ソースコードの構文木表現による構造類似性を用いた自動関数生成方式

北 椋太　岡田 龍太郎　峰松 彩子　中西 崇文

武蔵野大学データサイエンス学部　TransMedia Tech Lab

目次

研究概要

- 研究背景
- 研究目的

提案方式

- システムの全体像
- 構文解析 -> 部分木抽出 -> 類似度計量 -> 構造同一化 -> 関数生成

実験

1. 最適な深さの検討
2. 一致するコードクローンへの有効性
3. 類似するコードクローンへの有効性

おわりに

- まとめ
- 今後の課題

研究概要

研究概要

- 研究背景
- 研究目的

提案方式

- システムの全体像
- 構文解析 -> 部分木抽出 -> 類似度計量 -> 構造同一化 -> 関数生成

実験

1. 最適な深さの検討
2. 一致するコードクローンへの有効性
3. 類似するコードクローンへの有効性

おわりに

- まとめ
- 今後の課題

研究背景 : ソフトウェア開発の現状

ソフトウェアの利用分野の拡大

- システムの不具合が社会的に問題となることが増えている

ソフトウェア開発ライフサイクル

- 保守作業のコストが占める割合が非常に高い

-> **保守性の向上**が課題となっている

研究背景 ： 保守作業の抱える問題

保守作業の効率化

- ソースコードの理解や，修正を容易にすることが重要

  -> これらを困難にする要因 ： **コードクローン**

  コードクローン ... ソースコード中の一部分(=コード片)のうち，
  類似または一致するコード片が他に存在するもの

```
if a > b:
    b = b+1
    a = 1
```

オリジナルの
コード片

```
if a > b:
    b = b + 1
    a = 1
```

空白を含む
コードクローン

```
if i > j:
    j = j+1
    i = 1
```

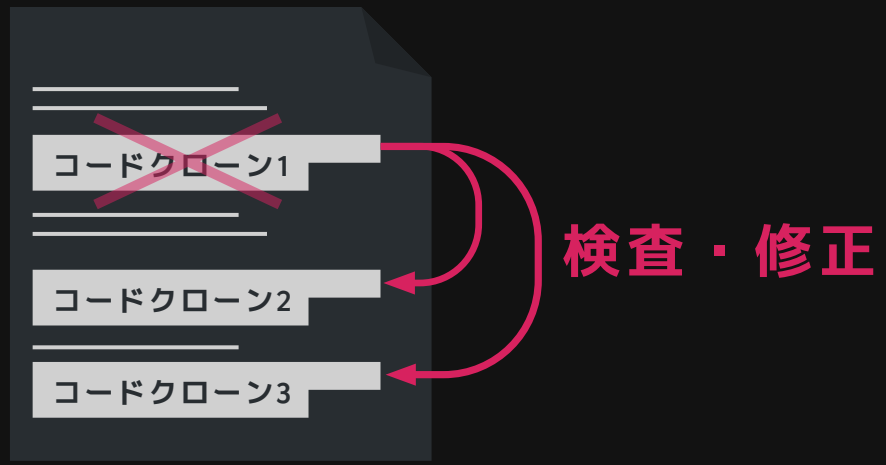
識別子の違う
コードクローン

```
if a > b:
    b = b+1
    a = a+1
```

構造の違う
コードクローン

コードクローンのデメリット

- あるコード片に欠陥が発見されたとき，
  そのコード片に対応するすべてのコードクローンを検査する必要がある

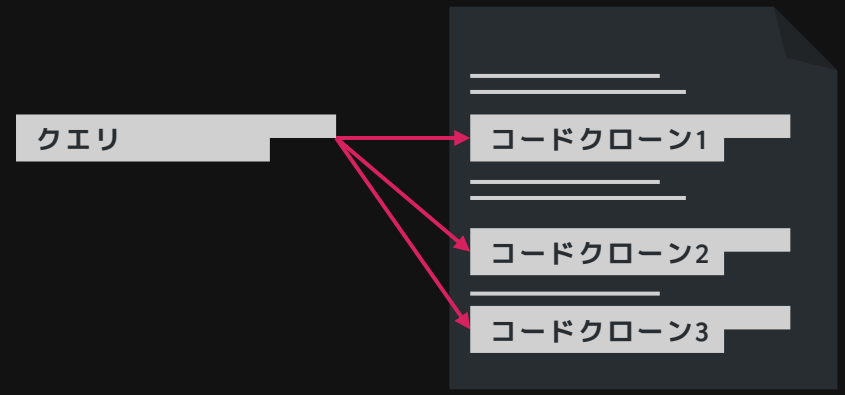


-> **コードクローンの検出・解消**が課題となっている

研究背景 ： コードクローンの検出手法

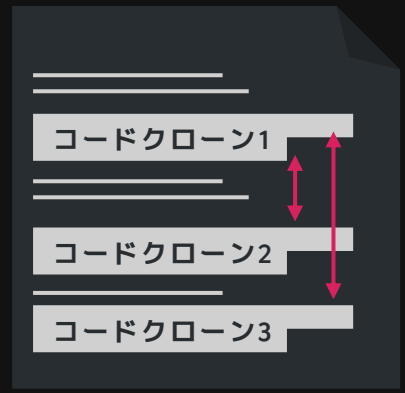
キーワード検索

- 欠陥を含むコード片から
  抽出した**キーワード**を用いて検索する



コードクローン検出ツール

- **トークン列**や**グラフ**を用いて
  ソースコード内から同一パターンを検出する

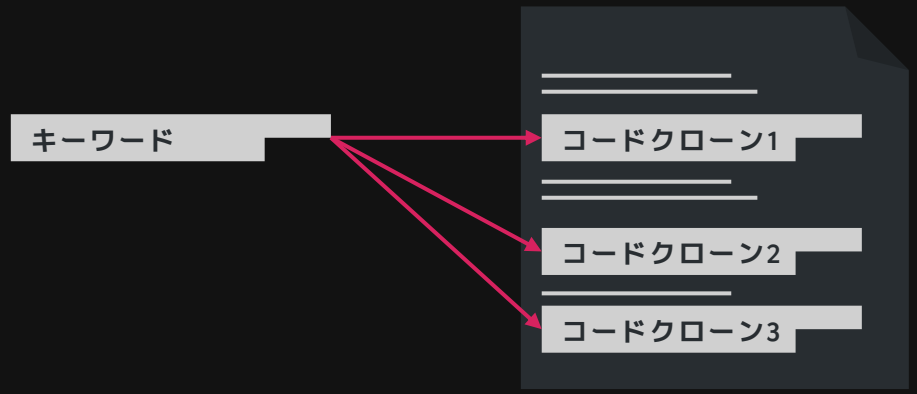


研究背景 ： コードクローン検出の既存手法

キーワード検索

- キーワードと**完全一致するコード片**のみを出力する
- 空白・コメントに影響を受ける
- 例 ： grep[1] など

-> 適切なキーワードの選定には，ソースコードを理解している必要がある



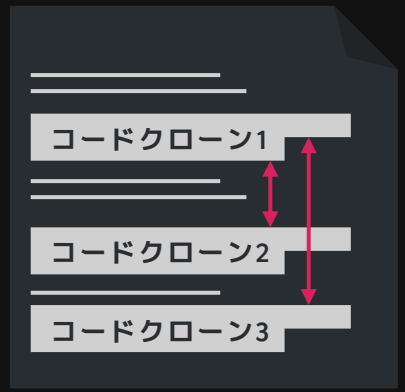
[1]http://www.gnu.org/software/grep/

研究背景 ： コードクローン検出の既存手法

コードクローン検出ツール

- 入力したコード片のコードクローンの**位置情報**や**類似度**などを出力する
- 識別子の違いに対応している
- 例 ： CCFinder[2] など

-> 構造が完全一致でないコードクローンへ対応できない

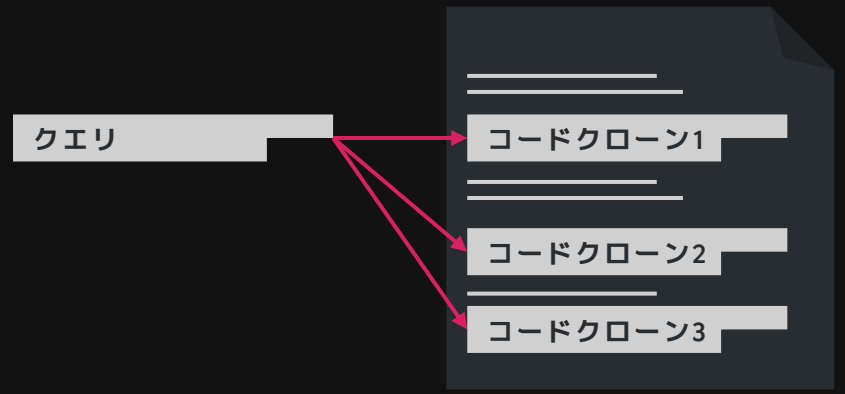


[2] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, 28(7), pp. 654-670, 2002.

研究背景 ： コードクローン検出の既存手法

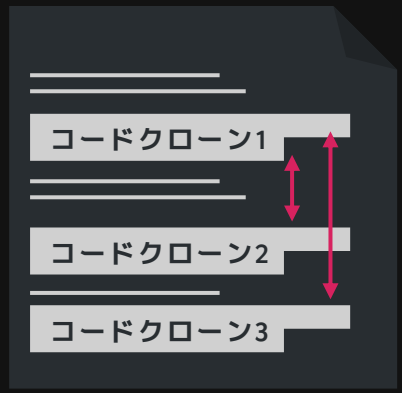
キーワード検索

- 欠陥を含むコード片から
  抽出したキーワードを用いて検索する



コードクローン検出ツール

- トークン列やグラフを用いて
  ソースコード内から同一パターンを検出する



-> 検出された**コードクローン**に対して，手作業で修正を行う必要がある

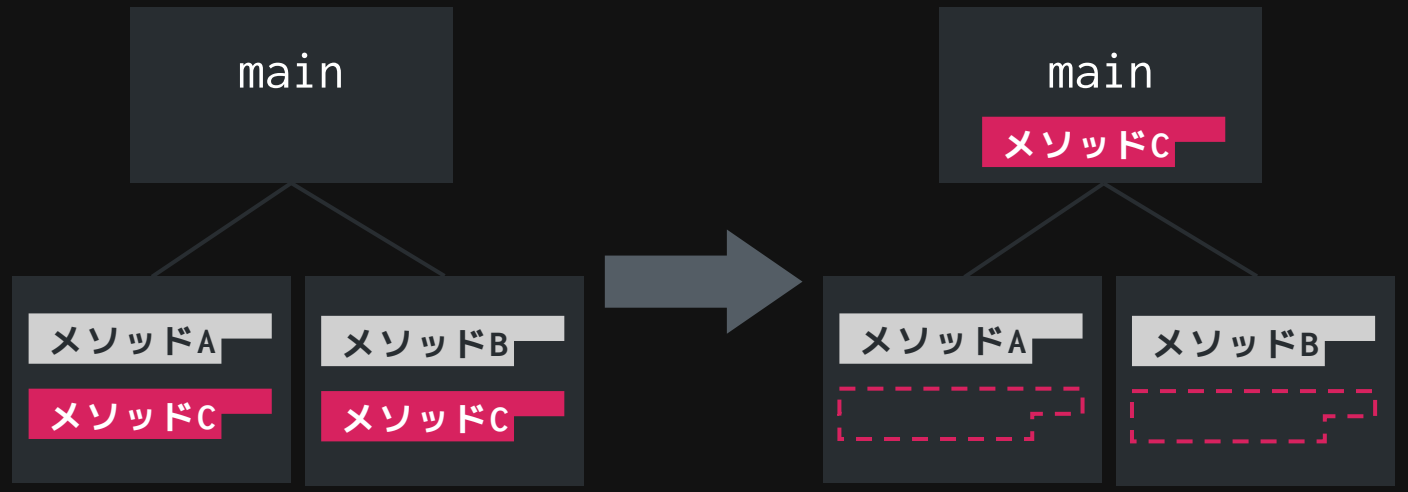
Department of Data Science, Musashino University.

