Informatics

Enabling Program Analysis Through Provenance-Preserving Translation into A-Normal Form

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Master program Software Engineering & Internet Computing

by

Marc Goritschnig, BSc
Registration Number 11808600

to the Faculty of Informatics

at the TU Wien

Advisor: Assoc. Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc
Assistance: Dipl.-Ing. Michael Schröder, BSc

Vienna, 3rd May, 2024 [redacted] [redacted]

Marc Goritschnig Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Marc Goritschnig, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Mai 2024

[redacted]

Marc Goritschnig

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub

Acknowledgements

I am thankful that I was able to realize this work at the Faculty of Informatics at the TU Wien, guided by Assoc. Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc, who headed the project, and mentored by Dipl.-Ing. Michael Schröder, BSc, who advised me with his expertise.

I have already had the privilege of working with this team during my bachelor's degree and it was even more delightful to be allowed to contribute during my master's degree in the field of Program Analysis. Through the intensive work on this project, I was not only able to gain immense knowledge, but also grew personally and acquired valuable insights that I will use in my future academic and professional career.

Furthermore, I would like to express my gratitude to all the people who supported me during this time. Your help and support have been instrumental to my success, and I am deeply thankful.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Diese Arbeit untersucht die Transformation von imperativen Hochsprachen in eine intermediäre Repräsentation, die A-Normalform, und zurück. Dies umfasst die Entwicklung einer kompletten Pipeline, die die Daten transformiert, wodurch die Optimierung und Analyse des Codes in A-Normalform erleichtert werden. Die Erkenntnisse aus dieser Analyse werden dann wieder in die imperative Sprache zurücktransformiert.

Normalerweise gehen bei jeder Transformation semantische Informationen verloren. Im Gegensatz dazu präsentiert diese Arbeit eine neuartige Methodik, die eine vertrauenswürdige Rücktransformation in den Originalquellcode ermöglicht, indem wichtige Herkunftsinformationen, die normalerweise verloren gehen würden, beibehalten werden. Basierend auf einer umfangreichen Literaturrecherche wird deutlich, dass diese spezielle Form der Transformation bisher nicht dokumentiert oder veröffentlicht wurde.

Im Verlauf dieser Arbeit werden die untersuchten Forschungsfragen behandelt und gelöst, darunter wie Herkunftsinformationen während des Transformationsprozesses erhalten bleiben können und ob eine bijektive Beziehung zwischen dem imperativen Quellcode und der A-Normalform mit Herkunftsinformationen hergestellt werden kann. Um diese Ziele zu erreichen, haben wir eine Transformationspipeline mit Zwischenrepräsentationen definiert und entwickelt, die vom imperativen Quellcode in Python über abstrakte Syntaxbäume, Kontrollflussgraphen und statische Einzelzuweisungen in A-Normalform führt, die für den Transformationsprozess wesentlich sind.

Die Syntaxdefinitionen und Transformationsregeln für jede Repräsentation werden streng festgelegt, um die Genauigkeit und Konsistenz des Transformationsprozesses zu gewährleisten. Die Implementierung der vorgeschlagenen Methodik erfolgt durch die Entwicklung einer Python-Bibliothek, die die Transformationen erleichtert, und eines Webprojekts zur Visualisierung.

Die Nutzung bestehender Bibliotheken wie „AST-comments“ und „Scalpel“, die erweitert und öffentlich zugänglich gemacht werden, unterstützen die nahtlose Integration des Transformationsprozesses in bestehende Workflows. Die Implementierung wird einer kritischen Evaluation unterzogen, wobei externe Code-Dateien auf Abweichungen zwischen dem Original- und dem rücktransformierten-Code überprüft werden. Die ausführliche Dokumentation des Evaluierungsergebnisses unterstreicht die Effizienz und Zuverlässigkeit der vorgeschlagenen Methodik.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Während die entwickelte Bibliothek eine hervorragende Leistung bei der Erhaltung von Herkunftsinformationen und der präzisen Transformation des Codes in A-Normalform und zurück zeigt, werden bestimmte Einschränkungen anerkannt. Dazu gehört die Transformation von komplexen Strukturen wie Klassen, Exceptions und asynchronem Verhalten, die weiterhin herausfordernd sind und eine weitere Erkundung zur Verbesserung erfordern.

Zusammenfassend trägt diese Arbeit zu einer umfassenden Dokumentation des Transformationsprozesses bei, erläutert Transformationsregeln und Syntaxdefinitionen für die verschiedenen Zwischenrepräsentationen und ist Wegbereiter für die Erhaltung von Herkunftsinformationen zur Ermöglichung der neuartigen Rücktransformation in den Originalquellcode. Durch sorgfältige Implementierung und Evaluation liefert die Arbeit Belege für die Wirksamkeit und Praktikabilität der vorgeschlagenen Methodik und legt eine solide Grundlage für zukünftige Fortschritte in der Programmanalyse und -optimierung dar.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

This thesis explores the transformation of imperative high-level languages into an intermediate representation, the A-normal form, and back. It involves developing an entire pipeline that transforms the data, which thereby facilitates the optimization and analysis of the code in A-normal form. The optimized code is then transformed back into the imperative language.

Usually, semantic information is lost between each transformation. In contrast, this thesis presents a novel methodology that enables the trustworthy backwards transformation into the original source code by retaining crucial provenance information, that would normally be lost. Based on an extensive literature research, it is evident that this particular form of transformation has not been documented or published yet.

In the course of this thesis, the research questions addressed and resolved, include how provenance information can be retained during the transformation process and whether a bijective relation between imperative source code and A-normal form with provenance can be established. To achieve these objectives, we defined and developed a transformation pipeline with intermediate representations, starting from the imperative source code in Python, via abstract syntax trees, control flow graphs and static single assignments into A-normal form, essential for the transformation process.

The syntax definitions and transformation rules for each representation are rigorously established to ensure the accuracy and consistency of the transformation process. The implementation of the proposed methodology is realized through the development of a Python library that facilitates the transformations and a web-project for its visualization.

Leveraging existing libraries such as „AST-comments“ and „Scalpel“, which are extended and made publicly available, assist the seamless integration of the transformation process into existing workflows. The implementation is subjected to critical evaluation with external code files tested for deviations between the original- and the back-propagated-code. The detailed documentation of the evaluation result underscores the efficiency and reliability of the proposed methodology.

While the developed library demonstrates great performance in preserving provenance information and accurately transforming code into A-normal form and back, certain limitations are acknowledged. This includes the transformation of complex structures such as classes, exceptions and asynchronous behavior, which remain challenging and warrant further exploration for enhancement.

In summary, this thesis contributes a comprehensive documentation of the transformation process, elucidates transformation rules and syntax definitions for the various intermediate representations, and pioneers the preservation of provenance information for enabling the novel backwards transformation into the original source code. Through careful implementation and evaluation, the work provides evidence of the effectiveness and practicality of the proposed methodology and establishes a solid foundation for future advances in program analysis and optimization.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

Programming language translations from high-level imperative languages into intermediate representations are an important part of code analysis in various fields such as software engineering, security, performance optimization and many others. Many analysis steps as well as optimizations can be done more easily and effectively using specialized intermediate representations. Alternative representations that offer simplicity include static single assignment and A-normal form.

The static single assignment form describes a representation in which variables are assigned only once throughout the program [Alpern et al., 1988]. This property is desirable as it simplifies various optimizations, including register allocation and efficient data-flow analysis [Hack et al., 2006]. A-normal form represents a transformation into a lambda calculus with `let`-bindings together with the property that all function parameters must be atomic (constants or variables) [Chakravarty et al., 2004]. It facilitates program analysis tasks by simplifying control flow and making data dependencies explicit, which aids optimizations such as constant propagation, dead code elimination performance optimizations and more [Appel, 1998a].

In order to project the results of such analyses back to the original source code, precise and complete provenance information is mandatory. This includes information that is lost throughout the transformation without which an exact backwards transformation is impossible. Using Python as the source language, this information includes different types of variables, syntactic complexity which comes from syntax unique to Python (list comprehension etc.) and more. It serves to establish connections between terms in the intermediate representation and their counterparts in the original surface syntax. It is essential to maintain this linkage not only during transformations of the intermediate representation itself but also during conversions between different representation forms.

Currently, there is no formally defined translation process that describes the transformation of code from Python to A-normal form and, crucially, back to its original form.

Thus, the following research questions will guide the investigation throughout this thesis:

RQ1 How can provenance information be preserved while transforming high-level imperative source code into A-normal form?

RQ2 Can a bijective relation between imperative source code and A-normal form with provenance be established?

We answer these questions throughout the thesis, introducing a new translation approach from Python source code to A-normal form and vice versa. The translation procedure is implemented with our developed library and involves several stages. First, a control flow graph is created based on the original Python source code. Subsequently, it is converted into static single assignment format from where the conversion proceeds into A-normal form.

In all these phases, information is usually lost. Even during the initial transition from the representative Python source code to its abstract syntax tree, details such as syntax, layout, whitespace and comments might not be retained.

Similarly, transitioning from the control flow graph to static single assignment further obscures syntax details: variable names and expressions change, and new variables are introduced. The renaming of variables during compilation can result in breaking the link to original identifiers, complicating code traceability. Eventually, during the transition from static single assignment to A-normal form, the structural information required to access the control flow, introduced by the control flow graph, is discarded. In this phase, the expressions are decomposed into a λ-calculus including `let`-bindings that prescribes a strict order of execution.

Our paper aims to demonstrate a method for converting an imperative language into A-normal form, but also facilitates future exploration of analysis and optimizations in this context. This is because considerable progress has already been made in these areas within A-normal form, and such processes are inherently more straightforward in this intermediate representation compared to imperative programming languages [Hatcliff and Danvy, 1997] [Blume and Appel, 1997] [Chakravarty et al., 2004] [Buszka and Biernacki, 2021].

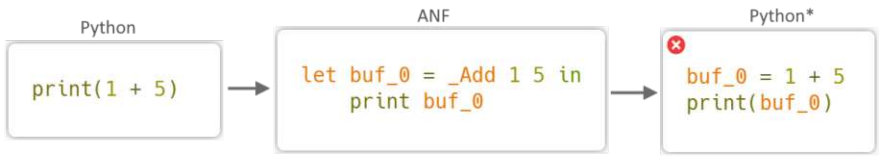
To showcase how the lost information within the transformation affects the result, Figure 1.1 shows an example of Python code, its corresponding A-normal form code and then the back-transformed code in Python. It is clearly visible that without information about `buf_0` just being a temporary variable introduced as an artifact of the translation, the resulting code is different to the original code. However, we do not want to change the code and therefore need certain information that has to be tracked throughout the transformation.

Figure 1.2 illustrates the solution to this issue. During the translation of the code into A-normal form, we maintain supplementary information about the type of the new

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

variable (bv - buffer variable). This information is then utilized when converting the code back to Python, ensuring that the code remains unchanged.

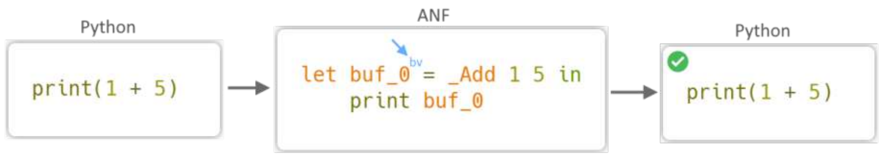


```
print(1 + 5)
```

```
let buf_0 = _Add 1 5 in
  print buf_0
```

```
buf_0 = 1 + 5
print(buf_0)
```

Figure 1.1: Transformation without provenance information



```
print(1 + 5)
```

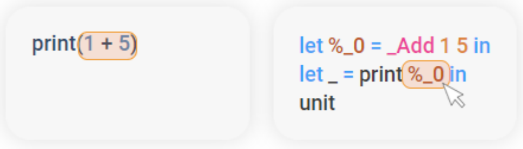
```
let buf_0 = _Add 1 5 in
  print buf_0
```

```
print(1 + 5)
```

Figure 1.2: Transformation with provenance information

This method prevents the loss of information and syntactic complexity of the original source code, allowing modifications to be incorporated into the backwards transformation and resulting in an updated source code in the original programming language.

