The Fun of Programming with Attribute Grammars

Attribute Grammars are a well-known and convenient formalism for specifying the semantic analysis phase of a compiler and for modeling complex multiple traversal algorithms.
In an AG setting, programmers do not have to con- cern themselves with scheduling complex traversal functions, nor do they have to define gluing data structures: such complex (functional) programs are generated by AG-based systems.
In an AG setting, a new attribute has to be defined to compute/synthesize the transformed tree. Moreover, attribute equations have to be associated to most productions of the AG. These equations define the transformation needed in the desired (type of) nodes, but also the construction of (similar) new nodes in most of the productions of the AG.
It relies on strategies (recursion schemes) to apply term rewrite rules in defining transformations. A strategy is a generic transformation function that can traverse into heterogeneous data structures while mixing uniform and type-specific behavior.

To provide a proper embedding of attribute grammars we extend our embedding with function memoization. By memoizing the calls to the zipper functions, we do avoid attribute re-calculation, and consequently we improve the performance of our zippers: the non-memoized version behaves as a quadratic function, while the memoized behaves as a linear one. -> **Util para provar a terminação do meu algoritmo**

Types as Grammars (Farei algo parecido mas pra scopes)

Grammars specify formal languages, where a formal language consists of a (possibly non-finite) set of sentences.

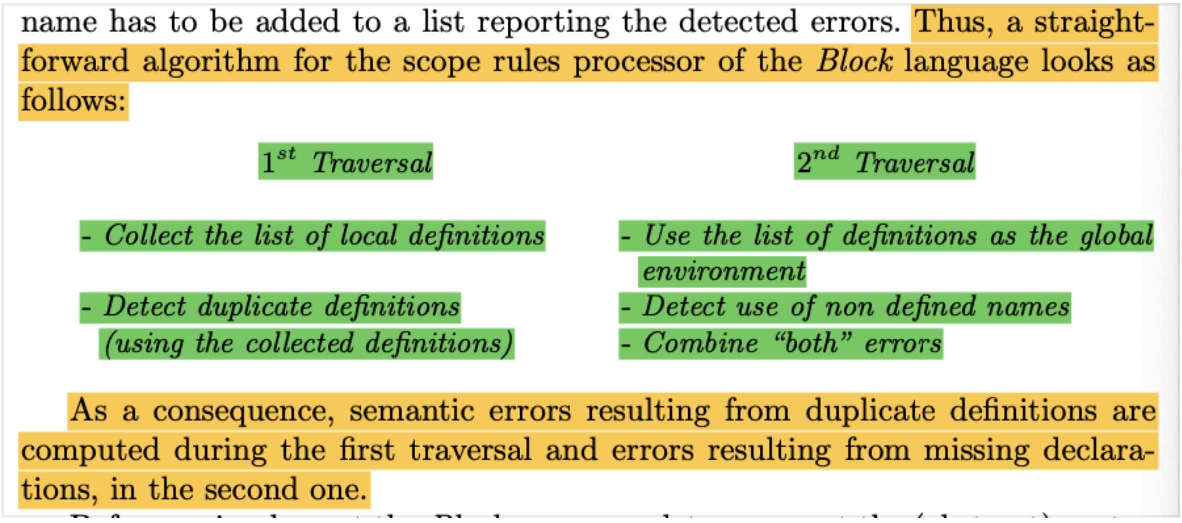
Functional Zippers

Zippers were originally conceived by Huet to represent a tree together with a subtree that is the focus of attention. During a computation, the focus may move left, up, down or right within the tree. Generic manipulation of a zipper is provided through a set of predefined functions that allow access to all of the nodes of the tree for inspection or modification.
Using the idea of having pairs of information composed by paths and trees, we may represent any positions in a tree and, better yet, this setting allows an easy navigation as we only have to remove parts of the path if we want to go to the top of the tree, or add information if we want to go further down.

Generic Zippers (Vou usar pra depois utilizar estratégias para as

minha verificações)
Generic zippers are available as a Haskell library, which works for both homogeneous and heterogeneous data types.

