

Pythonプログラムにおける 関数抽出の検証手法の 提案および実装

谷元柊介

目次

- 研究背景
- 研究目的及び研究内容
- プログラムの一致の定義
- 実際の動作過程
- 入力プログラムに対する制約条件
- まとめ

目次

- 研究背景
- 研究目的及び研究内容
- プログラムの一致の定義
- 実際の動作過程
- 入力プログラムに対する制約条件
- まとめ

リファクタリング

- ソフトウェアの外部的な振る舞いを変更することなく内部の構造を改善することをリファクタリングという
- リファクタリングの目的→コードの再利用性、可読性の向上

関数抽出

- リファクタリングの中でも、ひとまとまりの処理を1つの関数で置き換えることでソースコード全体の見通しをよくしようとする取り組みは、「関数抽出」と定義されている[1].
- 関数抽出の中でも、特にクラスの中の関数で行うものは「メソッド抽出」と定義されている.
- 本研究のプログラムでは、メソッド抽出の検証は対象外

[1] M. Foeler, Refactoring Catalog, <https://refactoring.com/catalog/>, 2025/1/15 時点.

関数抽出の例

```
1 #プログラムA
2 list=[1,2,3,4]
3 result=0
4 for i in list:
5     result+=i
6 print(result)
7 list.append(5)
8 result=0
9 for i in list:
10     result+=i
11 print(result)
```

重複部分を置換するための
関数を新たに定義

重複している
箇所を新たに
定義した関数で
置換

複数か所に重複したコードが
あるプログラム

```
1 #プログラムB
2 def sum(a):
3     result=0
4     for i in a:
5         result+=i
6     return result
7
8 list=[1,2,3,4]
9 print(sum(list))
10 list.append(5)
11 print(sum(list))
```

左のプログラムに関数抽出
を行ったプログラム

リファクタリング検証の必要性(1/2)

- リファクタリングはプログラム改善のためによく行われる
- 誤ったリファクタリングは、意図せぬ処理の変更を引き起こす
 - 正しいリファクタリングかどうかを検証するツールが必要

リファクタリング検証の必要性(2/2)

- 現在, 大規模言語モデル(LLM)を使ってリファクタリングを行う手法がすでに提案されている[2].
- しかし, LLMによる出力はその過程がブラックボックス化している
 - 説明可能性が確保されていない. (=なぜその出力が正しいのか説明できない)
 - 説明可能性を確保するために検証が必要

[2]Zhang, Z., Xing, Z., Ren, X., Lu, Q., & Xu, X. (2024). Refactoring to pythonic idioms: A hybrid knowledge-driven approach leveraging large language models. *Proceedings of the ACM on Software Engineering*, 1(FSE), 1107-1128.

リファクタリング検証ツールの必要性

- 正しいリファクタリングかどうか検証するには**テストケースを使うのが一般的**
- 十分なテストケースがない場合は**難しい**(ex. オープンソースプロジェクト等)
- テストケースなしでも検証できる手法およびツールが必要

目次

- 研究背景
- 研究目的及び研究内容
- プログラムの一致の定義
- 実際の動作過程
- 入力プログラムに対する制約条件
- まとめ

研究概要

研究目的: 十分なテストケースがない状況下でのリファクタリングの検証を容易にする

研究内容: 関数抽出が行われた2つのPythonプログラム の処理の一致をテストケースなしで検証する手法の提案および実装

先行研究

先行研究[3]ではC/C++向けリファクタリング検証ツールPOM/EQを開発した.

- ◆ 抽象構文木を基礎としたモデルの比較により検証
 - プログラムの文法事項の中で, プログラムの意味に関わる物のみを木構造で表現したものを**抽象構文木**という
 - **本研究も同様に抽象構文木の比較により検証**

[3]市井 誠, 小川 秀人: モデル変換を用いたリファクタリング検証手法. ソフトウェア工学の基礎研究会 (FOSE2014). ソフトウェア科学会, 2014.

