homotopy Documentation

Release 0.1

Nenad Vasic

Sep 16, 2018

Contents:

CHAPTER 1

Getting started

Homotopy is a snippet language. It is designed with the focus on writing code and not caring about how it looks. That is handled automatically.

This document designed to get you up and running with using Homotopy. It uses *C++* as a working language. To see snippets from standard library go to *Snippet library*.

1.1 Hello world

```
stdinc$stdio.h& &[[main]]>printf("Hello, world\!");& &return 0;
```

```
#include <stdio.h>

int main(int argc, char* argv[]){
    printf("Hello, world!");

    return 0;
}
```

That was a lot of code for one line. Following sections wil gradually introduce concepts used in the *“Hello, World!”* example.

1.2 Language

1.2.1 Snippet definitions

Snippet is defined as text with placeholders for parameter values.

```
{"name": "stdinc","language": "C++","snippet": "#include <$$$>"}
```

This is the definition for *C++* snippet for including from standard library. `$$$` is a placeholder for parameter `$`.

Example:

```
stdinc$stdio.h
```

```
#include <stdio.h>
```

1.2.2 Parameters

Parameters are used to bind values to a snippet definition. Parameters begin with on of the following characters.

NOTE: This is just a convention that standard library follows. It is not enforced by any part of the tool. Have that in mind if creating custom snippets.

Parameter name	Description
!	Class name
~	Implements
:	Extends
^	Template
@	Method/function name
#	Type
$	Value
%	Other

```
func#void@foo
```

```
void foo(){

}
```

Parameters can be used more than once in the same snippet.

```
func#void@foo#int$i#int$j
```

```
void foo(int i, int j){

}
```

In the last example there are two adjacent int parameters of foo. Passing two value parameters first and then a single type parameter gives the same result.

```
func#void@foo$i$j#int
```

```
void foo(int i, int j){

}
```

1.2.3 Into

There is another, special, parameter, into >. It separates a parent and child snippets. Child snippet is substituted inside parent snippet at the placeholder >>> position.

```
if$i==4>printf("four");
```

```
if(i==4){
    printf("four");
}
```

In this way, several snippets can be combined to make up a larger construct.

Following control characters are used to combine snippets.

Parameter name	Description
>	Into
<	Out
&	And

Example with `<`:

```
if$i==4>printf("four");<printf("out of if");
```

```
if(i==4){
    printf("four");
}

printf("out of if");
```

Example with `&`:

```
printf("first line");&printf("second line");
```

```
printf("first line");

printf("second line");
```

1.2.4 Escape

Control character are picked so they don't interfere with code too much. Yet sometimes code contains control characters. To make any character part of snippet and not treat it as a parameter just put backslash character `'\'` in front of that character.

Example:

```
if$a \> max>max = a
```

```
if(a > max){
    max = a
}
```

1.2.5 Shortcuts

There are common construction that are tedious to write. To help with that, Homotopy has a concept of shortcuts.

Anywhere in the snippet, place a definition inside double square brackets and it gets expanded before compilation.

Example:

```
[[main]]
```

```
int main(int argc, char* argv[]){

}
```

1.3 Hello world deep dive

Lets take a look at the hello world snippet once again and go through the process of compilation in detail. This is a fairly large example but includes most of the feathers of Homotopy.

```
stdinc$stdio.h& &[[main]]>printf("Hello, world\!");&return 0;
```

At the top level there are three snippets:

1. `stdinc$stdio.h` and
2. Space for an empty line.
3. `[[main]]>printf("Hello, world\!");&return 0;`

They are implicitly inside an implicit `block` snippet. Block snippet just separate snippets by lines.

Definitions used to compile this snippet:

```
[
{"name": "stdinc","language": "C++","snippet": "#include <$$$>"},
{"name": "main","language": "C++","snippet": "func#int@main#int$argc#char*$argv[]"},
{"name": "func","language": "C++","snippet": "### @@@({{params}}){\n{{inside_wblock}}
↪\n}"},
{"name": "params","language": "C++","snippet": "### $$${{opt_params}}"},
{"name": "opt_params","language": "C++","snippet": ", ### $$${{opt_params}}"},
{"name": "inside_wblock","language": "C++","snippet": "\t>>>{{opt_inside_block}}"},
{"name": "opt_inside_block","language": "C++","snippet": "\n\t>>>{{opt_inside_block}}
↪"}
]
```

`stdinc` has the definition `#include <$$$>` and `stdio.h` just gets replaced in to get `#include <stdio.h>`

Lets now go through the third snippet step by step:

1. `[[main]]` gets expanded into `func#int@main#int$argc#char*$argv[]`.
2. `func` get expanded into `### @@@({{params}}){\n{{inside_block}}\n}`.
3. `###` gets replaced with `int`.
4. `@@@` gets replaced with `main`. Now, partial result is `int main({{params}}){\n{{inside_block}}\n}`.
5. `###` is not present in the current partial result so `{{params}}` gets expanded because it contains `###`. New partial result is `int main(### $$${{opt_params}}){\n{{inside_block}}\n}`.
6. `###` gets replaced with `int`.
7. `$$$` gets replaced with `argc`.
8. Similar to **5**, `{{opt_params}}` gets replaced with `, ### $$${{opt_params}}`.
9. `###` gets replaced with `char*`.

10. `$$$` gets replaced with `argv[]`.
11. Similar to **5** and **8**, `{{inside_block}}` gets replaced with `\t>>>{{opt_inside_block}}`.
12. Compile snippet `printf("Hello, world\!");`. This is trivial in this case and the result `printf("Hello, world!");`. `!` gets escaped and everything else stays the same.
13. `>>>` gets replaced with `printf("Hello, world!");`.
14. Similar to **5**, **8** and **11**, `{{opt_inside_block}}` gets replaced with `\n\t>>>{{opt_inside_block}}`.
15. `return 0;` get trivially compiled to `return 0;`.
16. `>>>` gets replaced with `return 0;`.
17. Result gets cleaned from sub-snippets like `{{opt_inside_block}}` and `{{opt_params}}`.

CHAPTER 2

Snippet library

This document contains snippet examples for most of snippets from standard library.

- *Core*
- *C++*
- *C*
- *Java*
- *Python*
- *JavaScript*

2.1 Core

There are few snippets that are used to create the structure of snippets. They can also be used to add parameters that can be used inside the snippet.

block:

```
block>line1&line2
```

```
line1
line2
```

---

wblock (wide block):

```
wblock>line1&line2
```

```
line1

line2
```

Note that `block` *is the implicit parent of all snippets. Next snippet sill have the same result.*

```
line1&line2
```

```
line1
line2
```

2.2 C++

2.2.1 Commands

call (function call):

```
call@foo$param1>param2
```

```
foo(param1, param2);
```

call2 (function call in multiple lines):

```
call2@foo$param1>param2
```

```
foo(
  param1,
  param2
);
```

stdinc:

```
stdinc$stdio.h
```

```
#include <stdio.h>
```

inc:

```
inc$homotopy.h
```

```
#include "homotopy.h"
```

2.2.2 Flow control

for:

```
for#int$i%0%n
```

```
for(int i=0; i<n; i++){

}
```

forr:

```
forr#int$i%n%0
```

```
for(int i=n; i>=0; i--){

}
```

forin:

```
forin#int$i%array
```

```
for(int i: array){

}
```

if:

```
if$true>printf("Always");
```

```
if(true){
    printf("Always");
}
```

else:

```
if$i==2>return 4;<else>return 3;
```

```
if(i==2){
    return 4;
}
else {
    return 3;
}
```

while:

```
while$true>printf("Forever and always");
```

```
while(true){
    printf("Forever and always");
}
```

switch:

```
switch$i>case$1>printf("one");<case$2>printf("two");
```

```
switch(i){
    case 1:
        printf("one");
        break;

    case 2:
        printf("two");
        break;
}
```

```
switch$i>case$1$2>printf("one or two");
```

```
switch(i){
    case 1:
    case 2:
        printf("one or two");
        break;
}
```

2.2.3 Objects

struct:

```
struct!pair>int first, second;
```

```
struct pair {
    int first, second;
};
```

class:

```
class!A:B%public>private>int a;<public>int b;
```

```
class A: public B {
private:
    int a;
public:
    int b;
};
```

enum:

```
enum!Colors>red&green&blue
```

```
enum Colors {
    red,
    green,
    blue
};
```

enum1 (enum in single line):

```
enum1!Colors>red&green&blue
```

```
enum Colors { red, green, blue };
```

2.2.4 Functions

func (function):

```
func#int@five>return 5;
```

```
int five(){
    return 5;
}
```

```
func#int@plus#int$i#int$j>return i+j;
```

```
int plus(int i, int j){
    return i+j;
}
```

```
func#int@plus$i$j#int>return i+j;
```

```
int plus(int i, int j){
    return i+j;
}
```