In addition to the library implementation, we have developed a web project to illustrate the code conversion process. This project involves tracking linking information between the A-normal from and Python code so that we can highlight the corresponding elements in both. Users are able to hover over different code fragments in the A-normal form representation and the corresponding sections in the original Python code are highlighted, as displayed in Figure 1.3. This visualization tool serves several purposes, from teaching students about transformation processes to helping experts understand the transformation processes applied to the code.



```
print(1 + 5)
```

```
let %_0 = _Add 1 5 in
let _ = print %_0 in
unit
```

Figure 1.3: Transformation web-project showcase

In summary, this thesis aims to provide a comprehensive overview of the topic by providing detailed background information, developing and extensively testing a robust transformation pipeline through library implementation and supplementing it with graphical visualization via a web-project. Furthermore, it aims to explore the relevant research questions in detail and ensures the integrity of data and provenance information during transformations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Background

This chapter provides a comprehensive overview of different forms of intermediate representations (IRs). In particular, the representations abstract syntax tree, control flow graph, static single assignment and A-normal form are examined in more detail, since they are essential for the implementation of this thesis. An example code is used within each section to demonstrate and explain the appropriate IRs in more detail.

2.1 Python Abstract Syntax Tree

The source language selected for the transformation in our thesis is Python. However, it is a high-level language which first has to become readable for the machine. It undergoes the lexical analysis to generate tokens, followed by parsing to construct an abstract syntax tree (AST)[1]. The AST is compiled into bytecode, which is then interpreted at runtime, allowing for dynamic execution of Python programs. In accordance with the Python language reference[2], [Mai et al., 2021] and [Tinman, 2015], the IR is created during the compilation process and comprises the following stages: Source Code, Lexemes, Tokens, Parse Trees and ASTs. These stages are delineated by two primary processing steps, as depicted in Figure 2.1.

Lexical Analysis The first step of the lexical analysis includes the scan of the provided source code file and breaks down the code into a stream of lexemes, which are then assigned with tokens. For instance, the token `NEWLINE` indicates the end of a code line. Tokens are also divided into different categories such as identifiers, keywords, literals, operators and delimiters.

[1]https://docs.python.org/3/library/ast.html
[2]https://docs.python.org/3/reference/index.html

TU WIEN Bibliothek Your knowledge hub

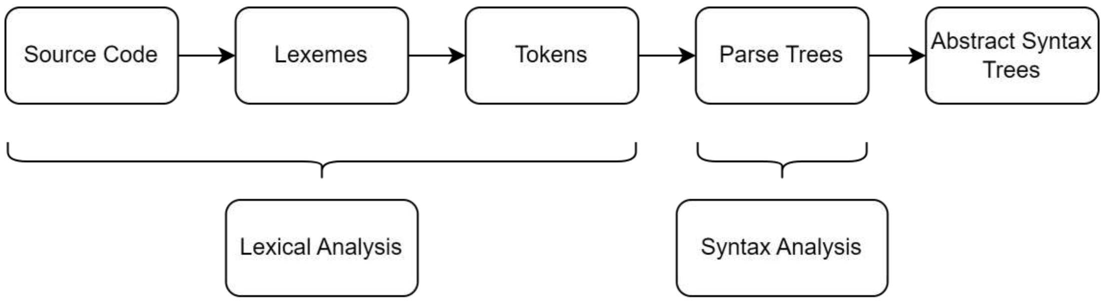


Figure 2.1: Compilation Process to AST

Syntax Analysis The parser, respectively syntax analysis, creates a parse tree and ensures the exact use of the input code elements, whereby the rules of the Python language are strictly adhered to.

Abstract Syntax Tree The output of the syntax analysis, the AST, represents the structure of the code as a hierarchical, tree-like data format. It is abstract, since it only provides the structural information of the code, using the tokens as its nodes rather than including code elements of the original source file. The main program is thus defined as the root node, with the various code elements as child nodes that represent the relationship between the code elements.

Figure 2.2 shows a reduced illustration of the AST tree from the source code in Figure 2.3, simplifying the tasks of analysis, manipulation and optimization. Here, the root node(Body) node contains three sub-nodes, `Assign`, `If` and `Expr`. Each node again consists of other nodes, building the structure of the code within the tree. A complete list of all Python AST nodes is provided in Table A.2, indicating the supported nodes in our transformation (Section 3.3).

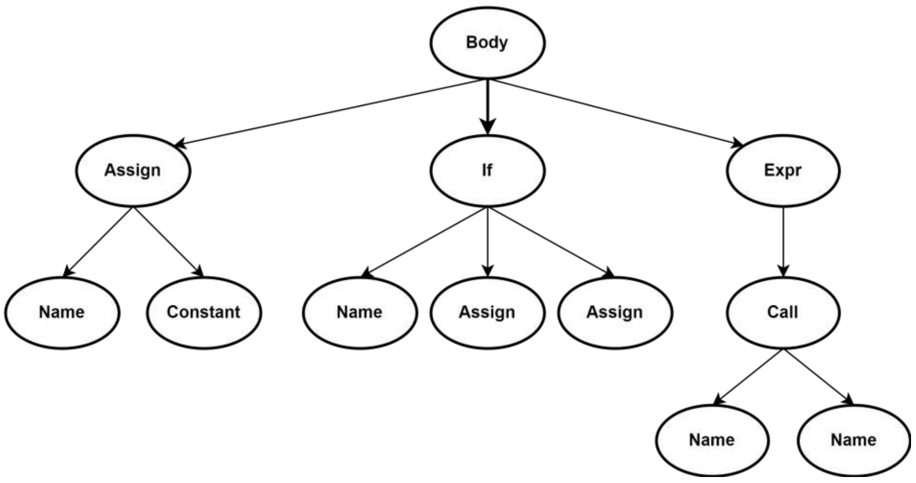


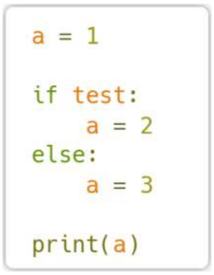
Figure 2.2: AST for the function presented in Figure 2.3

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

2.2 Control Flow Graph

In order to gain a more in-depth understanding of the functionality of the code, to facilitate the analysis of complex programs as well as identifying potential problems or causes of errors, the use of a control flow graph (CFG) provides valuable information. The concept of the CFG, as mentioned in [Lowry and Medlock, 1969], provides details about the control flow of the code by presenting the code in a directed graph (including loops). Thereby, the code blocks are defined as nodes, describing a linear sequence of statements and edges that constitute the transfer between the nodes. Per definition, only the last statement in each block might branch to other blocks, which always leads to the first statement in the target node.

The CFG enhances code accessibility by facilitating traversal through the code, in addition to providing a visual representation of the code's control flow, as demonstrated in the example depicted in Figure 2.3. It depicts instances where two possible paths emerge, leading to a branching process. These paths diverge, creating separate branches with unique nodes, only to merge later. Alternatively, a branch might stop if the program concludes at that point. Considering the `if` statement in the given example, the code could follow either the `if` or `else` block, leading to a branch. Afterwards, both branches lead to the same result (the `print` call) and therefore merge again into one single branch.

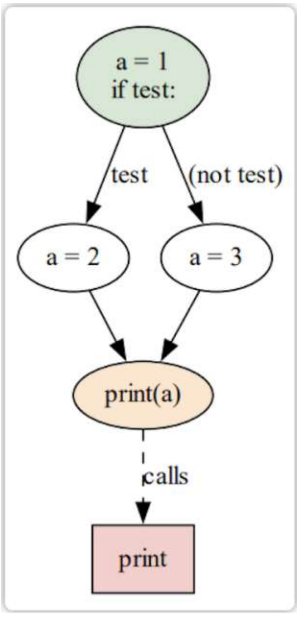


```
a = 1

if test:
    a = 2
else:
    a = 3

print(a)
```

(a) Python code of an if statement



(b) CFG of if statement, including dominator (green) and immediate dominator (orange) with resp. to (red)

Figure 2.3: Python code and its corresponding CFG

Another aspect to be considered in CFG are dominance relations, which define the rules of the execution sequence and specify the dominator definition. According to [Tarjan,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

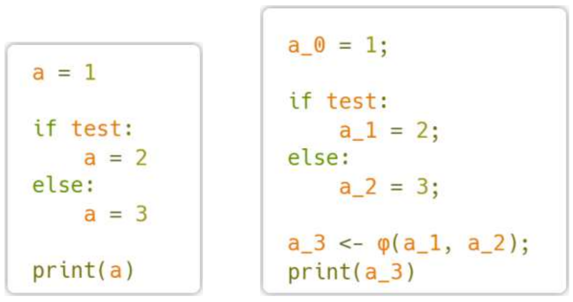
1974], a dominator is defined by the fact that every flow of code execution must pass this node before continuing. This is visualized in Figure 2.3, where the green and orange nodes are dominators of the red node, since every code execution must pass through them before it reaches the red node. The orange node is the immediate dominator of the red node, since it is closest to it. This understanding of statements that must be executed before others serves as a foundation for various optimizations such as common expression elimination, loop identification, and, more broadly, for utilizing CFG in dead code elimination, as demonstrated by [Knoop et al., 1994].

2.3 Static Single Assignment Form

The IR called static single assignment (SSA) utilizes the concept of exclusivity. As Alpern et al. [1988] mentions, each variable must only be assigned once before it is used. To ensure this property, a renaming step must be carried out. It should be mentioned, that the renaming process is not uniquely defined and the procedure could therefore simply give every variable an entirely new name. In order to better clarify the readability and association, the renaming process within this thesis is defined by preserving the original name and adding a post-fix counter.

Continuing our example from before, Figure 2.4 shows the renaming of the variable `a`. The φ-function, declared for `a_3`, is of importance, because whenever there are multiple options available (as in this example, owing to the `if else` statement), variables may receive assignments in various branches. Consequently, it is necessary to determine the source of the correct value for `a_3`. For this purpose, the φ-function is used, taking multiple inputs that represent the values of the target variables from all branches, converging at a given point. Although the function itself does not contain any executable code, it serves as a specification of the SSA behavior.

The SSA form allows for more efficient sparse static analysis, as demonstrated in [Choi et al., 1991]. These analyses include relational domains [Mirliaz and Pichardie, 2022] and partitioned per-variable analyses [Tavares et al., 2014].



```
a = 1

if test:
    a = 2
else:
    a = 3

print(a)
```

(a) Original Source Code

```
a_0 = 1;

if test:
    a_1 = 2;
else:
    a_2 = 3;

a_3 <- φ(a_1, a_2);
print(a_3)
```

(b) SSA Output

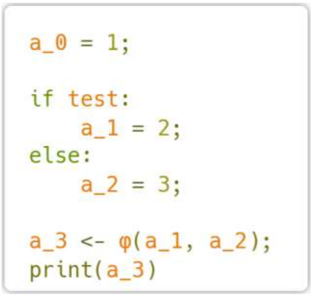
Figure 2.4: Transformation from original Source Code to SSA

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

2.4 A-Normal Form

The so-called A-normal norm (ANF) is a representation of code based on the λ-calculus and `let`-bindings, where function parameters must be atomic. In ANF, all computations are made explicit through `let`-bindings according to [Flanagan et al., 1993], ensuring that every function parameter is either a constant or a variable that is promptly evaluated. This restriction defines the execution order of functions, ensuring clarity in program behavior. Parameters that are not trivial introduce ambiguity regarding their execution order, emphasizing the significance of this property in maintaining predictable program semantics.

