Universidade do Minho
Escola de Engenharia

André Bernardo Coelho Nunes

Scope Rules in Ztrategic

dezembro 2024

Universidade do Minho
Escola de Engenharia

André Bernardo Coelho Nunes

Scope Rules in Ztrategic

Master dissertation
Master in Informatics Engineering

Dissertation supervised by
João Alexandre Batista Vieira Saraiva dezembro 2024

Declaração de Integridade

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Universidade do Minho, Braga, dezembro 2024

Nome completo do autor André Bernardo Coelho Nunes

Resumo

A atribuição de nomes é crucial nas linguagens de programação, identificando declarações de entidades como variáveis, funções, tipos e módulos. A resolução de nomes liga cada referência à declaração pretendida, o que é fundamental para operações como a verificação estática, a tradução, a semântica mecanizada e os serviços **IDE**. Este processo é complexo, atravessando a estrutura do programa tal como descrita por uma árvore de sintaxe abstrata **AST**. Por exemplo, um nome introduzido numa parte de uma **AST** pode ser referenciado muito longe, tornando a resolução difícil.

Os espaços de nomes explícitos, como os do *Java*, aumentam a complexidade ao exigirem uma resolução em várias etapas: primeiro, a resolução do nome da classe ou do pacote e, posteriormente, o nome do membro dentro desse contexto. Apesar das variações entre as linguagens, conceitos semelhantes de resolução de nomes são muito comuns em linguagens com âmbito lexical. Esta consistência sublinha a importância da resolução de nomes, mas nenhuma técnica modular e reutilizável fornece uma solução universal. Cada linguagem requer algoritmos adaptados que, muitas vezes, necessitam de múltiplas passagens **AST** para garantir a ligação correcta do nome e a adesão à regra de Scope, reflectindo a natureza intrincada e abrangente da nomeação na conceção de linguagens de programação.

Resolvemos este problema apresentando uma solução genérica de resolução de nomes que oferece uma interface simples e intuítiva que abstrai a maior parte das complexidas relativas às regras de Scope e permite a deteção e anotação de erros referentes às mesmas no código fonte.

Palavras-chave Programação Estratégica, Scope, análise de nomes, regras de Scope, Ztrategic

Abstract

Naming is crucial in programming languages, identifying declarations of entities like variables, functions, types, and modules. Name resolution links each reference to its intended declaration, which is vital for operations such as static checking, translation, mechanized semantics, and **IDE** services. This process is complex, cutting across the program's structure as described by an abstract syntax tree **AST**. For instance, a name introduced in one part of an **AST** may be referenced far away, making resolution challenging.
Explicit namespaces, such as those in *Java*, add further complexity by requiring multi-step resolution: first resolving the class or package name, then the member name within that context. Despite variations across languages, similar name resolution concepts recur widely in lexically scoped languages. This consistency underscores the importance of name resolution, yet no modular, reusable technique provides a universal solution. Each language requires tailored algorithms that often necessitate multiple **AST** traversals to ensure correct name binding and scope rule adherence, reflecting the intricate and pervasive nature of naming in programming language design.
We have solved this problem by presenting a generic name resolution solution that offers a simple and intuitive interface that abstracts away most of the complexities related to Scope rules and allows errors related to them to be detected and annotated in the source code.

Keywords Strategic Programming, Scope, Name resolution, Scope rules, Ztrategic

Contents

Chapter 1

Introduction

In the context of software language engineering, name resolution and Scope rules play a crucial role. The search for more robust, efficient and flexible software has led to a search for more precise and ductile approaches in order to meet the specific challenges of each language.
However, current practices often prove to be limited and unable to adapt to the diverse nuances of existing languages. This inability is reflected in development methods that use static and manual analysis, which need to be updated whenever the language evolves.
The new abstraction will enable compiler-writters to model and apply Scope rules and name analysis in an efficient and adjustable way, recognizing the dynamic nature of programming languages. Consequently, for each language, the user only needs to provide some information about it for the system to derive Scope analysis rules automatically.
Finally, this project aims to change the approach of this kind of developers to the analysis of names and Scope rules, designing these changes to have a lasting impact on the software industry, elevating the concept of strategic programming as an innovative and appropriate approach to this type of problem and helping to turn name resolution into a plug-and-play component of languages, with well-defined rules and practices.

1.1 Motivation

Name resolution, also known as the name analysis phase of language processors, plays a crucial role in the design and implementation of programming languages. It helps in identifying declarations of program entities such as variables, functions, types, and modules, enabling these entities to be easily referenced from various parts of the program. The scope rules of the language specify how names can be declared and used in a program. Different languages use different scope rules, although the basic concepts of resolution reappear in similar form across a broad range of languages.

Let us consider a simple and concise example of the name resolution of a programming language. We consider the (sub)language of **Let** expressions as incorporated in most functional languages, including *ML* Harper (2011) and *Haskell* HAS. While being a concise example, our **Let** language holds central characteristics of name resolution, such as the mandatory but unique declarations of names. Next, we show an example of a **Let** program, where we use *Haskell* comments to clarify the declaration and use of names.

```
program = let a = b + 3         -- decl a , use b
              c = 2             -- decl c
              b = c * 3 - c     -- decl b , use c, use c
          in (a + 7) * c        -- use a, use c
```

In our **Let** language a program is valid when all names used are indeed declared, and a name is not declared more than once. In fact, *program* is a valid **Let** program (it is also a valid *Haskell* expression). This happens, because **Let** uses the scope rules of *Haskell* which do not force a *declare-before-use* discipline, meaning that a variable can be declared after its first use. There are other languages, however, where a name must be declared before is first use. This is the case of the *C* language C. Thus, if we would consider the scope rules of *C* in defining the static semantics of our **Let** language, then *program* is invalid: name *b* is used before it is declared!

Like most languages, **Let** also supports nested block-based structures as our next **Nested Let** program shows.

```
nested = let a = b + 3               -- decl a , use b
             c = 2                   -- decl c
             d = let c = a * b       -- decl d, [decl c, use a, use b
                     e = 6           -- decl e
                 in c * b            -- use c, use b]
             b = c * 3 - c           -- decl b , use c, use c
         in (a + 7) * c + [e](ERROR) -- use a, use c, use e
```

If we consider *Haskell* scope rules, then the use of *a* and *b* in the inner **Let** block comes from the outer block expression since they are not defined in the inner one. Since *c* is defined both in the inner and in the outer block, then we must use the inner *c* (defined to be *a * b*) when calculating *c * b*, but we use the outer *c* (defined to be *2*) when calculating *(c * 3) - c*. However, this is an invalid **Let** program: the inner declaration of *e* is not visible in its outer block. Thus, an error should report the invalid use of *e* in the *in* part of the outerblock. Instead, if we would consider the simpler scope rules of *C*, then we have two more errors: the two invalid uses of name *b* since in both cases *b* is defined after being used.

As our simple **Let** language already shows, name resolution and the implementation of scope rules is a complex task:

- *Real Languages are Represented by Large and Complex Heterogenous* ***AST****s*: Programming languages contain many types and constructors. Let us consider, for example, the *Haskell* language. In the *Haskell* library, the **AST** is defined by 120 constructors across 30 data types[1]. Because a name introduced by a let node in the **AST** may be referenced by an arbitrary distant node, then an *Haskell* name resolution algorithm needs to traverse most of the nodes of the **AST**. Thus, this immediately leads to large recursive functions consisting of (almost) 120 different case statements or pattern matching alternatives!

- *Scope Rules Rely on Complex Multiple Traversal Algorithms*: Modern scope rules, as used by languages such as *Haskell* or *Java*, allow the use of names before they are defined. This leads to name resolution algorithms that need to traverse the **AST** twice: once to collect all declarations in the environment, also called symbol table, while detecting duplicated declarations, and a second traversal to check for invalid uses. Such a clear and straightforward algorithm, however, may require a complex scheduling of the traversal functions. Let us consider the **Let** language again. In **Let**, a nested definition inherits the names defined in its outer one. As a consequence, only after collecting the declarations of a outer let (performed in its first traversal), it can descend to its nested definitions. Thus, only in the second traversal of an outer let, the implementation performs the first traversal of inner ones, intermingling traversals.

- *Different languages use slight different scope rules*: All programming languages require name resolution to link identifiers like variables and function names to their corresponding entities in code. Different languages use various scope rules to achieve this. Lexical Scope, or Static Scope, is determined by the variable's location in the source code and is common in languages like *Python*, *JavaScript*, and *C++*. Dynamic Scope is determined at runtime based on the call stack and is used in some older languages like early *Lisp*. Other scoping mechanisms include Block Scope, which limits variable visibility to the block where it is declared. Function Scope limits variable visibility to the function where it is declared. Module Scope means variables are scoped to the module, used in languages like *Python*. Namespace Scope groups related identifiers under a common name

[1] `https://hackage.haskell.org/package/haskell-src`

to avoid naming conflicts, used in *C++* and *C#*. Understanding these scope rules is essential for writing clear and error-free code.

As a result, defining a generic technique, supported by and off-the-shelve library is a complex task. Such an automated name resolution technique should rely on a powerful formalism suitable to describe intricate scope rules of programming languages. Thus, such a formalism should allow to concisely and elegantly express scope rules, such as *Haskell* scope rules, which can then be automatically applied to any programming language. Because scope rules differ between programming languages, new rules should be easily defined by reusing and extending existing ones. That is to say that existing scope rules are the building blocks to express new ones. For example, the simpler scope rules of *C* could be defined by reusing and extending the *Haskell* ones. Then, if we use the *C* scope rules in our first example we should get the following result:

```
program = let a = [b] + 3
              c = 2
              b = c * 3 - c
           in (a + 7) * c
```

We have just shown the first example of applying our automated name resolution technique to a programming language: the **Let** language. In fact, the pretty printed **AST**s showing the *Haskell* and *C* name resolution results were automatically produced by our off-the-shelve scope rules library.

1.2 Main objectives

This project aims to transform the way in which name analysis and Scope rules are designed and implemented, with substantial impacts on the quality of the source code as well as its validation, evolution and maintenance.
One of the goals of this project is therefore to create this innovative abstraction layer that aims to simplify the expression of Scope rules, making them more accessible and intuitive for developers.
The proposed solution aims to facilitate this process by eliminating the complexity associated with analyzing names and Scope rules. Thus, the interface developed allows programmers to specify Scope rules in an intuitive and universal way. This abstraction is able to carry out all the necessary checks, producing an error report after processing, and thus validating or not the coherence of names and Scope rules in the code to be analyzed.
It should be noted that this project is not a search for technical efficiency, but rather a quest to improve code quality and increase expressiveness for language developers.

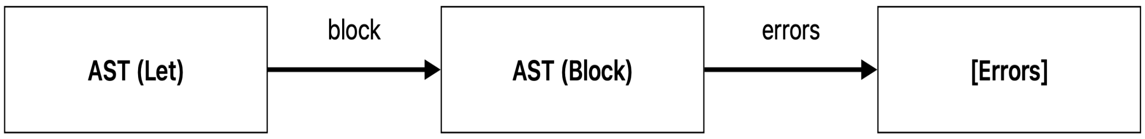


Figure 1: Scope rules processor relational diagram concept

1.3 Research questions

We should discuss that name resolution can be easier in a single **AST**, and not in the larger and complex **AST** of real programming languages. Thus, it is common practice to compute for the original **AST** where we express the complex relations between names. This (traditional) approach has several problems: the definition of the rules to build it has to ”go through” all the nodes/constructors of the original language, and naming errors are flagged in the second (higher-order) tree and not in the source language programs. There are currently several solutions to the problem of resolving names and Scope rules. However, none of them are comprehensive enough to be adapted to any language in an elegant and effective way, making this practice simple and carried out in a ”plug-and-play” way.
The aim of this work is to provide developers with a practical and efficient reusable library, but at the same time sufficiently comprehensive to deal with this problem by making transparent all the intricacies relating to name resolution and Scope rules.
Although many questions arose collaterally and spontaneously during the research phase, the following Thesis Research Questions **(TRQ)** are the focus of this work:

- **TRQ1:** Is it possible to automatically infer this whole process and construction of the representation where we express the analysis of names?
- **TRQ2:** Is it possible to define a mechanism that allows errors to be flagged in the original tree and not in the one where we define the naming rules?
- **TRQ3:** Is it viable to merge the workings of the Block language processor (**??**) with the parsing process of a language in order to parse names and Scope rules without losing information about the language in question?

Chapter 2

State of the art

This chapter describes the concepts covered in this dissertation. Initially, all the theory related to name resolution will be presented and interpreted, Neron et al. (2015), as well as declarative name binding and Scope rules, Konat et al. (2013), establishing the necessary notions for the subsequent exposition of concepts.

We will then explore how the current **Integrated development environments** deal with the nuances and problems surrounding name resolution and Scope rule checking, in order to see both their limitations and strengths and to verify which techniques can be reused and which aspects need to evolve in this area. Later, we'll cover the concepts related to the association of Scopes as types, Antwerpen et al. (2018) , which present the basic theory for the use of attribute grammars and Strategic Programming in the resolution of names and verification of Scope rules.

Finally, we will theorize how we can combine these tools to develop the new abstraction layer in the Strategic Programming library (Ztrategic), using an approach based on attribute grammars, Zippers and strategies.

2.1 Declarative name binding and Scope rules

Declarative name binding concerns the relationship between identifier definitions and references in textual software languages, including the Scope rules that govern these relationships according to the work of Konat et al. (2013). In the context of language processors, it is essential to gather information about the available definitions and their references. These practices play a fundamental role in various language engineering processes such as **IDE**'s, reference resolution, code completion, refactorings, type checking and compilation.

The different requirements inherent in different languages lead to multiple re-implementations of the Name Scope rules for each of these goals, or to a non-trivial manual integration of a single implementation that supports all the goals. This results in code duplication, resulting in errors, inconsistencies and increased maintenance effort.

