## Abstract State

### הקדמה

המצב האבסטרקטי עוטף מבנה נתונים של גרף, שלהלן הסבר עליו ועל איך מתבצעות פעולות עליו.

בגרף אנו נדרשים לאיזשהו lattice, כדי להגיד דברים כמו "תכונה מסוימת חייבת להתקיים", או "תכונה מסוימת אולי מתקיימת". ה-lattice שבו בחרנו להשתמש נראה כך:

Top

|

|

MAY\_HAVE

/ \

/ \

/ \

/ \

MUST\_HAVE MUST\_NOT\_HAVE

\ /

\ /

\ /

\ /

\ /

Bottom

### תיעוד הגרף

כל קשת בגרף מכילה:

* מי הצמתים שביניהם היא מחברת (אב ובן)
* התווית שלה
* רמת הידע שיש לנו עליה- איבר ב-lattice (שבפועל יהיה L\_MUST\_HAVE או L\_MAY\_HAVE. יכול להיות גם L\_MUST\_NOT\_HAVE, אבל כרגע אין משהו שגורם לזה).

כל צומת בגרף מכיל:

* אינדקס
* אינדקס של קבוע שהאובייקט שווה אליו (על מנת לא לשמור הופעות מרובות של קבועים מאותו הטיפוס).
* רשימה של בנים. מכל תווית יכול להיות לכל היותר בן אחד.
* רשימה של אבות. מכל תווית יכולים להיות הרבה אבות.
* רמת הידע על הצומת- איבר ב-lattice, שבפועל יהיה L\_TOP או L\_MUST\_HAVE, בהתאם להאם הצומת הזה ידוע כ-TOP או לא.
* mutabillity- איבר ב-lattice. בפייתון, לא לכל משתנה אפשר להוסיף אטריבוטים. בשדה זה נשמור את רמת הידע שלנו על חוקיות הוספת האטריבוטים למשתנה זה.
* callability- חלק מהמשתנים שנעקוב אחריהם יהיו פונקציות, שלעתים נרצה לקרוא להן. בשדה זה נשמור את רמת הידע שיש לנו לגבי חוקיות הקריאה של משתנה זה.
* metadata- שדה חופשי שמאפשר לשמור קבוצה של פריטי מידע אפשריים. בפועל, נשתמש בשדה זה כדי להחזיק את כל המימושים האפשריים של פונקציה מסוימת.

הגרף:

* הגרף הוא DAG, עד כדי לולאות עצמיות.
* יש שורש לגרף, הקודקוד 0.
* הידע שיש לנו על attribute מסוים של צומת כלשהו, שמור בבן של הצומת בקשת שהתווית שלה היא שם ה-attribute. (נדבר בהמשך על מקרים שבהם שומרים ידע בצורה implicit, ואז לא בטוח שיהיה בן).

דוגמא לגרף שמייצג את הקוד הבא:

a = 5

b = a.real

f = a.\_\_add\_\_.\_\_call\_\_

**0**

/|\

/ | +--------+

real

root#b

\_\_add\_\_

\_\_call\_\_

root#a

root#f

root#b / | \

+-------+ | root#a |

| | |

| **1** | root#f

| / \ |

\ / \ \_\_add\_\_ |

\ real/ \ |

\ / \ +

\ / \ /

**2** **3** /

| /

\_\_call\_\_ | /

| /

**4**

בכל אחד מהצמתים מוחזק מידע לגבי הקבוע שבו- בצמתים 1 ו-2 זוכרים שיש שם int, ובצמתים 3 ו-4 זוכרים שיש שם method-wrapper.

