Refactoring Dynamic Languages

Rafael Reia
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
rafael.reia@tecnico.ulisboa.pt

António Menezes Leitão
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT

Typically, beginner programmers do not master the style rules of the programming language they are using and, frequently, do not have yet the logical agility to avoid writing redundant code. As a result, although their programs might be correct, they can also be improved and it is important for the programmer to learn about the improvements that, without changing the meaning of the program, simplify it or transform it to follow the style rules of the language. These kinds of transformations are the realm of refactoring tools. However, these tools are typically associated with sophisticated integrated development environments (IDEs) that are excessively complex for beginners.

In this paper, we present a refactoring tool designed for beginner programmers, which we made available in DrRacket, a simple and pedagogical IDE. Our tool provides several refactoring operations for the typical mistakes made by beginners and is intended to be used as part of their learning process.

Keywords

Refactoring Tool, Pedagogy, Racket

1. INTRODUCTION

In order to become a proficient programmer, one needs not only to master the syntax and semantics of a programming language, but also the style rules adopted in that language and, more important, the logical rules that allow him to write simple and understandable programs. Given that beginner programmers have insufficient knowledge about all these rules, it should not be surprising to verify that their code reveals what more knowledgeable programmers call "poor style," "bad smells," etc. As time goes by, it is usually the case that the beginner programmer learns those rules and starts producing correct code written in an adequate style. However, the learning process might take a considerable amount of time and, as a result, large amounts of poorly-written code might be produced before the end of the process. It is then important to speed up this learning process by showing, from the early learning phases, how a poorly-written fragment of code can be improved.

After learning how to write code in a good style, programmers become critics of their own former code and, whenever they have to work with it again, they are tempted to take advantage of the opportunity to restructure it so that it conforms to the style rules and becomes easier to understand. However, in most cases, these modifications are done without complete knowledge of the requirements and constraints that were considered when the code was originally written and, as result, there is a serious risk that the modifications might introduce bugs. It is thus important to help the programmer in this task so that he can be confident that the code improvements he anticipate are effectively applicable and will not change the meaning of the program. This has been the main goal of *code refactoring*.

Code refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [1]. Nowadays, any sophisticated IDE includes an assortment of refactoring tools, e.g., for moving methods along a class hierarchy, to extract interfaces from classes, and to transform anonymous classes into nested classes. It is important to note, however, that these IDEs were designed for advanced programmers, and that the provided refactorings require a level of code sophistication that is not present in the programs written by beginners. This makes the refactoring tools inaccessible to beginners.

In this paper, we present a tool that was designed to address the previous problems. In particular, our tool (1) is usable from a pedagogical IDE designed for beginners [2, 3], (2) is capable of analyzing the programmer's code and inform him of the presence of the typical mistakes made by beginners, and finally (3) can apply refactoring rules that restructure the program without changing its semantics.

To evaluate our proposal, we implemented a refactoring tool in DrRacket, a pedagogical IDE [4, 5] used in schools around the world to teach basic programming concepts and techniques. Currently, DrRacket has only one simple refactoring operation which allows renaming a variable. Our work significantly extends the set of refactoring operations available in DrRacket and promotes their use as part of the learning process.

2. RELATED WORK

There are many refactoring tools available. The large majority of these tools were designed to deal with large

DOI: 10.1145/1235

statically-typed programming languages such a Java or C++ and are integrated in the complex IDEs typically used for the development of complex software projects, such as Eclipse or Visual Studio.

On the other hand, it is a common practice to start teaching beginner programmers using dynamically-typed programming languages, such as Scheme, Python, or Ruby, using simple IDEs. As a result, our focus was on the dynamic programming languages which are used in introductory courses and, particularly, those that promote a functional programming paradigm.

In the next sections, we present an overview of the refactoring tools that were developed for the languages used in introductory programming courses.

2.1 Scheme

In its now classical work [6], Griswold presented a refactoring tool for Scheme that uses two different kinds of information, namely, an Abstract Syntax Tree (AST) and a Program Dependence Graph (PDG).

