)

  def apply(l: Expr, r: String): Sub = new Sub(l, Symbol(r))

  def apply(l: String, r: Expr): Sub = new Sub(Symbol(l), r)

  def apply(l: String, r: String): Sub = new Sub(Symbol(l), Symbol(r))
}

object Div {
  def apply(l: AnyVal, r: AnyVal): Div = new Div(Literal(l.toString), Literal(r.toString))

  def apply(l: String, r: AnyVal): Div = new Div(Symbol(l), Literal(r.toString))

  def apply(l: AnyVal, r: String): Div = new Div(Literal(l.toString), Symbol(r))

  def apply(l: Expr, r: AnyVal): Div = new Div(l, Literal(r.toString))

  def apply(l: AnyVal, r: Expr): Div = new Div(Literal(l.toString), r)

  def apply(l: Expr, r: String): Div = new Div(l, Symbol(r))

  def apply(l: String, r: Expr): Div = new Div(Symbol(l), r)

  def apply(l: String, r: String): Div = new Div(Symbol(l), Symbol(r))
}
```

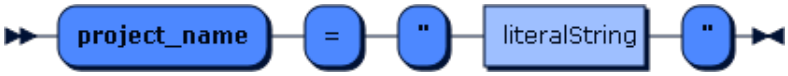
Appendix C

APDL EBNF Diagrams

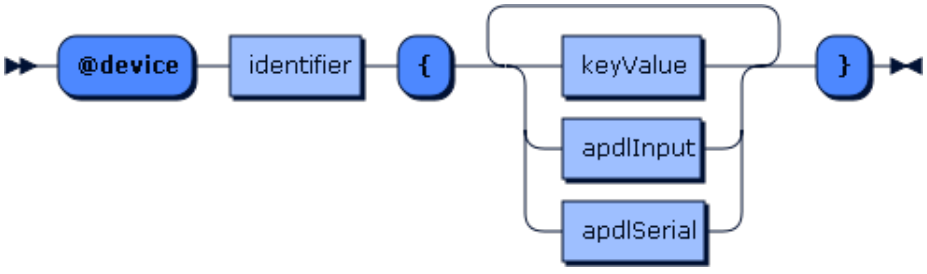
program



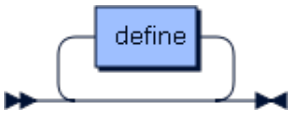
projectName



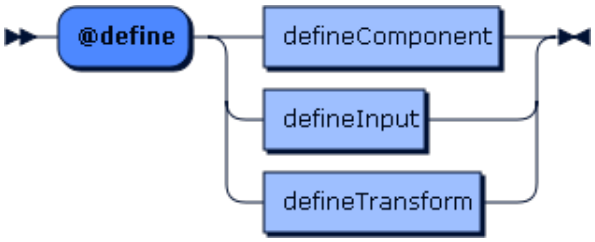
apdlDevice



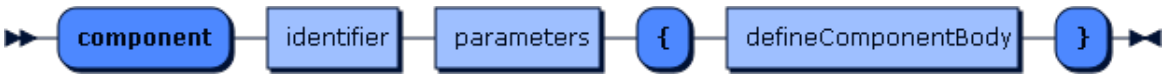
defines



define



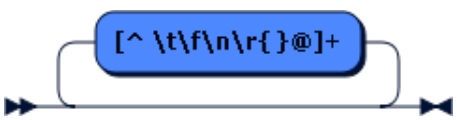
defineComponent



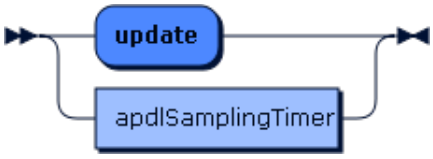
apdlInput

apdlSerial

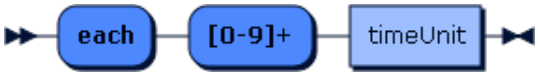
apdlParameters



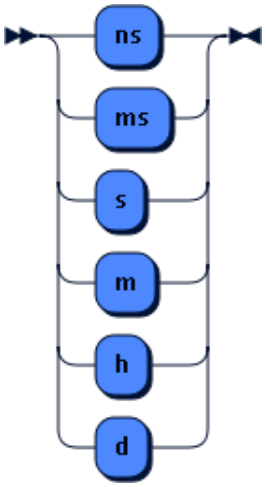
apdlSampling