2.1.1 Scopes

A Scope refers to the area in which a function or variable is visible and accessible to other code, restricting the visibility of definition sites. For example, when you define a variable, it will be enclosed in a Scope along with all the other declarations at the same level.
As such, it is necessary to classify these Scopes into two distinct categories, as mentioned in Konat et al. (2013), Named Scopes and Anonymous Scopes. Named Scopes are the scopes that can be referenced by name, for example a *Java* Class. On the other hand, Anonymous Scopes do not define a name but rather a place of action.
Due to their nature and purpose, Scopes are usually nested and therefore name resolution looks for an implementation from inner Scopes to outer Scopes.

```
//Named Scope
public class Animal {
    String type = "Dog";
}
```

```
//Anonymous Scope
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.forEach(n -> System.out.println(n * n));
```

2.1.2 Name resolution algorithm

According to Konat et al. (2013), an algorithm that seeks to answer this problem should be divided into three distinct phases, the annotation phase, the analysis site definition phase and the use site analysis phase.
In the first phase the **AST**[1] is traversed in a descending manner in order to collect information on all the definition and usage sites in order to generate a reference for the following phases.
The second phase analyzes each definition location, again in a top-down manner, with the aim of gathering information about these definitions, such as their type, and stores this information to be processed by the next phase.
Finally, references are sorted out and the types that depend on non-local information are determined.

[1] Abstract syntax trees are tree data structures that represent syntactic structures of strings, according to a formal grammar.

2.2 Name resolution theory and Scopes as types

Name resolution is often complex, since it cuts across the local inductive structure of programs (described by an **AST**). This resolution underlies most operations in languages and programs, including static checking, translation, mechanized description of semantics and providing editor services in **IDE**s.

To do this, we need a theory that unifies all the inherent concepts, but at the same time is flexible enough to take into account the specific nuances of each language.

The work of Neron et al. (2015) provides a language-independent theory suitable for languages with complex Scope rules including lexical Scoping[2] and modular Scoping[3]. We are presented with a declarative specification of the resolution of references to statements by means of a Lambda Calculus[4] which defines this same resolution in a Scope graph. This theory was fundamental in the development of the new abstraction layer on the Strategic Programming library, Macedo (2022), since it served as the theoretical basis for the implementation produced, given that it proves the complexities of this type of operation through formal methods.

2.2.1 Scope graphs and their construction

A Scope graph captures the binding structure of a program. A Scope is a location in a program that behaves uniformly with respect to name resolution.

According to Antwerpen et al. (2018), in the context of a Scope graph, name declarations and references are associated with Scopes. Thus, these are represented in a graph by nodes and their visibility and accessibility is modeled by edges between Scopes.

A name resolution algorithm interprets a Scope graph to resolve references to declarations, finding the most specific well-formed path.

To express the Scope rules of a programming language, a mapping from **Abstract syntax trees** to Scope graphs is defined. The mapping groups all **AST** nodes that behave uniformly in relation to name resolution into a single Scope node. For any language, the construction can be specified by a conventional syntax-driven definition of the language grammar. In the work of Neron et al. (2015) it is explained that an algorithm for resolving and finding paths in a Scope graph is correct if it is acyclic and if all names have a visible and accessible Scope path.

The algorithm described in that same article provides a basis for implementing tools based on Scope

[2] It is the area of definition of an expression. In other words, the lexical Scope of an item is the place where it was created.

[3] Scope referring to an area of modular code such as a Java class or interface.

[4] A formal system in mathematical logic for expressing computation based on the abstraction and application of functions using variable binding and substitution.

graphs, as is the aim of this thesis.

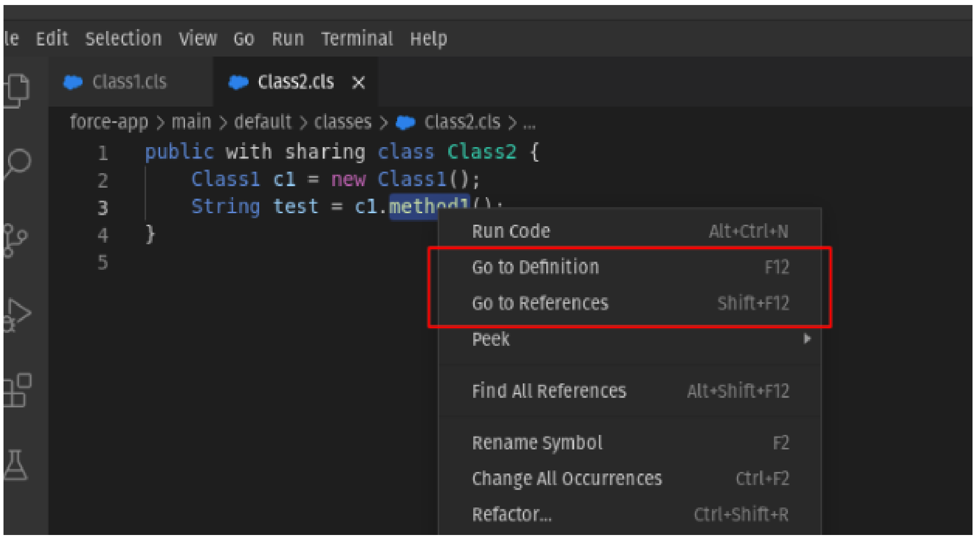
2.3 Integrated development environments in name resolution

Today's **IDE** provide a variety of services when it comes to name resolution. However, as mentioned earlier, they all lack a generic method to deal with the specific nuances of each language.
Generally, these services are developed manually and made specifically for each language, even if there are some outliers like the *Language Server Protocol*[5] from VSC, which requires a substantial amount of work, both in their development and in their maintenance and evolution. In most cases the services for each language are independent packages maintained by a certain group of people, meaning that there are several groups of people repeating what is mostly the same work for different languages.
Nonetheless, by modeling these relationships it is possible to generate a name resolution algorithm that can later be used by these editing services in a less complex and laborious way, as mentioned in the work by Konat et al. (2013).

2.3.1 Reference resolution

When it comes to resolving references, most **IDE** already offer a wide range of solutions with features such as finding the definition of a name, the definition of its type, its references and its implementations if applicable.
However, as already mentioned, these tools only exist for specific languages for which these services have been developed manually.

[5] The Language Server Protocol is an open, JSON-RPC-based protocol for use between source code editors or integrated development environments (IDEs) and servers that provide "language intelligence tools": programming language-specific features like code completion, syntax highlighting and marking of warnings and errors, as well as refactoring routines.



```
public with sharing class Class2 {
    Class1 c1 = new Class1();
    String test = c1.method1();
}
```

le Edit Selection View Go Run Terminal Help
Class1.cls
Class2.cls
force-app > main > default > classes > Class2.cls > ...
Run Code Alt+Ctrl+N
Go to Definition F12
Go to References Shift+F12
Peek
Find All References Alt+Shift+F12
Rename Symbol F2
Change All Occurrences Ctrl+F2
Refactor... Ctrl+Shift+R

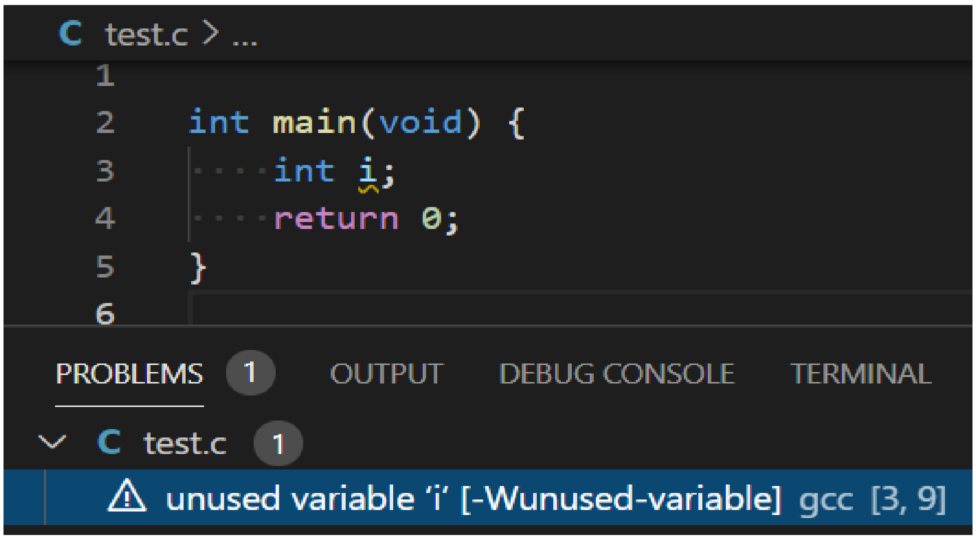
Figure 2: VSCode code navigation features

2.3.2 Restrictions check

Currently, **Integrated development environments** automatically perform some static checks for a wide range of constraints.

These checks take place in real time while the code is being typed and are presented to the user directly in the editor by means of error markers in the text.

The restrictions checked include common name binding errors such as unresolved references, duplicate definitions, use before definition and unused definitions.



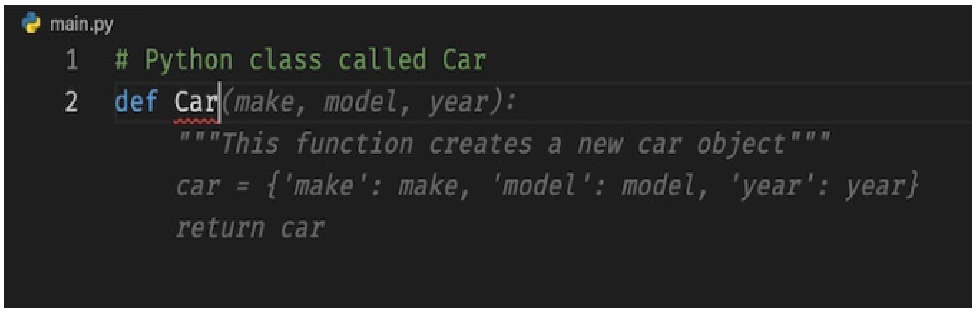
```

int main(void) {
    int i;
    return 0;
}

```

```
unused variable 'i' [-Wunused-variable] gcc [3, 9]
```

Figure 3: VSCode unused variable warning

2.3.3 Code completion

As far as name completion is concerned, there are already some services that fill in incomplete code with valid references in the context of code execution.



main.py

```
# Python class called Car
def Car(make, model, year):
    """This function creates a new car object"""
    car = {'make': make, 'model': model, 'year': year}
    return car
```

Figure 4: GitHub Copilot code completion sugestion

In this sense, *GitHub Copilot* has recently come to the foreground, as mentioned in Finnie-Ansley et al. (2022), as an artificial intelligence tool developed by *GitHub* in conjunction with *OpenAI*, to help users of **Integrated development environments** such as Visual Studio Code(VSC), InteliJ(IJ) and Neovim(NV), providing developers with code suggestions for automatic completion.

GitHub Copilot is an AI-powered coding assistant designed to help developers write code more quickly and effortlessly, enabling them to dedicate more time to problem-solving and collaboration. Copilot provides coding suggestions, ranging from completing the current line to generating entire blocks of code. It has also been shown to boost developer productivity and speed up the software development process.

2.4 Attribute grammars

According to Knuth (1968), attribute grammars are a formalism that makes it possible to specify the semantic analysis of a compiler and model complex traversal algorithms.

In the context of attribute grammars, programmers do not need to develop complex traversal functions since these mechanisms are generated by systems based on attribute grammars.

For example the **Backus-Naur form (BNF)** notation is a notation for specifying context-free grammars, generally used for specifying the exact grammar of programming languages. It was proposed by Backus (1959), to describe the language of what became known as *ALGOL 59*. In the following example, the **BNF** notation is used to describe the grammar of arithmetic expressions.

```
Exp  :  Exp ' + ' Exp
     |  Exp ' - ' Exp
     |  Exp ' * ' Exp
     |  Exp ' / ' Exp
     |  ' ( ' Exp ' ) '
     |  int
     |  id
```

This grammar is inherently recursive: an expression (**Exp**) can be defined as an expression, an arithmetic sign ('+') and another expression. An environment (**env**) is an *inherited attribute* of non-terminal (**Exp**) that will be used to pass downwards the value of each name used in an expression. A (**value**) is a a *synthesized attribute* of non-terminal (**Exp**) that computes (upwards) the value of an expression. The entirety of a grammar is expressed in this way, which is simple to understand and powerful when compared to embedding the grammar in the code. The **BNF** notation is extremely interesting as it laid the foundation to having the grammar separated from the code, such that it would be easy to change the grammar without having to change any of the remaining code. Usually attribute grammars are defined over the abstract representation of the language (grammar or abstract tree) and not over the concrete notation (which is needed for the parser only).

2.4.1 Zippers

The concept of Zippers was first conceived by Huet (1997) in order to make it possible to represent and navigate data structures uniformly, regardless of the data they represent.
In a Zipper, there is a focus of attention on the structures to be analyzed that is completely mobile, allowing movement in all directions.
Also, a Zipper is manipulated using a set of predefined functions, which allow generic access to all the nodes in the tree for consultation or modification.
In the work of Martins et al. (2013), we can see that it is indeed possible to combine Zippers and attribute grammars in order to create an adaptable, efficient and elegant mechanism for manipulating and traversing trees, without the need to design all the machinery required for this purpose. The library (ZAG) was created using Zippers as the core.
This library will be fundamental to this thesis, since the aim is to extend it according to the terms mentioned above.

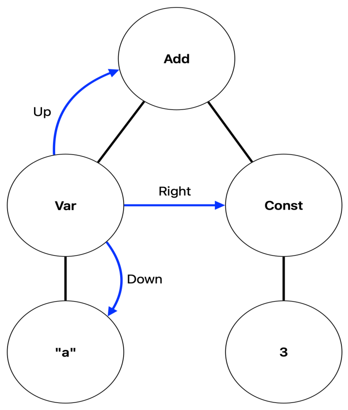


Figure 5: Visual representation of a Zipper

2.4.2 Strategic programming and name analysis with the Block processor

A strategy is a generic transformation function that can traverse heterogeneous data structures, combining uniform and type-specific behavior, as mentioned in the work of Saraiva (2007). Thus, in a Strategic Programming environment, only the nodes to be transformed are included in the solution, omitting the productions in which no changes need to be made. By resorting to this type of mechanism, we get a much more elegant and synthetic way of expressing transformations on trees. This concept will be extremely important in this project, since we will use the same line of thought and the library that models it to develop a new layer of abstraction on top of that. We are presented with the concept of a list-based language (**Block**), which consists of a possibly empty list of items. An item can be a declaration of a name, the use of a name or a nested block. This language allows the user to collect information about the use of names in a given piece of code, describing the behavior of most programming languages with regard to declarations and the use of variables. This approach guarantees a systematic analysis of the Scope rules in the **Block** language, including the identification of local definitions, the avoidance of duplicates and verification of the appropriate use of concisely and efficiently defined names.
However, it's not comprehensive enough and requires translating the source code into **Block** language to perform the checks. This transformation is feasible but has its problems, namely not being able to pinpoint where the error was originally located. Therefore, the intention is to employ the concept of how **Block** operates, but to conduct this analysis on the original data type as generically as possible, thus making this process universal and transparent.

