Dissertation

A FRAMEWORK FOR STATIC TYPE-CHECKING AND PRIVACY POLICY VALIDATION

Christos Makridis

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2019

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

A Framework for Static Type-Checking and Privacy Policy Validation

Christos Makridis

Supervisor

Dr. Anna Philippou

The Individual Diploma Thesis was submitted towards partially meeting the requirements for obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus

May 2019

Acknowledgments

For the completion of this thesis, I would like to express my heartfelt gratitude to both my supervisor Dr. Anna Philippou and the expert Dr. Dimitrios Kouzapas. I deeply appreciate the time and effort they have put on guiding me through this thesis, as well as for teaching me the necessary knowledge for finally achieving its fulfillment.

Abstract

This thesis discusses the steps taken into the creation of a framework that is responsible for type checking a system, regarding its privacy integrity. The framework follows the principles of formal methods and the Privacy calculus, as to how a system can combine those two in order to process private data. The concept of the framework has lead to the creation of a new language called π-val, which allows the consideration of the previous principles into high-level programming. The grammar and the compiler of π-val are both worth of explanation in this thesis and thus described extensively, along with a case study scenario.

Table of Contents

Chapter 1 Introduction **1**

1.1. Thesis motivation 1

1.2. Thesis purpose 2

1.3. Outline of document 4

Chapter 2 Previous Work **5**

2.1. Formal methods for privacy 5

2.2. Privacy Policies and Calculi 6

2.2.1. π-calculus 7

2.2.2. π-calculus with groups 7

2.2.3. Privacy calculus 8

2.3. Static Type Checking 9

2.4. Using a meta-compilation system 10

2.4.1. General Compiler Design 10

2.4.2. Introduction to JastAdd 12

Chapter 3 Framework **15**

3.1. A Framework for Static Type-Checking on Policy Validations 15

3.2. The need for a new language 16

Chapter 4 Building the language **18**

4.1. Introduction 18

4.2. Lexical Analysis 20

4.3. Syntax Analysis 21

4.3.1. Backbone 22

4.3.2. Channels 32

4.3.3. Policies 33

4.4. Class Hierarchy 35

4.4.1. Backbone 36

4.4.2. Channels 45

4.4.3. Policies 46

4.5. Semantic Analysis 47

4.5.1. Basic aspects 47

4.5.2. Finding errors 48

4.5.3. Expanding aspects 49

4.5.4. Finding violations 51
4.6. Code Generation..51
4.7. Extra Features...52
Chapter 5 Case study 54
5.1. Test case scenario 54
5.2. Defining the policy 54
5.3. Writing the program 57
5.4. Errors and violations...63
Chapter 6 Conclusions 68
6.1. Overall 68
6.2. Limitations 69
6.3. Future work...69
Bibliography 71
Appendix A 1
Appendix B 1
Appendix C 1

Chapter 1

Introduction

1.1. Thesis motivation 1
1.2. Thesis purpose..2
1.3. Outline of document 4

1.1. Thesis motivation

Nowadays, privacy holds an enormous role in modern societies. There is a lot of debate on whether or not something is considered a violation of privacy for an individual. Moreover, for different cultures, ages and ways of life, people seem to perceive the concept of privacy differently. What is more, things get even more complicated when technology itself, along with people's interaction through technology is introduced to societies.

On the one hand, focusing on the technological nature of privacy, services that exist throughout the globe, find personal information collections valuable for their own growth and maturity. Having an enormous database of such data, makes it possible to analyze the behavior of their clients in order to enhance their experience. Therefore, it is truly understandable that the more information a service collects, the better for that service's development.

On the other hand, when speaking about services and clients, usually a physical person is involved on that equation. This means that the unreasonable collection of personal information of any individual, might have an impact on their integrity. The anonymized collection of a single information of a person, say for example their age, is not something challenging for that individual. Moreover, a series of information might be collected without, again, disturbing the privacy of that particular person. Although this does not form any problem yet, there will be a point where that collection of information has enough knowledge in order to finally match and identify the individual. This is where things become complicated and the question “How many is too many?” arises.

Furthermore, a collaboration between services can also be possible into the real world. This means that the information gained by an office can automatically become a part of the collection of another business. This, again, brings back the previous question, as the data held in a database may not inflict any problem, but combined with another collection, might give enough knowledge to violate the privacy of a person.

As a result, many societies and organizations have come to the point of realizing that a structural policy that ensures the integrity of private data should be introduced to their official documents. An example of such policy is the General Data Protection Regulation, GDPR [2, 14], which is a regulation on data protection and privacy for all individuals within the European Union and the European Economic Area. This law ensures that data may be collected or stored only when users have accepted the reasons of that data collection and when data are indeed necessary for those purposes. As a result, companies are faced with an extra burden to ensure that their work respects privacy requirements. This expands to the use of computer systems industrially and thus interests this thesis.

Finally, similarly to GDPR’s formation, by defining formally what privacy is, computer systems can be used, in order to satisfy the urgency of protecting private data. Therefore, a framework based on formal definitions for privacy violations, should be able to locate such breaches in computer systems and report the issue, before even the systems begin the execution of their services.

1.2. Thesis purpose

The purpose of this thesis is the development of a programming framework, capable of static type-checking the validity of a given system’s privacy integrity. This goal can be achieved by using formal methods for privacy [13]. Based on those, privacy policies can be split into some basic abstract categories, which are going to be described in Section

2.1. Therefore, the framework can be implemented with the concepts of such categories, in order to be able to inspect the structure of the policy of a given system.

Furthermore, the framework must also be following the principles of Privacy calculus [8], as it introduces the concept of private data. The Privacy calculus is based on the π-calculus [11] with groups [3], which is extended via a privacy model that allows to consider private data. More on these calculi will be discussed on Section 2.2. While further trying to implement the operations of Privacy calculus throughout the framework, static type-checking for the purposes of privacy policy validation can become possible.

The presence of this framework will allow system developers to change the way a program is developed, while keeping the traditional software development methodologies. This means that software can be written in a language similar to an existing one, while the programmer takes into consideration the privacy policy validation scheme. The latter can be a set of rules, derived by any law enforcement document regarding privacy in systems. As a result, this assures the abstraction of the framework in terms of policy declaration and thus allowing the same program to be checked for validation by different privacy policies, depending on the needs of the validator.

In more detail, the framework will allow programmers to write code in an environment similar to Java [1]. The programmer will be able to declare classes, variables, methods and constructors, using Java’s conventional types or even perform logical and arithmetical expressions upon them. Branches and loops will also be allowed in the language. At that point, the programmer can define which classes should be considered as private data, by also declaring a specific policy for each one.

Furthermore, any private information exchanging between classes should be made only by a specific process. This process follows the same principles as the channels [8] of the Privacy calculus, which allow communication along themselves, describing in such way concurrent computations. Therefore, developers should have in mind that, upon the construction of a program, classes should include instances of such channels, as well as parallel methods, in order to achieve private data exchanging.

The above limitation makes this framework a link between the Privacy calculus and a high level programming language. As a result, operations on the language can easily be categorized into formal methods of privacy and thus be checked for validity.

Finally, with the success of this framework, the theoretical part of the above concept takes shape into real-world implementations. Of course, this will be just a proof of concept, leaving great room for further research and extension of its original capabilities.

1.3. Outline of document

This document will cover the steps taken in creating the framework discussed in the previous sections. Firstly, in Chapter 2, previous work will be discussed, including formal methods for privacy, privacy policies, all calculi that were involved in this thesis concept, as well as the meta-compilation [16] system that made the creation of this framework possible. Moreover, in Chapter 3, there will be explained some more aspects of the idea of the framework and its functional demands. What is more, in Chapter 4, all steps for the language's compilation will be explained thoroughly. Chapter 5 will cover a scenario that can take advantage of the framework's abilities. Finally, Chapter 6 will give an overview on what is the framework, what are its limitations and how it can be used in the future.

Chapter 2

Previous Work

2.1. Formal methods for privacy 5
2.2. Privacy Policies and Calculi 6
    2.2.1. π-calculus 7
    2.2.2. π-calculus with groups 7
    2.2.3. Privacy calculus 8
2.3. Static Type Checking 9
2.4. Using a meta-compilation system 10
    2.4.1. General Compiler Design 10
    2.4.2. Introduction to JastAdd 12

2.1. Formal methods for privacy

As mentioned before, privacy is a controversial topic. However, in order to understand and reason privacy-related requirements, it is mandatory to define what privacy is. There exist different types of reasoning regarding the definition and the concept of privacy. One of them is an analysis by the scholar Daniel J. Solove [12], which categorizes the taxonomy of privacy violations into four groups, called Invasion, Collection, Processing and Dissemination. This discrimination of privacy is the one used by the Privacy calculus [8], as well as the one that will concern this thesis.

The first group, Invasion, is irrelevant for this thesis, as it has nothing to do with information. Its notion is about physical abuse or personal influence and therefore cannot be measured in computer systems.

The second group, Collection, has to do with the gathering of personal information of a subject. This can apply when either the subject is unaware of such an assemblage, or is unwilling to contribute to that.

The third group, Processing, is about the handling of critical information. This category can be partitioned into some subcategories, but for the purposes of this thesis only two are being used, the Aggregation and the Identification. As mentioned in Chapter 1, a single piece of private information regarding an individual may not be challenging for their privacy's integrity, while a vast collection of information might become a threat for them. This forms the first subcategory, called Aggregation. An example of why Aggregation can be dangerous, is when gathering information about an individual's address, along with the hours that they are away from home. These information alone cannot harm the individual, but their combination can give a motive to potential burglars. What is more is the second subcategory of Processing violation, called Identification, which describes the possibility that some private information might be matched against other data, resulting in identifying the owner of those private Information.

The final group is Dissemination, where again, it can be partitioned into more subcategories, but for the purposes of this thesis, only its general idea will be used and explained. This category has to do with the propagation of private information into different agents in the system. If the owner of those information does not consent, then there is a breach of Dissemination.

2.2. Privacy Policies and Calculi

Depending on how policy is defined and categorized, various approaches exist on forming privacy policies. These are described by a set of rules for handling private data, while their purpose is to restrict any misuse of personal information regarding individuals or other entities. Moreover, the expressiveness of the model of privacy policies, specifies what aspects of privacy can be covered by those policies. As a result, a rigorous, yet flexible model is necessary for the purposes of this thesis, able to describe a typical computer system. This can be achieved by following and expanding the already existing rules of some computer science calculi.

2.2.1. π-calculus

In theoretical computer science, the π-calculus [11] is defined as a process calculus. As seen in Figure 2.1, it has a small syntax. However, this syntax gives great expressiveness, as it can describe any functional program. The calculus itself is based on two principles, processes and names. The former are a consecutive set of names' operations and other processes, while the latter are just entities, able to perform some operations. Its main characteristic is the concept of channels, where names can be used as such, in order to exchange information between parallel processes. What is more is that the information exchanged through channels can be channels themselves, giving further access to various processes.

The syntax in Figure 2.1, refers to P and Q as processes, while to x and y as names. It shows that a process can be one of the following operations.

P,Q	::=	x(y).P	Channel x waits to receive the name y and bind it. When the channel receives, the P process can carry on its execution.
	|	$\overline{\mathbf{x}}$<y>.P	Channel x waits to send the name y. When the channel sends, the P process can carry on its execution.
	|	P|Q	P and Q run simultaneously.
	|	(vx)P	Create a new channel x and immediately run P.
	|	!P	Repeatedly run copies of P.
	|	0	Process termination.

Figure 2.1: π-calculus syntax

2.2.2. π-calculus with groups

This extension of π-calculus [11], introduces the concept of groups [3]. In detail, groups consist of a hierarchical notion of entities, where no participant of a group can leak information to entities in other groups without permission. This is a step closer to the purpose of this thesis, as it gives the ability to monitor the transition of information between entities and therefore deny any illegal move.

2.2.3. Privacy calculus

Having in mind the formal methods mentioned in Section 2.1, a Privacy Model for Information Systems was introduced, called the Privacy calculus [8]. It is based on π-calculus [11], extended by the π-calculus with groups [3], which both were discussed in Section 2.2.1 and Section 2.2.2, respectively.

In this model, private data are part of the Privacy calculus. Such data must be declared to belong to individual entities, in order to differ from ordinary data. Therefore, an id must be declared for every piece of private data. Furthermore, they must be held by agents [18], which can perform various operations on each other's private data, such as store, send, receive or process them. These agents must be noted into a group hierarchy, therefore following the idea behind π-calculus with groups. Depending on the operations they perform, a series of permissions may be invoked and thus potential violations. For each pair of entities and private data, these permissions can include some of the categories of the taxonomy of privacy violations mentioned in Section 2.1. However, the same operations on regular data can never trigger an action. The permissions in Privacy calculus will be briefly explained on Table 2.1, regarding their relation with private data under the policy declaration of a specific group.

Moreover, the Privacy calculus is based on two axes. The first one is the definition of the program itself, which similarly to the π-calculus, it shows what operations its processes will make during the runtime. The other axis is the definition of a policy statement, which gives the set of permissions that the processes, depending on their group hierarchy are allowed to perform on certain private data. An example of a privacy policy statement and a program, based on the Privacy calculus can be found on Chapter 5.

Permission	**Explanation**
read	Private data may be read by processes belonging to their corresponding group.
update	Stored private data can be updated with a new piece of private data.
reference	Allows to gain access on a reference of the private data.
disseminate	Private data can be disseminated to processes that belong to the group G.
store	Allows to store private data.
readid	Allows to read the id of private data.
usage P	Allows to compare but not check equality of the private data with the specified constant P.
identify T	Allows to check equality of the private data with the specified private data type T.
aggregate	Allows the gathering of more than one information from the same data subject.

Table 2.1: Permission table

2.3. Static Type Checking

Static type checking is a technique for validating the integrity of a computer program. It applies in some programming languages such as Java [1] and is responsible for recognizing errors that are avoidable before the execution of the program. This allows the programmer to get informed early and thus change their development approach, in order to compile a program with no errors.

This technique is mostly used in order to ensure that the correct types of variables are being used throughout a program. Any action between incompatible types should be easily found. Therefore, by defining private data as types inside a language, any operation upon them can be tracked. As a result, if the permissions invoked by the processes that handle those private data are considered illegal, they can be reported.

What is more, the Privacy calculus [8], illustrates such operations by Labelled Transition Semantics, which give a mathematical notation about how data can travel between groups and what permissions from Table 2.1 are being triggered by their transitions.

2.4. Using a meta-compilation system

Meta-compilation systems [16] through various computations are responsible of transforming a computing machine into a meta-machine, which can control, analyze and imitate the work of the former. In order to achieve that, these systems use various approaches based on semantics, which enhance the initial machine.

The steps for using such a system follow the general compiler design process which will be expanded in Section 2.4.1. Based on those principles, the meta-compilation system JastAdd [4] can be used in order to fulfil the purposes of this thesis. As a result, it is the pillar of this thesis' framework and thus explained on Section 2.4.2.

2.4.1. General Compiler Design

A compiler design follows a series of phases in order to read, analyze and finally compile a program into an executable code. For the purposes of this thesis, some phases were skipped, as there is no need to perform some operations such as optimization of the input program. The stages that were followed and are going to be explained in the next paragraphs are the Lexical Analysis, the Syntax Analysis, the Semantic Analysis and the Code Generator [10]. All these stages make use of the same symbol table and are subjects to the same error handler, as seen in Figure 2.2.

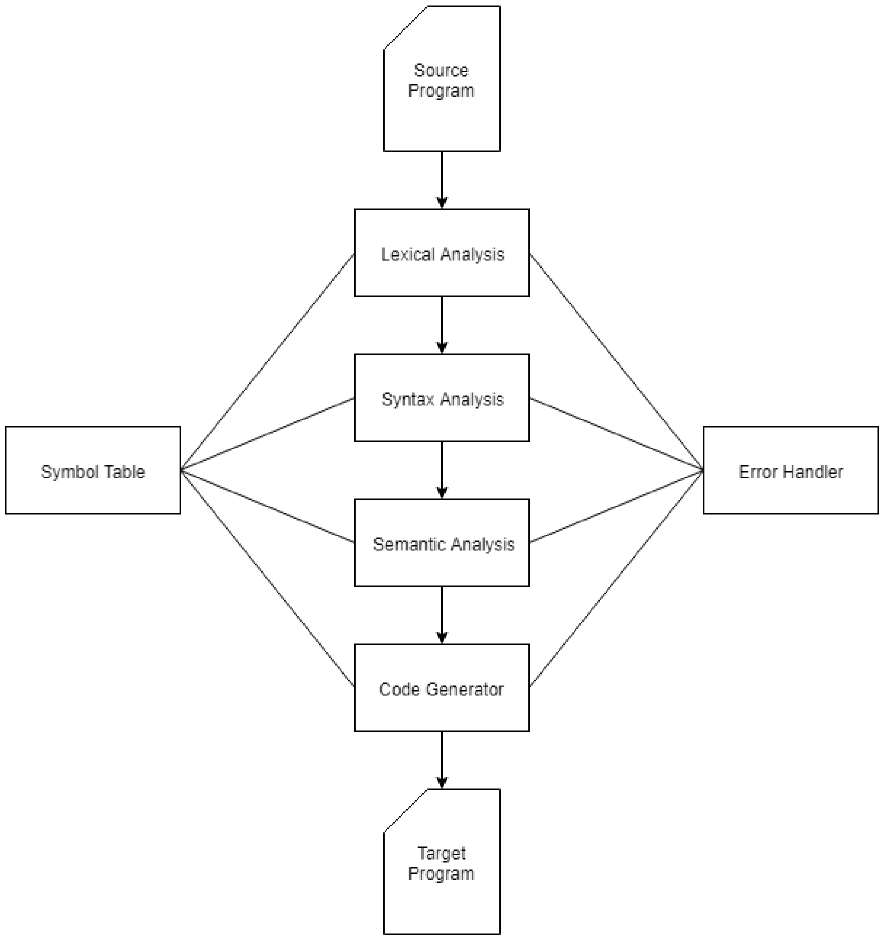


Figure 2.2 Compiler phases

The first part of the compiler for this framework, is the Lexical Analysis. This analysis is responsible for scanning a series of words that represent a program. These words are then translated into keywords, called tokens, making it possible to be considered later as a part of the grammar of a programming language.

The second part is the Syntax Analysis. In this stage (parsing stage), the tokens produced in the previous part, are coming together to form various groups, stated in a series of rules, called the grammar. If a series of tokens cannot be matched into a rule, then the program is rejected as it does not follow the correct grammar. These groups of tokens form the Syntax Tree, which is the tree to be traversed in order to gather information about the program as a whole.

The third part is the Semantic Analysis. Semantics are used in order to enhance the grammar, with rules that could not be expressed by its structure. This is where all the type-checking takes place, as well as the static policy checking for the purposes of this thesis. In order to achieve that, the compiler will have to traverse the Syntax Tree.

Finally, the last part of the compiler, which concerns this thesis, is the Code Generation. In this part, the Syntax Tree is being traversed for the final time, while it gets printed into a file, in such a way that it can be run by a computer. For the purposes of this thesis, the Code Generation transforms the input program into a runnable Java program, instead of machine code.

2.4.2. Introduction to JastAdd

JastAdd is a meta-compilation system [4]. It is based on Reference Attribute Grammars [4, 6] which means that inside the Syntax Tree, a node can reference another node, in a complete different place of the Syntax Tree. Moreover, the major advantage of this system is that it allows the developer of a compiler, analyzer or any other type of implementation tool to extend their work with minimal transfiguration of the original project.

This means that given a complete project on JastAdd, someone can just add some new modules called aspects [4, 7] in the project's grammar as separate files and the project will be extended immediately to those new rules. In order to achieve this modularity, JastAdd uses four types of files, each for a different purpose of the compiler design phases.

The first one is .flex type, which is used to make the Lexical Analysis of the program. Is uses the open-source software Flex, which is a lexical analyzer generator [9] and it gives the ability to use regular expressions in order to match lexical tokens.

The second type of files is .ast, which are responsible for creating all the Java Classes that can form the Abstract Syntax Tree. This is ensured by the system, as the Abstract Syntax Tree can only be made out of classes that extend a specific class, the ASTNode Class. These files also hold information about the Class Inheritance between those Classes in Java, as well as the Classes of their Children under the Abstract Syntax Tree. Although its necessity, this hierarchy is somewhat unrelated to the grammar itself, as it exists just to denote the types of children that are applicable under each class.

The third type is .parser files, which are responsible for the Syntax Analysis and basically hold the Grammar that is valid for the language being created, while using the Classes stated in the .ast files and the tokens stated in the .flex file.

Finally, the last type of files is either .jrag or .jadd, where both extensions are called aspect files [4, 7] and JastAdd does not differ between those two types. They are responsible for creating aspects (the former for declarative i.e. attributes and equations and the latter for imperative i.e. ordinary fields and methods). This is where the Semantics Analysis happens, as well as for the framework of this thesis the translation from the given language into pure Java happens.

In order to form those aspects throughout the classes that make up the Abstract Syntax Tree, JastAdd gives the ability to declare Synthesized and Inherited attributes that work very similar to Java methods. On the one hand, a Synthesized attribute of an ASTNode is computed by traversing some, or all of the children of that particular ASTNode. This gives the ability to declare the type, name and other aspects of the ASTNode. On the other hand, an Inherited attribute works in a similar way, but on the other way round. More specifically, instead of traversing the children of an ASTNode, it traverses their parents, in order to “inherit” information. JastAdd’s documentation notes that this is a different meaning for the word “inherit” than the one used in object-orientation. This means that in order to declare an inherited attribute for a particular ASTNode, JastAdd needs to know how to compute it in all the possible parents of that node.

What is more, if an attribute is known to be unchanged throughout the whole process, then it can be declared as “lazy”. This means that the attribute will be computed only the first time that it is being accessed and its value will be saved. Therefore, any other reference to that attribute will not have it re-calculated, but instead it will use the already calculated value.

Finally, JastAdd gives the ability to the developer to add ordinary Java fields and methods on ASTNode classes. These of course will reduce the extensibility of an ASTNode as they are not handled as attributes, but they can be very helpful for side computations.

