Universiteit Leiden
The Netherlands

Computer Science

Implementing skipping faulty code

for the Hedy programming language

Toni Škulj

Supervisors:
Giulio Barbero & Anna van der Meulen

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

29/06/2023

Abstract

This thesis will provide an overview of how in Hedy, a child-friendly programming language, the possibility of skipping code that contains mistakes is added. Originally, when a user makes a mistake, only an error message is show and no code is executed. After the implementation discussed in this paper, these mistakes will be skipped. Also, it will show how a source map is implemented to map Hedy code to Python code. The research shows that implementing these features significantly increased the time necessary to execute Hedy code when it contains a mistake.

Contents

1 Introduction

Learning from mistakes is a critical part of learning a new skill [10]. This is also true when studying programming. Unfortunately, when students make a mistake when writing code it often results in an error without any execution, leaving the student with a bad experience [13], this could especially be true when they are starting out, potentially dropping the subject [9]. An alternative way is to skip, or simply remove, the mistakes that are made and leaving everything that is correct intact. The student may be more inclined to have a more positive experience when at least something that the student did works correctly. Additionally, if the student is informed on why the code is skipped it could lead to more engagement with errors. This thesis will show an implementation of such a feature in the Hedy programming language and what the performance impact is on the code execution for the user.

This paper will guide the reader on how the skipping faulty code feature is implemented. Additionally, before the feature can be implemented a source map will be necessary for the Hedy programming language, since it is translated to Python before execution. This thesis will therefore show how a source map has been implemented in Hedy [11]. All the code written for this thesis is publicly available in the Hedy public GitHub repository[7]. Finally, quantitative data will be shown to reflect on the performance impact the additional features have on the code execution.

As of writing this thesis, no implementation was found in a programming language that has the feature of skipping mistakes in textual code and showing the user what the specific error is.

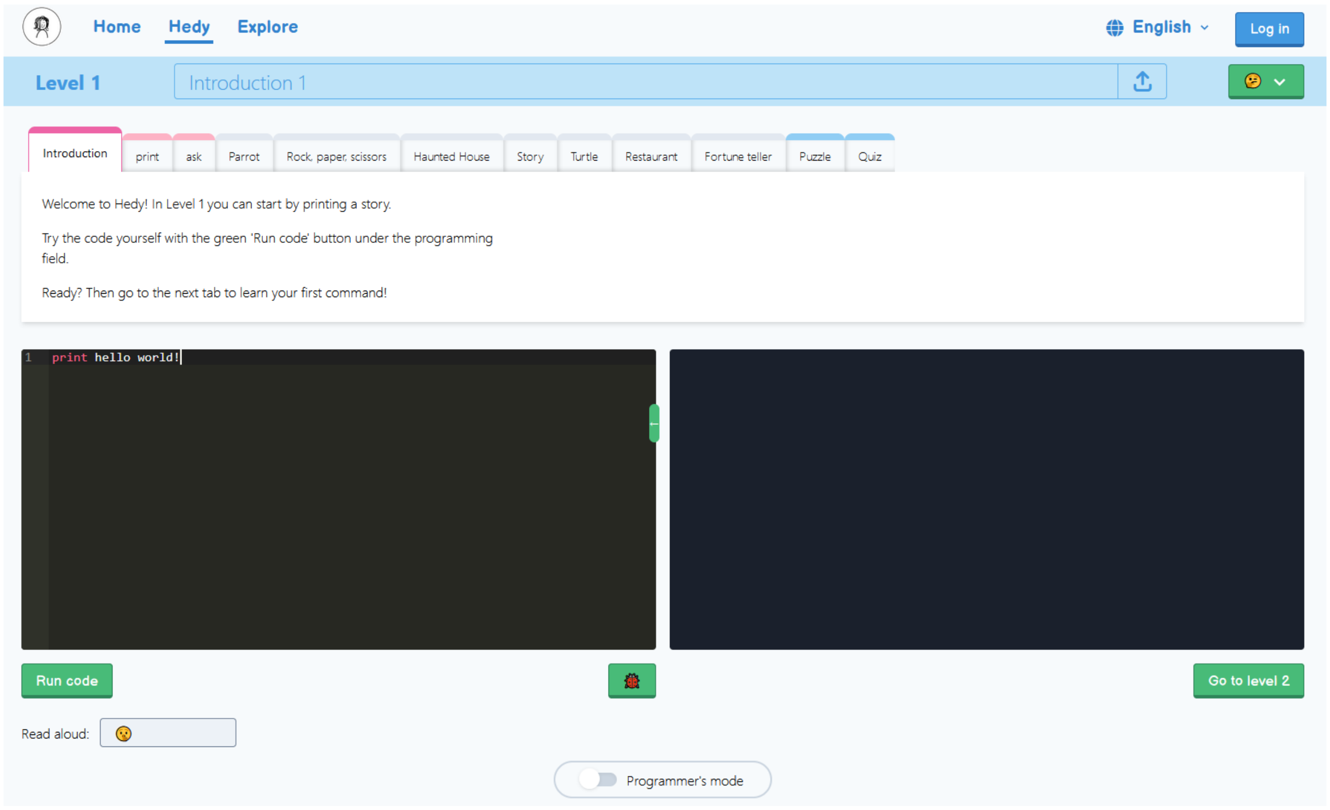
2 Background knowledge

Some background knowledge is required about a couple of concepts to understand the next chapters of this paper. Knowledge about Hedy, Hedy's architecture, and some terminology will be given.

2.1 Hedy

Hedy is a web-based, free-to-use, open-source, and multilingual programming language designed for children to learn to program [4]. The learning experience is gradual and done in levels [6], which means that children can learn one concept and its syntax at a time. This is done by introducing a new concept per level, the children are encouraged to solve these levels and continue to the next one. They learn new concepts along the way and ultimately learn how to partially write code in Python. The Hedy language has much resemblance to the Python programming language.

The Hedy web application is written in Python, using Flask, a lightweight WSGI web application framework [3]. For the front end, Hedy uses Typescript and Tailwind alongside the HTML that is generated by Jinja. A flexible NoSQL DynamoDB database is used for storing data [2].



Welcome to Hedy! In Level 1 you can start by printing a story.