Figure 2.5 illustrates the conversion of the ongoing example into ANF. The representation leads to a simplification of the code due to the requirement of ANF to have atomic arguments, resulting in a clear control flow of the program. However, this transformation demonstrates the necessity of retaining block information from the original CFG in Figure 2.3b to accurately represent the behavior of jumps, such as the conditions or function calls of this example.

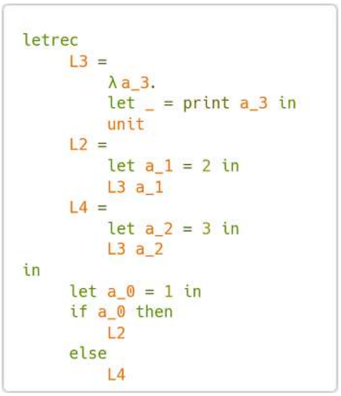


```
a_0 = 1;

if test:
    a_1 = 2;
else:
    a_2 = 3;

a_3 <- φ(a_1, a_2);
print(a_3)
```

(a) SSA representation



```
letrec
    L3 =
        λ a_3.
        let _ = print a_3 in
        unit
    L2 =
        let a_1 = 2 in
        L3 a_1
    L4 =
        let a_2 = 3 in
        L3 a_2
in
    let a_0 = 1 in
    if a_0 then
        L2
    else
        L4
```

(b) ANF representation

Figure 2.5: Transformation from SSA to ANF

The clarity and block information of the control flow improve various program analysis tasks, including data dependency tracking, optimization, constant propagation, and dead code elimination, as described in [Chakravarty et al., 2004]. These benefits provide a solid foundation for subsequent transformations, as stated in [Buszka and Biernacki, 2021].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

Approach

This chapter is intended to describe our approach and development in defining the syntax of all IR languages used and explains how the transformation process between them is distinguished, implemented and utilized by our library for code translation.

Here, we address **RQ1** concerning the preservation of provenance information throughout the transformation process. Each section outlines the unavoidable loss incurred at the particular transformation step. We explore and identify what information we aim to retain and what can be considered as an acceptable loss, using strategic methods to preserve critical information that might otherwise be lost.

The transformation describes the combination of the various IRs to enable a progressive conversion of Python code via AST, CFG and SSA to ANF, and vice versa. Thereby, we also address **RQ2**, which asks whether a bijective transformation from imperative Python source code to ANF is possible. This chapter therefore contains all the necessary definitions and extensions that we include throughout the course of development, how the library uses these definitions and the implementation itself.

Section 3.1 describes the transformation pipeline, which combines the stages mentioned above. The employed IRs are already explained and described in Chapter 2. In order to connect these forms of representation with each other, we have to incorporate extensions such as pre- and post- processing steps, which are reflected in our pipeline. Furthermore, we define the syntax of SSA and ANF to guarantee the language's validity and usage, elaborated in Section 3.2.

Section 3.3 explores the question of which structures should be transformed and which ones should remain unchanged. The basis for making this decision relies on the complexity of the respective code blocks. Section 3.4 describes the fundamental implementation of the pipeline and the utilization of the IRs, along with the extensions devised by us, employing a Python library. Furthermore, it encompasses a web-project for visualizing the transformation steps.

3.1 Transformation Pipeline

Our developed transformation pipeline can be seen in Figure 3.1. It involves a number of IRs to enable the transformation of Python code into an ANF representation. The preprocessing step serves to prepare the code to simplify the transformations into the other representation forms. The backwards transformation from ANF to Python occurs in reverse order, requiring an additional postprocessing step to undo the simplifications made during the preprocessing step. This is done to restore the syntactic complexity within the transformed code resulting from Python-specific syntax, e.g., list comprehensions.

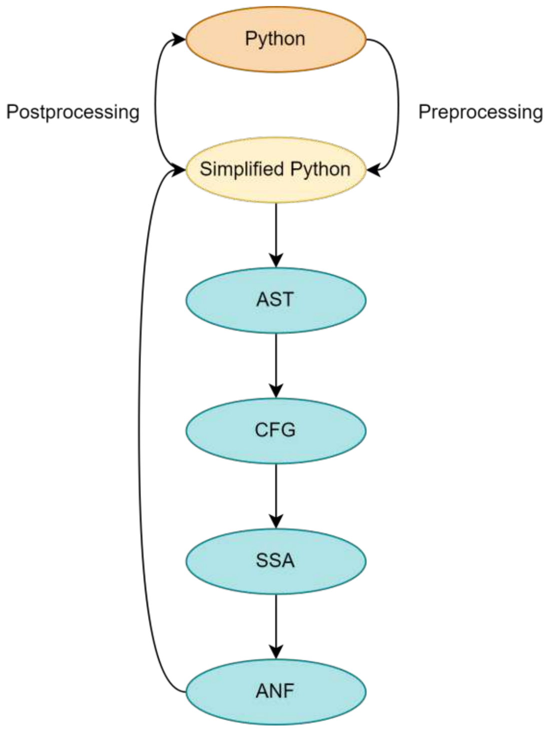
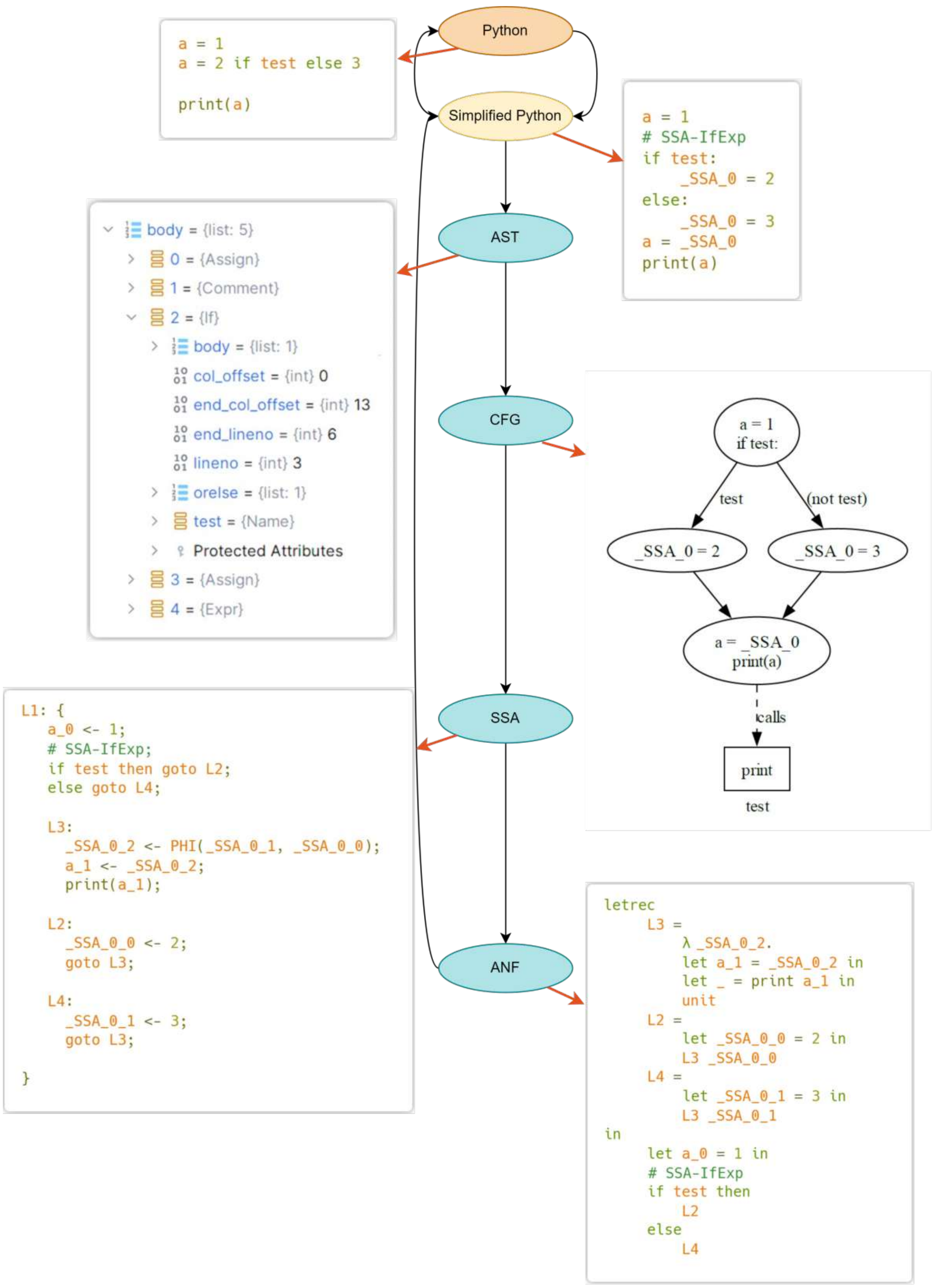


Figure 3.1: Transformation Pipeline

We will now apply the entire transformation pipeline to an example. To do this, we use the same example given in Chapter 2. The individual transformation steps are shown in Figure 3.2. The original source code is depicted under the Python node, along with its simplification in the subsequent step. Although only a snippet of the AST representation is shown, the tree structure of the data format is clearly visible. The next step involves transforming it into a CFG, which is a directed graph with dominators and immediate dominators, as described in Section 3.2. The subsequent transformation will then be performed into SSA, showcasing how the block information of the CFG is kept. The last transition to ANF not only simplifies the representation of the code even more, it also enhances the efficiency of subsequent analysis and understanding processes [Bowman, 2024]. This streamlined representation underscores the efficiency and clarity of ANF, highlighting its utility in facilitating program analysis and optimization efforts.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



```
a = 1
a = 2 if test else 3

print(a)
```

```
a = 1
# SSA-IfExp
if test:
    _SSA_0 = 2
else:
    _SSA_0 = 3
a = _SSA_0
print(a)
```

```
body = {list: 5}
  0 = {Assign}
  1 = {Comment}
  2 = {If}
    body = {list: 1}
    col_offset = {int} 0
    end_col_offset = {int} 13
    end_lineno = {int} 6
    lineno = {int} 3
    orelse = {list: 1}
    test = {Name}
    Protected Attributes
  3 = {Assign}
  4 = {Expr}
```

```
L1: {
   a_0 <- 1;
   # SSA-IfExp;
   if test then goto L2;
   else goto L4;

   L3:
     _SSA_0_2 <- PHI(_SSA_0_1, _SSA_0_0);
     a_1 <- _SSA_0_2;
     print(a_1);

   L2:
     _SSA_0_0 <- 2;
     goto L3;

   L4:
     _SSA_0_1 <- 3;
     goto L3;

}
```

```
letrec
     L3 =
         λ _SSA_0_2.
         let a_1 = _SSA_0_2 in
         let _ = print a_1 in
         unit
     L2 =
         let _SSA_0_0 = 2 in
         L3 _SSA_0_0
     L4 =
         let _SSA_0_1 = 3 in
         L3 _SSA_0_1
in
     let a_0 = 1 in
     # SSA-IfExp
     if test then
         L2
     else
         L4
```

Figure 3.2: Transformation Pipeline on an example

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.2 Syntax Definition

This chapter includes all syntax definitions that are required to describe the transformation pipeline in a formal manner. $\overline{x}$ should be understood as a designation for a variable list of type x. Comments within the SSA syntax are not displayed separately but remain valid and are denoted by a '#' at the beginning, while in ANF, they are indicated by '--'.