The AST represents the abstract syntactic structure of the program, while the PDG explicitly represents the key relationship of dependence between operations in the program. The graph vertices's represent program operations and the edges represent the flow of data and control between operations. However, the PDG only has dependency information of the program and relying only in this information to represent the program could create problems. For example, two semantically unrelated statements can be placed arbitrarily with respect to each other. Using the AST as the main representation of the program ensures that statements are not arbitrarily reordered, allowing the PDG to be used to prove that transformations preserve the meaning and as a quick way to retrieve needed dependence information. Additionally, contours are used with the PDG to provide scope information, which is non existent in the PDG, and to help reason about transformations in the PDG. With these structures it is possible to have a single formalism to reason effectively about flow dependencies and scope structure.

2.2 Python

Rope[7, p. 109] is a Python refactoring tool written in Python, which works like a Python library. In order to make it easier to create refactoring operations, Rope assumes that a Python program only has assignments and function calls. Thus, by limiting the complexity of the language it reduces the complexity of the refactoring tool.

Rope uses a Static Object Analysis, which analyses the modules or scopes to get information about functions. Because its approach is time consuming, Rope only analyses the scopes when they change and it only analyses the modules when asked by the user.

Rope also uses a Dynamic Object Analysis that requires running the program in order to work. The Dynamic Object Analysis gathers type information and parameters passed to and returned from functions. It stores the information collected by the analysis in a database. If Rope needs the information and there is nothing on the database the Static object inference starts trying to infer it. This approach makes the program run much slower, thus it is only active when the user allows it. Rope uses an AST in order to store the syntax information about the programs.

Bicycle Repair Man[1] is another refactoring tool for Python and is written in Python itself. This refactoring tool can be added to IDEs and editors, such as Emacs, Vi, Eclipse, and Sublime Text. It attempts to create the refactoring browser functionality for Python and has the following refactoring operations: extract method, extract variable, inline variable, move to module, and rename.

The tool uses an AST to represent the program and a database to store information about several program entities and their dependencies.

Pycharm Educational Edition,[2] or Pycharm Edu, is an IDE for Python created by JetBrains, the creator of IntelliJ. The IDE was specially designed for educational purposes, for programmers with little or no previous coding experience. Pycharm Edu is a simpler version of Pycharm community which is the free python IDE created by JetBrains. It is very similar to their complete IDEs and it has interesting features such as code completion and integration with version control tools. However, it has a simpler interface than Pycharm Community and other IDEs such as Eclipse or Visual Studio.

Pycharm Edu integrates a python tutorial and supports teachers that want to create tasks/tutorials for the students. However, the refactoring tool did not received the same care as the IDE itself. The refactoring operations are exactly the same as the Pycharm Community IDE which were made for more advanced users. Therefore, it does not provide specific refactoring operations to beginners. The embedded refactoring tool uses the AST and the dependencies between the definition and the use of variables, known as def-use relations.

2.3 Javascript

There are few refactoring tools for JavaScript but there is a framework [8] for refactoring JavaScript programs. In order to guarantee the correctness of the refactoring operation, the framework uses preconditions, expressed as query analyses provided by pointer analysis. Queries to the pointer analysis produces over-approximations of sets in a safe way to have correct refactoring operations. For example, while doing a rename operation, it over-approximates the set of expressions that must be modified when a property is renamed in a safe manner.

To prove the concept, three refactoring operations were implemented, namely rename, encapsulate property, and extract module. By using over-approximations it is possible to be sure when a refactoring operation is valid. However, this approach has the disadvantage of not applying every possible refactoring operation, because the refactoring operations for which the framework cannot guarantee behavior preservation are prevented. The wrongly prevented operations accounts for 6.2% of all rejections.

3. ARCHITECTURE

In this section we present the architecture of our refactoring tool. Although the tool can work with different programming languages, its main focus is, at this moment, the Racket programming language and, more specifically, the DrRacket IDE.

