digital whisper
Technologic Papers

Practical Python Internals
חלק א' - אופרטור ++

מאת אלי קסקי

הקדמה

זהו המאמר הראשון בסדרת מאמרים שבה אני מציג אספקטים שונים במימוש של CPython. בחרתי להוסיף את המילה "Practical" לכותרת מכיוון שלכל אורך המאמר, תוך כדי הלמידה נצלול עמוק למימוש של CPython ונבצע בפועל שינויים בשפה.

במסגרת מאמר זה אתמקד בתהליך הקומפילציה של קוד Python, ותוך כדי כך אציג איך הוספתי לשפה את האופרטור ++:

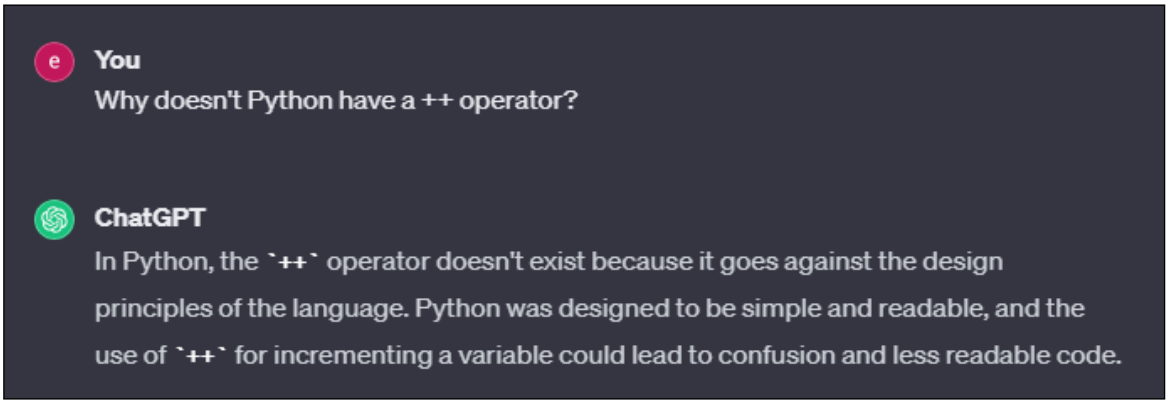
```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 5
>>> y = ++x
>>> y, x
(6, 6)
>>> y = x++
>>> y, x
(6, 7)
>>>
```

מוטיבציה

מאז ומתמיד אהבתי כל מה שקשור באיך שדברים עובדים מאחורי הקלעים. בהקשר של שפות תכנות מדובר על הקסם שקורה בתהליך המעבר בין טקסט בשפה גבוהה לבין רצף פקודות אסמבלי שהמעבד מריץ בסוף. השפה הגבוהה יכולה להיות יחסית קרובה לאנגלית ומספקת לנו אבסטרקציה מאוד חזקה, בעוד שפקודות האסמבלי שהמעבד מריץ הן בסיסיות ופרימיטיביות מאוד. במאמר זה לא אגיע עד לרמה של פקודות אסמבלי, אלא עד לרמה של פקודות Bytecode, שהן פקודות "כמו אסמבלי" של שפת Python.

במשך המון זמן רציתי להבין את שפת Python יותר לעומק וללמוד איך היא ממומשת בפועל. כדי לעשות זאת ולהוכיח לעצמי שאני אכן לומד דבר או שניים בתהליך, החלטתי להכניס שינויים מסוימים בשפה תוך

כדי הלמידה. לבסוף הצבתי לעצמי למטרה ליצור גרסאת Python משלי שבה יהיה האופרטור ++, שכידוע לא קיים ב-Python. בגלל שלדעתי יצאה תוצאה די מגניבה, החלטתי לכתוב מאמר על תהליך הלמידה שלי.



[שיחה עם ChatGPT]

אני לא השתכנעתי מהתשובה של ChatGPT, ואתם?

קצת על CPython

קיימות מספר הפצות (distributions) של Python כשההפצה ברירת המחדל היא CPython. זוהי הפצה open-source, כתובה בשפת C, והיא הפופולרית ביותר. נכון לכתיבת שורות אלה (פברואר 2024), הגרסא העדכנית ביותר של CPython היא 3.13.0 ולכן כל קטעי הקוד והשינויים שאציג במאמר זה רלוונטים לגרסא זו בלבד. יש לציין שהפרויקט CPython חי ומשתנה באופן יחסית תדיר (כ-10 commit-ים ביום בממוצע) ולכן קוד שאציג במאמר זה עלול להשתנות בעתיד. עם זאת, במאמר זה אשנה קוד שהוא בסיסי מאוד בשפת Python ולא משתנה יותר מדי, ולכן סביר שהוא יהיה רלוונטי גם בגרסאות הבאות.

לנוחיות הקורא כל השינויים שמופיעים במאמר נמצאים ב-fork שעשיתי לפרויקט CPython הרשמי בקישור.

איך לבנות את CPython

כדי לבנות את CPython יש לבצע git clone לפרויקט. אני עובד בסביבת Windows ולכן אתמקד בה, אך כמובן שניתן גם לעבוד בסביבות אחרות. הוראות הבניה לכל סביבה נמצאות בעמוד ה-GitHub של CPython. ב-Windows יש להתקין Microsoft Visual Studio 2017 או מאוחר יותר, ולהריץ את הסקריפט build.bat. לאחר מכן לפתוח את הקובץ PCbuild/pcbuild.sln ב-Visual Studio, לבחור בגרסא הרצויה (Win32/x64/ARM/ARM64 ו-Debug/Release) וללחוץ על Build Solution. תחת התיקיה PCbuild/amd64 נוצר קובץ python.exe שמכיל את הגרסא שלנו.

Bytecode על קצה המזלג

אולי שמעתם ש-Python היא שפה Interpreted ולא Compiled, אבל מה זה אומר בפועל? בשפה שהיא Compiled, כמו C למשל, הקוד עובר קומפילציה (Compilation) ישירות לפקודות אסמבלי שהמעבד יכול להריץ. לעומת זאת בשפה שהיא Interpreted, כמו Python למשל, הקוד עובר קומפילציה לפקודות Bytecode ולאחר מכן פקודות אלה מורצות בעזרת Interpreter. אפשר לחשוב על פקודות אלה כ"שפת ביניים" בין Python לבין אסמבלי שרץ על מעבד. ה-Interpreter נקרא גם "מכונה וירטואלית", ובגדול בלב הפעולה שלו הוא מכיל switch-case ענקי עם case לכל פקודת Bytecode, ומבצע אותן אחת אחרי השניה.