目次

- 研究背景
- 研究目的及び研究内容
- プログラムの一致の定義
- 実際の動作過程
- 入力プログラムに対する制約条件
- まとめ

プログラムの一致の定義

本研究ではプログラムの一致を3つに分類

完全一致: 入力した2つのプログラムに対する抽象構文木の構造および
ノードの要素が完全に一致

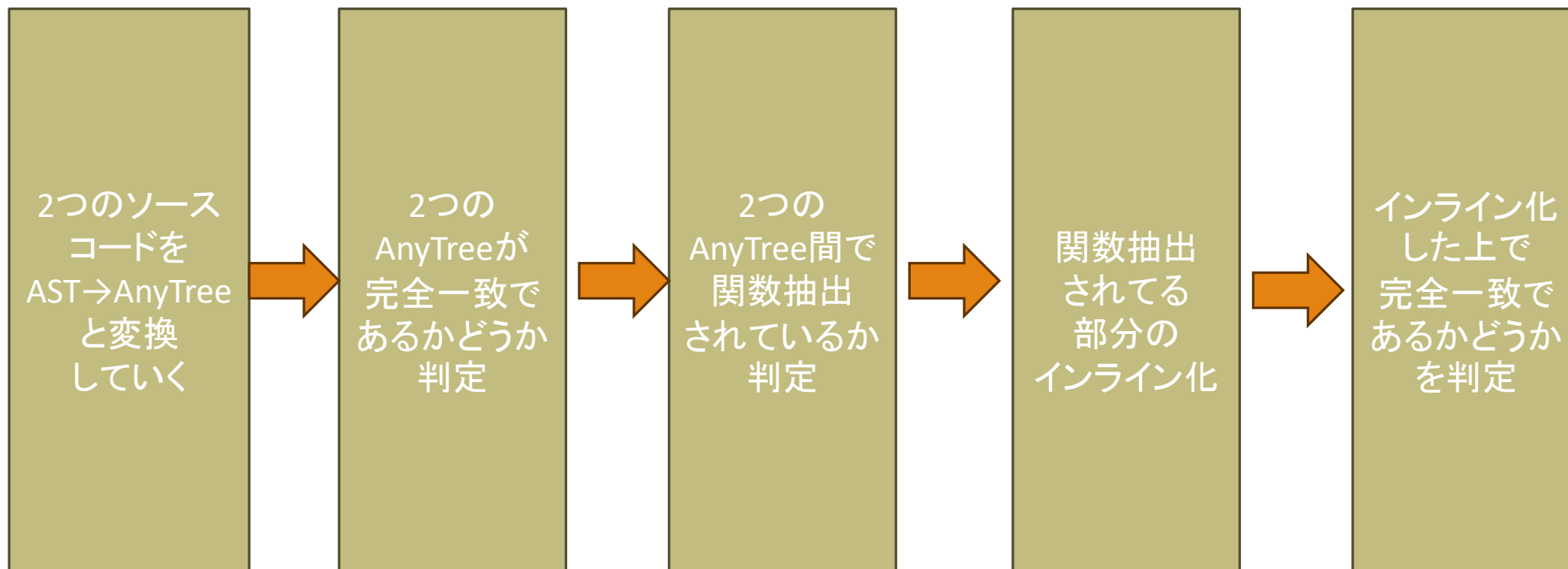
構造的に一致: 完全一致ではないが、関数抽出された箇所を元に戻せば
完全に一致

一致していない: 上記2つのどちらにも該当していない

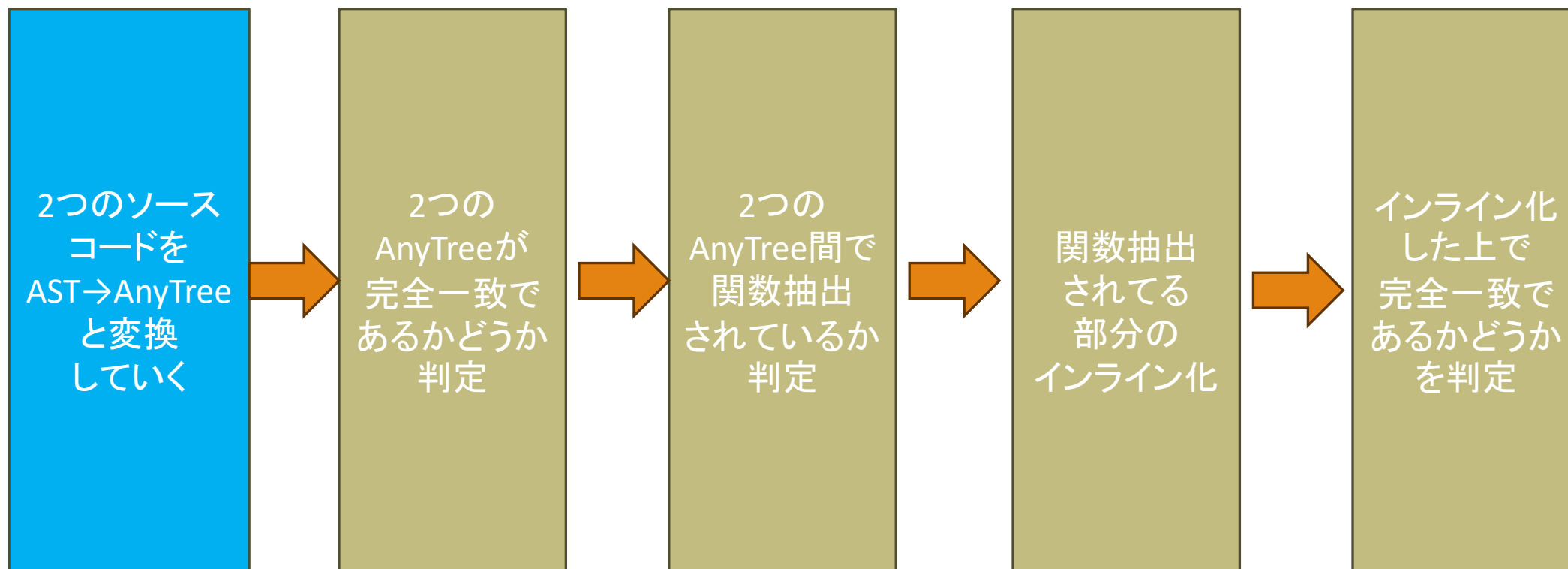
目次

- 研究背景
- 研究目的及び研究内容
- プログラムの一致の定義
- 実際の動作過程
- 入力プログラムに対する制約条件
- まとめ

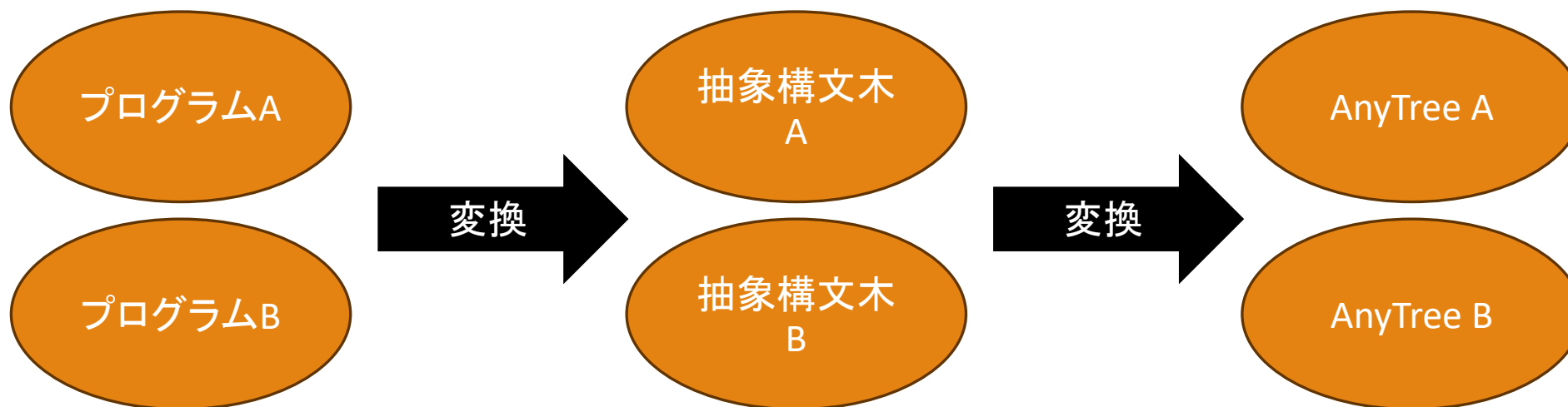
動作の流れ



動作の流れ



判定プログラムの動作その1



入力として2つのプログラムを受け取り, それを木構造のデータを扱いやすくするデータ型AnyTreeに変換していく (一部既存モジュールにより実現)

プログラム→AST

Pythonのastモジュールを使ってASTを構築

Pythonプログラム

```
1 print(0)
2 print(1)
```

Moduleノードの配下にそれぞれの文に該当するASTが本文での順番通りに格納されている

AST

```
Module(
  body=[
    Expr(
      value=Call(
        func=Name(id='print', ctx=Load()),
        args=[
          Constant(value=0)],
        keywords=[])),
    Expr(
      value=Call(
        func=Name(id='print', ctx=Load()),
        args=[
          Constant(value=1)],
        keywords=[])),
  type_ignores=[])
```

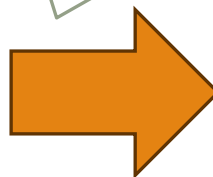
AST→AnyTree

ASTをより扱いやすくするためにAnyTreeモジュールのデータ型であるAnyTreeに変換

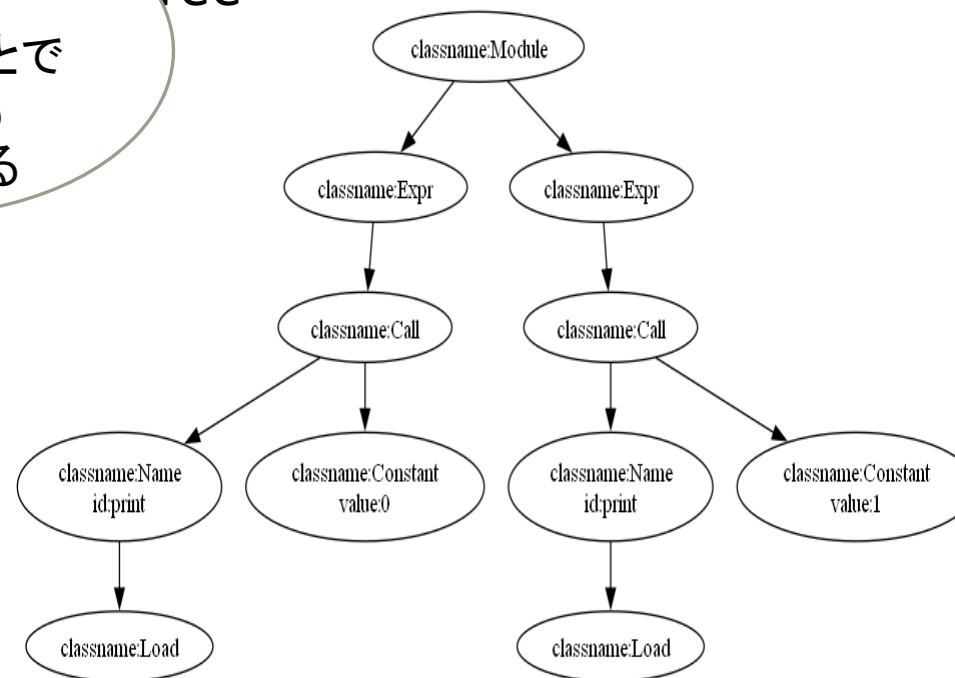
AST

```
Module(  
  body=[  
    Expr(  
      value=Call(  
        func=Name(id='print', ctx=Load()),  
        args=[  
          Constant(value=0)],  
        keywords=[])),  
    Expr(  
      value=Call(  
        func=Name(id='print', ctx=Load()),  
        args=[  
          Constant(value=1)],  
        keywords=[])),  
    type_ignores=[])
```