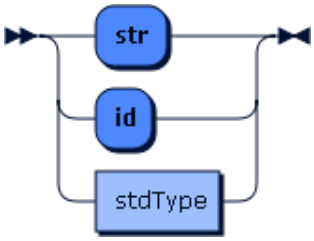
apdlSamplingTimer



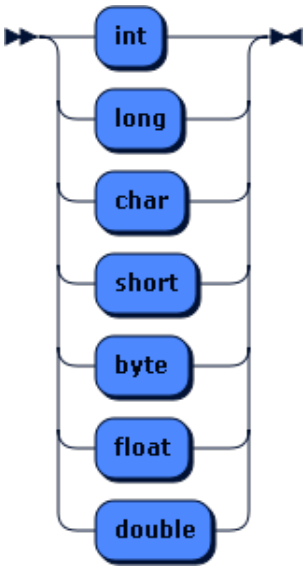
timeUnit



type



stdType

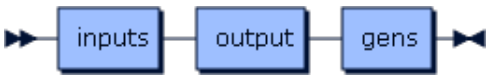


arg

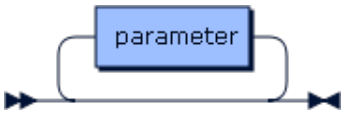
defineInput

defineTransform

defineComponentBody



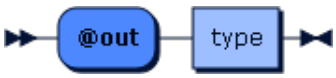
parameters



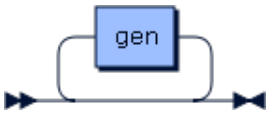
parameter

inputs

output



gens



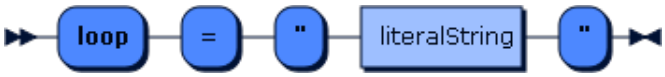
gen

genBody

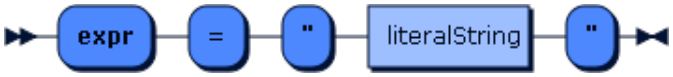
global

setup

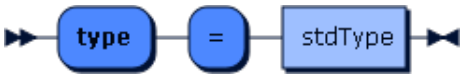
loop



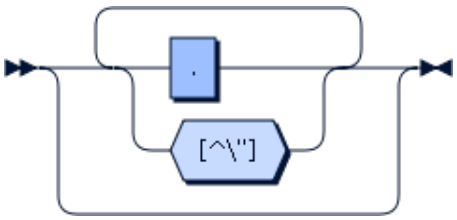
expr



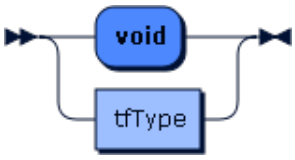
genType



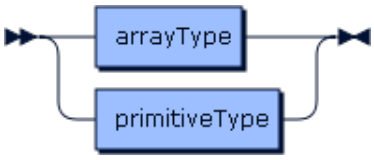
literalString



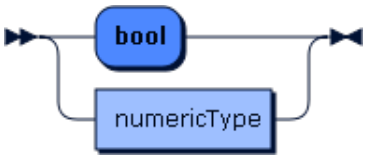
returnType



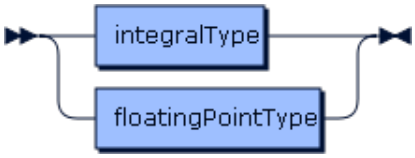
tfType



primitiveType

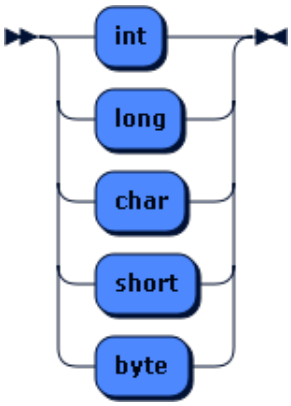


numericType

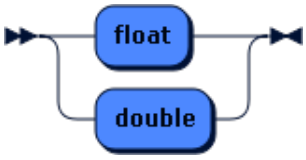


arrayType

integralType

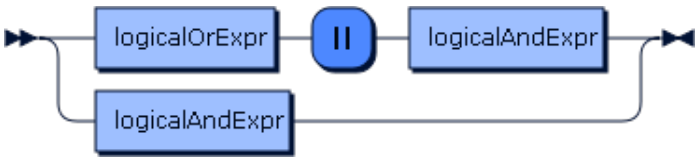


floatingPointType

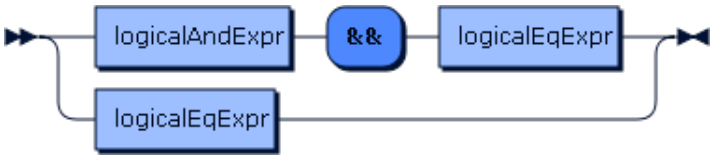


constantExpr

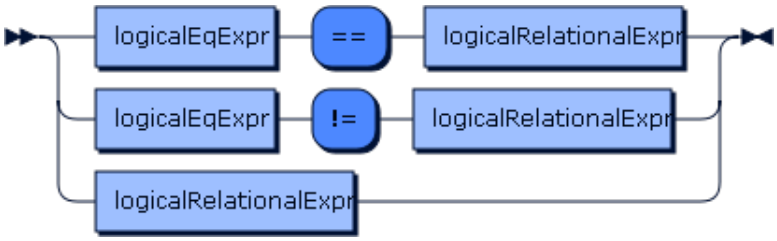
logicalOrExpr



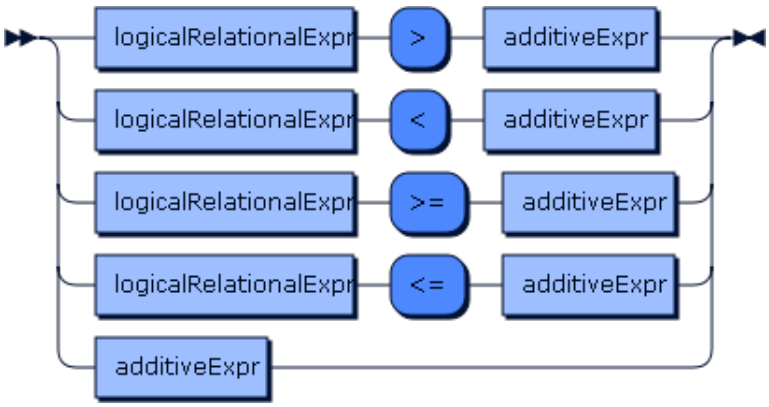
logicalAndExpr



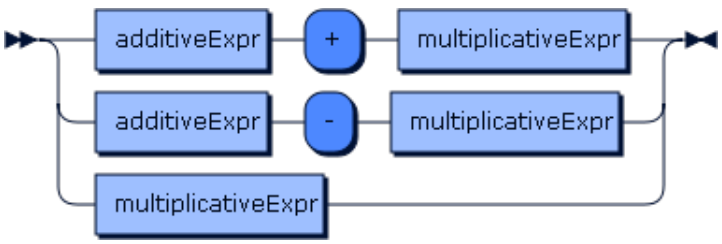
logicalEqExpr



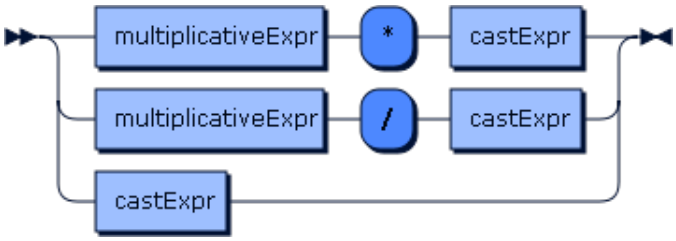
logicalRelationalExpr



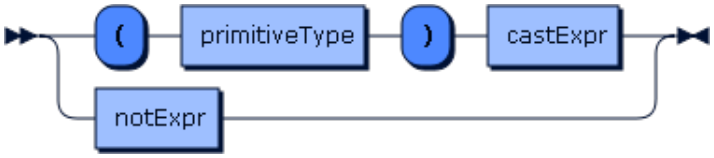
additiveExpr