3.2.1 SSA

Figure 3.3 defines the syntax for SSA. Within this definition, tracked provenance information is denoted by blue superscript. Here, the additional provenance data only includes positional information; later IRs will include more information as part of the provenance data.

Procedures	p	$::=$	$\textbf{\textit{proc}}\ l(\overline{v})\{b\}\ p \mid b$	one procedure per function
Blocks	b	$::=$	$e^{\textbf{pos}}$	terms
		$\mid$	$b;\ l{:}e^{\textbf{pos}}$	block
		$\mid$	$b_1;\ l{:}\{b_2\}$	dominating block
Terms	e	$::=$	$v \leftarrow \varphi(\overline{v});\ e^{\textbf{pos}}$	phi assignment
		$\mid$	$v \leftarrow f;\ e^{\textbf{pos}}$	assignment
		$\mid$	$\textbf{\textit{goto}}\ l;$	jump to block
		$\mid$	$\textbf{\textit{ret}}\ f;$	return variable
		$\mid$	$\textbf{\textit{if}}\ f\ \textbf{\textit{then}}\ e_1^{\textbf{pos}}\ \textbf{\textit{else}}\ e_2^{\textbf{pos}}$	branching
Function Parameter	f	$::=$	$v^{\textbf{pos}} \mid v^{\textbf{pos}}(\overline{f})$	variable or function call
Values	v	$::=$	$x^{\textbf{pos}} \mid \text{constant}^{\textbf{pos}}$	
Label			$l^{\textbf{pos}}$	
Variables			$x, y, z \ldots$	

Figure 3.3: SSA Syntax Definition

To further simplify the transformation from SSA to ANF, we integrate the concept of CFG into SSA. This includes the principle of **Blocks** and **Procedures**, as outlined in Figure 3.3, gathering information from CFG.

Our definition closely aligns with that of [Chakravarty et al., 2004], except for one significant deviation. In the cited paper, the SSA is restricted to contain only atomic function parameters (constants or variables), a property corresponding to ANF. In our approach, however, we do not subject SSA to this restriction. Instead, we apply this property one step later in our pipeline (SSA → ANF) to preserve the generality of the library. It allows for a more flexible implementation in terms of the enhancement to other

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

languages, due to more degrees of freedom within the transformation from the original language to SSA. The difference becomes evident through our definition in Figure 3.3, using the variable `f` within assignments, function calls, return values and `if` blocks.

The syntax itself, along with the ANF definition, is recursive. Thereby, the outermost level consists of a list of procedures terminated by a list of blocks. Each procedure and dominating block begins with a list of terms describing the initial block to be executed within its CFG, followed by a list of subsequent blocks. This is evident through the declaration of block label `b`, which generates blocks from bottom to top (`b::=b;x:e` and `b::=b;x:{`b_2`}`) and only terminates in a term (which recursively generates a term list). Finally, within each block (excluding the ones building a new scope), there is a list of terms.

The critical aspect here relates to the fact that we already integrate the structure and order of the blocks from the CFG into the SSA representation, as briefly mentioned by [Chakravarty et al., 2004], to pass on this information for the subsequent step of transforming SSA to ANF. This integration is achieved by constructing a dominator tree based on the given CFG, following the methodology outlined by [Cytron et al., 1991]. Given the declaration of the dominator tree, each node appears only once in the tree and forms a vertical structure of dependencies between the blocks, which we represent in our SSA code.

3.2.2 ANF

The syntax definition we developed for ANF is specified in Figure 3.4. This type of syntax allows for the structuring and simplification of declarations according to the principles described in Chapter 2.4. Consequently, a reduction in complexity is achieved by using the A-reduction property of ANF, which assumes that function parameters may only be atomic (constants and variables). As a result, a clear and definite sequence for the execution of functions is established.

The SSA and ANF syntax allow for an easy replacement of the original source language, which is currently Python, without having to change the back-end of our pipeline. This is ensured by separating the language-specific details from the transformation from SSA to ANF, thus guaranteeing independence from the source language.

Within the syntax declaration in Figure 3.3, the provenance information transferred from SSA to ANF is highlighted in blue superscript. The necessity of this information and its significance will be described in Section 3.3.3.

Additional insights into the importance of integrating these syntax definitions to facilitate the transformation process and their practical application will be provided in Section 3.4.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Variables			$x, y, z \ldots$	
Values	v	$::=$	$c^{\mathbf{c}\mid\mathbf{ret}} \mid x^{\mathbf{v}\mid\mathbf{fv}\mid\mathbf{ret}}$	constants | variable
Terms	e	$::=$	$v^{\mathbf{c}\mid\mathbf{v}\mid\mathbf{fv}\mid\mathbf{ret}}$	value
		$\mid$	$v^{\mathbf{fv}}\ \overline{v}^{\mathbf{v}\mid\mathbf{c}\mid\mathbf{ret}}$	application
		$\mid$	$\textbf{let}\ x^{\mathbf{v}\mid\mathbf{bv}} = v^{\mathbf{v}\mid\mathbf{c}\mid\mathbf{ret}}\ \textbf{in}\ e$	let binding
		$\mid$	$\textbf{let}\ x^{\mathbf{v}\mid\mathbf{bv}} = v^{\mathbf{fv}}\ \overline{v}^{\mathbf{v}\mid\mathbf{c}\mid\mathbf{ret}}\ \textbf{in}\ e$	let binding
		$\mid$	$\textbf{letrec}\ \overline{f}^{\mathbf{v}\mid\mathbf{lc}}\ \textbf{in}\ e$	letrec binding scope
		$\mid$	$\textbf{if}\ v^{\mathbf{v}\mid\mathbf{c}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	branching within lambda
	f	$::=$	$x^{\mathbf{v}\mid\mathbf{bv}} = e \mid x^{\mathbf{v}\mid\mathbf{bv}} = f_2$	binding within scope
	f_2	$::=$	$\lambda\ x^{\mathbf{v}\mid\mathbf{bv}}\ .\ f_2 \mid \lambda\ x^{\mathbf{v}\mid\mathbf{bv}}\ .\ e$	abstraction

Figure 3.4: ANF Syntax Definition

3.3 Transformation Rules and Guidelines

In this section, we will present the formal and readable declarations of all our transformations, along with the pre- and postprocessing steps. These definitions will illustrate the input tabularly on the left-hand side, followed by the corresponding output on the right.

Our transformation does not encompass every AST node from Python. For clarity, we provide a breakdown of all supported constructs in Table A.2 (see in Appendix). Some nodes are unsupported due to concerns regarding complexity. For instance, `import` nodes may reference local files, inaccessible when analyzing a single code file, or libraries whose versions may vary between code composition and analysis. Additionally, asynchronous nodes are excluded due to challenges in representing asynchronous behavior in ANF.

3.3.1 Preprocessing

Having established the syntax, we now proceed to the transformation process itself. The transformation is starting from Python code and its corresponding representation in AST, displaying the entire code with nodes in a tree structure.

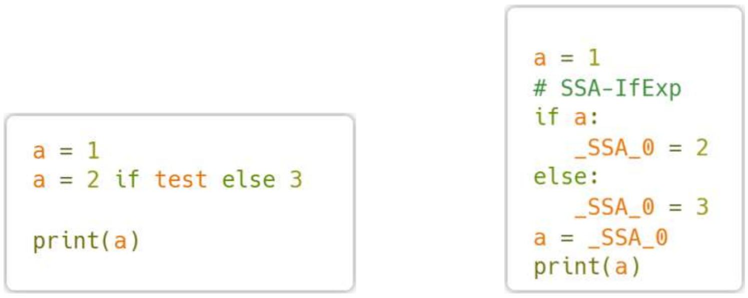
The code not only includes simple assignments, function calls and declarations, but also more sophisticated structures such as list comprehensions. However, since these complex code pieces cannot be directly represented in SSA and ANF, a preprocessing step is necessary to get a more straightforward and less complex code structure (e.g., assignments, conditions, loops and so on).

The equations below define how various AST nodes from Python are preprocessed and converted. It is evident that not all nodes are modified during the process of preprocessing.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

This is due to the fact that they do not need any special treatment before the next translation part is continued.

The translation of code is applied according to (Preprocessing), as can be seen in Figure 3.5. Each time, a statement or expression is found that matches one of the specified forms of AST nodes as input, we replace its code with the first entry inside the brackets ⟨⟩ and place the second part preceding the current statement in order to execute this code before the changed statement. This process continues until there are no more changes to be made, which means that the preprocessing step is completed.



```python
a = 1
a = 2 if test else 3

print(a)
```

(a) Original Source Code (Python)

```python
a = 1
# SSA-IfExp
if a:
    _SSA_0 = 2
else:
    _SSA_0 = 3
a = _SSA_0
print(a)
```

(b) Preprocessing step, applying IfExp

Figure 3.5: Transformation from Python to simplified Python

After this preliminary processing step, the AST tree is passed on to the subsequent translation phase.

$$[\![S,\ \overline{S}]\!] = \begin{cases} [\![R,\ \hat{S},\ \overline{S}]\!] & \text{if } [\![S]\!] = \langle \hat{S}; R\rangle \\ [\![R,\ S[e:=\hat{e}],\ \overline{S}]\!] & \text{if there is an expr e in S such that} [\![e]\!] = \langle \hat{e}; R\rangle \\ S;\ [\![\overline{S}]\!] & \text{otherwise} \end{cases} \tag{Preprocessing}$$

$$[\![[f(i,\ i_2,\ \dots)\ \textbf{for}\ i\ \textbf{in}\ x\ \textbf{for}\ i_2\ \textbf{in}\ x_2\ \dots]]\!] =$$

$$\left\langle \text{\_SSA\_0},\quad \begin{array}{l} \text{\# SSA-ListComp} \\ \text{\_SSA\_0} = [] \\ \textbf{for}\ i\ \textbf{in}\ x: \\ \quad \textbf{for}\ i_2\ \textbf{in}\ x_2: \\ \quad\quad \dots \\ \quad\quad\ \text{\_SSA\_0.append}(f(i,\ i_2,\ \dots)) \end{array} \right\rangle \tag{ListComp}$$

$$[\![\{f(i,\ i_2,\ \dots)\ \textbf{for}\ i\ \textbf{in}\ x\ \textbf{for}\ i_2\ \textbf{in}\ x_2\ \dots\}]\!] =$$

$$\left\langle \text{\_SSA\_0},\quad \begin{array}{l} \text{\# SSA-SetComp} \\ \text{\_SSA\_0} = \{\} \\ \textbf{for}\ i\ \textbf{in}\ x: \\ \quad \textbf{for}\ i_2\ \textbf{in}\ x_2: \\ \quad\quad \dots \\ \quad\quad\ \text{\_SSA\_0.add}(f(i,\ i_2,\ \dots)) \end{array} \right\rangle \tag{SetComp}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$[\![\{f(i,\ i_2,\ \ldots):\ f_2(i,\ i_2,\ \ldots)\ \textbf{for}\ i\ \textbf{in}\ x\ \textbf{for}\ i_2\ \textbf{in}\ x_2\ \ldots\}]\!] =$$

$$\left\langle \_\text{SSA}\_0,\quad \begin{array}{l} \#\ \textbf{SSA-DictComp} \\ \_\text{SSA}\_0 = \{\} \\ \textbf{for}\ i\ \textbf{in}\ x: \\ \quad \textbf{for}\ i_2\ \textbf{in}\ x_2: \\ \quad\quad \ldots \\ \quad\quad \_\text{SSA}\_0[f(i,\ i_2,\ \ldots)] = f_2(i,\ i_2,\ \ldots) \end{array} \right\rangle \tag{DictComp}$$