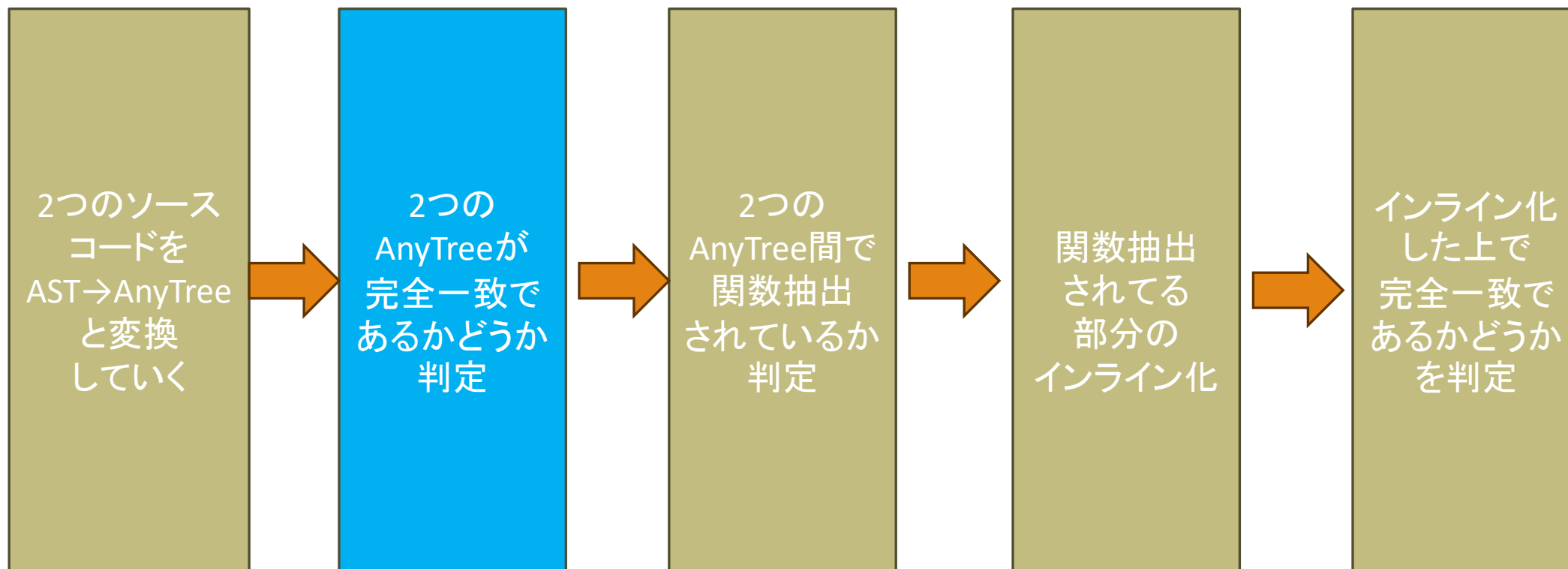
ASTの余分な
要素を
削除することで
簡素化も
行っている



AnyTree



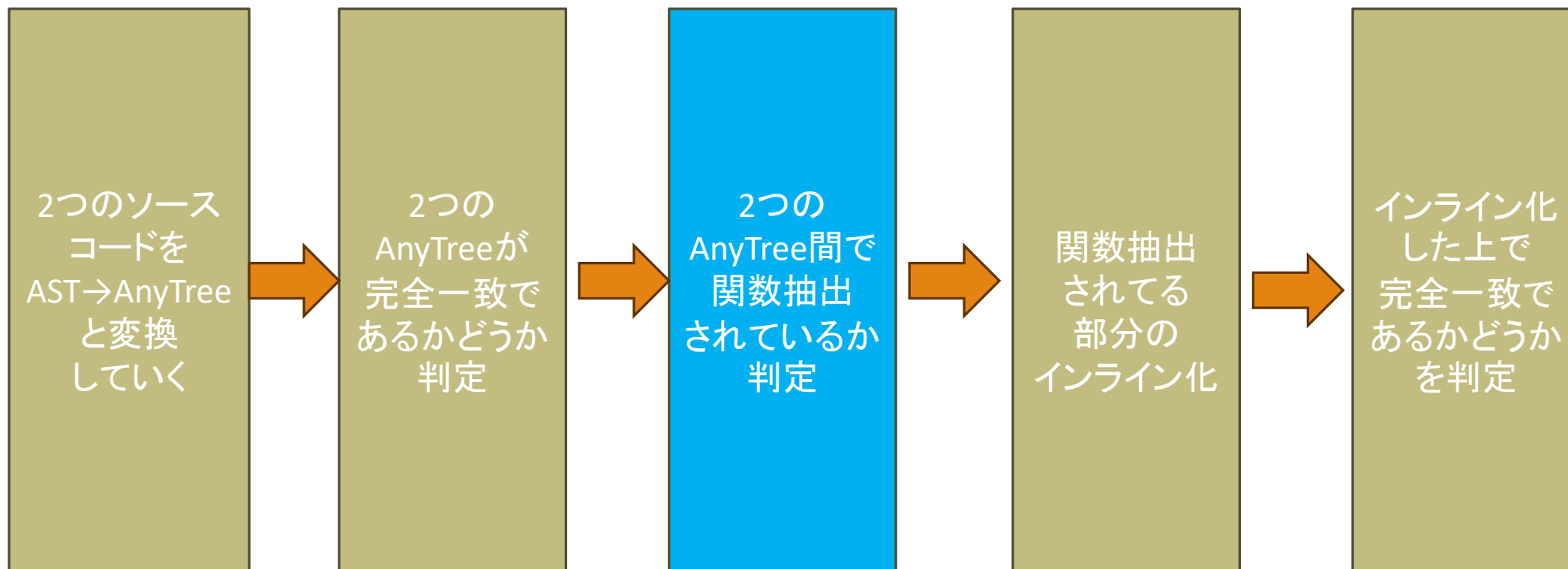
動作の流れ



判定プログラムの動作その2

- 変換で得られたAnyTreeの**比較**を行う
- 構造, ノードの値すべてが等しければ「**完全一致**」, そうでなければ, 「**完全一致でない**」場合の処理に進む

動作の流れ



関数抽出されているかどうかの判定

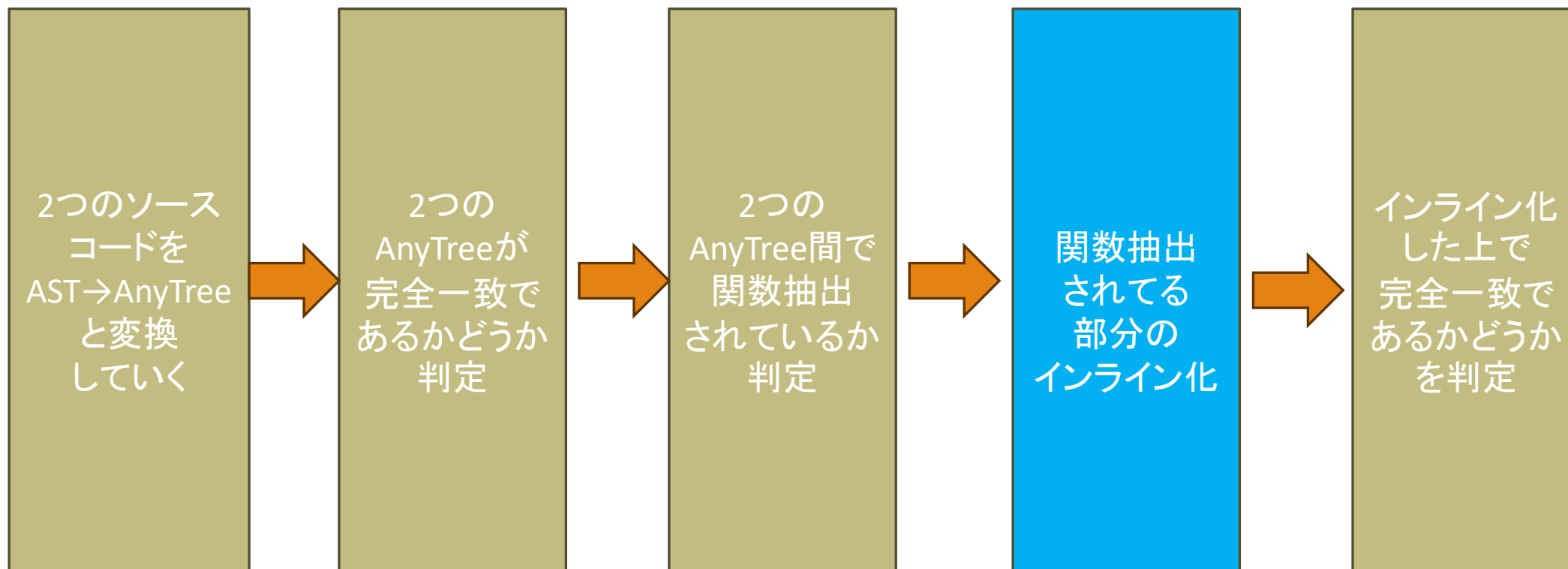
関数抽出が行われているかどうかの条件をみる

- ・ Bで新たに関数fが定義されている
- ・ Aのコードの一部Sが, Bにおいて関数fの呼び出しで置き換えられている
- ・ 上記S が, Bにおいて関数fの定義のソースコードと一致している

「完全一致でない」場合の処理

- 前述の3つの条件を満たしていれば関数抽出を元に戻す処理へ, そうでなければ「一致していない」と出力

動作の流れ

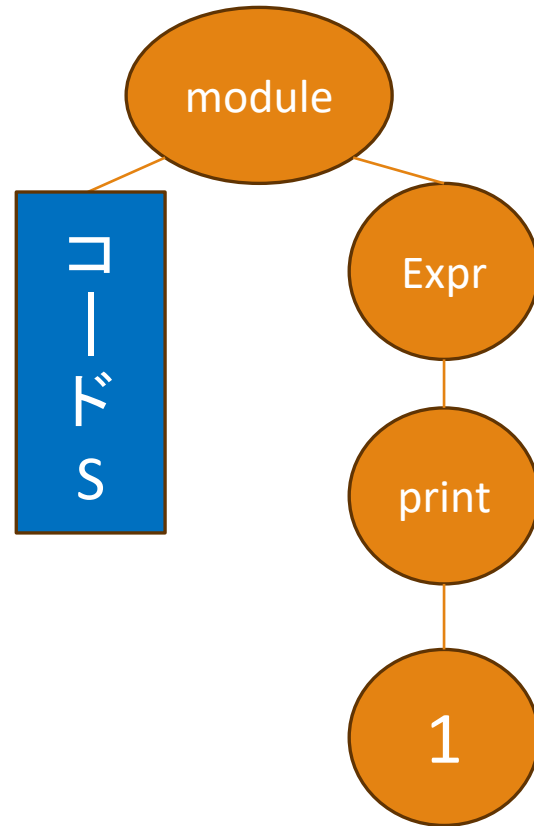


「完全一致でない」プログラムの組の例

プログラムA

コードS

print(1)



