UNIVERSITATEA DIN
BUCUREȘTI

FACULTATEA DE
MATEMATICĂ ȘI
INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

TINYJIT: COMPILATOR JIT PENTRU PYTHON

Absolvent
Richițeanu Mihai-Sebastian

Coordonator științific
Conf. Univ. Dr. Ing. Paul Irofti

București, iunie 2022

Rezumat

Compilarea este primul pas către îmbunătățirea performanței oricărui limbaj de programare. Limbajul Python duce lipsă de acestă însușire, el fiind un limbaj interpretat. În acest scop, se pot folosi compilatoare de sine stătătoare sau biblioteci care compilează părți ale codului, unele servind limbajului Python, iar altele unor extensii ale acestuia.

În această lucrare prezint TinyJit, un compilator JIT pentru limbajul Python, care se bazează pe un decorator pentru a compila doar funcțiile necesare. Compilatorul este integrat în biblioteca LLVM și folosește reprezentarea intermediară a acestuia. Codul nedecorat continuă a fi interpretat. Această flexibilitate de a alege între interpretor și compilator, unde este cazul, aduce îmbunătățiri semnificative timpilor de execuție a programelor Python.

Abstract

Compiling is the first step towards improving the performance of any programming language. Python lacks this feature, being an interpreted language. For this purpose, standalone compilers or libraries that compile parts of the code can be used, some using the Python language and some extensions of it.

In this paper I present TinyJit, a JIT compiler for the Python language, which relies on a decorator to compile only the necessary functions. The compiler is integrated into the LLVM library and uses its intermediate representation. The undecorated code continues to be interpreted. This flexibility to choose between interpreter and compiler, where appropriate, brings significant improvements to Python runtime.

Cuprins

Listă de figuri

Listă de tabele

Capitolul 1

Introducere

1.1 Motivație

Utilizatorii se așteaptă ca programele pe care le utilizează să fie simplu de folosit, cât mai eficiente și cât mai rapide. Satisfacerea acestor cerințe este un efort suplimentar pentru dezvoltatorii de *software*[1]. Am ales această temă, deoarece mereu am fost captivat de îmbunătățirea performanțelor și de modul de lucru al compilatoarelor. Cercetarea în acest subiect mi-a dezvoltat o nouă idee asupra adâncimii domeniului și m-a ajutat să înțeleg mai bine complexitatea și necesitatea lor.

1.2 Scopul lucrării

Programul descris în această lucrare își propune să vină în ajutorul programatorilor care doresc să îmbunătățească performanțele limbajului de programare Python. Compilatorul poate adnota una sau mai multe funcții, care vor fi compilate, pentru a maximiza eficiența.

Este de preferat ca TinyJit să fie utilizat după ce a fost folosit un profilator de performanță, care indică *bottleneck*-urile din implementare [16], dar acest aspect nu face teza lucrării.

Codul sursă este disponibil pe GitHub: https://github.com/SebastianRichiteanu/TinyJit.

[1]Pe parcursul lucrării am utilizat o convenție pentru termenii preluați din limba engleză, și anume să fie utilizat un font italic.

1.3 Obiective

Implementarea acestei lucrări are drept obiective:

- **Generalitate:** Poate fi folosit pe orice tip de program Python, indiferent de arhitectura calculatorului.
- **Eficiență:** Compilează funcții foarte rapid, iar executarea lor durează foarte puțin până și la funcțiile foarte costisitoare din punct de vedere al timpului de executie.

1.4 Structura lucrării

În structura lucrării, au fost urmărite șase aspecte: introducere, preliminarii, abordări recente, soluția propusă, implementare și concluziile.

În introducere, este prezentată o perspectivă de ansamblu asupra lucrării.

Ulterior, în preliminarii, este descris modul de lucru al unui compilator, prin compararea de implementări și soluții pentru diferite concepte, ca de exemplu: strategii de compilare, reprezentări intermediare, etc.

Capitolul de abordări recente prezintă diferitele abordări ale unor compilatoare asemănătoare dezvoltate recent.

Acesta este urmat de soluția propusă, care prezintă în ansamblu modul de funcționare, problemele întâmpinate în urma alegerii utilizării acestuia, dar și un mod de evaluare al soluției.

Implementarea descrie modul de lucru al compilatorului în amănunt, detaliază alegerile făcute în cadrul implementării și prezintă câteva restricții ale soluției.

În final, lucrarea este încheiată cu un set de concluzii, care au scopul de a rezuma informațiile prezentate și de a indica dezvoltări ulterioare.

Capitolul 2

Preliminarii

2.1 Compilatoare

Un compilator este un program care traduce un limbaj de programare de nivel înalt în cod mașină, pentru a crea un program executabil.

2.1.1 Componentele unui compilator

Un compilator efectuează mai multe sau toate operațiile următoare, denumite faze: analizare lexică, analizare sintactică, analizare semantică, conversie într-o reprezentare intermediară, optimizare de cod și generare de cod, aceste șase faze putând fi grupate în două categorii, după cum urmează [1]:

1. **Analizare:**

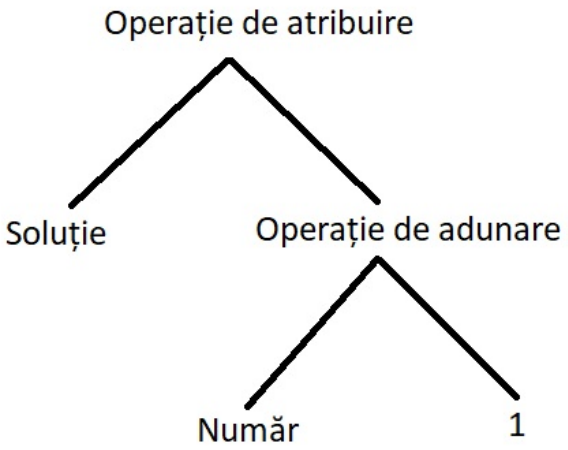
   - **Analizare lexică:** Este prima fază a unui compilator, numită și scanare. Ea are ca scop eliminarea de comentarii și spații și descompunerea sintaxei într-o serie de unități lexice. O unitate lexică poate fi compusă dintr-un singur caracter sau o serie de caractere, denumite *lexeme*, și din valoarea atributului. De exemplu: <Nume-unitate-lexică, valoare-atribut>. Unitățiile lexicale sunt clasificate ca fiind: identificatori, cuvinte cheie, separatori sau comentarii. Un simplu exemplu ar fi: **Soluție = număr + 1**

Tabela 2.1: Exemplu analizare lexică pentru operația Soluție = număr + 1

Lexeme (colecție de caractere)	**Unitate lexică (categorie lexeme)**
Soluție	Identificator
=	Operator de atriburie
număr	Identificator
+	Operator de adunare
1	Constantă de tipul întreg

- **Analizare sintactică:** Denumită și parsare, este următoarea fază a unui compilator. Ea are ca scop construirea unui arbore de sintaxă din unitățiile lexicale produse de către scanare. Un exemplu de astfel de analiză este făcută de biblioteca Abstract Syntax Trees (AST) din Python[1], care are ca rezultat:

Figura 2.1: Exemplu analizare sintactică: un AST al operației Soluție = număr + 1



- **Analizare semantică:** A treia fază a unui compilator, ea are ca scop verificarea existenței greșelilor produse de etapa anterioara, precum: compatibilități de date, variabile nedeclarate, etc.

2. **Sinteză:**

- **Generare reprezentare intermediară:** Această fază are ca scop generarea unui cod intermediar, ce se află la granița dintre limbaj mașină și limbaj înalt. Pe acest cod generat se vor aplica următoarele etape pentru a optimiza și traduce codul. Câteva exemple de limbaje intermediare ar fi: C, C--, Java Virtual Machine, LLVM, etc.
- **Optimizare de cod:** Faza aceasta optimizează codul intermediar produs de etapa anterioară, prin eliminarea și schimbarea unor instrucțiuni redundante, pentru a maximiza timpul de execuție și spațiul ocupat.
- **Generare de cod:** Ultima fază a unui compilator presupune generarea de cod mașină din reprezentarea intermediară optimizată la pasul anterior. Această fază depinde de arhitectura mașinii pe care urmează să fie executat codul.

[1]ast - Abstract Syntax Trees - https://docs.python.org/3/library/ast.html

2.1.2 Strategii de compilare

1. ***Just-In-Time*:** Este o tehnică de compilare ce presupune compilarea codului în timpul execuției unui program. Această tehnică este o combinație între compilarea statică și interpretare, câștigând avantaje și dezavantaje din ambele tehnici[7], câteva exemple fiind:

   - Mult mai folositoare pentru dezvoltare, deoarece oferă mai multe informații în timpul execuției
   - Poate optimiza codul mai bine decât un compilator AOT
   - Majoritaea aplicațiilor mari se execută mai lent cu un compilator JIT

2. ***Ahead-Of-Time*:** Presupune compilarea codului înainte ca acesta să fie executabil [21]. Din acest motiv, compilarea AOT are câteva avantaje:

   - Este compilarea standard, beneficiind de mai multe resurse
   - Compilarea este efectuată o singură dată
   - Timp de execuție mult mai scăzut comparativ cu JIT sau interpretarea
   - Securitate ridicată, codul sursă nefiind dezvăluit

3. ***Transcompilation*:** Presupune traducerea unui program scris într-un limbaj de nivel înalt în alt limbaj de nivel înalt. Această strategie de compilare face conversia între două limbaje ce lucrează la același nivel de abstractizare, comparativ cu un compilator tradițional, care traduce un limbaj de programare de nivel înalt într-un limbaj de programare de nivel scăzut.