$$[\![\ x_1, \ldots,\ x_n = y_1, \ldots\ y_n]\!] =$$

$$\left\langle \varnothing,\quad \begin{array}{l} \#\ \textbf{SSA-Tuple} \\ \_\text{SSA}\_0 = \_\text{new}\_\text{tuple}\_n(y_1, \ldots, y_n) \\ x_1 = \_\text{Tuple}\_\text{Get}(\_\text{SSA}\_0,\ 0) \\ \ldots \\ x_n = \_\text{Tuple}\_\text{Get}(\_\text{SSA}\_0,\ n-1) \end{array} \right\rangle \tag{Assign(Tuple)}$$

$$[\![\textbf{lambda}\ x\textbf{:}\ f(x)]\!] =$$

$$\left\langle \_\text{SSA}\_0,\quad \begin{array}{l} \textbf{def}\ \_\text{SSA}\_0(x)\textbf{:} \\ \quad \#\ \textbf{SSA-Lambda} \\ \quad \textbf{return}\ f(x) \end{array} \right\rangle \tag{Lambda}$$

$$[\![x := y]\!] =$$

$$\left\langle x,\quad \begin{array}{l} \#\ \textbf{SSA-NamedExpr} \\ x = y \end{array} \right\rangle \tag{NamedExpr}$$

$$[\![x\ \bullet\!= y]\!] =$$

$$\left\langle x = x\ \bullet\ y,\quad \#\ \textbf{SSA-AugAssign} \right\rangle \quad \text{where} \bullet \in \left\{+, -, *, /\right\} \tag{AugAssign}$$

$$[\![x\ \textbf{if}\ t\ \textbf{else}\ y]\!] =$$

$$\left\langle \_\text{SSA}\_0,\quad \begin{array}{l} \#\ \textbf{SSA-IfExp} \\ \textbf{if}\ t\textbf{:} \\ \quad \_\text{SSA}\_0 = x \\ \textbf{else:} \\ \quad \_\text{SSA}\_0 = y \end{array} \right\rangle \tag{IfExp}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$[\![x\mathbf{:}\ type = y]\!] =$

$$\left\langle x = y,\quad \textbf{\# SSA-AnnAssign|[type]|0} \right\rangle \tag{AnnAssign}$$

$[\![x[s] = y]\!] =$

$$\left\langle x = \textbf{dict\_set}(x,\ s,\ y),\quad \textbf{\# SSA-SubscriptSet} \right\rangle \tag{Assign}$$

$[\![x.y(args)]\!] =$

$$\left\langle \begin{array}{ll} & \textbf{\# SSA-Attribute} \\ \textbf{\_SSA\_}0, & \textbf{\_SSA\_}0 = \textbf{\_obj\_}y(x,\ args) \end{array} \right\rangle \tag{Attribute}$$

$[\![x[y](args)]\!] =$

$$\left\langle \begin{array}{ll} & \textbf{\# SSA-FuncSub} \\ \textbf{\_SSA\_}0(args), & \textbf{\_SSA\_}0 = x[y] \end{array} \right\rangle \tag{Call}$$

$[\![\textbf{import}\ x]\!] =$

$$\left\langle \textbf{\# import}\ x,\quad \textbf{\# SSA-Import} \right\rangle \tag{Import}$$

$[\![\textbf{class}\ C\mathbf{:}\ body]\!] =$

$$\left\langle \begin{array}{ll} \textbf{\# class}\ C\mathbf{:}\ body, & \textbf{\# SSA-ClassStart} \\ \textbf{\# SSA-ClassEnd} & \end{array} \right\rangle \tag{ClassDef}$$

$[\![\textbf{for}\ i\ \textbf{in}\ a\mathbf{:}\ body]\!] =$

$$\left\langle \begin{array}{r} \textbf{for}\ i\ \textbf{in}\ a\mathbf{:} \\ body \\ \textbf{\# SSA-Placeholder} \end{array}, \varnothing \right\rangle \tag{For}$$

$[\![\textbf{for}\ i_1, i_2, \ldots\ \textbf{in}\ a\mathbf{:}\ body]\!] =$

$$\left\langle \begin{array}{l} \textbf{\# SSA-ForTuple} \\ \textbf{for}\ x\ \textbf{in}\ a\mathbf{:} \\ \quad i_1 = x[0] \\ \quad i_2 = x[1] \\ \quad \ldots \\ \quad body \end{array} , \varnothing \right\rangle \tag{ForTuple}$$

Figure 3.6: Definition of the preprocessing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Provenance Information

The preprocessing step simplifies the given source input in order to enable the IR of complex code structures, such as assignments, conditions and so on. However, it is associated with the drawback of losing syntactic complexity of the Python-specific syntax (e.g. list comprehensions). To avoid this loss, we add a comment marker to each processed code fragment (beginning with '#' and colored in blue), as depicted in the equations above (Figure 3.6). By doing so, these markers will retain sufficient information about the original code, enabling us to reconstruct it during the backwards transformation.

3.3.2 AST $\rightarrow$ SSA (via CFG)

Following the preprocessing step, the code is simplified, which results in a streamlined version. Our next task involves translating this simplified code into the form of SSA via CFG, as can be seen in Figure 3.8. But to do so, we first have to transform the preprocessed segments into CFG.

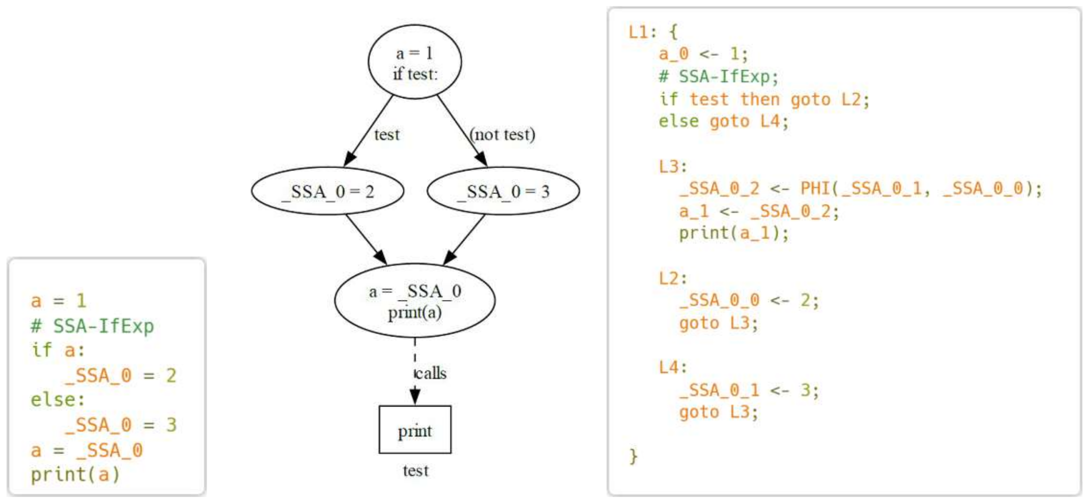
Within the generated AST nodes, besides encapsulating the code, we also capture important provenance information. This includes details such as the original appearance of the code and the naming of function parameters. Such details play an important role in the subsequent phases of our analysis and transformations and ensure the accuracy and completeness of the entire process. This means that within this transformation, provenance information is not exclusively used, but extracted for the next transformation steps in our pipeline, discussed in Sections 3.3.3 and 3.3.4.

Using the AST tree, we systematically iterate through all nodes to generate the CFG, see Figure 3.7a to 3.7b. During this process, we maintain a list of φ-nodes per block, using the Scalpel library from Li et al. [2022], which is crucial for handling variable assignments when merging blocks. In addition, we perform the renaming, a fundamental feature of SSA, simultaneously. This procedure adheres closely to the definition in [Cytron et al., 1991]. The details of the implementation are explained in Section 3.4.4.

The resulting CFG is comprehensive and not only encompasses the main code flow, but also separate CFGs for each identified function. These function-CFGs may contain inner function definitions that lead to further CFGs nested within them. Thereby, each CFG consists of a list of connected blocks, where each block contains a potentially empty list of φ-assignments, a set of terms and a list of function-CFGs which are assembled in the same way.

Figure 3.8 contains the resultant specifications for our transformation from CFG into SSA, containing the details of the AST. The definitions are structured to interpret code of a specific input type, shown in bold on the left, along with additional arguments. These inputs are then translated into the corresponding SSA code, which is displayed on the right-hand side. The transformation adheres to the syntax and scoping rules described in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



```
a = 1
# SSA-IfExp
if a:
    _SSA_0 = 2
else:
    _SSA_0 = 3
a = _SSA_0
print(a)
```

```
L1: {
   a_0 <- 1;
   # SSA-IfExp;
   if test then goto L2;
   else goto L4;

   L3:
     _SSA_0_2 <- PHI(_SSA_0_1, _SSA_0_0);
     a_1 <- _SSA_0_2;
     print(a_1);

   L2:
     _SSA_0_0 <- 2;
     goto L3;

   L4:
     _SSA_0_1 <- 3;
     goto L3;
}
```

(a) Simple Python (b) CFG (c) SSA

Figure 3.7: Transformation from simplified Python to SSA (via CFG)

Section 3.2 and ensures that the blocks are translated in accordance with the dominator tree information of the given CFG and scoped accordingly.

Utilizing this structure, we determine the execution order of blocks, see Figure 3.7b to 3.7c. Those that always precede others are declared in an outer scope. Blocks dominated by a particular block are then declared within it. If such a block dominates multiple others, they are declared at the same level within it. This organization is essential because earlier code blocks may reference later ones, necessitating their definition first. Following the definition of the dominator tree, if any subsequently created block would reference the earlier block (resulting in an undefined reference call), the block has to dominate the earlier block and must therefore be located in an outer area, which eliminates the issue itself. Additionally, inner blocks may have their own dominated blocks, creating a nested structure of block definitions. Such a scope of a block, along with its code and dominated blocks, is captured with braces in our syntax definition, prefixed by the block label b.

The first segment of our transformation definition outlines the source language as an extension of the existing AST nodes. In this transformation we work with a CFG, which is enriched by AST data on the left side (referring to Figure 3.8) and associate each node with a specific fragment of SSA code. The extension of the AST nodes (**AST***) by customized nodes is formulated as follows:

CFG Node that encapsulates all **F_CFG**s parsed from functions and a block list comprising all blocks within the main CFG.

F_CFG Representation of a CFG specific to a function. It includes a function name, a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

parameter list, a collection of **F_CFG**s for functions declared within the current one and a list of blocks defined within this function.

Block Block from the CFG, containing a list of φ assignments and a list of statements.

DomBlock **Block** which dominates other blocks, possesses a list of blocks rather than a statement list.

In all transformation definitions, colored text corresponds to the syntax of a specific language (**AST***, ***SSA***, **ANF**, **Python**).