פונקציות עיקריות בגרף, שבאות להמחיש את רוח הדברים:

* השמה של משתנה לקבוע-
  + בשלב ראשון, בודקים האם אביו של הצומת (אם קיים) הוא mutable, ואם לא (או אם עלול לא להיות), מדווחים על כך שגיאה (או התראה).
  + לאחר ניתוק הצומת מבניו הישנים (אם קיימים כאלה), מסמנים בעבור הצומת:
    - האינדקס של הקבוע של הצומת יצביע ל-type של הקבוע הנתון.
    - שדה ה-metadata מתנקה.
    - שדה ה-callable מתעדכן בהתאם לערך של הקבוע (האם הוא פונקציה או לא)
    - שדה ה-mutable מתעדכן בהתאם לערך של הקבוע.
    - הצומת נהיה מסומן כ-"לא TOP"- כלומר הידע שיש לנו עליו יהיה L\_MUST\_HAVE.
* הוספת צומת חדש-
  + דורש תווית ואב.
  + אם כבר קיים לאב בן עם אותה התווית, מוחק את הקשת מהאב לבן הקודם.
* הפיכת משתנה ל-TOP (או הוספת משתנה וסימונו כ-TOP)-
  + מסמנים שהמשתנה (כלומר הצומת) הוא TOP.
  + מסמנים שלצומת אין קבוע שהוא שווה לו.
  + הופכים את שדה ה-mutable להיות BOTOM.
  + מסמנים את שדה ה-callable בתור TOP.
  + מוסיפים ל-metadata מימוש של פונקציה מנוונת, שמחזירה משתנה שהוא TOP.
  + למרות שאפשר להחליט שהופכים את הידע של כל הקשתות להיות L\_MAY\_HAVE, החלטנו לא לעשות זאת כרגע, כי בכל התרחישים המצב כבר מטופל בפונקציות אחרות.
* בדיקה האם קיים בן- (בהינתן אב ותווית)
  + בן יכול להיות קיים באחד מ-3 מצבים:
    - הקשת אליו קיימת
    - האב הוא TOP
    - באב מאוחסן קבוע שהבן יכול להיות attribute שלו
  + במקרה של קבוע, מכיוון שבכל צומת יכולים להיות מאוחסנים כמה קבועים, אז הפונקציה יכולה להחזיר גם "אולי קבוע", במקרה וה-attribute מופיע רק חלק מהקבועים.
* Collect garbage-
  + מוחקים מהגרף צמתים שלא ניתן להגיע אליהם מהצומת הראשי.
  + מוחקים קבועים שלא בשימוש של אף צומת.
  + מעדכנים את האינדקסים שנמצאים בשימוש (גם של הצמתים וגם של הקבועים), כך שיהיו קומפקטים.
  + זמן טוב לקרוא לפונקציה הוא לפני ביצוע פעולת lub, וזה אכן מה שקורה.
* lub-
  + הספר מפורט יותר בהמשך

### פונקציות נוספות של המצב האבסטרקטי

המצב האבסטרקטי:

* פונקציית שאילתא למשתנה/attribute מסוים- (לדוגמא root#foo#x.a.b)
  + מטיילים על הגרף לפי התוויות, בהתאם לביטוי שאותו מחפשים.
    - מגיעים באופן רקורסיבי לאבא של הביטוי
    - אם הבן קיים בצורה לא מפורשת (באמצעות שימוש בקבועים), מוסיפים אותו לגרף
    - בכל פעם שמזהים קשת שאינה MUST\_HAVE, מדווחים על שגיאה/התראה, ומעדכנים את הקשת (מוסיפים משתנה TOP במקרה הצורך)
  + הפונקציה מחזירה רשימה של שגיאות והתראות אפשריות בגישה למשתנה (כולל attribute-ים).
* השמה של משתנה לא פשוט-
  + עדכון כאשר נעשית השמה מהצורה x=y, כאשר y אינו משתנה פשוט.
    - בדומה להשמה של משתנה פשוט (שמתמודדים איתה ברמת הגרף), מנתקים את המשתנה המושם מהבנים שלו (אם הוא כבר קיים), ובודקים לגבי mutability של אביו.
    - לאחר מכן, משנים את מיקומן של קשתות בגרף כדי לעדכן את הגרף למצב החדש.
* clone-
  + החזרת מצב אבסטרקטי חדש וזהה לקיים.
* מחיקת משתנה מהמצב האבסטרקטי-
  + מטפסים מהצומת שמתאים למשתנה הזה ועד לצומת 0. אם מתישהו ביקרנו בצומת שיש לו יותר מאב אחד, לא עושים כלום. אחרת, מוחקים מהגרף את הקשת הראשונה במעלה המסלול (העמוקה ביותר).
* שמירת רישום של מתודה כלשהי לאובייקט מסוים-
  + כאשר מאתחלים אובייקט מסוים להיות ממחלקה מסוימת, צריך לשייך לו את המימושים של המתודות שיש לו.
* אחזור המימוש של מתודה מסוימת-
  + הפעולה ההפוכה לפעולה הקודמת.