Note that values `i` *and* `j` *are specified first and type int after. This makes both* `i` *and* `j` *ints without typing int twice.*

method:

```
class!A>public>method#int@five>return 5;
```

```
class A {
public:
    int five(){
        return 5;
    }
};
```

nimethod (not implemented method):

```
class!A>public>nimethod#int@five
```

```
class A {
public:
    int five();
};
```

amethod (abstract method):

```
class!A>public>amethod#int@five
```

```
class A {
public:
    int five() = 0;
};
```

dmethod (deleted method):

```
class!A>public>dmethod#int@five
```

```
class A {
public:
    int five() = delete;
};
```

methodi1 (single method implementation):

```
methodi1!A#int@five>return 5;
```

```
int A::five(){
    return 5;
}
```

mithodi (method implementation):

```
wblock!A>methodi#int@five>return 5;<methodi#int@six>return 6;
```

```
int A::five(){
    return 5;
}

int A::six(){
    return 6;
}
```

Note that `wblock` *is used here to bind class parameter that is used by both children snippets.*

constr (constructor):

```
class!A>public>constr#int$i
```

```
class A {
public:
    A(int i){

    }
};
```

2.2.5 Templates

template:

```
class!A^T
```

```
template <class T>
class A {

};
```

```
func@nothing#void^T
```

```
template <class T>
void nothing(){

}
```

```
class!A>public>method#void@nothing^T
```

```
class A {
 public:
     template <class T>
     void nothing(){

      }
 };
```

2.2.6 Design Patterns

singleton:

```
class!A>[[singleton]]
```

```
class A {
public:
    A& getInstance(){
        static A instance;

        return instance;
    }
private:
    A(){}
    A(A const& origin);
    void operator=(A const& origin);
};
```

composite (class and method):

```
class!Composite:Component%public>[[compositeclass]]&public>method#void@traverse>
↪[[compositemethod]]
```

```
class Composite: public Component {
public:
    void add(Component *item){
        children.puch_back(item);
    }
private:
    std::vector<Component*> children;
public:
    void traverse(){
        for(int i=0; i<children.size(); i++){
            children[i]->traverse();
        }
    }
};
```

2.3 C

2.3.1 Commands

call (function call):

```
call@foo$param1>param2
```

```
foo(param1, param2);
```

call2 (function call in multiple lines):

```
call2@foo$param1>param2
```

```
foo(
  param1,
  param2
);
```

stdinc:

```
stdinc$stdio.h
```

```
#include <stdio.h>
```

inc:

```
inc$homotopy.h
```

```
#include "homotopy.h"
```

2.3.2 Flow control

for:

```
for#int$i%0%n
```

```
for(int i=0; i<n; i++){

}
```

forr:

```
forr#int$i%n%0
```

```
for(int i=n; i>=0; i--){

}
```

if:

```
if$true>printf("Always");
```

```
if(true){
    printf("Always");
}
```

else:

```
if$i==2>return 4;<else>return 3;
```

```
if(i==2){
    return 4;
}
else {
    return 3;
}
```

while:

```
while$true>printf("Forever and always");
```

```
while(true){
    printf("Forever and always");
}
```

switch:

```
switch$i>case$1>printf("one");<case$2>printf("two");
```

```
switch(i){
    case 1:
        printf("one");
        break;

    case 2:
        printf("two");
        break;
}
```

```
switch$i>case$1$2>printf("one or two");
```

```
switch(i){
    case 1:
    case 2:
        printf("one or two");
        break;
}
```

2.3.3 Objects

struct:

```
struct!pair>int first, second;
```

```
struct pair {
    int first, second;
};
```

tdstruct (typedef struct):

```
tdstruct!pair>int first, second;
```

```
typedef struct{
     int first, second;
 } pair;
```

enum:

```
enum!Colors>red&green&blue
```

```
enum Colors {
    red,
    green,
    blue
};
```

enum1 (enum in single line):

```
enum1!Colors>red&green&blue
```

```
enum Colors { red, green, blue };
```

2.3.4 Functions

func (function):

```
func#int@five>return 5;
```

```
int five(){
    return 5;
}
```

2.4 Java

2.4.1 Commands

call (function call):

```
call@foo$param1>param2
```

```
foo(param1, param2);
```

call2 (function call in multiple lines):

```
call2@foo$param1>param2
```

```
foo(
  param1,
  param2
);
```