Racket is a language designed to support meta-programming

[1]https://pypi.python.org/pypi/bicyclerepair/0.7.1
[2]https://www.jetbrains.com/pycharm-edu/

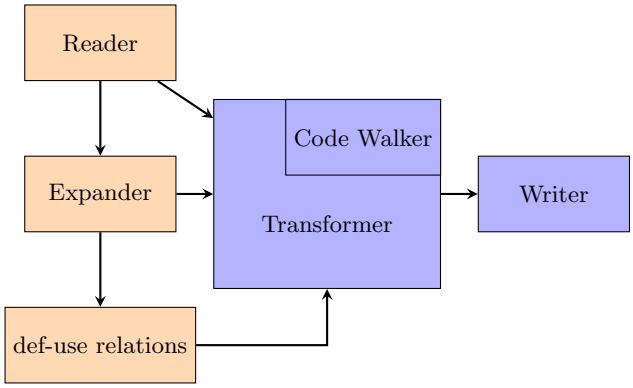

Figure 1: Main modules and information flow between modules. Unlabeled arrows represent information flow between modules.

and, in fact, most of the syntax forms of the language are macro-expanded into combinations of simpler forms. This has the important consequence that programs can be analyzed either in their original form or in their expanded form.

In order to create correct refactoring operations, the refactoring tool uses two sources of information, the def-use relations and the AST of the program. The def-use relations represent the links between definition of an identifier and its usage. In the DrRacket IDE these relations are visually represented as arrows that point from a definition to its use. The opposite relation, the use-def relation is also visually represented as an arrow from the use of an identifier to its definition. The AST is the abstract syntax tree of the program which, in the case of the Racket language, is represented by a list of syntax-objecs. Given that Racket

Figure 1 summarizes the workflow of the refactoring tool, where the Reader produces the non expanded AST of the program while the Expander expands the AST produced by the Reader. In order to produce the def-use relations it is necessary to use the expanded AST produced by the Expander because it has the correct dependency information. The Transformer uses the Code Walker to parse the ASTs and the information of the def-use relations to correctly perform the refactoring operations. Then it goes to the Writing module to produce the output in DrRacket's definitions pane.

3.1 Syntax Expressions

The syntax-object list represents the AST, which provides information about the structure of the program. The syntax-object list is already being produced and used by the Racket language and, in DrRacket, in order to provide error information to the user. DrRacket already provides functions which computes the program's syntax-object list and uses some of those functions in the Background Check Syntax and in the Check Syntax button callback.

3.1.1 Syntax Expression tree forms

DrRacket provides functions to compute the syntax-object list in two different formats. One format is the expanded program, which computes the program with all the macros expanded. The other format is the non-expanded program and computes the program with the macros unexpanded.

The expanded program has the macros expanded and the identifier information correctly computed. However, it is harder to extract the relevant information when compared with the non expanded program.

For example, the following program is represented in the expanded form, and in the non expanded form.

Listing 1: Original Code

```
(and alpha beta)
```

Listing 2: Expanded program

```
#<syntax:2:0
 (#%app call-with-values
 (lambda ()
    (if alpha beta (quote #f)))
  print-values)>
```

Listing 3: Non-expanded program

```
#<syntax:2:0 (and alpha beta)>
```

Note that the expanded program transforms the `and`, `or`, `when`, and `unless` forms into `if`s which makes refactoring operations harder to implement.

Racket adds internal representation information to the expanded program which for most refactoring operations is not necessary. In addition, the expanded program has a format that is likely to change in the future. Racket is an evolving language and the expanded form is a low level and internal form of representation of the program. However, the expanded program has important information regarding the binding information that is not available in the non-expanded form, and this information might be useful, e.g., to detect if two identifiers refer to the same binding. Additionally, we do not consider macro definitions as part of the code that needs to be refactored, since the refactoring tool is targeted at unexperienced programmers and these programmers typically do not define macros.

Taking the previous discussion into consideration, it becomes clear that it is desirable to use the non expanded form for the refactoring operations whenever possible and use the expanded form only when needed.