Try the code yourself with the green 'Run code' button under the programming field.

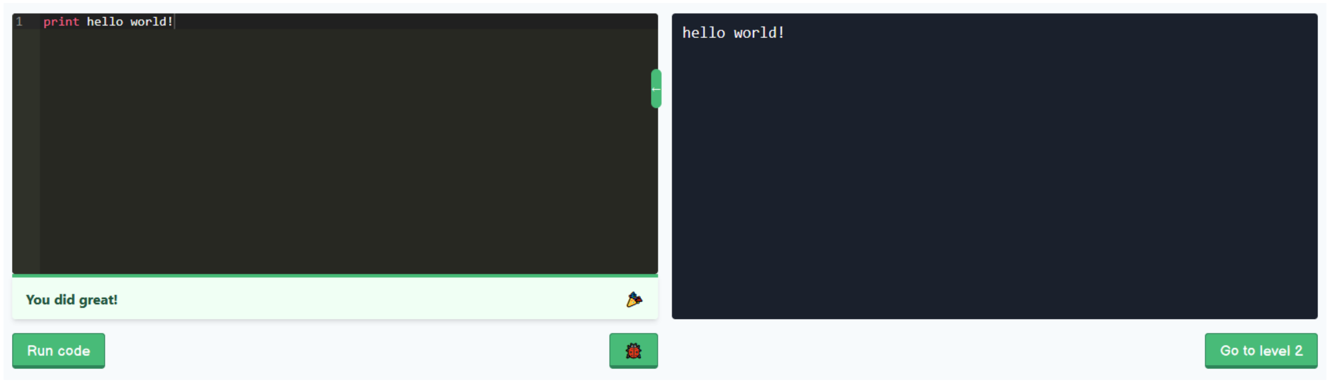
Ready? Then go to the next tab to learn your first command!

```
print hello world!
```

Figure 1: Hedy's layout, level 1

2.2 Hedy's architecture

When a user writes and executes valid Hedy code, the Hedy code is sent as a parameter within a POST request to the server where it is sanitized and parsed, Hedy makes use of Lark, a parsing toolkit for Python for defining grammar and parsing text[14]. The text generated is Python code. In simple terms, Hedy code is translated to Python. After that, the Python code is sent back to the user and the client side executes the Python code using Skulpt, a web-based Python interpreter [8]. Finally, the user sees the result of their written program next to the written code, visible in Figure 2



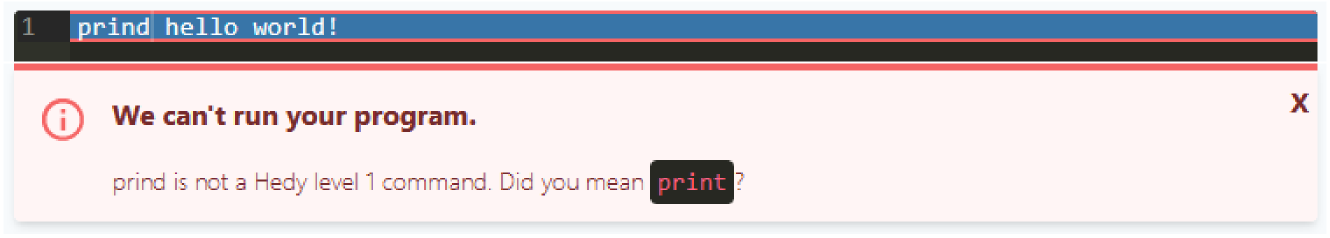
```
print hello world!
```

```
hello world!
```

You did great!

Figure 2: The result of executing a print statement in level 1

That is the case when the Hedy code is valid, meaning that the defined grammar is able to parse the text to a tree. When the code is invalid an exception is raised during sanitization or parsing, an error message is returned to the client, and no Python code is returned. This message is then displayed to the user. Also, the line that contains the incorrect code is highlighted if possible.



```
prind hello world!
```

We can't run your program.

prind is not a Hedy level 1 command. Did you mean `print`?

Figure 3: An error message is displayed when a mistake has been made

2.3 Hedy's parsing & transforming

Since Hedy is a gradual programming language that introduces new concepts at different levels, the grammar of the language differs per level. Before parsing the Hedy text received from the user the text is sanitized and checked for mistakes, if there are mistakes an exception is raised, and the text is not parsed [5]. Otherwise, the text is parsed. After parsing, the grammar rules are transformed, the Hedy text is transformed into the corresponding Python text. Again, since the Hedy language changes per level, each level has its own transformer and the necessary methods to transform the rules. Each transformer method name must correspond to the grammar rule name. Because this is necessary information for Lark to be able to know what method to use for what rule. Listing 1 and Listing 2 show a pair of grammar rules and the corresponding transformer method for the print statement in level 1.

```
_PRINT: ("print" | "print") _SPACE?
print: _PRINT (text)?
```

Listing 1: Grammar rules used for the print command in level 1

```python
@v_args(meta=True)
@hedy_transpiler(level=1)
class ConvertToPython_1(ConvertToPython):
    def print(self, meta, args):
        # escape needed characters
        argument = process_characters_needing_escape(args[0])
        return "print('" + argument + "')"
```

Listing 2: Transformer method used for the print grammar rule in level 1

Level 1 contains the base grammar, further levels inherit the previous levels and contain the wanted changes. The same methodology applies to the transformer classes.

2.4 Hedy's error handling

There are four places in the Hedy code that can result in an exception, voluntarily raised or not, that will stop the parsing or the transforming of the Hedy text.