2.4.2 Flow control

for:

```
for#int$i%0%n
```

```
for(int i=0; i<n; i++){

}
```

forr:

```
forr#int$i%n%0
```

```
for(int i=n; i>=0; i--){

}
```

forin:

```
forin#int$i%array
```

```
for(int i: array){

}
```

if:

```
if$true>printf("Always");
```

```
if(true){
    printf("Always");
}
```

else:

```
if$i==2>return 4;<else>return 3;
```

```
if(i==2){
    return 4;
}
else {
    return 3;
}
```

while:

```
while$true>printf("Forever and always");
```

```
while(true){
    printf("Forever and always");
}
```

switch:

```
switch$i>case$1>printf("one");<case$2>printf("two");
```

```
switch(i){
   case 1:
       printf("one");
       break;

   case 2:
       printf("two");
       break;
}
```

```
switch$i>case$1$2>printf("one or two");
```

```
switch(i){
   case 1:
   case 2:
       printf("one or two");
       break;
}
```

2.4.3 Objects

class:

```
class!A:B~C
```

```
class A extends B implements C {

}
```

method:

```
class!A:B>method#int@five
```

```
class A extends B {
    public int five(){

    }
}
```

pmethod (private method):

```
class!A:B>pmethod#int@five
```

```
class A extends B {
    private int five(){

    }
}
```

promethod (protected method):

```
class!A:B>method#int@five
```

```
class A extends B {
    protected int five(){

    }
}
```

amethod (abstract method):

```
class!A:B>amethod#int@five
```

```
class A extends B {
    public int abstract five();
}
```

amethod (abstract method):

```
class!A:B>amethod#int@five
```

```
class A extends B {
    public abstract int five();
}
```

pamethod (private abstract method):

```
class!A:B>pamethod#int@five
```

```
class A extends B {
    private abstract int five();
}
```

method1 (single line method):

```
class!A:B>method1#int@five>return 5;
```

```
class A extends B {
    public int five(){ return 5; }
}
```

constr (constructor):

```
class!A:B>constr#int$i
```

```
class A extends B {
    public A(int i){

    }
}
```

constr1 (single line constructor):

```
class!A:B>constr1
```

```
class A extends B {
    public A(){ }
}
```

pconstr (private constructor):

```
class!A:B>pconstr
```

```
class A extends B {
    private A(){

    }
}
```

enum:

```
enum!Colors>red&green&blue
```

```
enum Colors {
    red,
    green,
    blue
}
```

enum1 (single line enum):

```
enum1!Colors>red&green&blue
```

```
enum Colors { red, green, blue }
```

2.4.4 Templates

class template:

```
class!A^T
```

```
class A<T> {

 }
```

method template:

```
class!A>method#void@nothing^T
```

```
class A {
     public void nothing<T>(){

     }
 }
```

2.4.5 Design Patterns

singleton:

```
class!A>[[singleton]]
```

```
class A {
    private static A instance = null;
    peconstr

    public static A getInstance(){
        if(instance == null){
            instance = new A();
        }

        return instance;
    }
}
```

composite (class and method):

```
class!Composite:Component>[[compositeclass]]&method#void@traverse>[[compositemethod]]
```

```
class Composite extends Component {
    private List<Component> children = ArrayList<Component>();
    public void add(Component item){
        children.add(item);
    }

    public void traverse(){
        for(int i=0; i<children.size(); i++){
            children[i].traverse();
        }
    }
}
```

2.5 Python

2.5.1 Flow control

forin:

```
forin$i%collection>pass
```

```
for i in collection:
    pass
```

for (same as forin but shorter):

```
for$i%collection>pass
```

```
for i in collection:
    pass
```

if:

```
if$i==0>print("i is zero")
```

```
if i==0:
    print("i is zero")
```

else:

```
if$i==0>print("i is zero")<else>print("i is not zero")
```

```
if i==0:
    print("i is zero")
else:
    print("i is not zero")
```

elif:

```
if$i==0>print("i is zero")<elif$i==1>print("i is one")<else>print("i is not zero nor␣
↪one")
```

```
if i==0:
    print("i is zero")
elif i==1:
    print("i is one")
else:
    print("i is not zero nor one")
```

while:

```
while$condition>doSomething()
```

```
while condition:
    doSomething()
```

2.5.2 Objects

func:

```
func@foo$i>pass
```

```
def foo(i):
    pass
```

class:

```
class!A:B>pass
```

```python
class A(B):
    pass
```

method:

```
class!A:B>method@foo$i>pass
```

```python
class A(B):
    def foo(self, i):
        pass
```

smethod (static method):

```
class!A:B>smethod@foo$i>pass
```

```python
class A(B):
    @staticmethod
    def foo(i):
        pass
```

cmethod (class method):

```
class!A:B>cmethod@foo$i>pass
```

```python
class A(B):
    @classmethod
    def foo(cls, i):
        pass
```

constr (constructor):

```
class!A:B>constr$i>pass
```

```python
class A(B):
    def __init__(self, i):
        pass
```