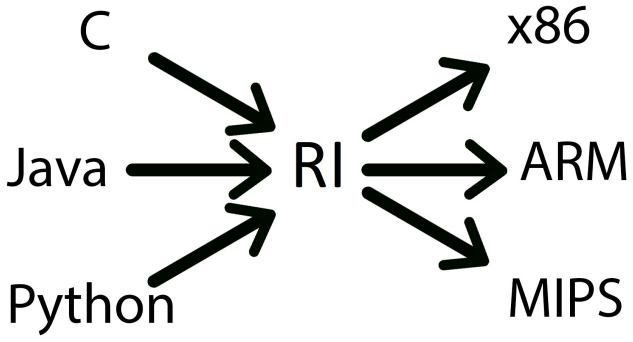
4. ***Dynamic Recompilation*:** Această strategie este adesea folosită în emulatoare și mașini virtuale, unde sistemul recompilează o parte a unui program în timpul execuției. Astfel, el poate adapta codul generat pentru a reflecta mediul de rulare.

2.2 Reprezentare intermediară (RI)

O reprezentare intermediară constituie o structură de date sau cod folosită de un compilator pentru a reprezenta codul sursă. O reprezentare intermediară este construită în așa fel încât ea să avantajeze următoarele etape ale compilării, precum: analizarea, optimizarea și traducerea.

Un compilator poate folosi una sau mai multe reprezentări intermediare [5], care aduc câteva beneficii, cel mai considerabil fiind eliminarea nevoii de a construi câte un compilator pentru fiecare limbaj către fiecare arhitectură. Folosind o reprezentare intermediară, se pot construi semi-compilatoare pentru trecerea din RI în orice arhitectură, acest semi-compilator fiind independent de codul sursă.

Figura 2.2: Exemplu reprezentare intermediară: o RI care acceptă cod sursă pentru 3 limbaje și generează cod pentru 3 arhitecturi



Reprezentările intermediare pot fi clasificate in 3 categorii [5]:

1. **RI Grafice:** Ele codifică elementele compilatorului într-o reprezentare grafică, precum: noduri, muchii, liste sau arbori.
2. **RI Liniare:** Elementele sunt codificate într-un pseudo-cod cu operații liniare.
3. **RI Hibride:** După cum sugerează și numele, ele combină elemente din ambele RI menționate anterior. O implementare comună a acestei strategii presupune folosirea unei RI liniare de nivel scăzut pentru blocurile de cod și un graf care reprezintă fluxul de control pentru aceste blocuri.

2.2.1 RI Grafice

Reprezentările intermediare grafice sunt folosite în multe dintre compilatoare. În majoritatea RI grafice structura arborelui corespunde cu sintaxa codului sursă. Există mai multe implementări pentru acest tip de reprezentare intermediară [5, p. 226-235]:

- ***Parse Trees*:** Ei corespund codului sursă, creându-și câte un nod pentru fiecare element al codului sursă, de exemplu: variabile, constante, expresii, operații, etc. Prin urmare, acestea nu gestionează memoria foarte bine și au nevoie de optimizări.
- ***Abstract Syntax Trees*:** AST reține informațiile necesare, la fel ca și *parse trees*, dar elimină termenii redunanți, precum expresii și termeni. Nodurile reprezintă elemente concrete, iar muchiile operații pentru aceste elemente.

2.2.2 RI Liniare

Reprezentările intermediare liniare sunt o alternativă pentru RI grafice. Aceste reprezentări intermediare sunt formate din succesiuni de instrucțiuni ce se execută în ordinea apariției lor, asemenea limbajelor de asamblare.

Există multe RI liniare, de exemplu: *One-address codes, Two-address codes, Three-address codes, Stack-Machine codes* [5, p. 235-243]. Cele care încă sunt relevante și au rămas în folosință sunt:

1. ***Stack-Machine Code*:** Această RI este compactă ca număr de instrucțiuni, fiind simplu de generat și de executat. Ea se folosește de un *stack* pentru calculul de operații. De exemplu, pentru operația a / b + c * 5 am avea:

Figura 2.3: Exemplu RI Stack-Machine pentru operația a / b + c * 5

```
1       push 5
2       push c
3       multiply
4       push b
5       push a
6       divide
7       addition
```

2. ***Three-Address Code*:** Această RI este și ea destul de compactă, folosind pentru operațiile sale cel mult 4 obiecte (3 variabile/constante și o operație), acest aspect oferind compilatorului avantajul de a putea refolosi variabilele. Cu excepția instrucțiunilor ce nu folosesc 4 obiecte, de exemplu *jump*, acestea sunt construite astfel: rezultat = operand1 operație operand2. Pentru operația a / b + c * 5 am avea:

Figura 2.4: Exemplu RI Three-Address Code pentru operația a / b + c * 5

```
1       t1 = 5
2       t2 = c
3       t3 = t1 x t2
4       t4 = b
5       t5 = a
6       t6 = t5 / t4
7       t7 = t6 + t3
```

2.2.3 RI Hibride

- ***Control-Flow Graph*:** CFG este un graf orientat, unde nodurile reprezintă secvențe de cod care se execută întotdeauna împreună, mai puțin în ceea ce privește cazul în care o instrucțiune aruncă o excepție. Muchiile reprezintă fluxul de lucru, compilatorul trecând dintr-o stare în alta.

- ***Dependence Graph*:** Aceste grafuri sunt folosite pentru a codifica relațiile dintre definiția unei valori și utilizările ei, nodurile reprezentând operații, iar muchiile unind noduri de tip valoare - variabilă.

- ***Call Graph*:** Este folosit pentru a reprezenta transferuruile de control dintre proceduri. Nodurile reprezintă proceduri, iar muchiile apeluri distincte către aceste proceduri. *Call graph* este folosit de compilatoare pentru a realiza analizare interprocedurală și optimizări.

2.3 Interpretoare

Un interpretor este un program care execută direct comenzile scrise într-un anumit limbaj de programare, fără ca acestea să fie compilate în cod mașină. Câteva exemple de limbaje de programare interpretate sunt: Ruby, Python, JavaScript.

Interpretoarele traduc codul sursă într-o reprezentare intermediară care va fi executată, această interpretare intermediară fiind independentă de arhitectură. Din acest motiv, interpretoarele sunt mai ușor de implementat [15]. Ele oferă un mod de depanare al erorilor mult mai bun decât în cazul compilatoarelor, prin programele lor de depanare, facându-le favorabile pentru dezvoltarea de software rapidă. Astfel, execuția codului este mai lentă, deoarece codul trebuie interpretat de fiecare dată, comparativ cu un compilator care creează un program executabil.

2.4 Python

Python [20] este un limbaj de programare de nivel înalt, interpretat, design-ul său punând accent pe claritatea codului, prin utilizarea unei sintaxe dictată de formatare și spațiere strictă [18]. Python este *dynamically-typed*, din acest motiv el este atractiv pentru dezvoltarea rapidă de software, dar care face dificilă detectarea timpurie a erorilor și compilarea eficientă [9]. Fiind un limbaj de programare interpretat câștigă independență de platformă și un mod de depanare a defectelor mai bun, dar duce lipsă de avantajele unui compilator, precum timpul de execuție scăzut.

2.5 Decoratoare

În Python avem posibilitatea de a folosi decoratoare pentru funcții. Decoratoarele sunt funcții care se execută înaintea funcțiilor cărora sunt aplicate, primind ca argument funcțiile pe care au fost aplicate și argumentele acestora [12]. Decoratoarele pot fi și imbricate, dar acest aspect nu face obiectul lucrării. Un simplu exemplu ar fi:

Figura 2.5: Exemplu utilizare decorator

```
1       @decorator
2       def suma(a, b):
3           return a + b
```

Această bucată de cod s-ar traduce în:

Figura 2.6: Exemplu traducere decorator

```
1       suma = decorator(a, b)(suma).
```

2.6 Just-In-Time

Just in time (JIT) sau *dynamic compilation* este o metodă de compilare ce presupune compilarea codului în timpul execuției programului, spre deosebire de metoda clasică de a compila codul într-un executabil.

Primele apariții ale JIT au fost în 1960, când John McCarthy a menționat funcții care traduc codul în timpul rulării, în lucrarea sa *Recursive functions of symbolic expressions and their computation by machine* [13]. O altă apariție timpurie a fost în 1968, când Ken Thompson, în lucrarea sa denumită *Programming techniques: Regular expression search algorithm* [17], a folosit JIT pentru a mări viteza de compilare a expresiilor regulate în cod IBM7090.

JIT este folosit pentru a obține avantajele compilării statice, dar și ale interpretării [2]. Câteva beneficii ar fi:

- Programele compilate rulează mai rapid decât cele interpretate
- Programele interpretate au de obicei o mărime scăzută
- Programale interpretate sunt adesea mai portabile

În lucrarea „A brief history of just-in-time” [2], scrisă de John Aycock, a fost propusă următoarea clasificare pentru compilarea JIT, bazată pe 3 însușiri comune ale acestor sisteme:

- **Invocare:** Ea poate fi de 2 tipuri: explicită și implicită. Explicită când utilizatorul trebuie să facă o acțiune pentru a începe compilarea, iar implicită când utilizatorul nu trebuie să facă nimic.
- **Executabilitate:** Și ea poate fi de 2 tipuri: mono-executabil și poly-executabil. Mono-executabil este întâlnit atunci când compilatorul poate executa un singur limbaj, iar poly-executabil atunci când poate executa mai multe.
- **Concurență:** Reprezintă proprietatea compilatorului de a rula concomitent cu programul, fără a crea întreruperi. Compilatorul poate fi un alt *thread* sau un alt proces, care poate executa pe un procesor diferit.