ER1: During sanitization, the text contains mistakes

ER2: During parsing, no grammar rules are defined for the given text

ER3: After parsing, the parse tree contains error production nodes

ER4: During transforming, the corresponding transformer method raises an error

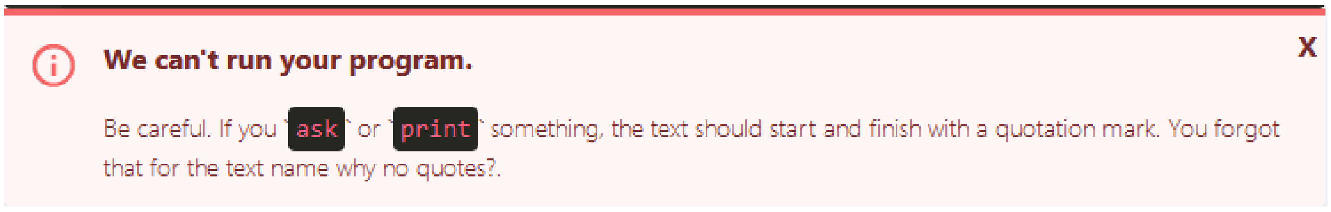
For **ER1**, the Hedy text is checked for various mistakes, including but not limited to:

- if the program is not longer than 100 lines
- if the code starts with a space
- use of the correct indentation amount (from level 8)

For **ER2**, if Lark cannot find a grammar rule to use for a certain piece of text a parse exception is raised.
For **ER3**, Hedy has some additional grammar rules that catch mistakes that are more clearly defined, so that the error message returned to the user is more helpful. For example, Listing 3 shows a grammar rule defining a rule when the *ask* command is used without quotes (' or ") in level 4. When such grammar rules are encountered a specific exception is raised. Resulting in the specific error message visible in Figure 4. This is done by searching the parse tree for these *error production nodes*, in the example, the tree contains an *error_ask_no_quotes* node.

```
error_ask_no_quotes: var _IS _ASK text  -> error_print_nq
```

Listing 3: Grammar rule for missing quotes when *ask* is used, in level 4


We can't run your program.

Be careful. If you `ask` or `print` something, the text should start and finish with a quotation mark. You forgot that for the text name why no quotes?.

Figure 4: The result of running **name is ask quotes** in level 4, the *ask* command is missing quotes

For **ER4**, the errors may be involuntarily raised where a certain unknown exception is raised. Even so, some errors are indeed raised explicitly. For example, when using “‘ in the print statement in level 12 the *print_ask_args* transformer method raises an exception.

2.5 Source map

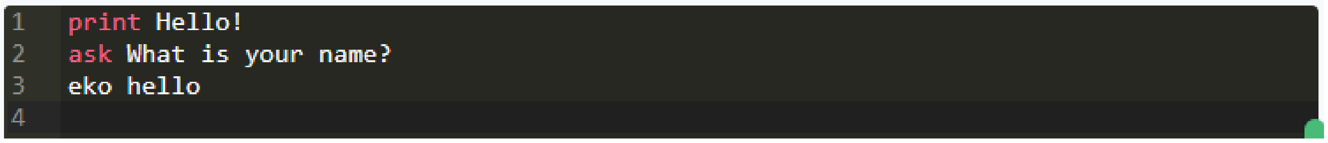
Throughout this paper, the word *source map* will be used repeatedly. This section will try to explain the meaning of the terms as well as to give an explanation. Later on in chapter 4 it will be shown how a source map is implemented within Hedy.

A source map is an object that maps the original source code to the converted source code[12]. A source map will be necessary before the skipping of mistakes can be implemented. Before we can skip mistakes, we need to know what Hedy code is incorrect. The Hedy code that is incorrect will need to be mapped to the null operation in Python. Furthermore, the Hedy code that is correct will need to be mapped to the corresponding Python translation. This mapping of incorrect and correct Hedy code will be the basis of the skipping of mistakes implementation.

3 Research problem & questions

3.1 Research problem

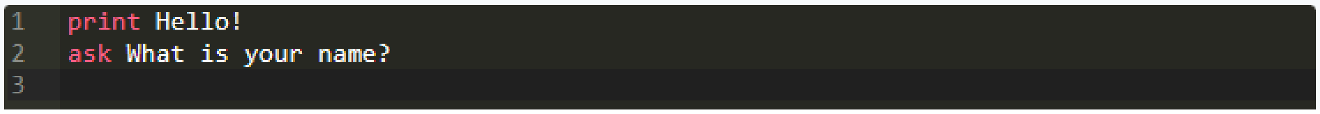
When an user makes a mistake within the Hedy code, only an error message is shown, without executing any code that might be correct. Instead, we want the mistakes to be skipped. This section of the paper will illustrate what the deliverable of the research is to solve this. This will be demonstrated with some visualizations. The goal is to skip faulty sections of Hedy code. The Hedy code snippet in Figure 5 contains a mistake on line 3.



```
1  print Hello!
2  ask What is your name?
3  eko hello
4
```

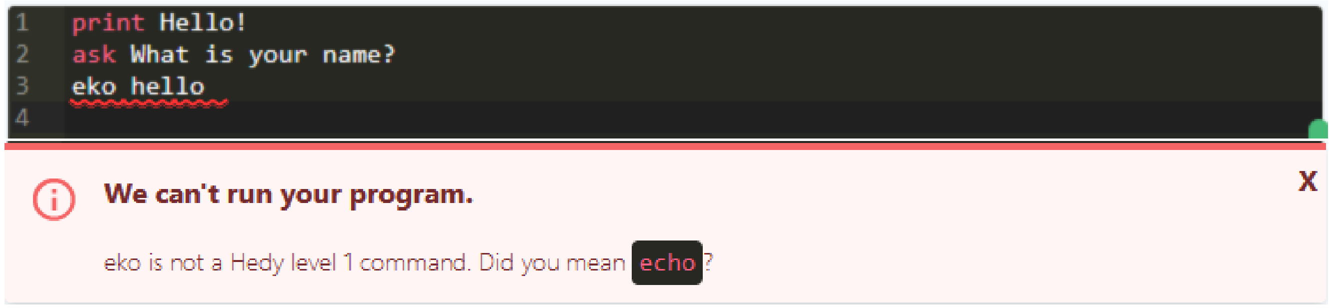
Figure 5: Hedy code snippet with a mistake in line 3

The *echo* command is misspelled as *eko*. The goal is to remove *eko hello* from the program. The goal is to semantically give the code the same meaning as the code snippet seen in Figure 6. Here lines 1 and 2 are still intact, but line 3 is removed since it was invalid. Additionally, the user must be informed of this mistake. The sections that are removed will be marked with red underlining. We still want to retain the original error message to be able to explain to the user what the mistake is. When the user clicks on the underlined section the original error message should be shown as depicted in 7.



```
1  print Hello!
2  ask What is your name?
3
```

Figure 6: Hedy code snippet with mistake in line 3



```
1  print Hello!
2  ask What is your name?
3  eko hello
4
```

We can't run your program.

eko is not a Hedy level 1 command. Did you mean `echo`?