研究背景 : リファクタリングの既存手法

関数(メソッド)の引き上げ

- 複数のオブジェクトに共通する関数(メソッド)を親クラスへ引き上げる

-> 関数(メソッド) 単位でしか適用することができない



Department of Data Science, Musashino University.

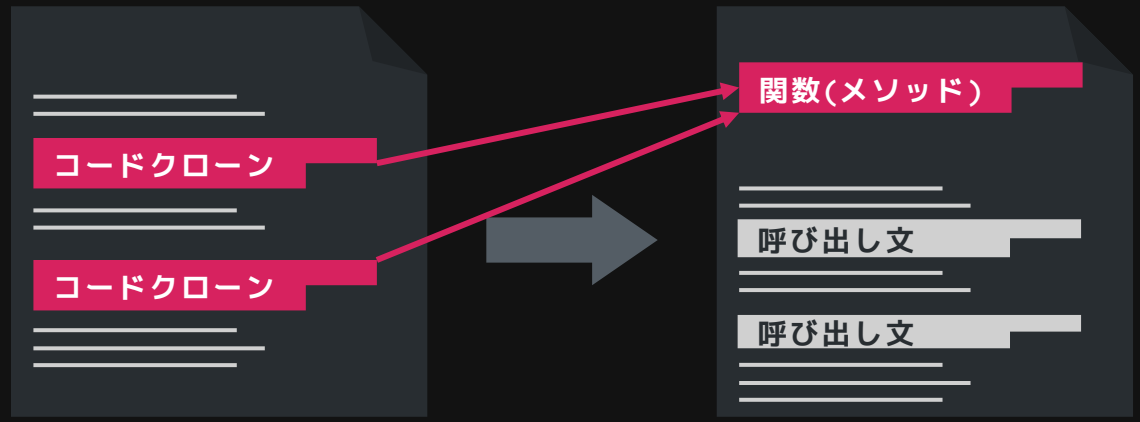
[3] 肥後芳樹，吉田則裕，“コードクローンを対象としたリファクタリング”，コンピュータソフトウェア，28(4)，pp. 43-56，2011.

研究背景 : リファクタリングの既存手法

関数(メソッド)の抽出

- 共通する複数の文を関数(メソッド)として抽出する
- 変数やリテラルを引数として渡すことで，
  識別子の異なるコードクローンに対応可能

-> 識別子レベルの差異しか認められない



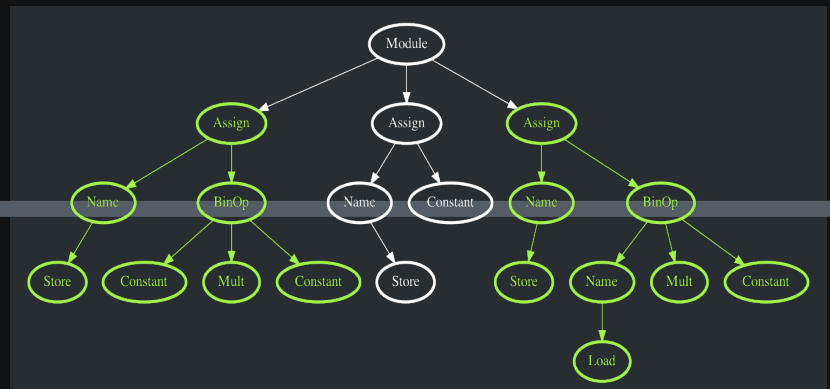
[3] 肥後芳樹，吉田則裕，“コードクローンを対象としたリファクタリング”，コンピュータソフトウェア，28(4)，pp. 43-56，2011.

研究目的

- ソースコードの構造から**コードクローン**を検出する
- 完全一致だけでなく**構造の異なるコードクローン**にも適用する
- コードクローンを**関数**に置き換えることで，保守コストの低減させる

```
a = 5 * 10
b = 3
c = a * 10
```

入力



構文木

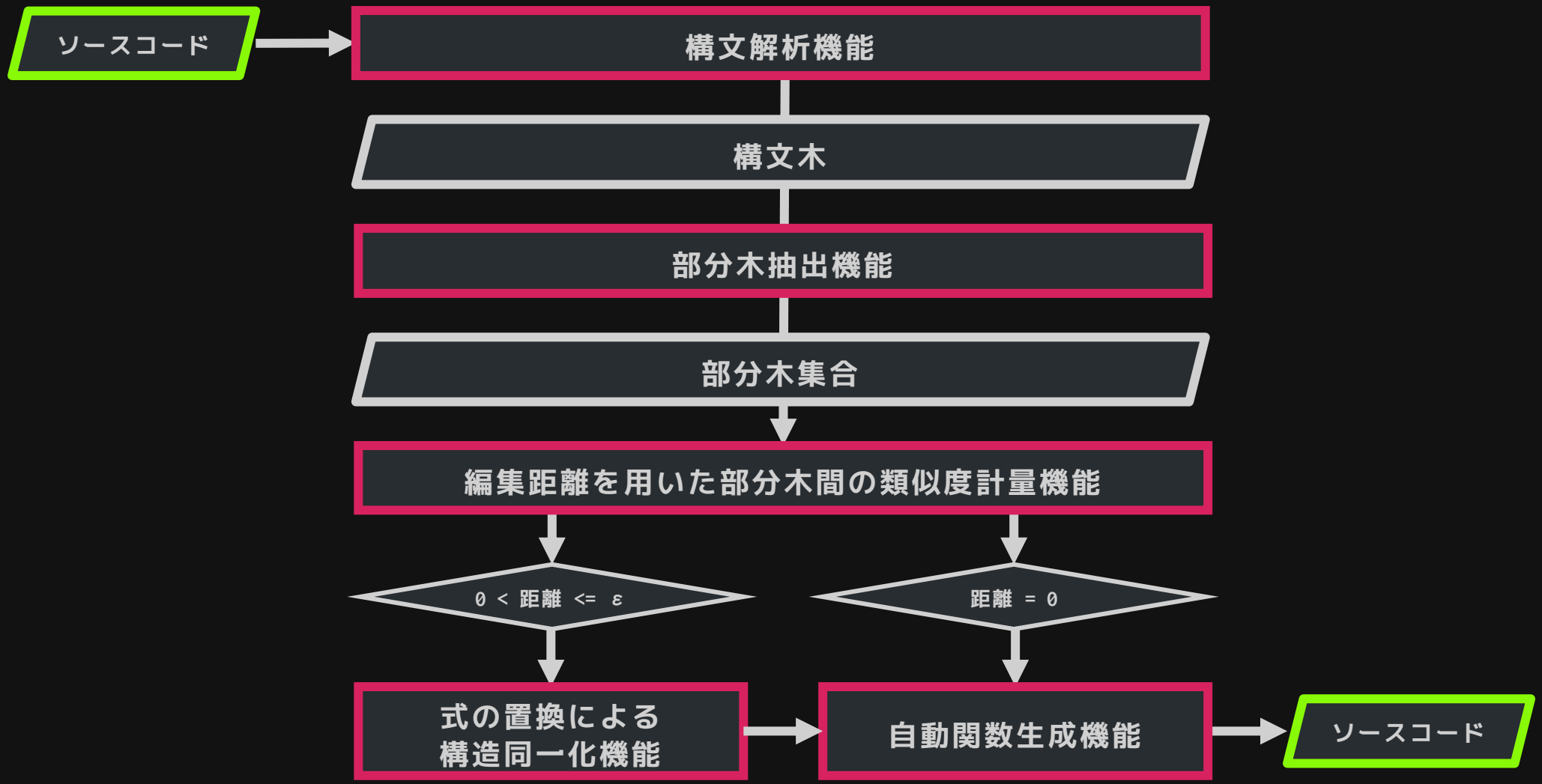
```
def function1(var2):
  var1 = var2 * 10
  return var1

a = function1(5)
b = 3
c = function1(a)
```