Capitolul 3

Abordări recente

Înainte de existența compilatoarelor, *software*-ul era scris în limbaj de asamblare sau direct în cod mașină. Reducerea timpului de execuție a fost necesară încă de la primele mașini de calcul.

Din nevoia de a scurta timpul de implementare și de a crește productivitatea, programatorii au început să folosească limbaje de nivel înalt care au nevoie de compilare. Aceste compilatoare au fost construite pentru fiecare limbaj, în ziua de astăzi ajungând foarte complexe.

În această secțiune voi prezenta diferitele abordări recente ale unor compilatoare pentru Python, puse deja în industrie, și anume: Cython [3], PyPy [4] și Numba [10]. Fiecare implementare are un scop diferit și, din acest motiv, arhitectura lor diferă.

Pentru a realiza o comparație cât mai clară între aceste compilatoare, voi exemplifica și modul de folosință. Funcția pe care am ales-o pentru exemplificare este o simplă implementare a șirului lui Fibonacci [6], ea arătând în felul următor:

Figura 3.1: Funcția fib: o implementare a șirului lui Fibonacci

```
1  def fib(n):
2      a = 0
3      b = 1
4      while b < n:
5          print(b)
6          a = b
7          b = a + b
```

3.1 Cython

Cython este un compilator static pentru limbajul Python, dar și pentru limbajul de programare Cython, care este un superset al limbajului Python. El oferă puterea combinată a limbajului Python, și anume lejeritatea și rapiditatea dezvoltării software, cu puterea limbajului C, adică performanța și execuția foarte rapidă a codului, odată ce a

fost compilat. Cython se integrează cu codul din biblioteci existente și aplicații vechi, are posibilitatea de a seta variabile statice, oferă un mod bun de depanare a erorilor, făcând depanarea la nivel de cod sursă și lucrează eficient cu seturi mari de date, deoarece folosește biblioteca NumPy.

Modul său de lucru se bazează pe crearea unor module din fișiere PYX (Pyrex Source Code Files), modulele fiind construite cu ajutorul funcției setup() din cadrul bibliotecii, care crează un fișier de C, modulul înlocuind codul sursă cu acest program C generat. Odată compilate modulele, ele pot fi importate în alte programe Python și folosite ca orice bibliotecă generică, de exemplu:

Figura 3.2: Exemplu folosire Cython

```
1 import modul_fib
2 modul_fib.fib(number)
```

3.2 PyPy

PyPy este un înlocuitor al CPython.

CPython este implementarea de bază a limbajului Python și cea mai des întâlnită. Ea este scrisă în C și în Python. CPython compilează codul Python în *bytecode*, care va fi interpretat. Din acest motiv el poate fi definit drept compilator, dar și ca interpretor.

PyPy este construit folosind limbajul RPython, un limbaj care a fost dezvoltat împreună cu PyPy. El oferă o viteză superioară și pentru programe mari. Din punct de vedere al memoriei, acesta reușește să ocupe mai puțin spațiu, comparativ cu CPython. O altă caracteristică interesantă a PyPy este că acesta oferă suport pentru mai multe limbaje de programare, precum: Prolog, JavaScript, Smalltalk, etc.

Deoarece PyPy este un compilator de sine stătător, utilizatorul nu are contact direct cu modul său de lucru. Tot codul, în acest caz doar funcția fib, este compilat și executat. Utilizatorul trebuind doar să aleagă ca interpretor fișierele binare ale pachetul PyPy.

3.3 Numba

Numba este un compilator JIT, care traduce un subset de instrucțiuni de Python și NumPy în cod mașină. El a fost dezvoltat pentru calculul științific, utilizând comenzi și instrucțiuni din biblioteca NumPy. El se folosește de biblioteca de compilatoare LLVM, aceasta oferind o gamă largă de tehnologii modulare și reutilizabile de compilare. Numba se folosește de un decorator pentru a desemna funcțiile care trebuiesc compilate, iar algoritmii compilați se pot apropia de vitezele limbajului C sau FORTRAN. Deoare-

ce Numba a fost dezvoltat pentru lucrul științific, el oferă suport pentru *notebook*-urile Jupyter, acestea fiind des utilizate în cercetare.

Modul de lucru al Numba este diferit de implementările anterioare. Numba a fost proiectat ca un modul de Python, ce poate fi instalat prin pip [1] și nu numai. Partea care relevă implementarea în cea mai mare măsură este decoratorul **@jit**, ce poate fi aplicat oricărei funcții care folosește biblioteca NumPy sau cod generic Python. Numba înlocuiește funcțiile decorate cu noi binare generate, restul codului fiind interpretat. În exemplul ales, pentru a putea compila funcția fib, trebuie adăugat doar decoratorul menționat înainte de declararea funcției, adică:

Figura 3.3: Exemplu folosire Numba

```
1       @jit
2       def fib(n):
```

[1] pip - documentation - https://pip.pypa.io/en/stable/

Capitolul 4

Soluția propusă

4.1 Concept

Noul compilator JIT, denumit TinyJit, este un compilator pentru Python care analizează fiecare apel de funcție cu decoratorul explicit, îl traduce în limbaj intermediar și execută acest cod, cu ajutorul unor biblioteci specifice. Ideea fundamentală provine de la compilatoare JIT deja existente, discutate anterior.

Programul are un mod de funcționare specific compilatoarelor JIT. El analizează codul, îl parsează în AST și apoi, pentru fiecare instrucțiune din arborele abstract de sintaxă, construiește codul intermediar care urmează să fie executat.

Python este un limbaj interpretat și *dynamically-typed*, acest lucru creează multe probleme când codul trebuie tradus în cod mașină, deoarece acest cod mașina este *statically-typed*. Compilatorul cere tipuri statice pentru fiecare variabilă și denumiri diferite pentru fiecare variabilă/condiție/bloc, indiferent dacă este explicită sau implicită, de exemplu (acest exemplu a fost simplificat pentru a facilita exemplificarea):

Figura 4.1: Exemplu cod Python de tradus

```
1    a = 5
2    while a:
3        a -= 1
```

Se va traduce în:

Figura 4.2: Exemplu traducere cod Python

```
1       bloc_configurare:
2           var a
3       bloc_intrare:
4           a = 5
5       bloc_while:
6           a = a - 1
7           tmp = compare(a, 0)
8           jump(tmp, block_while, bloc_while_end)
9       bloc_while_end:
10          ...
```

Aceste probleme vor fi tratate în secțiunea următoare.

4.2 Probleme de adaptare

4.2.1 Problema tipurilor

Problema tipurilor apare atunci când compilatorul este nevoit să calculeze operații pe tipuri de date diferite, precum int și double. Un exemplu simplu ar fi:

Figura 4.3: Exemplu cod pentru problema tipurilor

```
1       a = 1
2       b = 1.7
3       c = a + b
```

Care se traduce în c = 1 + 1.7.
Această problemă a fost rezolvată astfel:

- Verificăm dacă este posibilă o transformare între tipurile de date ale variabilelor. De exemplu între int și float există, deoarece 1 = 1.0, dar între array și float nu este posibil.
- Transformăm variabila de tip inferior în tipul superior al celeilalte. Adică creăm o variabilă temporară de tip superior care primește valoarea transformată a variabilei de tip inferior.
- Operația se va face acum între variabila de tip superior și variabila temporară.
- Întoarcere rezultat.

4.2.2 Problema variabilelor

Problema variabilelor presupune schimbarea tipurilor variabilelor. Cum Python este *dynamically-typed*, el acceptă fară problemă cod de forma:

Figura 4.4: Exemplu cod dinamic pentru problema variabilelor

```
1    a = 1
2    a = 1.7
3    a = True
```

Dar codul mașină are nevoie de tipurile de date explicte pentru fiecare variabilă, de exemplu:

Figura 4.5: Exemplu cod static pentru problema variabilelor

```
1    int a = 1
2    float a = 1.7
3    bool a = True
```

Această problemă a fost rezolvată astfel:

Primul pas este verificarea existenței variabilei în memorie. Acestă verificare desemnează 2 cazuri:

1. Variabila nu a fost declarată anterior. În acest caz creăm o instanță pentru aceasta și introducem în dicționarul de variabile referința ei.

2. Variabila a fost declarată anterior. În acest caz apar alte 2 cazuri:

   (a) Tipul anterior este același cu tipul curent care trebuie atribuit. În acest caz doar modificăm valoarea variabilei.

   (b) Tipul anterior este diferit de tipul curent. În acest caz executăm următorii pași:

   - Creăm o nouă variabilă cu tipul curent de date.
   - Inserăm în acea variabilă valoarea atribuită în cod.
   - Schimbăm referința variabilei în dicționarul de variabile cu noua variabilă creată.