Figure 7: Hedy code snippet with marked faulty line, error when clicked on faulty line

It should also be made possible to skip multiple sections and keep any code in between, if valid, still intact. This adds the possibility to receive multiple errors, or feedback, on one single run, instead of multiple runs of the same code snippet.

3.2 Research questions

We want to answer the following research questions:

RQ1: Can we implement a source map that maps Hedy code to Python?

RQ2: Can we implement skipping faulty code for Hedy?

RQ3: What performance impact does adding the source map and skipping faulty code have on Hedy?

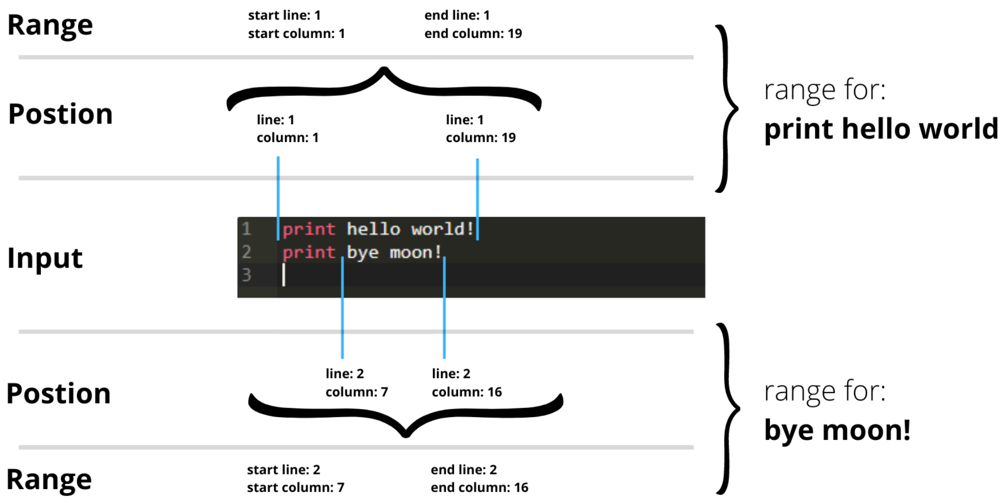
4 Methology of the source map

4.1 Representation

We will first illustrate how the source map is defined. Also, some critical design choices are depicted when the source map was implemented.

4.1.1 Defining source ranges

First, the ranges of the text (the location of the mapped code) within the full document, must be defined and implemented. We have decided to use the line number and column number to indicate the start and end position of a certain text segment. Both starting with the number 1. The source range consists of a start and end line number as well as a start and end column number. This way a range of text can be defined for the sources. An illustration of two source ranges in the Hedy code can be found in Figure 8. In the figure the text segments *print hello world* and *bye moon!* are mapped to their source ranges.



```
print hello world!
print bye moon!
```

Figure 8: Representation of the source ranges

Both the Hedy and Python source will use this method to define ranges within the source code.

4.1.2 Defining source codes

We define the source code to be nothing more than then Hedy or Python code and the corresponding source range. This way we know the range of the specific code. An illustration of the source code object can be found in Figure 9.

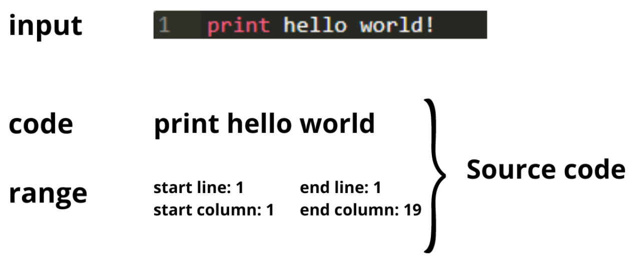
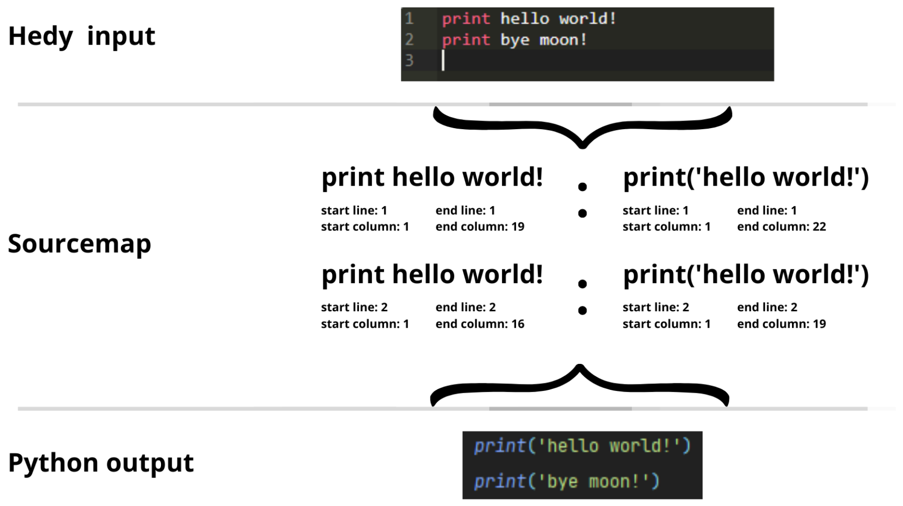


Figure 9: Representation of the source code

Here the code *print hello world!* and the appropriate source range is stored in the source code object. Again, both Hedy and Python code will use this definition.

4.1.3 Defining the source map

Because the mappings from Hedy to Python have a 1-to-1 relation we decided to use a dictionary to represent the source map. The key is the source code of the Hedy code and the value is the source code of the Python code, *source code* being the representation defined in 4.1.2. Next to the dictionary with mappings we also store the full Hedy program text as well as the Python program text inside of the source map object. An illustration of the source map object can be found in Figure 10. In this figure the mapping from the Hedy input to the Python output is visible. Notice that the source map keys contain Hedy code and the values contain Python code.



```
print hello world!
print bye moon!
```

```
print('hello world!')
print('bye moon!')
```

Figure 10: Representation of the source map and the corresponding code

4.2 Mapping Hedy to Python

Now that the source map is defined in chapter 4, we need a method to add mappings to the source map in Hedy. In order to do this, we implement a wrapper function that will add these mappings to the source map.