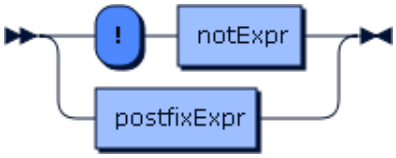
multiplicativeExpr



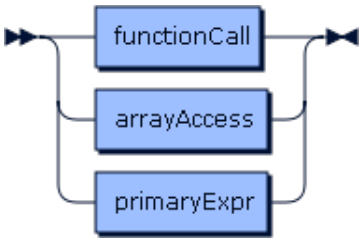
castExpr



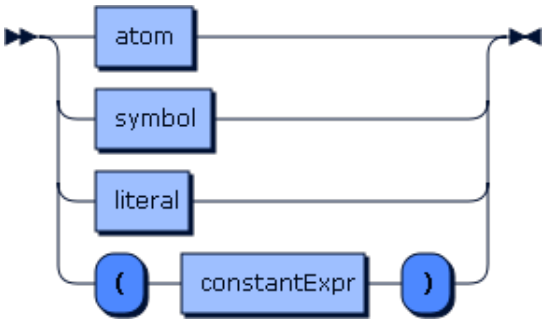
notExpr



postfixExpr

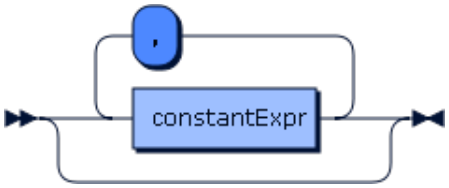


primaryExpr

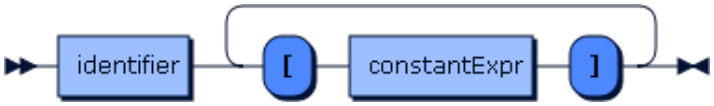


functionCall

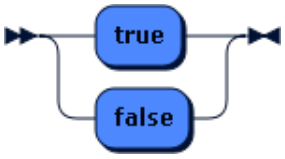
functionArgs



arrayAccess

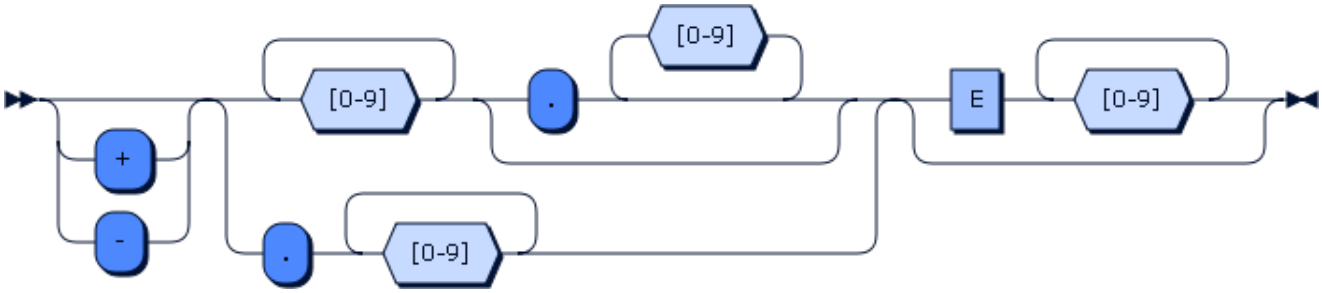


atom

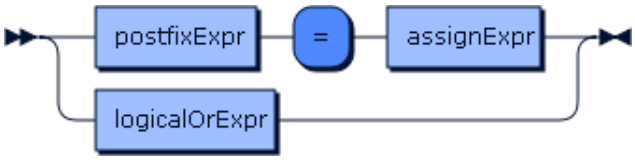


symbol

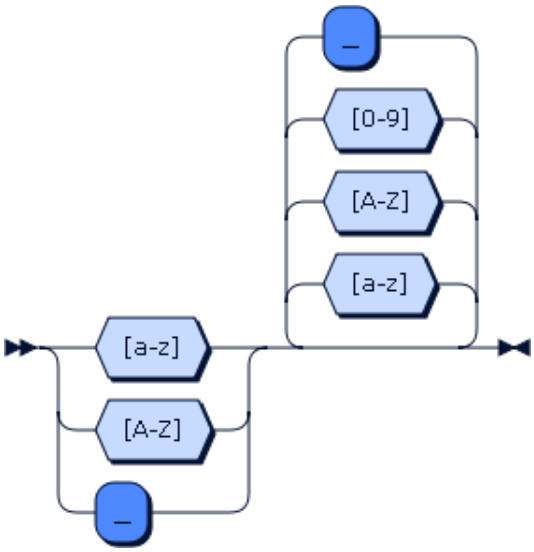
literal



assignExpr

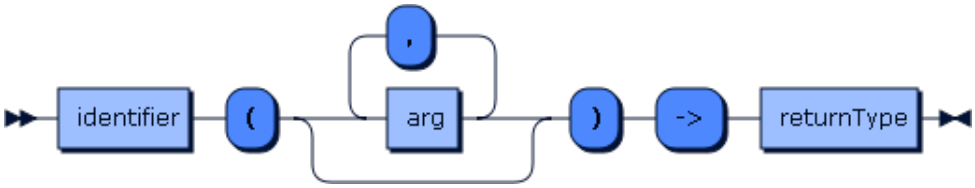


identifier



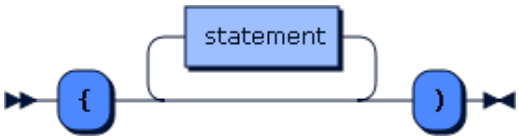
functionDeclaration

functionHeader

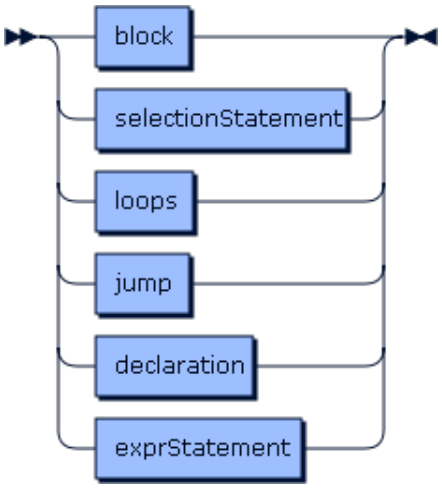


functionBody

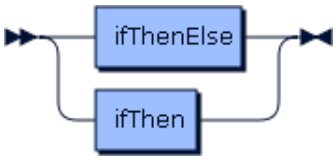
block



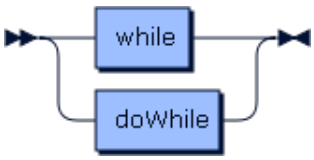
statement



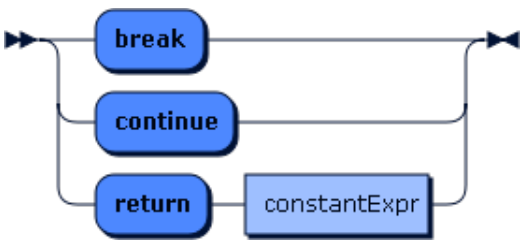
selectionStatement



loops

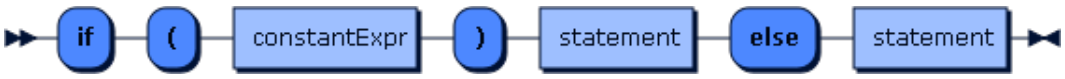


jump



exprStatement

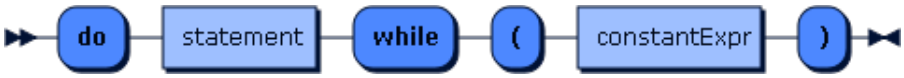
ifThenElse



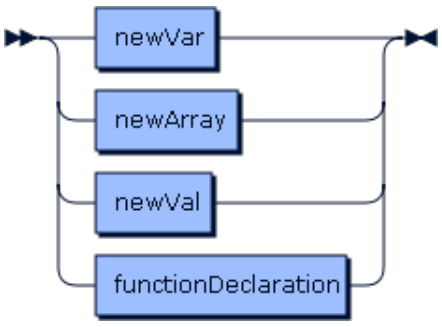
ifThen

while

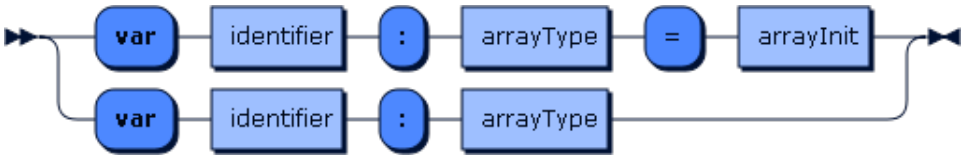
doWhile



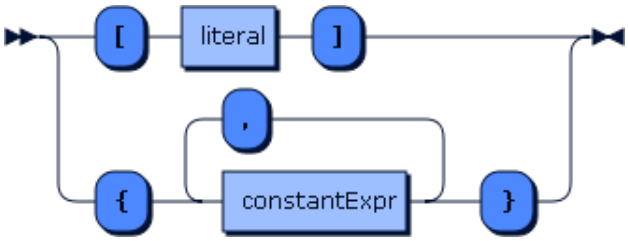
declaration



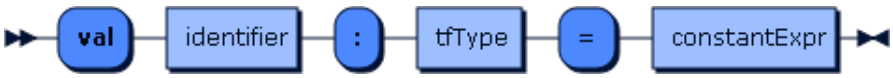
newArray



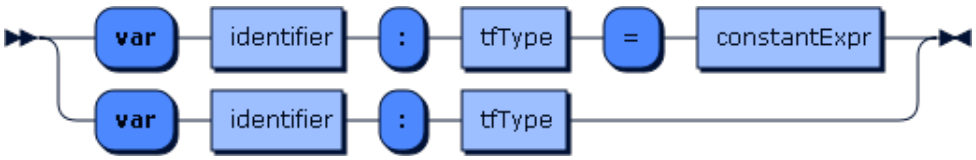
arrayInit



newVal



newVar



Appendix D

APDL Recursive Inclusion Algorithm Implementation in Scala