PS (Python AST to SSA)

$$\llbracket \mathbf{CFG}\ f_1, .., f_n\ \overline{b} \rrbracket = \llbracket f_0 \rrbracket \dots \llbracket f_n \rrbracket \llbracket \overline{b} \rrbracket \tag{3.1}$$

$$\llbracket \mathbf{F\_CFG}\ v\ \overline{e}\ f_1, .., f_n\ \overline{b} \rrbracket = \llbracket f_0 \rrbracket \dots \llbracket f_n \rrbracket \textbf{\textit{proc}}\ v\textbf{\textit{(}}\llbracket e \rrbracket\textbf{\textit{)\{}}\llbracket \overline{b} \rrbracket\textbf{\textit{\}}} \tag{3.2}$$

Blocks

$$\begin{aligned}
\llbracket \mathbf{Block}\ \overline{\varphi}\ (\mathbf{For}\ e_1\ e_2\ \overline{e_3}),\ b_2,\ \dots,\ b_n \rrbracket = {} & \llbracket \overline{\varphi} \rrbracket_{b_1} \\
& \quad \textbf{\#SSA-FOR;} \\
& \quad \textbf{\textit{\_SSA\_}}0 \leftarrow \llbracket e_2 \rrbracket\textbf{\textit{;}} \\
& \quad \#(e_1)\textbf{\textit{\_0}} \leftarrow \textbf{\textit{next(\_SSA\_}}0\textbf{\textit{);}} \\
& \quad \textbf{\textit{goto}}\ \#(b)\textbf{\textit{\_2;}} \\
& \#(b)\textbf{\textit{\_2:}} \\
& \quad \#(e_1)\textbf{\textit{\_1}} \leftarrow \varphi(\#(e_1)\textbf{\textit{\_0}}, \#(e_1)\textbf{\textit{\_1\_2);}} \\
& \quad \textbf{\textit{\_SSA\_}}1 \leftarrow \textbf{\textit{\_IsNot(}}\#(e_1)\textbf{\textit{\_1, None);}} \\
& \quad \textbf{\textit{if \_SSA\_}}1\ \textbf{\textit{then goto}}\ \#(b \downarrow 0)\textbf{\textit{;}} \\
& \begin{cases} \textbf{\textit{else goto}}\ \#(b \downarrow 1)\textbf{\textit{;}} & \#(b \downarrow 1) \text{ exists} \\ \textbf{\textit{else ret;}} & \text{otherwise} \end{cases} \\
& \llbracket b_2\ +\ \mathbf{Call}(\#(e_1)\textbf{\textit{\_1\_2}}),\ b_3,\ \dots,\ b_n \rrbracket
\end{aligned} \tag{3.3}$$

$$\llbracket b_1, \dots, b_n \rrbracket = \llbracket b_1 \rrbracket_{b_1}\textbf{\textit{;}}\dots\textbf{\textit{;}}\llbracket b_n \rrbracket_{b_n} \tag{3.4}$$

$$\llbracket \mathbf{Block}\ \overline{\varphi}\ \overline{s} \rrbracket_b = \#(b)\textbf{\textit{:}}\ \llbracket \overline{\varphi} \rrbracket_b\textbf{\textit{;}}\llbracket \overline{s} \rrbracket_b \tag{3.5}$$

$$\llbracket \mathbf{DomBlock}\ \overline{\varphi}\ \overline{b_1} \rrbracket_b = \#(b)\textbf{\textit{:\{}}\ \llbracket \overline{\varphi} \rrbracket_b\textbf{\textit{;}}\llbracket \overline{b_1} \rrbracket\textbf{\textit{\}}} \tag{3.6}$$

Statements

$$\llbracket s_1, \dots, s_n \rrbracket_b = \llbracket s_1 \rrbracket_b\textbf{\textit{;}}\dots\textbf{\textit{;}}\llbracket s_n \rrbracket_b \tag{3.7}$$

$$\llbracket \mathbf{Assign}\ e_1\ e_2 \rrbracket_b = \llbracket e_1 \rrbracket_b \leftarrow \llbracket e_2 \rrbracket_b \tag{3.8}$$

$$\llbracket \mathbf{AnnAssign}\ e_1\ e_2 \rrbracket_b = \llbracket e_1 \rrbracket_b \leftarrow \llbracket e_2 \rrbracket_b \tag{3.9}$$

$$\begin{aligned}
\llbracket \mathbf{If}\ e \rrbracket_b = {} & \textbf{\textit{if}}\ \llbracket e \rrbracket_b\ \ \textbf{\textit{then goto}}\ \#(b \downarrow 0)\textbf{\textit{;}} \\
& \quad \begin{cases} \textbf{\textit{else goto}}\ \#(b \downarrow 1)\textbf{\textit{;}} & \#(b \downarrow 1) \text{ exists} \\ \textbf{\textit{else ret;}} & \text{otherwise} \end{cases}
\end{aligned} \tag{3.10}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$\llbracket \textbf{While } e \rrbracket_b = \textit{if } \llbracket e \rrbracket_b \textit{ then goto } \#(b \downarrow 0)\textit{;} \begin{cases} \textit{else goto } \#(b \downarrow 1)\textit{;} & \#(b \downarrow 1) \text{ exists} \\ \textit{else ret;} & \text{otherwise} \end{cases} \tag{3.11}$$

$$\llbracket \textbf{Expr } e \rrbracket_b = \llbracket e \rrbracket_b \tag{3.12}$$

$$\llbracket \textbf{Return } e \rrbracket_b = \textit{ret } \llbracket e \rrbracket_b \tag{3.13}$$

$$\llbracket \textbf{Delete } \bar{e} \rrbracket_b = \textit{delete\_n(}\llbracket \bar{e} \rrbracket_b\textit{)} \text{ where } n = len(\bar{e}) \tag{3.14}$$

$$\llbracket \textbf{Raise } ex \rrbracket_b = \textit{\_Raise(}ex\textit{(}\llbracket \overline{ex.args} \rrbracket_b\textit{))} \tag{3.15}$$

$$\llbracket \textbf{Raise } ex\ e \rrbracket_b = \textit{\_Raise\_2(}ex\textit{(}\llbracket \overline{ex.args} \rrbracket_b\textit{), } \llbracket \bar{e} \rrbracket_b\textit{)} \tag{3.16}$$

$$\llbracket \textbf{Assert } e_1 \rrbracket_b = \textit{\_Assert(}\llbracket e_1 \rrbracket_b\textit{)} \tag{3.17}$$

$$\llbracket \textbf{Assert } e_1\ e_2 \rrbracket_b = \textit{\_Assert\_2(}\llbracket e_1 \rrbracket_b\textit{,}\llbracket e_2 \rrbracket_b\textit{)} \tag{3.18}$$

$$\llbracket \textbf{Pass} \rrbracket_b = \textit{\_Pass()} \tag{3.19}$$

$$\llbracket \textbf{Break} \rrbracket_b = \textit{\_Break()} \tag{3.20}$$

$$\llbracket \textbf{Continue} \rrbracket_b = \textit{\_Continue()} \tag{3.21}$$

$$\llbracket \textbf{Comment } v \rrbracket_b = \#v \tag{3.22}$$

Expression

$$\llbracket e_1, ..., e_n \rrbracket_b = \llbracket e_1 \rrbracket_b\textit{,}...\textit{,}\llbracket e_n \rrbracket_b \tag{3.23}$$

$$\llbracket \textbf{NamedExpr } e_1\ e_2 \rrbracket_b = \llbracket e_1 \rrbracket_b = \llbracket e_2 \rrbracket_b \tag{3.24}$$

$$\llbracket \textbf{BinOp } e_1\ op\ e_2 \rrbracket_b = \textit{\_op(}\llbracket e_1 \rrbracket_b\textit{,}\llbracket e_1 \rrbracket_b\textit{)} \tag{3.25}$$

$$\llbracket \textbf{BoolOp } op\ e_1, ..., e_n \rrbracket_b = \textit{\_op(}\llbracket \textbf{BoolOp } op\ [e_1, ..., e_{n-1}] \rrbracket_b\textit{,}\llbracket e_n \rrbracket_b\textit{)} \tag{3.26}$$

$$\llbracket \textbf{UnaryOp } op\ e \rrbracket_b = \textit{\_op(}\llbracket e \rrbracket_b\textit{)} \tag{3.27}$$

$$\llbracket \textbf{Call } \bar{e}\ f \rrbracket_b = f\textit{(}\llbracket e \rrbracket_b\textit{)} \tag{3.28}$$

$$\llbracket \textbf{Call } \overline{e_1}\ e_2.f \rrbracket_b = f\textit{(}\llbracket e_2 \rrbracket_b\textit{, } \llbracket e_1 \rrbracket_b\textit{)} \tag{3.29}$$

$$\llbracket \textbf{Compare } e\ [],\ [] \rrbracket_b = e \tag{3.30}$$

$$\llbracket \textbf{Compare } e\ op_{1..n}\ e_{1..n} \rrbracket_b = \textbf{Compare } \textit{\_op}[0]\textit{(}\llbracket e \rrbracket_b, \llbracket e_1 \rrbracket_b\textit{)}\ [op_{2..n}]\ [e_{2..n}] \tag{3.31}$$

$$\llbracket \textbf{Constant } v \rrbracket_b = \begin{cases} \text{‘}v\text{’} & \text{v is string} \\ v & \text{otherwise} \end{cases} \tag{3.32}$$

$$\llbracket \textbf{Slice } v \rrbracket_b = v \tag{3.33}$$

$$\llbracket \textbf{Tuple } \bar{e} \rrbracket_b = \textit{\_new\_tuple\_n(}\llbracket e \rrbracket_b\textit{)} \text{ where n=len}(\bar{e}) \tag{3.34}$$

$$\llbracket \textbf{Dict } \overline{e_1}\ \overline{e_2} \rrbracket_b = \textit{\_new\_dict\_n(}\llbracket zip(e_1,\ e_2) \rrbracket_b\textit{)} \text{ where n=len}(\bar{e}) \tag{3.35}$$

$$\llbracket \textbf{Set } \bar{e} \rrbracket_b = \textit{\_new\_set\_n(}\llbracket e \rrbracket_b\textit{)} \text{ where n=len}(\bar{e}) \tag{3.36}$$

$$\llbracket \textbf{List } \bar{e} \rrbracket_b = \textit{\_new\_list\_n(}\llbracket e \rrbracket_b\textit{)} \text{ where n=len}(\bar{e}) \tag{3.37}$$

$$\llbracket \textbf{Subscript } e_1\ lo \rrbracket_b = \textit{\_List\_SliceL(}\llbracket e_1 \rrbracket_b\textit{, } \llbracket lo \rrbracket_b\textit{)} \tag{3.38}$$

$$\llbracket \textbf{Subscript } e_1\ lo\ up \rrbracket_b = \textit{\_List\_SliceLU(}\llbracket e_1 \rrbracket_b\textit{, } \llbracket lo \rrbracket_b\textit{, } \llbracket up \rrbracket_b\textit{)} \tag{3.39}$$

$$\llbracket \textbf{Subscript } e_1\ lo\ step \rrbracket_b = \textit{\_List\_SliceLS(}\llbracket e_1 \rrbracket_b\textit{, } \llbracket lo \rrbracket_b\textit{, } \llbracket step \rrbracket_b\textit{)} \tag{3.40}$$

$$\llbracket \textbf{Subscript } e_1\ lo\ up\ step \rrbracket_b = \textit{\_List\_SliceLUS(}\llbracket e_1 \rrbracket_b\textit{,}\llbracket lo \rrbracket_b\textit{,}\llbracket up \rrbracket_b\textit{,}\llbracket step \rrbracket_b\textit{)} \tag{3.41}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$\llbracket \textbf{Subscript}\ e_1\ up \rrbracket_b = \textit{\_List\_SliceU}(\llbracket e_1 \rrbracket_b, \llbracket lo \rrbracket_b) \tag{3.42}$$
$$\llbracket \textbf{Subscript}\ e_1\ up\ step \rrbracket_b = \textit{\_List\_SliceUS}(\llbracket e_1 \rrbracket_b, \llbracket up \rrbracket_b, \llbracket step \rrbracket_b) \tag{3.43}$$
$$\llbracket \textbf{Subscript}\ e_1\ step \rrbracket_b = \textit{\_List\_SliceS}(\llbracket e_1 \rrbracket_b, \llbracket step \rrbracket_b) \tag{3.44}$$
$$\llbracket \textbf{Subscript}\ e_1\ e_2 \rrbracket_b = \textit{\_LSD\_Get}(\llbracket e_1 \rrbracket_b, \llbracket e_2 \rrbracket_b) \tag{3.45}$$
$$\llbracket \textbf{Attribute}\ v\ e \rrbracket_b = v(\llbracket e \rrbracket_b) \tag{3.46}$$
$$\llbracket \textbf{Name}\ v \rrbracket_b = v \tag{3.47}$$
$$\llbracket \textbf{JoinedStr}\ e_1, ..., e_n \rrbracket_b = \textit{\_Add}(\llbracket e_n \rrbracket_b, \llbracket \textbf{JoinedStr}\ [e_1, ..., e_{n-1}] \rrbracket_b) \tag{3.48}$$
$$\llbracket \textbf{FormattedValue}\ e_1\ v \rrbracket_b = \textit{\_str\_format2}(\llbracket e_1 \rrbracket_b, v) \tag{3.49}$$
$$\llbracket \textbf{FormattedValue}\ e_1\ v\ e_2 \rrbracket_b = \textit{\_str\_format3}(\llbracket e_1 \rrbracket_b, v, \llbracket e_2 \rrbracket_b) \tag{3.50}$$