Chapter 3

Framework

3.1. A Framework for Static Type-Checking on Policy Validations 15
3.2. The need for a new language 16

3.1. A Framework for Static Type-Checking on Policy Validations

For the purposes of this thesis, as discussed in Chapter 1 and having in mind previous work from Chapter 2, a framework for Static Type-Checking regarding Policy Validation had to be created. This framework, which acts as a compiler, given a standalone program written in a specific language that the framework can understand, should be able to verify the program's policy integrity using another input, called the policy statement. What is more, the program should be independent of any policy and thus the framework could check its policy integrity between different policy statements. The program and the policy statement can be two different or even more files, which follow the principles of the Privacy calculus as stated in Section 2.2.3. Such process can be visualized in Figure 3.1.

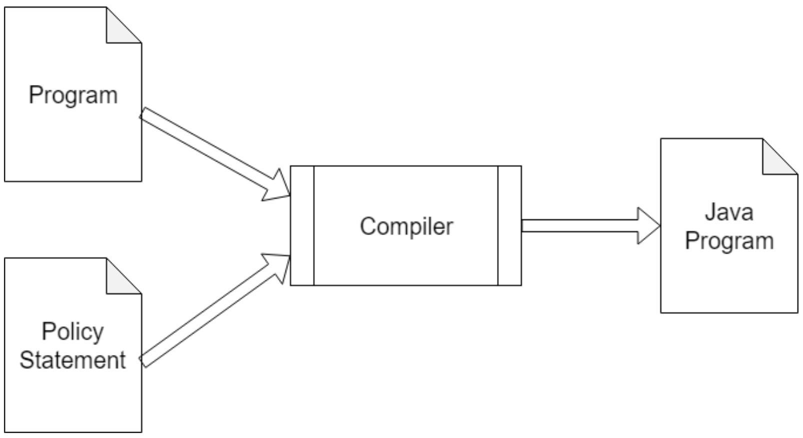


Figure 3.1: Policy validation process

As a result, this allows the programmers that are going to validate their work through this framework, to program in a similar to the traditional programming methodology, without fearing about interferences with the policy statement. Of course, each policy statement will give guidelines of what is permitted in each program. Therefore the policy statement should be adapted to the program's architecture rather than the other way around. The only thing the programmer needs to have in mind is the permissions that must not be breached, depending on the particular policy statement that their program would be validated against.

The goal of this framework is that, after the validation of the program through a policy statement, it should be transformed into a normal Java program. Therefore, knowing that the initial program is valid in terms of grammar and policy regulations, its exported Java version should be run directly by a Java compiler.

3.2. The need for a new language

In order to be able to create such a framework, a language that interprets the concepts of basic programming languages infused with channels, threaded methods and policies needed to be created. Therefore, it would be more practical to create such a language, in order to adapt it to the needs of the framework, rather than finding an existing one which might be limited in some aspects that are crucial to the framework. As a result, using the meta-compilation system JastAdd, which is explained in Section 2.4.2, the language π-val was built. It was named after a combination of its own theoretical principle and purpose. The former part of the name comes from the first letter of π-calculus [11], whose extension, the Privacy calculus [8] bases the functionality of the language, while the latter is a short for its goal which is the validation of privacy policies.

What is more, π-val gives the ability to the developer to keep the traditional software development methodologies, as it follows very similar grammar to the conventional programming language Java [1]. Moreover, the program is independent of any Privacy Policy scheme, therefore π-val developers can each time try to validate their projects with various Privacy Policy declarations, based on different law specifications.

The characteristic of π-val is that it is split in three abstractions, “Backbone”, “Channels” and “Policies”. Each part has its own grammar, Abstract Syntax Tree, class hierarchy and attributes. All of these were fused together in some key-points of the grammar, making it possible for the language to fulfil its original purpose.

Firstly, although the “Backbone” of π-val is independent of any privacy policy and can be a sole standalone program (even when it is fused with channels), the other way round does not hold. Therefore, the “Backbone” can describe alone any conventional program, while the other two parts need the presence of the former in order to operate as expected.

Secondly, the “Channels” part is the addition of the concepts of π-calculus in the language. It introduces a new type which gives the ability to transport information between classes in parallel processes. As a result, it brings the framework one step closer, but not there yet, to fulfilling its purpose of following the rules of the Privacy calculus.

Finally, the “Policies” part sets the hierarchy and the permissions of the classes of the program. In order to do so, it needs its policy declarations to follow the structure of the main program (the “Backbone”). Its concept is fully based on the Privacy calculus and thus completes this thesis’ goal. Although this part is independent from the “Channels” part, Privacy calculus is partially based on channels and parallel processes, therefore their usage is mandatory for giving purpose to the “Policies” part.

More about the structure of π-val will be discussed in Chapter 4.

Chapter 4

Building the language

4.1. Introduction....................18
4.2. Lexical Analysis....................20
4.3. Syntax Analysis....................21
4.3.1. Backbone....................22
4.3.2. Channels....................32
4.3.3. Policies....................33
4.4. Class Hierarchy....................35
4.4.1. Backbone....................36
4.4.2. Channels....................45
4.4.3. Policies....................46
4.5. Semantic Analysis....................47
4.5.1. Basic aspects....................47
4.5.2. Finding errors....................48
4.5.3. Expanding aspects....................49
4.5.4. Finding violations....................51
4.6. Code Generation....................51
4.7. Extra Features....................52

4.1. Introduction

As discussed in Chapter 3, π-val, the language needed for this framework interprets the concepts of basic programming languages infused with channels, threaded methods and policies. Therefore it is split in three abstractions, “Backbone”, “Channels” and “Policies”. The “Backbone” is a standalone part, whilst the “Channels” and “Policies” parts are expanding and thus depending on the “Backbone”. This concept can be illustrated, as seen in Figure 4.1.

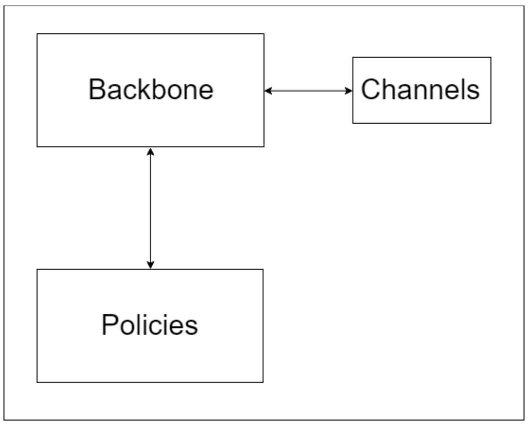


Figure 4.1: The framework's concept

The "Backbone" part is the main trunk of the language. It is based on a Java BNF [15] grammar which was reduced and changed to the needs of this thesis. Moreover, it is able to describe basic programming procedures such as declarations of variables and methods, branching, looping, casting, expressions, as well as more complex concepts such as objects and arrays, thus it has a fair expressiveness. It does not allow inheritance except that every class is an extension of the Object class. Finally, it is split into six parts, "Program", "Declarations", "Types", "Blocks and Commands", "Expressions" and "Primaries", which are going to be discussed in the next subsections.

The "Channels" part is the first extension of π-val. It introduces to the language a new type of Objects called Channels. This type will represent a communication line for asynchronous data transfer between parallel processes, similarly to the π-calculus [11]. In order to achieve that, threaded method calls are also introduced and handled by the framework in this part, in order to allow sending data to a channel or receiving data from it via parallel methods. The data that a channel can transfer must be of a private data type, which will be discussed in the next paragraph.

The "Policies" part is another addendum to the backbone of the language. While the "Backbone" can describe a standalone language, the addition of policies makes the purpose of this thesis possible. Nevertheless, "Policies" cannot exist without the "Backbone". Each policy affects one particular class that has already been declared in the backbone as a normal class, transforming it into a private data class. For that private data class, the policy provides a set of permissions, for each group of classes that might use it.

This means that each class of each group invokes different permission rules varying on the use of that particular private data class. After that, these rules will be matched to the policy and if there are any violations, the framework will reject the input program-policy pair.

4.2. Lexical Analysis

The Lexical Analysis part consists of a .flex file containing a set of regular expressions, responsible for recognizing various symbols and keywords called tokens that are necessary for the language. These tokens will be used throughout the grammar and are shown in Table 4.1, where their corresponding symbol or word is also displayed. This allows JastAdd to recognize these symbols, in order to split an input program into tokens.

Symbol	Token	Symbol	Token	Symbol	Token
[]	LRSBR	|	BOR	while	WHILE
<=	LE	+	PLUS	Channel	CHANNEL
>=	GE	-	MINUS	<-	REC
<<	SL	*	MUL	@	AT
>>	SR	/	DIV	policy	POLICY
>>>	SRR	%	MOD	STORE	STORE
<	LT	=	EQUALS	READ	READ
>	GT	null	NULL	READID	READID
==	EQCHECK	boolean	BOOLEAN	AGGREGATE	AGGREGATE
!=	NEQCHECK	break	BREAK	REFERENCE	REFERENCE
&&	AND	char	CHAR	UPDATE	UPDATE
||	OR	class	CLASS	IDENTIFY	IDENTIFY
(	LPAREN	continue	CONTINUE	USAGE	USAGE
)	RPAREN	double	DOUBLE	DISSEMINATE	DISSEMINATE
{	LBRACE	else	ELSE	{Identifier}	IDENTIFIER
}	RBRACE	final	FINAL	{Filename}	FILENAME
[	LSBRACE	float	FLOAT	"true"|"false"	BOOLEANLITERAL
]	RSBRACE	if	IF	{Integer}	INTEGERLITERAL
;	SEMICOLON	int	INT	{Float}	FLOATLITERAL
,	COMMA	new	NEW	{Character}	CHARACTERLITERAL
.	DOT	return	RETURN	{String}	STRINGLITERAL
^	XOR	this	THIS		
&	BAND	void	VOID		

Table 4.1: Lexical Analysis tokens

What is more, as seen on Table 4.2, there are aliases for regular expressions in order to make Lexical Analysis more modular. These rules are put at the very end of the .flex file, thus any combination of letters and digits that start with a letter and is not matched with previous rules is considered to be an identifier. Identifiers will be used for class types and variable declarations throughout the language. Furthermore, there are regular expressions for integer, floats, strings and characters. Finally, both line and block comments are being recognized in the Lexical Analysis and are being ignored.

ALIAS	REGEX
{LineTerminator}	\r|\n|\r\n
{Letter}	[a-zA-Z_$]
{Comment}	("/*"([^*] | (*+[^*/]))**+\/)|("//".*)
{WhiteSpace}	{LineTerminator} | [\t\f]
{Non_zero_digit}	[1-9]
{Digit}	0|{Non_zero_digit}
{Digits}	{Digit}{Digit}*
{Integer_type_suffix}	[lL]
{Decimal_numeral}	"0"|{Non_zero_digit}{Digits}?
{Dec}	{Decimal_numeral}{Integer_type_suffix}?
{Integer}	{Dec}
{Signed_integer}	[+-]?{Digits}
{Exponent_part}	[eE]{Signed_integer}
{Float_suffix}	[fFdD]
{Float}	{Digits}[.]{Digits}?{Exponent_part}?{Float_suffix}? {Digits}{Exponent_part}?{Float_suffix}?
{Single_character }	[^\'\\]
{Escape_sequence}	[\b\n\t\r\f\'\"\\]
{Character}	\'({Single_character}|{Escape_sequence})\'
{String}	\"(\\.|[^\"\\])*\"
{Idenifier}	[a-zA-Z$]({Letter}|{Digit})*
{Filename}	"~"{Identifier}"."{Identifier}

Table 4.2: Lexical Analysis regular expressions

4.3. Syntax Analysis

The Syntax Analysis of the language, consists of some .parser files. These files describe the exact grammar of the language. This is the part where the language starts to split up into the 3 abstractions that were discussed previously. In the following paragraphs, these abstractions' grammars will be thoroughly explained, while they will be using the tokens shown in the Lexical Analysis (Section 4.2).

The notation can also be visualized in diagrams which can give a top-eye view of the Abstract Syntax Tree that will be created based on the grammar. These diagrams can be found in Appendix A.

The BNF [15] grammar that will be shown in the next pages, is a notation for context-free grammars where the symbol "::=" denotes that the nonterminal on its left side must be replaced with the expression on the right of the symbol. Wherever a "|" symbol exists, it means that there are more than one choices for the substitution of the left nonterminal. For this representation, let us assume four things:

1) Any phrase with small letters, inside <angle brackets> is a nonterminal.
2) Any phrase with CAPITAL letters, inside <angle brackets> is a terminal token.
3) Any word with CAPITAL letters that stands alone is a keyword.
4) Any nonterminal that ends in a question mark (?) is an optional token.

4.3.1. Backbone

The "Backbone" of the language is the main abstraction. It consists of 6 more splits, as mentioned in Section 4.1, the "Program", "Declarations", "Types", "Blocks and Commands", "Expressions" and "Primaries" in order to make the grammar more readable and understandable.

4.3.1.1. Program

The "Program" part is the goal of the grammar. All expressions must finally be substituted into the *goal* nonterminal. In order to achieve this, the compiler is responsible for combining all the files that were read into a single file, with a separating token denoting the file's name. This way, as stated in the grammar section below, each *goal* can include many files, each called *my file*, where each on of them can include various *type declarations*, discussed in the next subsection.

```
<goal>                  ::= <my files>
<my files>              ::= <my file>
                        | <my files> <my file>
<my file>               ::= <FILENAME> <type declarations>?
<type declarations>     ::= <type declaration>
                        | <type declarations> <type declaration>
```

Figure 4.2: Program grammar

4.3.1.2. Declarations

The "Declarations" part is where all possible declarations of the backbone are created. These include class, fields, constructors, methods, formal parameters and variable declarations. Note that the grammar is similar to Java, while it's missing the access and static modifiers. Moreover, there is only one option for final declarations, which is for fields. This will be used by the policy semantics analysis.

In more detail, for the "Backbone" part of the language, *type declaration* can only be either empty or have a *class declaration*. This will be enhanced and discussed later, under the "Policies" part of the language. A *class declaration,* defines a class similarly to how it is defined in Java, which can also be used as an Object type. The *identifier* denotes the name of the class, while the *class body* consists of various *class body declarations*. Finally, a *class body declaration* can either be a *class member declaration*, or a *constructor declaration*, where the former can be one of a *final field*, a *field* or a *method declaration*.

```
<type declaration>            ::= <class declaration> | SEMICOLON

<class declaration>           ::= CLASS <IDENTIFIER> <class body>
<class body>                  ::= LBRACE <class body declarations>? RBRACE
<class body declarations>     ::= <class body declaration>
                              | <class body declarations> <class body  declaration>

<class body declaration>      ::= <class member declaration>
                              | <constructor declaration>
<class member declaration>     ::= <field declaration>
                              | <final field declaration>
                              | <method declaration>
```

Figure 4.3: Declaration grammar part 1 of 4

As we go further down the grammar, *constructor declaration* consists of a *constructor declarator* and a *constructor body*. The former gives the *identifier* of the constructor and a list of *formal parameters,* while the latter is a list of *block statements* that might be lead by an *explicit constructor invocation*. The latter can only exist in the first line of a *constructor body*. *Formal parameters*, similarly to Java, are declared by a *type* and a *variable declarator id*, which will be both discussed in the next paragraphs. Finally, an *explicit constructor invocation* consists of the keyword *"THIS"* and an *argument list* which consists of *expressions* which again will be discusses in the next subsections.

```
<constructor declaration>            ::= <constructor declarator> <constructor body>
<constructor declarator>             ::= <IDENTIFIER> LPAREN <formal parameters>? RPAREN

<formal parameters>                  ::= <formal parameter>
                                     | <formal parameters> COMMA <formal parameter>
<formal parameter>                   ::= <type> <variable declarator id>

<constructor body>                   ::= LBRACE  <block statements>? RBRACE
                                     | LBRACE <explicit constructor invocation>
                                                    <block statement list>? RBRACE

<explicit constructor invocation>    ::= THIS LPAREN <argument list>? RPAREN
```

Figure 4.4: Declaration grammar part 2 of 4

Similarly to a *constructor declaration* in the previous paragraph, a *method declaration* consists of a header and a body. The *method header* is declared by a *type* or a *void type* and a *method declarator*. The *method body* on the other hand, consists only by one *block* which will be discussed later. The *method declarator* follows the exact same principles as the *constructor declarator*.

```
<method declaration>                 ::= <method header> <method body>
<method header>                      ::= <type> <method declarator>
                                     | <void type> <method declarator>
<method declarator>                  ::= <IDENTIFIER> LPAREN <formal parameters>? RPAREN
<method body>                        ::= <block>
```

Figure 4.5: Declaration grammar part 3 of 4

Finally, each *field declaration* can be either normal or final. It consists of a *type* and a list of *variable declarators*. Each *variable declarator* can either be a *variable declarator id*, or a *variable declarator id* with an *initializer* depending on the type of that variable (this will be checked under the Semantics Analysis (Section 4.5). Again, *initializers* will be discussed in the next subsections. Finally, a *variable declarator id* consists of an *identifier*, identifying the name of the variable.

```
<field declaration>              ::= <type> <variable declarators> SEMICOLON
<final field declaration>        ::= FINAL <type> <variable declarators> SEMICOLON

<variable declarators>           ::= <variable declarator>
                                 | <variable declarators> COMMA <variable declarator>
<variable declarator>            ::= <variable declarator id>
                                 | <variable declarator id> EQUALS <variable initializer>
                                 | <variable declarator id> EQUALS <array initializer>
<variable declarator id>         ::= <IDENTIFIER>
```

Figure 4.6: Declaration grammar part 4 of 4

4.3.1.3. Types

The "Types" part is responsible for all the types that are valid in the language. Similarly to Java, there exist the primitive types *boolean*, *int*, *char*, *float*, *double*, as well as the custom *class types* (Objects), which include by default the *String* type. For all these types, their corresponding *array type* can be matched in a recursive way, so that there can be multidimensional arrays. The *cast type* exist only to be matched under the *cast expressions*, while the *void type* exists only for *method declarations*.

```
<type>                    ::= <primitive type>
                          | <array type>
                          | <class type>

<primitive type>          ::= <numeric type>
                          | STRING
                          | BOOLEAN
<numeric type>            ::= <integral type>
                          | <floating point type>
<integral type>           ::= INT
                          | CHAR
<floating-point type>     ::= FLOAT
                          | DOUBLE

<array type>              ::= <type> LRSBR

<class type>              ::= <IDENTIFIER>

<cast type>               ::= LPAREN <IDENTIFIER> RPAREN
<void type>               ::= VOID
```

Figure 4.7: Types grammar

4.3.1.4. Blocks and Commands

The "Blocks and Commands" part is where each *block* is matched. A *block* can implement the body of a method, a constructor, a branch or a loop. Each *block* consists of various *block statements,* which can either be a *local variable declaration statement* or a *statement*. The former is formed by a *type* and a *variable declarator* list.

```
<block>                                   ::= LBRACE <block statements>? RBRACE
<block statements>                        ::= <block statement>
                                          | <block statements> <block statement>
<block statement>                         ::= <local variable declaration statement>
                                          | <statement>
<local variable declaration statement>    ::= <local variable declaration> SEMICOLON
<local variable declaration>              ::= <type> <variable declarators>
```

Figure 4.8: Blocks and Commands grammar part 1 of 4

A statement, can be an *if then statement*, an *if then else statement*, a *while statement* or a *statement without trailing substatement*. The latter consists of *statements* that have no continuity. These can be a single *block*, the *empty statement*, the *expression statement*, the *break statement*, the *continue statement*, or the *return statement*. Moreover, there is another category of statements, the *statement no short if*, which can either be a *statement without trailing substatement*, an *if then else statement no short if* or a *while statement no short if*. This category is the key to ensure that there is no ambiguity in the grammar, as this is a very common problem in programming grammars.

```
<statement>                                  ::= <statement without trailing substatement>
                                             | <if then statement>
                                             | <if then else statement>
                                             | <while statement>
<statement no short if>                      ::= <statement without trailing substatement>
                                             | <if then else statement no short if>
                                             | <while statement no short if>
<statement without trailing substatement>    ::= <block>
                                             | <empty statement>
                                             | <expression statement>
                                             | <break statement>
                                             | <continue statement>
                                             | <return statement>
```

Figure 4.9: Blocks and Commands grammar part 2 of 4

Furthermore, while the *statements* under the *statement without trailing substatement* are self-explanatory, the *statement expression* can be one of the three non terminals, *assignment*, *method invocation*, or *class instance creation expression*, which are all going to be discussed under the next subsections.

```
<break statement>          ::= BREAK SEMICOLON
<continue statement>       ::= CONTINUE SEMICOLON
<return statement>         ::= RETURN <expression>? SEMICOLON
<empty statement>          ::= SEMICOLON
<expression statement>     ::= <statement expression> SEMICOLON

<statement expression>     ::= <assignment>
                           | <method invocation>
                           | <class instance creation expression>
```

Figure 4.10: Blocks and Commands grammar part 3 of 4

Finally, the *if then statement* and the *while statement*, both need to be expressed by an *expression* and a *statement*. Moreover, in order to take advantage of the *statement no short if* rule, the *if then else statement* needs an *expression*, a *statement no short if* and another *statement*, while the *if then else statement no short if* needs an expression and two *statement no short if*. Similarly, the *while statement no short if*, needs an *expression* and a *statement no short if*.

```
<if then statement>                    ::= IF LPAREN <expression> RPAREN <statement>
<if then else statement>               ::= IF LPAREN <expression> RPAREN
                                                   <statement no short if> ELSE
                                                                <statement>
<if then else statement no short if>   ::= IF LPAREN <expression> RPAREN
                                                   <statement no short if> ELSE
                                                         <statement no short if>

<while statement>                      ::= WHILE LPAREN <expression> RPAREN <statement>
<while statement no short if>          ::= WHILE LPAREN <expression> RPAREN
                                                   <statement no short if>
```

Figure 4.11: Blocks and Commands grammar part 4 of 4

4.3.1.5. Expressions

The “Expressions” part is responsible for all the valid expressions of the language. These include *initializers* for both variables and arrays, *assignment expressions*, *conditional* and *equality expressions*, *shift*, *additive* and *multiplicative*, as well as *cast* and *unary expressions*. The latter includes *primaries* which will be discussed in the next section. Their exact syntax is shown in Figure 4.12 and Figure 4.13.

```
<array initializer>              ::= LBRACE <variable initializers>? RBRACE
<variable initializers>          ::= <variable initializer>
                                 | <variable initializers> COMMA <variable initializer>
<variable initializer>           ::= <expression>
<expression>                     ::= <assignment expression>
<assignment expression>          ::= <conditional expression>
                                 | <assignment>
<assignment>                      ::=  <field access> EQUALS <assignment expression>
                                 | <array access> EQUALS <assignment expression>


<conditional expression>         ::= <conditional or expression>
<conditional or expression>      ::= <conditional and expression>
                                 | <conditional or expression> OR <conditional and
expression>
<conditional and expression>     ::= <inclusive or expression>
                                 | <conditional and expression> AND <inclusive or
expression>
<inclusive or expression>        ::= <exclusive or expression>
                                 | <inclusive or expression> BOR <exclusive or expression>
<exclusive or expression>        ::= <and expression>
                                 | <exclusive or expression> XOR <and expression>
<and expression>                 ::= <equality expression>
                                 | <and expression> BAND <equality expression>
<equality expression>            ::= <relational expression>
                                 | <equality expression> EQCHECK <relational expression>
                                 | <equality expression> NEQCHECK <relational expression>
```

Figure 4.12: Expressions grammar part 1 of 2

```
<relational expression>        ::= <shift expression>
                               | <relational expression> LT <shift expression>
                               | <relational expression> GT <shift expression>
                               | <relational expression> LE <shift expression>
                               | <relational expression> GE <shift expression>
<shift expression>             ::= <additive expression>
                               | <shift expression> SL <additive expression>
                               | <shift expression> SR <additive expression>
                               | <shift expression> SRR <additive expression>
<additive expression>          ::= <multiplicative expression>
                               | <additive expression> PLUS <multiplicative expression>
                               | <additive expression> MINUS <multiplicative expression>
<multiplicative expression>    ::= <unary expression>
                               | <multiplicative expression> MUL <unary expression>
                               | <multiplicative expression> DIV <unary expression>
                               | <multiplicative expression> MOD <unary expression>
<cast expression>              ::= LPAREN <primitive type> RPAREN <unary expression>
                               | LPAREN <array type> RPAREN <unary expression>
                               | <cast type> <unary expression>
<unary expression>             ::= MINUS <unary expression>
                               | PLUS <unary expression>
                               | <primary>
                               | <cast expression>
```

Figure 4.13: Expressions grammar part 2 of 2

4.3.1.6. Primaries

The “Primaries” is the final part of the Backbone. It is responsible for all the handlers of variables that the program can have. This means that anything matched under the primary nonterminal can be a part of an expression, such as a *field access*, a *method invocation* or even a *literal*. Specifically, a *primary* can either be a *primary no new array*, or an *array creation expression*. The former can be a *literal*, *“THIS”* in order to refer to the current instance of an object, an *expression*, a *class instance creation expression*, a *field access*, a *method invocation* or an *array access*.

