## Abstract State

### פונקציות ממשק של AbstractState

המצב האבסטרקטי:

* תומך ב:
  + query- האם קיים משתנה מסוים
  + set\_var\_to\_var- עדכון כאשר נעשית השמה מהצורה x=y, כאשר y אינו משתנה פשוט
  + set\_var\_to\_const- עדכון כאשר נעשית השמה מהצורה x=y, כאשר y הוא משתנה פשוט.
  + clone- החזרת מצב אבסטרקטי חדש וזהה לקיים
  + lub- ביצוע פעולת lub בין שני מצבים אבסטרקטים.
  + add\_var\_and\_set \_top\_top- הפיכת צומת ל-TOP, כולל את הוריו אם אינם קיימים בגרף.
  + remove\_var- מחיקת משתנה מה-state (הסרת צומת מהגרף)

### תיעוד הגרף (בשביל העבודה העיונית)

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, ואולי גם L\_MUST\_NOT\_HAVE)

כל צומת בגרף מכיל:

* אינדקס
* אינדקס של קבוע שהאובייקט שווה אליו
* רשימת שכנים (מופרד לאבות ולבנים). לכל תווית יכול להיות לכל היותר בן אחד, אבל הרבה הורים.
* רמת הידע על הצומת- איבר ב-lattice, שבפועל יהיה L\_TOP או L\_MUST\_HAVE

הגרף:

* הגרף הוא DAG (עד כדי לולאות עצמיות. כרגע אין טיפול בלולאות עצמיות)
* כל קודקוד הוא או TOP או שיש לו קבוע (צריך עוד לחשוב על זה קצת)

פונקציות עיקריות בגרף:

* השמה של משתנה לקבוע-
  + מסמנים שהקבוע של הצומת הוא ה-type של המשתנה.
  + מסמנים את הצומת כ-לא TOP (L\_MUST\_HAVE)
* הוספת צומת חדש-
  + דורש תווית ואב. אם הבן כבר קיים אצל האב, מוחק את הקשת מהאב לבן הקודם.
* הפיכה לאבא-
  + כמו הוספת צומת חדש, אבל בלי ליצור צומת חדש
* 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
* מחיקה של צומת-
  + באמצעות ניתוק כל הקשתות (אבות ובנים) של הצומת
* Consolidate (garbage collector)-
  + מוחקים מהגרף צמתים שלא ניתן להגיע אליהם מהצומת הראשי.
  + מוחקים קבועים שלא בשימוש של אף צומת.
  + מעדכנים את האינדקסים שבשימוש כך שיהיו קומפקטים
* Lub
  + הספר מפורט יותר בהמשך
* הפיכת משתנה ל-TOP-
  + מסמנים שהמשתנה (כלומר הצומת) הוא TOP
  + בנוסף, מסמנים שלצומת אין קבוע שהוא שווה לו
  + לא נוגעים בקשתות, כרגע

### תיעוד של המצב האבסטרקטי

המצב האבסטרקטי תומך ב:

* פונקציית שאילתא למשתנה מסוים
  + מחזירה שגיאות אפשריות בגישה למשתנה
* השמה של משתנה לא פשוט-
  + עדכון כאשר נעשית השמה מהצורה x=y, כאשר y אינו משתנה פשוט
* השמה של משתנה פשוט-
  + עדכון כאשר נעשית השמה מהצורה x=y, כאשר y הוא משתנה פשוט
* clone-
  + החזרת מצב אבסטרקטי חדש וזהה לקיים
* lub-
  + ביצוע פעולת lub בין שני מצבים אבסטרקטים.
* הוספת משתנה שדבר אינו ידוע עליו, וסימונו כ-TOP-
  + הקשתות עד אליו יסומנו כ-L\_MUST\_HAVE (אם הן לא קיימות)
  + אבות של המשתנה יווצרו גם כן כ-TOP אם הם לא קיימים כבר
* מחיקת משתנה מהגרף-
  + האם צריך לעשות query ולתקן את הגרף?
  + גשדגדשגדש
* המרת ביטוי לצומת בגרף-
  + מטיילים על הגרף לפי התוויות, בהתאם לביטוי
    - מגיעים באופן רקורסיבי לאבא של הביטוי
    - אם הבן קיים בצורה לא מפורשת, מוסיפים אותו לגרף
    - אחרת, מחזירים שהצומת לא קיים
* Query-
  + עטיפה של הפונקציה האחרונה
  + זוכר את כל הדברים שיכלו להשתבש (קשתות לא קיימות, קשתות עם L\_MAY\_HAVE)
  + כל attribute שיכל להשתבש, נוסף לגרף. אופציונאלי.
  + קשתות שהיו L\_MAY\_HAVE הופכות להיות L\_MUST\_HAVE.

### Lub בין מצבים אבסטרקטים

בשלב ראשון, נשנה את שמות הצמתים והקבועים, כך שלא יהיו שמות זהים (פרט לשורש שיישאר 0 בשני הגרפים).

נעבור על שני הגרפים ב-DFS במעבר ראשון, ונמצא צמתים עם אותו נתיב אליהם (וזה אומר שהם זהים). אם קיימות קשתות לבנים בגרף אחד ולא בשני, נבדוק האם הקשתות יכולות להתווסף גם בגרף השני (אם האב הוא TOP או מכיל קבוע כלשהו), ואם כן נוסיף את הקשתות ואת הבנים הרלוונטים (ונמשיך לטייל על הבנים ב-DFS).

נעבור על שני הגרפים מעבר שני (ב-DFS), ונחלק את הצמתים והקשתות ש:

* נמצאים רק בגרף הראשון
* נמצאים רק בגרף השני
* נמצאים בשני הגרפים יחד

טיפול ב:

* קשתות משותפות:
  + עושים lub בין ה-knowledge שלהן.
* צמתים משותפים:
  + איחוד של כל הקבועים שהם אולי שווים אליהם
  + Lub על ה-knowledge שלהם (אם אחד הוא TOP, אז כך יהיה גם האיחוד)
* צמתים וקשתות לא משותפים:
  + כל הקשתות הופכות להיות L\_MAY\_HAVE (כולל הקשת שמחברת צמתים "ראשיים" לצומת -1)
  + צמתים **לא** נהפכים להיות TOP.
  + צריך לשים לב מה קורה כאשר שני צמתי הקצה משותפים, אבל הקשת לא.

## 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 אקספליסית.

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.