### Lub בין מצבים אבסטרקטים (בין גרפים)

פעולת ה-lub בין גרפים היא הפעולה המורכבת ביותר שמתבצעת על שני מצבים אבסטרקטים.

* נשנה את שמות הצמתים והקבועים, כך שלא יהיו שמות זהים (פרט לשורש שיישאר 0 בשני הגרפים).
* נרצה למצוא את הצמתים שהנתיב אליהם בשני הגרפים הוא זהה.
  + כדי להתמודד עם צמתים שמופיעים בצורה explicit בגרף אחד, ו-implicit בגרף אחר (בנים של קבועים ידועים או של TOP), נשתמש בפונקציה fill\_graphs, שפועלת על שני גרפים ויוצרת בצורה explicit צמתים שמופיעים בגרף אחד בצורה explicit ובגרף האחר בצורה implicit.
  + אחרי שהרחבנו את הגרפים כדרוש, נעבור ב-BFS על הגרף שצמתיו הם וקשתותיו הן:
* בין כל הקודקודים המשותפים לשני הגרפים, נעשה vertex\_lub:
  + נעשה lub לשני ערכי ה-knowledge של הצמתים (אם צומת אחד הוא TOP, אז כך יהיה גם האיחוד שלהם)
  + נאחד את קבוצת הקבועים האפשריים לכל קודקוד.
  + נאחד את קבוצת מימושי הפונקציות (metadata) של כל קודקוד.
  + נעשה lub בין ערכי ה-mutable וה-callable של שני הקודקודים.
* נעבור על הקשתות המשותפות שמצאנו בשלבים הקודמים, ונעשה lub בין ערכי ה-knowledge שלהן.
* נוסיף לגרף הראשון את הגרף האחר, כאשר כל הקשתות החדשות יהפכו להיות MAY\_HAVE, והצמתים יישארו ללא שינוי.
* נשנה את קשתות הגרף הראשון שלא מופיעות בגרף השני להיות MAY\_HAVE.

#### איחוד צמתים

פעולת ה-lub שתוארה אינה שלמה. נסתכל למשל על שני הגרפים הבאים:

a

b

x

a

c

b

x

הצומת שמסומן "1" בשניהם הוא זהה, כי הוא מתאים למשתנה x. המשתנה "a.b" מתאים בגרף הימני לצומת "2", ובגרף שמאלי לצומת "1". זה אומר שנצטרך לאחד בין הצמתים "1" ו-"2"!

פעולת האיחוד מזכירה קצת איחוד בין שני גרפים שונים- על הצמתים נעשה vertex\_lub, בין קשתות משותפות לשניהם (הורים או בנים) נעשה lub לידע שלנו עליהן, וקשתות שמופיעות רק באחד מהצמתים נהפוך להיות בעלות ידע MAY\_HAVE.

### דילמות בתכנון המצב האבסטרקטי (TODO)

#### בן יחיד לכל תווית

המצב האבסטרקטי יכל להיות מדויק יותר, לו היינו יכולים לכל צומת לשמור **כמה** בנים עם אותו ה-label. אנחנו בחרנו (על מנת לפשט את האבסטרקציה) לבחור בבן יחיד לכל label.

איפה ניתקל בבעיה זו באופן מפורש? כשנצטרך לאחד צמתים בעת ביצוע פעולת lub בין הגרפים.