3.2 Def-use relations

Def-use relations hold important information needed in order to produce correct refactoring operations. They can be used to check whether there will be a duplicated name or to compute the arguments of a function that is going to be extracted.

The def-use-relations is computed by the compiler that runs in the background. However, it is only computed when a program is syntactically correct.

3.3 Code-walker

The code-walker is used to parse the syntax tree represented by a syntax element that is a list of syntax-object in Racket. A syntax-object can contain either a symbol, a syntax-pair, a datum (number, boolean or string), or an empty list. While a syntax-pair is a pair containing a syntax object as it first element and either a syntax pair, a syntax element or an empty list as the second argument. Each syntax-object has information about the line where they are defined and this information is used to search for the correct elements.

Most of the time, the code-walker is used to search for a specific syntax element and the location information contained in the syntax-object is used to skip the syntax blocks that are before the syntax element wanted in the first place.

The Code-walker is a core part of the refactoring tool, ensuring that the selected syntax is correctly fed to the refactoring operations.

3.4 Pretty-printer

Producing correct output is an important part of the refactoring tool. It is necessary to be careful to produce indented code and we decided to use a pretty-printer that is already available in the Racket language. However, it should be noted that this pretty-printer does not follow some of the Racket style convention, such as `cond` clauses surrounded by square brackets. This is not considered a problem because Racket supports both representations. One possible solution is to use a different pretty-printer in order to keep the language conventions.

3.5 Comments preservation

Preserving the comment information after a refactoring transformation is an important task of the refactoring tool. If the comment in determined place of the program changes its location, affecting a different part of the program, it could confuse the programmer. However, comment preservation is not implemented yet, making it a limitation of this prototype.

One possible solution is to modify the syntax reader and add a comment node to the AST. While the new node will not be used during refactoring transformations it is used during the output part of the refactoring operation, preserving the comment with the correct syntax expression.

3.6 Syntax-Parse

The Syntax-Parse[9] function provided by Racket is very useful for the refactoring operations. It provides a wide range of options to help matching the correct syntax, using backtracking to allow have several rules to be matched in the same syntax parser, which helps to create more sophisticated rules.

4. REFACTORING OPERATIONS

In this section we explain some of the more relevant refactoring operations and some limitations of the refactoring tool.

4.1 Semantic problems

There are some well-known semantic problems that might occur after doing a refactoring operation. One of them occurs in the refactoring operation that removes redundant `and`s in numeric comparisons. Although rarely known by beginner programmers, in Racket, numeric comparisons support more than two arguments, as in `(< 0 ?x 9)`, meaning the same as `(and (< 0 ?x) (< ?x 9)`, where, we use the notation `?x` to represent an expression. Thus, it is natural to think about a refactoring operation that eliminates the `and`. However, when the `?x` expression somehow produces side-effects, the refactoring operation will change the meaning of the program.

Despite this problem, we support this refactoring operation because, in the vast majority of the cases, there are no side-effects being done in the middle of numerical comparisons and, therefore, the refactoring operation does not change the meaning of the program.

Another example of a semantic problem occurs when refactoring the following `if` expression.

Listing 4: Code sample

```
(if ?x
    (begin ?y ...)
    #f)
```

There are two different refactoring transformations possible:

Listing 5: Refactoring option 1

```
(when ?x
  ?y ...)
```

Listing 6: Refactoring option 2

```
(and ?x (begin ?y ...))
```

Note that the first refactoring option changes the meaning of the program, because if the test expression, in this case `?x`, is false, the result of the `when` expression is `#<void>`. However, the programmer may still want to choose the first refactoring option if the return value when the `?x` is false is not important.

4.2 Extract Function

Extract function is an important refactoring operation that every refactoring tool should have. In order to extract a function it is necessary to compute the arguments needed to the correct use of the function. While giving the name to a function seams quite straightforward, it is necessary to check for name duplication in order to produce a correct refactoring as having two identifiers with the same name in the same scope produces an incorrect program. After the previous checks, it is straightforward to compute the function body and replacing the original expression by the function call.