Chapter 3

Zipper-based Travessal Functions

Attribute grammars are traditionally implemented by functions that traverse the underlying abstract syntax tree **AST** while computing attribute values. These evaluators, known as tree-walk evaluators, traverse up and down the tree to evaluate attributes. In a functional programming setting, Zippers provide a simple yet generic tree-walk mechanism. We will use Zippers to model attribute grammars directly in a Haskell program. In other words, Zippers will serve as the fundamental tool to embed attribute grammars within the general-purpose *Haskell* language, allowing us to execute our Scope Rules analysis.

3.1 Functional Zippers

Zippers were originally conceived by Huet (1997) to represent a tree together with a subtree that is the focus of attention. During a computation, the focus can shift to the left, up, down, or right within the tree. Zippers offer a set of predefined functions that enable generic manipulation, allowing access to all tree nodes for inspection or modification. To demonstrate how zippers work, let's look at a binary leaf-tree represented by a single data type called *Tree*:

```
data Tree = Leaf Int
          | Fork Tree Tree


t :: Tree
t = Fork (Leaf 3)
            (Fork (Leaf 2) (Leaf 4))
```

When we examine the tree *t* as a whole, we observe that each subtree occupies a specific location within the tree. Essentially, a subtree's location can be represented by the subtree itself along with the remaining part of the tree. Therefore, the rest of the tree serves as the context for that subtree. One approach to represent this context is by defining it as a path from the top of the tree to the node currently under focus, where the subtree originates. This implies that contexts and subtrees together define any

position within the tree. This approach establishes a framework where navigation is facilitated by the contextual information of a path applied to the tree.
For example, if we wish to put the focus in *Leaf 2*, its context in tree *t* is:

```
tree = Fork  (Leaf  3)
             (Fork  O
                    (Leaf  4))
```

In this context, let's consider that *O* points to the exact location where *Leaf 4* occurs within the tree. One approach to representing this context involves defining a path from the tree's root to the specific position receiving attention. For example, to locate *Leaf 2* within *tree t*, we navigate right (down the right branch) and then left (down the left one). In practical terms, this implies that given *tree t*, the path [right, left] suffices to indicate the desired location within the tree. This concept lies at the core of Zippers. By pairing paths with trees, we can represent any position within a tree. Moreover, this framework facilitates navigation: removing parts of the path brings us back to the tree's root, while adding information allows us to delve deeper into the structure. Using this idea, we may represent contexts as instances of the following data-type:

```
data Cxt a  =  Top
            |  L  (Cxt a)  a
            |  R  a        (Cxt a)
```

In this representation, a value (L c t) signifies the left part of a branch, where the right part is *t*, and the parent has context *c*. Similarly, a value (R t c) denotes the right part of a branch, where the left part is *t*, and the parent has context *t*. The value Top represents the top of the tree. Returning to our example, we can represent the context of *Leaf 2* in the tree as:

```
context :: Cxt Tree
context = L (R (Leaf 3) Top) (Leaf 4)
```

Essentially, this implies that the focus arises from navigating left within a subtree, where the right side consists of *Leaf 4*. This subtree is derived from moving right from the root tree, where the left side is *Leaf 3*. With the context of a subtree established, we can define a location within a tree as one of its subtrees along with its associated context.

```
type Loc a = (a, Cxt a)
```

We are now ready to present the definition of useful functions that manipulate locations on binary trees. First, we can define a function that goes down the left branch of a tree:

```
left_Tree :: Loc Tree -> Loc Tree
left_Tree (Fork l r, c) = (l, L c r)
```

This function accepts a tuple containing a tree and a context (a Loc), then generates a new tuple. In this new tuple, the left side of the tree becomes the updated tree, and a new context is constructed. This new context extends the previous one using a data constructor *L*, where the right side becomes the right side of the original tree. Similarly, we introduce a function that traverses down the right branch:

```
right_Tree :: Loc Tree -> Loc Tree
right_Tree (Fork l r, c) = (r, R l c)
```

Going up in the tree corresponds to remove "directions" from the path. The function *parent_Tree* goes up on a tree location:

```
parent_Tree :: Loc Tree ->  Loc Tree
parent_Tree (t, L c r) = (Fork t r, c)
parent_Tree (t, R l c) = (Fork l t, c)
```

This function presents two alternatives: when provided with a location whose context is on the left side *L*, it generates a new location where the right side of the context is transferred to a new tree, while the left side remains as the context *c*. Conversely, the second alternative operates oppositely when the location's context is on the right side *R*. Finally, we define a function *zipper_Tree* that constructs a tree location from a tree, initially defining an empty context.

```
zipper_Tree :: Tree -> Loc Tree
zipper_Tree t = (t, Top)
```

Also we define another function that extracts a tree from a tree location, by simply taking the first element of the location pair:

```
value_Tree :: Loc Tree -> Tree
value_Tree = fst
```

Having defined a zipper-based function to navigate in binary-trees, we can now focus the attention on subtree *Leaf 2* of *t_1* as follows:

```
leafWith2 :: Loc Tree
leafWith2 = left_Tree (right_Tree (zipper_Tree t_1))
```

After reaching the subtree that is the focus of our attention, various actions can be performed on it. One action could involve editing the subtree, such as decrementing its leaf value by one. Utilizing the zipper functions established thus far, this operation can be expressed as follows:

```
edit = let  (Leaf v, cxt) = leafWith2
            subtree'      = (Leaf (v-1), cxt)
       in   value_Tree (parent_Tree (parent_Tree subtree'))
```

with *edit* producing the expected result:

```
edited = Fork (Leaf 3)
              (Fork (Leaf 1)
                    (Leaf 4))
```

To conclude, we have demonstrated a zipper implementation tailored for the *Tree* data type, showcasing navigation and transformation capabilities on such trees. In the next section, we introduce a generic Zipper library in *Haskell*, extending this functionality to any data type.

3.2 Generic Zippers

Generic zippers are available as a *Haskell* library[1], which works for both homogeneous and heterogeneous data types. In order to show the expressiveness of this library we shall consider the heterogeneous trees consisting of nodes of two types *Prog* and *Tree*, as defined in 3.1. We define *rt_1* as follows:

```
data Prog = Root Tree

rt_1 :: Prog
rt_1 = Root t_1
```

In a simple functional implementation, traversing this tree requires two functions: one to handle the *Prog* node and another to recurse through *Tree* nodes. Generic zippers offer a consistent method for navigating these heterogeneous data structures without needing to identify node types. To navigate the tree *rt_1*, you must first wrap it using the function *toZipper :: Data a => a -> Zipper a*. This allows any data type with instances of the *Data* and *Typeable* type classes, which can be generated automatically, to be navigated using generic zippers.

```
t1' = Zipper Prog
t1' = toZipper rt_1
```

This function creates an aggregate data structure that is easy to traverse and update. For instance, we can shift the focus in *t1'* from the topmost node to the leaf containing the number 3, as follows:

```
leafWith3 :: Tree
leafWith3 = (fromJust . getHole . fromJust . down' . fromJust . down') t1'
```

[1] https://hackage.haskell.org/package/syz

The zipper library function *down' :: Zipper a -> Maybe (Zipper a)* navigates to the leftmost (immediate) child of a node, while the function *getHole :: Typeable b => Zipper a -> Maybe b* extracts the node currently in focus from a zipper. The library also includes a down function to go to the rightmost (immediate) child of a node, as well as *up*, *left*, and *right* functions to navigate in the respective directions. All these functions wrap their results in a *Maybe* data type to ensure totality. To simplify our example, we unwrap this using the *fromJust* function, though this approach doesn't check for totality and assumes the presence of valid results for each function call.
The function *leafWith3* navigates from the tree's root along the shortest path to the desired leaf. We can then navigate to that leaf after visiting all nodes in the tree as described by *fullVisit*, in a compositional style. To avoid using *fromJust* and achieve a more natural left-to-right writing/reading of tree navigation, we can adopt an applicative functional style, as shown by *fullVisit'*. Here, the predefined monadic binding function *(»=)* allows the value returned from one computation to influence the choice of the next one.

```
fullVisit :: Tree
fullVisit = ( fromJust.getHole.
              fromJust.left.
              fromJust.up.
              fromJust.left.
              fromJust.down.
              fromJust.down.
              fromJust.down') t1'
```

```
fullVisit' :: Tree
fullVisit' = fromJust \$ down' t1'
                 >>= down
                 >>= down
                 >>= left
                 >>= up
                 >>= left
                 >>= getHole
```

Obviously, not all tree paths are valid. Next, we present an unfeasible one:

```
noPath :: Maybe Tree
noPath =  (getHole . fromJust . left . fromJust . down') t1'
```

Using the generic zippers library as a foundation, we have created a set of simple combinators that enable programmers to write zipper-based functions in a way that resembles how attribute grammar (AG) writers work. In other words, we are embedding AGs in Haskell, leveraging zippers to mimic AG notation Martins et al. (2013). More precisely, the following combinators:

- The combinator "*child*", written as the infix function *.$* to access the child of a tree node given its index (starting from 1).

  ```
  (.$) :: Zipper a -> Int -> Zipper a
  ```

Thus, when using the basic combinators, *tree.$i* translates to applying *(fromJust . down')* tree once, followed by applying *(fromJust . right) i-1* times to the result. For instance, if we consider the following tree:

```
t' :: Zipper Tree
t' = toZipper (Fork (Leaf 3) (Fork (Leaf 2) (Leaf 4)))
```

then, we can access its second child as follows:

```
sndChild  :: Maybe Tree
sndChild  = getHole (t'.$2)
```

where *sndChild == Just (Fork (Leaf 2) (Leaf 4))*, corresponds to the same subtree as the following definition via basic combinators:

```
sndChild  = (getHole . fromJust . right . fromJust . down') t'
```

In this trivial example, the second child is also the rightmost child and could be accessed directly using the *down* function. However, this situation is not typical.

- The combinator *parent* to move the focus to the parent of a tree node, that corresponds to the *up* basic combinator.

  ```
  parent :: Zipper a -> Zipper a
  ```

- The combinators *.$<* (left) and *.$>* (right) navigate to the ith sibling on the left or right of the current node:

  ```
  (.$<) , (.$>) :: Zipper a -> Int -> Zipper a
  ```

- The combinator *(.|)* checks whether the current location is a sibling of a tree node, or not.

  ```
  (.|) :: Zipper a -> Int -> Bool.
  ```

These combinators are versatile and can be applied to define various zipper-based programs with an AG style. However, each program requires some specific boilerplate code. Moreover, this code can be automatically generated using template meta-programming techniques [2]. To utilize the generic zipper library effectively, the data types defining the underlying program need to implement instances of the *Data*

[2] https://hackage.haskell.org/package/template-haskell

and *Typeable* type classes. In Haskell, this can be easily achieved by utilizing the deriving primitive with the data type.

```
data Prog  = Root  Tree
             deriving (Data, Typeable)

data Tree  =  Leaf  Int
           |  Fork  Tree Tree
             deriving (Data, Typeable)
```

When traversing heterogeneous trees, it's essential to know the constructor of the node we're visiting. Consequently, we need to generate a new boilerplate function to identify the constructor of the root node of each subtree. This enables us to perform pattern matching in our zipper-based trees. These data types consist of three constructors, which we group into a single data type named *Constructor*. To prevent name conflicts, each constructor name is suffixed with the type's name. Thus, the resulting data type looks like this:

```
data Constructor  =  CRootT
                  |  CFork
                  |  CLeaf
```

Now, we can create the *constructor* function. This function matches the root node of the given zipper tree with the constructors of the original tree and returns one of the newly defined constructors.

```
constructor :: Typeable a => Zipper a -> Constructor
constructor a =  case  ( getHole a :: Maybe Tree) of
                   Just (Fork _ _)  ->  CFork
                   Just (Leaf _)    ->  CLeaf
                   otherwise        ->  case  (getHole a :: Maybe Prog)  of
                                          Just (Root _) -> CRootT
```

To access tree values in the original tree, such as when defining the local minimum in a leaf node, we define the *lexeme* function. This function computes the value stored in a leaf of a tree.

```
lexeme :: Zipper a -> Int
lexeme z =  case  (getHole z :: Maybe Tree) of
              Just (Leaf i) -> i
```

Lastly, to present the types of the zipper functions that conduct the tree-walk evaluation on *Prog* trees in a way that aligns better with AG notation, we define the *AGTree* type synonym as follows:

```
type AGTree a  = Zipper Prog -> a
mkAG = toZipper
```

Chapter 4

Zipper-based Embedding of Attribute Grammars

To make specifying scope rules easier, we'll focus on a simpler scenario where we ignore the potentially complex right-hand side of a name definition (like arithmetic expressions in our example). This section covers the zipper-based Strategic Attribute Grammars embedding introduced in Macedo (2022), which merges strategic programming and attribute grammars. Before we delve into the details of this embedding, let's consider a motivating example that involves two common language engineering techniques: language analysis and language optimization. We will look at the sublanguage of **Let** expressions, which are included in most functional languages, such as *Haskell*. Here is an example of a valid *Haskell* let expression.

```
p = let a = b + 0
        c = 2
        b = let c = 3
            in b + c
    in a + 7 − c
```

We define the heterogeneous data type **Let** that we use to model let expressions in *Haskell* itself. We take this definition from previous work with strategies and attribute grammars in Macedo (2022).

```
data Let = Let List Exp
         deriving (Show, Data, Typeable)

data  List
      = NestedLet   String Let List
      | Assign      String Exp List
      | EmptyList
         deriving (Show, Data, Typeable)

data Exp = Add Exp Exp
         | Sub Exp Exp
         | Neg Exp
```

```
        | Var String
        | Const Int
         deriving (Show, Data, Typeable)
```

4.1 The Zipper-based Block Program

The formal specification of scope rules is a core concept in the Attribute Grammar (AG) formalism Knuth (1968). AGs are especially effective for language engineering tasks that require gathering and utilizing context information. We start by specifying the scope rules for **Let** expressions using an AG. We use a visual AG notation, which is often employed by AG writers to draft their grammars. The scope rules for **Let** are visually depicted in 4.1. Additionally, we introduce a type Root to identify the root of the tree.

```
data Root = Root Let
```

The diagrams in the figure are read as follows: For each constructor or production (labeled by its name), we display the type of the production above it and the types of its children below. Inherited attributes, computed top-down, are listed to the left of each symbol, while synthesized attributes, computed bottom-up, are listed to the right. Arrows between attributes indicate the information flow needed to compute an attribute. Thus, the AG depicted in 4.1 functions as follows: The inherited attribute **dcli** acts as an accumulator to gather all names defined in a **Let** expression. Initially empty in the *Root* production, when a new name is defined in the *Assign* and *NestedLet* productions, it's added to the accumulator. The total list of defined names is synthesized into the **dclo** attribute, which is then passed down as the environment (inherited attribute **Env**) at the *Root* node. Additionally, a nested let inherits the environment of its outer let through the **dcli** attribute. These attributes' types are lists of pairs, associating each name with its **Let** expression definition. Using zipper-based AG combinators, we can express the scope rules of **Let** in an AG programming style. For instance, consider the synthesized attribute **dclo**. In our visual AG diagrams, we observe that in the *NestedLet* and *Assign* productions, **dclo** is defined as the **dclo** of the third child. Moreover, in the **EmptyList** production, the **dclo** attribute is a copy of **dcli**. This mirrors how such equations are formulated in the zipper-based AG. This attribute returns a value of type **Env**, representing a list of names with their associated nesting level and definition. The **Env** type is defined as the following type synonym:

```
type Env = [(String, Int, Maybe Exp)]
```

In most diagrams, the attribute **Env** is typically defined as a copy of its parent attribute. However, there are two exceptions: in the productions *Root* and *NestedLet*, where **Env** obtains its value from the synthesized attribute **dclo** of the same non-terminal or type.