נסתכל על קטע הקוד הבא (דוגמא 22):

1 class A(object):

2 def \_\_init\_\_(self):

3 self.a = 1

4

5 class B(object):

6 def \_\_init\_\_(self, x):

7 self.a = x

8 self.b = 1

9

10 a1 = A()

11 a2 = A()

12 l = [a1, a2]

13 l.append(B(a1))

14

15 for x in l:

16 x.a

17 x.b

18 x.c

מה שהיינו רוצים שיקרה זה ששורה 16 לא תדווח על שגיאה, שורה 17 תדווח על התראה (וששורה 18 תדווח על שגיאה).

נסתכל על שני המצבים האבסטרקטים שעליהם עושים lub בין שורות 12 ל-13: (הקשת המקווקת מסמנת את l\_vars\_lub)

a

root#b

b

\_\_init\_\_

a

\_\_init\_\_

root#a1  
root#a2

**10**

**12**

**11**

a

root#b

b

\_\_init\_\_

a

\_\_init\_\_

root#a1  
root#a2

גם לקודקוד "1" וגם לקודקוד "9" ניתן להגיע ע"י root#l\_vars\_lub.a. קודקוד "9" וקודקוד "3" גם כן אמורים להיות אותו הקודקוד, כי לשניהם אפשר להגיע ע"י root#a1.

לכן נצטרך לאחד את קודקודים "1" ו-"3" בגרף הימני.

לצומת "3" יש בן עם התגית "a" (הצומת "1"), אבל לצומת "1" אין בן עם התגית "a". לכן התוצאה תהיה שמהצומת המאוחד תהיה קשת עצמית עם התגית "a", ועם הידע MAY\_HAVE:

a

כתוצאה מכך, הידע שיהיה לנו על ה-attribute "a" של איבר כלשהו ברשימה l הוא MAY\_HAVE.

### פונקציות מובנות

בפייתון קיימות הרבה פונקציות מובנות ופשוטות. כמה דוגמאות:

* \_\_add\_\_ לחיבור בין שני מספרים
* \_\_len\_\_ לחישוב אורך של מחרוזת
* \_\_isalpha\_\_ לבדיקה האם מחרוזת היא אלפא-נומרית

התבלטנו האם להוסיף תמיכה לפונקציות מהצורה הזו, מכיוון שמדובר בחישוב המזכיר קומפילציה, דבר שיוצא מה-scope של עבודתנו.

בכל זאת, בהרבה מקרים החישוב יהיה מאוד פשוט, וחבל יהיה להכריז על משתנה שהוא TOP.

ההתמודדות שלנו היא לשמור בנק של פונקציות מוכרות, ולסווג אותן לפי טיפוס החזרה. כאשר קוראים לפונקציה מסוימת של משתנה שאנחנו יודעים בוודאות מה הקבוע שלו, אנחנו יכולים להחזיר מימוש של פונקצית dummy שרק מחזירה אובייקט מהטיפוס הנכון.

אם הפונקציה לא נמצאת בבנק שלנו, נחזיר מימוש של פונקציה שמחזירה משתנה שהוא TOP.

ההתמודדות שלנו היא לא מושלמת. למשל, הנחנו שהפונקציה \_\_add\_\_ של מספר, תמיד תחזיר מספר, ללא קשר לטיפוס של המשתנה שהועבר כפרמטר, או לכמות הפרמטרים (TypeError). לדעתנו, התמודדות מורכבת יותר לא תהיה בעלת ערך מוסף לרוח הפרוייקט.

### adaptive state

סוגיה נוספת שדורשת החלטה היא מה לעשות במקרה שבו דיווחנו על שגיאה בגישה למשתנה/attribute.

אם אנחנו מדווחים על שגיאה, יכול להיות שאנחנו צודקים ושהשגנו את מטרתנו. אבל גם יכול להיות שטעינו איפשהו, ונדווח על שגיאה בכל גישה למשתנה, דבר שיגרום ל"טעות נגררת" ולדיווח על מספר רב של שגיאות מיותרות.

הדרך שבה בחרנו היא שבכל פעם שמדווחים על שגיאה, להוסיף את המשתנה שגרם לשגיאה בתור TOP. אם קיבלנו התראה- לשנות את הידע של הקשתות להיות MUST\_HAVE.