However, the refactoring raises the problem where should the function be extracted to. A function can not be defined inside an expression, but it can be defined at the top-level or at any other level that is accessible from the current level.

As an example, consider the following program:

Listing 7: Extract function levels

```
;;top-level
(define (level-0)
  (define (level-1)
    (define (level-2)
      (+ 1 2))
    (level-2))
  (level-1))
```

When extracting the `(+ 1 2)` to a function where should it be defined? Top-level, Level-0, level-1, or in the current level, the level-2? The fact is that is extremely difficult to know the answer to this question because it depends on what the user is doing and the user intent. Accordingly, we decided that the best solution is to let the user decide where the user wants the function defined.

4.2.1 Computing the arguments

In order to compute the function call arguments we have to know in which scope the variables are being defined, in other words, if the variables are defined inside or outside the extracted function. The variables defined outside the function to be extracted are candidates to be the arguments of that function. However, imported variables, whether from the language base or from other libraries do not have to be passed as arguments. To solve this problem, we considered two possible solutions:

- Def-use relations + Text information
- Def-use relations + AST

The first approach is simpler to implement and more direct than the second one. However, it is less tolerant to future changes and to errors. The second one combines the def-use relations information with the syntax information to check whether it is imported from the language or from other library.

We choose the second approach in order to provide a more stable solution to correctly compute the arguments of the new function.

4.3 Let to Define Function

A `let` expression is very similar to a function, which may led the user to mistakenly use one instead of the other. Therefore we decided to provide a refactoring operation that would make such transition simpler.

There are several let forms, but since we want to explore the similarity between the `let` and the function we are going to focus in the ones that are more similar to a function, namely the `let` and the `named let`.

There are some differences between them, the `named let` can be directly mapped to a named function, using the `define` keyword, whereas the `let` can only be directly mapped to an anonymous function, `lambda`. We decided to focus first in the transformation of a `named let` to a function.

However, this refactoring operation which transforms a `named let` into a `define` function, could have syntax problems since a `let` form can be used in expressions, but the `define` can not. In the vast majority of cases this refactoring is correct, however, when a `named let` is used in an expression it transforms the program in an incorrect one. e.g.

Listing 8: Let in an expression

```
(and (let xpto ((a 1)) (< a 2)) (< b c))
```

Modifying this `named let` into a `define` would raise a syntax error since a `define` could not be used in an expression context. Encapsulate the `define` with the `local` keyword, which is an expression like the `named let`, can solve the problem. However, the `local` keyword is not used very often and might confuse the users. Therefore we decided keep the refactoring operation without the local keyword that works for most of the cases.

4.4 Wide-Scope Replacement

The Wide-Scope replacement extends the extract function functionality by replacing the duplicated of the extracted code with the function call of the extracted function.

The Wide-Scope replacement refactoring operation searches for the code that is duplicated of the extracted function and then replaces it for the call of the extracted function and it is divided in two steps:

- Detect duplicated code
- Replace the duplicated code

Replacing the duplicated code is the easy part, however the tool might has to compute the arguments for the duplicated code itself.

Correctly detecting duplicated code is a key part for the correctness of this refactoring. Even the simplest form of duplicated code detection, where it only detects duplicated code when the code is exactly equal, may have some problems regarding the binding information. For example, if the duplicated code is inside a `let` that changes some bindings that must be taken into consideration. In order to solve the binding problem we can use functions already provided in Racket. However, that does not work if we use the program in the not expanded form to do the binding comparisons because there is not enough information for those bindings to work. Therefore, in order to compute the correct bindings, it is necessary to use the expanded form of the program.

The naive solution is to use the expanded program to detect the duplicated code and then use this information to do the replacing of the duplicated code. However, when expanding the program Racket adds necessary internal information to run the program itself that are not visible for the user. While this does not change the detecting of the duplicated code, it adds unnecessary information that would have to be removed. In order to solve this in a simple way we can use the expanded code to detect the correctly duplicated code and use the non expand program to compute which code will be replaced.