2.5.3 Main

main:

```
[[main]]>pass
```

```python
if __name__ == "__main__":
    pass
```

2.6 JavaScript

2.6.1 Commands

call (function call):

```
call@foo$param1>param2
```

```
foo(param1, param2);
```

call2 (function call in multiple lines):

```
call2@foo$param1>f>3
```

```
foo(
  param1,
  function (){ return 3; }
);
```

var:

```
var$x$3
```

```
var x = 3;
```

```
var$plus>a$i$j>i+j
```

```
var plus = (i, j) => i+j;
```

let:

```
let$x$3
```

```
let x = 3;
```

2.6.2 Flow control

for:

```
for#var$i%0%n
```

```
for(var i=0; i<n; i++){

}
```

forr:

```
forr#var$i%n%0
```

```
for(var i=n; i>=0; i--){

}
```

forin:

```
forin$i%array
```

```
for(let i of array){

}
```

forof:

```
forof$i%array
```

```
for(let i of array){

}
```

if:

```
if$true>console.log("Always");
```

```
if(true){
    console.log("Always");
}
```

else:

```
if$i==2>return 4;<else>return 3;
```

```
if(i==2){
    return 4;
}
else {
    return 3;
}
```

while:

```
while$true>console.log("Forever and always");
```

```
while(true){
    console.log("Forever and always");
}
```

switch:

```
switch$i>case$1>console.log("one");<case$2>console.log("two");
```

```
switch(i){
   case 1:
       console.log("one");
       break;
```

(continues on next page)

(continued from previous page)

```
    case 2:
        console.log("two");
        break;
}
```

```
switch$i>case$1$2>console.log("one or two");
```

```
switch(i){
    case 1:
    case 2:
        console.log("one or two");
        break;
}
```

2.6.3 Object

dict (dictionary):

```
dict>key$item1$1&key$item2$2
```

```
{
  item1: 1,
  item2: 2
}
```

d (dictionary, short):

```
d>k$item1$1&k$item2$2
```

```
{
  "item1": 1,
  "item2": 2
}
```

d (dictionary, nested example):

```
d>k$item1>d>k$nested$1<<&k$item2$2
```

```
{
  "item1": {
    "nested": 1
  },
  "item2": 2
}
```

dict1 (dictionary, single line):

```
dict1>key$"item1"$1&k$item2$2
```

```
{"item1": 1, "item2": 2}
```

Note that `key` is used to build arbitrary key and `k` is used to build a string key.

array:

```
array>item1&item2
```

```
[
  item1,
  item2
]
```

array1 (array, single line):

```
array1>item1&item2
```

```
[item1, item2]
```

2.6.4 Functions

func (function):

```
func@plus$i$j>return i+j;
```

```
function plus(i, j){
  return i+j;
}
```

```
func$i$j>return i+j;
```

```
function (i, j){
  return i+j;
}
```

func1 (function, single line):

```
func1$i$j>return i+j;
```

```
function (i, j){ return i+j; }
```

f (function, single expression):

```
f$i$j>i+j
```

```
function (i, j){ return i+j; }
```

arrow (arrow function):

```
arrow$i$j>return i+j;
```

```
(i, j) => {
  return i+j;
}
```

arrow1 (arrow function, single line):

```
arrow1$i$j>return i+j;
```

```
(i, j) => { return i+j; }
```

a (arrow function, single expression):

```
a$i$j>i+j
```

```
(i, j) => i+j
```

CHAPTER 3

Contributing

3.1 Development process

Code changes should be made in the following way:

- Create a new branch.
- Add code and unit tests. All code should be covered with tests.
- Run tests locally.
- Make a pull request and make sure that travis build succeeds.

3.2 Local copy

Getting local copy:

```
$git clone https://github.com/Ahhhhmed/homotopy.git
```

Running tests (run from the project root):

```
$python -m unittest
```

3.3 Internals

The code is separated in several components:

- *Preprocessor*
- *Parser*
- *Syntax tree*

- *Snippet provider*
- *CodeGenerator*
- *Application frontend*

3.3.1 Preprocessor

Before parsing and creating syntax tree some prepossessing is done to enable some feathers. They are described in following sections.

Decorators

Consider following example:

```
1. for#int$i%n
2. for[[sup]]
```

And following snippet definition:

```
{"name": "sup","language": "C++","snippet": "#int$i%n"}
```

This pattern occurs often and it is useful to name it and use it by name ('sub' stands for 'standard up'). More realistic examples are design pattern implementation (singleton for example). Another advantage of this approach is that multiple decorators can be combined to make a more complex construction.