4.2.1 Wrapper function around transformer methods

As mentioned in chapter 2.3, all grammar rules must have a corresponding transformer method if we want to transform the text to Python code. We will use this to add mappings to the source map. Every time a rule is visited we want to add a source code entry with the Hedy code and the translated Python code to the source map. We make a wrapper function that we can use to wrap every transformer method. This function can be found in Listing 4.
Notice that the source range for the Python code is initiated as *none*, on line 20 in Listing **??**. This is because the parse tree is transformed in a recursive manner. We do not know the location of the mapped Python code within the full document because, during transforming, code may be added before the current processed rule, shifting the current rule downwards. How this is solved can be found later in chapter 4.3.

4.2.2 Wrapper function around transformer classes

Because adding the wrapping function mentioned earlier to all transformer methods is a large operation and this will complicate future adding of rules. We decided to make a *class wrapper* that will wrap all transformer methods with the before-mentioned wrapper function. This way only the transformer class needs to be decorated with this class wrapper named *source_map_transformer*. Listing 5 shows level 1 being decorated with this class wrapper, all 18 transformers have been decorated as such.
This is done by first extracting all grammar rule names from the grammar. The methods in the transformer classes that have grammar rule names are wrapped with the wrapper function dynamically.

4.3 Source ranges for Python

After all mappings have been added, the only task left is to find the source ranges of the Python code within the full Python document. When the full Python document is generated we loop over all mappings, take the Python code, and use Python's built-in *find* to find the start character position within the document. We add the length of the mapped code to this start position to obtain the end character position. We use an auxiliary function to split up the character position into a line number and a column number. This is then stored in the source map, replacing the None's that were used.

4.3.1 Resolving duplicates in Python code

A problem arises when the same Python code is used multiple times within the full document. When using the *find* built-in function it returns the first occurrence in the given text. To solve this, we make sure to change the start character position of the lookup within the full document

```python
def source_map_rule(source_map: SourceMap):
    def decorator(function):
        def wrapper(*args, **kwargs):
            meta = args[1]
            generated_python = function(*args, **kwargs)

            hedy_code_input = source_map.hedy_code[meta.start_pos:meta.end_pos]
            hedy_code_input = hedy_code_input.replace('#ENDBLOCK', '')

            hedy_code = SourceCode(
                SourceRange(
                    meta.container_line, meta.start_pos,
                    meta.container_end_line, meta.end_pos
                ),
                hedy_code_input
            )

            python_code = SourceCode(
                # We don't know now, set_python_output will set the ranges later
                SourceRange(None, None, None, None),
                generated_python
            )

            source_map.add_source(hedy_code, python_code)
            return generated_python

        return wrapper

    return decorator
```

Listing 4: wrapper function to add mapping to the source map

```python
@v_args(meta=True)
@hedy_transpiler(level=1)
@source_map_transformer(source_map)
class ConvertToPython_1(ConvertToPython):
```

Listing 5: Transformer class level 1 wrapped with source_map_transformer

based on the number of times we already have seen it. We first take the character position of the first occurrence and set the source range. After that, we store the mapped code in a list. For every mapping that we encounter, we use the built-in *count* on the stored list to see if the exact same mapped code already has been seen, and how often. The count is then used to loop that many times to change the start character index by shifting the start character position with the length of the code snippet. Essentially, shifting the start character position to the point after the last occurrence that already has been set and is stored within the list. This way we always set the source range for all occurrences appropriately. This method has been used because the source map is implemented as a dictionary, which in Python is unordered. This way, no sorting is required.

4.4 Client response object

It had been decided to make the response object a list of mappings returned to the client. The representation that we chose *per mapping* can be found in Listing 6, the response contains a list with these mappings, the amount depending on the number of mappings. Notice that the Hedy and Python code is not being returned. This is because, if the code is needed, it can be derived from the source range and the full Hedy and Python document.

```
'hedy_range': {
    'from_line': hedy_source_code.source_range.from_line,
    'from_column': hedy_source_code.source_range.from_column,
    'to_line': hedy_source_code.source_range.to_line,
    'to_column': hedy_source_code.source_range.to_column,
},
'python_range': {
    'from_line': python_source_code.source_range.from_line,
    'from_column': python_source_code.source_range.from_column,
    'to_line': python_source_code.source_range.to_line,
    'to_column': python_source_code.source_range.to_column,
},
```

Listing 6: Representation of the source map response object

5 Methology of skipping faulty code

Now that the source map has been implemented, the implementation of skipping mistakes can be added. To be able to skip mistakes we need to change the way the transpiling is done, adapt the Hedy grammar, and adapt the source map implementation.

5.1 Adapting the transpiling

When the client posts a request to execute code, the client initially has the *skip_faulty* parameter set to false. Therefore, the transpile function, which is responsible for translating Hedy to Python, will first execute the code without skipping faulty code. If on the other hand, the client posts a request with *skip_faulty* parameter set to true, the code will be transpilled with skipping faulty code. Why this is implemented in this way can be found in 5.7. The *transpile_inner_with_skipping_faulty* function is responsible for the transpiling of the Hedy code with skipping faulty code enabled. The adapted transpile function is visible in Listing 7.

```python
def transpile(input_string, level, lang="en", skip_faulty=False):
    if not skip_faulty:
        try:
            source_map.set_skip_faulty(False)
            transpile_result = transpile_inner(
                input_string, level, lang, populate_source_map=True
            )
        except Exception as original_error:
            # store original exception
            source_map.exception_found_during_parsing = original_error
            raise original_error
    else:
        original_error = source_map.exception_found_during_parsing
        source_map.clear()

        if isinstance(original_error, source_map.exceptions_not_to_skip):
            raise original_error

        try:
            source_map.set_skip_faulty(True)
            transpile_result = transpile_inner_with_skipping_faulty(input_string, level, lang)
        except Exception:
            raise original_error  # we could not skip faulty code, raise original exception

    return transpile_result
```

Listing 7: The function responsible of transpiling the Hedy code

5.2 Adapting the grammar rules

Some grammar rules must be adapted so that no exception is raised during transpiling when there are mistakes within the Hedy code. This chapter will explain those adaptations.