Figure 3.8: Transformation from AST to SSA (via CFG)

Provenance Information

Transforming the simplified Python code into AST leads to the loss of provided comments. The translation of code from AST to SSA through CFG results in a deviation of variable names in this IR form from those in the original source code. This is due to the SSA property, which mandates variables to be declared only once. In addition, the code formatting is lost, encompassing deviations from standard Python formatting, such as, e.g., additional parenthesis, newline placements and distinctions between single ' and double " quotes. Also, during transformation from CFG to SSA, the positional information (start-column and -row, end-column and -row) associated with a node is lost.

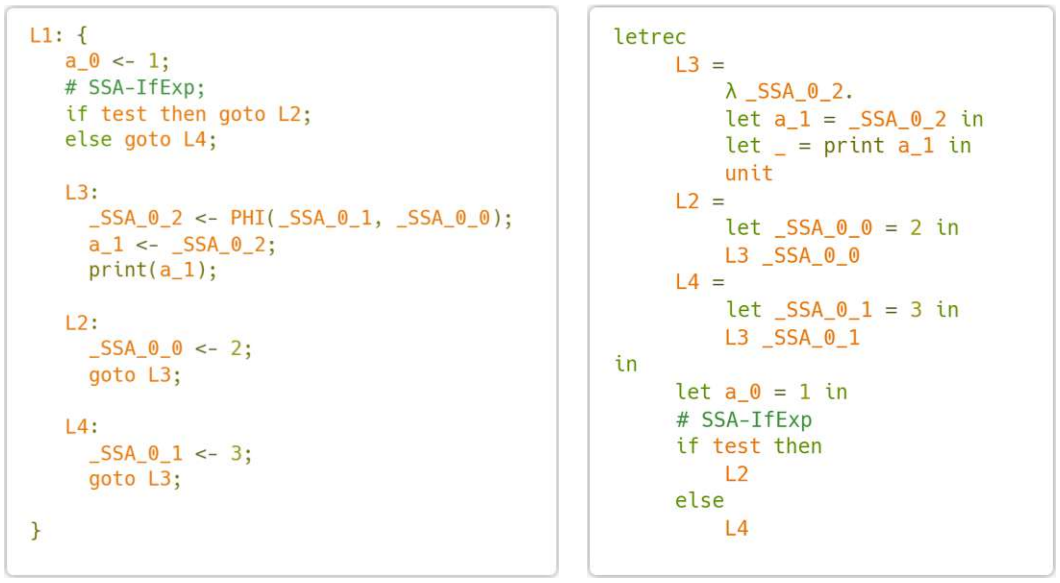
To ensure that this information is not lost, we record and track the position data from the AST for each node. Although these details are not explicitly emphasized in the syntax and transformation for readability reasons, it is nevertheless propagated into all generated SSA nodes. Retaining comments is of crucial importance as they contain valuable provenance information from the preprocessing step. To achieve this, we use an extension to the AST standard library known as ast-comments[1]. However, certain details such as spacing, parenthesis and quotes are not preserved in the definition of our library. This is due to the fact that Python's AST library makes minor formatting adjustments to the code, resulting in the loss of such information.

[1]https://pypi.org/project/ast-comments/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3.3 SSA → ANF

Using the generated SSA code as a basis, we continue the process of transforming it into ANF, as can be seen within the ongoing example from Figure 3.9a to 3.9b. The transformation steps presented in Figure 3.10, show the structure of the code on the left side according to the syntax definition of SSA from Section 3.2. We now proceed to select the first code structure from the generated SSA code and identify the first matching entry within the definitions on the left side. Thereafter, we map the variables and code to the corresponding ANF structure and continue the translation process for the further code sections.



```
L1: {
   a_0 <- 1;
   # SSA-IfExp;
   if test then goto L2;
   else goto L4;

   L3:
     _SSA_0_2 <- PHI(_SSA_0_1, _SSA_0_0);
     a_1 <- _SSA_0_2;
     print(a_1);

   L2:
     _SSA_0_0 <- 2;
     goto L3;

   L4:
     _SSA_0_1 <- 3;
     goto L3;

}
```

(a) SSA

```
letrec
     L3 =
         λ _SSA_0_2.
         let a_1 = _SSA_0_2 in
         let _ = print a_1 in
         unit
     L2 =
         let _SSA_0_0 = 2 in
         L3 _SSA_0_0
     L4 =
         let _SSA_0_1 = 3 in
         L3 _SSA_0_1
in
     let a_0 = 1 in
     # SSA-IfExp
     if test then
         L2
     else
         L4
```

(b) ANF

Figure 3.9: Transformation from SSA to ANF

The translation culminates in a fully translated ANF code, which can be seen on the right hand side of Figure 3.10, where `procs` and scoped blocks merge into `letrecs`, while the remaining code matches the corresponding ANF structures. It is worth mentioning that blocks within scoped areas are translated in reverse order, as shown in lines (3.55) and (3.56). This arrangement is necessary because earlier blocks refer into later blocks, but not vice versa; otherwise they would dominate the other blocks and would be placed in an outer scope.

The property of exclusively having atomic function parameters is demonstrated within the lines (3.64) to (3.67). Whenever a function call might occur, as in (3.57), (3.58), and (3.60), we initially bind all subexpressions to buffer variables. This process is performed recursively using the separate definition **EP**. The new variables utilize the positional information of the given provenance information from the original code as a reference, to be used after the extraction (3.66). We then can substitute subexpressions with the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

corresponding buffer variable when transforming the function call itself, as illustrated in definition (3.63). This substitution occurs only if a replacement was previously executed; otherwise, we can directly retain the function as its parameters were already trivial.

SA (SSA to ANF)

$$\llbracket \overline{p}\ \overline{b} \rrbracket = \textbf{letrec } \llbracket \overline{p} \rrbracket \textbf{ in } \llbracket \overline{b} \rrbracket \tag{3.51}$$

$$\llbracket \textbf{\textit{proc}}\ v_0\,\textbf{\textit{(}}v_1, ..., v_n\textbf{\textit{)\{}}\overline{b}\textbf{\textit{\}}},\ \overline{p} \rrbracket = v_0 = \lambda v_n\ .\ \ldots\ \lambda v_1\ .\ \llbracket \overline{b} \rrbracket;\ \llbracket \overline{p} \rrbracket \tag{3.52}$$

$$\llbracket [] \rrbracket = \tag{3.53}$$

$$\llbracket \overline{b}\ , v\textbf{:}\ \overline{e} \rrbracket_l = v^{\textbf{flc}} = \llbracket \overline{e} \rrbracket_v;\ \llbracket \overline{l} \rrbracket_v \tag{3.54}$$

$$\llbracket \overline{b_1}\ , v\textbf{:}\ \textbf{\{}b : \overline{b_s}\textbf{\}} \rrbracket_l = \textbf{letrec } \llbracket \overline{b_s} \rrbracket_v \textbf{ in } \llbracket \overline{b} \rrbracket_v;\ \llbracket \overline{b_1} \rrbracket_v \tag{3.55}$$

$$\llbracket v \leftarrow \varphi\textbf{\textit{(}}...\textbf{\textit{)}},\ \overline{e} \rrbracket_l = \lambda v\ .\ \llbracket \overline{e} \rrbracket_l \tag{3.56}$$

$$\begin{aligned}\llbracket v \leftarrow f,\ \overline{e} \rrbracket_l &= {}^{EP}\llbracket f \rrbracket_l \\ &\quad \textbf{let } v = \llbracket f \rrbracket_l \textbf{ in } \llbracket \overline{e} \rrbracket_l\end{aligned} \tag{3.57}$$

$$\begin{aligned}\llbracket \textbf{\textit{if}}\ f\ \textbf{\textit{then}}\ \overline{e_1}\ \textbf{\textit{else}}\ \overline{e_2} \rrbracket_l &= {}^{EP}\llbracket f \rrbracket_l \\ &\quad \textbf{if } \llbracket f \rrbracket_l \textbf{ then } \llbracket e_1 \rrbracket_l \textbf{ else } \llbracket e_2 \rrbracket_b\end{aligned} \tag{3.58}$$

$$\llbracket \textbf{\textit{goto}}\ v \rrbracket_l = v\ F_1(l,\ v) \tag{3.59}$$

$$\begin{aligned}\llbracket \#\ c,\ \overline{e} \rrbracket_l &= \textbf{--}\ c \\ &\quad \llbracket e \rrbracket_l\end{aligned} \tag{3.60}$$

$$\llbracket \textbf{\textit{ret}}\ f \rrbracket_l = \begin{cases} \textbf{unit} & f\ is\ None \\ {}^{EP}\llbracket f \rrbracket_l & \text{otherwise} \\ \llbracket f \rrbracket_l^{\textbf{ret}} & \end{cases} \tag{3.61}$$

$$\llbracket v \rrbracket_l = v^{\textbf{c|v}} \tag{3.62}$$

$$\llbracket v\,\textbf{\textit{(}}f_1, ..., f_n\textbf{\textit{)}} \rrbracket_l^{\textbf{pos}} = \begin{cases} \gamma_{position} & \gamma_{pos} \text{ not null} \\ v^{\textbf{f..}}\ \llbracket f_1 \rrbracket_l \ldots \llbracket f_n \rrbracket_l & \text{otherwise} \end{cases} \tag{3.63}$$

EP(Extracts the non trivial parameters)

$${}^{EP}\llbracket v\textbf{\textit{(}}\overline{f}\textbf{\textit{)}} \rrbracket_l = \llbracket \overline{f} \rrbracket_l \tag{3.64}$$

$${}^{EP}\llbracket v,\ \overline{f} \rrbracket_l = \llbracket \overline{f} \rrbracket_l \tag{3.65}$$

$$\begin{aligned}{}^{EP}\llbracket f,\ \overline{f} \rrbracket_l^{\textbf{pos}} &= \llbracket f \rrbracket_l \\ &\quad \textbf{let } \gamma_{pos}^{\textbf{b..}} = \textbf{SA}(\llbracket f \rrbracket_l) \textbf{ in } \llbracket \overline{f} \rrbracket_l\end{aligned} \tag{3.66}$$

$${}^{EP}\llbracket [] \rrbracket_l = \varnothing \tag{3.67}$$

Figure 3.10: Transformation from SSA to ANF

Additionally, we provide the ANF code with provenance information, which is indicated by blue superscript placed above certain variables or code structures, see e.g. transformation line (3.54). In this way, it is possible to translate the ANF code back into its original language. When printing the code, this provenance information will accompany each line, alongside other details such as variable types (constants, buffer variables, block identifiers, etc.).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In line (3.59), we apply F_1 as a function that takes two inputs: the label of the target block to which we aim to jump and the current block in which we are situated. As outlined in Algorithm 1, the function initially utilizes the target block label to obtain a reference to the block itself from a global map storing all blocks. Afterwards, it extracts all assigned variables within the current block. Thereby, it iterates through all φ-assignments of the target block, searching for a match in the current block with the same name. If no match is found, the process is recursively repeated in the predecessors of the current block until a matching variable assignment for all target φ-assignments is found, or until the first block from the original code is reached. If no assignment is found, it indicates that the variable was not set in this particular branch, and thus it is set to `None`. Finally, the function returns a resulting list of variable names to be utilized for the jump to another block.