出力

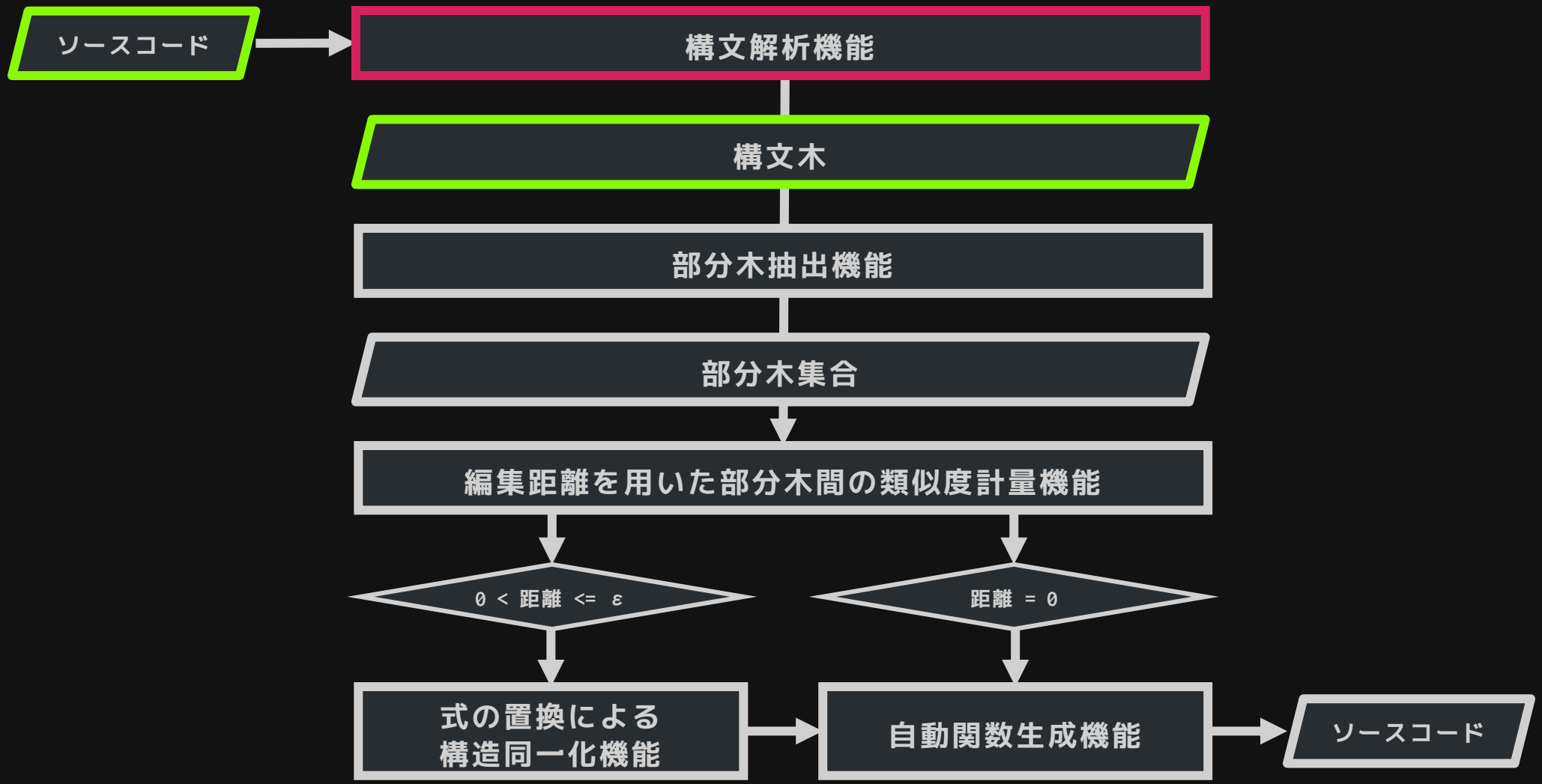
提案方式

研究概要	- 研究背景 - 研究目的
提案方式	- システムの全体像 - 構文解析 -> 部分木抽出 -> 類似度計量 -> 構造同一化 -> 関数生成
実験	1. 最適な深さの検討 2. 一致するコードクローンへの有効性 3. 類似するコードクローンへの有効性
おわりに	- まとめ - 今後の課題

システムの全体像



構文解析機能



構文解析機能

字句解析

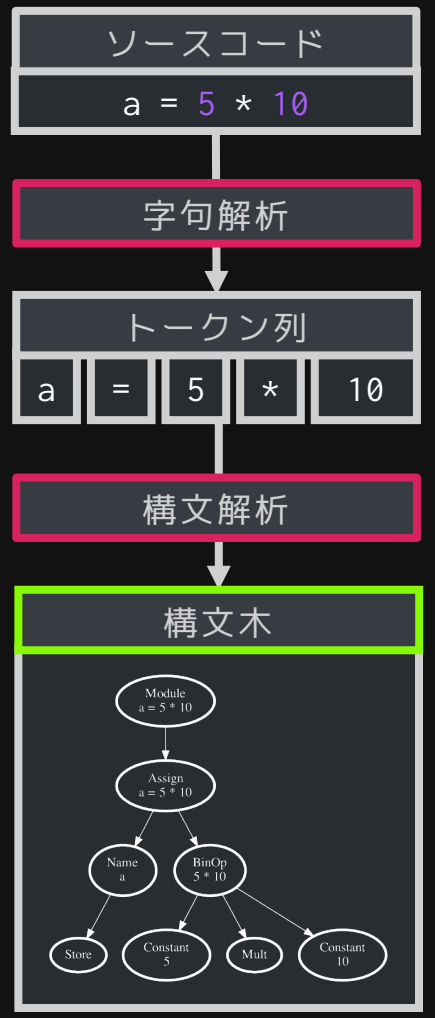
- トークン(最小単位)へ分割する処理のこと

構文解析

- トークン間の関係性を明確にする処理のこと

構文木

- 構文解析で得られた結果を木構造で表現したもの
- 動作に関係ある情報のみの構文木を**抽象構文木**と呼ぶ

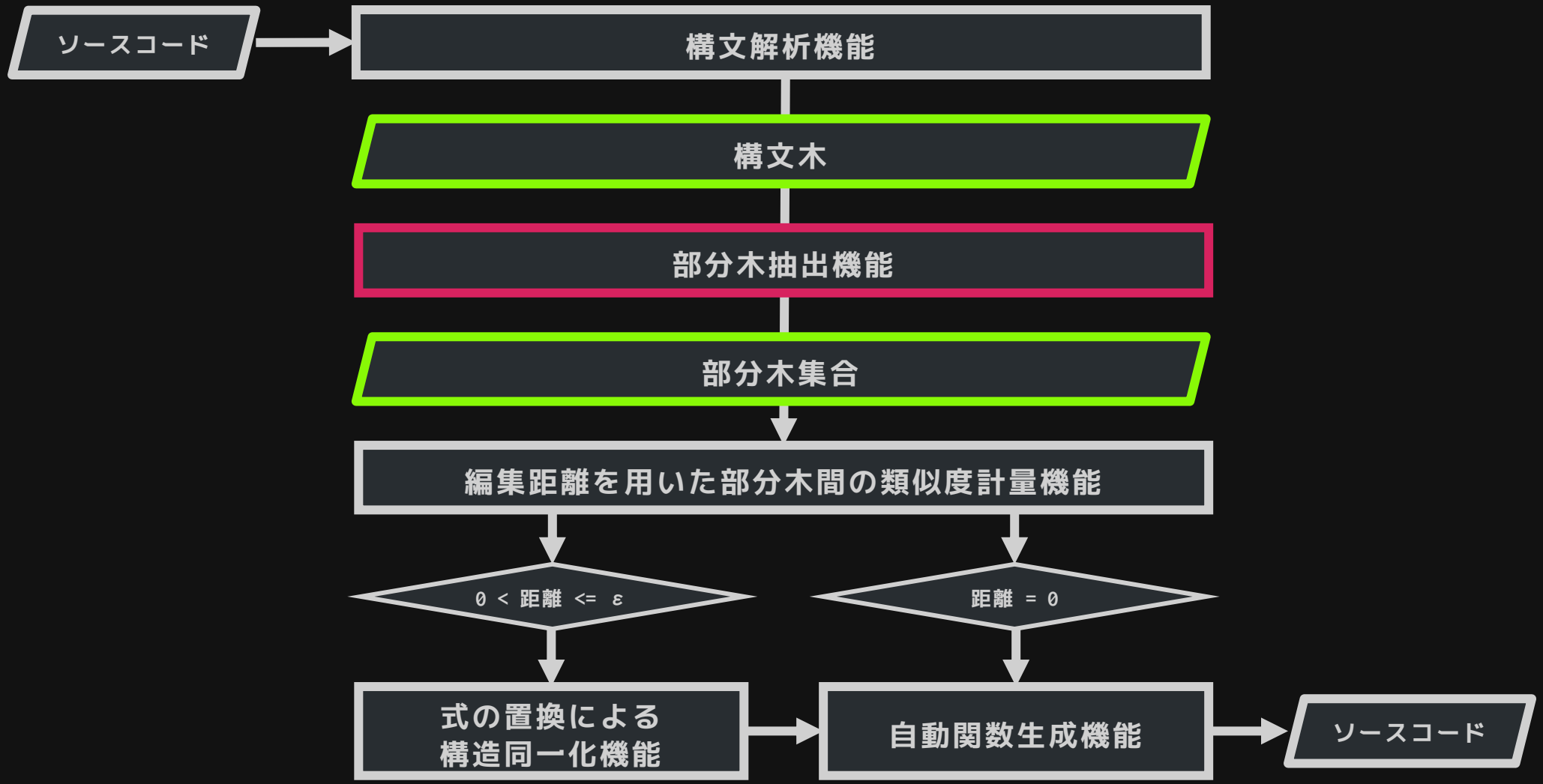


Department of Data Science, Musashino University.

ノードの保有する情報

- **編集距離を用いた部分木間の類似度計量機能**
  - 抽象文法名(Name, Constant, Call ...)
  - 親・子ノード
- **自動関数生成機能**
  - 行番号，列番号
  - テキスト(hoge, for, + ...)
  - ノード属性(根，内部，葉)
  - 構文木の深さ

部分木抽出機能



部分木抽出機能

- 小さなコード片を関数に置き換えても保守性の向上に繋がらない
- 処理の一部分(行の途中)だけを関数に置き換えることはできない

抽出条件

- **最大深さ**が任意に定める**k** 以上の部分木
- 根ノードが**文**となる部分木

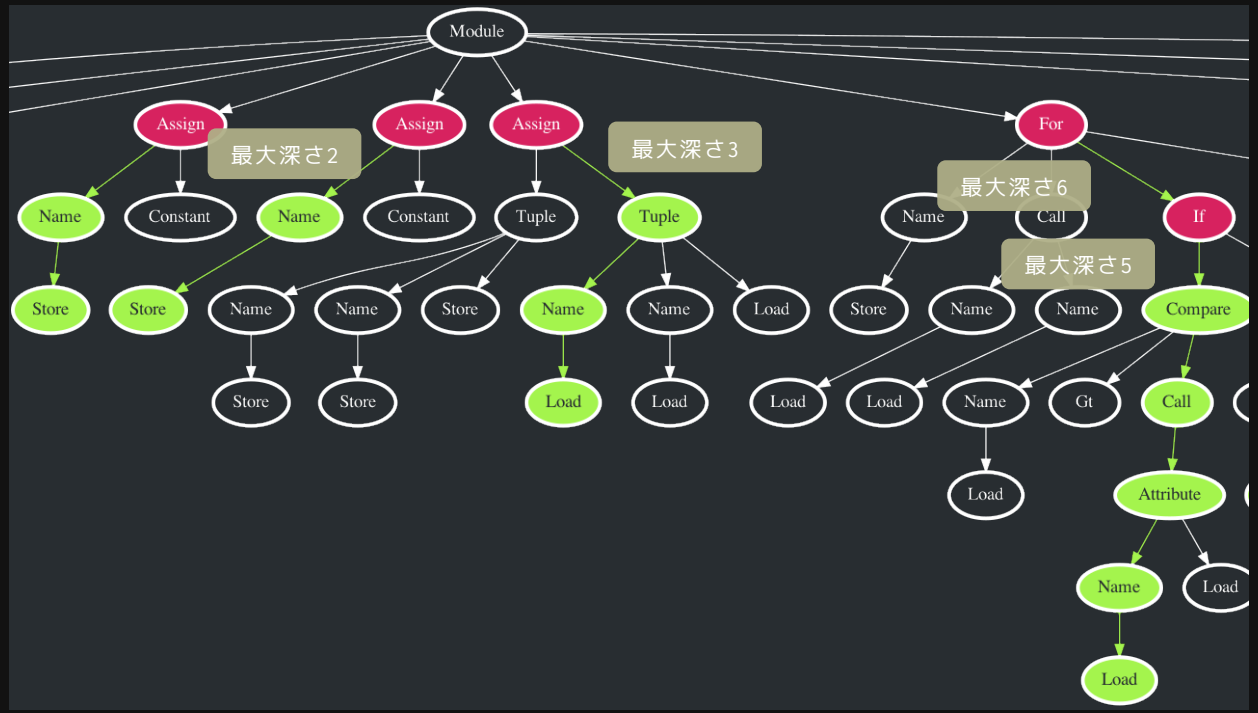
  文 ... トークンを組み合わせることで手続き，命令，宣言などを行う構成単位