Now, let's establish the equations for the inherited attribute **dcli**. As depicted in 4.1, the initial list of declarations, **dcli**, at the root of the tree is an empty list, reflecting the context-free nature of the outermost block. The attribute **dcli** of **Let** recurs in the *NestedLet* production, where it is defined as the inherited attribute of the parent. Similarly, the *List* nonterminal inherits **dcli**. In 4.1, we observe that *List* appears in the **Let**, *Assign*, and *NestedLet* productions, each with distinct equations. For instance, in *Assign*, the **dcli** attribute of *List* is defined by augmenting the defined name (String) with the **dcli** of the parent.

The inherited attribute **lev** follows a straightforward definition: it initializes with a value of 0 at the root and increments when passed to a nested **Let**. In other cases, **lev** simply mirrors its parent's definition.

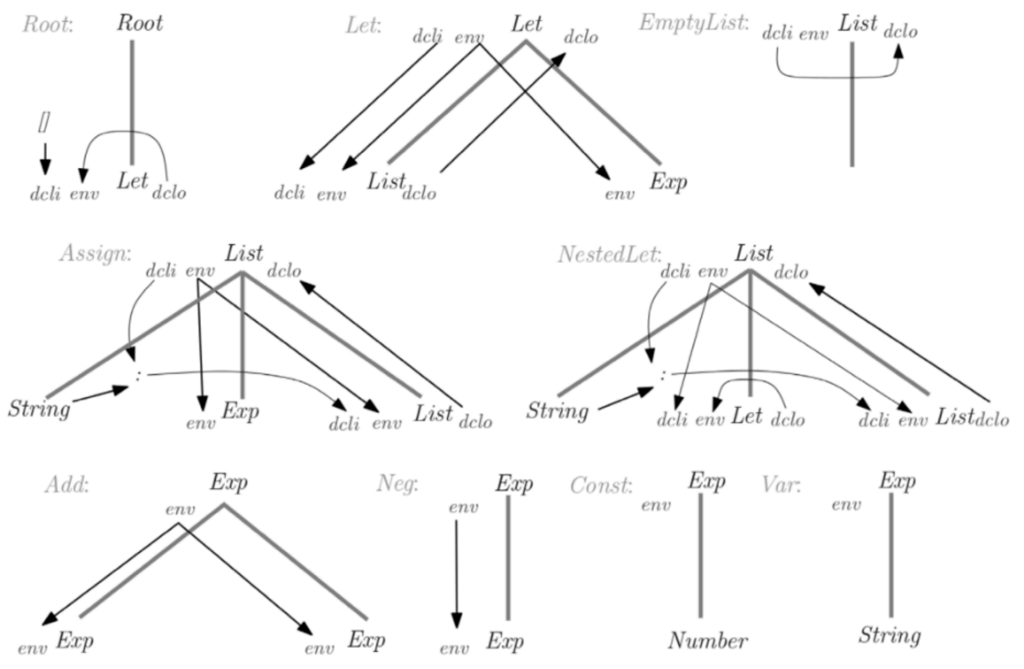


Figure 6: Attribute grammar specifying the scope rules of Let.

4.2 Higher-Order Attribute Grammars for the Block Program

Name analysis employs the scope rules of a language to associate uses of identifiers with their definitions. To simplify, we'll focus on the (sub)language of *let* expressions, common in functional programming languages like *Haskell*, using the previously explored example. Although concise, the **Let** language embodies fundamental features of programming languages, including block-based structures (including nesting) and the requirement for unique name declarations. Moreover, in the semantics of **Let**, there's no enforced declare-before-use rule, allowing variables to be declared after their initial use. Below is an example program written in the **Let** language, equivalent to code in the **Block** language.

```
let w = b + -16                      [decl w , use b
    a = 8                            , decl a
    w = let z = a + b                , decl w, [decl z, use a, use b
        in z + b                     , use z , use b ]
    b = c + 3 - c                    , decl b , use c, use c
in c + a - w                         , use c , use a, use w ]
```

As demonstrated earlier, AGs provide a modular and extensible software development environment, allowing new extensions to be seamlessly integrated without altering the existing solution. Furthermore, in our embedding, each added module can be compiled independently. However, AGs face a significant limitation: when an algorithm cannot be easily expressed using the underlying AG data structure, alternative structures cannot be utilized or computed. This limitation becomes apparent in tasks such as name analysis for *let* expressions, as outlined in 4. Due to the simplicity of the **Block** data type/structure, we expressed the complex scope rules of *let* expressions within the data type *P*. Recognizing this constraint, Swierstra introduced Higher-Order Attribute Grammars Vogt (1989), where conventional AGs are enhanced with higher-order attributes, known as attributable attributes. Higher-order attributes are attributes whose value is a tree, enabling the association of attributes with such trees once again. Attributes of these higher-order trees may also be higher-order attributes. Higher-order attribute grammars exhibit four main characteristics:

- Firstly, when a computation cannot be easily expressed based on the inductive structure of the underlying tree, it's possible to compute a more suitable structure beforehand. This enables transformations such as converting a *let* expression into a **Block** program, specifying name declarations and usage. The attribute equations define a synthesized higher-order attribute representing the **Block** tree. Consequently, decorating a **Let** tree results in constructing a higher-order tree: the

Block tree. The attribute equations of the **Block** AG define the scope rules of the **Let** language.

- Additionally, semantic functions are no longer required. Higher-order attribute grammars offer a framework where every computation can be captured through attribution rules alone. Essentially, inductive semantic functions can be replaced with higher-order attributes. For example, a common use case involves using higher-order attributes to represent the recursive lookup function in an environment. This eliminates the need for a separate notation or language to define semantic functions within AGs. Moreover, since we express inductive functions using attributes and attribute equations, the termination of these functions is statically validated using standard AG techniques like the circularity test.

- The third feature is that a portion of the abstract tree can be directly employed as a value within a semantic equation. In simpler terms, symbols from the grammar can transition from the syntactic realm to the semantic realm.

- Finally, as we advocated in Vogt (1989), attribute grammar components can be "glued" via higher-order attributes.

To articulate the name analysis task of let expressions, we leverage the primary feature outlined in Vogt (1989): synthesizing a more appropriate tree structure where scope rules are more straightforward to articulate. Given that these rules are already defined within the **Block** AG, our task involves associating attributes and equations with **Let** symbols and productions to synthesize a higher-order tree of type *P*. This attributable attribute is then enriched by the **Block** AG.

We commence by defining a (first-order) AG fragment where we synthesize a list of declarations and uses of names in a List of let expressions. Consequently, we formulate attribute equations in a manner where a *Var* constructor triggers a *Use* of the corresponding name, while an *Assign* induces a *Decl* of that name. Subsequently, we present the zipper-based definition of the requisite equations.

```
letAsBlock :: AGTree Its
letAsBlock t = case constructor t of
    AssignList →     ConsIts (Decl (lexeme Name t))
                     (concatIts (letAsBlock (t.$2)) (letAsBlock (t.$3)))
    NestedLetList →  ConsIts (Decl (lexeme Name t))
                     (ConsIts (Block (concatIts (letAsBlock (t.$2))
                                                (letAsBlock (t.$3))))
                     NilIts)
    EmptyListList →  NilIts
    ConstExp →        NilIts
    VarExp →          ConsIts (Use (lexeme Name t)) NilIts
    NegExp →          letAsBlock (t.$1)
    _ →               concatIts (letAsBlock (t.$1)) (letAsBlock (t.$2))
    where concatIts ≡ concatErrors
```

Next, we establish an attributable attribute within the **Let** production to generate the intended **Block** higher-order tree. This attribute, dubbed ata, requires decoration according to the **Block** scope rules. Initially, we transform this higher-order tree into a zipper using the mkAG function. Subsequently, we employ any zipper-based AG function to compute the attributes of this tree.

```
letErrors :: AGTree Errors
letErrors t = case constructor t of
        LetLet → let ata :: Zipper P
                     ata = mkAG (Root (letAsBlock (t.$1)))
        in errors ata
```

It's important to highlight that the **Block** AG serves as a readily available AG component seamlessly integrated into the **Let** AG specification. This integration underscores the potency and incremental nature of higher-order AGs in language design and implementation.

Chapter 5

Expressing Scope Rules with Zippers

The Scopes Interface was conceptualized to streamline the intricacies of the mechanisms and rules associated with name analysis and Scope rules verification. This Interface corresponds to a *Haskell* class that allows the user, for any language, to specify how to use and declare names, as well as nesting and global names. Based on this information and the choice of a set of predefined rules (declare-before-use, Algol 68, object-oriented), this interface is able to analyze any piece of code, inferring any Scope errors and flagging them in the original **AST**.
This is only possible because the interface has the ability to build a higher-order block tree and calculate the scope errors of that tree while calculating the path for each error in the original tree, merging the results of the two mechanisms at the end to obtain the desired result.

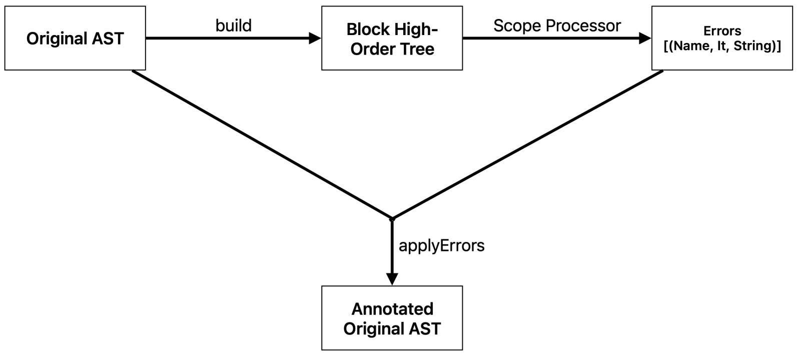


Figure 7: Relational diagram for the Scopes Interface

The original **AST** is processed by the **build** function to build the higher-order Block tree, which is then analyzed by the Scope processor. The result of this analysis is then applied to the original **AST** in order to note Scope errors in it.
In order to illustrate the use of this Interface let us consider the (sub)language of Let expressions, explored in chapter 4 of this document, that uses the Algol 68 set of Scope Rules.

5.1 Specifying Def/Use of Names and Nesting via type Classes

In order to use this interface, the user must first state how to define declarations, uses, nesting and global variables in the language being analyzed. To do this, the user needs to create an instance of the Scopes class and fill it in with the desired correspondences.

```
class (Typeable a, StrategicData a, Data a) => Scopes a where
    isDecl :: Zipper a -> Bool
    isUse :: Zipper a -> Bool
    isBlock :: Zipper a -> Bool
    isGlobal :: Zipper a -> Bool
    isGlobal _ = False
    getUse :: Zipper a -> String
    getUse a = getString a
    getDecl :: Zipper a -> String
    getDecl a = getString a
    setUse :: Zipper a -> String -> Zipper a
    setUse a b = modifyFunc a b
    setDecl :: Zipper a -> String-> Zipper a
    setDecl a b = modifyFunc a b
    initialState :: a -> [String]
    initialState = const []
```

This class provides the template to create an instance to each language. The user has to specify these three functions for their language, having the possibility to change the default behaviour for the rest, describing how it handles their naming rules. This simple pattern-matching gives every information the Scopes Interface needs to identify relevant nodes from the original tree. The example below gives a better idea of what is required and how the user can do it.

```
instance I.Scopes Let where
    isDecl ag = case constructor ag of
        CAssign -> True
        CNestedLet -> True
        _ -> False
    isUse ag = case constructor ag of
        CVar -> True
        _ -> False
    isBlock ag = case constructor ag of
        CLet -> True
        _ -> False
```

This code defines an instance of the Scopes class for the **Let** data type. Each **Assign** and **NestedLet** represent declarations and are therefore marked in the **isDecl** function as such. The remaining functions follow the same principle, but for the other variants. In the **isUse** function, it is described that the **Var** type corresponds to a use of a name. In the **isBlock** function, it is described that the **Let** type corresponds to a Scoping block. The **isGlobal** function is unsigned, since this language does not include global variables or classes. This nuance will be covered in more detail later in this document.

The **getUse** and **getDecl** functions define how the interface can derive information about the names of the data types under study from the original tree in order to build the higher-order Block tree. If the user does not define their own implementation, the interface is equipped with a default implementation, **getString**, which searches for the first String of each type.

The **setUse** and **setDecl** functions specify how the interface can modify the nodes of the original tree in order to note Scope errors in the latter. If the user doesn't define their own implementation, the interface is also equipped with a default implementation, **modifyFunc** which will be addressed later, that searches for the first String of each type and concatenates it with the specific error type.