Method invocation and *field access* are very similar in terms of grammar. They both need an *identifier* in order to define the method’s or the field’s name and both of them might have a *primary* handler in order to define where the method or the field is defined. Finally, a *method invocation* has an *argument list*. On the other hand, *array access* is a way to access an index of an array. The handler in this situation is a *primary no new array*, while an *expression* is needed to set the index of the array.

```
<primary>                   ::= <primary no new array>
                            | <array creation expression>
<method invocation>         ::= <IDENTIFIER> LPAREN <argument list>? RPAREN
                            | <primary> DOT <IDENTIFIER>
                                        LPAREN <argument list>? RPAREN
<field access>              ::= <IDENTIFIER>
                            | <primary> DOT <IDENTIFIER>
<primary no new array>      ::= <literal>
                            | THIS
                            | LPAREN <expression> RPAREN
                            | <class instance creation expression>
                            | <field access>
                            | <method invocation>
                            | <array access>
<array access>              ::= <primary no new array> LSBRACE <expression> RSBRACE

<literal>                   ::= <INTEGERLITERAL
                            | <FLOATLITERAL>
                            | <BOOLEANLITERAL>
                            | <CHARACTERLITERAL>
                            | <STRINGLITERAL>
                            | <NULL>
```

Figure 4.14: Primaries grammar part 1 of 2

Class instance creation expression is the expression responsible for creating a new Object. Therefore it needs to call a constructor using the identifier of its class (the *class type*) and an *argument list*. An *argument list* is just a set of *expressions*. Moreover, an *array creation expression* is responsible for creating a new array of a *primitive type*, or a *class type*. Therefore it needs the identifier of that type, along with a list of dimensional expressions (*dim epxrs*), and a list of empty dimensional expressions (*dim emps*). This allows to create new multidimensional arrays, while occupying memory only for some of their first dimensions, just like in Java [1].

```
<class instance creation expression>     ::= NEW <class type>
                                                 LPAREN <argument list>? RPAREN
<argument list>                          ::= <expression>
                                         | <argument list> COMMA <expression>
<array creation expression>              ::= NEW <primitive type> <dim exprs> <dim emps>?
                                         | new <class type> <dim exprs> <dim emps>?
<dim exprs>                              ::= <dim expr>
                                         | <dim exprs> <dim expr>
<dim expr>                               ::= LSBRACE <expression> RSBRACE
<dim emps>                               ::= LRSBR
                                         | <dim emps> LRSBR
```

Figure 4.15: Primaries grammar part 2 of 2

4.3.2. Channels

The Channels' grammar introduces to the language, the capabilities discussed in the introduction of this chapter (Channel types, sending and receiving between processes using channels and threaded methods). All these are shown below, as well as how they are infused to the "Backbone" of the language. Let us assume that in the next figures, rules that are empty, followed by the comment notation "%%" indicate that more rules exist in previous impressions of the same token.

A *channel type* is used to define a channel of a particular class that is able to transfer another channel, or object, thus it needs this information in its grammar. Its creation is similar to that of a *class instance creation*. The *channel receive expression* and the *channel send expression*, both need a *unary expression* to represent the channel to receive from or send to, while the latter needs another *expression*, which represents the data to be transferred in the channel.

Finally, in order to make the purposes of this extension of the language possible, *threaded method invocations* should be introduced. Without altering the backbone of the language, the framework is able to run a threaded instance of a method, without the need of a different method declaration. This can be achieved by adding the "@" symbol right after the *identifier* of the grammar of *method invocation*.

```
<channel type>                  ::= CHANNEL LT <class type>
                                          LSBRACE <class type> RSBRACE GT
                                 |  CHANNEL LT <channel type>
                                          LSBRACE <class type> RSBRACE GT
<channel creation expression>   ::= NEW <channel type> LPAREN RPAREN
<channel receive expression>    ::= REC <unary expression>
<channel send expression>       ::= <unary expression> REC expression
<threaded method invocation>    ::= <IDENTIFIER> AT LPAREN <argument list>? RPAREN
                                 | <primary> DOT <IDENTIFIER>
                                      AT LPAREN <argument list>? RPAREN
```

Figure 4.16: Channels grammar

While the above introduce new rules to the language, these rules need to be injected in some key points of the "Backbone". Therefore, a *primitive type* can also be a *channel type*, a *primary no new array* can also be a *channel creation expression* and both *statement expressions* and *assignment expression* can also be *channel receive expressions* or *channel send expressions*. Finally, a *method invocation* can also be a *threaded method invocation*, ending the injection phase of the two grammars.

```
%completing previous rules%

<type>                      ::= %%
                             | <channel type>
<primary no new array>      ::= %%
                             | <channel creation expression>
<statement expression>      ::= %%
                             | <channel receive expression>
                             | <channel send expression>
<assignment expression>     ::= %%
                             | <channel receive expression>
                             | <channel send expression>
<method invocation>         ::= %%
                             | <threaded method invocation>
```

Figure 4.17: Channels injection grammar

4.3.3. Policies

The Policies grammar as described in the introduction of this chapter, is responsible for denoting a class as a private data type. Therefore, it needs the name of that *class type*, along with the groups of classes and their corresponding permissions. This is described under the token *policy actions*, where each *policy action* must have a *class type*, a set of permissions (*actions*) and maybe an *internal policies* list. The latter is a set of more *policy actions*, in order to create a hierarchical group of classes that each has a different set of permissions. The syntax of "Policies" follows a similar grammar with the one described in the Privacy calculus [8].

```
<policy declaration>    ::= POLICY <class type> LBRACE <policy actions> RBRACE
<policy actions>        ::= <policy action>
                          | <policy actions> <policy action>
<policy action>         ::= <class type> LBRACE <actions> RBRACE
                                      <internal policies>? SEMICOLON
<internal policies>     ::= LSBRACE <policy actions>? RSBRACE

<actions>               ::= <action>
                          | <actions> <action>
```

Figure 4.18: Policies grammar part 1 of 2

The *actions* that can be declared under a policy are the permissions named in 2.2.3. There are also some peculiar types of *actions* called *special actions*. These *special actions* not only they need their corresponding name, but they also need to declare a *type*. Specifically, the IDENTIFY action needs to declare a *type* which will be checked in the Semantic Analysis (Section 4.5) whether it describes a private data class or not. The USAGE action needs to define a *type* and an *identifier*, in order to check that only a particularly named final variable of specific type was used along this permission. Finally, the DISSEMINATE action needs to define a *class type*, which will define the class that can have the dissemination.

```
<action>            ::=  READ
                       | UPDATE
                       | REFERENCE
                       | STORE
                       | READID
                       | AGGREGATE

<special action>    ::= IDENTIFY LBRACE <type> RBRACE
                      | USAGE LBRACE type <IDENTIFIER> RBRACE
                      | DISSEMINATE <class type>
```

Figure 4.19: Policies grammar part 2 of 2

Similarly to the “Channels” grammar, this grammar is also infused to the “Backbone” of the language. This time under only one rule, the *type declaration*. Again, it is assumed that rules that are empty, followed by the comment notation “%%” indicate that more rules exist in previous impressions of the same token.

```
%completing previous rules%

<type declaration>        ::= %%
                          | <policy declaration>
```

Figure 4.20: Policies injection grammar

4.4. Class Hierarchy

In order to allow the compiler to build the Abstract Syntax Tree, as mentioned in Section 2.4, JastAdd uses an .ast file which denotes the hierarchy of the classes that will constitute the tree. Therefore, investing in a structural hierarchy, not only does the grammar become more readable, but also the simplicity of its extensibility is escalating. Again, this is easier to visualize in the three abstractions that constitute the language, as well as the 6 part division of the “Backbone” of the language.

Similarly to the Syntax Analysis (Section 4.3), diagrams can give a brief summary of how the class hierarchy is formed. These can be found in Appendix B.

The notation of the .ast file also uses the symbol “::=”, which denotes that the class on the left side is the parent, while it consists of the classes that are stated on the right side of the symbol. Moreover, the symbol “:” denotes that the class on its left extends the class to the right side of the symbol, in the context of Java’s Object Orientation. This means that the class inherits the types of the needed children of its parent class (Object Oriented parent and not the Abstract Syntax parent). The same context of Java’s Object Orientation applies for the classes that are declared as abstract. For this representation, let us assume three things:

1) A star “*” represents a list of that particular class type.
2) Anything between <angle brackets> represents a token from the Lexical Analysis.

3) Any class between [square brackets] means that the class on the left of the equation can be described without the absolute need of having a child of that class.

4.4.1. Backbone

Following the same partitioning of the backbone's grammar, as described in Section 4.3.1, the class hierarchy is as follows.

4.4.1.1. Program

This notation denotes that the *Program* object consists of a list of *MyFile* objects, where each of them consists of a String called *Identifier*, which acts as the file name and a list of *TypeDeclaration* instances. JastAdd by default, holds the line in each file where each construct of non terminals was found. Therefore, having in mind the name of the file as well as the line of each non terminal, it can help on printing the stack trace in case of an error.

```
Program         ::= MyFile*;
MyFile          ::= <Identifier:String> TypeDeclaration*;
```

Figure 4.21: Program hierarchy

4.4.1.2. Declarations

In this section, although the *ClassDeclaration* class inherits the *TypeDeclaration* class which has no children, the former has two. These children *ClassType* and *ClassBody* denote the type and the body of the class respectively. The *UnknownClassDeclaration* class which inherits the *ClassDeclaration* class is just a helping class, responsible for allowing the compiler to understand an error, in cases where a class was not declared at the input.

```
TypeDeclaration                                          ;
ClassDeclaration             : TypeDeclaration          ::= ClassType ClassBody;
UnknownClassDeclaration      : ClassDeclaration         ;
```

Figure 4.22: Declaration hierarchy part 1 of 5

What is more, a *ClassBody* class consists of a list of *ClassBodyDeclaration* instances. The latter is an abstract class, following the Java meaning of abstract classes. This means that it cannot exist on its own but classes that inherit it also inherit its basic functionalities. This is useful as well for the *ClassMemberDeclaration* class which is also abstract and inherits the *ClassBodyDeclaration* class. Furthermore, similarly to the *UnknownClassDeclaration* class, the *UnknownDeclaration* class which inherits the *ClassMemberDeclaration* class, exists only for the purposes of the compiler to throw an error if the appropriate declaration does not exist. Following, the *NoNeededDeclaration* class, which again inherits the *ClassMemberDeclaration*, exists for the exact opposite reason. That is when a variable has no need for a declaration, thus its absence must not trigger an error. This can be useful for referencing the "this" object in some classes.

```
ClassBody                                                     ::= ClassBodyDeclaration*;
abstract ClassBodyDeclaration                                     ;
abstract ClassMemberDeclaration    : ClassBodyDeclaration         ;
NoNeededDeclaration                : ClassMemberDeclaration       ;
UnknownDeclaration                 : ClassMemberDeclaration       ;
```

Figure 4.23: Declaration hierarchy part 2 of 5

Moreover, in order to declare constructors under a class, the *ConstructorDeclaration* class is used. This class inherits the *ClassMemberDeclaration* class and consists of a *ConstructorDeclarator* and a *ConstructorBody* class. The former is responsible for the information about the constructor, thus it consists of a String *Identifier* and a list of *FormalParameter* instances. *FormalParameter* objects are also inheriting the *ClassMemberDeclaration* class but they consist of a *Type* and a *VariableDeclaratorId* class. On the other hand, the *ConstructorBody* class inherits the *Block* class and consists of a list of *BlockStatement* objects, which both will be explained in the next subsections. Moreover, the *ConstructorBody* can also have an optional child of an *ExplicitConstructorInvocation* class. This class consists of a list of *Expression* classes, which will be representing the arguments of an *ExplicitConstructorInvocation* instance.

```
ConstructorDeclaration         : ClassMemberDeclaration    ::= ConstructorDeclarator ConstructorBody;
ConstructorDeclarator                                      ::= <Identifier:String> FormalParameter*;

FormalParameter                : ClassMemberDeclaration    ::= Type VariableDeclaratorId;
ConstructorBody                : Block                     ::= [ExplicitConstructorInvocation] BlockStatement*;
ExplicitConstructorInvocation                              ::= Expression*;
```

Figure 4.24: Declaration hierarchy part 3 of 5

Furthermore, as classes in π-val must be capable of including fields, the *FieldDeclaration* class also inherits the *ClassMemberDeclaration* class, while it consist of a *Type* and a list of *VariableDeclarator* instances. This way, each variable declaration of a field can be matched with a type in the Semantic Analysis (Section 4.5). As an expansion of the language, the *FinalFieldDeclaration* class extends the *FieldDeclaration* class, which its only purpose is to declare a field that cannot change its value. Also, the *VariableDeclarator* and the *VariableDeclaratorId* classes inherit the *ClassMemberDeclaration* class. The former has a mandatory child of a *VariableDeclaratorId* and an optional *Initializer* instance, while the latter has a mandatory String *Identifier* for defining its name. The *Initializer* class will be discussed in the next paragraphs.

```
FieldDeclaration          : ClassMemberDeclaration       ::= Type VariableDeclarator*;
FinalFieldDeclaration     : FieldDeclaration             ;

VariableDeclarator        : ClassMemberDeclaration       ::= VariableDeclaratorId [Initializer];
VariableDeclaratorId      : ClassMemberDeclaration       ::= <Identifier:String>;
```

Figure 4.25: Declaration hierarchy part 4 of 5

Finally, in π-val there is a capability of adding methods under classes. Therefore, *MethodDeclaration* class also extends the *ClassMemberDeclaration* class. Similarly to the constructor, it consists of a *MethodHeader* and a *MethodBody* class. The former consists of a *MethodHeader*, which is responsible for defining the method's name and type. Therefore it consists of a *Type* and a *MethodDeclarator* class, which consists of a String *Identifier*, defining the name of the method, and a list of *FormalParameter* objects. On the other hand, the *MethodBody* class consists of a *Block* object.

```
MethodDeclaration      :ClassMemberDeclaration      ::= MethodHeader MethodBody;
MethodHeader                                        ::= Type MethodDeclarator;
MethodDeclarator                                    ::= <Identifier:String> FormalParameter*;
MethodBody                                          ::= Block;
```

Figure 4.26: Declaration hierarchy part 5 of 5

4.4.1.3. Types

In this section, classes concerning the type system of the language will be explained. Firstly, all classes ultimately inherit the abstract *Type* class. Therefore, any of the following classes can be held by an object that consists of *Type* objects. What is more, the existence of the abstract categories of types, *PrimitiveType* extending *Type*, *NumericType* extending *PrimitiveType*, *IntegeralType* extending *NumericType* and *FloatingPointType* extending *NumericType* are also defined. The presence of a *VoidType*, *NullType* and an *UnknownType* which extend the *Type* class, allows the language to have void methods, null objects, or even recognize the absence of a type declaration.

Moreover, the *ArrayType* class, which again extends the *Type* class, consists of a *Type* object, resulting in allowing any Object that extends the *Type* class to belong to an array (as long as it is allowed by the semantics of the grammar, described in Section 4.5). This even applies on other *ArrayType* objects, therefore allowing multidimensionality. Furthermore, the *ClassType* class, which extends the *PrimitiveType* class, defines the types that the program declares. Therefore it needs a String *Identifier* in order to name those types. This class is extended by the *CastType* class.

Finally, the *BooleanType* class extends the abstract *PrimitiveType* class, the *IntType* class and the *CharType* class extend the abstract *IntegralType* class, while the *FloatType* class and the *DoubleType* class extend the abstract *FloatingPointType* class.

```
abstract Type                                        ;
VoidType                      : Type                 ;
NullType                      : Type                 ;
UnknownType                   : Type                 ;
abstract PrimitiveType        : Type                 ;

ArrayType                     : Type                 ::= Type;
ClassType                     : PrimitiveType        ::= <Identifier:String>;
CastType                      : ClassType            ;
BooleanType                   : PrimitiveType        ;

abstract NumericType          : PrimitiveType        ;
abstract IntegralType         : NumericType          ;
IntType                       : IntegralType         ;
CharType                      : IntegralType         ;

abstract FloatingPointType    : NumericType          ;
FloatType                     : FloatingPointType    ;
DoubleType                    : FloatingPointType    ;
```

Figure 4.27: Types hierarchy

4.4.1.4. Blocks and Commands

As mentioned before, the *Block* instance can be used as the body of a method, a constructor, a branch or a loop. This means that the *Block* class must inherit the abstract class *StatementWithoutTrailingSubstatement*. The same principle applies for the *ReturnStatement*, the *ExpressionStatement*, the *StatementExpression*, the *EmptyStatement*, the *BreakStatement* and the *ContinueStatement* classes. This allows the grammar to find it easier to construct a *StatementWithoutTrailingSubstatement* object, as it can be any of the previous. What is more, the *Block* class consists of a list of *BlockStatement* instances, which will be discussed in the next paragraph. Following, the *ReturnStatement* consists of an optional *Expression* instance, thus allowing the construction of the grammar rule where the return can match both a typed and a void method. Finally, the *ExpressionStatement* consists of a *StatementExpression* and the latter consists of a *VariableInitializer*.

```
Block                   : StatementWithoutTrailingSubstatement       ::= BlockStatement*;
ReturnStatement         : StatementWithoutTrailingSubstatement       ::= [Expression];
ExpressionStatement     : StatementWithoutTrailingSubstatement       ::= StatementExpression;
StatementExpression     : StatementWithoutTrailingSubstatement       ::= VariableInitializer;
EmptyStatement          : StatementWithoutTrailingSubstatement       ;
BreakStatement          : StatementWithoutTrailingSubstatement       ;
ContinueStatement       : StatementWithoutTrailingSubstatement       ;
```

Figure 4.28: Blocks and Commands hierarchy part 1 of 4

The *LocalVariableDeclaration* class consists of a *Type* class and a list of *VariableDeclarator* objects, while extending the abstract class *LocalVariableDeclarationStatement*, which in turn extends the abstract class *BlockStatement*. As a result, any local variable declaration attempt is considered to be a *BlockStatement* instance.

```
abstract BlockStatement                                        ;
abstract LocalVariableDeclarationStatement: BlockStatement     ;
LocalVariableDeclaration   : LocalVariableDeclarationStatement ::= Type VariableDeclarator*;
```

Figure 4.29: Blocks and Commands hierarchy part 2 of 4

For the same purpose of considering *BlockStatement* instances, the abstract *Statement* class extends the *BlockStatement*, the abstract *StatementNoShortIf* extends the *Statement* and finally the abstract *StatementWithoutTrailingSubstatement* extends the *StatementNoShortIf*.

```
abstract Statement                                  : BlockStatement       ;
abstract StatementNoShortIf                         : Statement            ;
abstract StatementWithoutTrailingSubstatement       : StatementNoShortIf   ;
```

Figure 4.30: Blocks and Commands hierarchy part 3 of 4

Finally, the *IfThenStatement* class as well as the *WhileStatement* class, both extend the *Statement* class and consist of an *Expression* and a *Statement* object. The *IfThenElsStatement* class is similar to the *IfThenStatement* class, but consists of the same children and another *Statement* instance. Finally, the *IfThenElseStatementNoShortIf* and the *WhileStatementNoShortIf* classes are similar to the *IfThenElsStatement* and the *WhileStatement* classes respectively, as they need the same instances of children, but extend the *StatementNoShortIf* class instead of the *Statement* class.

```
IfThenStatement                 : Statement           ::= Expression Statement;
IfThenElseStatement             : Statement           ::= Expression Statement Statement;
IfThenElseStatementNoShortIf    : StatementNoShortIf  ::= Expression Statement Statement;
WhileStatement                  : Statement           ::= Expression Statement;
WhileStatementNoShortIf         : StatementNoShortIf  ::= Expression Statement;
```

Figure 4.31: Blocks and Commands hierarchy part 4 of 4

4.4.1.5. Expressions

There exists an abstract *Initializer* class, in order to be inherited by anything that can initialize a variable. There also exists an abstract *VariableInitializer* and an *ArrayInitializer* class. While they both extend the *Initializer* class, the latter consists of a list of *Initializer* instances.

```
abstract Initializer                                ;
abstract VariableInitializer      : Initializer     ;
ArrayInitializer                  : Initializer     ::= Initializer*;
```