```
def generateInputs(out: ApdlPrintWriter): Unit = {
  // Generate default input
  debug(s"\tGenerate default inputs")
  generateDefaultInputs(out)
  // Make the symbolTable for non-default input
  debug(s"\tGenerate non default inputs")
  generateNonDefaultInputs(out)
}

def generateNonDefaultInputs(out: ApdlPrintWriter): Unit = {
  val inputs = device.inputs
  val nonDefaultInputs = inputs.filter(isNonDefault)

  // For each inputs, take the ones who are generable
  def process(inputs: List[ApdlInput]): Unit = if (inputs.nonEmpty) {
    val (generableInputs, nonGenerableInputs) = inputs.partition(isGenerable)
    generableInputs.foreach { i =>
      if (isTransform(i)) {
        val sourceInput = symbolTable.get(i.args.head) match {
          case default: InputDefault => default
          case transformed: InputTransformed => transformed
          case componented: InputComponented => componented
          case _ => throw new ApdlProjectException(s"Can't find source input for input
            ↪  ${i.identifier}")
        }
        val definition = defines.find(_.identifier == i.defineInputIdentifier) match {
          case Some(value) => value
          case None => throw new ApdlProjectException(s"Unknow define for input
            ↪  ${i.identifier}")
        }
        assume(definition.isInstanceOf[ApdlDefineTransform])

        // code generation
        val functionDecl = definition.asInstanceOf[ApdlDefineTransform].functionDecl

        if (!symbolTable.contains(functionDecl.header.identifier)) {
          // A transform is a function
          out.println\nFunction {
            s"""
                | // Transform ${functionDecl.header.identifier}
                | ${transformCodeGen(functionDecl)}
                | // End transform ${functionDecl.header.identifier}
```

```
            """.stripMargin
          }
          // Add the transform into the symbol table
          symbolTable.add(functionDecl.header.identifier, Transform(functionDecl))
        }
        symbolTable.add(i.identifier, InputTransformed(i.identifier,
          ↪  definition.asInstanceOf[ApdlDefineTransform], sourceInput))

      } else if (isComponented(i)) {

        val definition = defines.find(_.identifier == i.defineInputIdentifier) match {
          case Some(value) => value
          case None => throw new ApdlProjectException(s"Unknow define for input
            ↪  ${i.identifier}")
        }

        assume(definition.isInstanceOf[ApdlDefineComponent])

        val component = definition.asInstanceOf[ApdlDefineComponent]

        val nbNonInputs = component.parameters.length
        val args = i.args.drop(nbNonInputs)
        val sourceInputs: List[Input] = symbolTable.gets(args).map {
          case default: InputDefault => default
          case transformed: InputTransformed => transformed
          case componented: InputComponented => componented
          case _ => throw new ApdlProjectException(s"Can't find source input for input
            ↪  ${i.identifier}")
        }


        symbolTable.add(i.identifier, InputComponented(i.identifier,
          ↪  definition.asInstanceOf[ApdlDefineComponent], sourceInputs))

      } else {
        // something is wrong...
        throw new ApdlProjectException(s"Input ${i.identifier} from device ${device.name}
          ↪  encountered some problems")
      }
    }
    process(nonGenerableInputs)
  }

  process(nonDefaultInputs)
}

def isTransform(input: ApdlInput): Boolean = defines.find {
  case ApdlDefineTransform(functionDecl) => functionDecl.header.identifier ==
    ↪  input.defineInputIdentifier
  case _ => false
} match {
  case Some(value) =>
    assert(value.isInstanceOf[ApdlDefineTransform])
    assert(input.args.length == 1)
    true
  case None => false
}

def isComponented(input: ApdlInput): Boolean = defines.find {
  case component: ApdlDefineComponent => component.identifier == input.defineInputIdentifier
  case _ => false
} match {
  case Some(value) =>
    assert(value.isInstanceOf[ApdlDefineComponent])
    true
  case None => false
```

```
}

def isGenerable(input: ApdlInput): Boolean = {
  if (isTransform(input)) {
    assert(input.args.length == 1)
    val sourceInput = input.args.head
    symbolTable.contains(sourceInput)
  }
  else if (isComponented(input)) {
    val componentDefine = defines.find(_.identifier == input.defineInputIdentifier) match {
      case Some(value) => value match {
        case component: ApdlDefineComponent => component
        case _ => throw new ApdlProjectException(s"Unexpected definition found for input
          ↪  ${input.identifier} from device ${device.name}")
      }
      case None => throw new ApdlProjectException(s"Unknow define component for input
        ↪  ${input.identifier} from device ${device.name}")
    }
    val nbNonInputs = componentDefine.parameters.length
    val args = input.args.drop(nbNonInputs)
    args.forall(symbolTable.contains)
  }
  else {
    throw new ApdlProjectException(s"Wrong input type for input ${input.identifier} from
      ↪  device ${device.name}")
  }
}

def isNonDefault(input: ApdlInput): Boolean = {
  !isDefault(input)
}

def isDefault(input: ApdlInput): Boolean = {
  defines.find(_.identifier == input.defineInputIdentifier) match {
    case Some(definition) => definition match {
      case _: ApdlDefineInput => true
      case _: ApdlDefineComponent => false
      case _: ApdlDefineTransform => false
    }
    case None =>
      throw new ApdlProjectException(s"Unknow type for input ${input.identifier}")
  }
}

// Generate the default input inside the symbol table
def generateDefaultInputs(out: ApdlPrintWriter): Unit = device.inputs.foreach { input =>
  defines.find(_.identifier == input.defineInputIdentifier) match {
    case Some(definition) => definition match {
      case ApdlDefineInput(name, parameters, gens) =>
        // A default input

        val gen = gens.getOrElse(framework.identifier, throw new
          ↪  ApdlProjectException(s"Unknow framework $framework for input definition :
          ↪  $name"))

        implicit val args = zipArgWithIdentifier(input.args, parameters, List()) + ("id" ->
          ↪  IdGenerator.nextVariable(input.identifier))

        out.println1Global(gen.global.replaceWithArgs)
        out.println1Setup(gen.setup.replaceWithArgs)
        out.println1Loop(gen.loop.replaceWithArgs)

        symbolTable.add(input.identifier, InputDefault(
          input.identifier,
          definition.asInstanceOf[ApdlDefineInput],
          input.args,
```

```
            gen.expr.replaceWithArgs))
        case _ =>
      }
      case None => throw new ApdlProjectException(s"Unknow type for input ${input.identifier}")
    }
  }
```

Appendix E

Generators Implementation for Property-based Testing

GeneratorUtils.scala