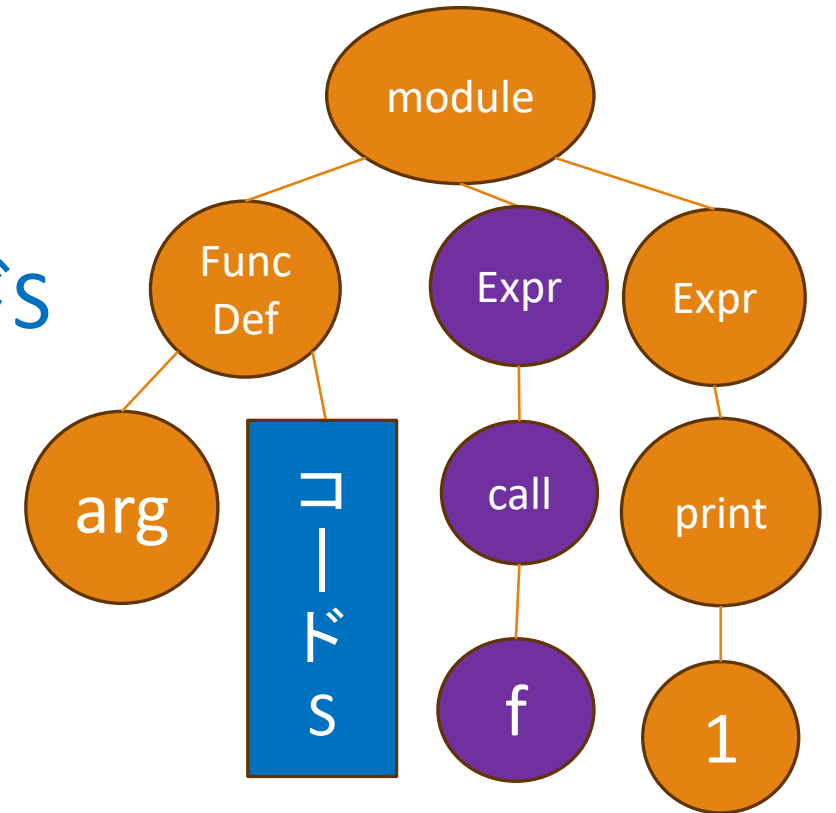
プログラムB

def f():

コードS

f()

print(1)

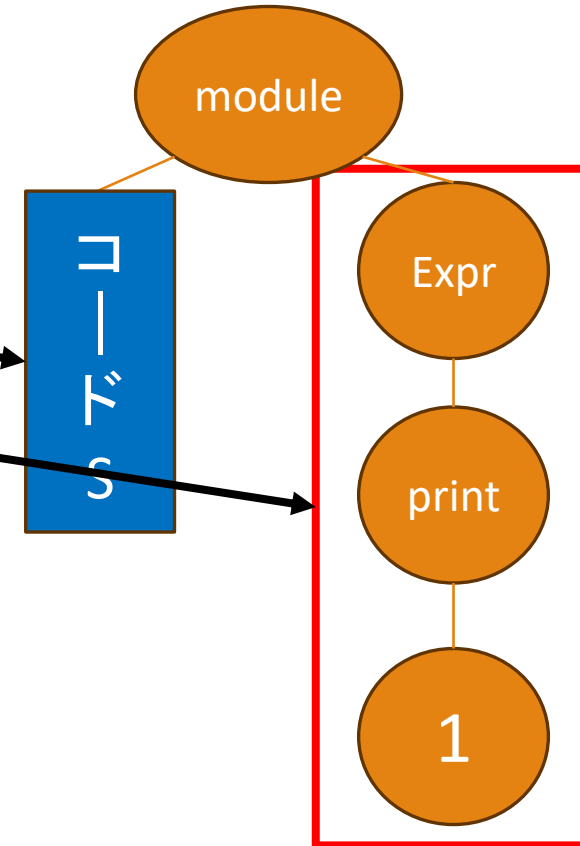


プログラムAのコードと木の対応関係

プログラムA

コードS

print(1)



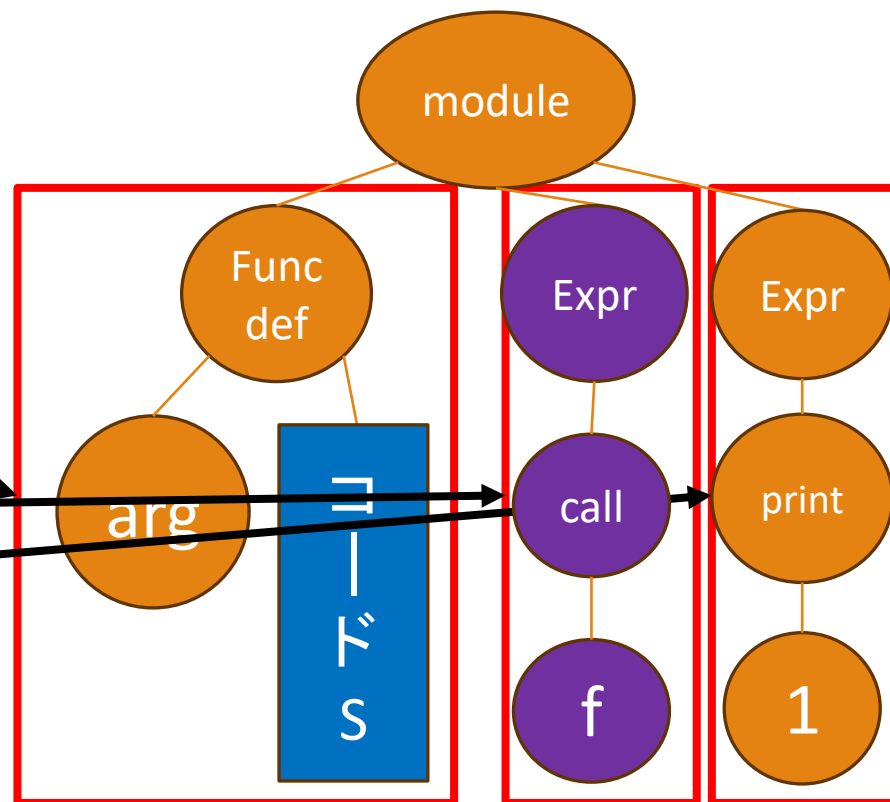
プログラムBのコードと木の対応関係

プログラムB

```
def f():  
    コード's
```

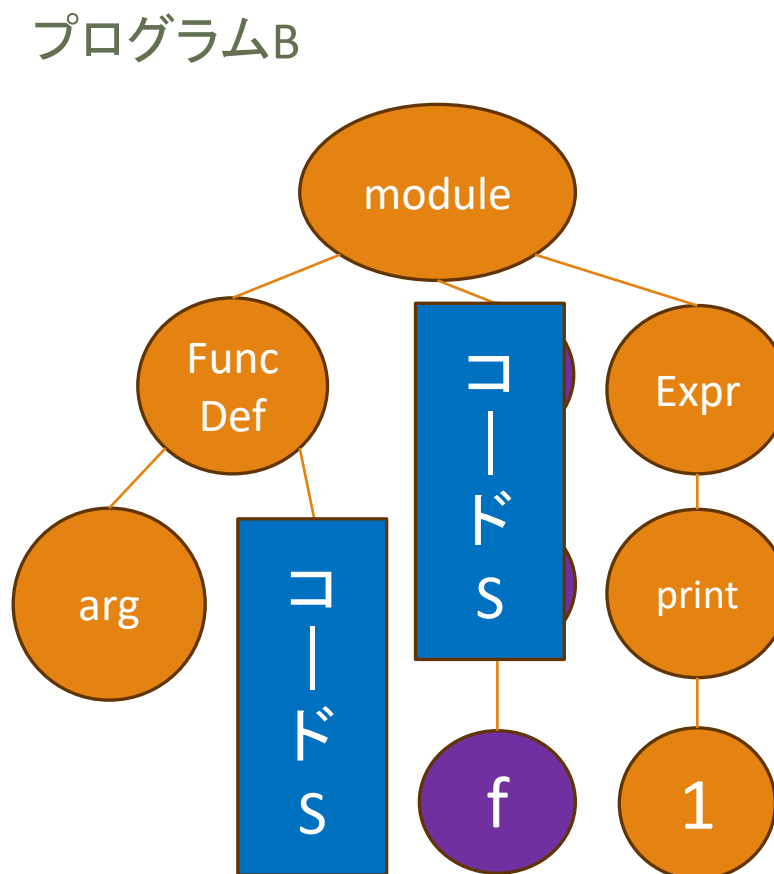
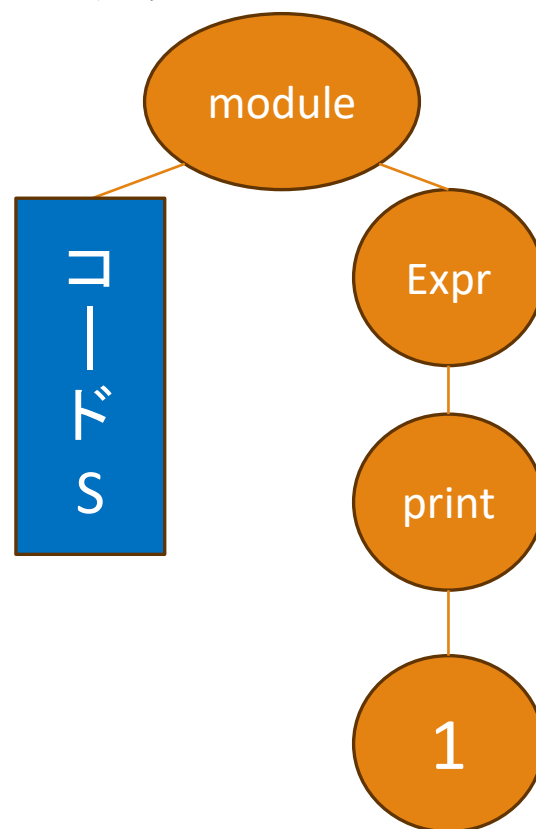
f()

print(1)