  Assign, For, If, While ... など

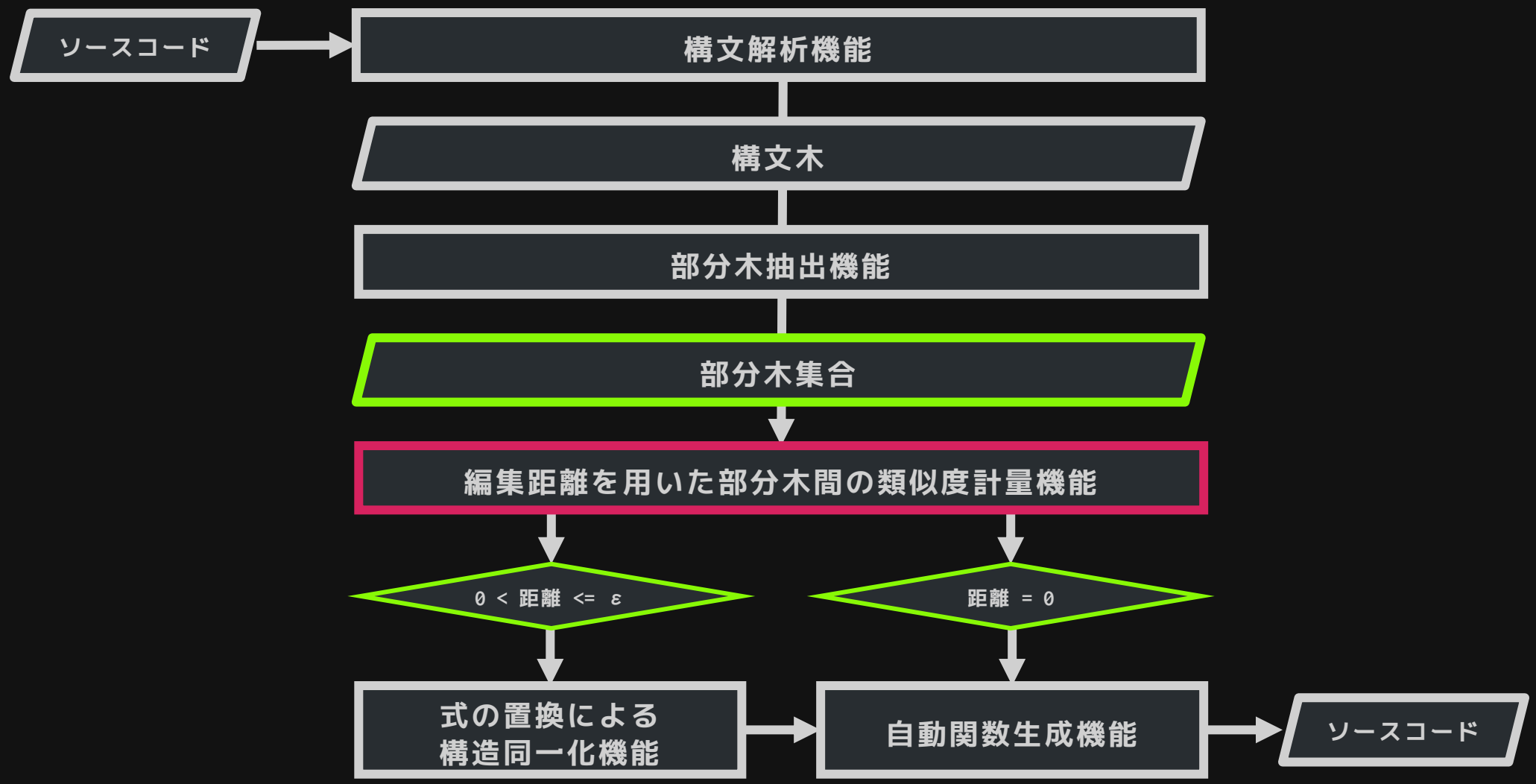
-> 抽出する**部分木の深さ**をパラメータとして任意の値で設定することで，**検出するスコープを変更**することができる

抽出条件

- **最大深さ**が任意に定めるk 以上の部分木
- 根ノードが**文**となる部分木



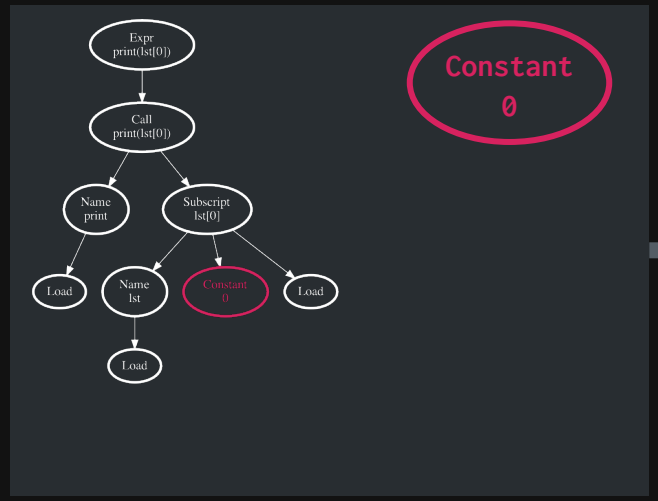
編集距離を用いた部分木間の類似度計量機能



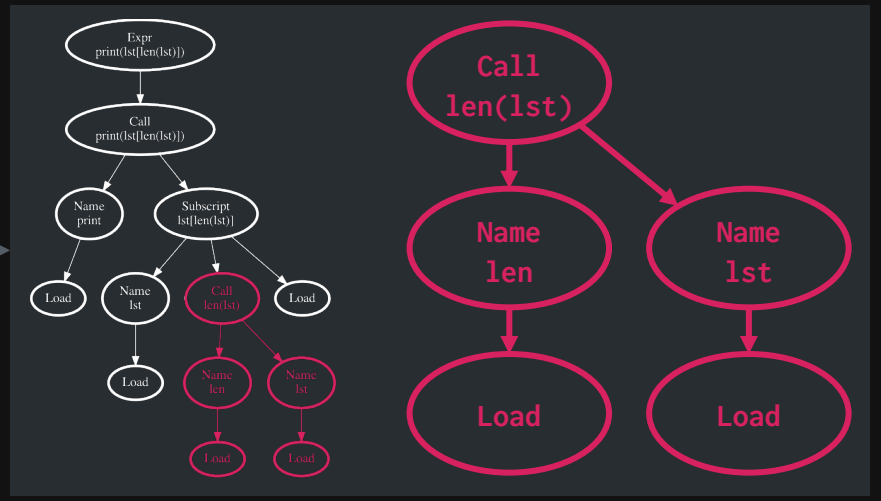
編集距離による類似度計量

- 根・ラベル付きの順序木を対象に，
  必要なノードの**挿入**・**削除**・**更新**の最小回数を**編集距離**として定義する
- ZhangとShashaによる$O(n^4)$時間アルゴリズム[4]を使用

`print(lst[0])`



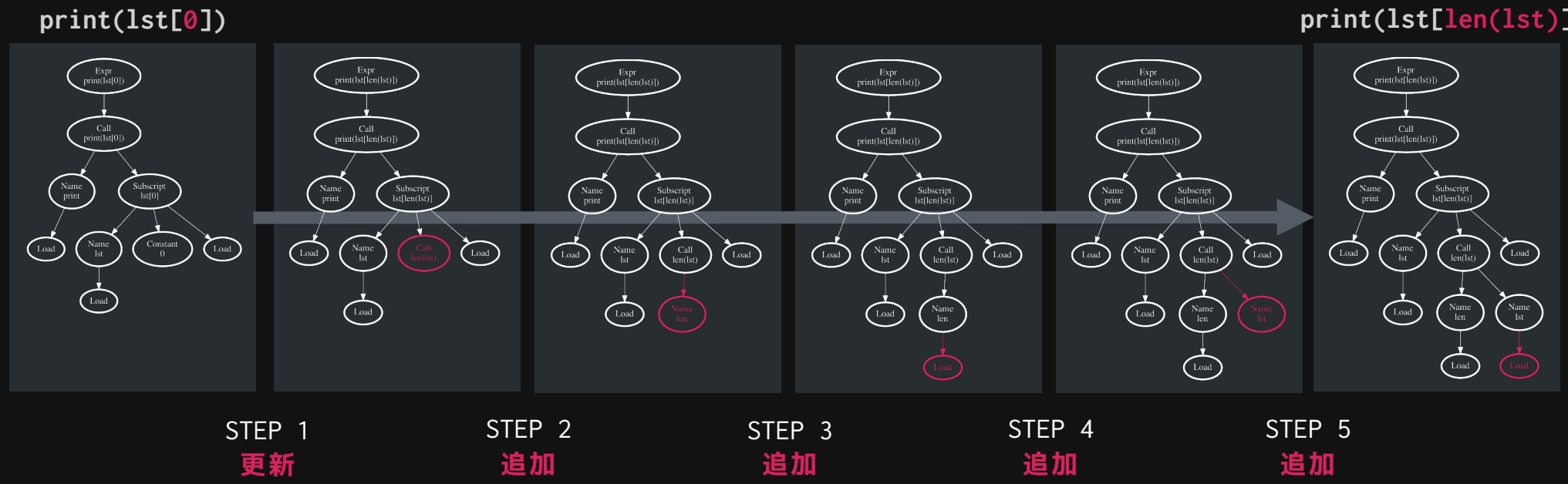
`print(lst[len(lst)]`



[4] Kaizhong zhang, Dennis Shasha, "SIMPLE FAST ALGORITHMS FOR THE EDITING DISTANCE BETWEEN TREES AND RELATED PROBLEMS", In SIAM Journal of computing, 18(6), pp. 1245-1262, 1989.