   Soluția propusă nu este foarte eficientă din punct de vedere al memoriei, deoarece la fiecare schimbare de tip al unei variabile creăm o altă variabilă, variabila veche ocupând memorie inutil, însă consecința soluționării problemei va fi abordată în capitolele viitoare.

4.2.3 Problema *branch*-urilor

După cum am văzut in secțiunea 4.1, blocurile de cod au nevoie de un nume specific pentru a putea fi accesate în fiecare moment al execuției, prin apeluri de tip *jump*, care schimbă fluxul de lucru către un alt bloc[1]. În cod putem întalni instrucțiuni condiționale complexe, de exemplu:

Figura 4.6: Exemplu cod pentru problema branch-urilor - 1

```
1       if c == 0:
2           return a == b
3       elif c == 1:
4           return a != b
5       elif c == 2:
6           return a > b
7       else:
8           return False
```

Sau folosirea instrucțiunilor de tip while, for, etc. de mai multe ori, de exemplu:

Figura 4.7: Exemplu cod pentru problema branch-urilor - 2

```
1       a = 5
2       while a:
3           a = a - 1
4       b = 5
5       while b:
6           b -= 1
7       c = 6
8       while c != 1:
9           c /= 2
```

Fiecare ramură trebuie să aibă denumiri diferite pentru fiecare bloc. Problema a fost rezolvată prin numerotarea succesivă a acestora, de exemplu:

- then, then.1, then.2, etc.
- while, while.1, while.2, etc.

4.3 AST

Metoda pe care am ales-o pentru parsarea codului către un arbore de sintaxă este de a folosi bibliotecile inspect[2] și AST[3]. Cu ajutorul funcției getsource() a bibliotecii inspect, am extras codul unei funcții sub formă de șir de caractere (funcția fiind trimisa ca obiect), iar acest șir de caractere a fost trimis mai departe către funcția parse() a bibliotecii AST,

[1]LLVM Manual - br instruction - https://llvm.org/docs/LangRef.html#br-instruction
[2]inspect - Inspect live objects - https://docs.python.org/3/library/inspect.html
[3]ast - Abstract Syntax Trees - https://docs.python.org/3/library/ast.html

care va întoarce un nod de arbore abstract, ce conține toate informațiile relevante pentru compilare.

4.4 Generare

O dată construit arborele de sintaxă, el este trimis către generator, care va genera toate instrucțiunile LLVM. Fiecare instrucțiune va trece prin funcția generate a clasei Generator, care îi caută funcția de generare specifică, cu ajutorul funcției getattr() a limbajului Python. Un astfel de apel arată așa:

Figura 4.8: Exemplu getattr: căutarea funcției de generare specifică

```
1    visit = getattr(self, f"visit_{instruction_type}", None)
```

Funcțiile de generare, conform instrucțiunii corespunzătoare, vor crea blocuri de cod, variabile, constante, etc. Doar instrucțiunile de calcul simple, precum: adunare, scădere, deplasare(*shitft*), comparare sunt generate în clasele de tipuri, deoarece ele sunt dependente de tipul de date.

Este de menționat și faptul că blocul setup, entry și exit sunt create înainte de generarea de cod, ele fiind obligatorii pentru compilarea LLVM.

1. **setup:** are ca scop alocarea memoriei pentru variabile
2. **entry:** desemneaza momentul de start al instrucțiunilor
3. **exit:** reprezintă terminarea execuției și returnarea valorilor

4.5 Exemplu

Pentru a înțelege mai bine generarea și compilarea de cod, vom aplica pașii specificați mai sus pe un exemplu simplu. Am ales un program ce calculează, în mod naiv, scăderea absolută a 2 numere. În practică s-ar folosi funcția abs()[4].

[4]Python manual - abs - https://docs.python.org/3/library/functions.html#abs

Figura 4.9: Exemplu cod naiv pentru scădere absolută

```
1       @tinyjit
2       def simple():
3           a = 20
4           b = 33
5           if a > b:
6               c = a - b
7           else:
8               c = b - a
9           return c
```

În timpul compilării este creat arborele abstract de sintaxă, el având forma:

Figura 4.10: Arbore abstract de sintaxă generat pentru funcția simple

```
1  Module(
2      body=[
3          FunctionDef(name='simple',
4              args=arguments(posonlyargs=[],args=[],kwonlyargs=[],
5                             kw_defaults=[],defaults=[]),
6              body=[Assign(targets=[Name(id='a', ctx=Store())],
7                           value=Constant(value=20)),
8                     Assign(targets=[Name(id='b', ctx=Store())],
9                           value=Constant(value=33)),
10                    If(test=Compare(
11                          left=Name(id='a', ctx=Load()),
12                          ops=[Gt()],
13                          comparators=[Name(id='b', ctx=Load())]),
14                      body=[Assign(
15                              targets=[Name(id='c', ctx=Store())],
16                                  value=BinOp(
17                                  left=Name(id='a', ctx=Load()),
18                                  op=Sub(),
19                                  right=Name(id='b', ctx=Load()))),
20                      orelse=[Assign(
21                                  targets=[Name(id='c', ctx=Store())],
22                                  value=BinOp(
23                                  left=Name(id='b', ctx=Load()),
24                                  op=Sub(),
25                                  right=Name(id='a', ctx=Load())))]),
26                      Return(value=Name(id='c', ctx=Load()))],
27              decorator_list=[Name(id='tinyjit', ctx=Load())])],
28       type_ignores=[])
```

După cum se poate observa, fiecare instrucțiune este tradusă într-o instrucțiune din biblioteca AST, cu atributele sale specifice: atribuirea devine Assign cu targets și value (liniile 6-7, 8-9, 14-19, 20-25), If devine tot If (liniile 10-25) având atributele: test ce desemnează condiția, body, orelse, etc. Return devine Return cu atributul value (linia 26).

Acest arbore este generat apoi în cod LLVM și are forma descrisă în figura 4.11.

Figura 4.11: Cod LLVM generat pentru funcția simple

```
1   ; ModuleID = "__main__"
2   target triple = "unknown-unknown-unknown"
3   target datalayout = ""
4
5   define i64 @"simple"()
6   {
7   setup:
8     %".3" = alloca i64
9     %".5" = alloca i64
10    %".14" = alloca i64
11    %".23" = alloca i64
12    br label %"entry"
13  entry:
14    store i64 20, i64* %".3"
15    store i64 33, i64* %".5"
16    %".7" = load i64, i64* %".3"
17    %".8" = load i64, i64* %".5"
18    %".9" = icmp sgt i64 %".7", %".8"
19    br i1 %".9", label %"then", label %"else"
20  exit:
21    %".26" = load i64, i64* %".23"
22    ret i64 %".26"
23  then:
24    %".11" = load i64, i64* %".3"
25    %".12" = load i64, i64* %".5"
26    %".13" = sub i64 %".11", %".12"
27    store i64 %".13", i64* %".14"
28    br label %"end"
29  else:
30    %".17" = load i64, i64* %".5"
31    %".18" = load i64, i64* %".3"
32    %".19" = sub i64 %".17", %".18"
33    store i64 %".19", i64* %".14"
34    br label %"end"
35  end:
36    %".22" = load i64, i64* %".14"
37    store i64 %".22", i64* %".23"
38    br label %"exit"
39  }
```

În blocul setup (liniile 7-12), sunt alocate spații pe stivă pentru fiecare variabilă. Blocul se termină cu instrucțiunea br, care începe execuția unui alt bloc, în acest caz blocul entry. În blocul entry (liniile 13-19) sunt atribuite valori variabilelor a și b, iar apoi sunt comparate. În funcție de rezultatul întors, se va executa în continuare blocul then (liniile 23-28) sau blocul else (liniile 29-34). Ele fac doar o scădere simplă (ordinea parametrilor fiind schimbată în funcție de valoarea lor), stocând rezultatul în variabila c. Apoi ambele fac saltul către blocul end (liniile 35-38), care desemnează returnarea valorii calculate.

Acest cod este ulterior compilat, iar în cazul în care compilarea a avut succes, *bitcode*-ul va fi reținut într-un fișier.

4.6 Evaluarea soluției

Principalele trăsături care diferențiază un compilator sunt timpul de execuție și memoria folosită.

În această secțiune voi descrie rezultatele unei comparații între compilatorul dezvoltat și interpretorul de bază al limbajului Python. Programul compilat și interpretat este un program Python care adună într-o variabilă toate numerele de la 0 la o limită superioară setată, cu ajutorul instrucțiunii for. El are forma:

Figura 4.12: Funcția calc_sum: suma numerelor din intervalul [0, number)

```
1    @tinyjit
2    def calc_sum():
3        sum = 0
4        for x in range(number):
5            sum += x
6        return sum
```

4.6.1 Timp de execuție

Pentru a putea calcula timpul de execuție cât mai eficient, am folosit biblioteca time[5] și am creat o variabilă înainte de execuție, care reține timpul curent. După execuție, calculez diferența dintre momentul curent de timp și cel salvat anterior.

Tabela 4.1: Timpii de execuție ai funcției calc_sum

Number	**Timp Interpretare**	**Timp Compilare**
1000	$< 1e^{-12}$ s	0.01200246810913086 s
1000000	0.03600788116455078 s	0.014003276824951172 s
10000000	0.35408902168273926 s	0.014004230499267578 s
1000000000	35.462895154953 s	0.015004158020019531 s

Se poate observa că pentru numere mici de calcule, interpretorul are avantaj asupra compilatorului creat, deoarece el nu trebuie să parcurgă modulul tinyjit, interpretând direct codul. Pentru numere mari de calcule, compilatorul are avantaj, deoarece execută codul compilat, acesta fiind generat foarte rapid de programul creat.