ה-Interpreter של Python הוא מבוסס מחסנית, מה שאומר שלצד רצף פקודות Bytecode קיימת מחסנית, וכמעט כל הפקודות דוחפות או מוציאות אובייקטים ממחסנית זו בשביל לבצע פעולות. למשל פקודת "BINARY_OP +" שתפקידה לחבר בין שני איברים, עושה זאת על ידי הוצאה של שני האיברים העליונים במחסנית, חישוב תוצאת החיבור שלהם, ודחיפת התוצאה למחסנית.

למזלנו קיימת ב-CPython ספריה בשם dis המציגה בצורה נוחה את פקודות ה-Bytecode שנוצרות עבור קטעי קוד Python. כך למשל הפקודות שנוצרות עבור הביטוי "x+1" הן:

```
Python 3.11 (64-bit)
>>> dis.dis("x+1")
  0           0 RESUME                   0

  1           2 LOAD_NAME                0 (x)
              4 LOAD_CONST               0 (1)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
>>>
```

- הפקודה LOAD_NAME דוחפת למחסנית את המשתנה x.
- הפקודה LOAD_CONST דוחפת למחסנית את הקבוע 1.
- הפקודה BINARY_OP שולפת מהמחסנית את שני הערכים הללו, מבצעת את החישוב, ודוחפת את התוצאה למחסנית.

ניתן לקרוא יותר לעומק על Bytecode ועל ה-Interpreter שב-CPython בקישור.

תהליך הקומפילציה ב-CPython

הקומפילציה ב-CPython היא תהליך של המרת קוד מקור שכתוב ב-Python לפקודות Bytecode, והיא מורכבת מהשלבים הבאים:

1. ניתוח לקסיקלי של הטקסט וחלוקה של המילים בו לרצף של Token-ים (Tokens).
   Token-ים לדוגמא הם: מספר, מחרוזת, שם (משתנה או פונקציה), אופרטור מתמטי, ועוד.
2. התאמת רצף ה-Token-ים על פי חוקי הדקדוק (Grammar) של השפה ויצירה של AST (Abstract Syntax Tree) מהם. חוקי דקדוק לדוגמא יכולים להיות:

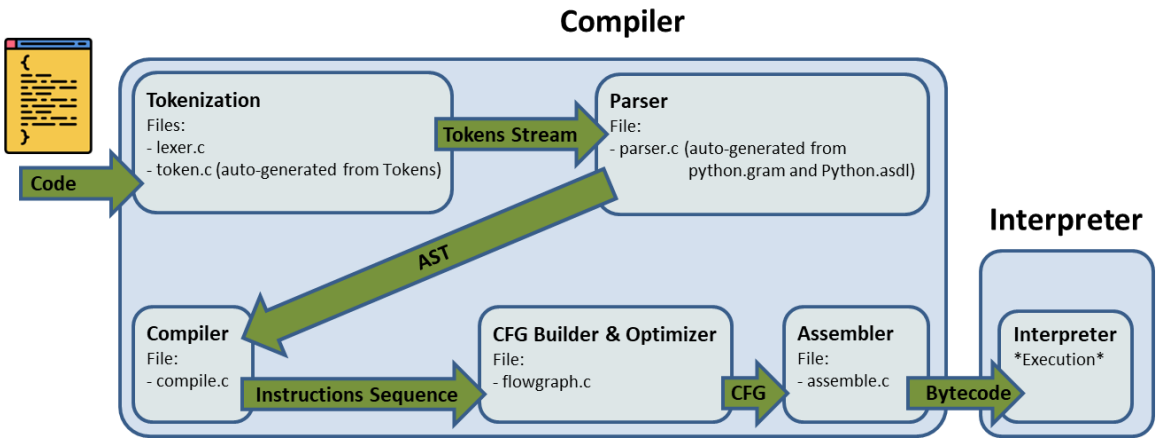
```
term: name | constant
sum: term | sum '+' term
```

בדוגמא זו החוק ששמו term יכול להתאים שם של משתנה או קבוע, והחוק ששמו sum יכול להתאים term או רצף של term, +, sum. שימו לב להגדרה הרקורסיבית. למעשה החוק sum יכול להתאים לרצף Token-ים של חיבור מספרים ומשתנים בכל אורך.

3. הפיכת ה-AST לרצף של פקודות Bytecode. בשלב זה פקודות ה-Bytecode שמסוג Jump נוצרות ללא היעד, ובהמשך פקודות אלה מתעדכנות בהתאם ל-CFG.
4. בניית CFG (Control Flow Graph) מתוך רצף פקודות ה-Bytecode. כל צומת בגרף היא רצף של פקודות Bytecode ונקראת basic block, ממש כמו בקוד אסמבלי. בשלב זה מתבצעות אופטימיזציות שונות על הגרף כמו למשל הסרה של basic blocks ריקים או שלא ניתן להגיע אליהם, הסרה של פקודות NOP וכו'.
5. חישוב offset-ים לכתובות היעד של פקודות ה-Jump מתוך ה-CFG, ועדכון ה-Bytecode שנוצר קודם לכן לקבלת ה-Bytecode הסופי.

פלט התהליך הזה הוא אובייקט מסוג PyCodeObject שמכיל קוד Bytecode שניתן להרצה. במסגרת מאמר זה לא נכנס לפרטים על אובייקט זה, אך במאמרים הבאים כן נלמד על תהליך ההרצה של ה-Bytecode.

להלן תרשים המתאר את שלבי הקומפילציה, כולל שמות הקבצים הרלוונטים:



ניתן לקרוא יותר לעומק על כל אחד מהשלבים בתיעוד הרשמי של הקומפיילר של Python.