Finally, the **initialState** function defines all names that the user wants to give as previously defined during the analysis of names and Scope rules and it is empty by default. In real cases, this can represent the use of external libraries or predefined functions of the language under analysis, in the case of *Haskell*, the (*show*) and (*++*) functions for example.

```
#initialState = const []                  #initialState = const ["b"]
a = b <= [Undeclared use!] + 3            a = b + 3
```

Following on from the construction of the instance of this interface, it is necessary to point out which types of data should be skipped during the traversal for the construction of the (higher-order) **Block** tree, as they have no relevance to the analysis in question. Terminal symbols of a grammar are usually handled outside the formalism. They are often specified via regular expressions, and not by grammars. Moreover, they are efficiently processed via efficient automata-based recognizers. To allow this behaviour, any data type to be traversed using this library must define an instance of *StrategicData* Macedo (2024). Any instance of *StrategicData* must define which nodes are considered terminal symbols and thus should be skipped.

```
instance StrategicData (Let) where
    isTerminal t = isJust (getHole t :: Maybe Int)
                || isJust (getHole t :: Maybe Name)
```

It is trivial to infer why the integer data type is not included in the case study, since the use of this data type does not have to comply with any Scope rules. However, the **Name** data type, which is a synonym for **String**, is also skipped. Although it may seem contradictory, since names are always represented by Strings in any language, what matters to us is the parent data type of each of these names because that is what defines whether they are uses or definitions, for example. This definition makes it possible to prevent the duplication of information in the construction phase of the higher-order **Block** tree.

5.2 Generic Traversal to Build the Block (higher-order) Tree

Ultimately, this function merely defines how the correlations described above should be handled, taking advantage of the "buildChildren" function, which is a fully generalized function that for any Zipper, build function and empty list of directions returns the corresponding higher-order **Block** tree.

```
build :: Scopes a => Zipper a -> P
build a = Root (buildChildren build' a [])

build' :: Scopes a => Zipper a -> Directions -> Its
build' a d | isDecl a  = ConsIts (Decl (getDecl a) d)
                              (buildChildren build' a d)
           | isUse a   = ConsIts (Use (getUse a) d)
                              (buildChildren build' a d)
           | isBlock a = ConsIts (Block $ buildChildren build' a d)
                              NilIts
           | otherwise = (buildChildren build' a d)
```

This predefined function returns the higher-order **Block** tree corresponding to the piece of code to be analyzed, which then just has to be passed as an argument to the processor you want to choose according to the rules of your language. This function can also be defined by the user if the requirements of the language in study are more intricate.

```
ag68_processor a = block (string2Env $ initialState a) (toBlock a)
```

All predefined Scope processors are prepared to receive a list of strings defined by the user, **initialState**, that is then derived as a global **ENV** by the **string2Env** function and used for the further analysis.

Finally, the errors and the original tree are passed to the function that applies the errors to obtain the expected result on the 4.2 program.

```
let w = b + -16
    a = 8
    w <= [Duplicated declaration!] = let z = a + b
                                     in z + b
    b = c <= [Undeclared use!] + 3 - c <= [Undeclared use!]
in c <= [Undeclared use!] + a - w
```

The processor found the scope errors in the piece of code to be analyzed and annotated the original tree with the respective errors. The variable "w" is declared twice and that the variable "c" is used several times without any kind of declaration.
Although this result is surprising for the little code the user has had to write, what is really remarkable is that this process has been fully generalized to be able to accommodate any language using this interface. All this is only possible because of the generic functions that have been produced for this purpose and which do all the work of constructing the higher-order tree and annotating the original tree with the corresponding errors.

5.2.1 Generic Traversal Functions

In order to make this interface as transparent and intuitive as possible, generic functions have been built that automate the entire process of building the higher-order block tree and annotating errors in the original code. In this section, we'll go into more detail about the generic construction functions, as they are the first part of the algorithm.
The first of these functions is **mergeIts** that simply merges two block instructions.

```
mergeIts :: Its -> Its -> Its
mergeIts (NilIts) its = its
mergeIts (ConsIts (Block NilIts) xs) its = mergeIts xs its
mergeIts (ConsIts x xs) its = ConsIts x (mergeIts xs its)
```

This function is the glue of the whole iterative construction process, since all the relevant original types are translated into single block instructions which then have to be composed with all the others.
The core of the entire construction process is the **buildChildren** function, which traverses the entire original tree, in this example **build**, merging all the translations into one and accumulating the path along which the construction traversal took place.

```
buildChildren buildFunc ag d = foldr mergeIts B.NilIts (children buildFunc ag d)
```

At each node in the original tree, the function is called recursively on its children via the **children** function, which tries to move down from the node (the place where the first child of a node is located in an **AST**). If this move is successful, the build function is applied to the child and called on its siblings, creating a list of simple block instructions that will then be processed by **mergeIts**, otherwise the function just returns an empty list. Remember that during this process each move is accumulated in the list of traversal directions.

```
children buildFunc ag d = case down' ag of
  Nothing -> []
  Just n -> do
    let x = d ++ [D]
    buildFunc n x : brothers buildFunc n x

brothers buildFunc ag d = case right ag of
  Nothing -> []
  Just n -> do
    let x = d ++ [R]
    buildFunc n x : brothers buildFunc n x
```

The "brothers" function works in a similar way to "children" but tries to move to the right of the node, where the brothers of each node in an **AST** are located.

5.3 Zipper-based Path to Access attribute in the Block (higher-order) Tree

Initially, the interface was only capable of detecting errors on the higher-order tree, but it was not able to find them in the original tree. To this end, a mechanism was developed to build paths, add these paths to the higher-order tree and annotate errors in the original tree using these paths.
The idea would be to build and save the path for each node translated in the process of building the higher-order tree. This way, any error found would have the correct path reference from the root in the original tree. From there, the annotation becomes trivial, since it is only necessary to traverse the corresponding path and change the node where the error exists by adding a message according to the specific type of error.

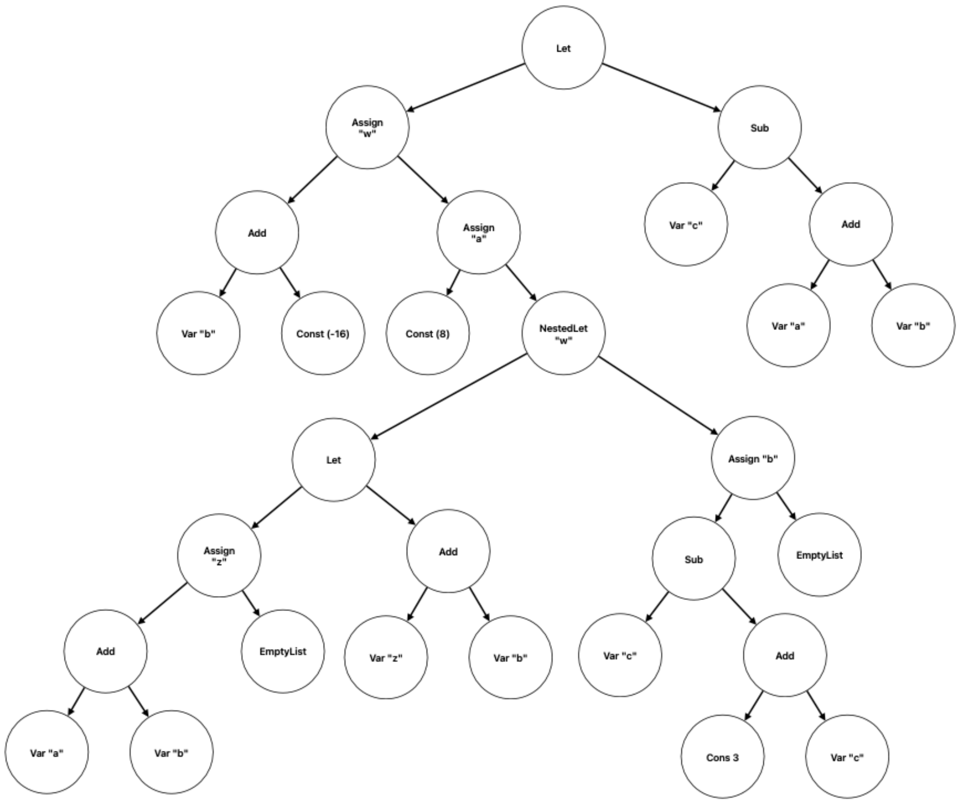


Figure 8: Let program Abstract Syntax tree

This mechanism begins in the construction phase with the functions discussed in the last chapter, *children* and *brothers*. These functions accumulate the traversal directions of the higher-order tree construction process. When one of the nodes in the original tree ends up being translated, the information about the path taken so far is added to the respective higher-order node.

```
data It = Decl Name Directions
          | Use Name Directions
          | Block Its
          deriving (Typeable , Data , Eq)
```

This method made it possible to unblock the process of annotating errors in the tree, once again in a completely generic way and without the user having to specify any more information than in a previous phase when this annotation process was not possible.

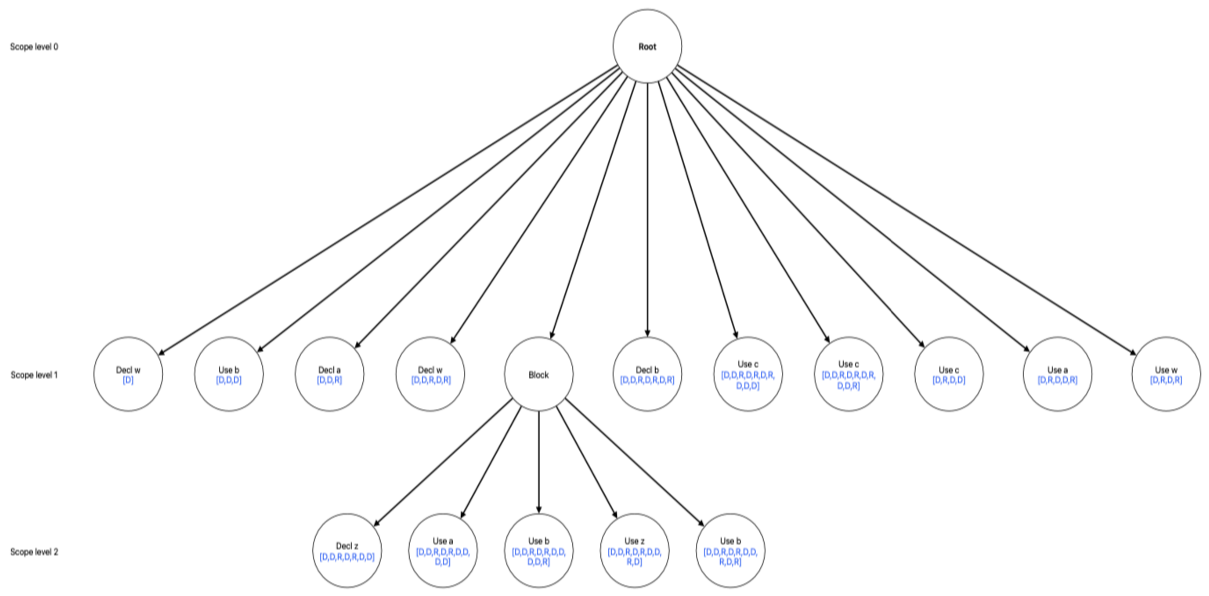


Figure 9: Let program higher-order tree

For each error found after the construction and analysis phase of the Scope rules, it is then necessary to go through the respective path from the root and write down that error. To this end, some generic functions have been built which, based on any **AST** and a list of errors, *(type Errors = [(Name, It, String)])*, are able to carry out the entire error annotation process.

The core of this phase is processed by the **applyErrors'** function, which performs the iterative process of going through and annotating the original tree for each error.

```
applyErrors' ag [] = ag
applyErrors' ag ((a,b):t) =
             applyErrors' (mkAG $ fromZipper $
                                        (modifyZipperAlongPath ag a b)) t
```

One detail to note is that at each iteration the original tree is transformed from Zipper to **AST** and from **AST** back to Zipper. To an attentive reader this may be redundant code, but there is a reason for this. At each iteration, the original tree must be returned to the function in its changed state, but with the focus on the root and not on the node that has just been changed. If this small transformation didn't take place, the next errors would be written down in a different place, since they would be traveling along a different path, not starting from the root, but from the last changed node. This small arrangement solves this very problem.

During this iterative process, the **modifyZipperAlongPath** function receives the original tree, a path and the error message to be annotated and calls two other functions, **applyDirections**, which traverses the given path and the **modifyFunc** function, which modifies the node of the original tree where the error found should be annotated.

```
modifyZipperAlongPath ag (_, Use   _ d, e)  = setUse  (applyDirections ag d) e
modifyZipperAlongPath ag (_, Decl  _ d, e)  = setDecl (applyDirections ag d) e
```

The **applyDirections** function uses the functions ("down", "right", "left", "up") from the *StrategicData* library to traverse the desired path if it is possible. These functions are affected by the instance of *StrategicData* that the user has specified, so that these traversals do not enter nodes that are not relevant to this analysis.

```
applyDirections ag [] = ag
applyDirections ag (h:t) = case movement ag of
  Nothing -> error "cant␣go␣there"
  Just n -> applyDirections n t
  where movement = case h of
                    D -> down'
                    R -> right
                    L -> left
                    U -> up
```

When this path is traversed and the desired node is found, the **modifyFunc** function changes it, if the user does not define a specific implementation for this type of transformation via **setUse** and **setDecl**. Although it seems trivial, this procedure turns out to be a little complex since the idea is to be completely generic, and therefore some additional care is needed to avoid errors in this process and ensure that the annotation is carried out in the correct place and in the proper way.

```
modifyFunc ag s = case down' ag of
 Nothing -> error "Error␣going␣down␣on␣modifyFunc"
 Just n -> case (getHole n :: Maybe String) of
             Just holeString -> setHole (holeString ++ s) n
             Nothing -> aux n s

aux ag s = case right ag of
 Nothing -> error "Error␣going␣right␣on␣modifyFunc"
 Just n -> case (getHole n :: Maybe String) of
             Just holeString -> setHole (holeString ++ s) n
             Nothing -> aux n s
```

The line of thought is that when the node is found we have to find the first string of that node in order to concatenate the *String* with the error. To do this, we need to check whether its first child is of type *String* and, if it is, modify that child using the generic Zipper function *setHole*, which allows us to modify the focus of a Zipper according to the user's needs. Otherwise, the check is extended to its siblings until the annotation is completed.