Zipper-based Embedding of Attribute Grammars


name has to be added to a list reporting the detected errors. Thus, a straightforward algorithm for the scope rules processor of the *Block* language looks as follows:

1st Traversal

- *Collect the list of local definitions*
- *Detect duplicate definitions (using the collected definitions)*

2nd Traversal

- *Use the list of definitions as the global environment*
- *Detect use of non defined names*
- *Combine "both" errors*

As a consequence, semantic errors resulting from duplicate definitions are computed during the first traversal and errors resulting from missing declarations, in the second one.

<Farei algo muito parecido com isto!>

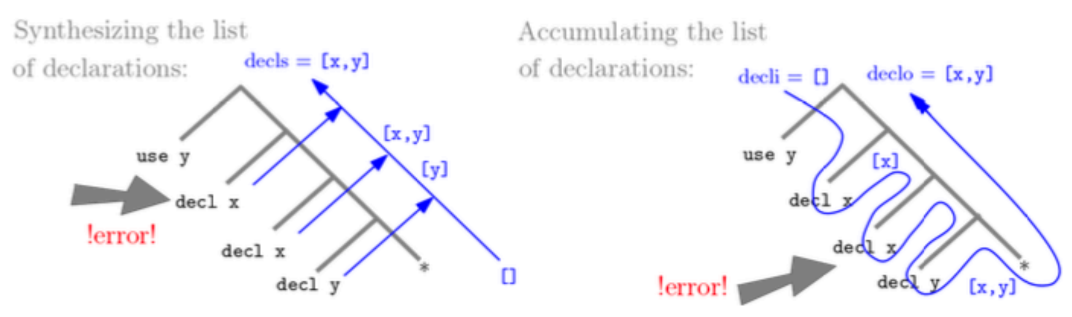


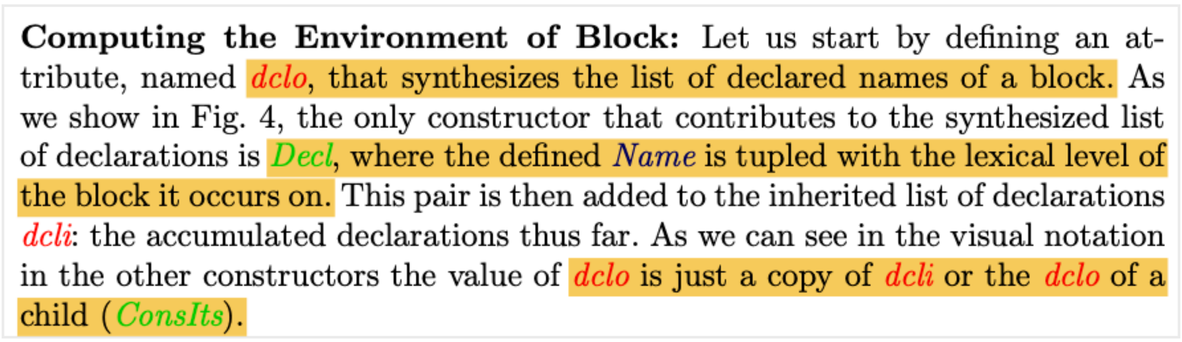
Fig. 2: Synthesizing versus accumulating the environment.

The dcli of the inner block is the dclo of its outer block. In other words, we say that inner blocks inherit the environment of their outer ones.
We may also notice that only in the second traversal of an outer block the inherited list of declarations needed by nested blocks is known. That is to say that only in the second traversal to an outer block, the inner blocks are traversed for the first time! As a result, traversal functions environment and errors are intermingled and it is not straightforward how to schedule them. -> **Super util para mim**

Indeed, the implementation of the straightforward algorithm of the Block processor in a (non-lazy) functional setting is a complex task: complex traversal functions need to be scheduled, additional gluing data types may have to be defined, and intermediate values need to be explicitly passed between traversals. A circular, lazy solution can also be defined which overcomes those issues, but there is a price to pay: first, it relies on the counter-intuitive circular definitions that are also difficult to define. Secondly, all aspects of the Block language are tupled into a single function definition, and, as consequence, such a lack of modularity makes it hard to write, to understand, and to (incrementally) extend the language with new features.