Figure 4.32: Expressions hierarchy part 1 of 3

As expected, the abstract *Expression* class extends the *VariableInitializer* class and as the abstract *AssignmentExpression* class is also an expression, it extends the *Expression* class. Finally, the *Assignment* class extends the *AssignmentExpression* class, while it consists of a *PrimaryOld* instance which will be discussed later, a String *Op* for the operand that will be used (say for example the equals "=" operand) and another *AssignmentExpression* instance. This gives the option for multiple assignments in a single expression if allowed by the grammar.

```
abstract Expression               : VariableInitializer         ;
abstract AssignmentExpression     : Expression                  ;
Assignment                        : AssignmentExpression
                                         ::= PrimaryOld <Op:String> AssignmentExpression;
```

Figure 4.33: Expressions hierarchy part 2 of 3

Moreover, by listing the *Expression* categories in the next figure with a chained extensibility between them, it is implied that all *Expressions* between the classes *ConditionalOrExpression* and *AdditiveExpression* have the same children as the

ConditionalExpression class. These are a *VariableInitializer*, a String *Op* denoting the Operand of the expression and another *VariableInitializer*. As explained later, the *Primary* class is an extension of the *VariableInitializer* class, thus it is allowed to be also part of these expressions. What is different is the *MultiplicativeExpression*, where although it follows the chained inheritance of the previous expressions, its third child is optional. This is beneficial for the grammar's structure, as it allows unary expression to be considered under this chained inheritance as well. As a result, the *UnaryExpression* class extends the *Expression* class, while it consists of a *VariableInitializer*. The same goes for the classes *PlusUnaryExpression* and *MinusUnaryExpression* which exist only to differenciate between positive and negative unary expressions. Finally, the *CastExpression* class extends the *UnaryExpression* class, consisting of the same child plus a *Type* instance, denoting the type that the *VariableInitializer* instance will be casted to.

```
ConditionalExpression          : AssignmentExpression
                                      ::=VariableInitializer <Op:String>VariableInitializer;
ConditionalOrExpression        : ConditionalExpression            ;
ConditionalAndExpression       : ConditionalOrExpression          ;
InclusiveOrExpression          : ConditionalAndExpression         ;
ExclusiveOrExpression          : InclusiveOrExpression            ;
AndExpression                  : ExclusiveOrExpression            ;
EqualityExpression             : AndExpression                    ;
RelationalExpression           : EqualityExpression               ;
ShiftExpression                : RelationalExpression             ;
AdditiveExpression             : ShiftExpression                  ;
MultiplicativeExpression       : AdditiveExpression
                                      ::=VariableInitializer <Op:String> [VariableInitializer];

UnaryExpression                : Expression                       ::= VariableInitializer;
PlusUnaryExpression            : UnaryExpression;
MinusUnaryExpression           : UnaryExpression;
CastExpression                 : UnaryExpression                  ::= Type VariableInitializer;
```

Figure 4.34: Expressions hierarchy part 3 of 3

4.4.1.6. Primaries

As mentioned before, Primaries will be used in various places of the grammar. These primaries are the handlers of variables, methods, as well as the creators of instances or arrays. As a result, the abstract *Primary* class extends the *VariableInitializer* class and it is extended by two subclasses, *PrimaryOld* and *PrimaryNew*. The former concerns any

Primary that has to do with already declared instances in π-val, while the latter is about creation of instances or arrays.

```
abstract Primary        : VariableInitializer      ;
abstract PrimaryOld     : Primary                  ;
abstract PrimaryNew     : Primary                  ;
```

Figure 4.35: Primaries hierarchy part 1 of 4

Beginning with the *PrimaryOld* extensions, the *MethodInvocation* class consists of an optional *Primary* instance, a String *Identifier* denoting the name of the method, and a list of *Expression* instances, which act as the arguments of the method to be invocated. Similarly, the *FieldAccess* class is exactly the same but lacks the list of *Expression* instances, as it concerns variables rather than methods. Moreover, the *PrimaryNoNewArray* class consists of an optional *VariableInitializer* instance. This allows the use of *"this"* in the grammar, while the class itself acts like a container of all the allowed primaries. Finally, The *ArrayAccess* class consists of a *PrimaryNoNewArray* instance and an *Expression* instance. This relation allows multidimensionality in accessing arrays.

```
MethodInvocation     : PrimaryOld    ::= [Primary] <Identifier:String> Expression*;
FieldAccess          : PrimaryOld    ::= [Primary] <Identifier:String>;
PrimaryNoNewArray    : PrimaryOld    ::= [VariableInitializer];
ArrayAccess          : PrimaryOld    ::= PrimaryNoNewArray Expression;
```

Figure 4.36: Primaries hierarchy part 2 of 4

Following with the *PrimaryNew* extensions, the *ClassInstanceCreationExpression* class consists of a *Type* class, denoting the type of the class to be created and a list of *Expression* instances, which act as the arguments of the constructor that will be invocated. Furthermore, the *ArrayCreationExpression* class, also consists of a *Type* class, denoting the type of the array to be created, but also needs a list of *DimExpr* instances and a list o *DimEmp* instances for its dimensions. The former's class consists of an *Expression* instance, therefore it is responsible for denoting the given dimension's index, while the latter's class does not contain anything, allowing the Java's concept of multidimensionality, where a multidimensional array is just an array consisting of other arrays (which can also be multidimensional).

```
ClassInstanceCreationExpression        : PrimaryNew    ::= Type Expression*;
ArrayCreationExpression                : PrimaryNew    ::= Type DimExpr* DimEmp*;
DimExpr                                : PrimaryNew    ::= Expression;
DimEmp                                                 ;
```

Figure 4.37: Primaries hierarchy part 3 of 4

Furhtermore, the abstract *Literal* class extends the *VariableInitializer* class and consists of a String *LiteralValue*. Finally, the classes *IntegerLiteral*, *FloatLiteral*, *BooleanLiteral*, *CharacterLiteral*, *StringLiteral* and *NullLiteral* extend the abstract class *Literal*.

```
abstract Literal        : VariableInitializer      ::= <LiteralValue:String>;
IntegerLiteral          : Literal                  ;
FloatLiteral            : Literal                  ;
BooleanLiteral          : Literal                  ;
CharacterLiteral        : Literal                  ;
StringLiteral           : Literal                  ;
NullLiteral             : Literal                  ;
```

Figure 4.38: Primaries hierarchy part 4 of 4

4.4.2. Channels

In the Channel expansion of the grammar, the classes that were used also need to be declared hierarchically. Therefore, following the same principles as in the previous sections, the *ChannelType* class extends the *PrimitiveType* class, while it consists of a *Type* instance and a *ClassType* instance. The former denotes the *Type* of private data that will travel through a channel of that *ChannelType*, while the latter stands for the owner of that channel.

Regarding the expressions section, the classes *ChannelSendExpression* and *ChannelReceiveExpression* both extend the *AssignmentExpression* class and consist of a *UnaryExpression* instance, denoting the channel which will be operated. The only difference comes when the *ChannelSendExpression* needs another child for its construction and that is the *Expression* instance, which represents the data that will travel through the channel.

Finally, with respect to the Primaries section, the *ChannelCreationExpression* class extends the *PrimaryNew* class, as it regards a constructor. It consists of only a *ChannelType* instance, as channels constructors have no arguments. Lastly, the *ThreadedMethodInvocation* class is just an extension of the standard *MethodInvocation* class.

```
/*Types*/
ChannelType                   : Type                   ::= Type ClassType;

/*Expressions*/
ChannelSendExpression         : AssignmentExpression ::= UnaryExpression  Expression;
ChannelReceiveExpression      : AssignmentExpression ::= UnaryExpression ;

/*Primaries*/
ChannelCreationExpression     :PrimaryNew             ::= ChannelType;
ThreadedMethodInvocation      :MethodInvocation       ;
```

Figure 4.39: Channels hierarchy

4.4.3. Policies

In the Policies expansion, there were introduced some more new classes in order to be used by the grammar. The *PolicyDeclaration* class extends the *TypeDeclaration* class which was mentioned before and it consists of a *ClassType* instance and a list of *PolicyAction* instances. The *UnknownPolicyDeclaration* class extending the *PolicyDeclaration* class, similar to the *UnknownDeclaration* class, exists only for the purposes of error handling.

The *PolicyAction* class on the other hand is a standalone class and consists of a *ClassType* instance, a list of *Action* instances and another list of *PolicyAction* instances, in order to hierarchically declare some more permissions for its contained classes.

Finally, the *Action* class consists of a String *Identifier*, defining its name. For the purposes of the existence of complicated actions, another class, the abstract *SpecialAction* was introduced, which extends the *Action* class and its children, but also needs a *Type* instance for its construction. This will become handy for the *Identify*, *Disseminate* and *Usage* classes, all three of which extend the *SpecialAction* class and its children, while the latter also needs a String *VarName* for its construction.

```
PolicyDeclaration              : TypeDeclaration        ::= ClassType PolicyAction*;
UnknownPolicyDeclaration       : PolicyDeclaration      ;

PolicyAction                                            ::= ClassType Action* PolicyAction*;
Action                                                  ::= <Identifier:String>;


abstract SpecialAction   : Action            ::= <Identifier:String> Type;
Identify                 :  SpecialAction    ;
Disseminate              :  SpecialAction    ;
Usage                    :  SpecialAction    ::= <Identifier:String> Type <VarName:String>;
```

Figure 4.40: Policies hierarchy

4.5. Semantic Analysis

As mentioned before, some characteristics of the grammar define various information about specific classes. Using the appropriate synthesized or inherited attributes on JastAdd [4, 6], these characteristics can be accessed anywhere in the Abstract Syntax Tree as long as there exists a reference to the instance of the node that needs to be processed. Consequentially, this means that even classes that implicitly do not have their characteristics defined by the grammar, can have them calculated by the characteristics of other classes that are either their ancestors, or descendants, recursively.

4.5.1. Basic aspects

For making a better classification on the calculated attributes, various aspects were defined, grouping similar attributes together for the three abstractions of the language. Some of the basic aspects concern all three abstractions of the language. The first one affects the calculation of the type of a variable, method, constructor or expression, called the TypeAnalysis aspect. The second one is about the comparison between different types, called the TypeComparison aspect. Moreover, these aspects are being used for making sure that π-val allows the same type primitive conversions as Java [1], therefore making the translation at the Code Generation (Section 4.6) easier. Another basic aspect is the one called NameResolution, which is responsible for calculating the name of a variable, method or constructor.

Furthermore, the PrettyPrint aspect is responsible for translating the Abstract Syntax Tree nodes into readable Java code. This aspect will be explained further in Section 4.6.

Based on the previous aspects, a new aspect called DeclAnalysis can be defined. DeclAnalysis is responsible for finding the existence of the declaration of a specific type, variable, method or constructor, based on their name. The way it searches for declarations, takes advantage of both the inherited and synthesized attributes of JastAdd. That is for every Block, ClassDeclaration, MethodDeclaration and ConstructorDeclaration instance, a synthesized lookup table is computed based on the statements, fields or formal parameters that belong to them. When a name needs to be searched, the aspect examines the first instance that has a computed lookup table in the parent hierarchy of the processed instance. If the name is not found, then the process repeats until there are no more instances in the parent hierarchy that have a lookup table. Finally, if the name exists under a lookup table, then the appropriate declaration instance is returned. Otherwise, it returns a new UnknownDeclaration object.

Finally, there exist some other minor but helpful aspects. Firstly, the InitCheck attribute is responsible for checking if a variable is initialized. Secondly, the Statics aspect checks if a statement is static or not, thus it can be checked later if it is not used properly. Last but not least, the StatementReturn aspect is responsible for checking if methods are guaranteed to return a value.

4.5.2. Finding errors

There exists an aspect dedicated for finding errors. The ErrorCheck aspect, by taking advantage of the previous aspects, as well as the ones explained in the next paragraphs, is responsible for finding potential errors under the semantics of the language and listing them along with the place they were invoked. These can be type mismatches, duplicate declarations, undeclared calls, missing return statements, wrong uses of break and continue, or even violations of final declarations.

Regarding the policy declarations, this aspect prohibits the declaration of two policies for the same class, as well as it demands that the hierarchy of the policy must not contain the same group more than once. Furthermore, private data classes must only declare non private data fields, while the declaration of a String id field is mandatory and constructors that can only include field assignments.

There are also some limitations concerning the methods and constructors of a program. In particular, methods cannot return private data types. Therefore, they cannot be used as alternates to the concept of channels. What is more, method invocations cannot occur on private data handlers. Moreover, both methods' and constructors' arguments must be private data free, unless the constructor is also creating private data. This means that fields derived from private data cannot be used as such. As an overall limitation regarding methods and constructors, is that objects that contain private data fields are also considered as a threat to the completeness of Privacy calculus when being passed as arguments and thus prohibited. Finally, private data fields cannot be assigned to non private data variables and therefore are unable to bypass the previous checks.

4.5.3. Expanding aspects

Things get more interesting, when adding the aspects of the Channel and Policy part. While some of them follow the same principles as the previous, there are two new aspects dedicated precisely for static type checking based on the Privacy Policy and Privacy calculus [8].

The first one, called classes, affects the helping classes that are going to be used along the language. The helping class that concerns the Channels part of the language is going to be used only by the generated output of the language, making it possible to interpret the meaning of channels in Java code. Therefore, any sending or receiving on a single channel will be done under the corresponding synchronized methods declared in that class. As expected, these methods will make the thread that wants to use a channel wait, until it becomes available.

On the other hand, the helping classes concerning the Policy part of the language are going to be used alongside with the ones contained in the Abstract Syntax Tree. These classes called "Rule" classes, are responsible for representing the various rules of Privacy calculus that were described in Section 2.2.3, as well as holding information about the hierarchy of the Abstract Syntax Tree nodes that invoked those rules. Furthermore, another class called “Group” is declared, responsible for the translation of those rules into new Abstract Syntax Tree nodes.

What is more, in order to use the above “Rule” classes for representing the Privacy calculus rules, another aspect was needed to be declared, which will be responsible for searching for those rules under all nodes of the Abstract Syntax Tree. This aspect, called rules, traverses the whole Abstract Syntax Tree, in order to find nodes that invoked any permissions. In order to keep track of them, this traverse uses a HashSet of Action instances, tracking any effort of reading, writing or checking private data, based on the permissions that are related to the Privacy calculus. Therefore, for every branch of the Abstract Syntax Tree, the HashSet of the parent node is either transformed into an ArrayList of “Rule” instances, or duplicated and sent to its children, in order to be considered further down. The process of duplication stops in places where there will be no more branches, or where the actions tracked so far do not concern the children of that particular node. In either case, the rule collection process is still carrying on, but with new HashSet instances. Finally, when the whole Abstract Syntax Tree has been traversed, the ArrayList of “Rule” instances of the root of the tree is returned, describing all the rules that were invoked in the whole program. This list has a formation similar to a tree, therefore it can be traversed later in order to cross-check its contents with the permissions that are indeed allowed by the policy, hierarchically. Any disagreement between them should invoke a Violation error.

The final aspect, called policyViolations, concerns the violations that happened throughout the program. It reads the above ArrayList of rules and using the “Group” class, it converts them into PolicyAction instances, exactly as they are described in the “Policies” grammar in Section 4.3.3. Therefore, each private data class that is affected by those rules, has them added into its own list of “Group” instances, hierarchically. This concludes to a simple traversal of the private data classes, which can give the full report of all the violations that occurred.

4.5.4. Finding violations

In order to check whether a group describes a PolicyAction instance that is considered a violation of the policy, a simple process is run, regarding the ArrayList that was generated by the policyViolations aspect. This process finds the Abstract Syntax Tree nodes that invoked some of the under examination group's rules, for each group in the hierarchy. These group's hierarchy also needs to follow the declared policy's hierarchy. If the policy declaration at the given step of the hierarchy contains an Action that is also contained in the group, then the Action is valid. Otherwise, the Action forms a violation and is added to an error list along with its hierarchy.

The only Action that is checked differently is the "Aggregate" action. For this case, rules regarding aggregation are named after a combination of the variable type, name and field that caused the aggregation, thus creating a unique rule name for each field. At the policyViolation aspect, this is taken into consideration, in order to determine how many fields of a single private data variable have been aggregated in a step of the policy hierarchy. The algorithm then scrapes the first occurrence of the aggregation for each variable in each step of the policy hierarchy and for every other occurrence that involves the same fields of the variable, while it invokes a real aggregation rule when a second field of the same variable is found to be triggering an aggregation.

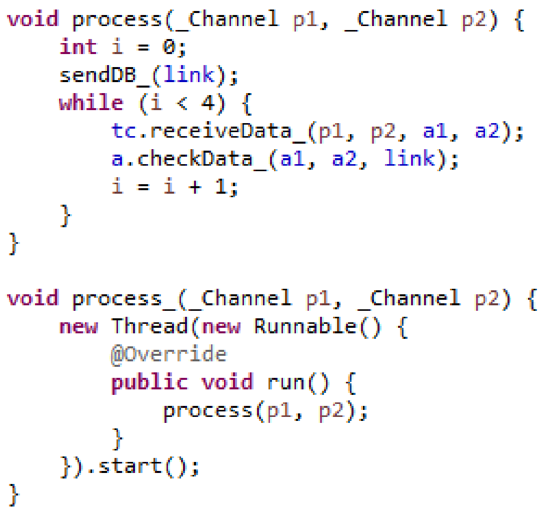
4.6. Code Generation

In this part of the compiler phases, if the under examination program has no errors, or policy violations, then its code should be generated. As this language is very similar to Java, the code generation becomes almost effortless regarding the "backbone" part, as the tokens are outputted almost identical to how they were inputted. This process regards a simple traversal of the Abstract Syntax Tree for printing its nodes.

Moreover, the "Policies" section of the language exists only for the purposes of validation, therefore it gets scrapped.

Finally, the only tricky part of the code generation process is the one concerning the ability of calling threaded methods. For each method that the compiler knows that it will be used as a new Thread, a new method is also generated to the final program, with a similar name to the original one. This new method creates a new Thread, that is responsible for calling the original method and runs it, as shown in Figure 4.41. If the method is not void, then a similar process is followed, using the Callable class that can return values that were computed in Threads.

This whole process results into having a full Java replica of the program which was originally written in π-val.



```
void process(_Channel p1, _Channel p2) {
    int i = 0;
    sendDB_(link);
    while (i < 4) {
        tc.receiveData_(p1, p2, a1, a2);
        a.checkData_(a1, a2, link);
        i = i + 1;
    }
}

void process_(_Channel p1, _Channel p2) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            process(p1, p2);
        }
    }).start();
}
```

Figure 4.41: Java representation of a threaded method

4.7. Extra Features

As π-val is based on Java, some of the basic functions of Java had to be incorporated. This leads to the need of introducing features that have nothing to do with the grammar of the language. As a result, a series of classes that are commonly used in Java programs should be implemented inside the framework. In order for any of these classes to be usable in π-val, a replica of them was defined, holding only the headers of their fields and methods.

One of the classes that were chosen to be incorporated to the language is the File class, allowing π-val to handle files just like Java. Moreover, the InputStream, OutputStream and PrintStream classes are included, as well as the Scanner and Object classes, making it easier for a π-val programmer to use basic Input/Output processes. All these give the ability to the programmer to use instances of those Objects, without the need of defining their class itself, just like they would have done in Java (e.g. creating a new File object and using methods on its instance, without declaring a File class).

What is more, the “length” field of any array is also declared in a similar way.

Moreover, the language allows System calls just like in Java [1]. This feature uses a static class in Java, but as the concept of statics does not exist in π-val, the word System is interpreted as an object of type System. Therefore, taking into consideration the previous inclusions, System.in, System.out and System.err are all valid for use, along with the System’s methods.

Finally, another extra feature, the π-val compiler gives the option to the user to compile multiple files or even a folder that contains files, by reading them as arguments.

Chapter 5

Case study

5.1. Test case scenario 54
5.2. Defining the policy 54
5.3. Writing the program 57
5.4. Errors and violations 63

5.1. Test case scenario

In this section, an example that was given in the “Typechecking Privacy Policies in the π-Calculus” paper [8], will be discussed. In that paper, a specific privacy policy was proposed for this particular scenario, but for the purposes of this thesis, there will be slight modifications, in order to adapt it to the framework’s compatibility. As a clarification, when developing a program on π-val it is not necessary to follow the same sequence of steps as the ones that are going to be shown in this chapter.

This example refers to a speed-control system which needs to check upcoming traffic, without violating the privacy of the drivers. The concept is that if a driver’s private data are stored, thus allowing the identification of the driver, while they are not exceeding the speed limit, then there is a privacy violation. This is because, as long as the driver is following the law, their personal data cannot be used in any way by the system.

5.2. Defining the policy

The paper proposes three privacy preservations:

1) Any data collected by the speed cameras must be used only for detecting speed-limit violations. Any other processing on those data must not be allowed.
2) The storage of any collected data must not be allowed, unless they evidence a speed-limit violation.

3) Any collected data that can identify the driver must remain hidden until a speed-limit violation is detected.

Moreover, the paper considers various groups, following the concept of π-calculus with groups [11, 3]. The first group refers to a system called SpeedControl, which is composed by car entities, thus creating the second group called Car. Furthermore, there is another group, defining the speed-control authority, called SCSystem, which is composed by speed camera entities, an authority for processing data and a database for storing data. These create three more groups called TrafficCam, Authority and DB respectively.

What is more, types should be declared, defining what data are considered private. Therefore, as the paper proposes, there should be three types of private data for this scenario, CarReg, CarSpeed and DriverReg. The former refers to the license plates of a car, while the latter for the license plates that belong to a driver. CarSpeed on the other hand considers the speed of a car.

Furthermore, having in mind the above propositions, as well as the grammar of π-val as mentioned in Chapter 4 and the rules that were set on the Privacy calculus [8], a policy regarding this scenario can be defined, as shown in Figure 5.1.

The private data CarReg, which refers to the car's plates can be seen by anyone who can view the car itself. Therefore, the car (Car) entity can STORE its license number or DISSEMINATE it to the speed control (SpeedControl). As a result, a traffic camera (TrafficCam) can gain access to a REFERENCE of this data (say for example by taking a photograph) and therefore DISSEMINATE it to the speed control system (SCSystem). The authority (Authority) can receive that REFERENCE and READ the actual private data (from the photograph). The authority might then want to IDENTIFY the owner of that car if it confirms that there was a traffic violation.