במסגרת מאמר זה נתמקד רק בחלק של ה-Compiler, שמתואר במלבן השמאלי שבתרשים. במאמר הבא נתמקד בחלק של ה-Interpreter, שמתואר במלבן הימני שבתרשים.

הוספת האופרטור ++

Disclaimer

הקוד "++x" הוא קוד חוקי ב-Python (נסו בעצמכם!). כידוע ב-Python קיים אופרטור אונרי (unary operator) של שלילה שמתאים לתו '-' (מינוס), למשל אם x הוא מספר אז הקוד "-x" יחזיר את המספר עם סימן הפוך. באופן סימטרי קיים ב-Python אופרטור אונרי של חיוב שמתאים לתו '+' (פלוס), למשל אם x הוא מספר אז הקוד "+x" יחזיר את המספר עצמו ללא שינוי.

אני לא מכיר שימוש פרקטי לאופרטור זה ב-types הנמצאים כברירת מחדל ב-Python, אך שימוש פוטנציאלי יכול להיות ביצירת מחלקה שמממשת את הפונקציה __pos__ עם לוגיקה משמעותית (הפונקציה האנלוגית לשלילה היא __neg__). למשל אם נגדיר מחלקה mylist בדומה ל-list של Python, ונגדיר שאופרטור אונרי חיובי מחזיר רק את האיברים החיוביים ברשימה, ואופרטור אונרי שלילי מחזיר רק את האיברים השליליים ברשימה, אז האופטור יצדיק את עצמו:

```
Python 3.11 (64-bit)
>>> class mylist:
...  def __init__(self, l):
...   self.l = l
...  def __pos__(self):
...   return [n for n in self.l if n > 0]
...  def __neg__(self):
...   return [n for n in self.l if n < 0]
...
>>> x = mylist([42, -10, 0, 12, -6])
>>> +x
[42, 12]
>>> -x
[-10, -6]
>>>
```

בכל מקרה, הקוד "++x" מפעיל את האופרטור האונרי החיובי על x, ואז מפעיל את האופרטור האונרי החיובי על התוצאה. השינוי שנכניס לשפה יגרום לכך שכל הופעה של "++" בקוד תיחשב כאופרטור החדש שלנו.

הוספת Token מתאים

השלב הראשון בתהליך הוא הוספה של Token עבור האופרטור ++. הקובץ שמכיל את כל ה-Token-ים הוא Grammar/Tokens, ובו נוסיף שורה עבור ה-Token של ++:

```
15   SEMI                                                    ';'
16   PLUS                                                    '+'
17   PLUSPLUS                                                '++'
18   MINUS                                                   '-'
19   STAR                                                    '*'
20   SLASH                                                   '/'
```

הערה טכנית: ישנם קבצים מסוימים ב-CPython, כמו למשל קובץ זה, שמג'ונרטים מהם קבצי c. ו-h. שונים. לאחר שינוי קבצים אלה בסביבת Windows, יש להריץ את הסקריפט שאחראי לג'נרט את קבצי ה-c. ו-h. עם הפקודה:

```
build.bat --regen
```

כדי לבדוק את השינוי ניתן להשתמש בספריה tokenize שמקבלת קוד Python ומדפיסה את ה-Token-ים שנמצאים בו. לאחר הוספת השורה וקימפול של CPython רואים שאכן כשנתקלים בקוד ב-"++", כבר לא מדובר באופרטור אונרי חיובי שמופיע פעמיים, אלא באופרטור אונרי חדש.

לפני השינוי:

```
C:\Windows\System32\cmd.exe - python.exe -m tokenize

C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64>python.exe -m tokenize
x++
1,0-1,1:            NAME           'x'
1,1-1,2:            OP             '+'
1,2-1,3:            OP             '+'
1,3-1,4:            NEWLINE        '\n'
```

אחרי השינוי:

```
C:\Windows\System32\cmd.exe - python.exe -m tokenize

C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64>python.exe -m tokenize
x++
1,0-1,1:            NAME           'x'
1,1-1,3:            OP             '++'
1,3-1,4:            NEWLINE        '\n'
```

עדכון דקדוק השפה

השלב הבא הוא להגדיר חוק דקדוק (Grammar) שיצור אובייקט AST כשנתקלים ב-Token החדש שלנו. שלב זה מתחלק לשני חלקים:

1. להגדיר במדויק איזה רצף של Token-ים אנחנו רוצים להתאים
2. להגדיר לכל רצף שכזה איזה סוג אובייקט AST אנחנו רוצים ליצור

החלק הראשון הוא קל - אנחנו יודעים שאנחנו רוצים להתאים את המקרה של "x++" ואת המקרה של "++x". הקובץ שבו מוגדרים חוקי הדקדוק הוא Python.gram, זהו קובץ די גדול שמכיל המון חוקים ולמעשה והוא בעצם מגדיר את כל שפת Python. ניתן לקרוא על איך הוא בנוי ועל ה-syntax לכתיבת החוקים בתיעוד שנמצא בתחילת הקובץ.

בשביל לבצע את החלק השני כדאי להתבסס על חוק של אופרטור אחר שאנחנו יודעים שדומה לאופרטור שלנו. בפעולת ++ שלנו אנחנו רוצים לקרוא ערך של משתנה, לכתוב ערך בחזרה לאותו משתנה, וגם להחזיר ערך. זה קצת מזכיר את האופרטור walrus (":=") שהתווסף ל-Python בגרסא 3.8 ומאפשר לחשב ביטוי, לכתוב את התוצאה למשתנה, וגם להחזיר את התוצאה. להלן החוק של אופרטור זה:

```
assignment_expression[expr_ty]:
    | a=NAME ':=' ~ b=expression {
        CHECK_VERSION(expr_ty, 8, "Assignment expressions are",
        _PyAST_NamedExpr(CHECK(expr_ty, _PyPegen_set_expr_context(p, a, Store)), b, EXTRA)) }
```

חוק זה מתאים את הרצף "NAME := expression" ויוצר ממנו אובייקט מסוג NamedExpr, שניתן לראות את הגדרתו בקובץ pycore_ast.h. אובייקט זה מקבל כפרמטר אובייקט target ואובייקט value.