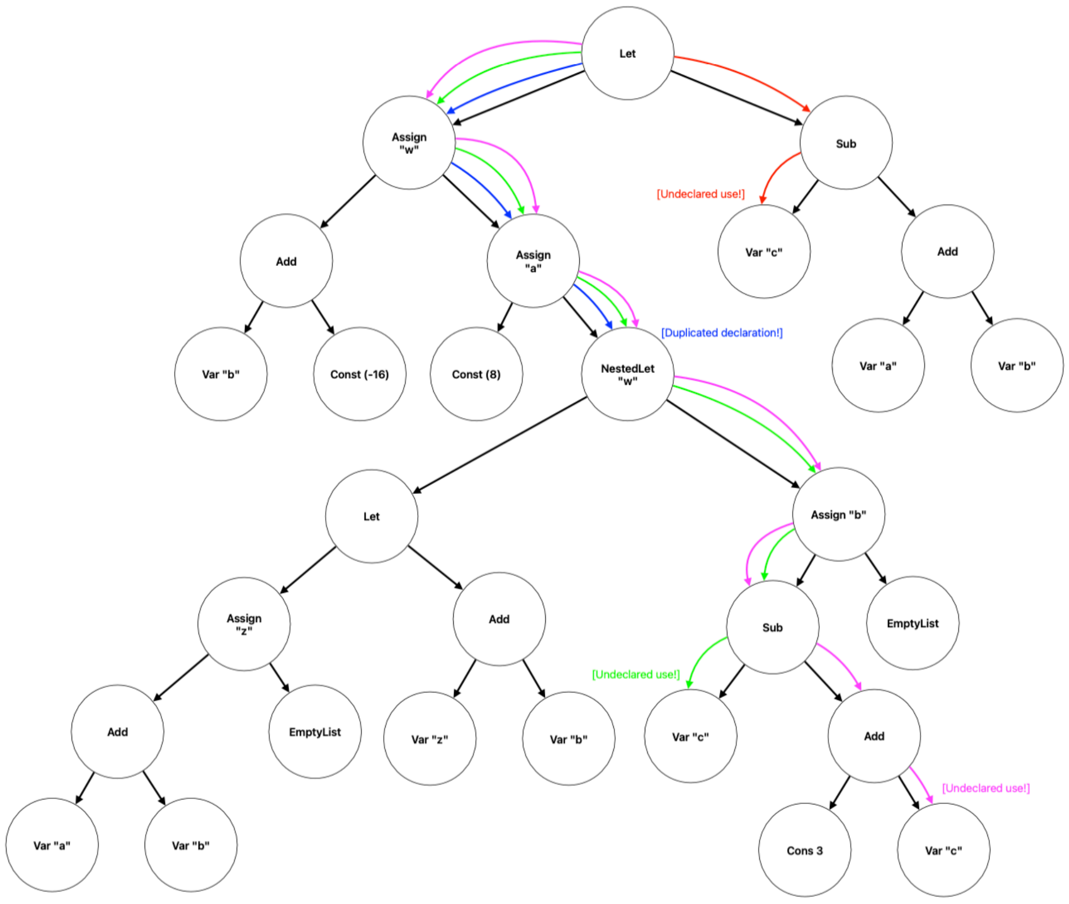


Figure 10: Let program error paths in the original AST

With this annotation process defined, fully automated and generic, the processor is able to flag errors in the original tree without any user intervention. This makes the whole process completely transparent and generic, since just by defining the rules described above, the rest of the process becomes completely automatic, merging the parsing process with analysis and annotation in a completely seamless way and without any loss of information about the language being analyzed and automatically inferring the entire process of building the higher-order tree as well as analyzing names and Scope Rules. Note that this process only works for languages with Scope Rules similar to declare-before-use or Algol 68. However, class mechanisms or global variables are not covered in this analysis. To this end, other mechanisms have been built to deal with these intricacies, which will be discussed in later sections. The problem associated with these object-oriented paradigm mechanisms is that in any scope, the processor must be aware of variables, functions or even classes that have been declared, even if they are not in the environment under analysis.

5.4 Predefined Scope processors

To make the whole process more agile and transparent for the user, three types of Scope processors have been built, corresponding to three programming paradigms (functional, imperative and object-oriented). These pre-defined processors mean that the user doesn't have to think about the intricacies of name analysis or Scope rules, but simply choose which set of rules best suits their language according to the paradigm in which they are included.
In this way, the user can choose to use existing presets or develop a processor that best meets the needs of their language, while still being able to take advantage of all the interface and machinery developed for building the higher-order tree and annotating the original tree because both are completely generic. This architecture allows the library to be expanded completely in an ad-hoc way. This line of thinking makes the whole topic of analyzing names and Scope rules much simpler because we've reduced the complexity to five attributes. Once the user understands these attributes and is equipped with the generic functions of the interface, it becomes trivial to derive the whole process of analyzing names and Scope rules for any language.

5.4.1 Algol 68 Scope Processor

The Algol 68 Scopes processor is equipped to obey the Algol 68 rules that many modern day languages follow. These have already been covered in more detail in 4. However, the core of this set of rules is based on the idea that any variable used must have a declaration in the Scope it is in or in the ones it inherited. Algol 68 (short for Algorithmic Language 1968) is an imperative programming language that was conceived as a successor to the Algol 60 programming language, designed with the goal of a much wider scope of application and more rigorously defined syntax and semantics. Many languages of the 1970s trace their design to Algol 68, among these is the language *C*, which was directly influenced by Algol 68, especially by its strong typing and structures. Most modern languages trace at least some of their syntax to either *C* or *Pascal*, and thus directly or indirectly to Algol 68. Each environment is influenced by its parent environment, for each use there must be a declaration in its environment and for each declaration there must be no duplicates in the attributes inherited by its parents. This processor has been developed and tested using the **Let** language explained above. To use this predefined processor the user simply needs to call it from the **IScopes** interface, calling the **processor_a68** function on the original **AST**.

5.4.2 Declare-Before-Use Scope Processor

The Declare-Before-Use Scope processor, as the name suggests, defines that variables or functions can only be used if they have been previously declared. This processor tries to imitate the Scope rules of languages like *C*, although it doesn't include global variables.

```
b = 2                                b = 2
c = a <= [Undeclared use!] + b       a = 3
a = 3                                c = a + b
```

In this sense, the Declare-Before-Use Scope processor is practically the same as Algol 68, the only difference being the search for errors regarding the use of variables and functions, since they don't look for declarations in their environment, but rather in the attributes inherited from their parents.

```
errors a t =  case constructor t of
        CUse -> mustBeIn (BS.lexeme t) (fromJust $ getHole t) (dcli a t) a
        ...
```

All the other attributes in the attribute grammar defined in previous chapters behave in the same way. This is once again proof of the expressiveness of these attributes and how the analysis of names and Scope rules becomes a transparent practice using this interface and line of thought. To use this predefined processor the user simply needs to call it from the **IScopes** interface, calling the **processor_io** function on the original **AST**.

5.4.3 Object-Oriented Scope Processor

Although the Algol 68 and Declare-Before-Use processors allow you to do simple name analysis and Scope rules, they are not prepared to deal with various challenges that emerge in object-oriented languages such as *Java* or *C++*. These challenges mainly concern the concept of classes and global functions or variables. In order to make this interface more complete and adaptable to real-world problems, the Object-Oriented Scope Processor was created. This processor has the ability to collect all global variables and functions as well as classes before analysis in order to meet the needs of this paradigm. The idea is to merge the algorithm of the processors previously analyzed with this concept of global Scope. So we add another phase to the beginning of the algorithm where we go through the entire original **AST** collecting all the global variables and functions as well as the classes in a list which is then given to the Scope rule processor as the argument, thus representing a global **ENV** concept. This global **ENV** will then be used in the several Scopes verifications, being concatenated with the local **ENV** of the verification focus to create is real **ENV**.

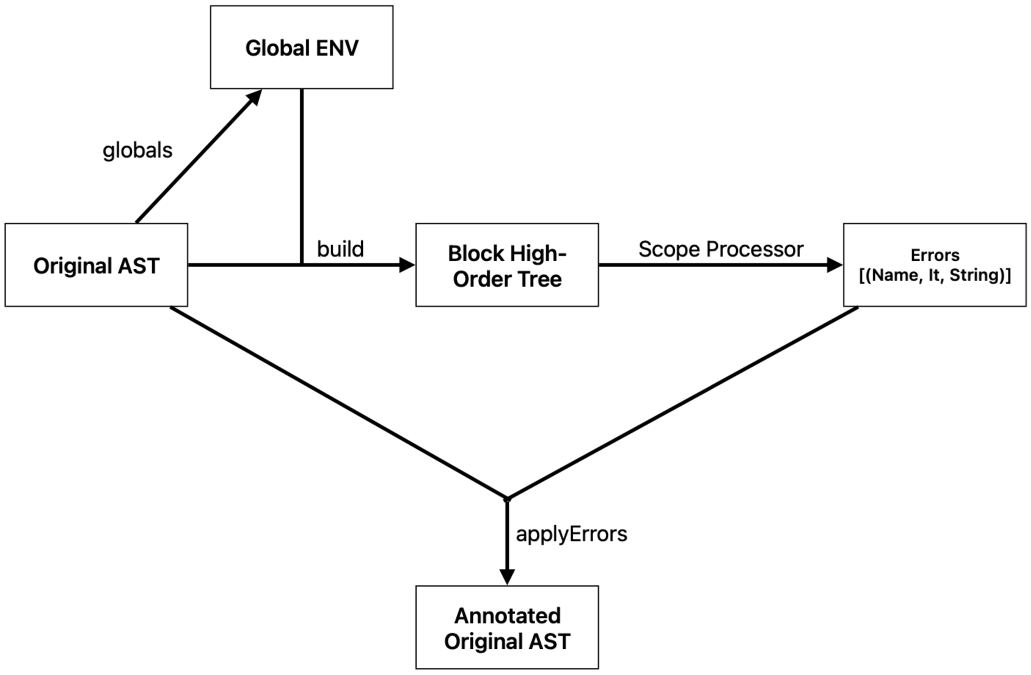


Figure 11: Relational diagram for the Scopes Interface using the Object-Oriented Scope Processor

In order to verify this theory, a model language was created with the notions of classes and global variables, **Toy-Java**.

```
data Exp = Add Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
         | Mul Exp Exp
         | Less Exp Exp
         | Greater Exp Exp
         | Equals Exp Exp
         | GTE Exp Exp
         | LTE Exp Exp
         | And Exp Exp
         | Or Exp Exp
         | Const Int
         | Var String
         | Return Exp
         | Bool Bool
         deriving (Show,Eq,Data)

data Item = Decl String Exp
```

```
          | Arg Exp
          | Global Item
          | Let Items Exp
          | Funcao String Items
          | DefFuncao String Items Items
          | If Exp Items
          | While Exp Items
          | DefClass String Items
          | Class String Item
          deriving (Show,Eq,Data)

data Root = Root Items deriving (Show,Eq,Data)

data Items = ConsIts Item Items
           | NilIts
           deriving (Show, Data ,Eq)
```

This language has most of the expressions used in the real world, as well as the concept of functions, classes and global variables. Let's consider this simple example for our analysis in which we declare two classes (Class1 and Class2). In the first class declaration we instantiate an object of the class (Class2), just like in the second. In the declaration of the class (Class2) we instantiate an object of the class (Class2). It is trivial to note that the Scope error in this code excerpt is the declaration of two classes with the same name. However, the implementation of a processor capable of recognizing this type of error by checking names outside its **ENV** requires a global environment recognition mechanism to be able to perform such verifications, which are quite common in most languages of the object-oriented paradigm.

```
Class Class1 {
        Class2 a
}

Class Class1 {
        Class2 b
}

Class Class2 {
        Class1 c
}
```

Listing 5.1: Program TJ

```
[ [ (Decl Class1 | Path: [D]),
    (Use Class2 | Path: [D,D,D]) ],
[ (Decl Class1 | Path: [D,R,D]),
    (Use Class2 | Path: [D,R,D,D,D]) ],
[ (Decl Class2 | Path: [D,R,D,R,D]),
    (Use Class1 | Path: [D,R,D,R,D,D,D]) ] ]
```

Listing 5.2: High-order Block tree of Program TJ

The goal is therefore to detect these errors and process the resolution of the remaining names, checking that they are processed correctly. Although it seems like a trivial check, a lot of intricacies relating to name analysis can be found in this short code excerpt. So let's first look at the changes made to the user interface to accommodate these changes. Incredibly, on the user side, it is only necessary to define one more function to recognize the global variables, give the result of this function over the original **AST** as an argument to the processor and identify the global datatypes in the Scopes interface instance.

```
instance Scopes Items where
(...)
isGlobal ag = case (constructor ag) of
             CGlobal   -> True
             CDefClass -> True
             _         -> False

globals' a = Root $ buildChildren globals a []

globals a d | isGlobal a = case constructor a of
                                CGlobal -> ConsIts (Decl (lexeme a)
                                    (d++[D])) (buildChildren globals a d)
                                CDefClass -> ConsIts (Decl (lexeme a) d)
                                    (buildChildren globals a d)
            | otherwise = buildChildren globals a d
```

This simple function creates a higher-order **Block** tree that represents the environment of the global variables of the excerpt in question with the types identified in the interface as global.

```
[("Class2",0,(Decl Class2 | Path: [D,R,D,R,D]),
("Class1",0,(Decl Class1 | Path: [D,R,D])),
("Class1",0,(Decl Class1 | Path: [D]))]
```

Two details to note in the implementation of the **globals** function is the addition of a further **DOWN** direction in the path to the "Global" data type and the use of the **buildChildren** function to perform the iterative process described by the remainder **AST**. The first is due to the fact that the path we want is not to the **Global** type, but to the data type that is actually global, since the **Global** type is just an indicator that what follows should be considered global, and is not in itself global, but rather the type that is attached to it, which is directly below it in the **AST**. The use of the **buildChildren** function in this function is a further demonstration of its versatility, since for any **AST** and build function, it can generate a higher-order Block tree.