関数抽出を元に戻す処理

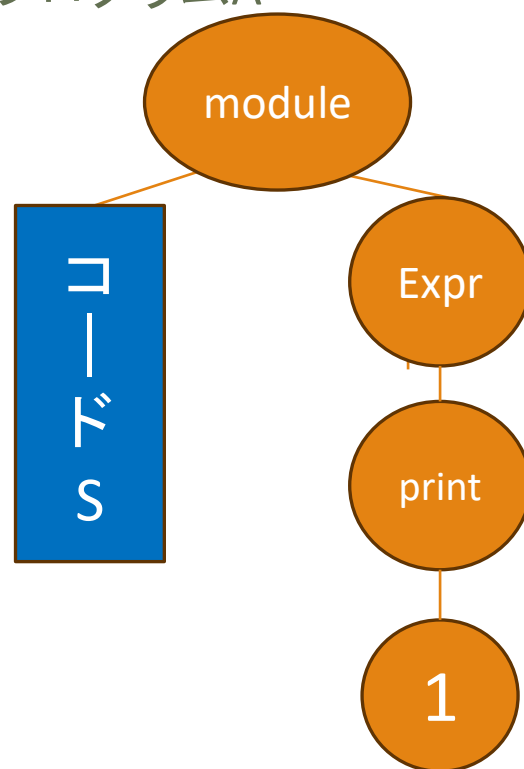
①プログラムBにおいて、プログラムA
関数抽出で
使われている関数の
呼び出しを、その関数の
部分コードsに置き換え
る



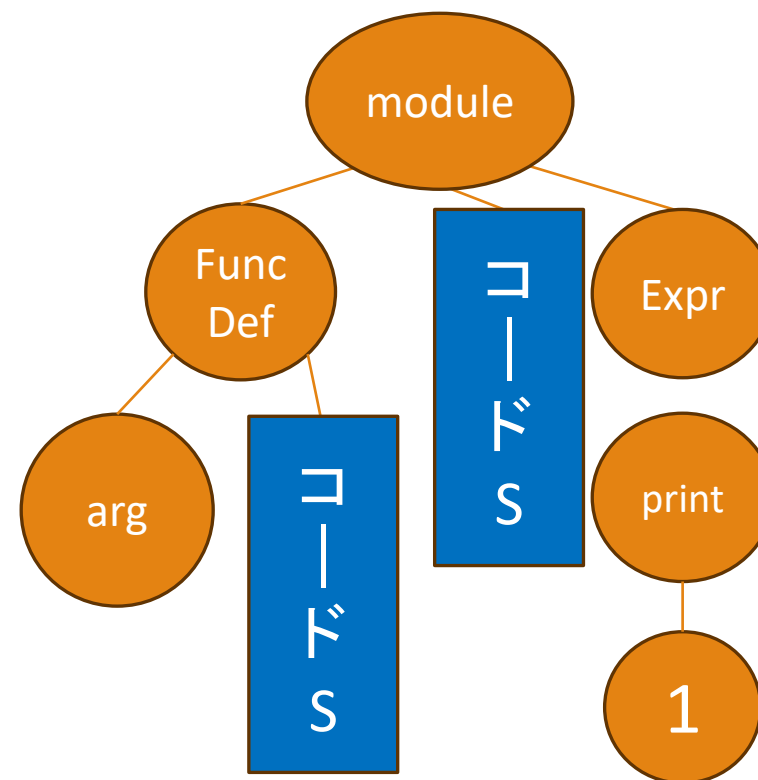
関数抽出を元に戻す処理

②関数抽出で
使われている関数
の定義部分を削除
する

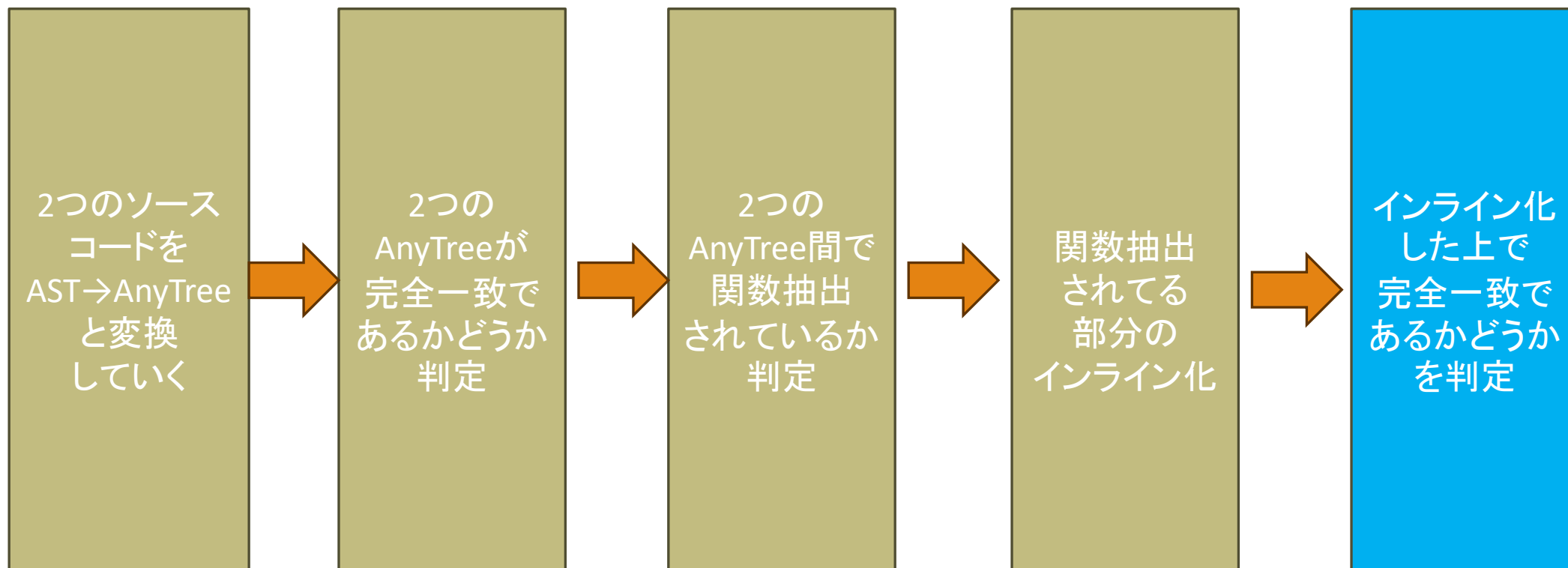
プログラムA



プログラムB



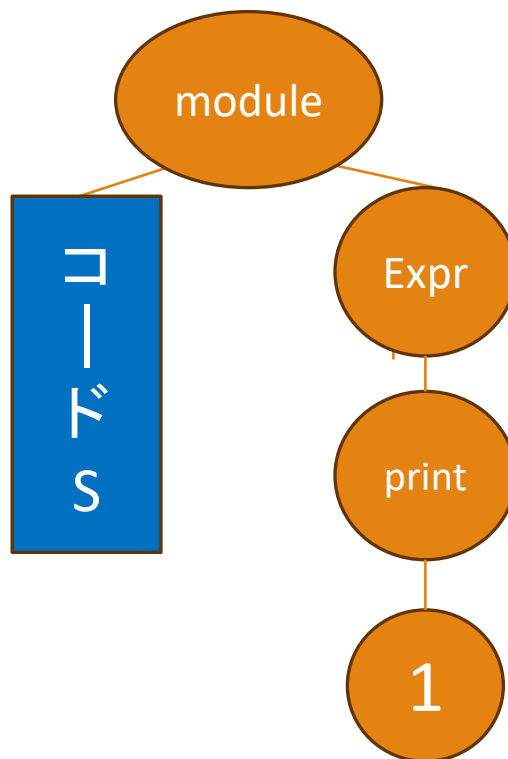
動作の流れ



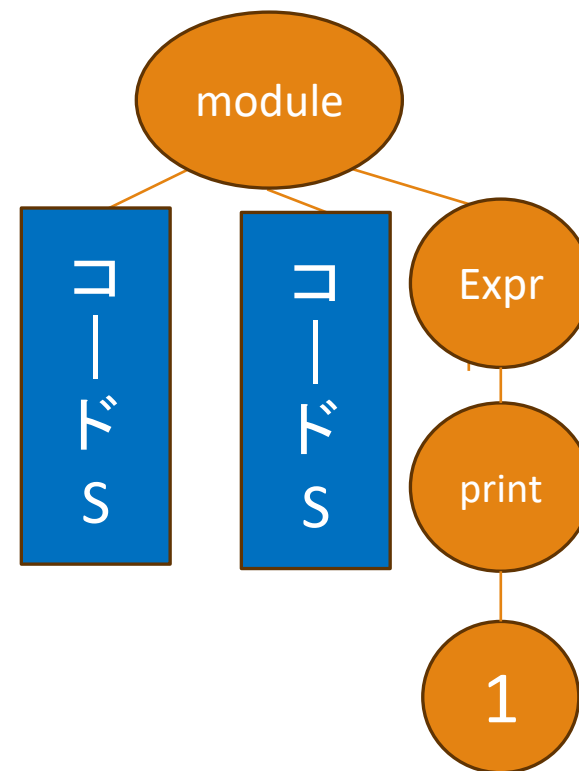
関数抽出を元に戻す処理

再度比較を行い、構造とノードの値が一致していれば「**構造的に一致**」
そうでなければ「**一致していない**」

プログラムA



プログラムB



目次

- 研究背景
- 研究目的及び研究内容
- プログラムの一致の定義
- 実際の動作過程
- 入力プログラムに対する制約条件
- まとめ

入力プログラムに対する制約条件

- 新たに定義される関数の呼び出しの引数は**変数を含まない式**でなくてはならない
- 以下のプログラムA, Bは、プログラムBの関数funcの呼び出しの引数に**変数x**を使っているため、判定できない