אובייקט ה-value הוא האובייקט שמתאים ל-expression שנמצא מימין לאופרטור, והוא יכול להיות מכל סוג. בחוק הדקדוק הזה ספציפית, אובייקט ה-target הוא אובייקט מסוג Name. אובייקט זה מקבל כפרמטר את שם המשתנה, ו-context שבמקרה זה הוא Store (ויכול להיות Load/Store/Del).

סוגי אובייקטים אפשריים נוספים של expression יכולים להיות Name - קריאה או כתיבה למשתנה, BinOp - פעולה בינארית כמו ביטויים מתמטיים למשל, Call - קריאה לפונקציה, ועוד. ניתן למצוא את רשימת סוגי ה-expression-ים והשדות שלהם בקובץ pycore_ast.h. ניתן לראות באופן נוח את ה-AST שנוצר בחוק זה באמצעות הספריה ast:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> import ast
>>> code = "(x := y + 1)"
>>> print(ast.dump(ast.parse(code).body[0], indent=4))
Expr(
    value=NamedExpr(
        target=Name(id='x', ctx=Store()),
        value=BinOp(
            left=Name(id='y', ctx=Load()),
            op=Add(),
            right=Constant(value=1))))
```

הגדרת אובייקט AST חדש

נרצה ליצור סוג אובייקט חדש של expression שייצג את חוק הדקדוק של האופרטור שלנו. אנחנו רוצים שהשדות של האובייקט יהיו שם המשתנה שאנחנו רוצים לשנות, ושדה op שיגדיר האם אנחנו מבצעים פעולת Pre-increment או Post-increment (כלומר האם מבצעים x++ או ++x), בדומה לשדה ה-context של הסוג Name. אובייקטי ה-AST מוגדרים בקובץ Python.asdl וממנו מג'ונרטים קבצי h. שונים בתהליך הקומפילציה של CPython. נוסיף לקובץ זה הגדרה לאובייקט שלנו, ונקרא לו UnaryExpr:

```
60      expr = BoolOp(boolop op, expr* values)
61           | NamedExpr(expr target, expr value)
62           | UnaryExpr(expr target, unaryexp_op op)
63           | BinOp(expr left, operator op, expr right)
```

נוסיף גם הגדרה ל-op שלנו, נקרא לו unaryexp_op והערכים האפשריים שלו יהיו Pre_inc ו-Post_inc:

```
99      expr_context = Load | Store | Del
100
101     unaryexp_op = Pre_inc | Post_inc
102
103     boolop = And | Or
```

בזאת סיימנו להגדיר את אובייקט ה-AST שלנו, ועכשיו נוכל לכתוב את החוק החדש שלנו באופן מלא. לפני שנשנה את הדקדוק, כדאי לציין שבתיעוד הרשמי של CPython ישנו Checklist של קבצים שיש לערוך כשמבצעים שינויים לדקדוק השפה.

ניצור חוק unaryexp שיתאים רצף Token-ים שמכיל שם משתנה שאחריו ++, וגם רצף שמכיל ++ שאחריו שם משתנה. החוק ייצור אובייקט UnaryExpr עם פרמטר op מתאים:

```
806 unaryexp[expr_ty]:
807     | '++' a=NAME { _PyAST_UnaryExpr(_PyPegen_set_expr_context(p, a, Store), Pre_inc, EXTRA) }
808     | a=NAME '++' { _PyAST_UnaryExpr(_PyPegen_set_expr_context(p, a, Store), Post_inc, EXTRA) }
809
```

אחרי שהגדרנו בהצלחה חוק דקדוק חדש, צריך להבין איך הוא משתלב בחוקי הדקדוק הקיימים של Python. מתי נרצה להפעיל את החוק הזה? אפשר להסתכל על חוקים אחרים ולנסות להבין איפה היינו רוצים לאפשר שימוש בחוק שלנו, ועם ניסוי וטעיה אפשר למצוא לו מקום מתאים.

הרעיון הוא לאפשר לאופרטור שלנו כמה שיותר חופש להיות בהקשר של פעולות אחרות, כלומר לאפשר לו להיות בתוך ביטוי מתמטי מורכב כרצוננו. הבחירה הזו היא כנראה החלק הכי קשה בתהליך הוספת האופרטור לשפה.

אני מצאתי שמוצלח לשים את החוק החדש כאחת האפשרויות בחוק שנקרא primary. חוק זה נמצא יחסית בסוף שרשרת החוקים שמגדירה ביטויים מתמטיים, אחרי התאמת Token-ים של פעולות bitwise, חיבור,

כפל, פעולות אונריות, וחזקה. למעשה חוק זה נמצא ממש לפני חוק בשם atom שמתאים למספרים קבועים או שמות משתנים:

```
822  primary[expr_ty]:
823      | a=primary '.' b=NAME { _PyAST_Attribute(a, b->v.Name.id, Load, EXTRA) }
824      | a=primary b=genexp { _PyAST_Call(a, CHECK(asdl_expr_seq*, (asdl_expr_seq*)_PyPegen_singleton_seq(p, b)), NULL, EXTRA) }
825      | a=primary '(' b=[arguments] ')' {
826          _PyAST_Call(a,
827                      (b) ? ((expr_ty) b)->v.Call.args : NULL,
828                      (b) ? ((expr_ty) b)->v.Call.keywords : NULL,
829                      EXTRA) }
830      | a=primary '[' b=slices ']' { _PyAST_Subscript(a, b, Load, EXTRA) }
831      | unaryexp
832      | atom
```

בחירה במיקום זה מאפשרת לנו להשתמש באופרטור ++ עם כל פעולה מתמטית אחרת.

אחרי כל התהליך הזה סוף סוף הוספנו את החוק החדש שלנו, ואפילו Python מציג את ה-AST המתאים לו בצורה נוחה:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> import ast
>>> code = "x++"
>>> print(ast.dump(ast.parse(code).body[0], indent=4))
Expr(
    value=UnaryExpr(
        target=Name(id='x', ctx=Store()),
        op=Post_inc()))
```

אך עדיין לא הגדרנו אילו פקודות Bytecode יווצרו כשננסה לקמפל את האובייקט החדש הזה.