בצורה כזו, אם קיים באג בקוד הנבדק, נמצא אותו רק בפעם הראשונה שהוא מופיע, אבל זה אמור להיות מספיק טוב, ולא נגרור איתנו טעויות.

דוגמא נוספת היא כשנעשתה השמה של קבוע מוכר ל-attribute של משתנה לא מוכר. כלומר:

unknown\_var.x = 5

במקרה זה, נסמן את unknown\_var בתור TOP, וכך גישה ל-unknown\_var.y לא תחזיר שגיאה.

אבל את x לא נסמן כ-TOP, אלא כקבוע.

כמובן שגם נדווח על שגיאה עבור המשתנה unknown\_var.

#### פונקציות

בצורה אנאלוגית להפיכה ל-TOP של משתנים לא מוכרים, כך נעשה גם עם פונקציות.

עם פונקציות העבודה היא כפולה- אנחנו יכולים לדווח על שגיאה גם אם אנחנו חושבים שה-attribute הוא לא callable, וגם כאשר אנחנו חושבים שאנחנו לא מכירים את המימוש לפונקציה.

התיקון כאן יהיה לשנות את המשתנה להיות callable, וגם להחזיר מימוש לפונקציית dummy שמחזירה משתנה שהוא TOP.

### שמירת קבועים בגרף

לכל משתנה שאנחנו יודעים את הקבוע שאליו הוא שווה, נרצה לשמור את הקבוע. כדי לחסוך במקום, לא נשמור כמה קבועים מאותו ה-type. כדי לחסוך עוד יותר במקום, אם הבן הוא attribute של הקבוע המקורי, אז בעצם לא צריך לשמור את הקבוע של הבן, אלא רק של האב. כך אפשר לעבוד בצורה שהיא יותר lazy.

זה דורש לפעפע את הקבוע במורד העץ, כאשר האב מתנתק מבניו.

החסכון הוא קטן מאוד, ודורש התמודדות עם יותר מקרים, ולכן זנחנו את הגישה הזו.

* איחוד צמתים
* בן יחיד לכל תווית
* פונקציות built-ins
* Query שמתקן את עצמו, כולל add\_var\_and\_set\_to\_top
* בנים של TOP
* קריאה לפונקציה שמתקנת את עצמה
* שמירה של בנים בצורה implicit
* Fill graph
* פעפוע של קבוע מאב לבן
* דוגמאות:
* Callable, mutable
* פונקציה שמחזירה TOP

## Simpler

כפי שראינו בשיעור, תוכנית האימות כשלעצמה מיועדת להתמודד עם syntax בסיסי. עם זאת, בעולם התוכנה, התוכניות נכתבות ע"י שימוש בsyntax מורכב מטעמי נוחות וחסכון.

לכן, בהינתן תוכנית, מטרתינו הראשונית היא לפשט אותה כך שהפונקציונאליות נותרת זהה, אך המאמת יכול להתמודד עמה.

### טיפול בIMPORTים

על פי עקרונות המודולוזיציה, מבנה סטנדרטי של ספריית פיית'ון הינו מהצורה:

Main\_project\_directory/

\_\_init\_\_.py

Secondary\_section\_1/

\_\_init\_\_.py

Code1.py

Code2.py

…

באופן טבעי, כל מודול מכיר את עצמו, ועל מנת שיכיר מודולים אחרים נעזרים בimport. ישנן שתי צורות לבצע את הפעולה:

From a import b

Import c

a יכול להיות ספריה או מודול, b יכול להיות מודול או איבר מוגדר (מחלקה, פונקציה, משתנה גלובאלי) בתוך a, c חייב להיות ספריה.

המטרה היא להפוך ספריה המורכבת ממספר מודולים, וביניהם ביטוי import למודול בודד.