Similarly for the private data CarSpeed, which refers to the car's speed, it can be both STORED and UPDATED during the life cycle of a car (Car) process. This data can be DISSEMINATED to the speed control (SpeedControl) by various ways (say for example a speed radar). A traffic cam (TrafficCam) can then get a REFERENCE of that data and then DISSEMINATE it to the speed control system (SCSystem). The authority (Authority) can receive that REFERENCE and READ the actual private data. It can then USE that data against a constant type of data (double overLim), in order to find out if a violation has occurred and conclude in STORING the data regarding the violation.

Finally, the private data DriverReg, which refers to the license plate that a driver owns, can only be processed by the authority (Authority) and the database (DB). The latter can REFERENCE, STORE, or DISSEMINATE such data to the speed control system (SCSystem), while the former can REFERENCE, READ, or read the ID (READID) of such data. Furthermore, the Authority can AGGREGATE those data, in order to file the information of the driver that caused the violation.

```
policy CarReg{
    SpeedControl {}[
        Car { STORE, DISSEMINATE SpeedControl};
        SCSystem {}[
            TrafficCam { REFERENCE, DISSEMINATE SCSystem};
            Authority { REFERENCE, READ, IDENTIFY{DriverReg}};
            DB {};
        ];
    ];
}

policy CarSpeed{
    SpeedControl {}[
        Car { UPDATE, STORE, DISSEMINATE SpeedControl};
        SCSystem {}[
            TrafficCam { REFERENCE, DISSEMINATE SCSystem};
            Authority { REFERENCE, READ, STORE, USAGE{double overLim}};
            DB {};
        ];
    ];
}

policy DriverReg{
    SpeedControl {}[
        Car {};
        SCSystem {}[
            TrafficCam {};
            Authority { REFERENCE, AGGREGATE, READ, READID};
            DB {REFERENCE, STORE, DISSEMINATE SCSystem};
        ];
    ];
}
```

Figure 5.1: The privacy policy of the example

5.3. Writing the program

The paper also gives the operation of the program, in the form of the Privacy calculus syntax [8], as seen in Figure 5.2. Although this is not necessary for the writing of a program in π-val, it is a good practice, as it gives an overview of the programmer. All code quoting highlighting in this thesis was made using the online tool syntax-highlight-word by PlanetB [17].

The System contains a group called SpeedControl, which encloses the other groups with the nested processes C, SC, A and D. The hierarchy of the nested processes and groups is the same as in the previous policy declaration in Figure 5.1. The above processes communicate by sharing some names. In particular, process C and SC share the name p, which represents a channel for the photograph of the running vehicle that the traffic camera has taken. Moreover, the authority and the traffic camera share the name a, which is another channel for the same data. Finally, the names r_1, …, r_n are used for the communication between the authority and the database.

$$
\begin{array}{rcl}
\mathit{System} & = & \mathsf{SpeedControl}[\ \mathsf{Car}[C] \parallel \mathsf{SCSystem}[\mathsf{trafficCam}[SC] \parallel \mathsf{Auth}[A] \parallel \mathsf{DBase}[D]\ ]] \\
C & = & (\nu\ r)(\nu\ s)(\overline{r} \triangleright [\mathsf{id} \otimes \mathit{reg}] \mid \overline{s} \triangleright [\mathsf{id} \otimes \mathit{speed}] \\
 & \mid & *cs?(y).\, s!\langle \mathsf{id} \otimes y \rangle.\, \mathbf{0} \\
 & \mid & *p!\langle r, s \rangle.\, \mathbf{0}) \\
SC & = & *p?(x, y).\, a!\langle x, y \rangle.\, \mathbf{0} \\
A & = & *a?(k_1, k_2).\, k_2?(\_ \otimes z).\, \mathtt{if}\ (z > \mathit{overLim})\ \mathtt{then}\ V\ \mathtt{else}\ \mathbf{0} \\
V & = & k_1?(\_ \otimes y). \\
 & & (r_1?(x \otimes w).\, \mathtt{if}\ w = y\ \mathtt{then}\ (\nu\ r)(\overline{r} \triangleright [x \otimes z])\ \mathtt{else}\ \mathbf{0} \\
 & \mid & \ldots \\
 & \mid & r_n?(x \otimes w).\, \mathtt{if}\ w = y\ \mathtt{then}\ (\nu\ r)(\overline{r} \triangleright [x \otimes z])\ \mathtt{else}\ \mathbf{0}) \\
D & = & \overline{r_1} \triangleright [\mathsf{id}_1 \otimes \mathsf{reg}_1] \mid \ldots \mid \overline{r_n} \triangleright [\mathsf{id}_n \otimes \mathsf{reg}_n]
\end{array}
$$

Figure 5.2: Basis of program in Privacy calculus

Now that the policy is written in Figure 5.1, and the basis of the program is ready in terms of the Privacy calculus in Figure 5.2, the groups that are defined must be matched to their corresponding classes in the π-val program. Therefore, the programmer needs to declare classes with the same names as the groups, as well as for the private types that the policy refers to. The latter is shown in Figure 5.3, while the other classes will be explained in the next paragraphs. As mentioned in Section 2.2.3, private data must contain an id along with the information that they hold, thus an id field is demanded on their class declaration.

```
class CarReg{
    String id;
    String reg;

    CarReg(String id,String reg){
        this.id=id;
        this.reg=reg;
    }
}

class CarSpeed{
    String id;
    double speed;

    CarSpeed(String id,double speed){
        this.id=id;
        this.speed=speed;
    }
}

class DriverReg{
    String id;
    String reg;

    DriverReg(String id,String reg){
        this.id=id;
        this.reg=reg;
    }
}
```

Figure 5.3: Private data classes

Moreover, the program in Figure 5.2 shows that in the Privacy calculus, the car process C, STORES two private data, the registration number (reg) and the speed (speed). The latter changes dynamically as shown in the second line of the process, via an external channel called cs. Finally, through the channel p, these information can be sent to the SC process. These can be easily transformed into π-val as seen in Figure 5.4, where the private data are stored as fields of private data types and the sub-processes are declared as methods which use channels. Note that the channels must declare to whom they should belong to.

```
class Car{
    CarReg r;
    CarSpeed s;

    Car(String id,String reg,double speed){
        this.r = new CarReg(id,reg);
        this.s = new CarSpeed(id,speed);
    }

    void updateSpeed(Channel<CarSpeed[Car]> cs){
        CarSpeed y = <-cs;
        if(y!=null){
            s.speed=y.speed;
        }
    }

    void sendData(Channel<CarReg[SpeedControl]> p1,
                  Channel<CarSpeed[SpeedControl]> p2){
        p1<-r;
        p2<-s;
    }
}
```

Figure 5.4: The Car class

Similarly, the database process D stores a series of private data which represent the registration numbers that belong to various drivers and tries to send them over the names r_1, …, r_n. Again, the same can be translated into π-val with minimal effort, as shown in Figure 5.5. The private data are stored as fields, while their transmission is represented by the method sendReg(Channel), which can be called in a parallel to the program infinite loop by an instance. This instance can be the owner of a bunch of instances of DB objects, thus illustrating a complete database.

```
class DB{
    String name;
    DriverReg r1;

    DB(String id,String name,String reg){
        this.r1=new DriverReg(id,reg);
        this.name=name;
    }

    void sendReg(Channel<DriverReg[SCSystem]> link){
            link<-this.r1;
    }
}
```

Figure 5.5: The DB class

Furthermore, the process of the traffic cam (SC) is responsible for receiving the information of a car from channel p, and then trying to send them over the channel a. This again is easy to be transformed into π-val, where both channels are split into two parts each, as shown in Figure 5.6.

```
class TrafficCam{
    void receiveData(Channel<CarReg[SpeedControl]> p1,
                            Channel<CarSpeed[SpeedControl]> p2,
                                 Channel<CarReg[SCSystem]> a1,
                                   Channel<CarSpeed[SCSystem]> a2){
        a1<-<-p1;
        a2<-<-p2;
    }
}
```

Figure 5.6: The TrafficCam class

What is more, the process A receives those information from channel a and binds the speed of the car to the name z, using the channel k2. Without reading the corresponding id, it checks if the name z is greater than the constant overLim. If so, the process V is run. This can be seen in π-val in the first part of the function checkData() in Figure 5.7.

Furthermore, when the process V runs (which is still part of the authority), it binds the registration number of the car to the name y, using the channel k1. It then receives from the channels r_1, …, r_n all the records that are contained in the database (process D). If there is a match of registration numbers, then the process stores the private data that correspond to the speed that led to the violation, along with the id of the owner of the car. This operation is quite different in the π-val translation, as seen on the last part of the function checkData() in Figure 5.7. The new private data that holds the violation details is stored in a dynamic array. In order to implement such array, the method expandViolationsTable() was used.

```
class Authority{
    final double overLim=30;
    CarSpeed[] violations = new CarSpeed[0];
    Channel<CarSpeed[Authority]> v =
                         new Channel<CarSpeed[Authority]>();

    void checkData(Channel<CarReg[SCSystem]> a1,
                    Channel<CarSpeed[SCSystem]> a2,
                      Channel<DriverReg[SCSystem]>[] links){
        CarReg k1= <-a1;
        CarSpeed k2= <-a2;
        if(k2.speed  > overLim){
            int i=0;
            while(i<4){
                DriverReg dr= <-links[i];
                if(dr.reg == k1.reg){
                    System.out.println("match");
                    expandViolationsTable();
                    violations[violations.length-1]
                         =new CarSpeed(dr.id, k2.speed);
                }
                i=i+1;
            }
        }
    }

    void expandViolationsTable(){
        CarSpeed[] temp = new CarSpeed[violations.length+1];
        int i=0;
        while(i<violations.length){
            temp[i]=violations[i];
            i=i+1;
        }
        violations=temp;
    }
}
```

Figure 5.7: The Authority class

Although the Privacy calculus does not define any processes for some groups, in π-val their classes should be implemented. This concerns the SCSystem and the SpeedControl. What is more, these classes are cruicial for the program, as they constitute the main program.

The SCSystem group, as defined by the privacy calculus policy in Figure 5.2, is holding information about the groups trafficCam, Auth and DBase, as well as for the processes SC, A and D. As a result, in the π-val representation, the SCSystem class is the one that holds instances of DB objects, along with their respective channel. This enables it to send the information of its database through those channels, using the method sendDB(). Moreover, another method, called process, is responsible for calling in a parallel thread the method sendDB(), as well as sequentially receive data from the channels of the traffic cam and redirecting them to the channels of the authority. This is also done by calling the corresponding processes of those instances in parallel threads.

```
class SCSystem{
        Channel<CarReg[SCSystem]> a1 = new Channel<CarReg[SCSystem]>();
        Channel<CarSpeed[SCSystem]> a2 = new Channel<CarSpeed[SCSystem]>();

        Channel<DriverReg[SCSystem]> link1 = new Channel<DriverReg[SCSystem]>();
        Channel<DriverReg[SCSystem]> link2 = new Channel<DriverReg[SCSystem]>();
        Channel<DriverReg[SCSystem]> link3 = new Channel<DriverReg[SCSystem]>();
        Channel<DriverReg[SCSystem]> link4 = new Channel<DriverReg[SCSystem]>();
        Channel<DriverReg[SCSystem]>[] links = {link1,link2,link3,link4};

        DB d1 = new DB("1","driver 1","ABC 123");
        DB d2 = new DB("2","driver 2","BCD 234");
        DB d3 = new DB("3","driver 3","CDE 345");
        DB d4 = new DB("4","driver 4","DEF 456");
        DB drivers[] = {d1,d2,d3,d4};

        TrafficCam tc=new TrafficCam();
        Authority auth = new Authority();


        void sendDB(){
            int i=0;
            while(true){
                drivers[i].sendReg(links[i]);
                i=(i+1)%drivers.length;
            }
        }

        int process(Channel<CarReg[SpeedControl]> p1,
                         Channel<CarSpeed[SpeedControl]> p2){
            int i=0;
            sendDB@();
            while(i<4){
                tc.receiveData@(p1,p2,a1,a2);
                auth.checkData@(a1,a2,links);
                i=i+1;
            }
            return 0;
        }
}
```

Figure 5.8: The SCSystem class

Finally, the last class, SpeedControl, is the one that "simulates" the presence of cars and thus holding their instances. Through a method called run(), it makes the cars run their process of sending data in parallel threads, as well as it starts the speed control system process, again in a parallel thread. As all these constitute the main procedure of the whole program, the static main method is also declared in this class, which is only purpose is to call the run() method.

```
class SpeedControl{
       Channel<CarReg[SpeedControl]> p1=new Channel<CarReg[SpeedControl]>();
       Channel<CarSpeed[SpeedControl]> p2=new Channel<CarSpeed[SpeedControl]>();

       Car c1=new Car("10","ABC 123",200);
       Car c2=new Car("9","BCD 234",10);
       Car c3=new Car("8","CDE 345",20);
       Car c4=new Car("7","DEF 456",150);
       Car cars[]={c1,c2,c3,c4};

       SCSystem scs=new SCSystem();

       void run(){
              int i=0;
              while(i<cars.length){
                  cars[i].sendData@(p1,p2);
                  i=i+1;
              }
              scs.process@(p1,p2);
       }

       void main(String args[]){
              System.out.println("Outcome");
              SpeedControl system=new SpeedControl();
              system.run();
       }
}
```

Figure 5.9: The SpeedControl class

This completes the scenario in π-val and it is now ready for compiling. If there are any violations, the compiler will reject the program, otherwise it will create a new Java file named after the class that holds the main method (if there are more than one main methods, it takes the name of the first instance's class). For this example, there should be no errors. This file can be now run normally by a Java compiler. The output program can be found in the Appendix C.

5.4. Errors and violations

This framework is responsible for finding possible errors and policy violations in a given program. Although the former holds a significant part of the implementation of this framework and plays a big role for its validity, it is trivial to show such examples in this chapter. These are typical errors that someone can encounter in most programming languages.

On the other hand, policy violations are what give purpose to this framework and thus will be demonstrated. Firstly, if the program was run on the framework, against an empty policy declaration, the framework would give no message and proceed to the code generation process, as there are no private data classes defined, thus there can be no violations.

Secondly, let us assume that the policy declaration which was defined in Figure 5.1 was just denoting what classes are considered as private data, without showing the hierarchy of the program, as shown in Figure 5.10.

```
policy CarReg{
}

policy CarSpeed{
}

policy DriverReg{
}
```

Figure 5.10: Empty privacy policy without any hierarchy

If the example program that was defined in Section 5.3 was run on the framework, against this policy declaration, the framework would just give policy violation errors about not following the correct hierarchy of the policy.

These errors are shown in Figure 5.11, where the first part of each error denotes the file and the line that the violation was found on. For this missing hierarchy violation, the file and line which is shown is the one regarding the policy declaration itself. Following, is the error message that regards the violation. Finally, the hierarchy that was supposed to be found is shown between square brackets, along with the line numbers that indicate the trace of the code that invoked the violation.

```
Errors:
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 35, Car]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 36, Car]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 37, Car]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 38, Car]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 46, Car]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 91, TrafficCam]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 91, TrafficCam]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 1: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 35, Car]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 36, Car]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 37, Car]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 38, Car]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 46, Car]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 91, TrafficCam]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 91, TrafficCam]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 5: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 41, SCSystem: 69, DB]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 41, SCSystem: 70, DB]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 41, SCSystem: 71, DB]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 54, SpeedControl: 41, SCSystem: 72, DB]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 89, SCSystem: 82, DB]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
~carPolicy.pmj: 9: Policy Violation:    Under policy declaration could not find hierarchy: [SpeedControl: 55, SpeedControl: 49, SCSystem: 92, Authority]
```

Figure 5.11: Hierarchy violation errors

Finally, let us assume that the hierarchy is defined correctly, but no permissions were given on the groups, as shown in Figure 5.12.

```
policy CarReg{
    SpeedControl {}[
        Car {};
        SCSystem {}[
            TrafficCam {};
            Authority {};
            DB {};
        ];
    ];
}

policy CarSpeed{
    SpeedControl {}[
        Car {};
        SCSystem {}[
            TrafficCam {};
            Authority {};
            DB {};
        ];
    ];
}

policy DriverReg{
    SpeedControl {}[
        Car {};
        SCSystem {}[
            TrafficCam {};
            Authority {};
            DB {};
        ];
    ];
}
```

Figure 5.12: Hierarchical privacy policy without permissions

As a result, anything that regards private data, causes a policy violation. These violations can be viewed in Figure 5.13. Similarly to the previous error messages, the first part of each error denotes the file and the line of code that each violation has occurred. Following, it prints the name of the private data that regards that violation, along with the type of permission that was violated. Finally, it prints the trace of the code that invoked the violation.

```
Errors:
~car.mj: 104: Policy Violation:  CarReg STORE [SpeedControl: 54, SpeedControl: 35] Car: 104
~car.mj: 104: Policy Violation:  CarReg STORE [SpeedControl: 54, SpeedControl: 36] Car: 104
~car.mj: 104: Policy Violation:  CarReg STORE [SpeedControl: 54, SpeedControl: 37] Car: 104
~car.mj: 104: Policy Violation:  CarReg STORE [SpeedControl: 54, SpeedControl: 38] Car: 104
~car.mj: 116: Policy Violation:  CarReg DISSEMINATE SpeedControl [SpeedControl: 55, SpeedControl: 46] Car: 116
~car.mj: 124: Policy Violation:  CarReg DISSEMINATE SCSystem [SpeedControl: 55, SpeedControl: 49, SCSystem: 91] TrafficCam: 124
~car.mj: 124: Policy Violation:  CarReg REFERENCE [SpeedControl: 55, SpeedControl: 49, SCSystem: 91] TrafficCam: 124
~car.mj: 136: Policy Violation:  CarReg REFERENCE [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 136
~car.mj: 142: Policy Violation:  CarReg READ [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 142
~car.mj: 142: Policy Violation:  CarReg ~AGGREGATE [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 142
~car.mj: 105: Policy Violation:  CarSpeed STORE [SpeedControl: 54, SpeedControl: 35] Car: 105
~car.mj: 105: Policy Violation:  CarSpeed STORE [SpeedControl: 54, SpeedControl: 36] Car: 105
~car.mj: 105: Policy Violation:  CarSpeed STORE [SpeedControl: 54, SpeedControl: 37] Car: 105
~car.mj: 105: Policy Violation:  CarSpeed STORE [SpeedControl: 54, SpeedControl: 38] Car: 105
~car.mj: 117: Policy Violation:  CarSpeed DISSEMINATE SpeedControl [SpeedControl: 55, SpeedControl: 46] Car: 117
~car.mj: 125: Policy Violation:  CarSpeed DISSEMINATE SCSystem [SpeedControl: 55, SpeedControl: 49, SCSystem: 91] TrafficCam: 125
~car.mj: 125: Policy Violation:  CarSpeed REFERENCE [SpeedControl: 55, SpeedControl: 49, SCSystem: 91] TrafficCam: 125
~car.mj: 137: Policy Violation:  CarSpeed REFERENCE [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 137
~car.mj: 138: Policy Violation:  CarSpeed READ [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 138
~car.mj: 138: Policy Violation:  CarSpeed USAGE double [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 138
~car.mj: 138: Policy Violation:  CarSpeed ~AGGREGATE [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 138
~car.mj: 145: Policy Violation:  CarSpeed READ [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 145
~car.mj: 145: Policy Violation:  CarSpeed ~AGGREGATE [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 145
~car.mj: 145: Policy Violation:  CarSpeed STORE [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 145
~car.mj: 141: Policy Violation:  DriverReg REFERENCE [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 141
~car.mj: 142: Policy Violation:  DriverReg IDENTIFY CarReg [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 142
~car.mj: 142: Policy Violation:  DriverReg READ [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 142
~car.mj: 142: Policy Violation:  DriverReg ~AGGREGATE [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 142
~car.mj: 145: Policy Violation:  DriverReg AGGREGATE [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 145
~car.mj: 145: Policy Violation:  DriverReg READID [SpeedControl: 55, SpeedControl: 49, SCSystem: 92] Authority: 145
~car.mj: 168: Policy Violation:  DriverReg STORE [SpeedControl: 54, SpeedControl: 41, SCSystem: 69] DB: 168
~car.mj: 168: Policy Violation:  DriverReg STORE [SpeedControl: 54, SpeedControl: 41, SCSystem: 70] DB: 168
~car.mj: 168: Policy Violation:  DriverReg STORE [SpeedControl: 54, SpeedControl: 41, SCSystem: 71] DB: 168
~car.mj: 168: Policy Violation:  DriverReg STORE [SpeedControl: 54, SpeedControl: 41, SCSystem: 72] DB: 168
~car.mj: 173: Policy Violation:  DriverReg DISSEMINATE SCSystem [SpeedControl: 55, SpeedControl: 49, SCSystem: 89, SCSystem: 82] DB: 173
```

Figure 5.13: Violation errors regarding permissions

Chapter 6

Conclusions

6.1. Overall 68
6.2. Limitations 69
6.3. Future work 69

6.1. Overall

In this thesis, a framework was created, responsible for type checking a given system, regarding its privacy integrity, based on a specific policy statement. The framework was created using a meta-compilation system called JastAdd [4] and it follows the principles of formal methods [12] and the Privacy calculus [8], as described in Chapter 2. Moreover, in order to be able to construct such a framework, a new language called π-val was created, which allows the consideration of the previous principles into high-level programming.

As a result, π-val is able to express the declaration of classes, variables, methods and constructors, using Java's conventional types or even perform logical and arithmetical expressions upon them. Branches and loops are also available in the language, while it is incorporated with the notion of channels and threaded methods. Therefore, a program in π-val can be modelled to be very similar to a π-calculus program which uses parallel processes.

Finally, π-val allows the expression of a policy statement, following again the principles of the Privacy calculus. In this statement, all valid actions regarding the various processes' use of private data are being defined. Therefore, when the framework is run, it can detect whether or not there was a violation of the policy. If there was a violation with the combination of the initial program and a particular policy statement, the program is rejected. If the program is valid, then it is translated into Java code and can be run by a typical Java compiler.