הגדרת הקומפילציה של אובייקט ה-AST לרצף פקודות Bytecode

כדאי להגדיר מה יקרה כשמנסים לקמפל את אובייקט ה-UnaryExpr שיצרנו. הקובץ שבו מגדירים לאיזה פקודות Bytecode מקמפלים כל אובייקט AST הוא compile.c. ספציפית הפונקציה שבה מקמפלים אובייקטים מסוג expression היא compiler_visit_expr1, ויש בה חלוקה למקרים לפי סוג ה-expression. להלן תחילת הפונקציה:

```
6120  static int
6121  compiler_visit_expr1(struct compiler *c, expr_ty e)
6122  {
6123      location loc = LOC(e);
6124      switch (e->kind) {
6125      case NamedExpr_kind:
6126          VISIT(c, expr, e->v.NamedExpr.value);
6127          ADDOP_I(c, loc, COPY, 1);
6128          VISIT(c, expr, e->v.NamedExpr.target);
6129          break;
6130      case BoolOp_kind:
6131          return compiler_boolop(c, e);
6132      case BinOp_kind:
6133          VISIT(c, expr, e->v.BinOp.left);
6134          VISIT(c, expr, e->v.BinOp.right);
6135          ADDOP_BINARY(c, loc, e->v.BinOp.op);
6136          break;
```

נוסיף לפונקציה זו התייחסות למקרה של אובייקט ה-UnaryExpr שלנו, ולבינתיים נגדיר שהוא יתקמפל לפקודת NOP שלא עושה כלום:

```
6120  static int
6121  compiler_visit_unary_expr(struct compiler *c, expr_ty e) {
6122      location loc = LOC(e);
6123      assert(e->kind == UnaryExpr_kind);
6124      ADDOP(c, loc, NOP);
6125      return SUCCESS;
6126  }
6127
6128  static int
6129  compiler_visit_expr1(struct compiler *c, expr_ty e)
6130  {
6131      location loc = LOC(e);
6132      switch (e->kind) {
6133      case UnaryExpr_kind:
6134          return compiler_visit_unary_expr(c, e);
6135      case NamedExpr_kind:
```

לאחר שמקמפלים את CPython ומנסים להשתמש באופטור ++ נתקלים בשגיאה:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 5
>>> x++
ValueError: Invalid CFG, stack underflow
>>>
```

מחיפוש השגיאה בקוד נגיע לפונקציה calculate_stackdepth בקובץ flowgraph.c. הפונקציה מקבלת basic block, עוברת על פקודות ה-Bytecode שנמצאות בו, ומוודאת שאף פקודה לא נוגעת בערכי מחסנית שנמצאים מחוץ לאותו basic block. בשביל להבין למה האופרטור שלנו גורם לכך נסתכל על רצף פקודות ה-Bytecode שנוצר בקומפילציה של ביטוי אחר:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> dis.dis("x")
  0           RESUME                   0

  1           LOAD_NAME                0 (x)
              RETURN_VALUE
>>>
```

נראה שמצופה שרצף פקודות ה-Bytecode שנוצר בקומפילציה של אובייקט ה-AST ישאיר את ערך התוצאה על המחסנית כדי שהוא יועבר הלאה. בעוד שבינתיים הגדרנו שה-Bytecode שיווצר בקומפילציה של האובייקט UnaryExpr יכיל פקודת NOP בלבד, כלומר לא ישאיר את ערך התוצאה על המחסנית. אז פקודת ה-RETURN_VALUE מוציאה מהמחסנית ערך שלא נמצא ב-basic block הנוכחי.

כדי לוודא זאת, נחליף את פקודת ה-NOP שלנו בפקודה ששמה על המחסנית ערך קבוע:

```
6120  static int
6121  compiler_visit_unary_expr(struct compiler *c, expr_ty e) {
6122      location loc = LOC(e);
6123      assert(e->kind == UnaryExpr_kind);
6124      ADDOP_LOAD_CONST_NEW(c, loc, PyLong_FromLong(1337));
6125      return SUCCESS;
6126  }
```

הפעם כשמשתמשים באופרטור שלנו - תוצאת הביטוי היא הערך 1337 וכצפוי הערך של המשתנה נשאר זהה - כי לא ביצענו שום פעולה עליו.

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 5
>>> x++
1337
>>> x
5
```

הגיע הזמן באמת להגדיר אילו פקודות Bytecode יווצרו עבור UnaryExpr. כדי לקבל השראה ניתן להסתכל על הפקודות שנוצרות בשימוש באופרטור walrus:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> import dis
>>> dis.dis("(x := y + 1)")
  0           RESUME                   0

  1           LOAD_NAME                0 (y)
              LOAD_CONST               0 (1)
              BINARY_OP                0 (+)
              COPY                     1
              STORE_NAME               1 (x)
              RETURN_VALUE
```

נראה שמבצעים evaluation ל-expression של ה-value (שמימין ל-":="), משכפלים אותו (COPY), ואת העותק שומרים ל-expression של ה-target (שמשמאל ל-":="), שהוא מסוג Name כזכור. כצפוי, רצף פקודות זה אכן משאיר את ערך החזרה בראש המחסנית. למעשה ראינו כבר קודם איך נראה הקוד הרלוונטי ל-walrus:

```
6124      switch (e->kind) {
6125      case NamedExpr_kind:
6126          VISIT(c, expr, e->v.NamedExpr.value);
6127          ADDOP_I(c, loc, COPY, 1);
6128          VISIT(c, expr, e->v.NamedExpr.target);
6129          break;
```

נראה תואם לניתוח שלנו ול-disassembly שראינו. יש לציין שהמאקרו VISIT קורא רקורסיבית לפונקציה compiler_visit_expr1, ועבור שדה ה-target (שכזכור בחוק הדקדוק מוגדר כ-NAME) נכנסים ל-case שמתאים לאובייקט Name_kind:

```
6278      case Name_kind:
6279          return compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

הפונקציה compiler_nameop יוצרת פקודת Bytecode שמוציאה ערך מהמחסנית לתוך המשתנה, או דוחפת את ערך המשתנה למחסנית, הכל בהתאם לשדה ה-context שבאובייקט. כזכור, בחוק הדקדוק שלנו יצרנו את אובייקט ה-NAME עם שדה context שערכו Store. אך אנחנו רוצים גם לקרוא את ערך המשתנה וגם לכתוב אליו בהמשך. לכן לא נשתמש במאקרו VISIT, אלא נקרא לפונקציה compiler_nameop ישירות עם context בהתאם לצורך שלנו.

כעת יש לנו את כל הכלים כדי לכתוב קוד דומה שיתאים לשימוש שלנו:

```
6120  static int
6121  compiler_visit_unary_expr(struct compiler *c, expr_ty e) {
6122      location loc = LOC(e);
6123      assert(e->kind == UnaryExpr_kind);
6124      expr_ty target = e->v.UnaryExpr.target;
6125      assert(target->kind == Name_kind);
6126      switch (e->v.UnaryExpr.op) {
6127          case Pre_inc:
6128              compiler_nameop(c, loc, target->v.Name.id, Load);
6129              ADDOP_LOAD_CONST_NEW(c, loc, PyLong_FromLong(1));
6130              ADDOP_BINARY(c, loc, Add);
6131              ADDOP_I(c, loc, COPY, 1);
6132              compiler_nameop(c, loc, target->v.Name.id, Store);
6133              break;
6134
6135          case Post_inc:
6136              compiler_nameop(c, loc, target->v.Name.id, Load);
6137              ADDOP_I(c, loc, COPY, 1);
6138              ADDOP_LOAD_CONST_NEW(c, loc, PyLong_FromLong(1));
6139              ADDOP_BINARY(c, loc, Add);
6140              compiler_nameop(c, loc, target->v.Name.id, Store);
6141              break;
6142      }
6143      return SUCCESS;
6144  }
```

במקרה של Pre_inc, אנחנו מחשבים את פעולת החיבור, משכפלים את התוצאה, כותבים עותק אחד בחזרה למשתנה ומחזירים את העותק השני. הפקודות שנוצרות הן:

- דחיפת ערך המשתנה למחסנית.
- דחיפת הקבוע 1 למחסנית.
- פעולת חיבור.
- פעולת שכפול.
- שליפת ראש המחסנית ושמירת הערך הנשלף בחזרה לתוך המשתנה.

במקרה של Post_inc אנחנו קודם משכפלים את הערך הישן של המשתנה, מבצעים פעולת חיבור ושומרים את התוצאה בחזרה למשתנה.

ערך החזרה הוא הערך המשוכפל, שהוא הערך הקודם של המשתנה. הפקודות שנוצרות הן:

- דחיפת ערך המשתנה למחסנית.
- פעולת שכפול.
- דחיפת הקבוע 1 למחסנית.
- פעולת חיבור.
- שליפת ראש המחסנית ושמירת הערך הנשלף בחזרה לתוך המשתנה.

בשלב הזה סוף סוף ניתן להשתמש באופרטור החדש שלנו!

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 5
>>> x++
5
>>> x
6
>>> ++x
7
>>> x
7
```

לכאורה נראה שסיימנו. כשמקמפלים את CPython בגרסאת Release, האופרטור החדש אכן עובד כמצופה. אך כשמקמפלים את CPython בגרסאת Debug, מתווספות בדיקות בזמן ריצה שנכשלות כשמנסים לעשות שימוש באופרטור החדש. לצורך שלמות השינוי שאנחנו מוסיפים, כן אתייחס לתיקונים שיש לבצע, אך הקורא יכול לדלג על חלק זה.

תיקון בדיקות שנכשלות בגרסאת Debug

כשמקמפלים את CPython בגרסאת Debug ומנסים להשתמש באופרטור ++, מקבלים את השגיאה הבאה:

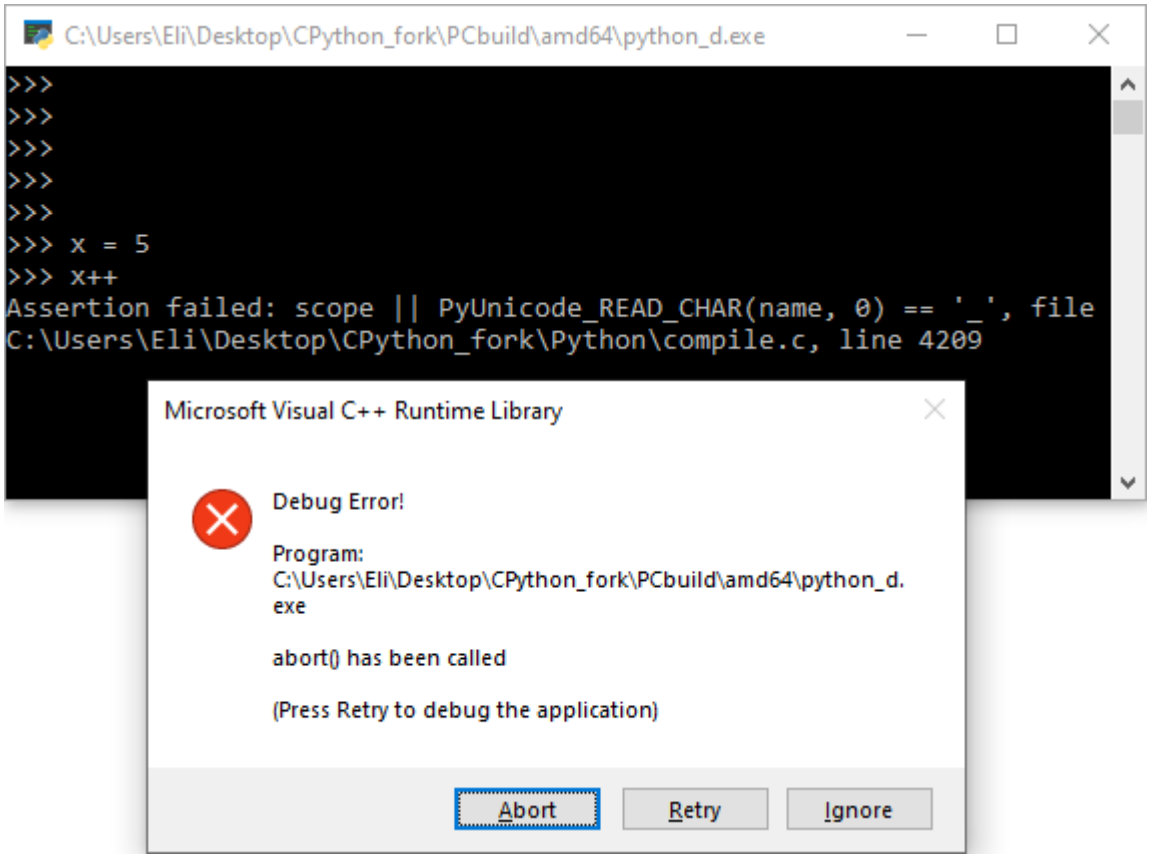
```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python_d.exe
>>> x = 5
>>> x++
SystemError: unexpected expression
```