プログラムA

```
1  x=2
2  y=x*x
3  print(y)
4
5
6
7
```

プログラムB

```
1  def func(z):
2      y=z*z
3      retu
4
5  x=2
6  print(func(x))
7
```

ここに変数
があるため
判定不可

入力プログラムに対する制約条件

- 返り値に変数を使う場合は置き換え前のコードにつかわれている変数名と同じでなくてはならない
- 以下のプログラムA, Bは上記の条件を満たさないため、本来は「構造的一致」であるが「一致していない」と出力される

プログラムA

```
1 y=2*2
2 print(y)
3
4
5
```

変数yがあったところにfuncの呼び出しがきている

プログラムB

```
def func(z):
    x=z*z
    return x
2
3
4
5 print(func(2))
```

入力プログラムに対する制約条件

- 返り値に変数を使う場合は置き換え前のコードにつかわれている変数名と同じでなくてはならない
- 以下のプログラムA, Bは上記の条件を満たさないため、本来は「構造的一致」であるが「一致していない」と出力される

プログラムA

```
1 y=2*2
2 print(y)
3
4
5
```

変数yがあったところにfuncの呼び出しがきている

比較の際、その構造に関わらず、xとyは変数名が異なるため異なる変数として扱われる

プログラムB

```
def func(z):
    x=z*z
    return x
print(func(2))
```

入力可能プログラムの例

制約条件さえ
満たしていれば、
右図のような
入れ子を含む
複雑な構造の
プログラムでも
判定可能

※赤枠は関数抽出が
行われた箇所

```
1  #九九のリストを作成
2  list_=[]
3  for i in range(1,10):
4      list_.append([])
5      for j in range(1,10):
6          list_[i-1].append(i*j)
7  list1=list_
8
9  print(*list1,sep='\n')
10
11 list_=[]
12 for i in range(1,12):
13     list_.append([])
14     for j in range(1,12):
15         list_[i-1].append(i*j)
16 list1=list_
17
18 print(*list1,sep='\n')
```

```
1  def make_list(a,b):
2      list_=[]
3      for i in range(a,b):
4          list_.append([])
5          for j in range(a,b):
6              list_[i-1].append(i*j)
7      return list_
8
9  list1=make_list(1,10)
10 print(*list1,sep='\n')
11 list1=make_list(1,12)
12 print(*list1,sep='\n')
```

まとめ

- Pythonプログラムにおいて、リファクタリングの1つである「関数抽出」を検証するための手法を提案した
- また、提案した手法で検証を行うプログラムを実装した
 - 新たに定義された関数の呼び出しの引数等、提案手法の一部は実装が困難であったため、該当部分は制約条件を設けた
 - 実装したプログラムに対して20通りの自作プログラムの組を入力し、正しく判定されていることを確認した

今後の課題

- 返り値の変数に関する制約条件を外すためのアルゴリズムの実装
- try-exceptやwith-asのようなPythonでよく使われるが対応できていない文法への対応

ここから予備スライド

完全一致

入力した2つのプログラムに対する抽象構文木の構造およびノードの要素が完全に一致

```
1  def make_list(a,b):
2      list_=[]
3      for i in range(a,b):
4          list_.append([])
5          for j in range(a,b):
6              list_[i-1].append(i*j)
7      return list_
8
9  list1=make_list(1,10)
10 print(*list1,sep='\n')
11 list1=make_list(1,12)
12 print(*list1,sep='\n')
```

```
1  def make_list(a,b):
2      list_=[]
3      for i in range(a,b):
4          list_.append([])
5          for j in range(a,b):
6              list_[i-1].append(i*j)
7      return list_
8
9  list1=make_list(1,10)
10 print(*list1,sep='\n')
11 list1=make_list(1,12)
12 print(*list1,sep='\n')
```

構造的ー致

完全ー致ではないが、関数抽出された箇所を元に戻せば完全にー致(=インライン化したら完全ー致)

```
1  #九九のリストを作成
2  list_=[]
3  for i in range(1,10):
4      list_.append([])
5      for j in range(1,10):
6          list_[i-1].append(i*j)
7  list1=list_
8
9  print(*list1,sep='\n')
10
11 list_=[]
12 for i in range(1,12):
13     list_.append([])
14     for j in range(1,12):
15         list_[i-1].append(i*j)
16 list1=list_
17
18 print(*list1,sep='\n')
```

インライン化すれば完全ー致

```
1  def make_list(a,b):
2      list_=[]
3      for i in range(a,b):
4          list_.append([])
5          for j in range(a,b):
6              list_[i-1].append(i*j)
7      return list_
8
9  list1=make_list(1,10)
10 print(*list1,sep='\n')
11 list1=make_list(1,12)
12 print(*list1,sep='\n')
```

一致していない

完全一致でも構造的に一致でもない

```
1 y=2*2
2 print(y)
3
4
5
```

そもそも引数が違うので
インライン化しても
完全一致することはない

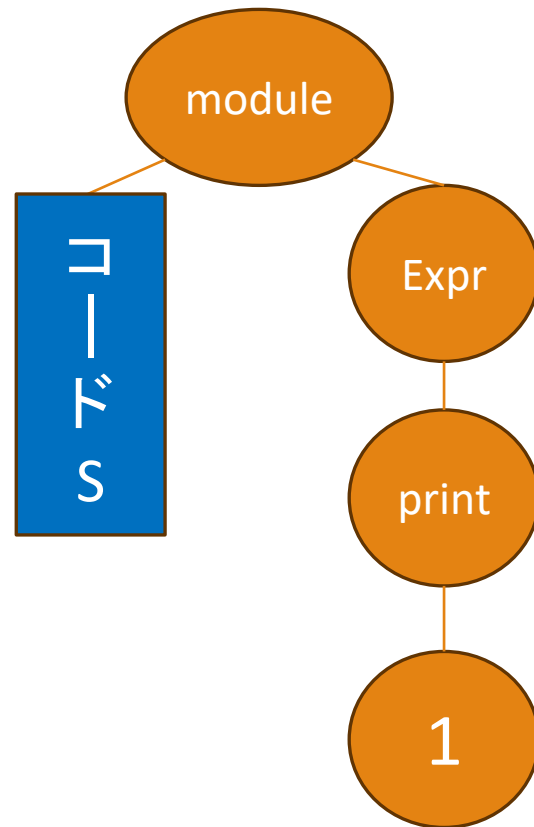
```
1 def func(z):
2     x=z*z
3     return x
4
5 print(func(3))
```

「完全一致でない」プログラムの組の例

プログラムA

コードS

print(1)



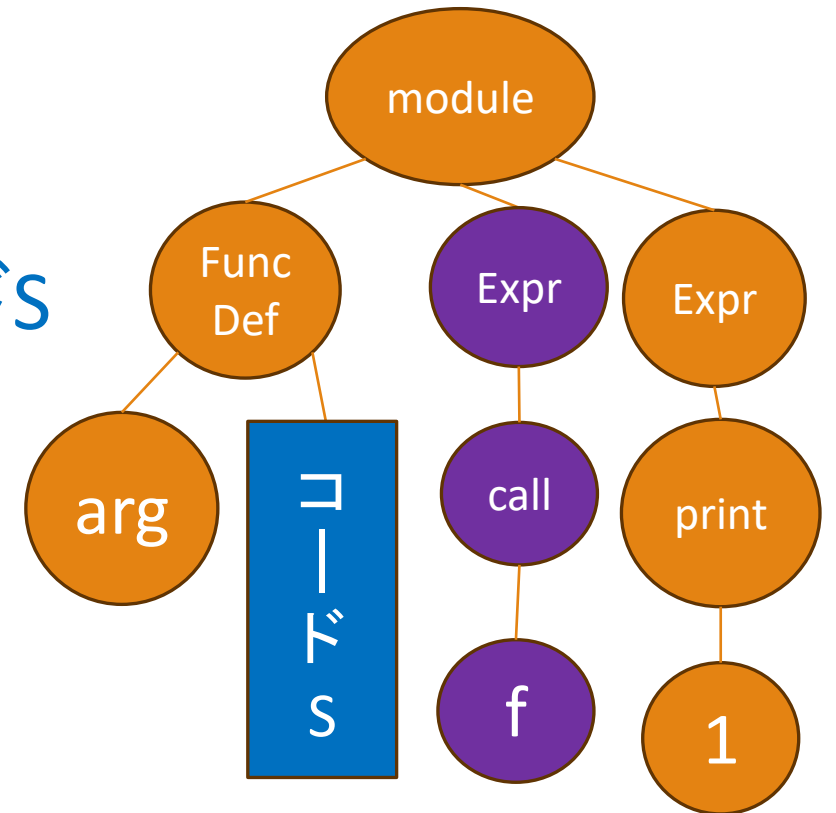
プログラムB

def f():

コードS

f()

print(1)



プログラムA

プログラムA

コードS

print(1)