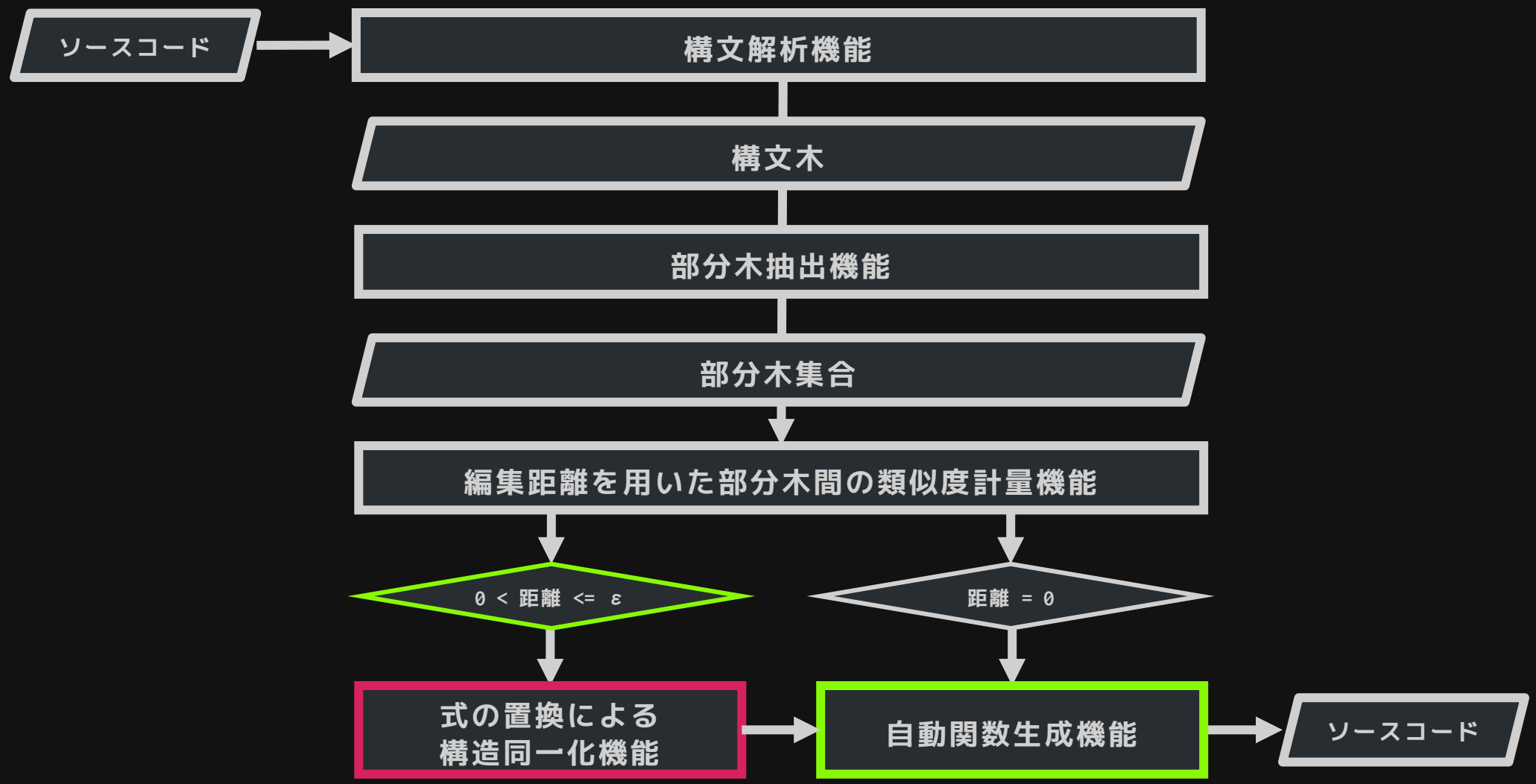
編集距離による類似度計量

- 根・ラベル付きの順序木を対象に，
  必要なノードの**挿入**・**削除**・**更新**の最小回数を**編集距離**として定義する
- ZhangとShashaによる$O(n^4)$時間アルゴリズム[4]を使用



[2] Kaizhong zhang, Dennis Shasha, "SIMPLE FAST ALGORITHMS FOR THE EDITING DISTANCE BETWEEN TREES AND RELATED PROBLEMS", In SIAM Journal of computing, 18(6), pp. 1245-1262, 1989.

式の置換による構造同一化機能



式の置換による構造同一化機能

- 自動関数生成は完全一致する部分木にしか適用できない．
- 本手法を**類似する部分木**を**同一構造に変形**することで適用範囲を拡張する．
- 対象 ： 編集距離が**0より大きい**かつ任意に定める**閾値 ε 以下**の部分木

置換対象 ： 距離計算時に編集操作を行なったノード群の最上位のノード

条件 ： 1. 対象ノードが**式**である

式 ... 評価後に値に変換されるもの．変数やリテラル，演算子式など．

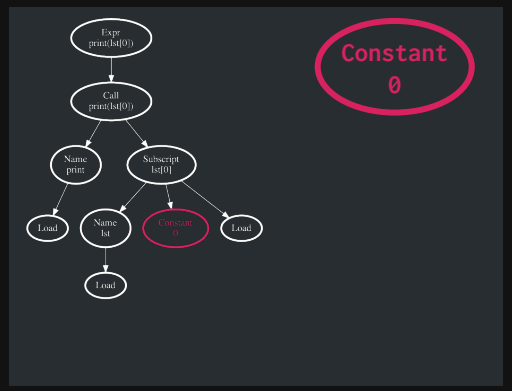
2. 対象ノードの兄弟ノードの個数が編集操作の前後で変化していない

操作 ： 抽象文法名を**expr**とする新たなノードを用意し，置換する

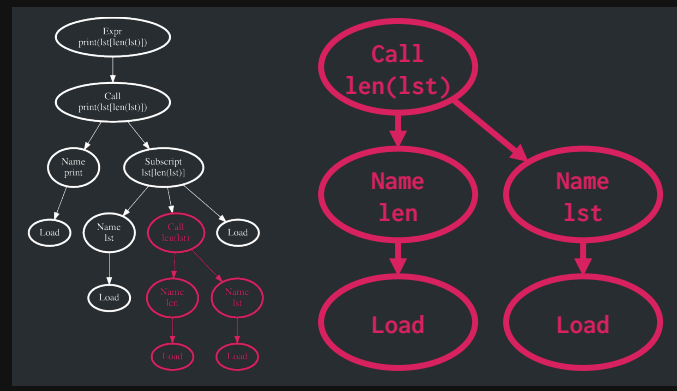
式の置換による構造同一化機能

置換前

print(lst[0])

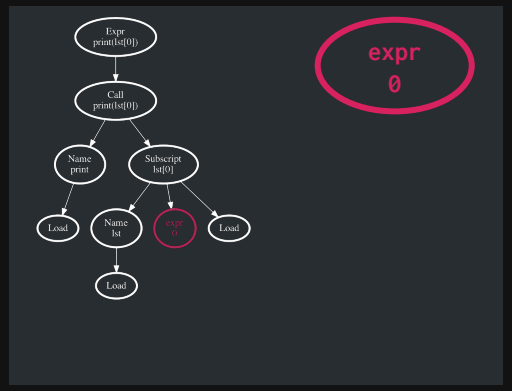


print(lst[len(lst)]

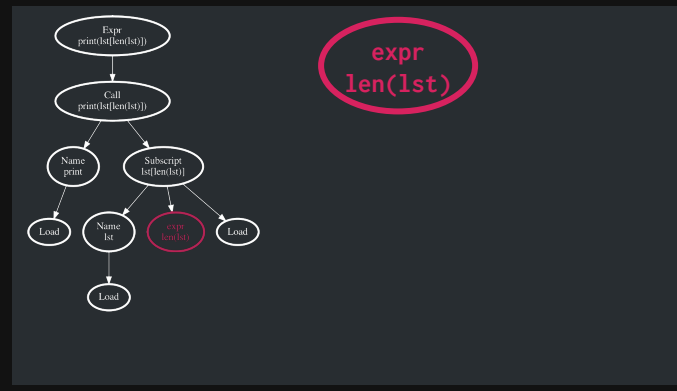


置換後

print(lst[0])

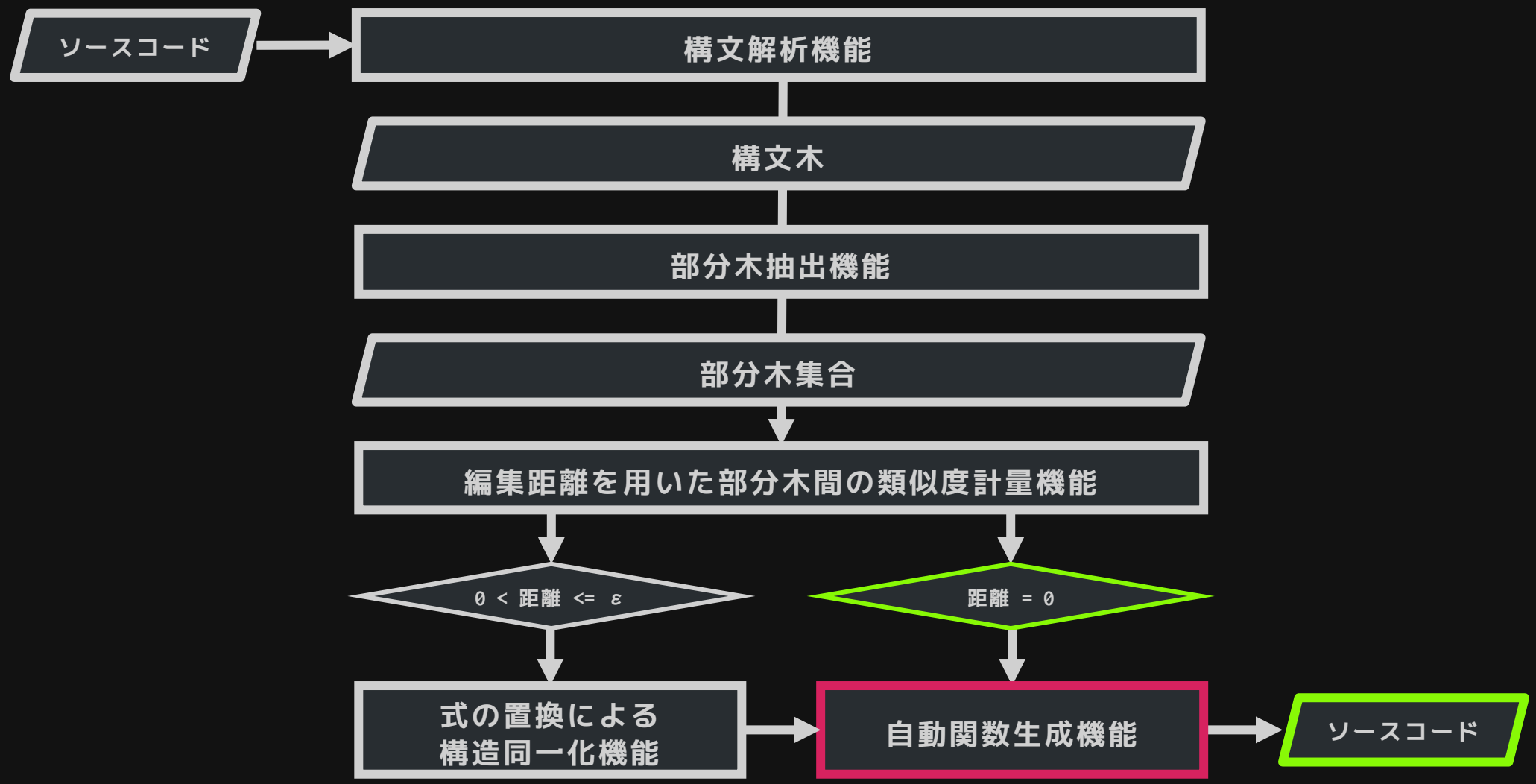


print(lst[len(lst)]



Department of Data Science, Musashino University.