כשמחפשים אותה בקוד מגיעים לפונקציה validate_expr שבקובץ ast.c. בפונקציה זו מוודאים שהשדות של אובייקט ה-AST שנוצר הם כמצופה. למשל עבור אובייקט NamedExpression בביטוי כמו "x=5" מוודאים שה-context של x הוא Store ולא Load, ומוודאים שהשם "x" הוא חוקי ולא "True" וכו'. באופן דומה, לכל סוג אובייקט יש את הבדיקות הרלוונטיות אליו. בפונקציה זו אין התייחסות לאובייקט החדש שלנו ולכן היא מחזירה שגיאה.

במקרה של האובייקט החדש שלנו יש רק שני שדות, ומתוכם יש הגבלה רק על השדה target שחייב להיות שם של משתנה ועם context של Store (כפי שהגדרנו בחוק הדקדוק שהוספנו). לכן נוסיף בדיקה כזאת לאובייקט שלנו:

```
272     case BinOp_kind:
273         ret = validate_expr(state, exp->v.BinOp.left, Load) &&
274             validate_expr(state, exp->v.BinOp.right, Load);
275         break;
276     case UnaryExpr_kind:
277         ret = validate_expr(state, exp->v.UnaryExpr.target, Store);
278         break;
```

לאחר תיקון זה, כשמנסים להשתמש באופרטור מקבלים שגיאה אחרת, והפעם התוכנית גם קורסת:



התוכנית קורסת בגלל assert שמבצע בדיקה על שם המשתנה וה-scope שבו הוא נמצא. אם מדבגים מעט את הקוד ניתן להבין שהמשתנה x לא נמצא ב-symbol table. למעשה, לפני שמקמפלים את אובייקט ה-AST לפקודות Bytecode, מתבצע מעבר ראשוני על ה-AST שבו יוצרים symbol table ומכניסים אליו את כל המשתנים שמשתמשים בהם ב-AST.

ניתן לקרוא על כך בהרחבה בתיעוד. התהליך הראשוני הזה קורה בקובץ symtable.c ובפרט בפונקציה _PySymtable_Build שקוראת לפונקציה symtable_visit_expr.

בפונקציה זו מוגדר לכל סוג expression אילו מבין השדות שלו צריכים להכניס לתוך ה-symbol table. במקרה שלנו מדובר בשדה ה-target. נוסיף התייחסות לאובייקט שלנו בפונקציה זו:

```
2091  static int
2092  symtable_visit_expr(struct symtable *st, expr_ty e)
2093  {
2094      if (++st->recursion_depth > st->recursion_limit) {
2095          PyErr_SetString(PyExc_RecursionError,
2096                          "maximum recursion depth exceeded during compilation");
2097          VISIT_QUIT(st, 0);
2098      }
2099      switch (e->kind) {
2100      case UnaryExpr_kind:
2101          VISIT(st, expr, e->v.UnaryExpr.target);
2102          break;
```

לאחר תיקון זה ניתן להשתמש באופרטור החדש גם כשמקמפלים את CPython בגרסאת Debug.

הוספת תמיכה ב-ast.unparse

ראינו שהספריה ast יכולה להציג את ה-AST של קוד Python על ידי שימוש בפונקציה ast.parse. ספריה זו יכולה גם להציג בחזרה קוד Python בהינתן ה-AST המתאים לו. הדבר האחרון שיש להתייחס אליו (אך לא חובה) הוא הוספת תמיכה באופרטור החדש בפעולה זו:

```
Python 3.11 (64-bit)
>>> import ast
>>> tree = ast.parse("x+1")
>>> print(ast.dump(tree.body[0], indent=4))
Expr(
    value=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Constant(value=1)))
>>> ast.unparse(tree)
'x + 1'
>>>
```

למזלנו קוד זה כבר כתוב ב-Python ולכן יהיה קל יותר להכניס את השינוי הנדרש. הקוד נמצא בקובץ ast.py שתחת התיקיה Lib. תיקיה זו מכילה ספריות מובנות (builtin) בהפצת ה-Python שמתקינים, וספריות אלה כתובות ב-Python. כשמתקינים את Python, הקבצים שבתיקיה מועתקים לתיקיית ההתקנה וניתן להשתמש בהם עם import מבלי לדאוג לגבי המיקום שלהם.

להלן הקוד הנדרש בשביל להוסיף את התמיכה באופרטור:

```
893     def visit_NamedExpr(self, node):
894         with self.require_parens(_Precedence.NAMED_EXPR, node):
895             self.set_precedence(_Precedence.ATOM, node.target, node.value)
896             self.traverse(node.target)
897             self.write(" := ")
898             self.traverse(node.value)
899
900     def visit_UnaryExpr(self, node):
901         with self.require_parens(_Precedence.UNARY_EXPR, node):
902             self.set_precedence(_Precedence.ATOM, node.target)
903             op = node.op.__class__.__name__
904             if op == "Pre_inc":
905                 self.write(" ++")
906                 self.traverse(node.target)
907                 self.write(" ")
908             elif op == "Post_inc":
909                 self.write(" ")
910                 self.traverse(node.target)
911                 self.write("++ ")
912
```