Preprocessor detects decorators that are defined in double square brackets and expands them using snippet provider.

Cursor marker

After expanding a snippet user should have the cursor at a convenient location. The following rule is followed.

Writing some text after expanding the snippet should have the same effect as writing that text and expanding the snippet afterwords.

To enable this preprocessor appends `&[{cursor_marker}]` to the snippet text so a plugin can put cursor at the marker location.

Usage

class `homotopy.preprocessor.`**`Preprocessor`**(*snippet_provider*)
Prepossess snippet text to enable extra feathers.

`expand_decorators`(*snippet_text*)
Expand decorators to enable concise writing of common patterns.

Parameters **`snippet_text`** – Snippet text

Returns Expanded snippet text

static **`put_cursor_marker`**(*snippet_text*)
Put cursor marker witch should be used by editor plugins for better experience.

Parameters **`snippet_text`** – Snippet text

Returns Snippet text with marker at the end

```
from homotopy.preprocessor import Preprocessor
from homotopy.snippet_provider import SnippetProvider

preprocessor = Preprocessor(SnippetProvider('python', ['folder1', 'folder2']))
snippet = "for"
expanded_snippet = preprocessor.expand_decorators(snippet)
expanded_snippet_with_cursor = preprocessor.put_cursor_marker(expanded_snippet)
```

3.3.2 Parser

Responsible for converting snippet string to a syntax tree instance.

Parameters

Basic functionality is adding parameters to a snippet.

```
snippet#parameter1$parameter2
```

This should become:

```
          $
         / \
        /   \
       #    parameter2
      / \
     /   \
snippet  parameter1
```

Following characters are used to indicate parameters:

```
{'!', '@', '#', '$', '%', ':', '~'}
```

Inside snippets

Snippets can have other snippets insight them (like body of a for loop for example). Snippets are separated by special characters that determine their relations.

- > move to the inside of a snippet
- < move to another snippet on the level above
- `&` move to another snippet on the same level

Example:

```
for>if>if<if&if
```

This should be translated to:

```
         >
        / \
       /   \
      >    if
     / \
```

(continues on next page)

(continued from previous page)

```
    /   \
   >     if
  / \
 /   \
for   >
     / \
    /   \
   if   if
```

Character > is used to donate the inside of a snippet in snippet definitions. This is why all occurrences of < and & are translated to >.

Implicit snippet

Consider following example:

```
for>if<while
```

The `while` is not a part of `for` snippet. Parser creates an implicit `block` snippet at the beginning. `block` snippet is implemented as a list of sub-snippets in new lines.

Escape sequence

Homotopy uses escape sequence to enable operator usage in snippets. Character after `'\'` will always be a part of snippet and not recognised as an operator.

Usage

class homotopy.parser.**Parser**

Class for parsing a string to produce a syntax tree.

static parse(*snippet_text*)

Parsing a string to produce syntax tree.

Parameters **snippet_text** – Text to parse

Returns syntax tree instance corresponding to given text

```
from homotopy.parser import Parser

parser = Parser()
snippet = "for"
syntax_tree = parser.parse(snippet)
```

3.3.3 Syntax tree

Tree structure of a snippet.

class homotopy.syntax_tree.**CompositeSnippet**(*left*, *operation*, *right*)

CompositeSnippet compose two snippets with the operand

accept(*visitor*)

Accepts a visitor

Parameters **visitor** – Visitor instance

Returns Visitor result

class homotopy.syntax_tree.**SimpleSnippet**(*value*)
BasicSnippet can be directly compiled.

accept(*visitor*)
Accepts a visitor

Parameters **visitor** – Visitor instance

Returns Visitor result

class homotopy.syntax_tree.**Snippet**
Base class for snippet syntax tree.

accept(*visitor*)
Accepts a visitor

Parameters **visitor** – Visitor instance

Returns Visitor result

class homotopy.syntax_tree.**SnippetVisitor**
Base class for visitors working on syntax tree.

visit_composite_snippet(*composite_snippet*)
Base visit logic for composite snippets. Process right and left subtrees recursively.

Parameters **composite_snippet** – CompositeSnippet instance

Returns None

visit_simple_snippet(*simple_snippet*)
Base visit logic for simple snippets. Does nothing.

Parameters **simple_snippet** – SimpleSnippet instance

Returns None

Usage

```
from homotopy.syntax_tree import SimpleSnippet, CompositeSnippet

simple_snippet = SimpleSnippet("for")
composite_snippet = CompositeSnippet(simple_snippet, '>', SimpleSnippet('code'))
```

3.3.4 Snippet provider

Provides snippets definitions from a database of snippets.