自動関数生成機能



自動関数生成機能

1. ソースコードの先頭に
   関数部の**テンプレート**を用意する

```
def function1( 引数 ):
    処理文
    return 戻り値
```

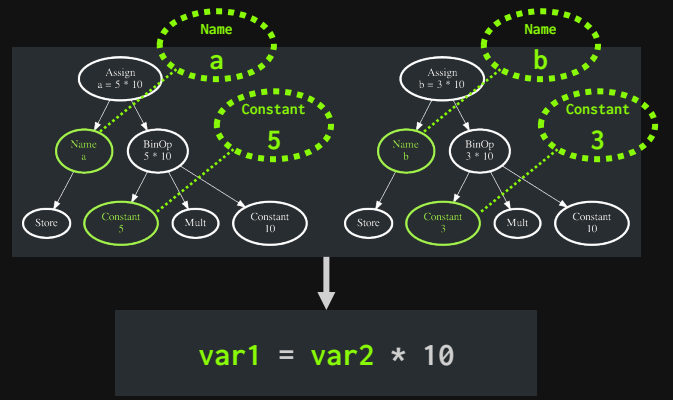
2. ソースコードから**一致する部分木**に
   該当するノード片を抽出する

```
a = 5 * 10
b = 3 * 10
print(a, b)
```

→

```
a = 5 * 10
b = 3 * 10
```

3. 全ての**変数**および，
   2つの部分木間で**識別子が異なる式**を
   関数内の変数として置換する




```
var1 = var2 * 10
```

自動関数生成機能

4. 3で置換した変数のうち，
   未割り当ての変数以外を**引数**，
   被処理変数を**戻り値**とする

```
def function1( var2 ):
    処理文
    return var1
```

6. 3で置換した後の文を
   関数内の**処理文**とする

```
def function1( var2 ):
    var1 = var2 * 10
    return var1
```

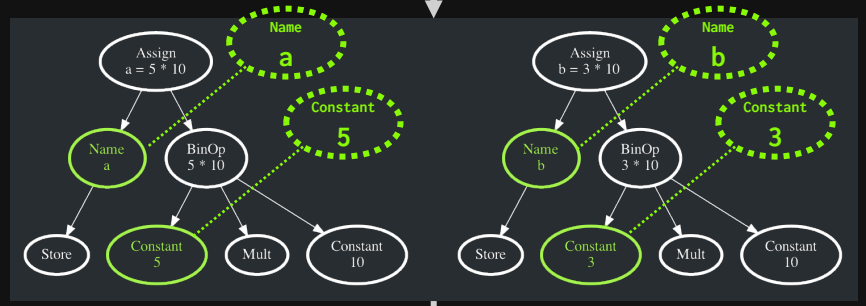
7. 元のコード片を削除し、
   関数の**呼び出し文**に置換する

```
a = function1(5)
b = function1(3)
```

自動関数生成機能


```
def function1( 引数 ):
    処理文
    return 戻り値
```

```
a = 5 * 10
b = 3 * 10

print(a, b)
```




```
var1 = var2 * 10
```

```
戻り値 = function(引数)
戻り値 = function(引数)
```

```
def function1(var2):
    var1 = var2 * 10
    return var1

a = function1(5)
b = function1(3)

print(a, b)
```

実験

研究概要	- 研究背景 - 研究目的
提案方式	- システムの全体像 - 構文解析 -> 部分木抽出 -> 類似度計量 -> 構造同一化 -> 関数生成
実験	1. 最適な深さの検討 2. 一致するコードクローンへの有効性 3. 類似するコードクローンへの有効性
おわりに	- まとめ - 今後の課題

実験1

- 部分木抽出時の最適な深さの検討

実験2

- 一致するコードクローンに対する自動関数生成機能の有効性の検証

実験3

- 類似するコードクローンに対する自動関数生成機能の有効性の検証

実験１ ： 部分木抽出時の最適な深さの検討［目的］

実験環境

- 使用言語 ： Python
- 構文木抽出ライブラリ ： ast
- 入力ソースコード ： 右記

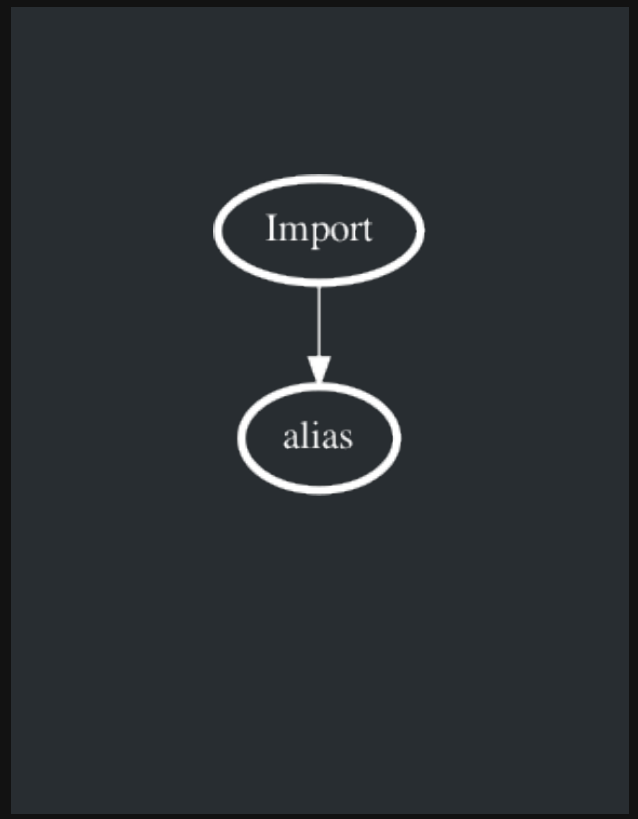
実験目的

- 関数生成をするにあたって，
  最適な部分木の最大深さについて
  検討する．

```python
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3
for _ in range(cannon1):
    if rate1 > random.random():
        cannon2_ -= 1
    if cannon2_ == 0:
        break
for _ in range(cannon2):
    if rate2 > random.random():
        cannon1_ -= 1
    if cannon1_ == 0:
        break
cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

実験１ ： 部分木抽出時の最適な深さの検討［結果］

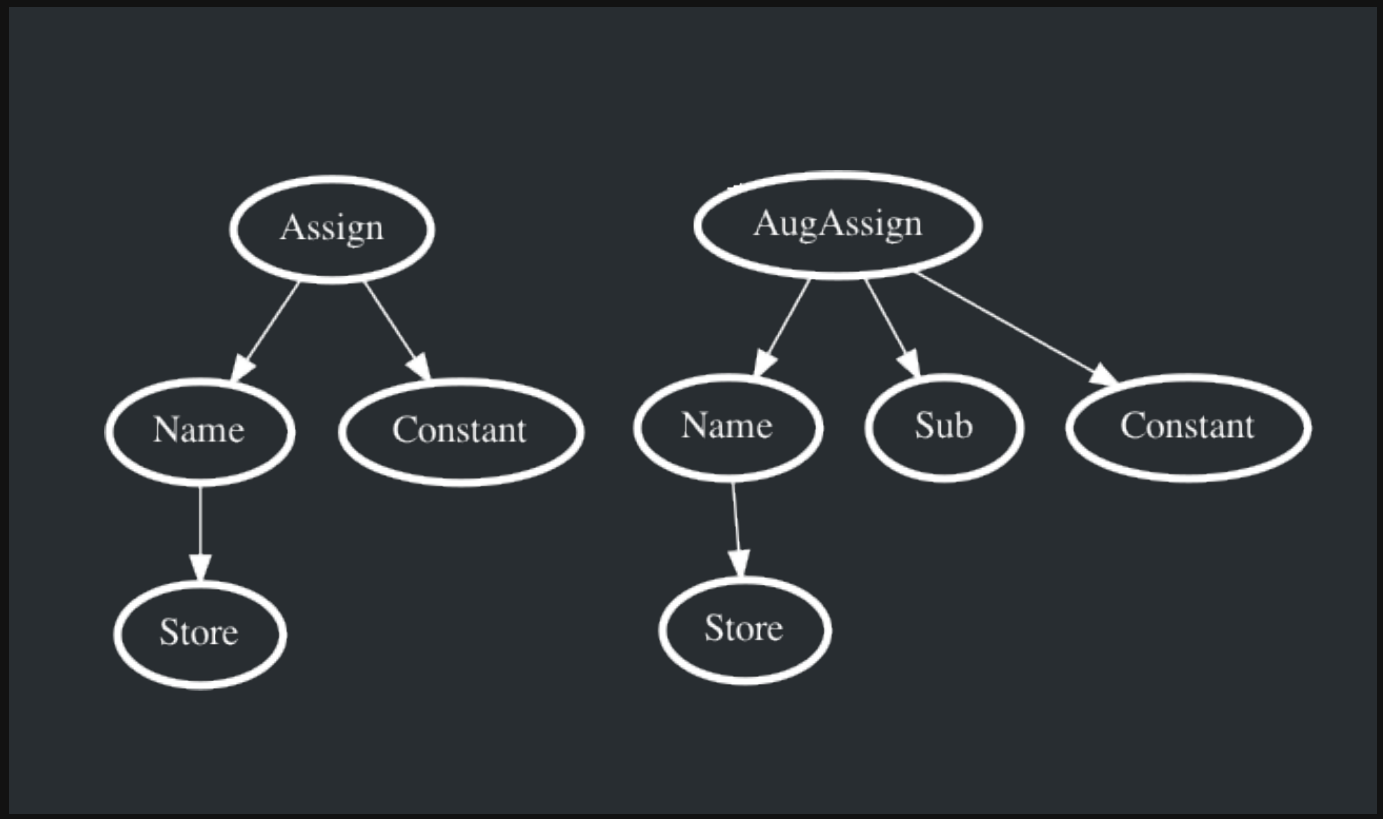