ראשית, נעבור על המודול ונחליף ביטוי import z לביטויים מהצורה from x import y על פי החוקים הבאים:

* אם z הוא ספריה אז הביטוי מייבא את הקובץ \_\_init\_\_.py ולכן הfrom הרצוי הוא from z import \_\_init\_\_
* אם z הוא מהצורה a.b אנחנו בודקים האם יש ספריה בשם a שבתוכה קובץ בשם b.py. במידה וכן, אנחנו הופכים את הביטוי לfrom a import b
* אחרת – b הוא יבטוי מוגדר בתוך a ואנחנו הופכים את הביטוי לfrom a import \*

לאחר מכן, אנחנו מתמודדים עם syntax חסכוני של פיית'ון; הביטוי from **.**a import b מציין "מהספריה הנוכית שבה אנו נמצאים יבא את a.b". אנחנו הופכים את האימרה הזאת להיות explicit.

לאחר ששינינו את הsyntax לכזה שנוח מבחינת הההתמודות שלנו, אנו בונים גרף תלויות. המטרה היא להבין את מערכת התלויות במודול ואז לפלוט לקובץ בודד את כל הקוד. אנו עושים זאת באמצעות בניית DAG שבו כל מודול מצביע למודול שהוא תלוי בו (ההנחה שהיא שניתן להריץ את התוכנית, ולכן אין import מעגלי).

לאחר שבנינו את הDAG אנו פולטים לקובץ אחוד את העלים, ואז עולים במעלה העץ עד אשר עוברים על כל המודולים.

לסיכום, אנחנו מסירים מהקוד את כל האיזכורים לחבילות הללו (לדוגמא כשעושים import a אז כשנרצה להשתמש בפ' מתוך a הסינטקס הנכון יהיה a.foo(). כיוון שהעתקנו את foo לקובץ אותו אנו מריצים הקריאה הנכונה צריכה להיות foo().

### פישוט הקוד

הפישוט עובד בצורה איטירטיבית כך שבכל איטרציה מתבצע תיקון בודד. כל עוד ביצענו פישוט באיטרציה הנוכחית אנחנו נמשיך לפישוט נוסף, כך שהקוד מבצע פישוטים מורכבים לפישוטים בסיסיים שהתבצעו. הקוד בנוי כך שהפישוטים מתבצעים לפי סדר שיוצר קוד נכון וקריא.

הפישוטים האפשריים:

* השמת tuple: בפיית'ון ניתן לבצע השמה לtuple, כך שהאיבר הראשון בחישוב מתמפה לאיבר הראשון, האיבר השני לשני וכו'. הפישוט מוציא את תוצאת החישוב למשתנה טמפלייטי ואז מתבצעת השמה איבר איבר.
* השמה לאיבר מורכב: איבר מורכב משמעותו פניה לאטריביוט פנימי (אטריביוט של אטריביוט, לדוגמא a.b.c), במצב כזה אנחנו עושים מוציאים את a.b למשתנה זמני, ואז מבצעים השמה לtmp.c. הסיבה שבגללה אנחנו לא מפשטים כך שתמיד תתבצע השמה למשתנה בודד היא מכיוון שניתן לייצר אטריביוטים בצורה דינמית, לדוגמא השמה a.x היא חוקית גם אם x אינו קיים, אך לא ניתן לבצע השמה tmp = a.x. כמו כן, על מנת לייצר את המשתנה a.x.y, על x להיות קיים.
* פניה לאטריביוטים: פניה לאטריביוט תמיד תתבצע ממשתנה ולא מאטריביוט. כלומר פניה של a.b.c אינה חוקית, אך פניה מהסוג tmp = a.b , tmp.c הינה חוקית. נשים לב שהדבר נכון מבחינה סמנטית כי אם a.b הוא אוביקט אזי לאחר השמה למשתנה זמני שניהם יפנו לאותו מקום בזיכרון, ואם הוא קבוע (לדוגמא מספר או מחרוזת) אזי שניהם מצביעים לאותו איבר בבריכת הקבועים.
* פישוט אריתמטי: ביטויים אריתמטיים מורכבים יהפכו לביטויים בינאריים תוך שימוש במשתנים זמניים
* CALL ARGS: הפעלת פונקציה תיעשה רק מפרמטרים שהם משתנים
* ביצוע אופרטור והשמה: פעולות כמו +=, -= וכו' מתורגמות לשתי שורות קוד – השמה למשתנה זמני של תוצאת החישוב והשמת המשתנה למטרה.
* איטרציה בלולאות For:הוצאת האיבר שעליו מתבצעת האיטרציה למשתנה.
* אופרטורים בינאריים: אופרטורים בינאריים הם בעצם פונקציה שקוצרה ע"י סוכר תחבירי. במילים אחרות, הsyntax a OP b (OP הוא פעולה אריתמטית) הוא סוכר תחבירי להפעלת הפונקציה בצורה הבאה a.\_\_OP\_\_(b).
* ערך return: פישוט הReturn Value כך שתמיד יהיה משתנה.
* פישוט list, dict, tuple*:* כל אחד מאבריהם הינו משתנה.