<Terei que fazer algo parecido com a Block language e o Block processor>

The Zipper-based Block Program


Computing the Environment of Block: Let us start by defining an attribute, named *dclo*, that synthesizes the list of declared names of a block. As we show in Fig. 4, the only constructor that contributes to the synthesized list of declarations is *Decl*, where the defined *Name* is tupled with the lexical level of the block it occurs on. This pair is then added to the inherited list of declarations *dcli*: the accumulated declarations thus far. As we can see in the visual notation in the other constructors the value of *dclo* is just a copy of *dcli* or the *dclo* of a child (*ConsIts*).

<Usarei isto mas "Decl" terá informação acerca do seu scope>
The attribute (function) dclo depends on the value (call) of (to) dcli. Thus, it defines an inherited attribute, which is typically inherited from the parent of its (sub)tree.

Distributing the Environment of Block
Type Env, to distribute it to all the items of a block. In the Root and Block constructors, this environment corresponds to the list of accumulated declarations.

Computing the Errors of Block
We are now able to synthesize the desired list of errors that follow the sequential structure of the input. There are two constructors that contribute to the list of errors: Decl and Use. In a declaration, the Name must not be in the accumulated list of declarations at the same lexical level (recall that it may be defined in an outer level). In the use of a Name, it must be in the (full) environment of its block. The AG fragment that synthesizes the errors is displayed in Fig. 8.

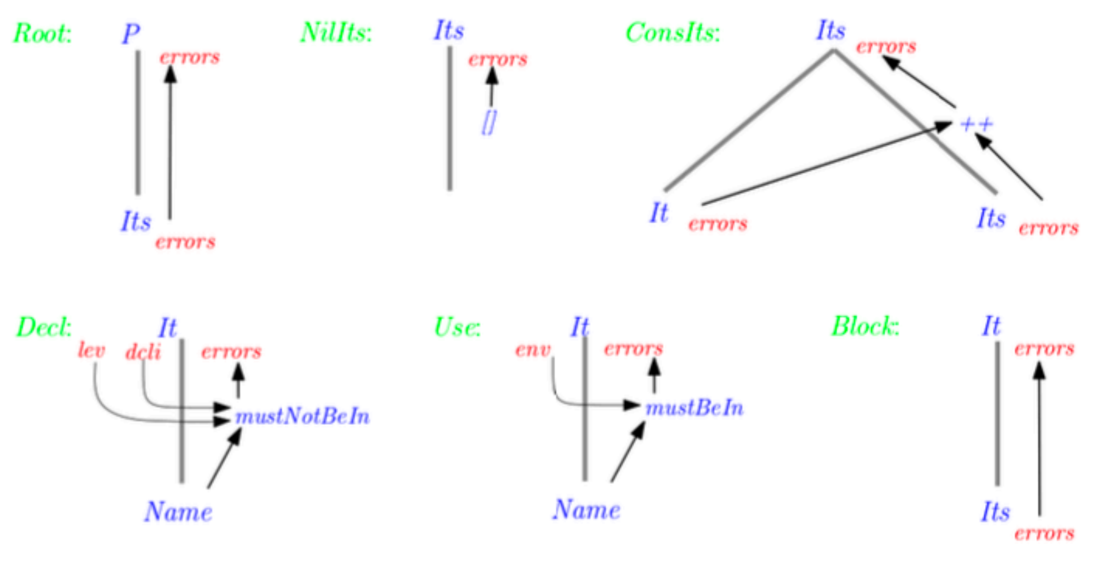
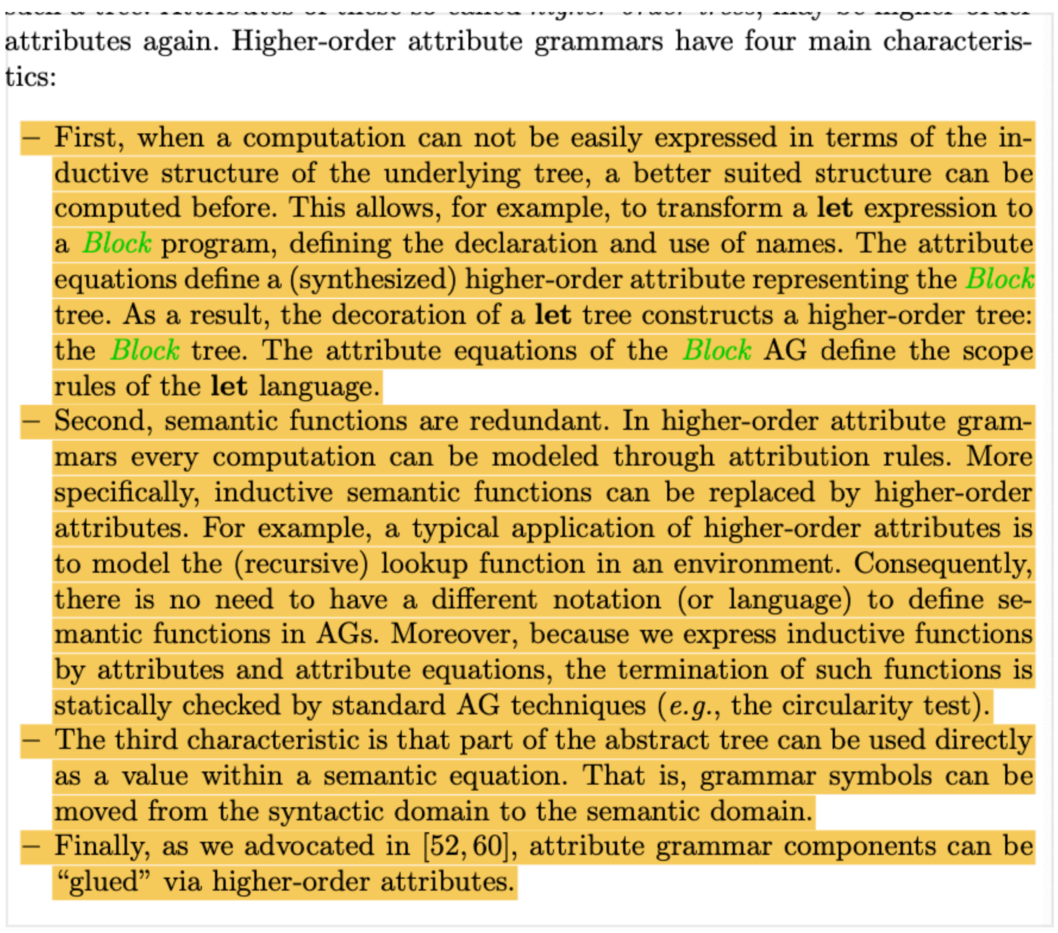