However, this duplicated code detection is a quadratic algorithm which might have some performance problems for bigger programs.

5. FEATURES

This section describes some of the features created to improve the usability by providing sufficient feedback to the user, and way to inform the user of the presence of the typical mistakes made by beginners.

5.1 User FeedBack

It is important to give proper feedback to the user while the user is attempting or preforming a refactoring operation. Previewing the outcome of a refactoring operation is an efficient form to help the users understand the result of a refactoring before even applying the refactoring. It works by applying the refactoring operation in a copy version of the AST and displaying those changes to the user.

5.2 Automatic Suggestions

Beginner programmers usually do not know which refactoring operations exist or which can be applied. By having a automatic suggestion of the possible refactoring operations available the beginner programmer can have an idea what refactoring operations can be applied or not.

In order to detect possible refactoring operations it parses the code from the beginning to the end and tries to check if a refactoring is applicable. To do that it tries to match every

Table 1: Data Structures

Name	AST	PDG	Database	Others
Griswold	X	X		
Rope	X		X	
Bicycle	X		X	
Pycharm Edu	X		X	
Javascript				X

syntax expression using syntax parse. In other words it uses brute force to check whether a expression can be applied a refactoring operation or not.

To properly display this information it highlights the source-code indicating that there is a possible refactoring. This feature could be improved by having a set of colors for the different types of refactoring operations. Moreover, the color intensity could be proportional to the level of suggestion. e.g the recommended level to use extract function refactoring increases with the number of duplicated code found.

6. ANALYSIS

The Table 1 summarizes the data structures of the analyzed refactoring tools. It is clear that the AST of a program is an essential part of the refactoring tool information with every refactoring tool having an AST to represent the program. Regarding the PDG and Database it contains mainly information about the def-use-relation of the program. The PDG has also control flow information of the program. Our implementation uses the same data structures, the AST and the def-use-relations.

Some tools like the one build by Griswold focus on the correctness of the refactoring operations and therefore need more information about the program, such as the information provided by the PDG. Others, focus in offering refactoring operations for professional or advanced users. However, the goal of our refactoring tool is to provide refactoring operations designed for beginners. Therefore we are not interested in refactoring operations formerly proven correct or provide refactoring operations only used in advanced and complex use cases.

We intend to have simple, useful, and correct refactoring operations to correct the typical mistakes made by beginners. With this we exclude from the refactoring tool scope macros usage, classes, and other complex structures not often used by beginners.

7. EVALUATION

In this section we present some code examples written by beginner programmers in their final project of an introductory course and their possible improvements using the refactoring operations available in our refactoring tool. The examples show the usage of some of the refactoring operations previous presented and here is explained the motivation for their existence.

The first example is a very typical error made beginner programmers.

```
(if (>= n_plays 35)
  #t
  #f)
```

It is rather a simple refactoring operation, but nevertheless it improves the code.

```
(>= n_plays 35)
```

The next example is related with the conditional expressions, namely the `and` or `or` expressions. We decided to choose the `and` expression to exemplify a rather typical usage of this expression.

```
(and
  (and
   (eq? #t (correct-moviment? player play))
   (eq? #t (player-piece? player play)))
  (and
   (eq? #t (empty-destination? play))
   (eq? #t (empty-start? play))))
```

Transforming the code by removing the redundant `and` expression makes the code cleaner and simpler to understand.

```
(and (eq? #t (correct-moviment? player play))
     (eq? #t (player-piece? player play))
     (eq? #t (empty-destination? play))
     (eq? #t (empty-start? play)))
```

However, this code can still be improved, the `(eq? #t ?x)` is a redundant way of simple writing `?x`.

```
(and (correct-moviment? player play)
     (player-piece? player play)
     (empty-destination? play)
     (empty-start? play))
```