4.6.2 Memoria folosită

Pentru a calcula memoria utilizată de program, am folosit biblioteca tracemalloc [6], care își setează *hooks* pe alocatoarele de memorie Python, urmărind fiecare alocare. Cu

[5]time - Time access and conversions - https://docs.python.org/3/library/time.html

[6]tracemalloc - Trace memory allocations - https://docs.python.org/3/library/tracemalloc.html

ajutorul funcției get_tracemalloc_memory() din cadrul bibliotecii, am obținut memoria folosită de program în momentul actual, dar și memoria maximă utilizată la un moment dat.

Tabela 4.2: Memoria folosită de program pentru funcția calc_sum

Tip	**Actual**	**Maxim**
Interpretat	0.000144 MB	0.00026 MB
Compilat	0.221992 MB	0.230079 MB

Compilatorul creat stochează arborele de sintaxă, instanțele pentru fiecare variabilă, instrucțiune, etc. Ca urmare a acestei stocări, compilatorul folosește mai multă memorie RAM decât interpretarea. Din tabelul 4.2 se poate observa că diferența maximă de memorie este de aproximativ 230KB, diferențele de acest ordin de mărime fiind neglijabile în sistemele actuale.

Capitolul 5

Implementare

Acest capitol descrie arhitectura, modul de folosire, punctele slabe și punctele forte ale compilatorului creat, dar și motivațiile alegerilor făcute.

5.1 Limbajul țintă

Am ales limbajul Python datorită faptului că este un limbaj foarte popular în acest moment, dar și pentru faptul că am lucrat mult cu el de-a lungul timpului, fiindu-mi foarte familiar și confortabil acum. De asemenea, există mai multe compilatoare JIT pentru Python (menționate în secțiunea 4.1) care constituie resurse foarte bune și surse de inspirație.

5.2 Limbajul de implementare

Am ales să implementez soluția propusă tot în limbajul Python, deoarece acesta este foarte flexibil, foarte bine documentat și oferă suport pentru o multitudine de biblioteci care minimizează timpul de implementare al acestui compilator. Un alt motiv pentru care am ales același limbaj și pentru implementare este că am considerat interesantă realizarea unui compilator în limbajul care va fi compilat.

5.3 Limbajul intermediar

Deoarece am decis că pentru compilatorul creat doresc independență de platformă, am fost nevoit să aleg o reprezentare intermediară. Am ales LLVM [11]. LLVM este un set de tehnologii ce pot fi folosite pentru a crea un *front-end* pentru orice limbaj de programare, dar și un *back-end* pentru orice arhitectură, acesta fiind un prim criteriu al alegerii. Un alt criteriu este faptul că LLVM vine cu un optimizator puternic al codului. Un ultim

criteriu a fost existența bibliotecii llvmlite[1]. Această bibliotecă face ușoară conexiunea dintre LLVM și Python, pentru a construi compilatoare JIT. Scopul principal al acestei biblioteci este de a "conecta" Python cu LLVM pentru biblioteca Numba [10].

5.4 Arhitectură

Instrumentul software creat este alcătuit din mai multe componente:

- Un modul de Python, denumit "tinyjit", ce conține toate programele pentru compilatorul creat.
- Un director de debug, care conține toate informațiile relevante referitoare la compilare.
- Un alt director pentru o suită de teste.

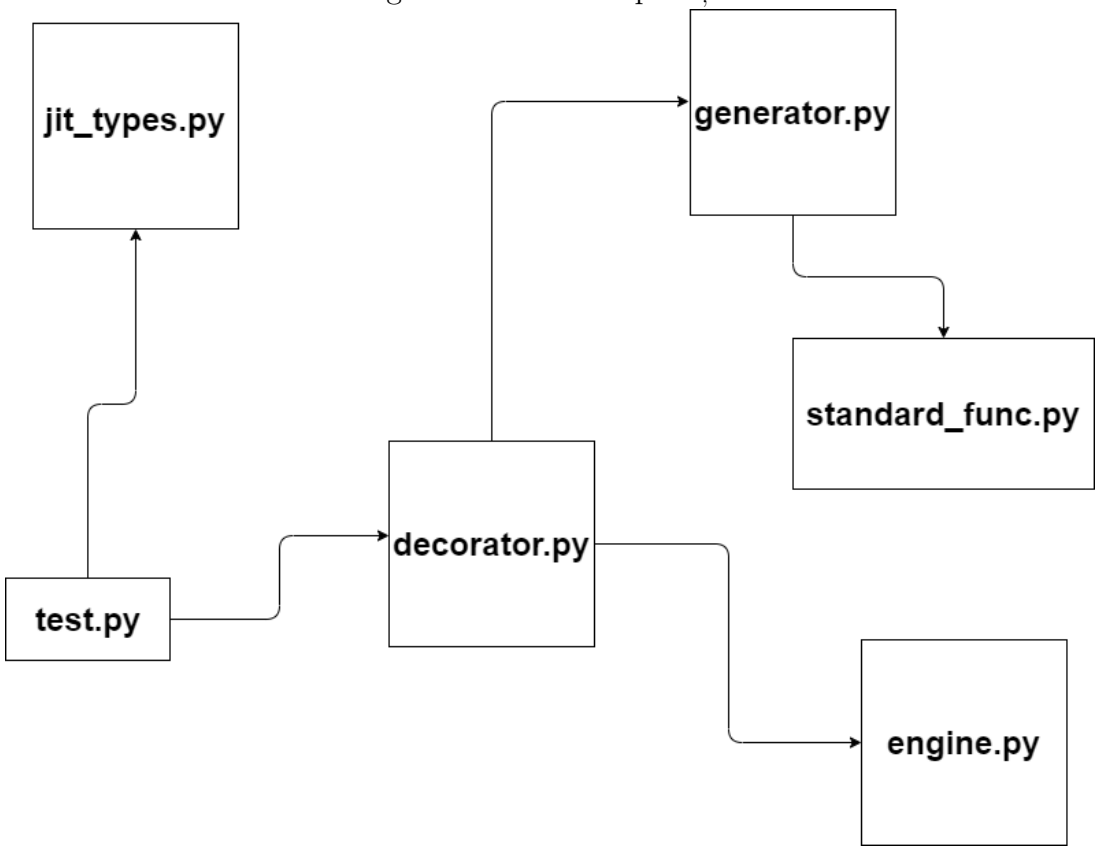
5.4.1 Modulul TinyJit

Această componentă conține fișierele necesare pentru compilator. În total au fost create 6 programe de Python pentru a administra și împărți codul, separând diferite funcționalități. Programele create sunt:

- **___init___.py:** Fișierul a fost necesar pentru ca Python să trateze acest director drept pachet. [19, p. 46]
- **jit_types.py:** Acest fișier conține toate clasele tipurilor de date, de exemplu: SignedInt, Float, Double, Array, etc. cu instanțele aferente fiecărui tip și fiecărei dimensiuni, dar și câteva funcții ajutătoare, precum conversii de tip.
- **decorator.py:** Acest fișier conține codul pentru decoratorul creat, ce va prelua argumentele și funcția adnotată și o va trimite mai departe în generator și în engine.
- **generator.py:** Acest fișier conține generatorul compilatorului, el va prelua codul Python transmis de decorator.py și va genera pentru fiecare instrucțiune cod LLVM.
- **engine.py:** Acest fișier conține motorul(*engine*) compilatorului, unde sunt stocate modulele și unde este compilat codul generat anterior.
- **standard_func.py:** În acest fișier au fost implementate funcțiile *default*, pe care compilatorul le poate folosi, precum: print.

[1]LLVMLITE Manual - https://llvmlite.readthedocs.io/

Figura 5.1: Fluxul aplicației



5.4.2 Debug

Această componentă a fost creată cu scopul de a analiza și corecta mai ușor posibilele greșeli de implementare. Compilatorul scrie în directorul respectiv doar dacă variabila de debug este setată ca fiind *True*. El creează aici 2 fișiere de text, după cum urmează:

1. **ast.txt:** În acest fișier este scrisă analizarea lexică facută de compilator, cu ajutorul bibliotecii AST.

2. **debug.llvm:** În fișier este scris codul LLVM generat, ce va fi compilat. Dacă compilatorul reușește să genereze tot codul pentru o funcție, aceasta va fi scrisă. Dacă el eșuează în momentul generării codului, aceasta nu va fi scrisă, compilatorul întorcând un mesaj de eroare.

5.4.3 Suita de teste

Componenta aceasta a fost creată cu scopul de a testa abilitățile compilatorului și de a putea compara timpul de execuție. Testarea acestora se va aduce în discuție ulterior, finalizându-se cu rezultatele evaluate din punct de vedere al abilității compilatorului.