Snippet definition files

Snippets definition are writen in json files as shown is the following example.

```
[
{"name": "for","language": "C++","snippet": "for(###){$$$}"},
{"name": "if","language": ["C++", "java"],"snippet": "if(###){$$$}"}
]
```

Note that language can be a string or a list of strings. Use `all` for snippets that should always be included. Language can be excluded by prefixing it with ~ (for example `~c++`).

Snippet provider reads all the files containing snippets. It searches all json files contained in the given list of folders files. The list of folders is specified in the `path` variable (similar to `os.path`).

Usage

class `homotopy.snippet_provider.`**`SnippetProvider`**(*language*, *path*)

Class for translating simple snippets into code. Uses a database provided in json files located in the path variable.

`__init__`(*language*, *path*)

Initialize snippet provider instance.

Parameters

- **`language`** – language to compile to
- **`path`** – list of directories to search for json files containing snippets

`__getitem__`(*item*)

Expand single snippet.

Parameters **`item`** – snippet

Returns snippet expansion

```
from homotopy.snippet_provider import SnippetProvider

provider = SnippetProvider("C++", ["folder1", "folder2"])
snippet = "for"
snippetExpansion = provider[snippet]
```

3.3.5 CodeGenerator

CodeGenerator is responsible for turning a syntax tree into output code. It uses *snippet provider* for simple snippets. General rules used by code generator will be discussed in this section.

Simple substitution

Consider the following snippet definition.

```
{"name": "if","language": "C++","snippet": "if(###){$$$}"}
```

Snippet `if#true$i=3;` is expanded in the following way:

- `if` becomes `if(###){$$$}` from the definition.
- `###` gets replaced by `true` to get `if(true){$$$}`.
- `$$$` gets replaced by `i=3;` to get the final output `if(true){i=3;}`.

The value of an operator gets replace by the value provided in the snippet. This is done for every operator to get the final result. Note that there are 3 characters in snippet definition and only one in the snippet. The reason for this is that special characters used by homotopy are also used by other programming languages. For example, `$` is a part of a variable name in php.

Definition expansion

Consider a snippet for a function call. Writing `fun!foo#a#b#c` should return `foo(a,b,c)`. To write a single snippet definition for all functions would mean supporting variable number of parameters.

This is possible using snippet definitions inside snippets.

```
[
{"name": "fun","language": "python","snippet": "!!!({{params}})"},
{"name": "params","language": "python","snippet": "###{{opt_params}}"},
{"name": "opt_params","language": "python","snippet": ", ###{{opt_params}}"}
]
```

Snippet `fun!foo#a#b` is expanded in the following way:

- `fun` becomes `!!!({{params}})` from the definition.
- `!!!` gets replaced by `foo` to get `foo({{params}})`.
- `###` does not exist in `foo({{params}})` so `{{params}}` get expanded to `###{{opt_params}}`.
- `###` gets replaced by `a;` to get `foo(a{{opt_params}})`
- `###` does not exist in `foo(a{{opt_params}})` so `{{opt_params}}` get expanded to `, ###{{opt_params}}`.
- `###` gets replaced by `b;` to get `foo(a, b{{opt_params}})`.
- `{{opt_params}}` gets removed from final result to get `foo(a,b)`.

Expansion is done in case *simple substitution* can't be done. This enables recursive constructs as shown in the example above.

Note that there might be multiple sub-snippets inside a single snippet. In that case the first one containing required operator in its definition gets expanded. Other sub-snippets do not get expanded.

Outer parameters

Accessing outer parameters can be done in the following way:

```
[
{"name": "constructor","language": "java","snippet": "public {{?###}}(){}"}
]
```

The snippet above would create a public empty constructor. `{{?###}}` binds to the same value as `{{?###}}` from the snippet above the current one.

Usage

class `homotopy.code_generator.`**`CodeGenerator`**(*snippet_provider*, *indent_manager*)

Compiler for snippets. Turns syntax tree into text.

generate_code (*snippet*)
Generate code for a snippet. Visit and then perform a clean.

Parameters **snippet** – Snippet

Returns Text of compiled snippet

```
from homotopy.code_generator import CodeGenerator
from homotopy.parser import Parser
from homotopy.snippet_provider import SnippetProvider
from homotopy.util import IndentManager

snippet_provider = SnippetProvider('python', ['folder1', 'folder2'])
indent_manager = IndentManager(snippet_provider)

code_generator = CodeGenerator(snippet_provider, indent_manager)
parser = Parser()

snippet = "for>code"
syntax_tree = parser.parse(snippet)

compiled_snippet = code_generator.generate_code(syntax_tree)
```

3.3.6 Application frontend

Homotopy class

`Homotopy` class scarves as a facade for the whole tool. It should be the eatery point for snippet compilation. It can be configured to include additional paths to user defined snippet libraries.