```
import apdl.parser._
import org.scalacheck.Gen

object StringGenerators {
  def typGen: Gen[String] = Gen.oneOf(ApdlType.values).map(_.toString)

  def stdTypGen: Gen[String] = Gen.oneOf(ApdlType.stdValues).map(_.toString)

  def parameterGen: Gen[String] = for {
    id <- Gen.identifier
    typ <- typGen
  } yield s"$id : $typ"

  def idGen: Gen[String] = Gen.identifier

  def genGen: Gen[String] = for {
    id <- Gen.identifier
    g <- Gen.alphaNumStr
    s <- Gen.alphaNumStr
    l <- Gen.alphaNumStr
    e <- Gen.alphaNumStr
    typ <- stdTypGen
    t <- Gen.oneOf(None, Some(typ))
  } yield
    s"""
       |@gen $id {
       |  global = "$g"
       |  setup = "$s"
       |  loop = "$l"
       |  expr = "$e"
       |  ${
      t match {
        case Some(value) => s"type = $value"
        case None => s""
      }
    }
       |}
    """.stripMargin
```

```
  def inGen: Gen[String] = for {
    params <- Gen.listOf(parameterGen) suchThat (_.nonEmpty)
  } yield s"@in ${params.mkString(" ")}"

  def outGen: Gen[String] = typGen.map(t => s"@out $t")

  def defineComponentGen: Gen[String] = (for {
    id <- Gen.identifier
    params <- Gen.listOf(parameterGen)
    in <- inGen
    out <- outGen
    gens <- Gen.listOf(genGen)
  } yield
    s"""
       |@define component $id ${params mkString " "} {
       |  $in
       |  $out
       |  ${gens mkString "\n"}
       |}
    """.stripMargin).label("Define component generator")

  def defineInputGen: Gen[String] = (for {
    id <- Gen.identifier
    params <- Gen.listOf(parameterGen)
    gens <- Gen.listOf(genGen)
  } yield
    s"""
       |@define input $id ${params mkString ""} {
       | ${gens mkString "\n"}
       |}
     """.stripMargin).label("Define input generator")
}

class ApdlBaseGenerators(maxIdentifierSize: Int = 20) {

  def genPrimitivesTyp: Gen[TfPrimitivesTyp] = Gen.oneOf(
    genTfBoolean, genTfInt, genTfLong, genTfByte,
    genTfShort, genTfChar, genTfDouble, genTfFloat
  )

  def genLiteral: Gen[Literal] = for {
    num <- Gen.choose(0, Double.MaxValue)
  } yield Literal(num.toString)

  def genSymbol: Gen[Symbol] = for {
    id <- genIdentifier
  } yield Symbol(id)

  def genIdentifier: Gen[String] = for {
    l <- Gen.choose(1, maxIdentifierSize)
    c <- Gen.alphaLowerChar
    cs <- Gen.listOfN(l, Gen.alphaNumChar)
  } yield (c :: cs).mkString

  def typGen: Gen[ApdlType] = Gen.oneOf(ApdlType.values)

  def stdTypGen: Gen[ApdlType] = Gen.oneOf(ApdlType.stdValues)

  def genParameter: Gen[Parameter] = for {
    id <- genIdentifier
    typ <- typGen
  } yield Parameter(id, typ)

  def genTypedIdentifier: Gen[TypedIdentifier] = for {
    id <- Gen.identifier
```

```
    typ <- genTyp
  } yield TypedIdentifier(id, typ)

  def genTyp: Gen[TfTyp] = Gen.lzy(
    Gen.oneOf(
      genTfBoolean, genTfInt, genTfLong, genTfByte,
      genTfShort, genTfChar, genTfDouble, genTfFloat, genTfArray
    )
  )

  def genRetTyp: Gen[TfRetTyp] = Gen.lzy(
    Gen.oneOf(
      genTfBoolean, genTfInt, genTfLong, genTfByte, genTfVoid,
      genTfShort, genTfChar, genTfDouble, genTfFloat, genTfArray
    )
  )

  def genTfBoolean: Gen[TfBoolean.type] = TfBoolean

  def genTfInt: Gen[TfInt.type] = TfInt

  def genTfLong: Gen[TfLong.type] = TfLong

  def genTfByte: Gen[TfByte.type] = TfByte

  def genTfShort: Gen[TfShort.type] = TfShort

  def genTfChar: Gen[TfChar.type] = TfChar

  def genTfDouble: Gen[TfDouble.type] = TfDouble

  def genTfFloat: Gen[TfFloat.type] = TfFloat

  def genTfVoid: Gen[TfVoid.type] = TfVoid

  def genTfArray: Gen[TfArray] = for {
    typ <- genTyp
  } yield TfArray(typ)

  def isValid(l: List[Statement]): Boolean = {
    if (l.length < 2) true
    else {
      val l1 = l.reverse.tail.reverse // drop last
      val l2 = l.tail // drop first
      val crtWithNext = l1 zip l2
      crtWithNext.forall {
        case (Return(_), ExpressionStatement(_)) => false
        case (ExpressionStatement(_), ExpressionStatement(Cast(_, _))) => false
        case _ => true
      }
    }
  }
}

class ApdlExprGenerators(maxExprSize: Int = 4) extends ApdlBaseGenerators {

  def genExpr: Gen[Expr] = genExprInner(maxExprSize)

  private def genExprInner(depth: Int): Gen[Expr] = {
    if (depth == 0) genExprTerminal
    else {
      val nextDepth = depth - 1
      Gen.oneOf(
        genAdd(nextDepth),
        genMul(nextDepth),
        genDiv(nextDepth),
```

```
        genSub(nextDepth),
        genCast(nextDepth),
        genOr(nextDepth),
        genAdd(nextDepth),
        genNot(nextDepth),
        genSmaller(nextDepth),
        genSmallerEquals(nextDepth),
        genEquals(nextDepth),
        genNotEquals(nextDepth),
        genGreater(nextDepth),
        genGreaterEquals(nextDepth),
        genFunctionCall(nextDepth),
        genSymbol,
        genLiteral,
        genTrue,
        genFalse
      )
    }
  }

  private def genExprTerminal: Gen[Expr] = Gen.oneOf(
    genLiteral, genSymbol, genTrue, genFalse
  )

  def genAdd(depth: Int): Gen[Add] =
    for {
      e1 <- genExprInner(depth)
      e2 <- genExprInner(depth)
    } yield Add(e1, e2)

  def genMul(depth: Int): Gen[Mul] = Gen.lzy(for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield Mul(e1, e2))

  def genSub(depth: Int): Gen[Sub] = Gen.lzy(for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield Sub(e1, e2))

  def genDiv(depth: Int): Gen[Div] = Gen.lzy(for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield Div(e1, e2))

  def genCast(depth: Int): Gen[Cast] = for {
    typ <- genPrimitivesTyp
    expr <- genExprInner(depth)
  } yield Cast(typ, expr)

  def genFunctionCall(depth: Int): Gen[FunctionCall] = for {
    id <- genIdentifier
    args <- Gen.listOfN(maxExprSize, genExprInner(depth))
  } yield FunctionCall(id, args)

  def genArrayAccess(depth: Int): Gen[ArrayAccess] = for {
    array <- genExprInner(depth)
    field <- genExprInner(depth)
  } yield ArrayAccess(array, field)

  def genTrue: Gen[True] = True()

  def genFalse: Gen[False] = False()

  def genOr(depth: Int): Gen[Or] = for {
    e1 <- genExprInner(depth)
```

```
    e2 <- genExprInner(depth)
  } yield Or(e1, e2)

  def genAnd(depth: Int): Gen[And] = for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield And(e1, e2)