Fig. 8: Attribute equations synthesizing the list of errors.

Higher-Order Attribute Grammars



attributes again. Higher-order attribute grammars have four main characteristics:

- First, when a computation can not be easily expressed in terms of the inductive structure of the underlying tree, a better suited structure can be computed before. This allows, for example, to transform a **let** expression to a *Block* program, defining the declaration and use of names. The attribute equations define a (synthesized) higher-order attribute representing the *Block* tree. As a result, the decoration of a **let** tree constructs a higher-order tree: the *Block* tree. The attribute equations of the *Block* AG define the scope rules of the **let** language.
- Second, semantic functions are redundant. In higher-order attribute grammars every computation can be modeled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. For example, a typical application of higher-order attributes is to model the (recursive) lookup function in an environment. Consequently, there is no need to have a different notation (or language) to define semantic functions in AGs. Moreover, because we express inductive functions by attributes and attribute equations, the termination of such functions is statically checked by standard AG techniques (*e.g.*, the circularity test).
- The third characteristic is that part of the abstract tree can be used directly as a value within a semantic equation. That is, grammar symbols can be moved from the syntactic domain to the semantic domain.
- Finally, as we advocated in [52, 60], attribute grammar components can be "glued" via higher-order attributes.

Resumo:

Artigo fala sobre gramáticas de atributos e como as usar via Zippers e estratégias. Para a minha implementação terei que fazer algo muito parecido:

- Scopes as types (parecido com o que fizeram aqui com grammars as types)
- Passar para zipper
- Fazer verificações de forma muito similar a Block language e ao Block processor
- Para essas verificações terei primeiro que decidir a linguagem e identificar todas as regras de scope
- As verificações funcionarão com recurso às estratégias que melhor se adequarem
- Terei que usar a versão memoized para garantir o término da verificação.