5.2.1 Adapting the error rules

When the *transpile_inner_with_skipping_faulty* function is called, the first rules that are adapted are the error grammar rules discussed in 2.4, which are defined for ER3. Normally, when these rules are encountered within the parse tree, the corresponding transformer method raises an exception, leading to a specific error shown to the user, this is done before the transpiling of the Hedy code to Python. These functions are dynamically changed to return *true*, leading to no exceptions. This is necessary because we want the text to be transformed later on, not be halted with an exception.

5.2.2 Adapting other rules

We add an *error_invalid* grammar rule that will catch any text for all commands if these are invalid. This rule is already defined for levels 1 to 5, we extend its use to all levels. This grammar rule will function as a 'bucket' for faulty commands. We also make sure that this grammar rule has the lowest priority when considering the parsing of text. We do this by setting the priority to -100. Some additional rules are adapted to improve ambiguity within the grammar. Notice that this rule is will also adapted by *transpile_inner_with_skipping_faulty*.

5.3 Adapting the source map

The wrapper function discussed in 4.2.1 needed to be adapted to support skipping faulty code and to map it into the source map. After adapting the error rules, when the faulty text segment is transpiled by the appropriate transformer method, one of the following may happen.

TR1: an exception may occur, the transformed method was not designed for this text

TR2: the transformer method may return a parse tree instead of a string

Either way, these indicate that there is something wrong with the code. We make sure to check for both of these cases, if present, we define the *error* variable, which is visible in Listing 8. The generated tree is casted to a string in some instances, we therefore check not only the type but also if the string matches a string representation of a Lark tree instance. We do this by using a regular expression, visible in line 8 in Listing 8.
When an exception is raised, we set the generated code to be *pass*, which is a null operation in Python, instead of the result of the transformer method. The faulty text segment is replaced with the null operation. Additionally, we make sure to add the error to the hedy mapping, visible in Listing 9. This marks the mapping as faulty. The error will be replaced with one of the regular errors in Hedy later on, discussed in 5.4.

```python
if not source_map.skip_faulty:
    generated_python = function(*args, **kwargs)
else:
    try:
        generated_python = function(*args, **kwargs)
        if (
            isinstance(generated_python, Tree) or
            bool(re.match(r".*Tree\(.*Token\(.*\).*\).*", generated_python))
        ):
            raise Exception('Can not map a Lark tree, only strings')

    except Exception as e:
        # If an exception is found, we set the Python code to pass (null operator)
        # we also map the error
        generated_python = 'pass'
        error = e
```

Listing 8: Skipping adaptation of source map wrapper function

```python
hedy_code = SourceCode(
    SourceRange(
        meta.container_line, meta.container_column,
        meta.container_end_line, meta.container_end_column
    ),
    hedy_code_input,
    error=error
)
```

Listing 9: source code object for the Hedy source code, including an error parameter

5.4 Finding the original error per mapping

After finding what mappings give an error, discussed in 5.3, the error that Hedy would normally give when not skipping faulty code needs to be found, per mapping. To achieve this, after mapping by skipping faulty code. The source map is searched for mappings with an error. For every mapping with an error we transpile the mapped Hedy code again without enabling skipping faulty code, this results in the correct exception. We replace the error on the mapping with this error. This is done at the end of the *transpile_inner_with_skipping_faulty* function, visible in Listing 10.

```python
at_least_one_error_found = False

for hedy_source_code, python_source_code in source_map.map.copy().items():
    if hedy_source_code.error is not None or python_source_code.code == 'pass':
        try:
            transpile_inner(hedy_source_code.code, source_map.level, source_map.language)
        except Exception as e:
            hedy_source_code.error = e

        if hedy_source_code.error is not None:
            at_least_one_error_found = True

if not at_least_one_error_found:
    raise Exception('Could not find original error for skipped code')
```

Listing 10: Setting the original error per mapping

If, for any reason, no exception is raised during the transpiling of the mapped faulty code, we raise an exception manually, which indicated that we could not find any original exception.

5.5 Adapting the method of testing

Before the implementation of skipping faulty code, the Hedy code-base contained integration tests to validate that a certain mistake within the Hedy code would result in an appropriate exception. Therefore, the correct error message to the user. Adding the implementation of skipping code invalidates these tests because the exceptions are still raised but captured and not returned to the user. To solve this problem we changed all the tests that assert an exception. We added an *SkippedMapping* object that will store the source range and the appropriate exception that we want to encounter on the defined source range. We added these to a list because more than one section can be skipped, and the developer can add more than one SkippedMapping objects to be tested this way. Listing 11 shows such a SkippedMapping object on line 6. When tested, the source map of the given test code will be searched to find if for every SkippedMapping object that source range contains the correct error. In addition to that, the expected Python code, that is generated from the Hedy code, after replacing the skipped code with *pass*, is also tested and checked for equality.

Listing 11 shows the test for a Hedy code snippet that starts with a space, found on line 2, which is invalid. The expected result is found on line 3, in this case, *pass* is the result.

```
1  def test_print_with_space_gives_invalid(self):
2      code = " print Hallo welkom bij Hedy!"
3      expected = "pass"
4
5      skipped_mappings = [
6          SkippedMapping(SourceRange(1, 1, 1, 30), hedy.exceptions.InvalidSpaceException)
7      ]
8
9      self.multi_level_tester(
10         code=code,
11         expected=expected,
12         skipped_mappings=skipped_mappings,
13         max_level=1)
```

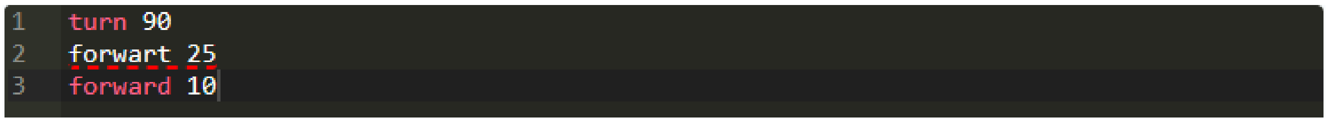
Listing 11: A test that tests skipping faulty line that starts with a space