לאחר הוספת הקוד ניתן להשתמש ב-ast.unparse על AST שמכיל את האופרטור ++:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> import ast
>>> tree = ast.parse("x++")
>>> print(ast.dump(tree.body[0], indent=4))
Expr(
    value=UnaryExpr(
        target=Name(id='x', ctx=Store()),
        op=Post_inc()))
>>> ast.unparse(tree)
'( x++ )'
```

התוצאה

בגלל האופן שבו הגדרנו את האופרטור בדקדוק של השפה כביטוי מסוג primary, ניתן להשתמש בו ככזה ולהריץ חישובים מתמטיים מגוונים:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 10
>>> y = 2 * x++
>>> y, x
(20, 11)
>>> x = 10
>>> y = x++ + ++x
>>> y, x
(22, 12)
>>> x = 1
>>> y = x++ * x++
>>> y, x
(2, 3)
```

ניתן גם להשתמש באופרטור כתנאי ב-if או בלולאה:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 5
>>> while x++ < 10:
...  print(x)
...
6
7
8
9
10
```

כפרמטר בקריאה לפונקציה:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 10
>>> while x < 15:
...  print(++x)
...
11
12
13
14
15
```

ב-yield:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> def gen():
...  x = 0
...  while True:
...   yield x++
...
>>> g = gen()
>>> next(g)
0
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
```

ובעוד הרבה דרכים.

בנוסף לכך, בגלל שבחרנו לקמפל את האופרטור לפקודת Bytecode של (+) BINARY_OP, למעשה האופרטור לא מוגבל למספרים שלמים בלבד, אלא לכל type שתומך באופרטור + עם מספר שלם כפרמטר.

למשל עבור משתנה מסוג float:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 3.5
>>> x++
3.5
>>> x
4.5
>>>
```

האופרטור גם יעבוד על כל אובייקט ממחלקה שמממשת את הפונקציה __add__:

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> class Person:
...  def __init__(self, name, age):
...   self.name = name
...   self.age = age
...  def __repr__(self):
...   return f"{self.name} is {self.age} years old"
...  def __add__(self, n):
...   return Person(self.name, self.age + n)
...
>>> e = Person("Eli", 29)
>>> p = e++
>>> p
Eli is 29 years old
>>> e
Eli is 30 years old
>>>
```

עם זאת, מכיוון שבדקדוק השפה הגדרנו שה-Token שמפעילים עליו את האופרטור חייב להיות מסוג Name, אי אפשר להשתמש באופרטור בביטויים כמו x[0]++. ניסיון לעשות זאת יתקל בשגיאת syntax error.

אופרטורים נוספים

ניתן להכליל את האופרטור שלנו ולהוסיף עוד פעולות כגבול הדמיון שלנו. למשל:

- -- - חיסור אחד
- <<< - היסט שמאלה אחד (shift left)
- >>> - היסט ימינה אחד (shift right)
- ^^ - bitwise xor עם אחד
- && - bitwise and עם אחד
- || - bitwise or עם אחד
- *** - כפל באחד
- /// - חלוקה באחד
- %% - שארית החלוקה באחד

שלושת האופרטורים האחרונים ברשימה אולי מצחיקים אבל בהחלט אפשריים! מי יודע, אולי מתישהו מישהו יצור type חדש שבו הקריאות ל-__mul__, __floordiv__, או __mod__ עם הקבוע 1 מבצעות פעולות משמעותיות.

לשמחתנו עשינו כבר את רוב העבודה ולכן השינויים שנצטרך להוסיף במקרה כזה יהיו מינימליים:

1. הוספת ה-Token החדש.
2. עדכון חוק הדקדוק.
3. הוספת enum ל-op הנוסף.
4. בחירת פקודות ה-Bytecode שיווצרו עבור ה-op הנוסף.

אני מימשתי בפועל את כל האופרטורים שהזכרתי כאן ב-fork שלי ל-CPython בקישור.

```
C:\Users\Eli\Desktop\CPython_fork\PCbuild\amd64\python.exe
>>> x = 10
>>> --x
9
>>> <<<x
18
>>> ^^x
19
```

רעיונות להמשך

כפי שהוספנו אופרטור ++ ל-Python, ניתן לבצע שינויים נוספים כרצוננו. כמה מהרעיונות שעלו לי הם: החלפת מילים של Token-ים שונים, למשל: "continue" ב "tryagain", "break" ב-"getout", "return" ב-"giveback", "while" ב-"aslongas", וכו'. התוצאה שיכולה להיות היא קוד שנראה קצת שונה ומצחיק.

רעיון נחמד נוסף יכול להיות הוספת האופרטור הטרנרי:

```
x = a ? b : c
```

ורעיונות קצת יותר מעניינים יכולים להיות שינויים שמשפיעים יותר על ה-control flow, למשל הוספת אפשרות לעשות לולאת do-while, או switch-case. אם כי אלה יצריכו להכניס שינויים במקומות שלא התייחסתי אליהם במסגרת מאמר זה.

סיכום

במאמר זה למדנו על פירוק קוד המקור ל-Token-ים, חוקי הדקדוק של השפה, אופן בניית ה-AST והגדרת האובייקטים שבו, ובחירת פקודות ה-Bytecode שיווצרו בתהליך הקומפילציה של אובייקטים אלה. עשינו זאת תוך כדי עריכת השפה בפועל, שזה די מגניב. בנוסף חשבנו על עוד רעיונות לשינויים וראינו שהגבול הוא הדמיון שלנו ושאפשר לשנות בשפה דברים כרצוננו.

ראינו על קצה המזלג מעט פקודות Bytecode אבל לא נכנסנו לעומק על איך פקודות אלה רצות בפועל. במאמר הבא נלמד על איך הפקודות רצות בפועל ב-CPython, על מימוש המחסנית של השפה, ואפילו נוסיף פקודת Bytecode חדשה משלנו! בנוסף נלמד לעומק על האובייקטים עצמם ב-CPython וייצוגם בזיכרון. בהמשך גם נמצא באגים במימוש של CPython, ונעשה עוד דברים מגניבים. Stay tuned!

על המחבר

אני אלי קסקי, בן 29, אוהב לכתוב ולפתור אתגרי CTF, במיוחד בקטגוריות Reverse Engineering וקריפטוגרפיה, ואוהב לעשות דברים מגניבים עם קוד.

לפניות בכל נושא מוזמנים ליצור איתי קשר ב-elikaski94@gmail.com או ב-LinkedIn.