  def genNot(depth: Int): Gen[Not] = for {
    e1 <- genExprInner(depth)
  } yield Not(e1)

  def genGreater(depth: Int): Gen[Greater] = for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield Greater(e1, e2)

  def genSmaller(depth: Int): Gen[Smaller] = for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield Smaller(e1, e2)

  def genGreaterEquals(depth: Int): Gen[GreaterEquals] = for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield GreaterEquals(e1, e2)

  def genSmallerEquals(depth: Int): Gen[SmallerEquals] = for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield SmallerEquals(e1, e2)

  def genEquals(depth: Int): Gen[Equals] = for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield Equals(e1, e2)

  def genNotEquals(depth: Int): Gen[NotEquals] = for {
    e1 <- genExprInner(depth)
    e2 <- genExprInner(depth)
  } yield NotEquals(e1, e2)
}

class ApdlStatementGenerators(maxExprSize: Int = 4, maxBlockSize: Int = 10) extends
  ↪  ApdlExprGenerators(maxExprSize) {

  def genStatement: Gen[Statement] = Gen.oneOf(
    genWhile,
    genDoWhile,
    genIfThen,
    genExpressionStatement,
    genBreak,
    genContinue,
    genReturn
  )

  def genBlock: Gen[Block] = for {
    size <- Gen.choose(1, maxBlockSize)
    statements <- Gen.listOfN(size, genStatement) suchThat (l => isValid(l))
  } yield Block(statements)

  def genExpressionStatement: Gen[ExpressionStatement] = for {
    expr <- genExpr
  } yield ExpressionStatement(expr)

  def genWhile: Gen[While] = for {
```

```
    cond <- genExpr
    statement <- Gen.oneOf(genStatement, genBlock)
  } yield While(cond, statement)

  def genDoWhile: Gen[DoWhile] = for {
    cond <- genExpr
    statement <- Gen.oneOf(genStatement, genBlock)
  } yield DoWhile(cond, statement)

  def genIfThenElse: Gen[IfThenElse] = for {
    cond <- genExpr
    trueStatement <- Gen.oneOf(genStatement, genBlock)
    falseStatement <- Gen.oneOf(genStatement, genBlock)
  } yield IfThenElse(cond, trueStatement, falseStatement)

  def genIfThen: Gen[IfThen] = for {
    cond <- genExpr
    statement <- Gen.oneOf(genStatement, genBlock)
  } yield IfThen(cond, statement)

  def genReturn: Gen[Return] = for {
    expr <- genExpr
  } yield Return(expr)

  def genBreak: Gen[Break] = Break()

  def genContinue: Gen[Continue] = Continue()

  def genVarAssignement(depth: Int): Gen[VarAssignement] = for {
    target <- genExpr
    value <- genExpr
  } yield VarAssignement(target, value)

  def genFunctionDecl: Gen[FunctionDecl] = for {
    header <- genFunctionHeader
    body <- genFunctionBody
  } yield FunctionDecl(header, body)

  def genFunctionHeader: Gen[FunctionHeader] = for {
    retTyp <- genRetTyp
    id <- Gen.identifier
    nbParameters <- Gen.choose(0,10)
    parameters <- Gen.listOfN(nbParameters,genTypedIdentifier)
  } yield FunctionHeader(retTyp, id, parameters)

  def genFunctionBody: Gen[FunctionBody] = for {
    block <- genBlock
  } yield FunctionBody(block)

  def genNewVal(depth: Int): Gen[NewVal] = for {
    symbol <- genSymbol
    typ <- genTyp
    init <- genExpr
  } yield NewVal(symbol, typ, init)

  def genNewVar(depth: Int): Gen[NewVar] = Gen.oneOf(genNewVarNone, genNewVarSome(depth))

  def genNewVarNone: Gen[NewVar] = for {
    symbol <- genSymbol
    typ <- genTyp
  } yield NewVar(symbol, typ, None)

  def genNewVarSome(depth: Int): Gen[NewVar] = for {
    symbol <- genSymbol
    typ <- genTyp
    init <- genExpr
```

```
  } yield NewVar(symbol, typ, Some(init))

  def genNewArray(depth: Int): Gen[NewArray] = for {
    symbol <- genSymbol
    typ <- genTfArray
    init <- genArrayInitValue(depth)
  } yield NewArray(symbol, typ, init)

  def genArrayInit(depth: Int): Gen[ArrayInit] = Gen.oneOf(genArrayInitValue(depth),
   ↪  genArrayInitCapacity)

  def genArrayInitValue(depth: Int): Gen[ArrayInitValue] = for {
    values <- Gen.listOf(genExpr)
  } yield ArrayInitValue(values)

  def genArrayInitCapacity: Gen[ArrayInitCapacity] = for {
    cap <- genLiteral
  } yield ArrayInitCapacity(cap)
}

class ApdlDefineGenerators(maxExprSize: Int = 4, maxBlockSize: Int = 10) extends
 ↪  ApdlStatementGenerators(maxExprSize, maxBlockSize) {

  def genGen: Gen[apdl.parser.Gen] = for {
    g <- Gen.alphaNumStr
    s <- Gen.alphaNumStr
    l <- Gen.alphaNumStr
    e <- Gen.alphaNumStr
    typ <- stdTypGen
    t <- Gen.oneOf(None, Some(typ))
  } yield apdl.parser.Gen(g, s, l, e, t)

  def genGens: Gen[Map[String, apdl.parser.Gen]] = for {
    id <- Gen.listOf(Gen.identifier)
    gen <- Gen.listOf(genGen)
  } yield (id zip gen).toMap

  def inGen: Gen[Inputs] = for {
    params <- Gen.nonEmptyListOf(genParameter)
  } yield Inputs(params)

  def outGen: Gen[Output] = for {
    t <- typGen
  } yield Output(t)

  def genDefineComponent: Gen[ApdlDefineComponent] = for {
    id <- Gen.identifier
    params <- Gen.listOf(genParameter)
    in <- inGen
    out <- outGen
    gens <- genGens
  } yield ApdlDefineComponent(id, params, in, out, gens)

  def genDefineInput: Gen[ApdlDefineInput] = for {
    id <- Gen.identifier
    params <- Gen.listOf(genParameter)
    gens <- genGens suchThat (m => m.nonEmpty)
  } yield ApdlDefineInput(id, params, gens)

  def genDefineTransform: Gen[ApdlDefineTransform] = for {
    funcDecl <- genFunctionDecl
  } yield ApdlDefineTransform(funcDecl)