```
import random
```



最大深さ 1

```
cannon1 = 10
```

```
cannon1_ -= 1
```



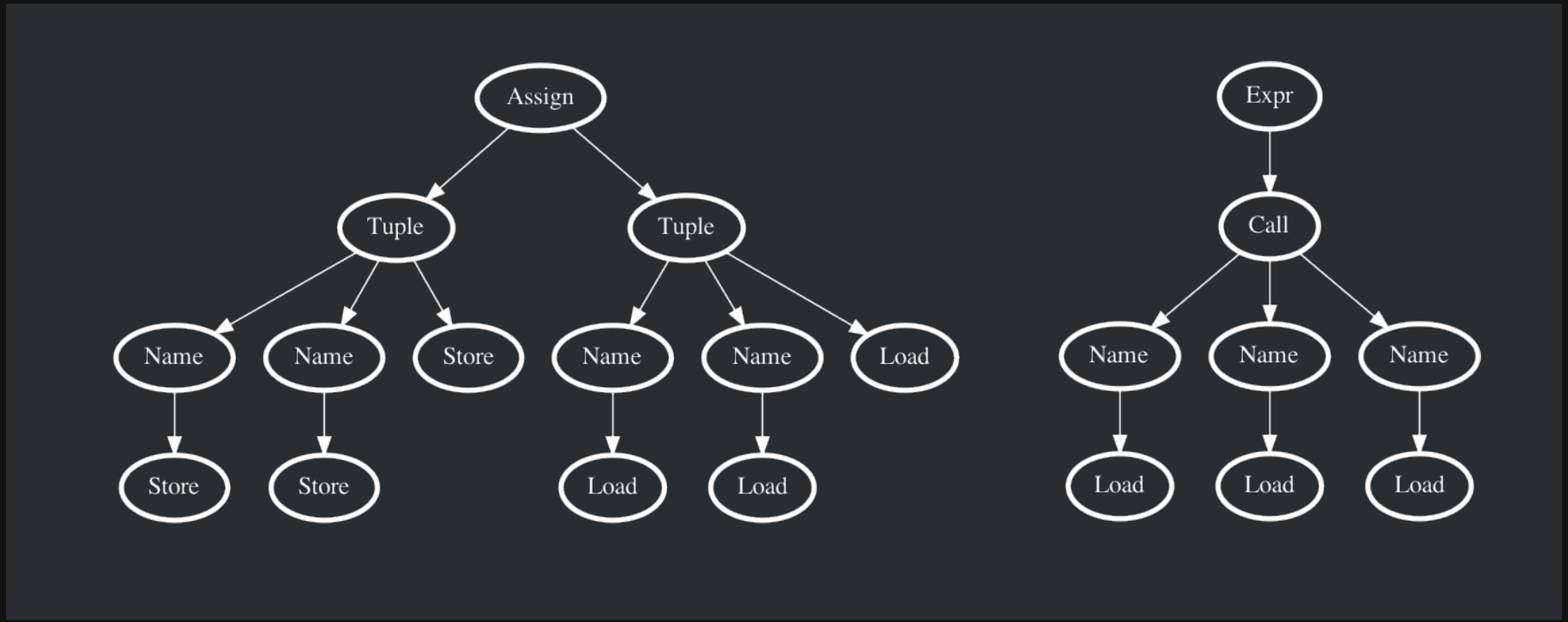
最大深さ 2

Department of Data Science, Musashino University.

実験１ ： 部分木抽出時の最適な深さの検討［結果］


```
cannon1_, cannon2_ = cannon1, cannon2
```

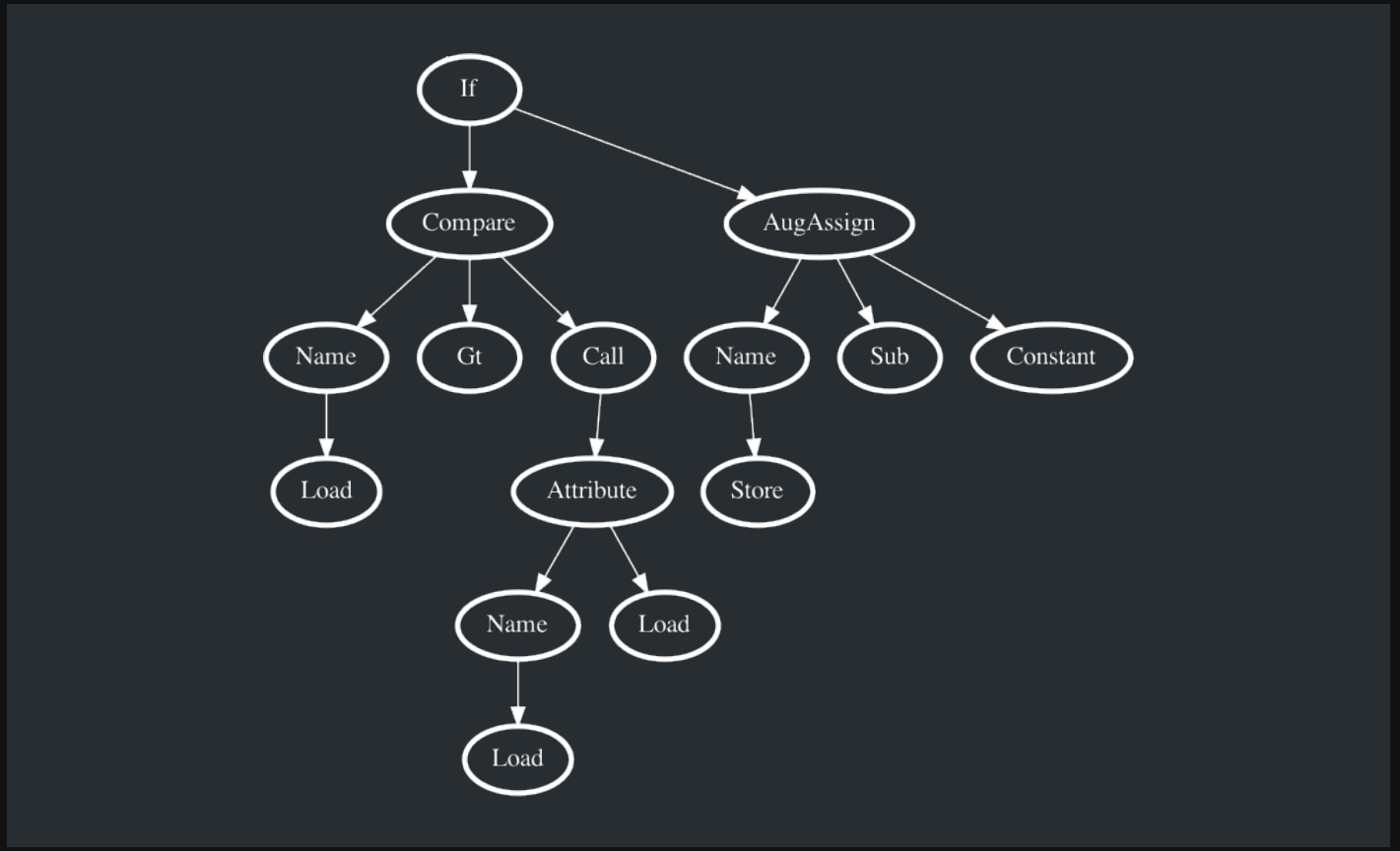
```
print(cannon1, cannon2)
```



最大深さ 3

実験１ ： 部分木抽出時の最適な深さの検討［結果］

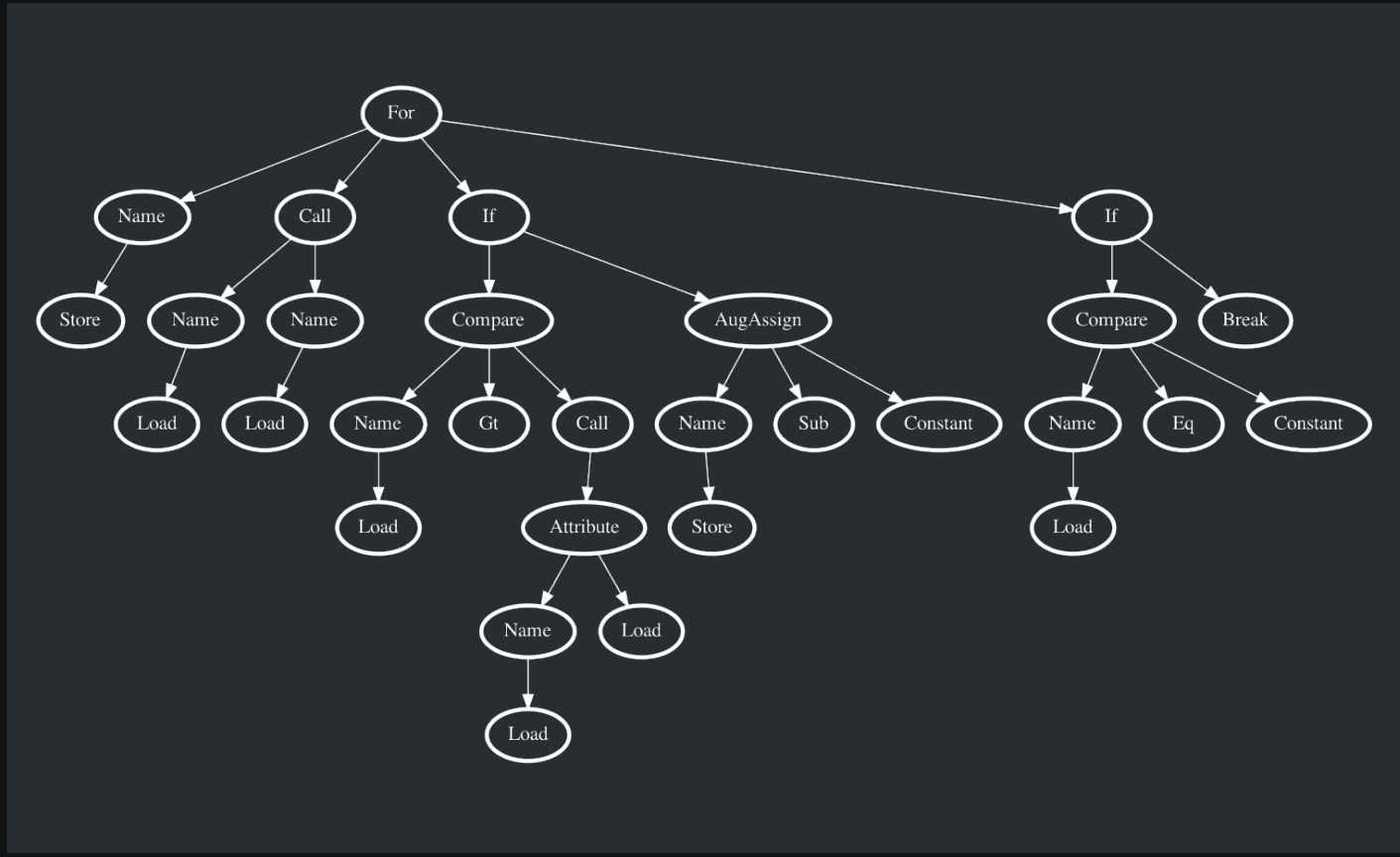

```
if rate1 > random.random():
      cannon2_ -= 1
```



最大深さ 5

実験１ ： 部分木抽出時の最適な深さの検討［結果］


```
for _ in range(cannon1):
    if rate1 > random.random():
        cannon2_ -= 1
    if cannon2_ == 0:
        break
```



最大深さ 6

考察

- 深さ3以下の部分木は，主に代入文などの1行の処理を表している
- これらを関数化することは，保守作業の効率化にあまり有用ではない
- 本ソースコードから抽出する部分木の深さ**kは5以上が適切**である

実験２ ： 自動関数生成機能の有効性(一致）[目的]

実験環境

- 使用言語 ： Python
- 入力ソースコード ： 右記
  - 総行数 ： 15行（空行除く）
- 部分木の最大深さ ： k=5，k=6

実験目的

- **一致するコードクローン**に対する自動関数生成機能の有効性を検証．
- 部分木の最大深さkの違いによる実行結果の変化を調査．

```
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3
for _ in range(cannon1):
    if rate1 > random.random():
        cannon2_ -= 1
    if cannon2_ == 0:
        break
for _ in range(cannon2):
    if rate2 > random.random():
        cannon1_ -= 1
    if cannon1_ == 0:
        break
cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```


Department of Data Science, Musashino University.