Regarding the processor, some changes were necessary based on the Algol 68 Scope Processor. The first is that all functions relating to the previously defined attribute grammar will receive and return to the global **ENV** calculated by the higher-order **Block** tree calculated by the **globals** function. This change is designed to take into account that the **mustBeIn** and **mustNotBeIn** checking functions will now have to be aware of the local and global **ENV** in order to perform their checks.

```
mustBeIn n i e a = if (null (filter ((== n) . fst3) (e++a)))
                   then [(n, i, "␣<=␣[Undeclared␣use!]")]
                   else []

fst3 (x, _, _) = x
```

With regard to the "mustbeIn" function, it was only necessary to concatenate the two **ENVs** to check for undeclared uses in the general focus environment. However, the implementation of the "mustNotBeIn" function was not so trivial, since checking whether a name had yet been declared would result in the error of giving a name as a duplicate because it found itself in the global **ENV**, since it was calculated before the check.

```
mustNotBeInE :: Int -> BS.Name ->  BS.It -> BS.Env -> BS.Errors
mustNotBeInE i name item e =
    let names = (map (\(name', lev, _) -> (name',lev)) e)
        errorMsg = "␣<=␣[Duplicated␣declaration!]"
    in
    if ((name,i) `elem` names) then [(name, item, errorMsg)] else []

mustNotBeInA :: BS.Name -> BS.It -> BS.Env -> BS.Errors
mustNotBeInA name item a =
    let items = filter (\(name', _, item') -> name' == name && item /= item') a
        errorMsg = "␣<=␣[Different␣declaration!]"
    in
    if not (null items) then [(name, item, errorMsg)] else []

mustNotBeIn :: Int -> BS.Name -> BS.It -> BS.Env -> BS.Env -> BS.Errors
mustNotBeIn i name item e a =
    mustNotBeInE i name item e ++ mustNotBeInA name item a
```

To solve this problem, a new algorithm was designed to accommodate the new requirements of the problem. As such, the "mustNotBeIn" function was split into two which are concatenated at the end in order to find the resulting errors. So "mustNotBeIn" checks for local and global errors separately. The local check works very similarly to the previous one in the Algo68 processor. However, the global check

works a little differently. The name under analysis is searched for in the general environment, but in order to ensure that the processor doesn't give an error because it finds its own focus and thinks it's a duplicate declaration, it can't have the same path in the original tree. So if the name is found and does not have the same path in the original tree, ensuring that it is not itself, a duplicate declaration error is flagged.
With these changes made and running the processor on our sample language example, we can see that the processor was indeed able to find the errors in the code snippet.

```
Class Class1 <= [Different declaration!] {
        Class2 a
}

Class Class1 <= [Different declaration!] {
        Class2 b
}

Class Class2 {
        Class1 c
}
```

Although this is a simple example, it is proof of how scalable and plug-and-play we can make our name analysis. For any language, the user is provided with a framework that facilitates this whole process. All they have to do is define an instance of the Scopes class and choose a set of predefined rules or develop their own using the generic functions developed that make the construction of the higher-order **Block** tree universal. This provides a simple and transparent solution to a problem that is handled manually most of the times and re-defined for every evolution of a given language, delivering a framework to build Scope Rule processors abstracting and simplifying the subtleties of the topic. However, further work is needed in this processor given the fact that the use of inner classes is not contemplated is this work for example.

Chapter 6

Evaluation

To evaluate the expressiveness of the Scopes Interface approach several case studies were implemented in the Haskell language. Three languages were built, representing the three programming paradigms and their Scope Rules. For each language, some tests with different **AST** and types of errors were built in order to test the full capabilities of the interface and the predefined Scope processors.

6.1 Case Studies

In order to assess the expressiveness and effectiveness of the approach outlined, we have designed languages that mimic various programming paradigms: **Let** (functional paradigm), **Core** (imperative paradigm), **Toy-Java** (object-oriented paradigm). Each language was used to test the respective processors that emulate the Scope rules for each paradigm.

The **Core** language was used to test the Scope Declare-Before-Use Scope Processor. The **Let** language was used to test the Algol 68 Scope Processor, previously used in the work of Saraiva (2007). Finally, the **Toy-Java** language was used to test the Object-Oriented Scope Processor. Note that although each language was used to test each processor, the processors could be applied to any of them and produced the errors associated with their Scope rules.

While such case studies do not demonstrate that the approach scales to specification of full-fledged languages (and the feature interaction that comes with those), they do provide evidence of the expressiveness of the approach and the ease of its expansion.

6.2 Validation

Each language was tested with several **AST**, testing for different types of errors such as duplicate declarations and the use of undeclared variables. These tests were created in order to verify that the processors would be able to find all possible Scope errors regardless of the language and Scope rules.

```
let w = b + -16
    a = 8
    w = let z = a + b
        in z + b
    b = c + 3 - c
in c + a - w
```

```
let w = b + -16
    a = 8
    w <= [Duplicated declaration!]
      = let z = a + b
        in z + b
    b = c <= [Undeclared use!]
        + 3 - c <= [Undeclared use!]
in c <= [Undeclared use!] + a - w
```

During the construction and annotation phase, several tests were also created to check that the paths built in the higher-order block tree to the nodes in the original tree were actually correct. When the construction of the higher-order tree was executed, a function traversed the generated paths, testing whether they would actually lead to the corresponding node. The translation and respective construction of the higher-order block tree were also thoroughly tested to ensure that all relevant information for the construction of the higher-order **Block** tree was collected and used correctly and that no information was lost or altered in the process. Finally, tests were performed directly on the **AST** of the *Haskell* which produced the expected results using the following specification.

```
instance Scopes HsModule where
    isDecl t = case constructor t of
                  CHsPVar -> True
                  _ -> False
    isUse t = case constructor t of
                  CHsVar -> True
                  _ -> False
    isBlock t = case constructor t of
                  CHsLet            -> True
                  CHsGuardedRhss    -> True
                  CHsUnGuardedRhs   -> True
                  CHsAlt            -> True
                  _ -> False
    getUse t = case constructor t of
                  CHsVar -> case lexeme_HsVar t of
```

```
                    Qual _ (HsIdent s) -> s
                    UnQual (HsIdent s) -> s
    getDecl t = case constructor t of
                CHsPVar -> case lexeme_HsPVar t of
                    HsIdent s -> s
instance StrategicData HsModule where
    isTerminal t = isJust (getHole t :: Maybe SrcLoc)
```

By reusing the *Haskell* front end and leveraging existing parser libraries alongside predefined data types, including definitions, usages, and nested block specifications, we were able to successfully parse identifiers across the entire language. This approach facilitated efficient detection of scope errors and enabled straightforward annotation of the original code, which was handled in a largely automated and effortless manner.

Chapter 7

Related work

In this section we discuss how our approach to model Scope rules processors compares to other approaches, focusing on the support for name binding, executability of specifications as type checkers and scalability.

Most of the processors used for this purpose and incorporated into today's **IDE** are written manually and individually for each language, being subject to changes whenever that language evolves. This approach is in considerable contrast to the one presented in this work, which offers a framework for developers to abstract away from the syntax intricacies and scope rules of their language and model their processor in a generic, ad-hoc way.

On the other hand, the work of Antwerpen et al. (2018) provides a generic framework for modelling the binding structures of programming languages, bridging the gap between formalization and implementation, supporting the definition of type checkers and the automation of type safety proofs, but its implementation is done using a syntax definition in **SDF3**[1], a type checker in **NaBL2**[2] and a solver in *Java*. The syntax definition provides an *Eclipse* editor with syntax checking and *Statix* definition highlighting. The **NaBL2** definition provides type checking and reference resolution (jump to declaration) for *Statix* definitions in the editor. The *Statix* solver interprets the **AST** of a *Statix* specification applied to the **AST** of an object language program.

At the moment, the solver only accepts or rejects an object language program. It does not yet provide error messages to explain a failure, unlike our approach. This complexity contrasts with our approach since the complexity of implementation and extensibility is much greater.

[1] The SDF family of Syntax Definition Formalisms provides support for declarative definition of the syntax of programming languages and domain-specific languages.

[2] The NaBL2 'Name Binding Language' supports the specification of name binding and type checking rules of a language and uses a constraint-based approach, and uses scope graphs for name resolution.

Chapter 8

Conclusions

This document outlines a formal approach to automatically determine the full process of creating a representation (higher-order Block tree) used to analyze names and scope rules. This formalism is part of a broader set of functions that equip the Scopes Interface with capabilities to verify and annotate errors directly in the original abstract syntax tree **AST**. By leveraging the Block tree construction, paths to nodes in the original tree are accumulated, making it possible to abstract key information while retaining its context. The approach demonstrates that combining parsing techniques with the Block processor ensures accurate, transparent, and intuitive analysis without information loss. The methodology is scalable, allowing users to easily define custom attribute grammars for new languages. This flexibility enables the seamless integration of new processors into the existing framework.

- **TRQ1:** Is it possible to automatically infer this whole process and construction of the representation where we express the analysis of names?
  In this document, we present a formalism that allows to automatically infer the entire process of building the representation used to analyze names and Scope rules (higher-order **Block** tree). This formalism is part of a set of generic functions that provide the Scopes Interface with the necessary capabilities to carry out these verifications.

- **TRQ2:** Is it possible to define a mechanism that allows errors to be flagged in the original tree and not in the one where we define the naming rules?
  As we have carefully shown, using this interface, the user has access to a mechanism that allows errors to be annotated in the original tree, which is automatically derived by the interface. This mechanism was only possible due to the work done in the construction phase of the higher-order **Block** tree to accumulate the paths to the respective nodes in the original phase. This procedure is totally generic and can be used not only for this purpose but for other types of analysis where it is necessary to abstract important information from an **AST** while being aware of its location in the

original tree.

- **TRQ3:** Is it viable to merge the workings of the Block language processor with the parsing process of a language in order to parse names and Scope rules without losing information about the language in question?
  The development of this library has shown that merging the parsing techniques of a language and the operation of the **Block** processor is a viable practice for this type of check that does not incur any loss of information and ends up making this type of analysis a transparent and intuitive practice.

Bibliography

C. https://www.cprogramming.com/. Accessed: 2024-01-20.

Haskell. https://www.haskell.org/. Accessed: 2024-01-20.

Intellij idea – the leading java and kotlin ide. https://www.jetbrains.com/idea/. Accessed: 2024-01-20.

Neovim. https://neovim.io/. Accessed: 2024-01-20.

Visual studio code – code editing. redefined. https://code.visualstudio.com/. Accessed: 2024-01-20.

Zipperag: An implementationg of attribute grammars using functional zippers. https://hackage.haskell.org/package/ZipperAG-0.9. Accessed: 2024-01-20.

Hendrik Van Antwerpen, Casper Bach Poulsen, Arjen Rouvet, and Eelco Visser. *Scopes as Types*. OOPSLA, 2018.

John W. Backus. *The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference*. IFIP Congress, 1959.

James Finnie-Ansley, Paul Denny, and James Prather Brett A. Becker, Andrew Luxton-Reilly. *The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming*. ACM ISBN, 2022.

Robert Harper. *Programming in Standard ML*. Carnegie Mellon University, 2011.

Gérard Huet. *FUNCTIONAL PEARL*. Cambridge University Press, 1997.

Donald E. Knuth. *Semantics of context-free languages*. Springer, 1968.

Gabriel D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. *Declarative Name Binding and Scope Rules*. Springer, 2013.

José Nuno Macedo. *Zipping Strategies and Attribute Grammars*. FLOPS, 2022.

José Nuno Macedo. *Zipper-based embedding of strategic attribute grammars*. The Journal of Systems and Software, 2024.

Pedro Martins, João Paulo Fernandes, and João Saraiva. *Zipper-Based Attribute Grammars and Their Extensions*. Springer, 2013.

Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. *A Theory of Name Resolution*. Springer-Verlag Berlin Heidelberg, 2015.

João Saraiva. *The Fun of Programming with Attribute Grammars*. Detailed summary of lecture as per Art. 8.o Dec.-Lei 239, 2007.

H.H. Vogt. *Higher order attribute grammars.* SIGPLAN, 1989.

Part I
Appendices

Appendice A
The Block Language

```
data Direction = U | D | R | L deriving(Show,Eq,Data)
type Directions = [Direction]

data P  = Root Its
          deriving (Typeable, Data,Eq)

data Its = ConsIts It Its
         | NilIts
       deriving (Typeable, Data,Eq)

data It = Decl Name Directions
        | Use Name Directions
        | Block Its
        deriving (Typeable, Data,Eq)

type Name = String

type Env    = [(Name, Int, It)]
type Errors = [(Name, It, String)]

data Constructor = CConsIts
                 | CNilIts
                 | CDecl
                 | CUse
                 | CBlock
                 | CRoot
                deriving Show

constructor :: (Typeable a) => Zipper a -> Constructor
constructor a = case ( getHole a :: Maybe Its ) of
                 Just (ConsIts _ _) -> CConsIts
                 Just (NilIts) -> CNilIts
                 otherwise -> case ( getHole a :: Maybe It ) of
                               Just (Decl _ _) -> CDecl
                               Just (Use _ _) -> CUse
                               Just (Block _) -> CBlock
                               otherwise -> case ( getHole a :: Maybe P) of
                                             Just (Root _) -> CRoot
                                             otherwise -> error "ERROR"

lexeme z = case (getHole z :: Maybe It) of
            Just (Use x _) -> x
            Just (Decl x _) -> x
            Just (Block a) -> (show a)
            _              -> error "ERROR"
```