While a student is still learning is common to forget whether or not a sequence of expressions need to be wrapped in a `begin` form. The `when`, `cond` and `let` expressions have a implicit `begin` and as a result it is not necessary to add the `begin` expression. Moreover, sometimes they still kept the `begin` keyword because they often use a trial and error approach in writing code. Our refactoring tool checks for those mistakes and corrects them.

```
(if (odd? line-value)
    (let ((internal-column (sub1 (/ column 2))))
      (begin
        (if (integer? internal-column)
            internal-column
            #f)))
    (let ((internal-column (/ (sub1 column) 2)))
      (begin
        (if (integer? internal-column)
            internal-column
            #f))))
```

This is a simple refactoring operation and it does not have a big impact, however it makes the code clear and helps the beginner programmer to learn that a `let` does not need a `begin`.

```
(if (odd? line-value)
    (let ((internal-column (sub1 (/ column 2))))
      (and (integer? internal-column)
           internal-column))
    (let ((internal-column (/ (sub1 column) 2)))
      (and (integer? internal-column)
           internal-column)))
```

The next example shows a nested `if`. Nested `if`s are difficult to understand, error prone, and a debugging nightmare.

```
(define (search-aux? board line column piece)
  (if (> column 8)
    #f
    (if (= line 1)
      (if (eq? (house-board board 1 column) piece)
        #t
        (search-aux? board line (+ 2 column) piece))
      (if (= line 2)
        (if (eq? (house-board board 2 column) piece)
          #t
          (search-aux? board line (+ 2 column) piece))
        (if (= line 3)
          (if (eq? (house-board board 3 column) piece)
            #t
            (search-aux? board line (+ 2 column) piece))
          (if (= line 4)
            (if (eq? (house-board board 4 column) piece)
              #t
              (search-aux? board line (+ 2 column) piece))
            (if (= line 5)
              (if (eq? (house-board board 5 column) piece)
                #t
                (search-aux? board line (+ 2 column) piece))
              (if (= line 6)
                (if (eq? (house-board board 6 column) piece)
                  #t
                  (search-aux? board line (+ 2 column) piece))
                (if (= line 7)
                  (if (eq? (house-board board 7 column) piece)
                    #t
                    (search-aux? board line (+ 2 column) piece))
                  (if (= line 8)
                    (if (eq? (house-board board 8 column) piece)
                      #t
                      (search-aux? board line (+ 2 column) piece))
                    null)))))))))
```

It is much simpler to have a `cond` expression instead of the nested `if`. In addition, every true branch of this nested if contains `if` expressions that are or expressions and by refactoring those `if` expressions to `or`s it makes the code simpler to understand.

```
(define (search-aux? board line column piece)
  (cond [(> column 8) #f]
        [(= line 1)
         (or (eq? (house-board board 1 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 2)
         (or (eq? (house-board board 2 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 3)
         (or (eq? (house-board board 3 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 4)
         (or (eq? (house-board board 4 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 5)
         (or (eq? (house-board board 5 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 6)
         (or (eq? (house-board board 6 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 7)
         (or (eq? (house-board board 7 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [(= line 8)
         (or (eq? (house-board board 8 column) piece)
             (search-aux? board line (+ 2 column) piece))]
        [else null]))
```

However, this code could still be further improved by refactoring it into a `case`.

The examples presented above appear repeatedly in almost every code submission of this final project supporting the need to proper support to beginner programmers.

As seen in Table 2 the average reduction in LOC is 9.34%

Table 2: Refactoring Operations

Code #	1	2	3	4	5	Total
Initial LOC	582	424	332	1328	810	3476
Final LOC	545	373	300	1259	701	3178
Difference	37	51	32	69	109	298
Percentage	-6.36	-12.03	-9.64	-5.19	-13.46	-9.34
Remove Begin	11	4	6	9	5	35
If to When	4	0	0	0	0	4
If to And	3	1	0	0	0	4
If to Or	6	6	1	13	20	46
Remove If	0	3	7	6	3	19
Remove And	0	2	0	4	0	6
Remove Eq	0	4	0	0	0	4
Extract Function	0	3	0	0	4	7
If to Cond	0	0	0	2	3	5

which shows how useful are these refactoring operations. It also shows how many refactoring operations were applied. This tool is not only for beginners, during the development of the tool we already used some of the refactoring operations, namely the extract function, in order to improve the structure of the code.