Expr → 式だけで構成される文であることを示すノード

print → 関数「print」を示すノード

1 → printに与えられている引数「1」を示すノード

コードS

Expr

print

1

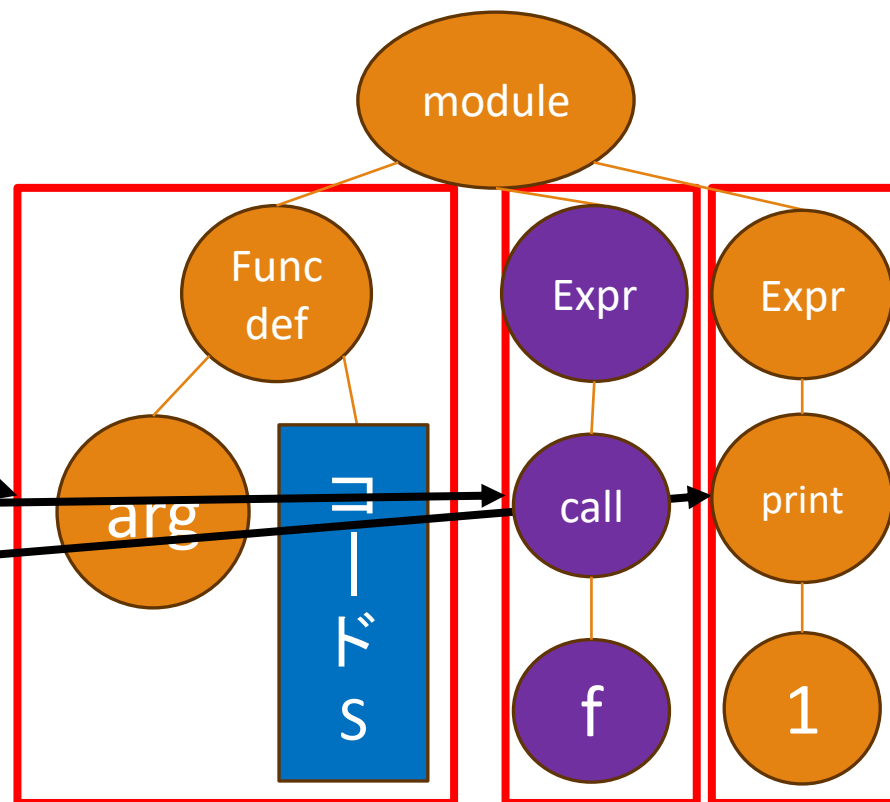
プログラムBのコードと木の対応関係

プログラムB

```
def f():  
    コードS
```

f()

print(1)