  /* APDL Transform DSL case class Generator */
}
```

```
class ApdlProjectGenerators(maxExprSize: Int = 4, maxBlockSize: Int = 10,
  ↪  maxGeneralListSize: Int = 10)
  extends ApdlDefineGenerators(maxExprSize, maxBlockSize) {

  def genProject: Gen[ApdlProject] = for {
    l <- Gen.choose(0, maxGeneralListSize)
    name <- genIdentifier
    devices <- Gen.listOfN(l, genDevice)
    defineInputs <- Gen.listOfN(l, genDefineInput)
    defineComponents <- Gen.listOfN(l, genDefineComponent)
    defineTransforms <- Gen.listOfN(l, genDefineTransform)
  } yield ApdlProject(name, devices, defineInputs, defineComponents, defineTransforms)

  def genDevice: Gen[ApdlDevice] = for {
    l <- Gen.choose(0, maxGeneralListSize)
    name <- genIdentifier
    id <- genIdentifier
    framework <- genIdentifier
    inputs <- Gen.listOfN(l, genInput)
    serials <- Gen.listOfN(l, genSerial)
    params <- Gen.listOfN(l, for {
      _1 <- genIdentifier
      _2 <- genIdentifier
    } yield (_1, _2))
  } yield ApdlDevice(name, id, framework, inputs, serials, params.toMap)

  def genInput: Gen[ApdlInput] = for {
    l <- Gen.choose(0, maxGeneralListSize)
    id <- genIdentifier
    inputName <- genIdentifier
    params <- Gen.listOfN(l, genParameter.map(p => p.id))
  } yield ApdlInput(id, inputName, params)

  def genSerial: Gen[ApdlSerial] = for {
    name <- genIdentifier
    sampling <- genSampling
  } yield ApdlSerial(name, sampling)

  def genSampling: Gen[ApdlSampling] = Gen.oneOf(genSamplingUpdate, genSamplingTimer)

  def genSamplingUpdate: Gen[ApdlSamplingUpdate.type] = ApdlSamplingUpdate

  def genSamplingTimer: Gen[ApdlSamplingTimer] = for {
    unit <- genTimeUnit
    value <- Gen.posNum[Int]
  } yield ApdlSamplingTimer(value, unit)

  def genTimeUnit: Gen[ApdlTimeUnit] = Gen.oneOf(ApdlTimeUnit.values)
}
```

Appendix F

APDL Code Generator for Property-based Testing

DslApdlBackendGenerators.scala

```
package apdl.parser

import apdl.parser.ApdlTimeUnit._
import apdl.parser.ApdlType.{Bool, Byte, Char, Double, Float, Id, Int, Long, Short, Str}

import scala.Function.tupled

/**
  * An code generators which target the apdl language itself
  * Primarly use for test and try
  */
trait DslApdlBackendGenerators extends TransformApdlBackendGenerators {

  def toApdlCode(define: ApdlDefine): String = define match {
    case ApdlDefineInput(name, parameters, gens) =>
      s"""
         |@define input $name ${parameters map toApdlCode mkString " "} {
         |  ${gens map toApdlCode mkString "\n"}
         |}
       """.stripMargin
    case ApdlDefineComponent(name, parameters, inputs, output, gens) =>
      s"""
         |@define component $name ${parameters map toApdlCode mkString " "} {
         |  ${toApdlCode(inputs)}
         |  ${toApdlCode(output)}
         |  ${gens map toApdlCode mkString "\n"}
         |}
       """.stripMargin
    case ApdlDefineTransform(functionDecl) =>
      s"""
         |@define transform ${toApdlCode(functionDecl)}
       """.stripMargin
  }

  def toApdlCode(parameter: Parameter): String = s"${parameter.id} : ${parameter.typ}"

  def toApdlCode(gen: (String, apdl.parser.Gen)): String = {
    val (id, g) = gen
```

```
    s"""
       |@gen $id {
       |  ${toApdlCode(g)}
       |}
     """.stripMargin
  }

  def toApdlCode(inputs: Inputs): String = s"@in ${toApdlCode(inputs.parameters)}"

  def toApdlCode(output: Output): String = s"@out ${toApdlCode(output.outputType)}"

  def toApdlCode(outputType: ApdlType): String = outputType match {
    case Str => "str"
    case Id => "id"
    case Int => "int"
    case Float => "float"
    case Long => "long"
    case Bool => "bool"
    case Double => "double"
    case Short => "short"
    case Char => "char"
    case Byte => "byte"
  }

  def toApdlCode(gen: Gen): String =
    s"""
       |global = "${gen.global}"
       |setup = "${gen.setup}"
       |loop = "${gen.loop}"
       |expr = "${gen.expr}"
       |${gen.typ match {
      case Some(value) => s"type = ${toApdlCode(value)}"
      case None => ""
    }}
     """.stripMargin

  def toApdlCode(x: Map[String, Gen]): String = x.map { case (k, v) =>
    s"""
       |@gen $k {
       | ${toApdlCode(v)}
       |}
    """.stripMargin
  } mkString "\n"

  def toApdlCode(parameters: Seq[Parameter]): String = parameters.map(toApdlCode).mkString("
    ↪  ")

  def toApdlCode(device: ApdlDevice): String =
    s"""
       |@device ${device.name} {
       |  id = ${device.id}
       |  framework = ${device.framework}
       |  ${device.inputs map toApdlCode mkString "\n"}
       |  ${device.serials map toApdlCode mkString "\n"}
       |  ${device.additionalParameters map tupled((k, v) => s"$k = $v") mkString "\n"}
       |}
     """.stripMargin

  def toApdlCode(input: ApdlInput): String =
    s"@input ${input.identifier} ${input.defineInputIdentifier} ${input.args mkString " "}"

  def toApdlCode(serial: ApdlSerial): String = s"@serial ${serial.inputName}
    ↪  ${toApdlCode(serial.sampling)}"

  def toApdlCode(timeUnit: ApdlTimeUnit): String = timeUnit match {
    case _: ns.type => "ns"
```

```
    case _: ms.type => "ms"
    case _: s.type => "s"
    case _: m.type => "m"
    case _: h.type => "h"
    case _: d.type => "d"
  }

  def toApdlCode(sampling: ApdlSampling): String = sampling match {
    case ApdlSamplingUpdate => "update"
    case ApdlSamplingTimer(value, timeUnit) => s"each $value ${toApdlCode(timeUnit)}"
  }

  def toApdlCode(project: ApdlProject): String =
    s"""
       |project_name = "${project.name}"
       |
       |${project.devices map toApdlCode mkString "\n"}
       |
       |${project.defineComponents map toApdlCode mkString "\n"}
       |
       |${project.defineInputs map toApdlCode mkString "\n"}
       |
       |${project.defineTransforms map toApdlCode mkString "\n"}
     """.stripMargin
}
```

TransformApdlBackendGenerators.scala

```
package apdl.parser

trait TransformApdlBackendGenerators {

  def toApdlCode(tfTyp: TfRetTyp): String = tfTyp match {
    case TfFloat => "float"
    case TfInt => "int"
    case TfDouble => "double"
    case TfLong => "long"
    case TfBoolean => "bool"
    case TfChar => "char"
    case TfArray(typ) => s"${toApdlCode(typ)}[]"
    case TfByte => "byte"
    case TfShort => "short"
    case TfVoid => "void"
  }

  def toApdlCode(expr: Expr): String = expr match {
    case Add(left, right) => s"(${toApdlCode(left)} + ${toApdlCode(right)})"
    case Mul(left, right) => s"(${toApdlCode(left)} * ${toApdlCode(right)})"
    case Sub(left, right) => s"(${toApdlCode(left)} - ${toApdlCode(right)})"
    case Div(left, right) => s"(${toApdlCode(left)} / ${toApdlCode(right)})"
    case Cast(tfTyp, ex) => s"((${toApdlCode(tfTyp)})${toApdlCode(ex)})"
    case Literal(value) => s"$value"
    case Symbol(name) => s"$name"
    case FunctionCall(funcName, args) => s"$funcName(${args map toApdlCode mkString ","})"
    case ArrayAccess(array, field) => s"${toApdlCode(array)}[${toApdlCode(field)}]"
    case VarAssignement(target, value) => s"${toApdlCode(target)} = ${toApdlCode(value)}"
    case True() => s"true"
    case False() => s"false"
    case Or(left, right) => s"(${toApdlCode(left)} || ${toApdlCode(right)})"
    case And(left, right) => s"(${toApdlCode(left)} && ${toApdlCode(right)})"
    case Not(booleanExpr) => s"(!${toApdlCode(booleanExpr)})"
```