6.2. Limitations

The approach for the framework of π-val's implementation tries to follow as much as possible the Privacy calculus' rules. Therefore, there is an attempt of following the same principles of π-calculus that bases the Privacy calculus. In order to do so, any interaction regarding private data between instances, must be narrowed down to the use of channels, as discussed in Section 4.5.2. These limitations that disallow any interaction without the use of channels, result in tightening the language's flexibility. Although it can be possible for some of the restrictions to be used in the language without violating any of the Privacy calculus' rules, it is better to throw an error when these constraints are met, until it is formally proved that they will not cause any breach on the completeness of the Privacy calculus. Nevertheless, the rule process is written in such a way, that it can detect the appropriate rules if they ever become enabled.

Finally, the language lacks the notion of polymorphism.

6.3. Future work

As mentioned in Section 6.1, the framework that was created for the purposes of this thesis covers the basic concepts of formal methods of privacy. Therefore, there exist many possible extensions that can be implemented in order to make the framework more powerful and compatible with the current industry's needs. This extensibility can be achieved easily, as the framework is developed in a meta-compilation system that allows straightforward additions of new concepts, while keeping unharmed the previous functionalities. This advantage of meta-compilation systems and especially of JastAdd, is discussed in Section 2.4 and 2.4.2.

Moreover, regarding the limitations of this framework, as mentioned in 6.2, some features of π-val have been disabled. This is because they are not following exactly the principles of Privacy calculus. Although the framework is able to operate along with those features enabled, further research can be made in order to prove their safeness. As a result, coding in π-val in the future can be even more similar to Java, as the use of channels might be replaced by methods who can transfer private data between processes.

What is more, in future work on this framework, the compiler of π-val could directly produce Java bytecode, in order to bypass the extra steps of having to run the exported Java program into a second compiler.

Furthermore, as privacy is diachronically considered as a debatable concept, it is still a topic under research. As a result in the future, there can exist more instances of formal methods for privacy, or even expansions of existing ones. This can lead to a vast variety of possible extensions for the current framework, as well as re-innovations of its ongoing features.

Finally, when the above will have been taken into consideration, this framework can find a great use in companies and governments, in order to ensure that their computer systems' software follow the current law. Of course, some preparation has to be made by the law enforcing authorities, in order to ensure that the law is indeed translated correctly into a policy declaration. Nevertheless, any new tool that is originally created for our convenience, needs to be thoroughly tested and constantly updated in order to fulfil our never-ending needs. Therefore, the cost of ensuring the validity of the translation of the law to a policy declaration is little compared to the benefits that this framework can provide to the preservation of the privacy of a human being.

Bibliography

[1] J. Alves-Foss, Formal syntax and semantics of Java, Springer Science & Business Media, 1999.

[2] D. Basin, S. Debois and T. Hildebrandt, "On purpose and by necessity: compliance under the GDPR," *Proceedings of Financial Cryptography and Data Security,* vol. 18, 2018.

[3] L. Cardelli, G. Ghelli and A. D. Gordon, "Secrecy and group creation," in *International Conference on Concurrency Theory*, 2000.

[4] T. Ekman and G. Hedin, "The JastAdd system—modular extensible compiler construction," *Science of Computer Programming,* vol. 69, pp. 14-26, 2007.

[5] M. Forsberg and A. Ranta, "The labelled bnf grammar formalism," *Department of Computing Science, Chalmers University of Technology and the University of Gothenburg,* 2005.

[6] G. Hedin, "An introductory tutorial on JastAdd attribute grammars," in *International Summer School on Generative and Transformational Techniques in Software Engineering*, 2009.

[7] G. Hedin and E. Magnusson, "JastAdd—an aspect-oriented compiler construction system," *Science of Computer Programming,* vol. 47, pp. 37-58, 2003.

[8] D. Kouzapas and A. Philippou, "Type checking privacy policies in the π-calculus," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems,* 2015.

[9] M. E. Lesk and E. Schmidt, *Lex: A lexical analyzer generator,* Bell Laboratories Murray Hill, NJ, 1975.

[10] S. Muchnick and others, Advanced compiler design implementation, Morgan kaufmann, 1997.

[11] D. Sangiorgi and D. Walker, The pi-calculus: a Theory of Mobile Processes, Cambridge university press, 2003.

[12] D. J. Solove, "A taxonomy of privacy," *U. Pa. L. Rev.,* vol. 154, p. 477, 2005.

[13] M. C. Tschantz and J. M. Wing, "Formal methods for privacy," in *International Symposium on Formal Methods*, 2009.

[14] S. Wachter, "Normative challenges of identification in the Internet of Things: Privacy, profiling, discrimination, and the GDPR," *Computer law & security review,* vol. 34, pp. 436-449, 2018.

[15] "Java Syntax Specification," [Online]. Available: http://cs.au.dk/ amoeller/RegAut/JavaBNF.html.

[16] "Metacompilation," 4 2019. [Online]. Available: https://en.wikipedia.org/wiki/Metacompilation.

[17] J. Beach, "Syntax Highlight Code in Word Documents," [Online]. Available: http://www.planetb.ca/syntax-highlight-word.

[18] "Software agent," 5 2019. [Online]. Available: https://en.wikipedia.org/wiki/Software_agent.

Appendix A

Below are the diagrams regarding the Syntax Analysis of π-val. Let us assume the following:

1. A white filled ellipse with a continuous outline represents a node in the Abstract Syntax Tree.
2. A dotted outlined ellipse represents an optional node. This means that the node which its child has a dotted outline, can be described without the absolute need of having a child of that type.
3. Yellow filled ellipses represent lists of the types of nodes that they describe. The star "*" symbol at the end of the class name represents the allowance of the emptiness of such a list.
4. Gray filled ellipses represent nodes that were described in a different diagram, but their functionality is being extended by this diagram.
5. Bold titled ellipses represent nodes that are described in detail in the same diagram.
6. Underlined titled ellipses represent nodes that are described in another diagram.
7. Straight arrow lines show the necessity of a child node in order to describe its parent.
8. Dotted arrow lines show that only one of the children with a dotted arrow line can describe their parent for each instance of the latter.