実験２ ： 自動関数生成機能の有効性(一致)［結果］

k=5

- **if文による条件分岐**のみ関数化
- 2つの引数、1つの戻り値
- 総行数 ： 17行
  - 関数部 ： 4行
  - 実行部 ： 13行

```
def function1(var1, var2):
    if var1 > random.random():
        var2 -= 1
    return var2
```

```
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3
for _ in range(cannon1):
    cannon2_ = function1(rate1, cannon2_)
    if cannon2_ == 0:
        break
for _ in range(cannon2):
    cannon1_ = function1(rate2, cannon1_)
    if cannon1_ == 0:
        break
cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

実験２ ： 自動関数生成機能の有効性(一致)［結果］

k=6

- for文による繰り返し文が関数化
- 3つの引数、1つの戻り値
- 総行数 ： 14行
  - 関数部 ： 7行
  - 実行部 ： 7行

```
def function1(var1, var2):
    for var1 in range(var2):
        if var3 > random.random():
            var4 -= 1
        if var4 == 0:
            break
    return var4
```

```
import random

cannon1, cannon2 = 10, 30
rate1, rate2 = 0.5, 0.3

cannon2_ = function1(cannon1, rate1, cannon2_)
cannon1_ = function1(cannon2, rate2, cannon1_)

cannon1, cannon2 = cannon1_, cannon2_
print(cannon1, cannon2)
```

考察

- 本ソースコードにおいて，
  k=5，k=6のどちらも機能のまとまりを関数に置き換えることに成功した．

- 空行を除く行数を比較すると， k=5のときは総行数が**増加**
  k=6のときは総行数を減少

-> **適切な深さを設定**することができれば，**保守性の向上**が可能である．

実験３ ： 自動関数生成機能の有効性(類似)［目的］

実験環境

- 使用言語 ： Python
- 入力ソースコード ： 右記
  - 総行数 ： 16行（空行除く）
- 部分木の最大深さ ： k=5，k=6
- 閾値 ： ε=9

実験目的

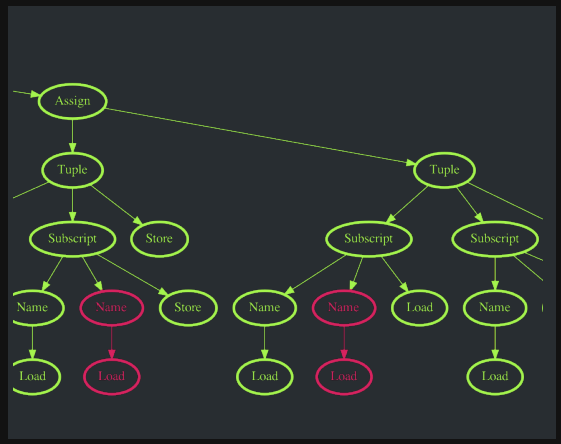
- **類似するコードクローン**に対する自動関数生成機能の有効性を検証．

```
# Exchange Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n):
     for j in range(i+1, n):
         if A[j] < A[i]:
             A[i], A[j] = A[j], A[i]
print(A)

# Bubble Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n-1):
    for j in range(n-1):
        if A[j+1] < A[j]:
            A[j], A[j+1] = A[j+1], A[j]
print(A)
```

実験３ ： 自動関数生成機能の有効性(類似)［目的］

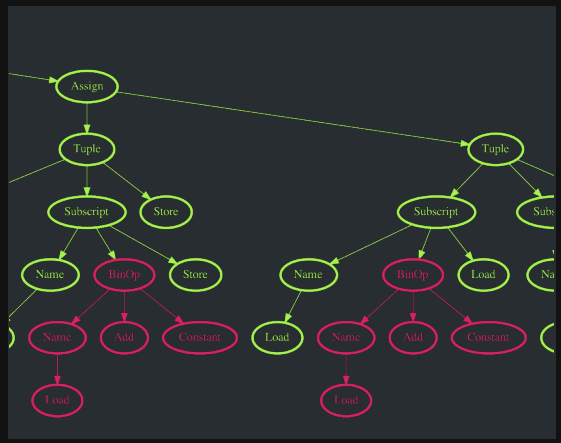
Exchange Sort

```
for i in range(n):
    for j in range(i+1, n):
        if A[j] < A[i]:
            A[i], A[j] = A[j], A[i]
```



Bubble Sort

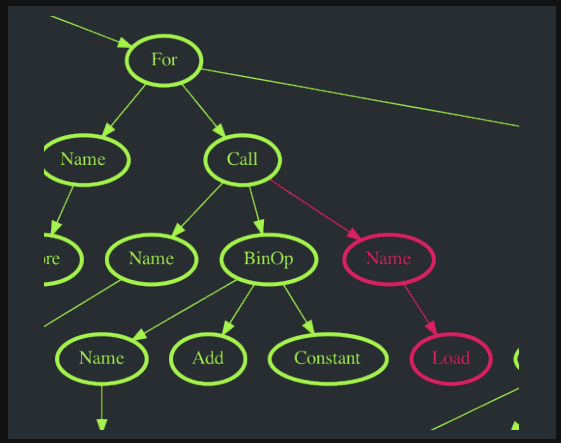
```
for i in range(n-1):
    for j in range(n-1):
        if A[j+1] < A[j]:
            A[j], A[j+1] = A[j+1], A[j]
```



実験３ ： 自動関数生成機能の有効性(類似)［目的］

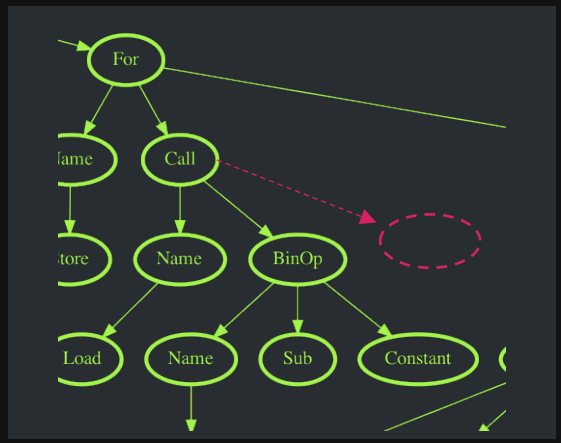
Exchange Sort

```
for i in range(n):
    for j in range(i+1, n):
        if A[j] < A[i]:
            A[i], A[j] = A[j], A[i]
```



Bubble Sort

```
for i in range(n-1):
    for j in range(n-1 ):
        if A[j+1] < A[j]:
            A[j], A[j+1] = A[j+1], A[j]
```



実験３ ： 自動関数生成機能の有効性(類似)［結果］

k=5

- **値の大小比較および**
  更新部分が関数化
- 総行数 ： 18行
  - 関数部 ： 4行
  - 実行部 ： 14行

```
def function1(var1, var2, var3):
    if var1[var2] < var1[var3]:
        var2[var3], var1[var2] = var1[var2], var1[var3])
    return var1
```

```
# Exchange Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n):
    for j in range(i+1, n):
        A = function1(A, j, i)
print(A)
# Bubble Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n-1):
    for j in range(n-1):
        A = function1(A, j+1, j)
print(A)
```

実験３ ： 自動関数生成機能の有効性(類似)［結果］

k=6以上

- 関数化されなかった

```
# Exchange Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n):
      for j in range(i+1, n):
          if A[j] < A[i]:
               A[i], A[j] = A[j], A[i]
print(A)

# Bubble Sort
A = [3, 5, 1, 9, 7, 8, 5]
n = len(A)
for i in range(n-1):
     for j in range(n-1):
          if A[j+1] < A[j]:
               A[j], A[j+1] = A[j+1], A[j]
print(A)
```

考察

- 本ソースコードにおいて，
  k=5のときは機能のまとまりを関数に置き換えることに成功した

- 行数の総量は増加してしまったが，実行部のコード量は減少している

-> 長いソースコードに対しては恩恵が得られる可能性がある

- 本ソースコードにおいて，kが6以上のときは**関数化されなかった**
  - for文に含まれる**range関数の引数の数**が違うことが**要因**である．
  -> 引数の数に応じてデフォルト引数を加えることで解決できる．

考察

- ソースコードを理解する上で，ネストが深いほど可読性が落ちる

- for文で頻繁に用いられるrange関数の**引数の違いに対応**する
  -> **ソースコードの簡略化**および**可読性の向上**に寄与できる

おわりに

研究概要

- 研究背景
- 研究目的

提案方式

- システムの全体像
- 構文解析 -> 部分木抽出 -> 類似度計量 -> 構造同一化 -> 関数生成

実験

1. 最適な深さの検討
2. 一致するコードクローンへの有効性
3. 類似するコードクローンへの有効性

おわりに

- まとめ
- 今後の課題

まとめ

ソースコードの構文木表現による構造類似性を用いた自動関数生成方式

- ソースコードの構造類似性から**コードクローンを検出**した.
- **類似するコードクローン**についても構造を変形させることで,
  関数生成の適用範囲を拡張した.
- **コードクローンを関数に置き換える**ことで構造の簡略化を実現した.

-> コードクローンを集約することにより,
保守性を向上させることが可能になった.

今後の展望

- 引数の数の違いに対応
- パラメータ(部分木の深さ・編集距離の閾値)の自動調整機能
- 識別子の命名機能
- ソースコードの等価性検証

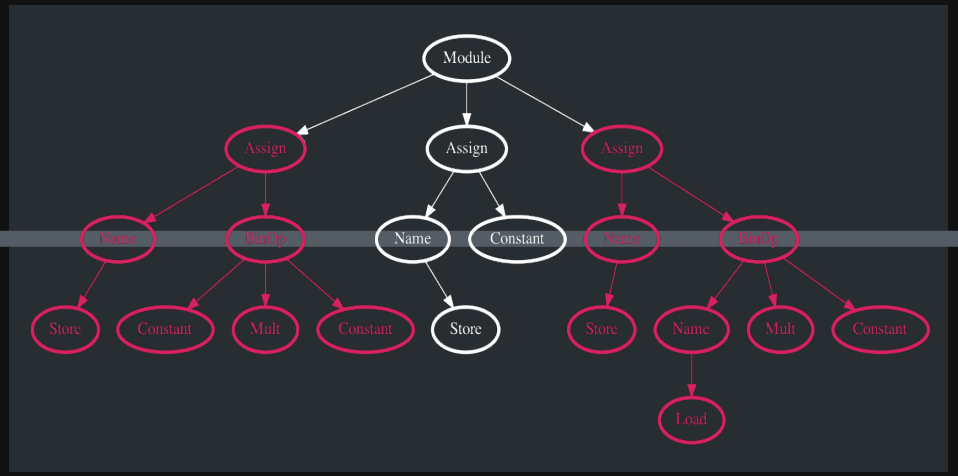
ソースコードの構文木表現による構造類似性を用いた自動関数生成方式

北 椋太　岡田 龍太郎　峰松 彩子　中西 崇文　武蔵野大学データサイエンス学部

1. 構文木の類似性からコードクローンを検出
2. 類似するコードクローンを変形し，構造を同一化
3. コードクローンを関数に置き換えることで保守コストの低減

```
a = 5 * 10
b = 3
c = a * 10
```

入力



構文木

```
def function1(var2):
  var1 = var2 * 10
  return var1


a = function1(5)
b = 3
c = function1(a)
```

出力