## Visitors

מטרתה של מערכת האימות היא הערכת הקוד וזיהוי שגיאות העלולות לעלות בזמן ריצה (runtime). על מנת לעשות זאת הוגדר עבורה אוסף חוקים עבור הפעולות המותרות והאסורות בקוד. כדי לבדוק האם קוד נתון עומד באוסף החוקים המותר יש צורך בעיבוד הקוד לכדי מבנה נתונים המאפשר לבדוק אותם. בהינתן אוסף זה יש צורך במנוע אשר יעבד את הקוד ויוודא אותו. המנוע אשר נכתב מורכב מתתי מנועים אשר כולם מתבססים על גבי תשתית ה-ast visitor של פייתון.

### אוסף החוקים – מותר ואסור

החוקים אשר הגדרנו מהווים תת-אוסף (subset) של שפת פייתון. הם נבחרו כך שנוכל לעבד נאמנה קוד פייתון בעל תחביר (syntax) סטנדרטי ובעל שימוש סטנדרטי במאפייני השפה. כחלק מהרצון לתמוך גם בתוכניות מורכבות יותר נכתב גם רכיב ה-Simpler אשר מתואר בפרק משל עצמו.

חוקי השפה:

* אסור לבצע השמת attributes למשתנים פשוטים (int, str, tuple, list, …)
* אסור לבצע השמה שמקורה במשתנה לא מוגדר
* אסור לקרוא ערך של משתנה לא מוגדר
* אסורה ירושה מרובה
* מה עוד?

### מנוע העיבוד

על מנת לעבד קוד נתון המערכת מניחה הנחות מקדימות:

* כלל הקוד כתוב בקובץ אחד ( תפקידו של ה-simpler)
* כלל הקוד עובד ל-Abstract Syntax Tree (תפקידו של מנוע ה-ast)

בהינתן ההנחות המקדימות המנוע עובר בצורה רקורסיבית מעל גבי ה-AST ומנסה לזהות פעולות בקוד אשר עלולות לעבור על האיסורים שהוגדרו. המנוע משתמש בתתי מנועים הנקראים Visitors ותפקידם הוא לעבד אזור מסוים בעץ (ה-AST) ולזהות בו דפוסים (כל Visitor מקבל אחריות אחרת).

### ProgramVisitor

מנוע ה-Visitor הראשי. תפקידו הוא לעבור על רצף השמות, הגדרות משתנים, הגדרות פונקציות ועוד מבני בקרת זרימה, לעבד אותם ולהפעיל את ה-Visitors האחרים בהתאם לתפקידם.

בתחילת ריצתו המנוע מאתחל את מבני הנתונים איתם נעבוד לאורך הדרך:

* Stack – מבנה נתונים המתאר את מחסנית זמן הריצה ומורכב ממסגרות (frames) המכילות מידע על המשתנים שמוגדרים ברמה המתאימה להן במחסנית.
* AbstractState – מבנה הנתונים (המתואר בפרק משל עצמו) המכיל את כל הידע על הקשרים בין משתנים.
* Functions – מידע על פונקציות שהוגדרו לאורך הדרך.
* Classes – מידע על פונקציות שהוגדרו לאורך הדרך.