5.6 Adding underlining of faulty code

To add the underlining to the faulty and removed code, we loop through the mappings and check if the mapping contains an error. If the mapping contains an error, we underline that source range with a red line in ACE, an embeddable code editor written in JavaScript [1], we do this by wrapping the text segment in a span element and giving it a CSS class with appropriate styling. Listing 12 shows the Typescript code responsible for doing this. The ACE editor uses 0 as the first number to define ranges, instead of 1, which has been implemented for the source map. We therefore subtract 1 from all values before passing to the ACE marker. Notice also that the markers all get a unique *index* to their CSS class on line 9 in Listing 12 which corresponds to the index in the source map, this is necessary so that later we can define what error message belongs to this text segment.
We chose a dashed line so that it is clearly distinctive as wrong to users with color vision deficiency. The result of that can be found in Figure 11. In this Hedy program *forward* has been misspelled as *forwart* on line 2. Additionally, we make sure that when the underlined mistake is clicked the appropriate error message is shown to the user. This is done by adding an event handler to the ACE editor when it is clicked. After clicking, we inspect what element is on the clicked page X and Y position, and retrieve the CSS class. We strip 'ace_incorrect_hedy_code_' away which leaves us with the index of the corresponding mapping in the source map. We take the error of that mapping and display it to the user, visible in Figure 12.

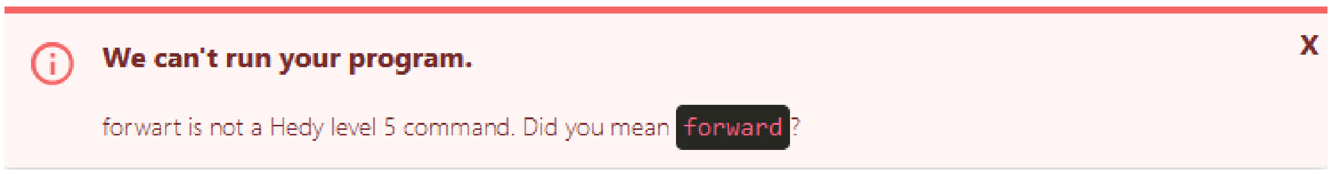
```
for (const index in sourceMap) {
  const map = sourceMap[index];
  const range = new Range(
    map.hedy_range.from_line-1, map.hedy_range.from_column-1,
    map.hedy_range.to_line-1, map.hedy_range.to_column-1
  )

  if (map.error != null){
    markers.addMarker(range, `ace_incorrect_hedy_code_${index}`, "text", true);
  }
}
```

Listing 12: Adding underlining markers to faulty code



```
1 turn 90
2 forwart 25
3 forward 10
```

Figure 11: Error underlined in line 2, *forward* has been misspelled



We can't run your program.

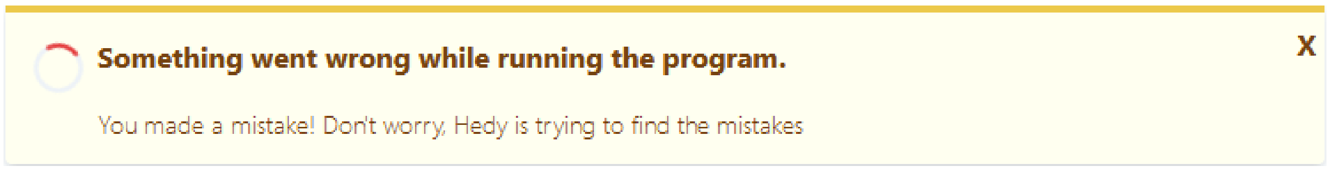
forwart is not a Hedy level 5 command. Did you mean forward?

X

Figure 12: Error message shown after clicking on the underline section in Figure 11

5.7 Adding a warning message

Because we transpile the mapped Hedy code that contains an error again it may take some time, leaving the user without feedback. Because of this, we have added a warning message that is displayed to the user that an error has been found and the server needs some time to transpile the code again. The error message that is shown can be found in Figure 13.



Something went wrong while running the program.

X

You made a mistake! Don't worry, Hedy is trying to find the mistakes

Figure 13: Error message shown to the user if a mistake is encountered during transpiling

This feature is implemented by making the client first send the code to the server with a parameter *skip_faulty* set to *false*. The code is transpiled as it would originally and if an error is found the appropriate exception is raised and returned to the user. On the client side, this error results in the client posting the code again but with *skip_faulty* set to *true*. After that, the error message is shown depicted in Figure 13. This time the server will transpile the code by also skipping faulty code. Another response is returned after posting the code the second time, be it an error or actual code, the warning message is removed and the error is shown or the code is executed with the faulty code underlined. To indicate to the user that the server is doing operations, a spinner has been added to the left top corner, visible in Figure 13.

6 Performance impact on Hedy

This section will discuss the performance impact the source map and skipping faulty code implementation, combined, have on Hedy's code execution time. The Firefox network develop tool has been used to obtain the results. All tests have been executed locally so that internet speed and stability do not interfere with the results.

6.1 With and without source map

A set of code samples was tested to obtain the performance impact of adding the source map to Hedy. Table 1 shows the results obtained. Only code snippets without mistakes have been tested since no source map is generated when the code results in an error.

	N	**Mean (ms)**
Without source map	5	583
With source map	5	604

Table 1: Results of waiting time on Hedy code snippets

The results show an increase of 3.6% on average in waiting time when adding the source map implementation.

6.2 With and without skipping faulty code

Multiple code snippets were tested on various levels and the waiting time, the time during the POST request and a response, were noted. Because the Hedy code with a mistake will be posted twice when skipping faulty code, discussed in 5.7, the sum of the two waiting times has been taken. The code snippets that contain a mistake contain only one mistake. The results are visible in Table 2. Notice that the standard deviation is relatively high between the executed tests, indicating that the tests are relatively spread out.