7.1 Refactoring Python

Python is being promoted as a good replacement for Scheme and Racket in science introductory courses. It is an high-level, dynamically typed programming language and it supports the functional, imperative and object oriented paradigms. Using the architecture of this refactoring tool and the capabilities offered by Racket combined with an implementation of Python for Racket[10] [11] it is also possible to provide refactoring operations in Python.

Using Racket's syntax-objects to represent Python as a meta-language [12] it is possible use the same structure used for the refactoring operations in Racket to parse and analyze the code in Python.

However, there are some limitations regarding the refactoring operations in Python. Since Python is a statement base language instead of expression base, it raises some problems regarding the possibility of some refactoring operations.

Example of removing an If expression:

```
True if (alpha < beta) else False
```

```
(alpha < beta)
```

The next one shows an extract function:

```
def mandelbrot(iterations, c):
    z = 0+0j
    for i in range(iterations+1):
        if abs(z) > 2:
            return i
        z = z*z + c
    return i+1
```

```
1  def computeZ(z, c):
2      return z*z + c
3
4  def mandelbrot(iterations, c):
5      z = 0+0j
6      for i in range(iterations+1):
7          if abs(z) > 2:
8              return i
9          z = computeZ( z,c)
10     return i+1
```

It is important to note that this refactoring operations for Python are just only a prototype and the work is still in progress.

8. CONCLUSION

A refactoring tool designed for beginner programmers would benefit them by providing a tool to restructure the programs and improve what more knowledgeable programmers call "poor style," "bad smells," etc. In order to help those users it is necessary to be usable from a pedagogical IDE, to inform the programmer of the presence of the typical mistakes made by beginners, and to correctly apply refactoring operations preserving semantics.

Our solution tries to help those users to improve their programs by using the AST of the program and the def-use-relations to create refactoring operations that do not change the program's semantics. This structure is then used to analyze the code to detect typical mistakes using automatic suggestions and correct them using the refactoring operations provided.

There are still some improvements that we consider important for the user. Firstly, the detection of duplicated code is still very naive and improving to understand if two variables represent the same even if the names are different or even if the order of some commutative expressions is not the same would make a huge improvement on the automatic suggestion. Then it is possible to improve the automatic suggestion of refactoring operations by having different colors for different types of refactoring operations. With a lower intensity for low "priority" refactoring operations and a high intensity for higher "priority". Thus giving the user a better knowledge of what is a better way to solve a problem or what is a strongly recommendation to change the code. Lastly it would be interesting to detect when a developer is refactoring in order to help the developer finish the refactoring by doing it automatically [13].

9. ACKNOWLEDGMENTS

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

10. REFERENCES

[1] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[2] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, 39(4):204–223, 2007.

[3] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[4] Clements J. Flanagan C. Flatt M. Krishnamurthi S. Steckler P. & Felleisen M. Findler, R. B. DrScheme: A programming environment for Scheme. *Journal of functional programming*, 12(2):159–182, 2002.

[5] Flanagan C. Flatt M. Krishnamurthi S. & Felleisen M. Findler, R. B. DrScheme: A pedagogic programming environment for Scheme. *Programming Languages: Implementations, Logics, and Programs*, 12(2):369–388, 1997.

[6] William G Griswold. Program restructuring as an aid to software maintenance. 1991.

[7] Siddharta Govindaraj. *Test-Driven Python Development*. Packt Publishing Ltd, 2015.

[8] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. *ACM SIGPLAN Notices*, 46(10):119–138, 2011.

[9] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[10] Pedro Palma Ramos and António Menezes Leitão. An implementation of python for racket. In *7 th European Lisp Symposium*, 2014.

[11] Pedro Palma Ramos and António Menezes Leitão. Reaching python from racket. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, page 32. ACM, 2014.

[12] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 154–164. IEEE, 2000.

[13] Xi Ge, Quinton L DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 211–221. IEEE, 2012.