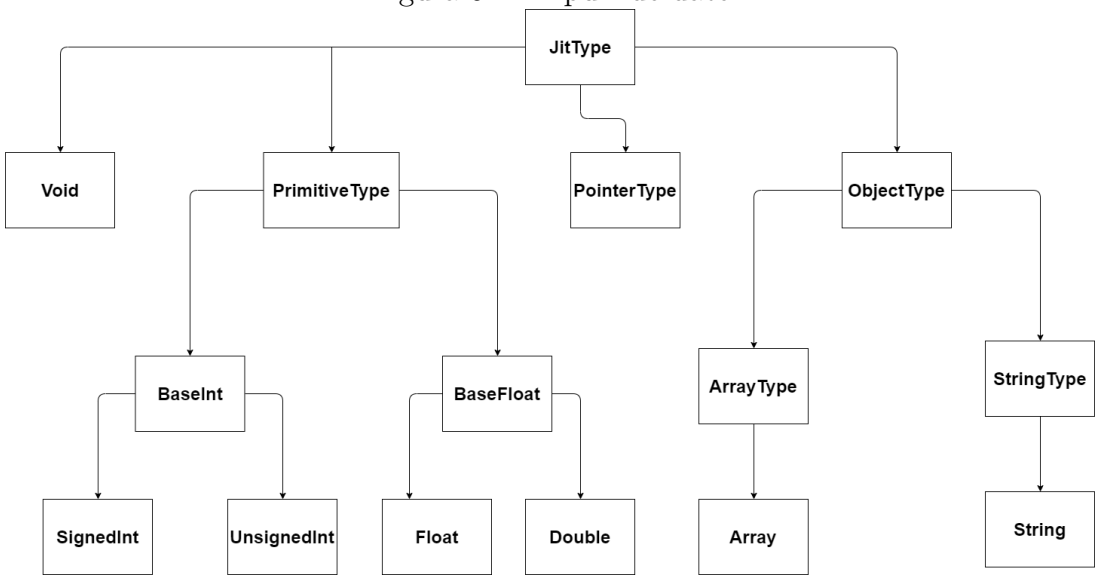
5.5 Re-compilare

Pentru a optimiza cât mai mult timpul de execuție, am decis să folosesc un fișier temporar, care se va crea după compilarea cu succes, el primind denumirea fișierului de compilat, unde am adăugat extensia .jit, de exemplu: main.py.jit. Înainte de compilare, programul creat va verifica existența acestui fișier și dacă fișierul de compilat a fost modificat (cu ajutorul parametrului st_mtime al funcției stat()[2], care întoarce timpul la care s-a realizat ultima modificare a fișierului). Dacă fișierul cu extensia .jit respectă aceste criterii, el va fi folosit pentru executare. Dacă nu, el va fi suprascris cu o nouă compilare a codului sursă.

5.6 Tipuri de date

Pentru a putea gestiona cât mai bine variabilele și constantele din cod, am ales să-mi creez clase pentru fiecare tip de date. Acestea au următoarea structură:

Figura 5.2: Tipuri de date



5.6.1 Asocierea claselor la tipuri LLVM

Fiecare clasă a fost asociată unui tip de date din C, cu ajutorul bibliotecii ctypes[3] și unui tip de date LLVM, pentru a putea genera un cod stabil și cât mai eficient din punct de vedere al memoriei. Toate asocierile se gasesc in tabelul 5.1.

Pentru asociere trebuie făcute câteva mențiuni:

[2]stat - Interpreting stat() results - https://docs.python.org/3/library/stat.html
[3]ctypes - A foreign function library for Python - https://docs.python.org/3/library/ctypes.html

1. Dacă variabilele din cod nu au tip explicit de date, adică sunt variabile Python, ele au fost asociate cu cel mai mare tip de date din punct de vedere al memoriei, care poate memora acea variabilă. Această soluție prezintă o problemă pentru gestionarea memoriei, dar ea va fi discutată mai târziu.

2. Tipul de date Bool este inexistent, el putând fi exprimat prin Unsigned Int de dimensiune 1.

3. Tipul de date Char este inexistent, el putând fi exprimat prin String de dimensiune 1. Char a fost asociat cu int de dimensiune 8, deoarece acesta poate reține un număr între -128 și 127 (total: 256).

4. Tipul de date String a fost asociat cu o listă de lungime variabilă, având tipul elementelor int de dimensiune 8.

5. Tipul de date Array a fost asociat cu o listă de lungime variabilă, având tipul elementelor tot variabil. Tipul elementelor poate fi oricare din tipurile int și float, de orice dimensiune.

Tabela 5.1: Asociere tipuri de date

Denumire Clasa	**Tip LLVM**	**Tip C**
Void	VoidType	None
SignedInt(8)	IntType	c_int8
SignedInt(16)	IntType	c_int16
SignedInt(32)	IntType	c_int32
SignedInt(64)	IntType	c_int64
UnsignedInt(1)	IntType	c_bool
UnsignedInt(8)	IntType	c_uint8
UnsignedInt(16)	IntType	c_uint16
UnsignedInt(32)	IntType	c_uint32
UnsignedInt(64)	IntType	c_uint64
Float	FloatType	c_float
Double	FloatType	c_double
String	ArrayType	[c_int8 * dimensiune_string]
Array	ArrayType	[tip_date * dimensiune_string]

5.6.2 Restricții tipuri

Deoarece Python folosește variabile de tip dinamic, iar codul generat trebuie să aibă variabile de tip static, pot interfera câteva probleme.

Cea mai relevantă este problema dimensiunilor pentru tipurile de date String și Array. Odată declarată o variabilă cu acest tip de date, ea nu își mai poate schimba dimensiunea.

O simplă soluție ar fi fost copierea datelor din aceste variabile într-o variabilă nou creată, cu același tip de date, dar de dimensiune diferită. Această soluție este foarte costisitoare din punct de vedere al timpului de execuție, dar și al memoriei, deoarece ar fi trebuit să creăm variabile noi pentru fiecare schimbare de dimensiune. Din aceste motive, am decis că tipurile de date String și Array să nu își poată schimba dimensiunea în timpul execuției.

5.7 Funcții standard

TinyJit oferă suport pentru 2 funcții incorporate ale limbajului Python și anume: print și range, deoarece am considerat că aceste funcții sunt cele mai folosite în programare.

5.7.1 Funcția Print

Funcția print a fost implementată prin asocierea ei cu funcția printf din C, aceasta primind diferite argumente în funcție de tipul de date. Dacă tipul de date trimis către afișare este numeric, funcția va crea în memorie o variabilă globală de tipul string, peste care va aplica argumentul trimis.

Dacă tipul de date trimis este șir de caractere, putem întâlni 2 cazuri:

- Șirul nu se află în memorie și este static, de exemplu:

Figura 5.3: Exemplu print - șir de caractere static

```
1    print("Salut!")
```

  În acest caz, compilatorul creează o variabilă globală de tip string, ce reține șirul de caractere trimis și-l va trimite către printf.

- Șirul este în memorie, adică dorim să afișăm o variabilă de tip string, de exemplu:

Figura 5.4: Exemplu print - șir de caractere variabil

```
1    var = "Hey!"
2    print(var)
```

  În acest caz, compilatorul nu mai creează o variabilă nouă, el trimițând către printf referința variabilei.

5.7.2 Funcția Range

Funcția range a fost implementată pentru a oferi suport instrucțiunii for. Ea poate primi 3 argumente: valoarea de început, valoarea de sfârșit și valoarea de increment. Din aceste 3 argumente doar valoarea de sfârșit este obligatorie, celelalte 2 argumente având valori predefinite, și anume 0 pentru valoarea de început și 1 pentru valoarea de increment. De aici rezultă că toate instrucțiunile următoare au același efect:

Figura 5.5: Comparație range: cu și fără argumente implicite

```
1    for a in range(10):
2    for a in range(0, 10):
3    for a in range(0, 10, 1):
```

Funcția range a fost implementată prin verificarea existenței fiecărui argument și asocierea sa cu o constantă de tip int, asta în cazul în care valoarea sa este constantă. În cazul variabilelor, este preluată valoarea lor și asociată cu variabilele temporare.

5.8 Restricții implementare

5.8.1 JIT

Deoarece TinyJit este de tipul JIT, el se folosește de compilarea leneșă, adică nu va compila orice funcție, ci va compila doar funcțiile apelate. Acest aspect creează o problemă, deoarece dacă dorim să folosim o funcție auxiliară, ea trebuie apelată și înainte de utilizare, pentru a fi compilată. Un simplu exemplu ar fi:

Tabela 5.2: Exemplu eroare "No such function"

Compilare reușită	**Eroare "No such function"**

`@tinyjit`
`def mod(a: t.i64, b: t.i64):`
`  return a % b`
 	
`@tinyjit`
`def call():`
`  return mod(15, 4)`
 	
`mod(1, 1)`
`call()`
	`@tinyjit`
	`def mod(a: t.i64, b: t.i64):`
	`  return a % b`
	
	`@tinyjit`
	`def call():`
	`  return mod(15, 4)`
	
	`#MISSING MOD CALL`
	`call()`

5.8.2 Memorie

O altă restricție a compilatorului creat este din punct de vedere al memoriei. Python poate reține variabile de marime infinită [14], prin metoda sa de a trata *overflow*-ul, adică acesta va muta variabila care a depășit limita într-o altă variabilă de mărime superioară, atât timp cât mai există memorie RAM. Deoarece tipurile de date implementate au o limită superioară fixată, independentă de memoria RAM, ele pot atinge ușor această limită și pot corupe memoria.

5.9 Experimente

În această secțiune voi descrie rezultatele unor comparații între compilatorul dezvoltat și compilatoarele prezentate în capitolul 3, pentru a oferi o viziune amplă asupra capacității compilatorului.