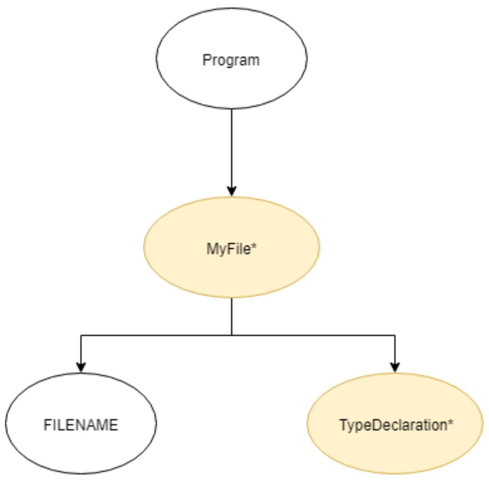


Figure A.1: Program grammar diagram

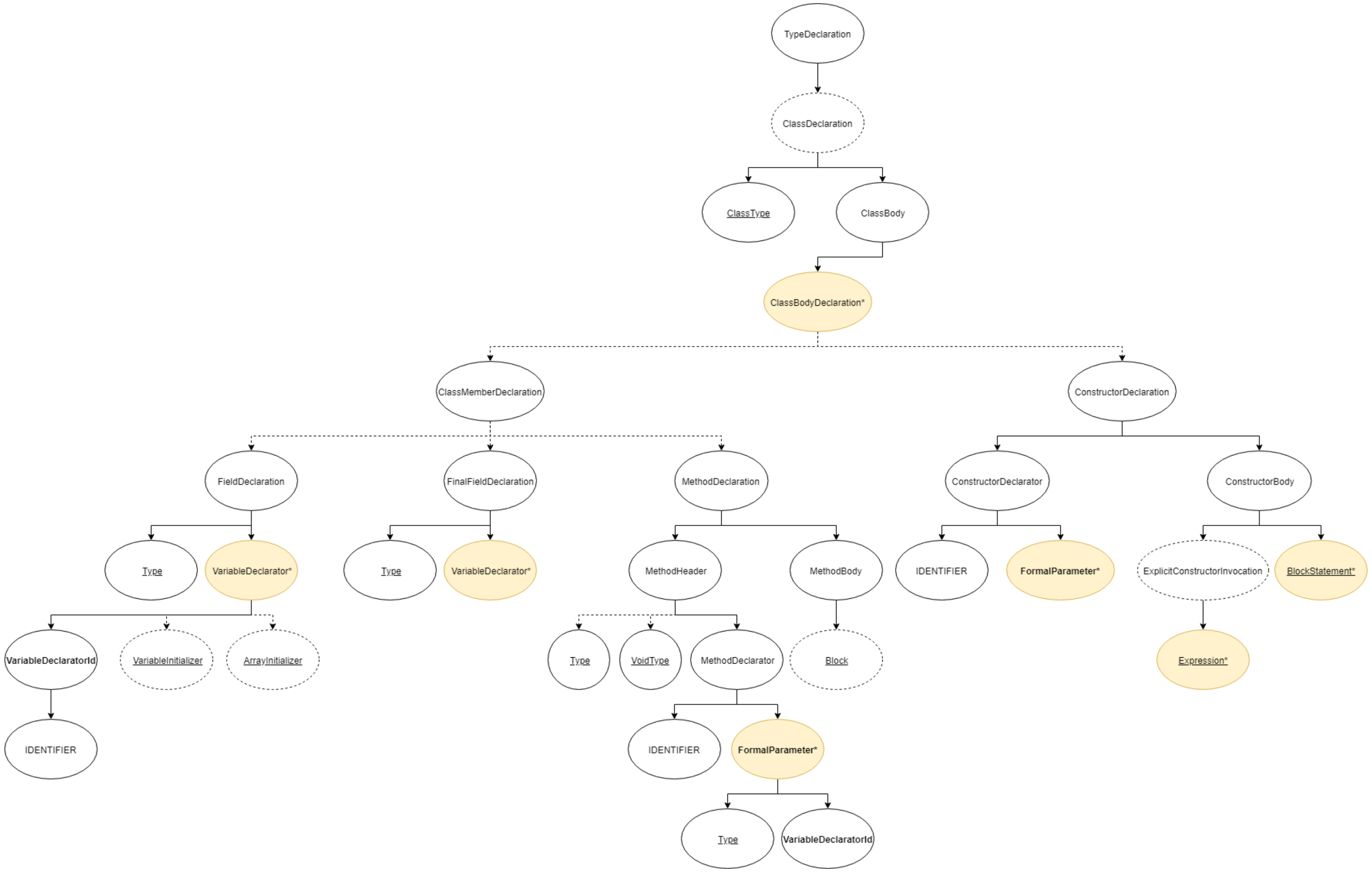


Figure A.2: Declarations grammar diagram

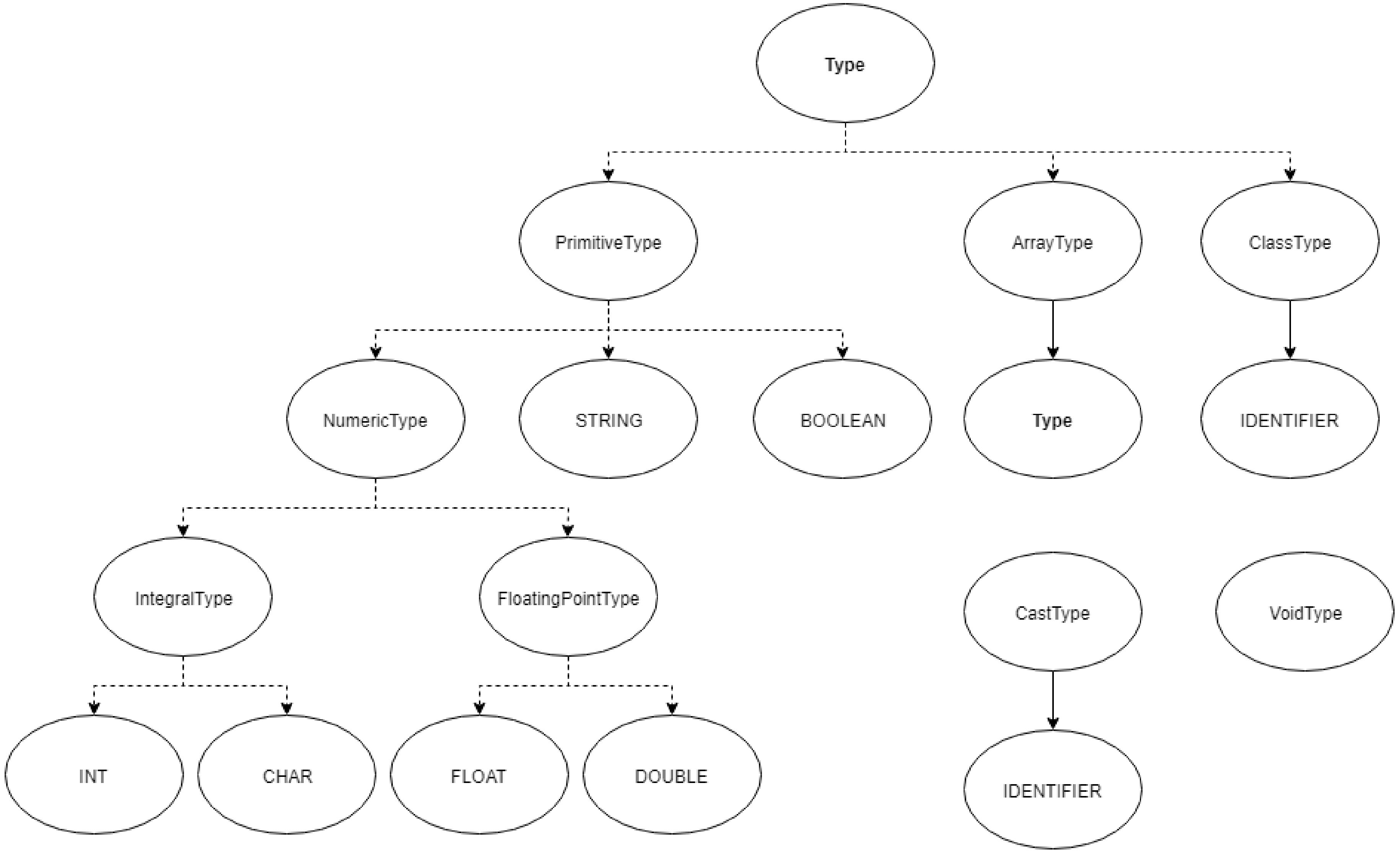


Figure 14 Types grammar diagram

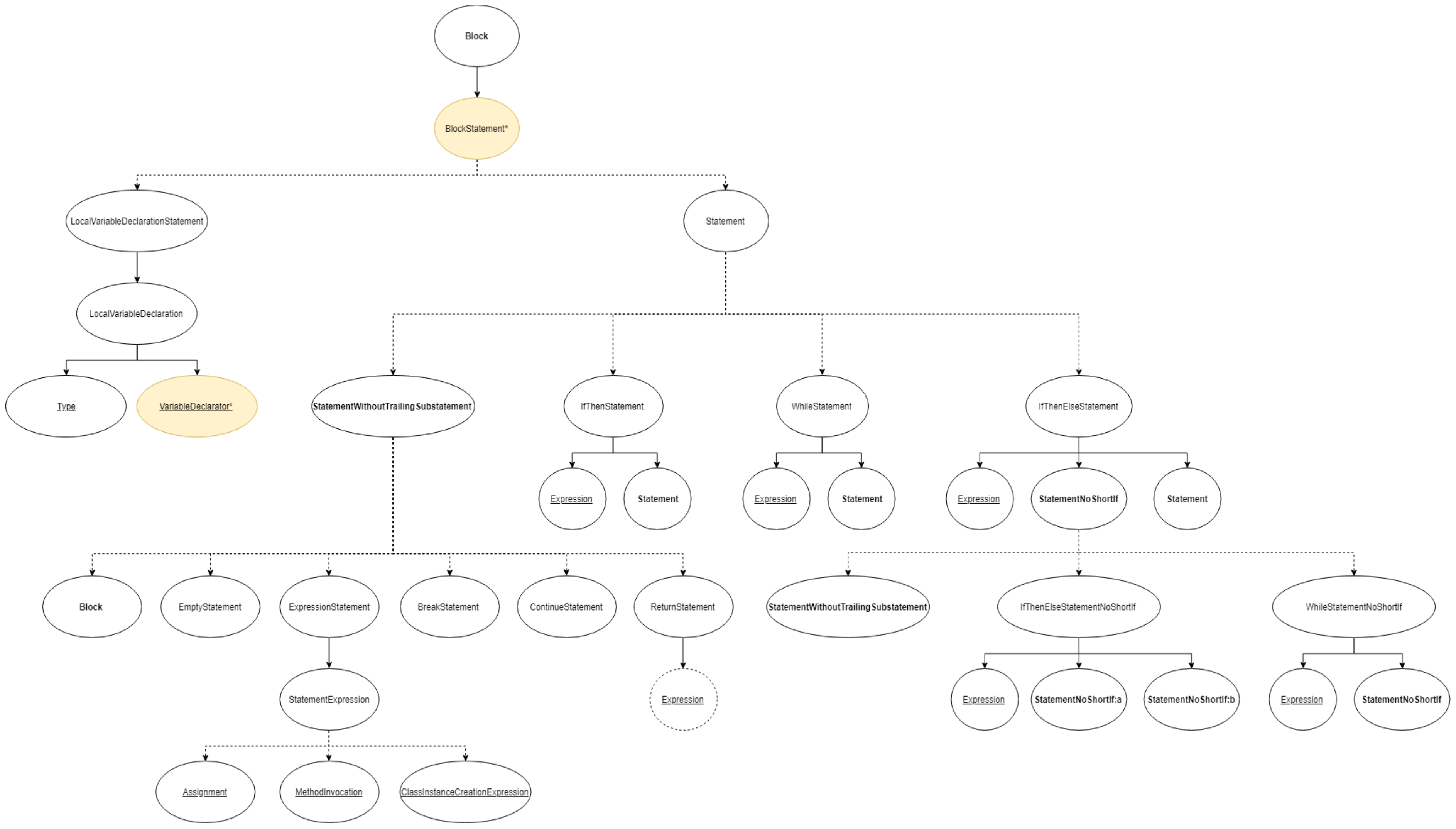


Figure A.4: Blocks and Commands grammar diagram

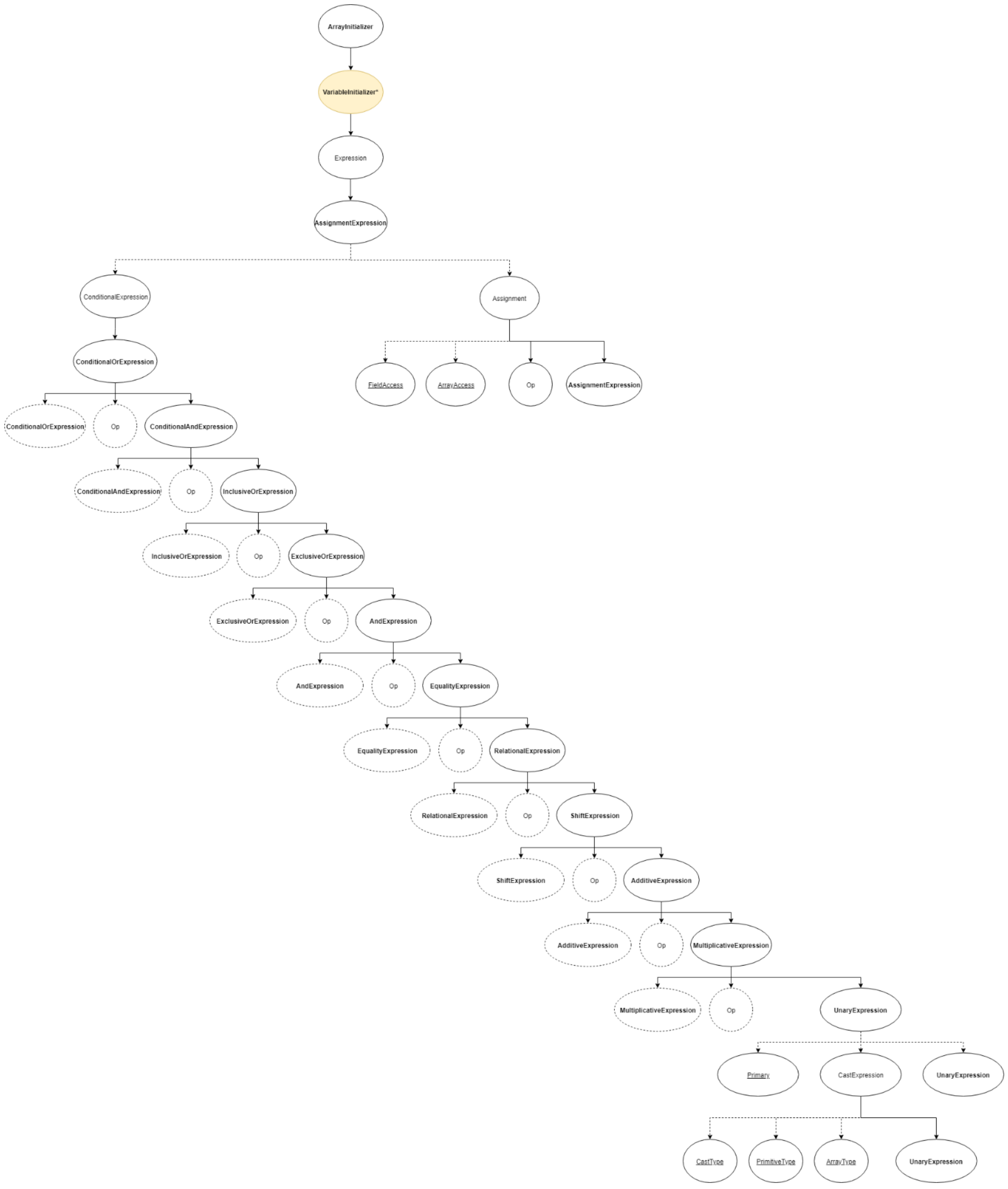


Figure A.5: Expressions grammar diagram

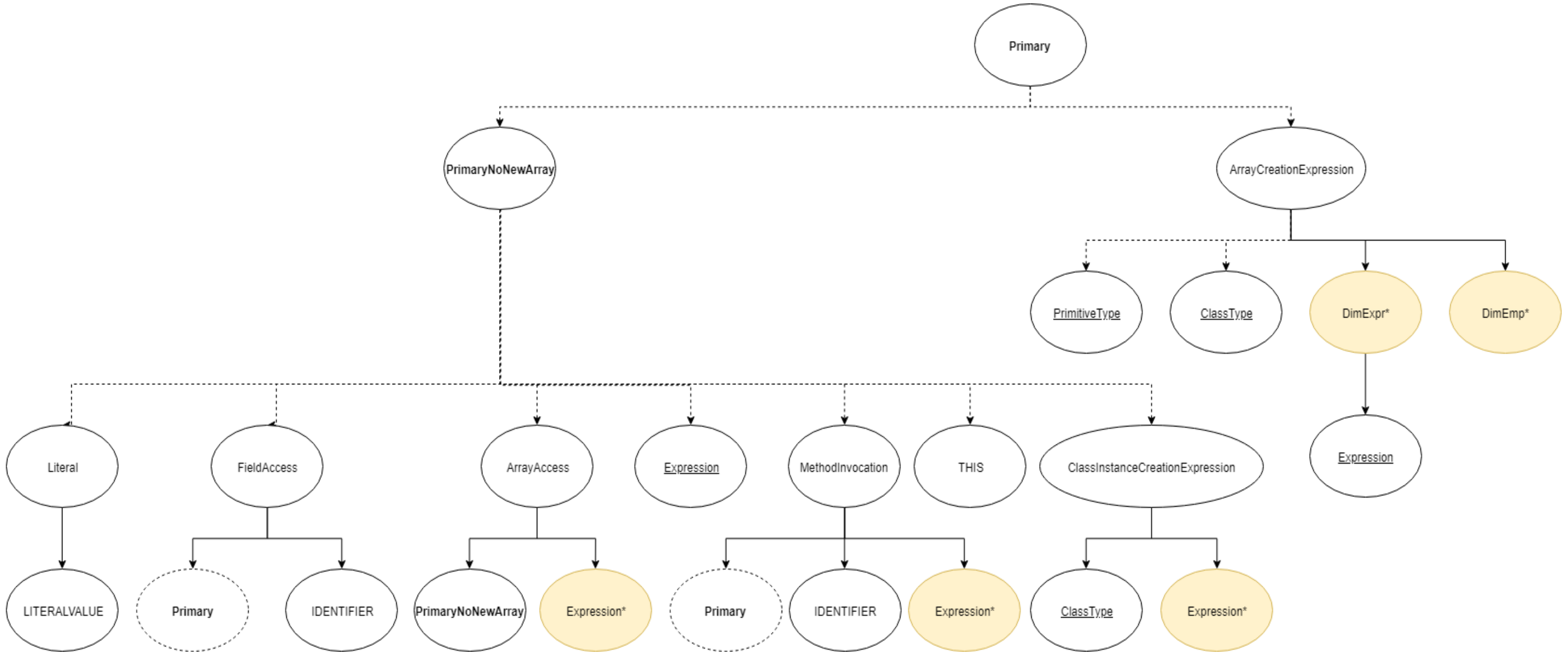


Figure A.6: Primaries grammar diagram

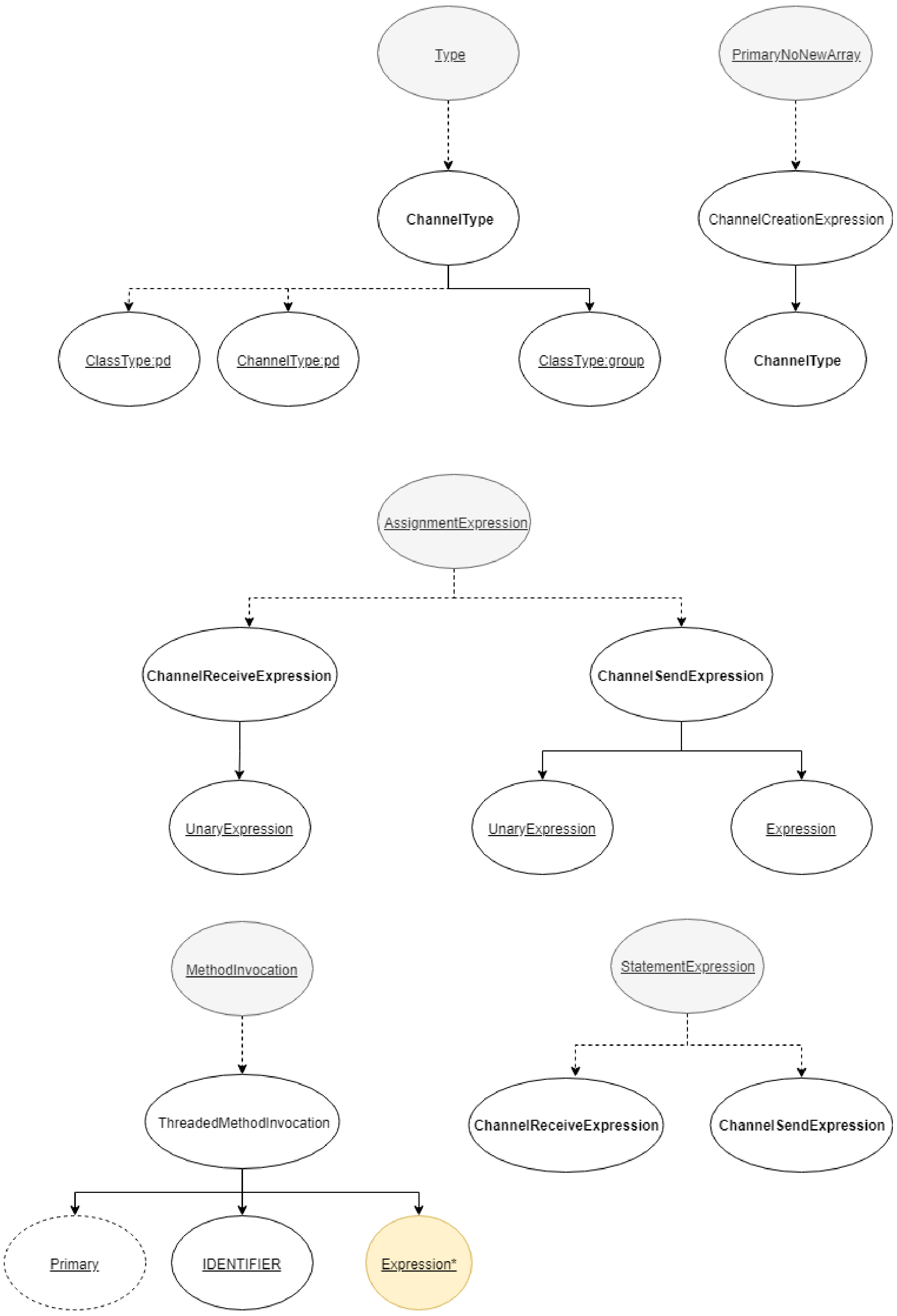


Figure A.7: Channels grammar diagram

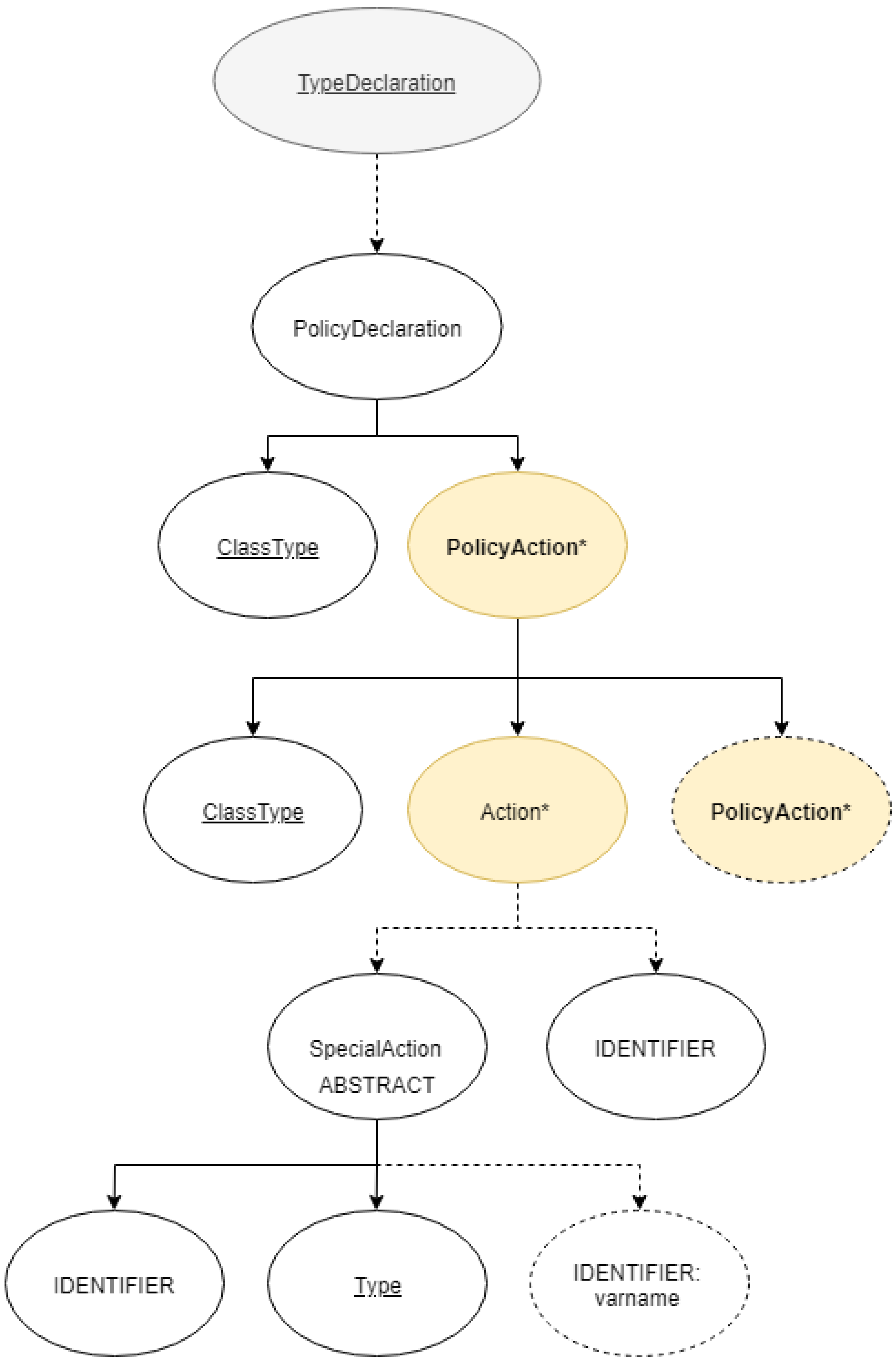


Figure A.8: Policies grammar diagram

Appendix B

Below are the diagrams regarding the Class Hierarchy of π-val. Let us assume the following:

1. A white filled ellipse with a continuous outline represents a class in the Class Hierarchy.
2. A dotted outlined ellipse represents an optional class. This means that the class which its child has a dotted outline, can be described without the absolute need of having a child of that type.
3. Yellow filled ellipses represent lists of the class that they describe. The star symbol at the end of the class name represents the allowance of the emptiness of such a list.
4. Bold titled ellipses represent classes that are described in detail in the same diagram.
5. Underlined titled ellipses represent classes that are described in another diagram.
6. Straight arrow lines show the necessity of a child in order to describe its parent class.
7. Shaped arrow lines show that the pointed class is the parent class, while the class where the line is directed from, extends its parent.