```
    case Greater(left, right) => s"(${toApdlCode(left)} > ${toApdlCode(right)})"
    case Smaller(left, right) => s"(${toApdlCode(left)} < ${toApdlCode(right)})"
    case GreaterEquals(left, right) => s"(${toApdlCode(left)} >= ${toApdlCode(right)})"
    case SmallerEquals(left, right) => s"(${toApdlCode(left)} <= ${toApdlCode(right)})"
    case Equals(left, right) => s"(${toApdlCode(left)} == ${toApdlCode(right)})"
    case NotEquals(left, right) => s"(${toApdlCode(left)} != ${toApdlCode(right)})"
  }


  def toApdlCode(statement: Statement): String = statement match {
    case While(cond, stat) => s"while(${toApdlCode(cond)})${toApdlCode(stat)}"
    case DoWhile(cond, stat) => s"do ${toApdlCode(stat)} while(${toApdlCode(cond)})"
    case IfThenElse(cond, trueBranch, falseBranch) => s"if(${toApdlCode(cond)})
     ↪  ${toApdlCode(trueBranch)} else ${toApdlCode(falseBranch)}"
    case IfThen(cond, ifTrue) => s"if(${toApdlCode(cond)}) ${toApdlCode(ifTrue)}"
    case Return(expr) => s"return ${toApdlCode(expr)}"
    case Break() => s"break"
    case Continue() => s"continue"
    case Block(statements) =>
      s"""
         |{
         |  ${statements map toApdlCode mkString "\n"}
         |}
      """.stripMargin
    case ExpressionStatement(expression) => s"${toApdlCode(expression)}"
    case decl: Declaration => decl match {
      case FunctionDecl(FunctionHeader(resultType, identifier, parameters),
       ↪  FunctionBody(body)) =>
       s"""
          | def $identifier (${parameters map toApdlCode mkString ","}) ->
↪  ${toApdlCode(resultType)}
          |  ${toApdlCode(body)}
       """.stripMargin
      case NewVal(symbol, typ, init) => s"val ${toApdlCode(symbol)} : ${toApdlCode(typ)} =
       ↪  ${toApdlCode(init)}"
      case NewVar(symbol, typ, init) => s"val ${toApdlCode(symbol)} : ${toApdlCode(typ)}" +
       ↪  s"${
       init match {
         case Some(value) => s" = ${toApdlCode(value)}"
         case None => ""
       }
      }"
      case NewArray(symbol, typ, init) => s"${toApdlCode(symbol)} ${toApdlCode(typ)} =
       ↪  ${toApdlCode(init)}"
    }
  }
  def toApdlCode(typedIdentifier: TypedIdentifier): String =
   ↪  s"${typedIdentifier.name}:${toApdlCode(typedIdentifier.typ)}"

  def toApdlCode(init: ArrayInit): String = init match {
    case ArrayInitValue(values) => s"{${values map toApdlCode mkString ","}}"
    case ArrayInitCapacity(capacity) => s"[${toApdlCode(capacity)}]"
  }
}
```

Appendix G

Implementation of the Preprocessor's Tests

IncludeTest.scala

```
import java.io.{File, PrintWriter, StringWriter}

import apdl.ApdlTestException
import apdl.parser.IncludeProcessor
import org.scalacheck.Gen
import org.scalacheck.Prop._
import org.scalatest.FlatSpec
import org.scalatest.prop.Checkers

class IncludeTest extends FlatSpec with Checkers {

  private val maxFileNameLength = 20
  private val maxNbOfFilesToInclude = 10

  def genApdlFileName: Gen[String] = for {
    l <- Gen.choose(1, maxFileNameLength)
    name <- Gen.listOfN(l, Gen.alphaNumChar)
  } yield name.mkString

  def genIncludableApdlFile: Gen[ApdlIncludableTestFile] = for {
    name <- genApdlFileName
    content <- Gen.listOf(Gen.alphaNumStr) suchThat (_.nonEmpty)
  } yield ApdlIncludableTestFile(name, content.mkString("\n"))

  def genMainApdlFile: Gen[ApdlMainTestFile] = for {
    name <- genApdlFileName
    nbFile <- Gen.choose(0, maxNbOfFilesToInclude)
    fileToIncludes <- Gen.listOfN(nbFile, genIncludableApdlFile)
    content <- Gen.listOf(Gen.alphaNumStr) suchThat (_.nonEmpty)
  } yield ApdlMainTestFile(name, content, fileToIncludes)

  def trimSpace(s: String): String = s
    .replaceAll(" ", "")
    .replaceAll("\n", "")
    .replaceAll("\t", "")
    .replaceAll("\f", "")

  behavior of "IncludeProcessor"
```

```
  it should "replace any string inside the double quote by content of that string" in {
    check {
      val includeProcessor = new IncludeProcessor
      forAllNoShrink(genMainApdlFile) { file =>
        // create the main file source
        val main = new StringWriter

        val rndContent = scala.util.Random.shuffle(file.content ::: file.includes)

        val expected = rndContent.map {
          case s: String => s
          case f: ApdlIncludableTestFile => f.content
          case _ => throw new ApdlTestException("Include test failed...")
        }.mkString("\n")

        val content = rndContent.map {
          case s: String => s
          case f: ApdlIncludableTestFile => s"""@include "${f.name}.apdl" """
          case _ => throw new ApdlTestException("Include test failed...")
        }.mkString("\n")

        main.append(content).flush()
        val mainSource = main.toString
        main.close()

        // create all the other file
        val filenames = file.includes.map(_.name)
        file.includes.foreach { f =>
          val name = f.name
          val content = f.content
          val file = new File(s"$name.apdl")
          val pw = new PrintWriter(file)
          pw.append(content).flush()
          pw.close()
        }

        // check property
        val result = includeProcessor.process(mainSource)

        val testResult = trimSpace(result) == trimSpace(expected)

        // delete all file
        filenames.foreach { n =>
          val file = new File(s"$n.apdl")
          file.delete()
        }

        // return result of forAll
        testResult
      }
    }
  }

  case class ApdlIncludableTestFile(name: String,
                                    content: String)

  case class ApdlMainTestFile(name: String,
                              content: List[String],
                              includes: List[ApdlIncludableTestFile])
}
```

Appendix H

APDL Component file

apdl_component.apdl

```
@define input analogInput pin:str {
    @gen mbed {
        global = "AnalogIn @id(@pin);"
        setup = ""
        loop = ""
        expr = "@id.read()"
        type = float
    }
    @gen arduino {
        global = ""
        setup = ""
        loop = ""
        expr = "analogRead(@pin)"
        type = int
    }
}

@define input digitalInput pin:str {
    @gen mbed {
        global = "DigitalIn @id(Dpin);"
        setup = ""
        loop = ""
        expr = "@id.read()"
        type = float
    }
    @gen arduino {
        global = ""
        setup = ""
        loop = ""
        expr = "digitalRead(@pin)"
        type = int
    }
}
```