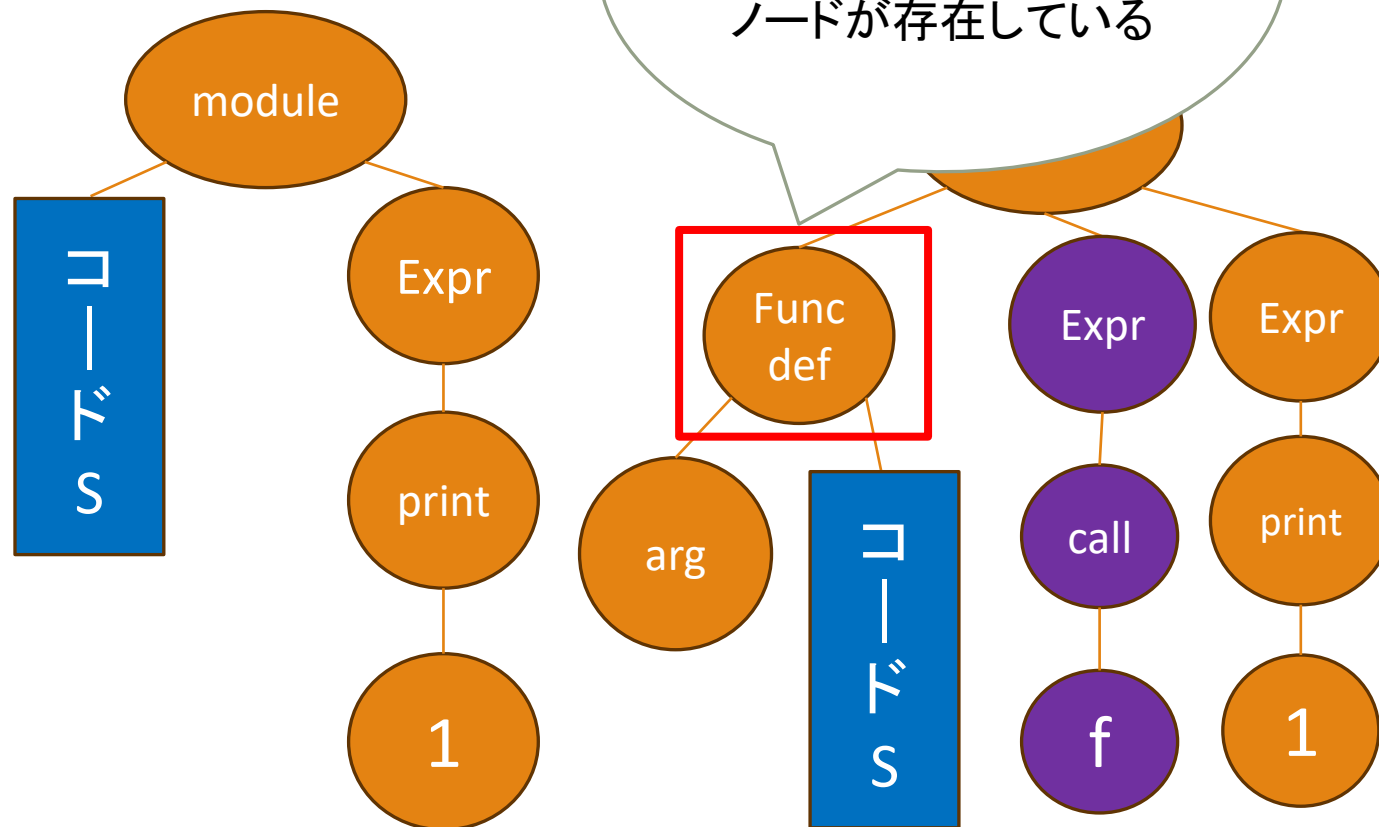


「完全一致でない」場合の処理

関数抽出が行われている
かどうかの条件をみる

・Bで新たに関数fが定義
されている

プログラムA



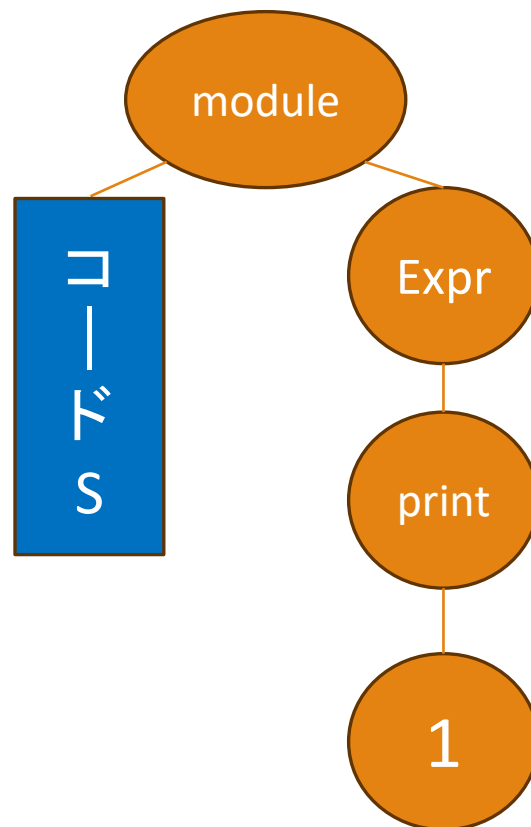
「完全一致でない」場合の処理

関数抽出が行われているかどうかの条件をみる

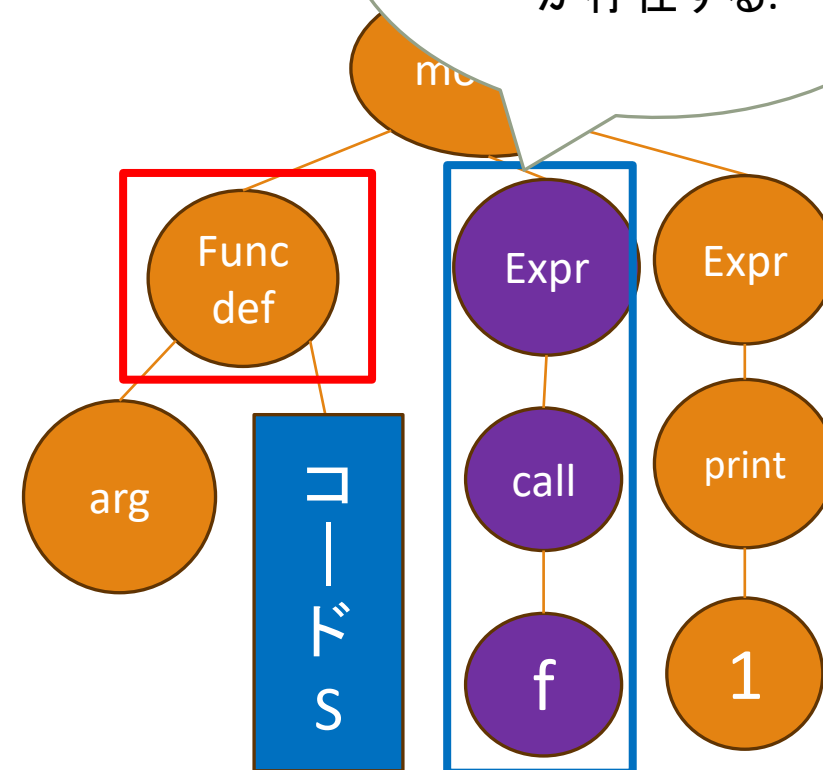
- ・Bで新たに関数fが定義されている

- ・Bに新たに定義された関数の呼び出しが存在する

プログラムA



プログラムB



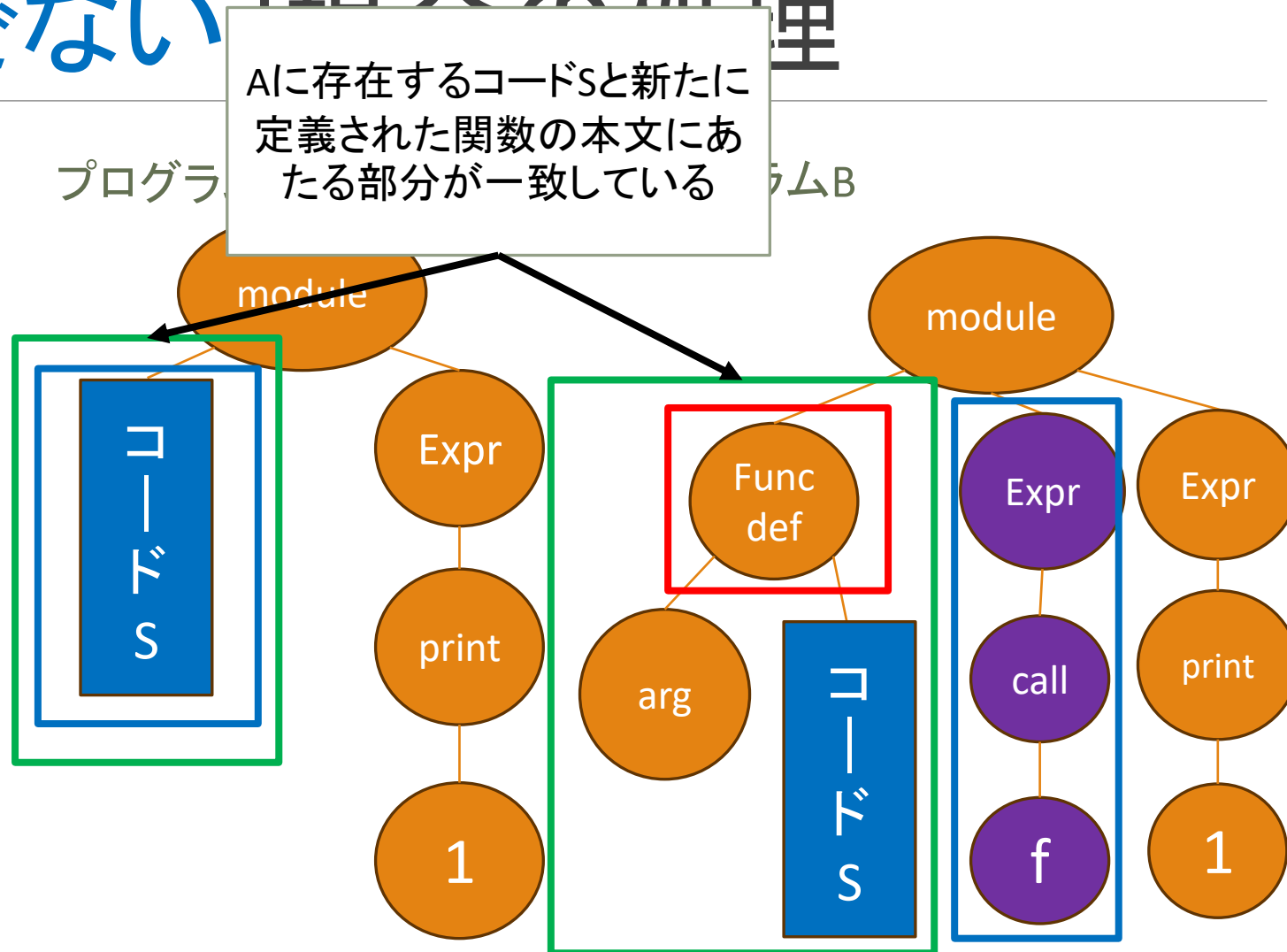
「完全一致でない」場合の処理

関数抽出が行われているかどうかの条件をみる

- ・Bで新たに関数fが定義されている

- ・Aの部分コードSが、Bにおいて関数fの呼び出しで置き換えられている

- ・上記Sが、Bにおいて関数fの定義のソースコードと一致している



(補足)抽象構文木を使うメリット

- ・テキストベースだと、テキストとして完全に一致しているもののしか扱うことができない

→プログラムに影響を与えない空白や改行が1つでも入ると、2つのプログラムは一致していないものとみなしてしまう

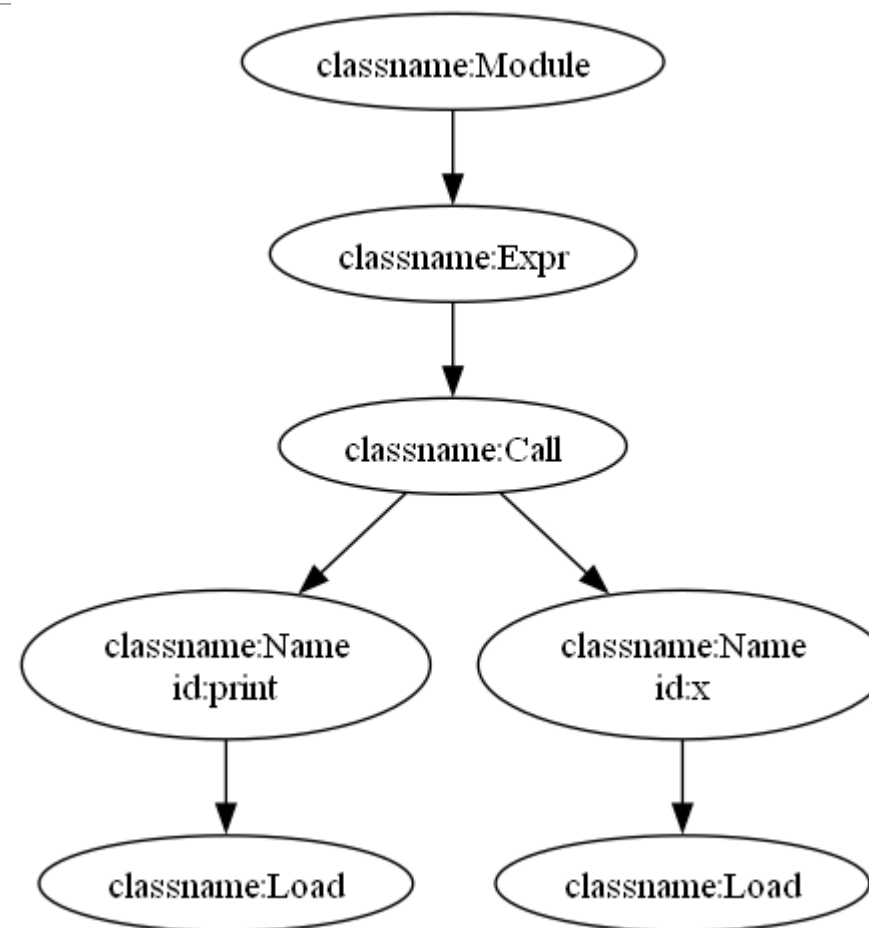
- ・抽象構文木だと、プログラムに影響を与えない空白や改行が入っても一致しているものとみなすことができる

対応している文法

- if 文とそれに伴うelif, else 文
- while 文とそれに伴うelse 文
- for 文とそれに伴うelse 文
- funcdef 文
- 式文 (関数や\$a+1\$ のような式 (expression)を文として呼び出すもの)
- 代入文
- pass,break,continue文

引数に関する制約条件の背景

- ・右図は関数printに対する呼び出し文print(x)を表すAnyTreeである
- ・この呼び出しから引数を取得するために、Callノード配下からNameノードを取得する方法を考案
- ・しかし、提案手法だと関数名printを示すNameノードまで取得してしまうため、引数ノードだけ取得することができなかった
- ・そのため、引数に変数を入れることはできなくなった



返り値に関する制約条件の背景

- ・本研究では変数同士の比較の際、その構造に関わらず、単に変数名のみでその変数が同一であるかどうかを見ていた。
- ・変数の構造を比較を実現するために、POM/EQでは変数依存グラフ(PDG)を構築するツールを活用していた。
- ・本研究でも活用を試みたが、Python向けの構築ツールは確認できず断念した。

評価のために用いたプログラムの例

以下に示すプログラムの組は「構造的一致」となるプログラムの組であり、この組に対して「構造的一致」と判定できることを確認している

```
1  count=0
2  for i in [1,2,3,2,3]:
3      if i==2:
4          count+=1
5      else:
6          pass
7
8  print(count)
9
10 count=0
11 for i in [1,2,3,2,3,4,2,3]:
12     if i==3:
13         count+=1
14     else:
15         pass
16
17 print(count)
```

```
1  def count_num(list_,num):
2      count=0
3      for i in list_:
4          if i==num:
5              count+=1
6          else:
7              pass
8      return count
9
10 print(count_num([1,2,3,2,3],2))
11 print(count_num([1,2,3,2,3,4,2,3],3))
```

(予備)判定プログラムの入出力

入力: 2つのPythonプログラム

出力: 「完全一致」、「構造的一致」、「一致していない」のいずれか