5.9.1 Python-RI

În sub-secțiunea curentă voi descrie comparațiile privind transformăriile codului sursă, efectuate de fiecare compilator. Pentru simplitate, voi compara fiecare instrucțiune în parte, sărind peste instrucțiunile irelevante pentru comparație, precum: instrucțiuni explicate deja în comparațiile trecute, comentarii, tratarea erorilor.

Trebuie menționat faptul că PyPy nu a fost inclus în această comparație, deoarece el nu oferă acces către reprezentarea intermediară.

1. **Atribuire.** Am folosit următorul cod sursă:

Figura 5.6: Cod sursă comparare RI - atribuire

```
1    variabila = 10;
2    return variabila
```

- **Cython:**

Figura 5.7: Comparare RI - atribuire - Cython

```
1    static PyObject *__pyx_n_s_variabila;
2    static const char __pyx_k_variabila[] = "variabila";
3    __pyx_v_variabila = 10;
```

Linia 1 semnifică crearea variabilei, linia 2 atribuirea cu numele variabilei, iar la linia 3 setarea valorii în cadrul variabilei.

- **Numba:**

Figura 5.8: Comparare RI - atribuire - Numba

```
1       store i64 10, i64* %retptr, align 8
```

Numba este foarte optimizat și din acest motiv introduce direct constanta 10 in valoarea de intoarecere.

- **Compilator:**

Figura 5.9: Comparare RI - atribuire - TinyJit

```
1       %".3" = alloca i64
2       store i64 10, i64* %".3"
```

Linia 1 semnifică alocarea de memorie pentru variabilă, iar linia 2 setarea valorii în cadrul variabilei.

2. **Operații simple.** Codul sursă folosit:

Figura 5.10: Cod sursă comparare RI - operații

```
1       variabila = 5
2       variabila2 = 10
3       variabila3 = 3
4       variabila4 = (variabila + variabila2) / variabila3 * 2
5       return variabila4
```

- **Cython:**

Figura 5.11: Comparare RI - operații - Cython

```
1       __pyx_t_1 = PyNumber_Add(__pyx_v_variabila, __pyx_v_variabila2);
2       ...
3       __pyx_t_2 = __Pyx_PyNumber_Divide(__pyx_t_1, __pyx_v_variabila3);
4       ...
5       __pyx_t_1 = PyNumber_Multiply(__pyx_t_2, __pyx_int_2);
```

Prima instrucțiune semnifică adunarea variabilelor ”variabila” și ”variabila2”. Următoarea semnifică împarțirea rezultatului cu ”variabila3”, iar ultima instrucțiune desemnează înmulțirea rezultatului obținut până acum cu constanta 2.

- **Numba:**

Figura 5.12: Comparare RI - operații - Numba

```
1    store double 1.000000e+01, double* %retptr, align 8
```

Numba a calculat constanta rezultată de codul sursă și a introdus-o direct în pointerul de întoarcere.

- **Compilator:**

Figura 5.13: Comparare RI - operații - TinyJit

```
1    %".11" = add i64 %".9", %".10"
2    ...
3    %".13" = sdiv i64 %".11", %".12"
4    %".14" = mul i64 %".13", 2
```

Asemănator cu Cython, prima instrucțiune desemnează suma, a 2 a împărțirea, iar ultima reprezintă înmulțirea.

3. **Instrucțiuni condiționale:**

Figura 5.14: Cod sursă comparare RI - condiționale

```
1    if x > 0:
2        return 1
3    else:
4        return -1
```

- **Cython:**

Figura 5.15: Comparare RI - condiționale - Cython

```
1    __pyx_t_1 = PyObject_RichCompare(__pyx_v_x, __pyx_int_0, Py_GT);
2    ...
3    __pyx_t_2 = __Pyx_PyObject_IsTrue(__pyx_t_1);
4    if (__pyx_t_2) {
5        ...
6        __pyx_r = __pyx_int_1;
7    } else {
8        ...
9        __pyx_r = __pyx_int_neg_1;
10   }
```

Prima linie de cod compară variabila x cu 0, rezultatul fiind verificat la linia 3, apoi este făcut un *branch* în funcție de acest rezultat.

- **Numba:**

Figura 5.16: Comparare RI - condiționale - Numba

```
1   entry:
2   %.5 = icmp sgt i64 %arg.x, 0
3   br i1 %.5, label %B10, label %B14
4   B10:
5   store i64 1, i64* %retptr, align 8
6   ret i32 0
7   B14:
8   store i64 -1, i64* %retptr, align 8
9   ret i32 0
```

Linia 2 compară argumentul x cu 0, iar apoi la linia 3 acesta face un *branch* în funcție de rezultatul întors de linia 2. Blocurile B10 (liniile 4-6) și B14 (liniile 7-9) desemnează cazurile, instrucțiunii **if**, de then și else, ele stocând diferite valori în pointerul de întoarcere.

- **Compilator:**

Figura 5.17: Comparare RI - condiționale - TinyJit

```
1   entry:
2   ...
3   %".7" = icmp sgt i64 %".6", 0
4   br i1 %".7", label %"then", label %"else"
5   exit:
6   %".16" = load i64, i64* %".9"
7   ret i64 %".16"
8   then:
9   store i64 1, i64* %".9"
10  br label %"exit"
11  else:
12  %".12" = sub i64 0, 1
13  store i64 %".12", i64* %".9"
14  br label %"exit"
```

Asemănător cu Numba, diferența este la branch-uri, acestea nu întorc singure un rezultat, ci stochează într-o variabilă valoarea care trebuie rezultată, întoarcerea rezultatului fiind făcută de block-ul exit (liniile 5-7).

4. **Instrucțiuni repetitive:**

Figura 5.18: Cod sursă comparare RI - repetitive

```
1   sum = 0
2   for x in range(nr):
3       sum += x
4   return sum
```

- **Cython:**

Figura 5.19: Comparare RI - repetitive - Cython

```
1   for (;;) {
2       if (__pyx_t_3 >= PyList_GET_SIZE(__pyx_t_2)) break;
3       __pyx_t_1 = PyList_GET_ITEM(__pyx_t_2, __pyx_t_3);
4       ...
5       __Pyx_XDECREF_SET(__pyx_v_x, __pyx_t_1);
6       ...
7       __pyx_t_1 = PyNumber_InPlaceAdd(__pyx_v_sum, __pyx_v_x);
8       ...
9   }
```

Linia 1 desemnează începutul instrucțiunii for, linia 2 verifică condiția de oprire a instrucțiunii repetitive, linia 3 obține următorul element din parcurgere, linia 5 setează valoarea variabilei de parcurgere, iar linia 7 desemnează adunarea variabilei de parcurgere cu variabila ”sum”.

- **Numba:**

Figura 5.20: Comparare RI - repetitive - Numba

```
1   entry:
2       ...
3       %.7 = call i32 @another_function(...)
4       ...
5   B14.endif.preheader:
6       ...
7       %9 = add i64 %spec.select, %8
8       ...
9       %sum.2.0.lcssa = phi i64 [0, %entry], [%10, %B14.endif.preheader]
10      ...
```

Blocul entry (liniile 1-4) face un apel către o altă funcție ce desemnează instrucțiunea for. Linia 7 reprezintă suma dintre variabila ”sum” și variabila de parcurgere, iar linia 9 verifică condiția de oprire și face *branch*.

- **Compilator:**

Codul se află pe următoarea pagină, la figura 5.21.

Linia 3 desemnează începutul instrucțiunii for. Blocul for_init (liniile 4-8) reprezintă setarea variabilelor necesare instrucțiunii for, și anume: valoarea de start, valoarea de oprire și valoarea de increment. Blocul for_cond (liinile 9-12) reprezintă condiția de oprire, adică dacă valoarea de start a depășit valoarea de oprire. Blocul for (liniile 13-20) conține instrucțiuniile din cadrul codului, în acest caz doar calcularea sumei (liniile 15-16), și incrementarea variabilei de start (liniile 18-19). Blocul for_end (liniile 21-24) desemnează sfârșitul instrucțiunii repretitive și conține instrucțiuniile necesare pentru următoarele etape.

Figura 5.21: Comparare RI - repetitive - TinyJit

```
1       entry:
2           ...
3           br label %"for_init"
4       for_init:
5           store i64 0, i64* %".8"
6           store i64 %".11", i64* %".10"
7           store i64 1, i64* %".12"
8           br label %"for_cond"
9       for_cond:
10          ...
11          %".20" = icmp slt i64 %".18", %".19"
12          br i1 %".20", label %"for", label %"for_end"
13      for:
14          ...
15          %".24" = add i64 %".22", %".23"
16          store i64 %".24", i64* %".6"
17          ...
18          %".28" = add i64 %".26", %".27"
19          store i64 %".28", i64* %".8"
20          br label %"for_cond"
21      for_end:
22          %".31" = load i64, i64* %".6"
23          store i64 %".31", i64* %".32"
24          br label %"exit"
```

5.9.2 Comparații *execute*

În această sub-secțiune voi descrie comparațiile privind timpul de execuție și memoria folosită. Pentru a obține rezultate cât mai veridice, calcularea timpului și a memoriei a fost făcută în instanțe diferite. Calcularea memoriei și a timpului de execuție urmează modelul exemplificat din secțiunea 4.6.

Trebuie menționat faptul că PyPy nu oferă suport pentru librăria tracemalloc și, din această cauză, nu am putut să compar memoria folosită de el.