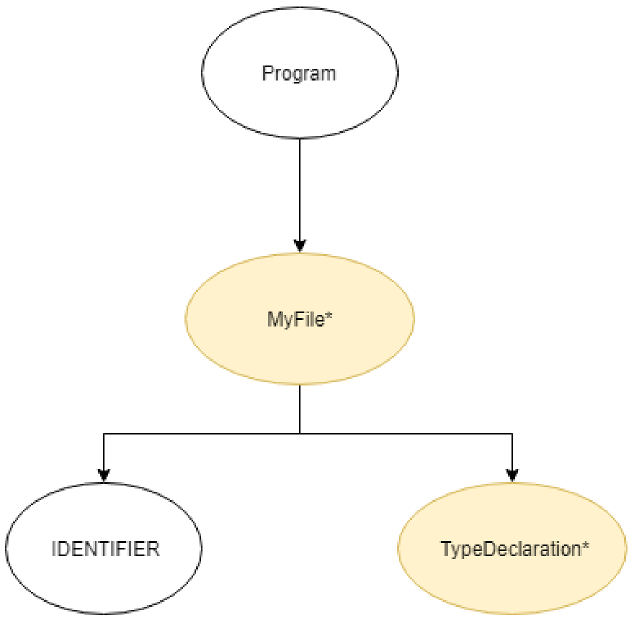


Figure B.1: Program hierarchy diagram

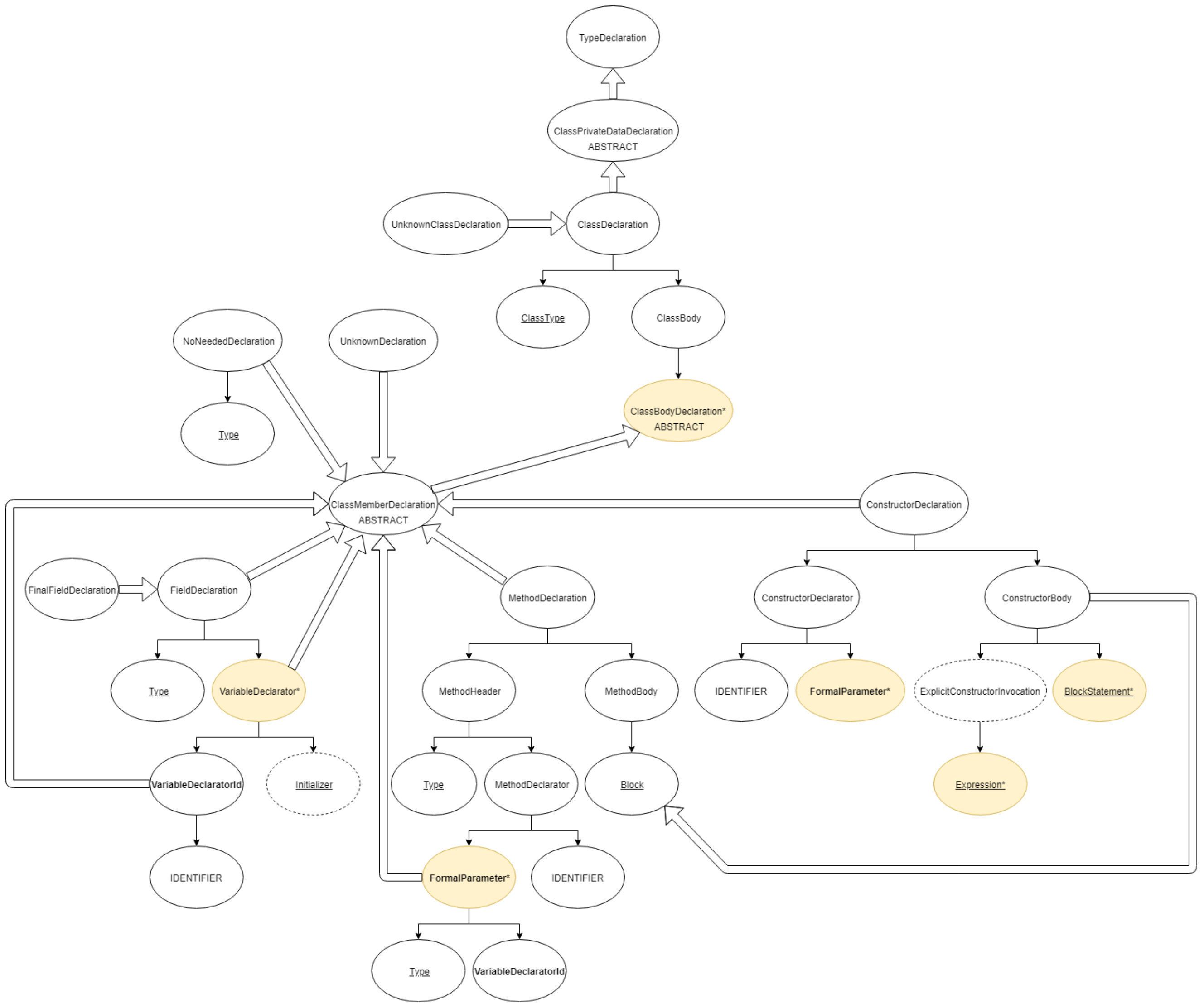


Figure B.2: Declarations hierarchy diagram

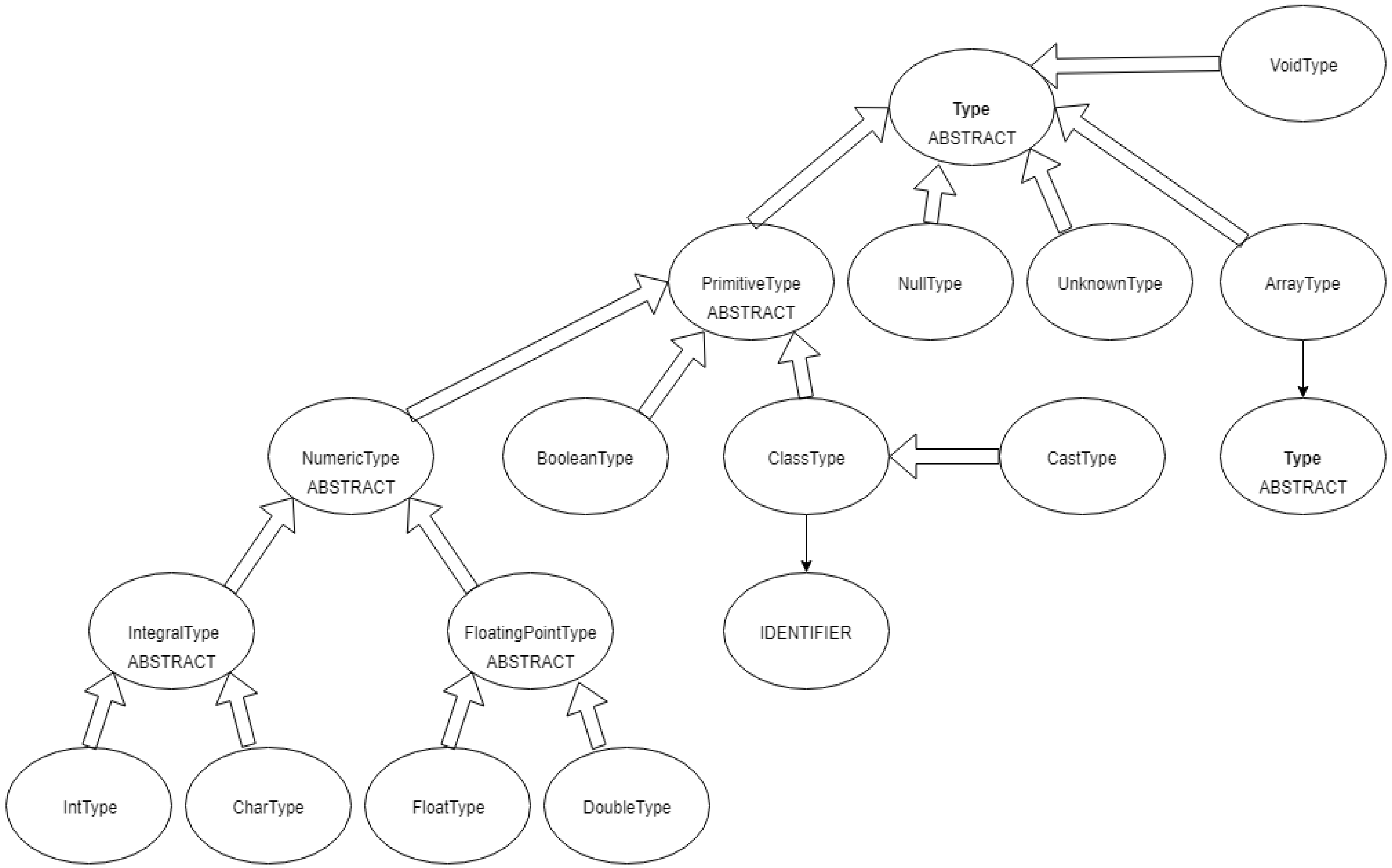


Figure B.3: Types hierarchy diagram

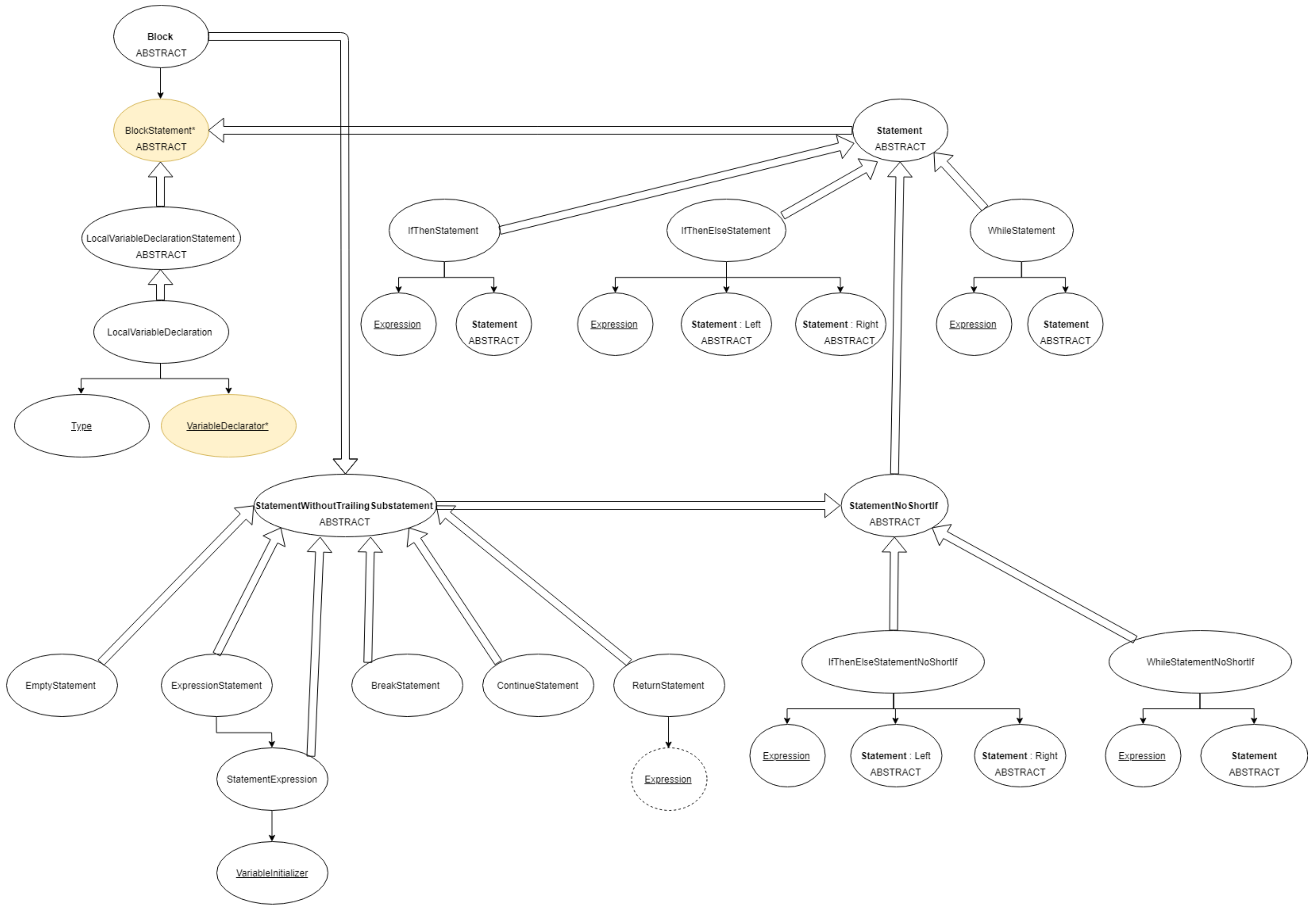


Figure B.4: Blocks and Commands hierarchy diagram

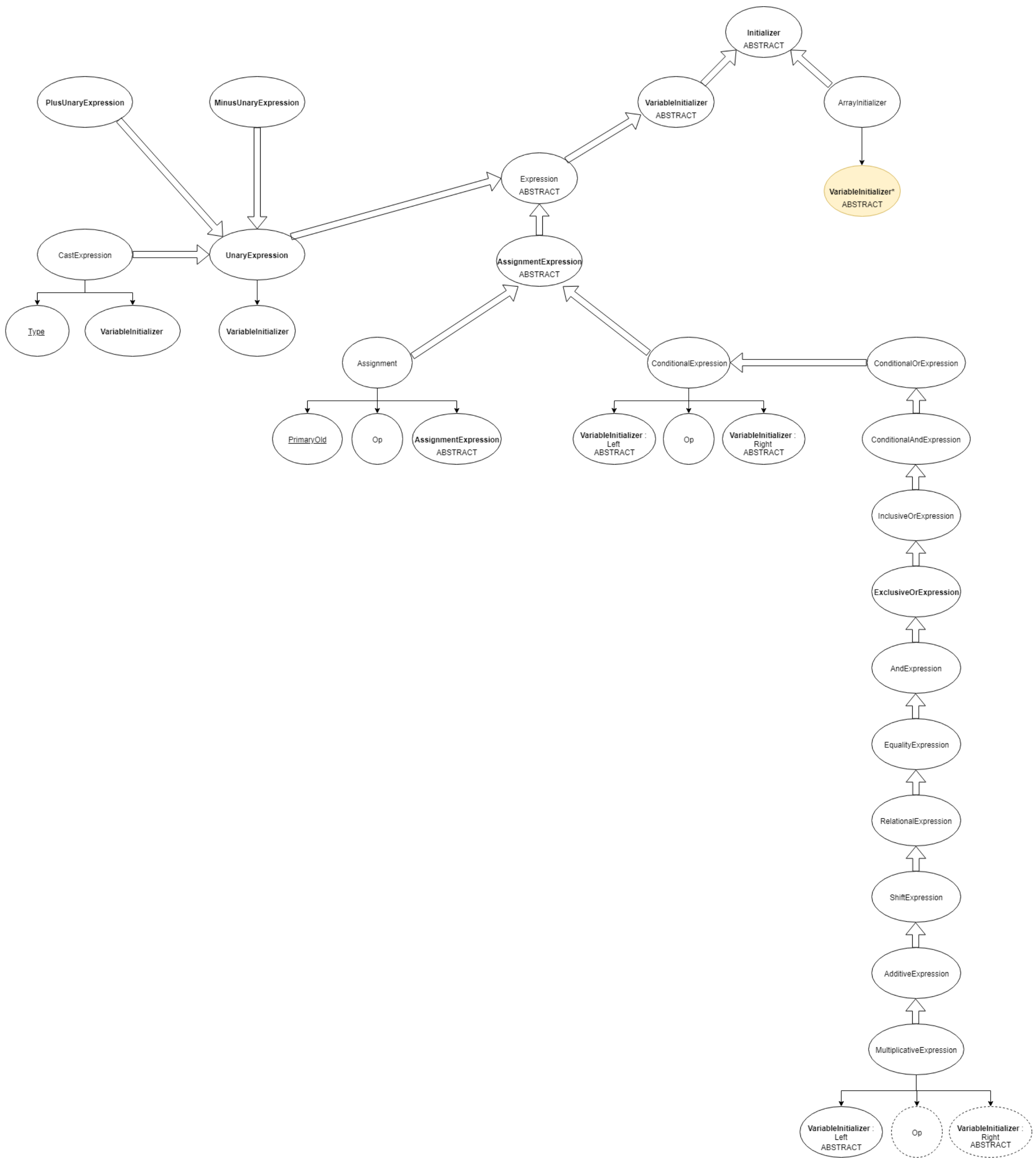


Figure B.5: Expressions hierarchy diagram

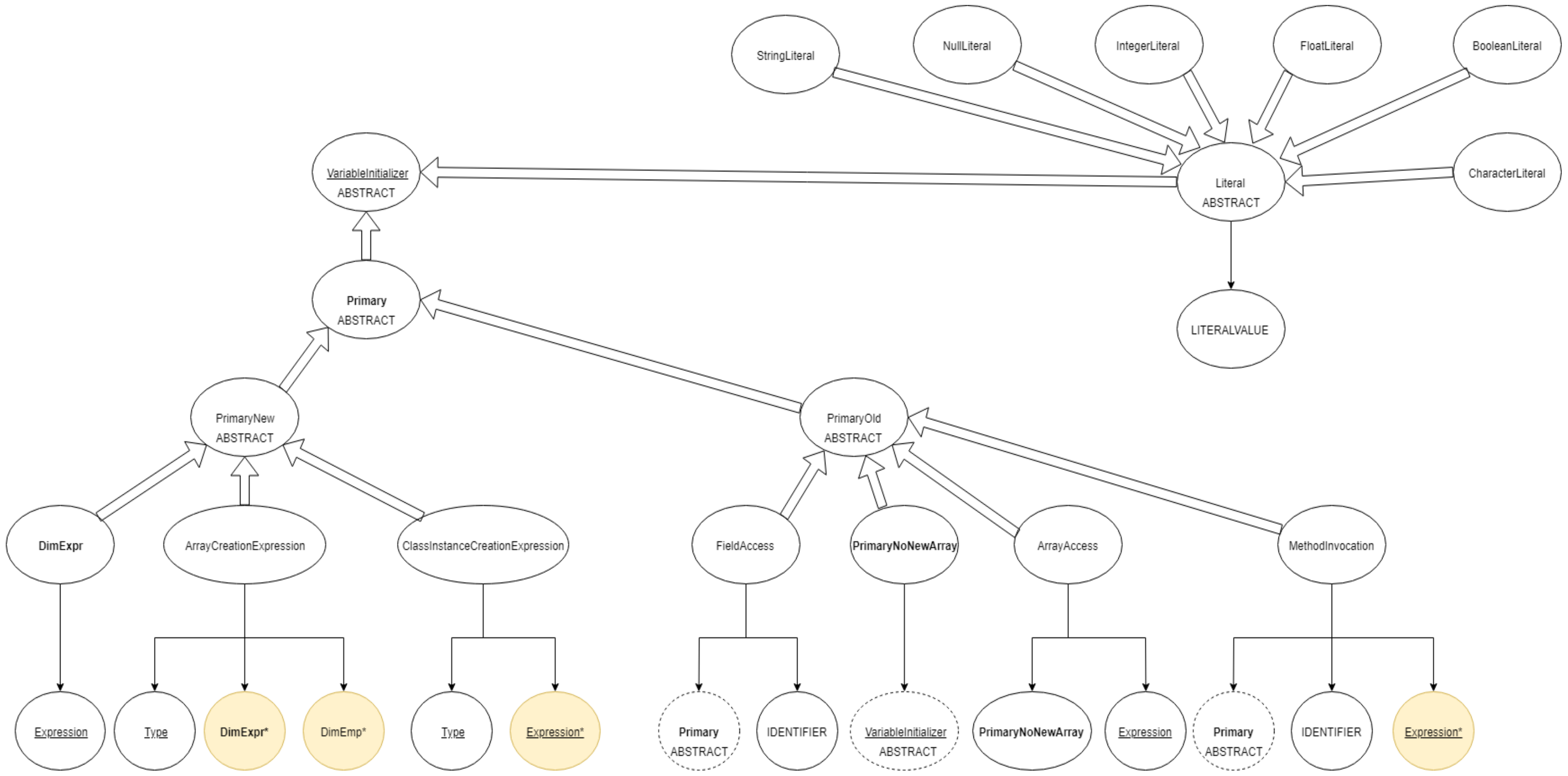


Figure B.6: Primaries hierarchy diagram

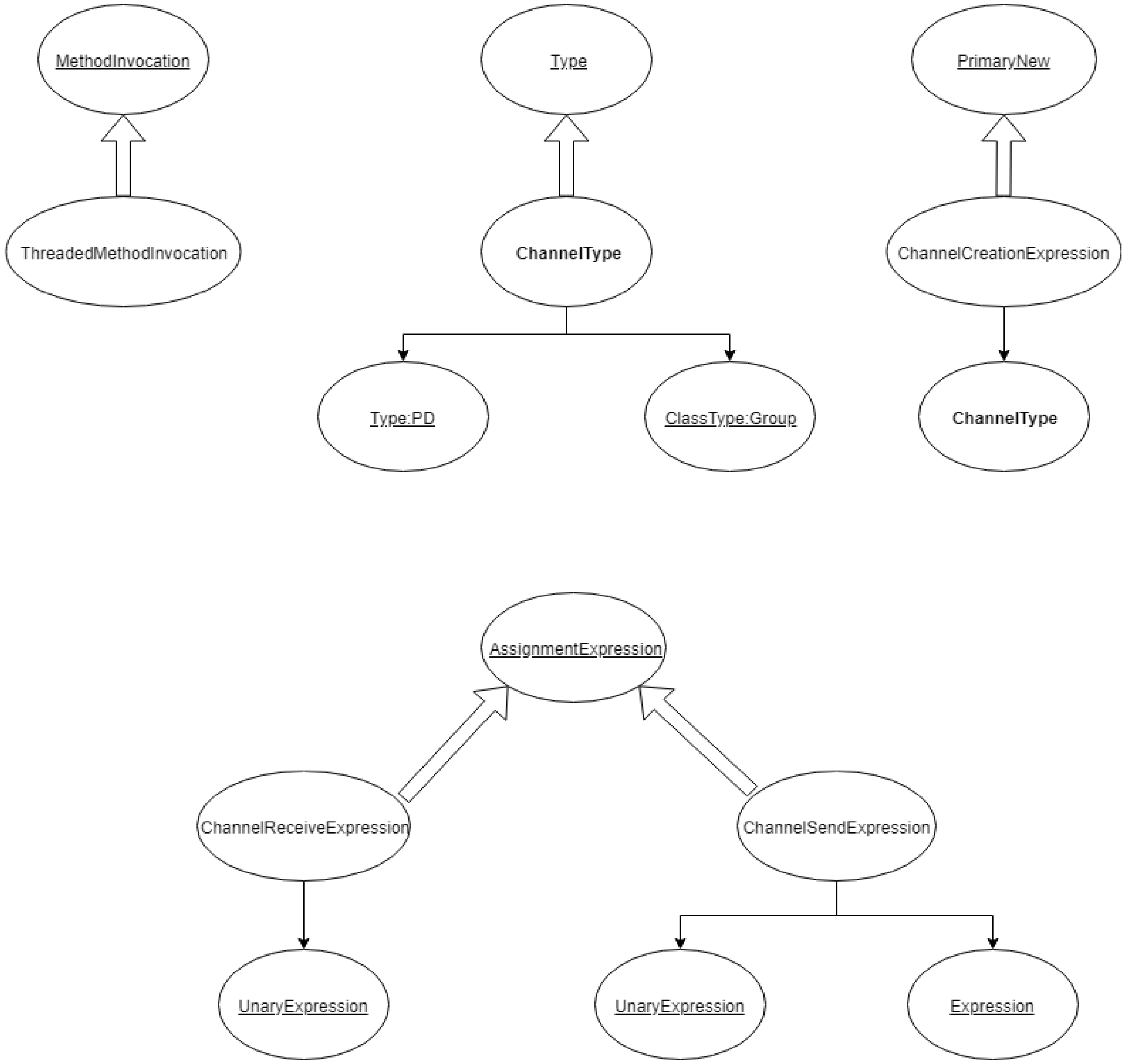


Figure B.7: Channels hierarchy diagram

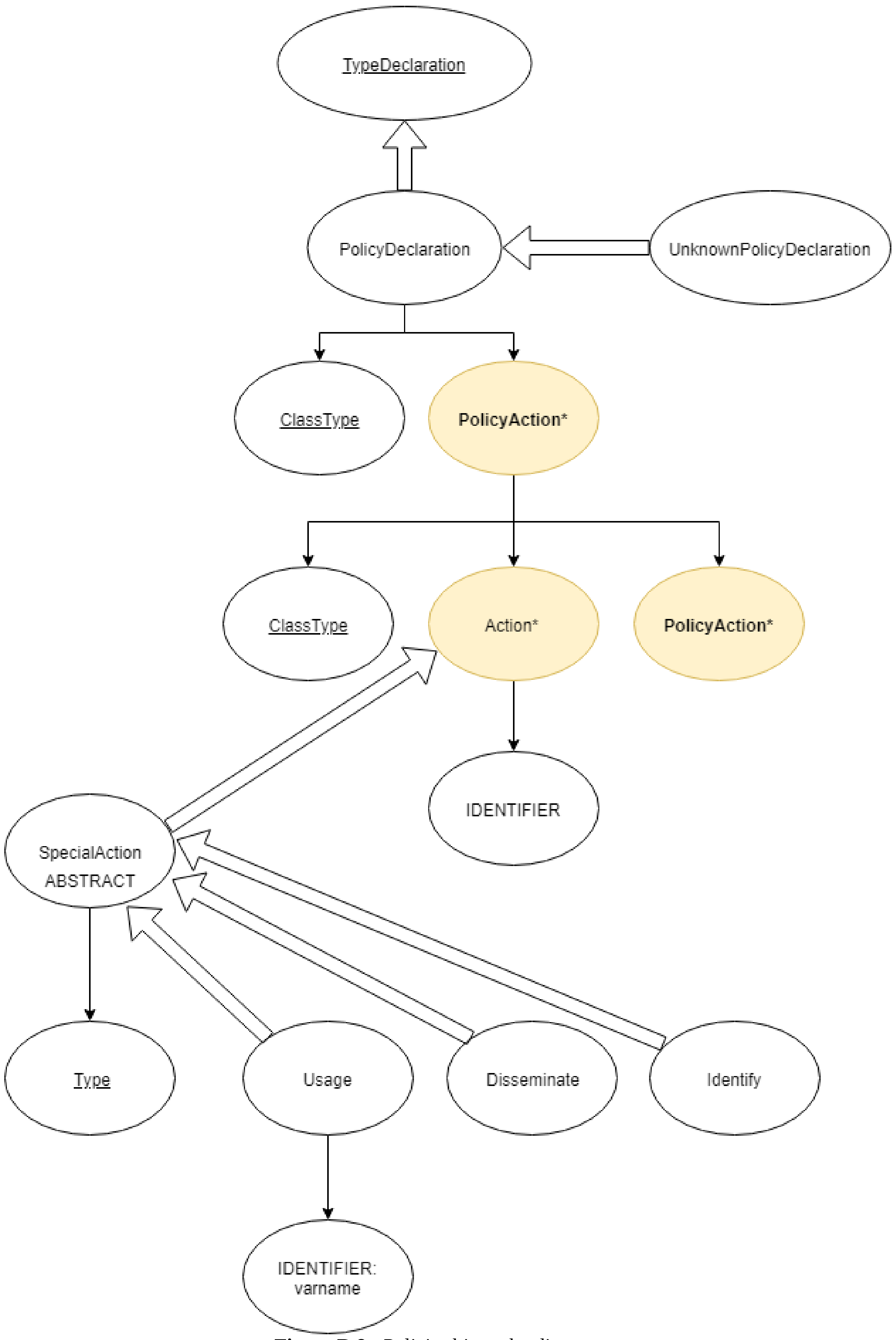


Figure B.8: Policies hierarchy diagram

Appendix C

Below is the generated Java [1] code of the example in Chapter 5. The code quoting highlighting was made using the online tool syntax-highlight-word by PlanetB [17].

```java
import AST._Channel;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class CarReg {
    String id = null;
    String reg = null;

    CarReg(String id, String reg) {
        this.id = id;
        this.reg = reg;
    }
}

class CarSpeed {
    String id = null;
    double speed = 0;

    CarSpeed(String id, double speed) {
        this.id = id;
        this.speed = speed;
    }
}

class DriverReg {
    String id = null;
    String reg = null;

    DriverReg(String id, String reg) {
        this.id = id;
        this.reg = reg;
    }
}

class SpeedControl {
    _Channel p1 = new _Channel();
    _Channel p2 = new _Channel();
    Car c1 = new Car("10", "ABC 123", 200);
    Car c2 = new Car("9", "BCD 234", 10);
    Car c3 = new Car("8", "CDE 345", 20);
    Car c4 = new Car("7", "DEF 456", 150);
    Car[] cars = {c1,c2,c3,c4};
    SCSystem scs = new SCSystem();

    void run() {
        int i = 0;
        while (i < cars.length) {
            cars[i].sendData_(p1, p2);
            i = i + 1;
        }
        scs.process_(p1, p2);
    }


    public static void main(String[] args) {
        System.out.println("Outcome");
        SpeedControl system = new SpeedControl();
        system.run();
    }

}

class SCSystem {
    _Channel a1 = new _Channel();
    _Channel a2 = new _Channel();
    _Channel link1 = new _Channel();
    _Channel link2 = new _Channel();
    _Channel link3 = new _Channel();
    _Channel link4 = new _Channel();
    _Channel[] links = {link1,link2,link3,link4};
    DB d1 = new DB("1", "driver 1", "ABC 123");
    DB d2 = new DB("2", "driver 2", "BCD 234");
    DB d3 = new DB("3", "driver 3", "CDE 345");
    DB d4 = new DB("4", "driver 4", "DEF 456");
    DB[] drivers = {d1,d2,d3,d4};
    TrafficCam tc = new TrafficCam();
    Authority auth = new Authority();

    void sendDB() {
        int i = 0;
        while (true) {
            drivers[i].sendReg(links[i]);
            i = (i + 1) % drivers.length;
        }
    }

    void sendDB_() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                sendDB();
            }
        }).start();
    }

    int process(_Channel p1, _Channel p2) {
                int i = 0;
                sendDB_();
                while (i < 4) {
                    tc.receiveData_(p1, p2, a1, a2);
                    auth.checkData_(a1, a2, links);
                    i = i + 1;
                }
                return 0;
            }

            int process_(_Channel p1, _Channel p2) {
                int _ret = 0;
                ExecutorService _es = null;
                try {
                    _es = Executors.newFixedThreadPool(1);
                    Future<Integer> _future =
                                   _es.submit(new Callable<Integer>() {
                        @Override
                        public Integer call() {
                            return process(p1, p2);
                        }
                    });
                    _ret = _future.get();
                } catch (InterruptedException e) {
                } catch (ExecutionException e) {
                }
                _es.shutdown();
                return _ret;
            }
        }

        class Car {
            CarReg r = null;
            CarSpeed s = null;

            Car(String id, String reg, double speed) {
                this.r = new CarReg(id, reg);
                this.s = new CarSpeed(id, speed);
            }

            void updateSpeed(_Channel cs) {
                CarSpeed y = (CarSpeed) cs.receive();
                if (y != null) {
                    s.speed = y.speed;
                }
            }


            void sendData(_Channel p1, _Channel p2) {
                p1.send(r);
                p2.send(s);
            }

            void sendData_(_Channel p1, _Channel p2) {
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        sendData(p1, p2);
                    }
                }).start();
            }
        }

        class TrafficCam {

            void receiveData(_Channel p1, _Channel p2,
                                      _Channel a1, _Channel a2) {
                a1.send((CarReg) p1.receive());
                a2.send((CarSpeed) p2.receive());
            }

            void receiveData_(_Channel p1, _Channel p2,
                                      _Channel a1, _Channel a2) {
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        receiveData(p1, p2, a1, a2);
                    }
                }).start();
            }
        }

        class Authority {
            final double overLim = 30;
            CarSpeed[] violations = new CarSpeed[0];
            _Channel v = new _Channel();
```

```java

        void checkData(_Channel a1, _Channel a2, _Channel[] links) {
            CarReg k1 = (CarReg) a1.receive();
            CarSpeed k2 = (CarSpeed) a2.receive();
            if (k2.speed > overLim) {
                int i = 0;
                while (i < 4) {
                    DriverReg dr = (DriverReg) links[i].receive();
                    if (dr.reg == k1.reg) {
                        System.out.println("match");
                        expandViolationsTable();
                        violations[violations.length - 1]
                               = new CarSpeed(dr.id, k2.speed);
                    }
                    i = i + 1;
                }
            }
        }

        void checkData_(_Channel a1, _Channel a2, _Channel[] links) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    checkData(a1, a2, links);
                }
            }).start();
        }

        void expandViolationsTable() {
            CarSpeed[] temp = new CarSpeed[violations.length + 1];
            int i = 0;
            while (i < violations.length) {
                temp[i] = violations[i];
                i = i + 1;
            }
            violations = temp;
        }

    }

    class DB {
        String name = null;
        DriverReg r1 = null;

        DB(String id, String name, String reg) {
            this.r1 = new DriverReg(id, reg);
            this.name = name;
        }

        void sendReg(_Channel link) {
            link.send(this.r1);
        }

    }
```