Example:

```
from homotopy import Homotopy

cpp_snippets = Homotopy("c++")
print(cpp_snippets.generate_code('int n=5;&for#int$i%n>printf("hello");'))
```

outputs:

```
int n=5;
for(int i=0; i<n; i++){
    printf("hello");
}
```

class homotopy.**Homotopy** (*language*)
Facade class for accessing homotopy tool.

add_lib_folder (*path*)
Add path to a snippet library

Parameters **path** – Path

compile (*snippet_text*)
Compile a snippet.

Parameters **snippet_text** – Snippet text

Returns Compiled snippet text

Console application

Homotopy can also be used as a console application. In fact, this is the intended way of using is via editor plugins.

```
C:\dev\homotopy>homotopy -h
usage: homotopy [-h] [-t N] [-c] [-p PATH] language snippet

Compile a snippet.

positional arguments:
  language              Language for the snippet to be compiled to
  snippet               A snippet to be compiled

optional arguments:
  -h, --help            show this help message and exit
  -t N, --tabsize N     Number of spaces in one tab. Tabs remain tabs if
                        absent
  -c, --cursor          Indicate cursor marker in compiled snippet
  -p PATH, --path PATH  Path to snippet library folders separated by ::
```

CHAPTER 4

Plugins

Homotopy is intended to be used by an editor plugin.

Homotopy plugins are intended to be very small since most of work is done in the engine. If you don't see a plugin for your favorite editor, consider *Making a plugin*.

4.1 Plugin list

- Atom

CHAPTER 5

Making a plugin

This page is intended as a guide for creating an editor plugin which uses homotopy to compile snippets.

Atom example can also be used as a reference.

5.1 Calling homotopy

Once installed with `pip`, Homotopy can be used by calling `homotopy` program. It takes several arguments (see *Arguments* section) and prints the resulting snippet to stdout. Warning messages are outputted to stderr.

On a high level, plugin should do the following:

- Collect arguments from editor.
- Call `homotopy`.
- Replace snippet text with the result of `homotopy` call.

5.2 Arguments

There are several arguments that can be passed to `homotopy`.

Only two are required, **language** and **snippet**, but all of them should be provided by the plugin for full experience.

5.2.1 Language

This is the first positional argument. It represents the name of the language the snippet should be compiled to.

5.2.2 Snippet

This is the text of the snippet.

```
homotopy c++ if$true
```

```
if(true){

}
```

5.2.3 Cursor (-c)

When flag `-c` is present `homotopy` will put text `[{cursor_marker}]` at the place where the cursor should be placed after the expansion of the snippet.

```
homotopy -c c++ "if$true>"
```

```
if(true){
    [{cursor_marker}]
}
```

5.2.4 Path (-p)

This optional argument that indicates the locations of custom snippets. Value is *string* containing paths to snippet locations separated by `::`.

```
homotopy -p folder1::folder2 c++ if$true
```

```
if(true){

}
```

5.2.5 Tab size (-t)

Homotopy uses tabs to indent code. If the editor is configured to use soft tabs (spaces instead of tabs), use this option to indicate the size of one tab. All tabs in the result will be converted to the given number of spaces.

If this argument is not present, code with tabs will be returned.

```
homotopy -t 4 c++ if$true
```

```
if(true){

}
```

5.3 Cursor position

To make the user flow more natural, homotopy can mark the location that the cursor should be in after the expansion of a snippet.

There are few things to keep in mind when implementing this feature.

- Look for `[{cursor_marker}]` and replace it with cursor. Keep in mind that text *[{cursor_marker}]* should not be displayed.
- If, for any reason, `[{cursor_marker}]` is not present, place the cursor in the line after the expanded snippet text.
- Make sure that the operation looks atomic with respect to undo/redo logic.

5.4 Settings

Following user settings should be supported.

5.4.1 Homotopy location

If `homotopy` is not in the user *PATH*, he/she should be able to configure the path to it.

5.4.2 User library

Locations to user library folders should be configurable and passed to the engine with `-p` argument.

5.5 Error handling

- If an error occurs (wrong path to `homotopy` for example) when calling `homotopy`, an error message should be displayed to the user.
- If any waring message is present (i.e. stderr is not empty), it should be displayed to the user (for example, so he/she can be informed when the snippet database is corrupt).

Python Module Index

Index