Prima comparație este făcută din punct de vedere al memoriei utilizate, folosind funcția calc_sum, descrisă în figura 4.12. Valoarea memoriei folosite este dată de valoarea maximă a acesteia pe întreaga execuție a programului.

Memoria folosită de programul calc_sum este independentă de variabila number.

Tabela 5.3: Memoria folosită a programului calc_sum

Cython	**Numba**	**TinyJit**
0.0003 MB	11.47 MB	0.23 MB

Din punct de vedere al timpului de execuție, vor fi făcute 2 comparații. Prima comparație utilizând tot funcția calc_sum, iar cea de a 2-a folosind o funcție puțin mai complexă, ce va fi descrisă ulterior.

Tabela 5.4: Timpii de execuție ai programului calc_sum

Number	**Cython**	**PyPy**	**Numba**	**TinyJit**
1000	$< 1e^{-12}$ s	$< 1e^{-12}$ s	0.135 s	0.012 s
1000000	0.026 s	0.002 s	0.137 s	0.014 s
10000000	0.26 s	0.008 s	0.138 s	0.014 s
1000000000	26.59 s	0.73 s	0.14 s	0.015 s

A 2-a comparație a fost facută pe o funcție mai complexă:

Figura 5.22: Funcția more_complex

```
1   @tinyjit
2   def more_complex(number: t.i64):
3       b = 0
4       if number > 0:
5           while number:
6               b *= number
7               number -= 1
8       else:
9           while number:
10              b = b << 0
11              number += 1
12      return b
```

Tabela 5.5: Timpii de execuție ai programului more_complex

Number	**Cython**	**PyPy**	**Numba**	**TinyJit**
1000	$< 1e^{-12}$ s	0.001 s	0.14 s	0.016 s
-1000	$< 1e^{-12}$ s	$< 1e^{-12}$ s	0.15 s	0.015 s
1000000	0.025 s	0.002 s	0.15 s	0.016 s
-1000000	0.027 s	0.002 s	0.15 s	0.014 s
10000000	0.246 s	0.008 s	0.15 s	0.018 s
-10000000	0.245 s	0.009 s	0.15 s	0.015 s
1000000000	24.4 s	0.76 s	0.15 s	0.26 s
-1000000000	24.04 s	0.73 s	0.16 s	0.014 s

Observație

Deoarece LLVM se bazează pe C++, acesta preia modul de calcul și vitezele sale. Se poate observa că pentru Number = 1000000000, compilatorul are un timp de execuție ridicat comparativ cu Number = -1000000000, deoarece el folosește înmulțirea care este scumpă, din punct de vedere al timpului de execuție [8]. Pentru Number = 1073741824 = 2^{30} , compilatorul execută codul în 0.015 s, deoarece el folosește deplasarea(*shift*) cu 30 de biți, care este mult mai rapidă.

Capitolul 6

Concluzii

În această lucrare am prezentat TinyJit, un compilator care este capabil să compileze cod Python, prin intermediul librăriei llvmlite, și oferă utilizatorilor o metodă simplă și rapidă de a face acest lucru cu ajutorul decoratorului implementat. Viteza de compilare a codului este una foarte mare, deoarece programul nu acoperă toate instrucțiuniile Python, spre deosebire de compilatoarele comparate. Memoria folosită de implementare este una scăzută în raport cu sistemele curente. Datorită acestor aspecte, el reprezintă o soluție viabilă.

6.1 Dezvoltări ulterioare

Compilatorul realizat nu se află într-un stadiu complet, el având câteva defecte, unele dintre ele fiind specificate pe parcursul lucrării.

1. **Problema memoriei:** variabilele sunt atribuite, dacă nu este explicitat tipul, la cel mai mare tip de date care le poate memora. Acest aspect are 2 urmări negative, și anume:

   - Dacă utilizatorul dorește să lucreze cu date mici, din punct de vedere valoric, atunci această implementare va ocupa memorie degeaba.
   - Dacă utilizatorul dorește să lucreze cu date foarte mari, acestea pot depăși limitele date de tipurile datelor și vor face *overflow*.

   O soluție a acestei probleme ar putea fi cea implementată de Python, adică el pornește cu cele mai mici tipuri de date care pot reține acea valoare, iar dacă aceasta depășește limita tipului de date, va fi mutată într-o altă variabilă mai spațioasă.

   Această soluție oferă lejeritate și un stil de lucru mai fluent pentru utilizatori, dar timpul de execuție s-ar mări considerabil, iar memoria ar fi și ea afectată, deoarece ar trebui construit un mecanism care să accepte calculul pe tipuri de date mai mari decât cele implementate, fiindcă LLVM nu acceptă acest lucru.

2. **Problema schimbării tipului:** când o variabilă își schimbă tipul de date, fosta ei referință si valoare va rămâne în memorie.

   O soluție la această problemă ar fi utilizarea ulterioară a acestei zone de memorie, dacă este creată o instanță cu acest tip de date.

   O altă soluție ar fi golirea zonei de memorie, dar biblioteca llvmlite nu oferă suport pentru acest lucru.

3. **Problema apelului înainte de compilare:** datorită arhitecturii JIT, funcțiile trebuie compilate înainte de a fi folosite.

   O soluție pentru problema descrisă este compilarea tuturor funcțiilor decorate, înainte de execuția programului, dar acestă implementare are efecte negative asupra timpului de execuție.

   O altă implementare ar fi reținerea în memorie a tuturor funcțiilor decorate și compilarea lor în momentul necesar, dar acestă soluție afectează timpul de execuție și memoria utilizată.

4. **Problema sirurilor de caractere:** deoarece șirurile de caractere au fost implementate doar pentru depanarea de erori, acestea sunt statice.

   O soluție pentru problemă este tratarea acestora ca array-uri de caractere.

5. **Problema tipurilor:** implementarea realizată nu oferă suport pentru toate tipurile și instrucțiunile Python, de exemplu: dicționare, seturi, lambda, assert, etc.

6.2 Utilizare

Calitatea unui produs *software* este dată de experiența utilizatorilor săi. Îmi doresc să îmbunătățesc suportul pentru toate instrucțiunile și tipurile de date ale limbajului Python, controlul memoriei și, eventual, publicarea pe PyPi [1], pentru a permite instalarea și actualizarea rapidă de către utilizatori.

[1] PYPI Manual - https://pypi.org/help/

Bibliografie

[1] Bashir S Abubakar, Abdulkadir Ahmad, Muktar M Aliyu, Muhammad M Ahmad și Hafizu U Uba, „An Overview of Compiler Construction", în (2021).

[2] John Aycock, „A brief history of just-in-time", în *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113.

[3] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn și Kurt Smith, „Cython: The best of both worlds", în *Computing in Science & Engineering* 13.2 (2010), pp. 31–39.

[4] Eli Biham și Jennifer Seberry, „Pypy: another version of Py", în *eSTREAM, ECRYPT Stream Cipher Project, Report* 38 (2006), p. 2006.

[5] Keith D Cooper și Linda Torczon, *Engineering a compiler*, Elsevier, 2011.

[6] Sergio Falcon și Ángel Plaza, „The k-Fibonacci sequence and the Pascal 2-triangle", în *Chaos, Solitons & Fractals* 33.1 (2007), pp. 38–49.

[7] Alexander S. Gillis, „just-in-time compiler (JIT)", în *The Server Side* (2019), URL: https://www.theserverside.com/definition/just-in-time-compiler-JIT (accesat în 9.6.2022).

[8] Matt Godbolt, „Optimizations in C++ Compilers: A practical journey", în *Queue* 17.5 (2019), pp. 69–100.

[9] Alex Holkner și James Harland, „Evaluating the dynamic behaviour of Python applications", în *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, 2009, pp. 19–28.

[10] Siu Kwan Lam, Antoine Pitrou și Stanley Seibert, „Numba: A llvm-based python jit compiler", în *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[11] Chris Lattner și Vikram Adve, „LLVM: A compilation framework for lifelong program analysis & transformation", în *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[12] Mark Lutz, *Learning python: Powerful object-oriented programming*, " O'Reilly Media, Inc.", 2013.

[13] John McCarthy, „Recursive functions of symbolic expressions and their computation by machine, part I", în *Communications of the ACM* 3.4 (1960), pp. 184–195.

[14] Travis E Oliphant, „Python for scientific computing", în *Computing in science & engineering* 9.3 (2007), pp. 10–20.

[15] Frank G Pagan, „Converting interpreters into compilers", în *Software: Practice and Experience* 18.6 (1988), pp. 509–527.

[16] Shoumik Palkar, Sahaana Suri, Peter Bailis și Matei Zaharia, „Exploring the Use of Learning Algorithms for Efficient Performance Profiling", în ().

[17] Ken Thompson, „Programming techniques: Regular expression search algorithm", în *Communications of the ACM* 11.6 (1968), pp. 419–422.

[18] Guido Van Rossum et al., „Python Programming language.", în 41.1 (2007), pp. 1–36, URL: https://www.wikizero.com/en/Python_(language) (accesat în 6.6.2022).

[19] Guido Van Rossum și Fred L Drake Jr, *Python tutorial*, vol. 620, Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[20] Guido vanRossum, „Python reference manual", în *Department of Computer Science [CS]* R 9525 (1995).

[21] April W Wade, Prasad A Kulkarni și Michael R Jantz, „AOT vs. JIT: impact of profile data on code quality", în *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2017, pp. 1–10.