		N	Mean (ms)	Std. Deviation
Without skipping faulty	**With mistake**	5	437	132
	Without mistake	5	583	450
With skipping faulty	**With mistake**	5	1876	1010
	Without mistake	5	633	508

Table 2: Results of waiting time on Hedy code snippets

The results show an 8% increase on average in waiting time when no mistake is made. The increase, with respect to the results in 6.1, could be explained due to the extra work that has to be done because of the adaption of the source map, discussed in 5.3. The result is much more significant between the two scenarios when a mistake is made, showing an average increase of 329%. This lines up with the implementation because the code containing a mistake must be posted twice by the client, resulting in more waiting time. Additionally, when a mistake is made, the mapped faulty code must be transpiled again, explained in 5.4.

6.3 Impact of multiple mistakes

A constant code snippet in level 14 has been tested on multiple mistakes. The performance impact of one or more mistakes is visible in Table 3. Showing the additional time it takes when a mistake is added to the same code snippet.

	# Mistakes	Time (ms)	Increase (ms)
With skipping faulty	1	3219	-
	2	4551	1332
	3	5695	1144
	4	6945	1250

Table 3: Results of waiting time on Hedy code snippets

Based on the results in Table 3, the average increase when adding an error is 1242ms. Again lining up with the implementation, since all mapped faulty code must be transpiled again, adding more mistakes will increase the amount of transpiling, resulting in a higher waiting time.

7 Limitations of skipping faulty code

Not all mistakes can be skipped, there is a possibility that the *transpile_inner_with_skipping_faulty* function will raise an exception. If the mistake is not mapped to any error production rule, not even error_invalid, a parse error will be raised. If this happens we simply raise the original exception. This way if something unforeseen happens, the original Hedy error will be returned to the user.

8 Conclusion

To conclude, this thesis consists of three research questions that were tried to be answered. This section will recap the questions and the results.

RQ1: Can we implement a source map that maps Hedy code to Python?

The presented paper shows that implementing a source map in Hedy is indeed possible. The source map successfully maps Hedy to Python code. This was done by capturing the result of the transformer methods and storing them in a dictionary.

RQ2: Can we implement skipping faulty code for Hedy?

The thesis shows that indeed such a feature is possible to implement. By adapting the grammar, way of transpiling and testing, it is possible to skip mistakes in the Hedy code. Furthermore, indicating to the user where the mistake is and what the mistake is, is also shown to be possible.

RQ3: What performance impact does adding the source map and skipping faulty code have on Hedy?

Finally, quantitative tests have been done to test how the addition of the source map and the skipping faulty code impacts the performance of Hedy. The waiting time, from posting the code to the servers, to the response, is substantially higher when code is executed that contains a mistake. This is due to the additional transpiling that has to take place, which is required to obtain the correct error message per mistake. Also, the code must be posted twice to the servers when mistakes are made, adding more waiting time.

9 Discussion

The outcome of this research has shown that implementing a source map and skipping mistakes is possible and also shows how this can be done. This could be of great benefit to children who start out learning programming. For them, the negativity that comes with making mistakes could prevent them to continue studying programming. Skipping mistakes could soften this negativity about making mistakes and therefore increase the likelihood that they will continue studying. However, it could also confuse more older students who are not used to mistakes that are simply skipped within code. Furthermore, not all mistakes are skipped, possibly confusing the students on what will and will not be skipped.
In section 6.3, Table 3 the results of the waiting time on Hedy code snippets can be seen. As said

before the increase in errors will lead to an increase in the waiting time. A possible solution to decrease this waiting time can be obtained by parallelizing the transpiling of the code instead of using a for-loop.

10 Further Work

In this chapter relevant work that could improve the source map, skipping faulty code, or the research in general is discussed.

10.1 Performance optimization

Currently, as also said in 9, the transpiling of the faulty mapped code happens in a for-loop. This could be optimized by parallelizing the transpiling of the code. This could significantly reduce the waiting time when multiple mistakes are made, discussed in 6.3.

10.2 Adding End-to-end tests

No end-to-end tests have been written for the skipping faulty code implementation. It would be beneficial for the robustness and correctness of the code base to add end-to-end tests. These end-to-end tests could test if certain mistakes are indeed underlined as expected, as well as give the correct error when clicked.

10.3 Adding error production rules

Because the implementation of skipping mistakes relies on the error production rules. It would be beneficial to extend these grammar rules, to be able to skip more specific mistakes. Subsequently, this would also improve the feedback given to the user, since the feedback would become more detailed and specific.

10.4 Effects on children studying programming

Unfortunately, no research could be done to research the impact of skipping faulty code due to the extended time that was needed to implement the necessary features. It could be of great interest to research if children, who start to learn to program, are more inclined to continue studying if the mistakes they make are skipped. Furthermore, it could be interesting to see if the learning rate is faster due to skipping faulty code. The implementation allows for multiple feedback that originally would require multiple executions.

References

[1] Ace. https://ace.c9.io/. accessed: 01.06.2023.

[2] Dynamodb. https://aws.amazon.com/dynamodb/. accessed: 02.06.2023.

[3] Flask readme. https://github.com/pallets/flask/blob/main/README.rst. accessed: 02.06.2023.

[4] Hedy. https://hedy.org/. accessed: 01.06.2023.

[5] Hedy design. https://github.com/hedyorg/hedy/blob/main/DESIGN.md. accessed: 03.06.2023.

[6] Hedy levels. https://github.com/hedyorg/hedy/blob/main/LEVELS.md. accessed: 02.06.2023.

[7] Hedy's github repository. https://github.com/hedyorg/hedy. accessed: 01.06.2023.

[8] Skulpt. https://skulpt.org/. accessed: 01.06.2023.

[9] Theresa Beaubouef and John Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bulletin*, 37:103–106, 06 2005.

[10] D.R. Chialvo and P. Bak. Learning from mistakes. *Neuroscience*, 90(4):1137–1148, 1999.

[11] Felienne Hermans. Hedy: A gradual language for programming education. pages 259–270, 08 2020.

[12] Mozilla. Use a source map. https://firefox-source-docs.mozilla.org/devtools-user/debugger/how_to/use_a_source_map/index.html. accessed: 12.06.2023.

[13] Martinha Piteira and Carlos Costa. Learning computer programming: Study of difficulties in learning programming. pages 75–80, 07 2013.

[14] Erez Shinan. Lark readme. https://github.com/lark-parser/lark/blob/master/README.md. accessed: 18.06.2023.