המנוע מסוגל להתמודד עם הצמתים הבאים בעץ:

* הגדרת מחלקה

בדיקת ירושה מרובה  
הפעלת ה-ClassDefVisitor

* הגדרת פונקציה

הפעלת ה-ClassDefVisitor

* הפעלת ביטוי

הפעלת FunctionDefVisitor

* ביצוע השמה

הפעלת AssignVisitor

* לולאת For

מחושב lub מעל גבי איברי הרשימה ולאחר מכן גוף הלולאה מחושב פעם אחת עבור ה-lub.

* בלוק if-else

עבור כל תת בלוק משוכפל מבנה הנתונים AbstractState ומורץ גוף הבלוק על גבי המבנה המשוכפל. לאחר מכן אנו מבצעים פעולת lub בין מבנה הנתונים המקורי (מלפני הכניסה לשרשרת ה-if-else) ולבין המבנה הנוכחי. זאת מכיוון שאין אנו יודעים אילו מתתי הבלוקים ירוצו (אם בכלל).

* בלוק try-except-finally

מכיוון שאיננו יודעים איזה חלק מבלוק ה-try ירוץ עד לזריקת שגיאה (אם בכלל) אנו משכפלים את מבנה הנתונים עובר כל רצף הרצות מתחילת הפונקציה ומבצעים lub שלו עם ה-AbstractState המקורי מלפני הכניסה לבלוק.

* הפעלת return

חישוב ערך החזרה אשר צריך לחזור מהפונקציה.

### ClassDefVisitor

מעבד הגדרת מחלקה בעץ ה-AST. מכיוון שהגדרת מחלקה אינה כוללת "הרצת" קוד המידע היבש נשמר במבני נתונים מתאימים ומחושב (קונסטרקטור, פונקציות) רק כאשר אחד מהמנועים מזהה הפעלה שלהם.

* חישוב ירושה

המנוע מזהה את עץ הירושה של האובייקט ושומר אותו במבנה נתונים מתאים על מנת שנוכל בהמשך לבצע פעולות בהתאם.

* זיהוי פונקציות

המנוע מפריד בין פונקציות סטטיות לפונקציות של האובייקט ומאחסן גם אותן במבנה נתונים מתאים.

### FunctionDefVisitor

Visitor בסיסי אשר מטרתו היא רק שמירת הגדרת פונקציה ושיוכה לשם המתאים לה לשם שימוש ע"י שאר המנועים.

### AssignVisitor

אחראי על ביצוע ההשמות לתוך משתנים/attributes. מנוע זה מקבך צומת העץ ה-AST ומחשב את ערכי ההשמה הרלוונטיים אליה בהתאם למידע אשר מזוהה בצומת. בהתאם לכך השמה יכולה להיות:

* משנה אחר (x=y)
* ערך attribute (x = y.a)
* string (x="hello")
* מספר (x=2)
* רשימה (x=[1,2,3])
* Tuple (x=(1,2,3))
* מילון (x={1:2,3:4})

### CallVisitor

אחראי על קריאות לפונקציות (כולל קונסטרקטורים). כאשר מתבצעת קריאה תפקידו הוא לזהות את סוג הקריאה (קונסטרקטור, פונקציה, פונקציה של אובייקט (מחלקה)) ולהפעיל את הלוגיקה המתאימה לביצוע המשך הפירסור. כאשר הפונקציה המופעלת שייכת לאובייקט (כלומר זוהי פונקציה של מחלקת אב שלו) תפקידו של מנוע זה הוא לאתחל את אובייקט ה-self לפני הקריאה לפונקציה על מנת שקריאת ורישום ערכים אליו תישאר עקבית גם בתוך גוף הפונקציה.

מקרה מיוחד הנו כאשר לאובייקט יכולים להיות מספר מימושים עבור אותה הפונקציה. מקרה זה מתרחש כאשר אובייקטים שונים עברו lub והייתה להם פונקציה בעלת שם זהה. במקרה זה נחשב הרצה של כל אחד מהמימושים ולאחר מכן נבצע lub על התוצאות.