Appendice B
The Core Language

```haskell
data Exp = Add Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
         | Mul Exp Exp
         | Less Exp Exp
         | Greater Exp Exp
         | Equals Exp Exp
         | GTE Exp Exp
         | LTE Exp Exp
         | And Exp Exp
         | Or Exp Exp
         | Not Exp
         | Const Int
         | Var String
         | Inc Exp
         | Dec Exp
         | Return Exp
         | Bool Bool
         deriving (Show,Eq,Data)

data Item = Decl String Exp
          | Arg Exp
          | Increment Exp
          | Decrement Exp
          | NestedIf If
          | OpenIf If
          | NestedWhile While
          | OpenWhile While
          | NestedLet Let
          | OpenLet Let
          | NestedFuncao Funcao
          | OpenFuncao Funcao
          | NestedReturn Exp
          deriving (Show,Eq,Data)

data Items = ConsIts Item Items
           | NilIts
           deriving (Show, Data ,Eq)

data Let = Let Items Exp
        deriving (Show,Eq,Data)

data Funcao = Funcao Name Items
            | DefFuncao Name Items Items
            deriving (Show,Eq,Data)
```

```
data Name = Name String
          deriving (Show,Eq,Data)

data If = If Exp Items
        deriving (Show,Eq,Data)

data While = While Exp Items
        deriving (Show,Eq,Data)

constructor a = case (getHole a :: Maybe Exp) of
    Just (Add _ _)          -> CAdd
    Just (Sub _ _)          -> CSub
    Just (Div _ _)          -> CDiv
    Just (Mul _ _)          -> CMul
    Just (Less _ _)         -> CLess
    Just (Greater _ _)      -> CGreater
    Just (Equals _ _)       -> CEquals
    Just (GTE _ _)          -> CGTE
    Just (LTE _ _)          -> CLTE
    Just (And _ _)          -> CAnd
    Just (Or _ _)           -> COr
    Just (Not _)            -> CNot
    Just (Const _)          -> CConst
    Just (Var _)            -> CVar
    Just (Inc _)            -> CInc
    Just (Dec _)            -> CDec
    Just (Return _)         -> CReturn
    Just (Bool _)           -> CBool
    otherwise               -> case (getHole a :: Maybe Item) of
        Just (Decl _ _)           -> CDecl
        Just (Arg _)              -> CArg
        Just (Increment _)        -> CIncrement
        Just (Decrement _)        -> CDecrement
        Just (NestedIf _)         -> CNestedIf
        Just (OpenIf _)           -> COpenIf
        Just (NestedWhile _)      -> CNestedWhile
        Just (OpenWhile _)        -> COpenWhile
        Just (NestedLet _)        -> CNestedLet
        Just (OpenLet _)          -> COpenLet
        Just (NestedFuncao _)     -> CNestedFuncao
        Just (OpenFuncao _)       -> COpenFuncao
        Just (NestedReturn _)     -> CNestedReturn
        otherwise                 -> case (getHole a :: Maybe Let) of
            Just (Let _ _)          -> CLet
            otherwise               -> case (getHole a :: Maybe Funcao) of
                Just (Funcao _ _)         -> CFuncao
                Just (DefFuncao _ _ _)    -> CDefFuncao
                otherwise                 -> case (getHole a :: Maybe Name) of
                    Just (Name _)             -> CName
                    otherwise               -> case (getHole a :: Maybe If) of
                        Just (If _ _)           -> CIf
                        otherwise               -> case (getHole a :: Maybe While) of
                            Just (While _ _)      -> CWhile
                            otherwise             -> case (getHole a :: Maybe Items) of
                                Just (ConsIts _ _)  -> CConsIts
                                Just (NilIts)       -> CNilIts
                                otherwise           -> error "Error␣in␣constructor"
```

```
lexeme a = case (getHole a :: Maybe Exp) of
              Just (Var a)           -> a
              otherwise              -> case (getHole a :: Maybe Item) of
                 Just (Decl a _)        -> a
                 otherwise              -> case (getHole a :: Maybe Funcao) of
                    Just (Funcao (Name a) _)          -> a
                    Just (DefFuncao (Name a) _ _)     -> a
                    otherwise                      -> case (getHole a :: Maybe Name) of
                       Just (Name a)               -> a
                       otherwise                   -> error "Error␣in␣lexeme!"
```

Appendice C
The Toy-Java Language

```
data Exp = Add Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
         | Mul Exp Exp
         | Less Exp Exp
         | Greater Exp Exp
         | Equals Exp Exp
         | GTE Exp Exp
         | LTE Exp Exp
         | And Exp Exp
         | Or Exp Exp
         | Const Int
         | Var String
         | Return Exp
         | Bool Bool
         deriving (Show,Eq,Data)

data Item = Decl String Exp
          | Arg Exp
          | Global Item
          | Let Items Exp
          | Funcao String Items
          | DefFuncao String Items Items
          | If Exp Items
          | While Exp Items
          | DefClass String Items
          | Class String Item
          deriving (Show,Eq,Data)

data Root = Root Items deriving (Show,Eq,Data)

data Items = ConsIts Item Items
           | NilIts
           deriving (Show, Data ,Eq)

data Constructor = CAdd
                 | CSub
                 | CDiv
                 | CMul
                 | CLess
                 | CGreater
                 | CEquals
                 | CGTE
                 | CLTE
                 | CAnd
```

```
                | COr
                | CConst
                | CVar
                | CReturn
                | CBool
                | CDecl
                | CArg
                | CLet
                | CGlobal
                | CFuncao
                | CDefFuncao
                | CName
                | CIf
                | CWhile
                | CConsIts
                | CNilIts
                | CClass
                | CDefClass
                | CRoot
                deriving Show

constructor :: (Typeable a) => Zipper a -> Constructor
constructor a = case (getHole a :: Maybe Exp) of
    Just (Add _ _)          -> CAdd
    Just (Sub _ _)          -> CSub
    Just (Div _ _)          -> CDiv
    Just (Mul _ _)          -> CMul
    Just (Less _ _)         -> CLess
    Just (Greater _ _)      -> CGreater
    Just (Equals _ _)       -> CEquals
    Just (GTE _ _)          -> CGTE
    Just (LTE _ _)          -> CLTE
    Just (And _ _)          -> CAnd
    Just (Or _ _)           -> COr
    Just (Const _)          -> CConst
    Just (Var _)            -> CVar
    Just (Return _)         -> CReturn
    Just (Bool _)           -> CBool
    otherwise -> case (getHole a :: Maybe Item) of
        Just (Decl _ _)           -> CDecl
        Just (Arg _)              -> CArg
        Just (Let _ _)            -> CLet
        Just (Funcao _ _)         -> CFuncao
        Just (DefFuncao _ _ _)    -> CDefFuncao
        Just (If _ _)             -> CIf
        Just (While _ _)          -> CWhile
        Just (Class _ _)            -> CClass
        Just (DefClass _ _)       -> CDefClass
        Just (Global _)           -> CGlobal
        otherwise -> case (getHole a :: Maybe Items) of
                            Just (ConsIts _ _)     -> CConsIts
                            Just (NilIts)          -> CNilIts
                            otherwise -> case (getHole a :: Maybe Root) of
                                    Just (Root _)     -> CRoot
                                    otherwise -> error "Error␣in␣constructor"
```

```
lexeme a = case (getHole a :: Maybe Exp) of
            Just (Var a)          -> a
            otherwise             -> case (getHole a :: Maybe Item) of
                Just (Decl a _)          -> a
                Just (Funcao a _)          -> a
                Just (DefFuncao a _ _)     -> a
                Just (DefClass a _ )     -> a
                Just (Class a _)     -> a
                Just (Global (Decl a _))     -> a
                Just (Global (DefFuncao a _ _))     -> a
                otherwise                   -> error "Error␣in␣lexeme!"
```

Appendice D
The Block Processor (Algol 68 Rules)

```
---- Synthesized Attributes ----

dclo :: Env -> Zipper P -> Env
dclo a t =  case constructor t of
                 CNilIts   -> dcli a t
                 CConsIts  -> dclo a (t.$2)
                 CUse      -> dcli a t
                 CBlock    -> dcli a t
                 CRoot     -> dclo a (t.$1)
                 CDecl -> (lexeme t,lev t, (fromJust $ getHole t)) : (dcli a t)

errors :: Env -> Zipper P -> Errors
errors a t =  case BS.constructor t of
                 CRoot     -> errors a (t.$1)
                 CNilIts   -> []
                 CConsIts  -> (errors a (t.$1)) ++ (errors a (t.$2))
                 CBlock    -> errors a (t.$1)
                 CUse  -> mustBeIn (lexeme t) (fromJust $ getHole t) (env a t) a
                 CDecl -> mustNotBeIn (lev t)
                             (lexeme t) (fromJust $ getHole t) (dcli a t) a

---- Inheritted Attributes ----

dcli :: Env -> Zipper P -> Env
dcli a t =  case constructor t of
                    CRoot     -> a
                    CNilIts   -> case  (constructor $ parent t) of
                                            CConsIts  -> dclo a (t.$<1)
                                            CBlock    -> env a (parent t)
                                            CRoot     -> []
                    CConsIts  -> case  (constructor $ parent t) of
                                            CConsIts  -> dclo a (t.$<1)
                                            CBlock    -> env a (parent t)
                                            CRoot     -> []
                    CBlock    -> dcli a (parent t)
                    CUse      -> dcli a (parent t)
                    CDecl     -> dcli a (parent t)

lev :: AGTree Int
lev t =  case constructor t of
                    CRoot     ->  0
                    CBlock    ->  lev (parent t)
                    CUse      ->  lev (parent t)
                    CDecl     ->  lev (parent t)
                    _         ->  case  (constructor $ parent t) of
```

```
                                                  CBlock     -> (lev (parent t)) + 1
                                                  CConsIts   -> lev (parent t)
                                                  CRoot      -> 0

env :: Env -> Zipper P -> Env
env a t =  case constructor t of
                        CRoot       ->  dclo a t
                        CBlock      ->  env a (parent t)
                        CUse        ->  env a (parent t)
                        CDecl       ->  env a (parent t)
                        _           ->  case (constructor $ parent t) of
                                                  CBlock     -> dclo a t
                                                  CConsIts   -> env a (parent t)
                                                  CRoot      -> dclo a t

block :: Env -> P -> Errors
block a p = errors a (mkAG p)
```

Appendice E
The Block Processor (Declare-Before-Use Rules)

```
---- Synthesized Attributes ----

dclo :: Env -> Zipper P -> Env
dclo a t =  case constructor t of
                 CNilIts   -> dcli a t
                 CConsIts  -> dclo a (t.$2)
                 CUse      -> dcli a t
                 CBlock    -> dcli a t
                 CRoot     -> dclo a (t.$1)
                 CDecl -> (lexeme t,lev t, (fromJust $ getHole t)) : (dcli a t)

errors :: Env -> Zipper P -> Errors
errors a t =  case BS.constructor t of
                 CRoot     -> errors a (t.$1)
                 CNilIts   -> []
                 CConsIts  -> (errors a (t.$1)) ++ (errors a (t.$2))
                 CBlock    -> errors a (t.$1)
                 CUse  -> mustBeIn (lexeme t) (fromJust $ getHole t) (dcli a t) a
                 CDecl -> mustNotBeIn (lev t) (lexeme t)
                                (fromJust $ getHole t) (dcli a t) a

---- Inheritted Attributes ----

dcli :: Env -> Zipper P -> Env
dcli a t =  case constructor t of
                    CRoot     -> a
                    CNilIts   -> case  (constructor $ parent t) of
                                             CConsIts  -> dclo a (t.$<1)
                                             CBlock    -> dcli a (parent t)
                                             CRoot     -> []
                    CConsIts  -> case  (constructor $ parent t) of
                                             CConsIts  -> dclo a (t.$<1)
                                             CBlock    -> dcli a (parent t)
                                             CRoot     -> []
                    CBlock    -> dcli a (parent t)
                    CUse      -> dcli a (parent t)
                    CDecl     -> dcli a (parent t)

lev :: AGTree Int
lev t =  case constructor t of
                    CRoot     ->  0
                    CBlock    ->  lev (parent t)
                    CUse      ->  lev (parent t)
                    CDecl     ->  lev (parent t)
                    _         ->  case  (constructor $ parent t) of
```

```
                                            CBlock     -> (lev (parent t)) + 1
                                            CConsIts   -> lev (parent t)
                                            CRoot      -> 0

env :: Env -> Zipper P -> Env
env a t =  case constructor t of
                    CRoot      ->  dclo a t
                    CBlock     ->  env a (parent t)
                    CUse       ->  env a (parent t)
                    CDecl      ->  env a (parent t)
                    _          ->  case (constructor $ parent t) of
                                            CBlock     -> dclo a t
                                            CConsIts   -> env a (parent t)
                                            CRoot      -> dclo a t

block :: Env -> P -> Errors
block a p = errors a (mkAG p)
```

Appendice F
The Block Processor (Object-Oriented Rules)

```
---- Synthesized Attributes ----
dclo :: Env -> Zipper P -> Env
dclo a t =  case BS.constructor t of
                CNilIts   -> dcli a t
                CConsIts  -> dclo a (t.$2)
                CDecl     -> (lexeme t,lev t, (fromJust $ getHole t)) : (dcli a t)
                CUse      -> dcli a t
                CBlock    -> dcli a t
                CRoot     -> dclo a (t.$1)

errors :: Env -> Zipper P -> Errors
errors a t = case constructor t of
               CRoot     -> errors a (t.$1)
               CNilIts   -> []
               CConsIts  -> (errors a (t.$1)) ++ (errors a (t.$2))
               CBlock    -> errors a (t.$1)
               CUse      -> mustBeIn (lexeme t) (fromJust $ getHole t) (env a t) a
               CDecl     -> mustNotBeIn (lev t)
                                (lexeme t) (fromJust $ getHole t) (dcli a t) a

---- Inheritted Attributes ----

dcli :: Env -> Zipper P -> Env
dcli a t =  case constructor t of
                CRoot     -> a
                CNilIts   -> case  (constructor $ parent t) of
                                CConsIts  -> dclo a (t.$<1)
                                CBlock    -> env a (parent t)
                                CRoot     -> []
                CConsIts  -> case  (constructor $ parent t) of
                                    CConsIts  -> dclo a (t.$<1)
                                    CBlock    -> env a (parent t)
                                    CRoot     -> []
                CBlock    -> dcli a (parent t)
                CUse      -> dcli a (parent t)
                CDecl     -> dcli a (parent t)

lev :: AGTree Int
lev t =  case constructor t of
            CRoot     ->  0
            CBlock    ->  lev (parent t)
            CUse      ->  lev (parent t)
            CDecl     ->  lev (parent t)
            _         ->  case  (constructor $ parent t) of
                                CBlock    -> (lev (parent t)) + 1
```

```
                                  CConsIts   -> lev (parent t)
                                  CRoot      -> 0

env :: Env -> Zipper P -> Env
env a t =  case constructor t of
                    CRoot       ->  dclo a t
                    CBlock      ->  env a (parent t)
                    CUse        ->  env a (parent t)
                    CDecl       ->  env a (parent t)
                    _           ->  case (constructor $ parent t) of
                                                 CBlock     -> dclo a t
                                                 CConsIts   -> env a (parent t)
                                                 CRoot      -> dclo a t

block :: Env -> P -> Errors
block a p = errors a (mkAG p)
```