Provenance Information

Representing the ANF of the pre-transformed SSA code leads to the indistinguishability of block information, since blocks and functions are translated in a similar manner. Furthermore, variable types become generalized into broader categories within this context. Consequently, the syntax of a variable closely resembles that of a function call without parameters, making them undifferentiable from each other. This similarity also extends to return- and buffer variables when compared to general parameters.

The conservation of such information is facilitated by the introduction of so-called keys. These keys, along with other provenance details, are depicted in the transformation with a blue superscript. By utilizing them, we effectively address all issues related to data loss mentioned earlier. Valid keys are listed in Table 3.1:

Key	**Description**
c	constant
v	variable
f	function call
b	buffer variable generated while unwrapping function parameters from SSA to ANF
l	block label
fv, **fc**, **flc**, etc.	combinations of the keys above
names=[$name_1$], ..., [$name_n$]	information about function calls where parameters are set with specific names
$r_1 : c_1, r_2 : c_2$	positional information (begin, end) of rows and columns, tracked within the preprocessed Python code extracted from the AST

Table 3.1: Key Descriptions

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 1 F1

```
 1: procedure EXTRACT_JUMP_VARIABLES(targetB, fromB)
 2:     toB ← global_blocks[targetB]
 3:     blockVariables ← []
 4:     resultList ← []
 5:     for $e$ in fromB.phiAssignments do
 6:         blockVariables ← blockVariables + e.var
 7:     end for
 8:     for $e$ in toB.phiAssignments do
 9:         found ← False
10:         for phiVar in $e$.args do
11:             if phiVar in blockVariables then
12:                 found ← True
13:                 resultList ← resultList + phiVar
14:             end if
15:         end for
16:         if not found then
17:             for b in fromB.predecessors do
18:                 res ← EXTRACT_JUMP_VARIABLES(toB, b.label)
19:                 if res.length > 0 then
20:                     for var in res do
21:                         if var == $e$.var then
22:                             resultList ← resultList + var
23:                             found ← True
24:                         end if
25:                     end for
26:                 end if
27:             end for
28:             if not found then
29:                 resultList ← resultList + Null
30:             end if
31:         end if
32:     end for
33:     return resultList
34: end procedure
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3.3.4 ANF → Python (with Postprocessing)

The inverse process, i.e., the backwards transformation from ANF to Python, differs from the previous transformations as it is carried out in one step. The ANF code is directly transformed into simplified Python, with consideration of provenance information propagated from prior transformations. The definitions in Figure 3.11 are applied using the same method as in the previous steps by assigning ANF-code sections on the left-hand side and translating them accordingly into the corresponding counterparts on the right-hand side, which represents the simplified version of Python before postprocessing (see Section 3.3.4). Here we also implement the renaming process of the variables, which is described in more detail in Section 3.4.

AP(ANF to Python)

$$
\begin{array}{lll}
\llbracket \textbf{letrec}\ \overline{f}\ \textbf{in}\ e \rrbracket & = \llbracket f \rrbracket\ \llbracket \mathbf{FB}(\llbracket e \rrbracket) \rrbracket & (3.68) \\
\llbracket v^{\mathrm{l..}} = e \rrbracket & = \ \mid ass[v] = \mathbf{FB}(\llbracket e \rrbracket) & (3.69) \\
\llbracket v = e \rrbracket & = \textbf{def}\ v(\mathbf{FP}(\llbracket e \rrbracket)): & (3.70) \\
 & \quad\ \mathbf{FB}(\llbracket e \rrbracket) & (3.71) \\
\llbracket \textbf{let}\ \_ = e_1\ \textbf{in}\ e_2 \rrbracket & = \llbracket e_1 \rrbracket & (3.72) \\
 & \quad \llbracket e_2 \rrbracket & (3.73) \\
\llbracket \textbf{let}\ v = e_1\ \textbf{in}\ e_2 \rrbracket & = v = \llbracket e_1 \rrbracket & (3.74) \\
 & \quad \llbracket e_2 \rrbracket & (3.75) \\
\llbracket \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rrbracket & = \textbf{if}\ \llbracket e_1 \rrbracket\textbf{:} & (3.76) \\
 & \quad\ \ \llbracket ass[e_2] \rrbracket & (3.77) \\
 & \quad \textbf{else:} & (3.78) \\
 & \quad\ \ \llbracket ass[e_3] \rrbracket & (3.79) \\
\llbracket \texttt{--}\ v\ e \rrbracket & = \#\ v & (3.80) \\
 & \quad \llbracket e \rrbracket & (3.81) \\
\llbracket v^{\mathrm{flc}}\ \overline{v} \rrbracket & = \begin{cases} \llbracket ass[v] \rrbracket \mid pb.add(v) & \text{where } v \text{ not in } pb \\ v(\llbracket \overline{v} \rrbracket) & \text{otherwise} \end{cases} & (3.82) \\
\llbracket v\ \overline{v} \rrbracket & = \begin{cases} FM[v]\ \%\ \llbracket \overline{v} \rrbracket & \text{where } v \text{ in } FM \\ FM_2[v]\ \%\ \llbracket \overline{v} \rrbracket & \text{where } v \text{ in } FM_2 \\ v(\llbracket \overline{v} \rrbracket) & \text{otherwise} \end{cases} & (3.83) \\
\llbracket v^{\mathrm{f..}} \rrbracket & = v() & (3.84) \\
\llbracket v^{\mathrm{ret..}} \rrbracket & = \textbf{return}\ v & (3.85) \\
\llbracket v \rrbracket & = \begin{cases} ass[v] & \text{v in ass} \\ v & \text{otherwise} \end{cases} & (3.86) \\
\llbracket v_{1..n} \rrbracket & = \llbracket v_1 \rrbracket, \ldots, \llbracket v_n \rrbracket & (3.87) \\
\llbracket \textbf{unit} \rrbracket & = \varnothing & (3.88) \\
 & & (3.89)
\end{array}
$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

FB (Get the function body)

$$[\![\lambda\ v\ .\ e]\!] = [\![e]\!] \tag{3.90}$$

$$[\![e]\!] = e \tag{3.91}$$

FP (Get the parameter list of a function)

$$[\![\lambda\ v\ .\ e]\!] = v,\ [\![e]\!] \tag{3.92}$$

$$[\![e]\!] = \varnothing \tag{3.93}$$

Figure 3.11: Backwards transformation from ANF to Python

Notable elements of the specifications in Figure 3.11 include the use of the | symbol to indicate side effects of globally stored variables, such as storing references to previously parsed blocks. Variables of the type `pb` (parsed blocks) and `ass` (assignments) serve as a repository for this global information. During this step, we can observe the reappearance of the provenance information that has been referred to throughout the transformation and is now highlighted in blue superscript. It emphasizes its importance for the backwards transformation process. Without this significant information, accurate parsing of the code into its original representation would not be possible.

The functions FM and FM_2 in (3.83), which stand for Function Mapping, serve to link the given input v back to a Python function. See Section 3.4.5 for further information.

Postprocessing

After applying the above mentioned transformation, the resulting output provides the simplified Python code. However, this transition necessitates an additional step to undo the preprocessing adjustments made during the initial transformation process, as defined in Section 3.3.1.

During this phase, we systematically identify the markers introduced in the preprocessing phase (in Figure 3.6, see Preprocessing) and reverse the alterations applied. By retracing our steps in this manner, we effectively restore the original structure and complexity of the code. A detailed explanation of this process is provided in Section 3.4.

Provenance Information

In the backward transformation phase of our pipeline, we use the collected provenance information from the previous steps. By utilizing the different keys, we distinguish between parsing an ANF code fragment into a function call, a variable or a buffer variable. We also use respective keys to distinguish between blocks and functions, which makes it possible to convert them into simplified Python code. This provenance information is now indicated by blue superscript on the left-hand side of the transformation. Finally, when transitioning from simplified Python back to regular Python, we use the provenance information obtained from the comments added in the preprocessing phase.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3.4 Implementation

In this section, we will provide an overview of the tools and methods utilized for the implementation, as well as a detailed explanation of the implementation process itself. The library and the web-project are publicly available on GitHub (Scalpel_SSA_ANF_Extension).

Fundamentals

The development of our library is based on Scalpel Li et al. [2022] with the chosen programming language Python, as it is compatible with its own AST library, facilitating the conversion of Python code. It describes a given program with nested objects in a tree structure. This allows for traversing the AST and thus enables the access for each variable, its value and other relevant information from the original code. Interpreting the Python code using AST guarantees that all essential components for executing the code are included in its representation, with the exception of comments.

To overcome this limitation, we use the „ast-comments“ package, which extends the AST to include custom nodes for comments. We have enlarged this library to cover all possible comment placements and ensure that no comments are omitted throughout the transformation process. Our developed extension is publicly available on PyPi[2] since version 1.1.1. Although Python is mainly interpreted, tools such as PyInstaller[3] and cx_Freeze[4] provide the ability to extract Python programs into standalone executables that contain both the interpreter and the bytecode. This allows Python programs to appear similar to compiled applications, although they still run in a Python environment. In our case, PyInstaller is used to build executable files for all systems.

3.4.1 Command-Line Interface and Web Application

To facilitate use in heterogeneous program analysis pipelines, we provide a stand-alone command-line interface for our library of transformations. The CLI tool adhered to offers a variety of customizable options. These options allow users to modify the behavior of the library according to their preferences. Possible operations are listed in Table 3.2:

We have developed a web application[5] (Figure 3.12) that enables interactive exploration of the mapping between the Python code and ANF, which allows for a better understanding of the code transformation process.

When using the ANF transformation library, a file is automatically generated containing both the simplified Python code and the ANF code with provenance information. Users

[2]https://pypi.org/project/ast-comments/
[3]https://pyinstaller.org/en/stable/
[4]https://cx-freeze.readthedocs.io/en/stable/
[5]https://github.com/Marc-Goritschnig/Scalpel_SSA_ANF_Extension/tree/main/CodeComparator

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Operation	Description
`--input_path`	Allows for the input path of the Python file to be transformed
`--out_path`	Requires the path to the output folder, where all files are generated, as an input
`--debug_mode`	Prints the debug logs and the IR of the code onto the console, for tracking the code changes between the IRs
`--parse_back`	Enables the step of parsing the code immediately back to compare it with the original code
`--only_parse_back`	If True, the input file will be interpreted as ANF code with annotations and parsed back into Python code
`--output_syntax`	Adjusts the format in which the output is printed (ASCII or Unicode)
`--no_output_files`	Does not print the code in the different IRs to the output folder
`--print_prov_info`	Prints ANF code, including provenance information
`--no_pos`	Prints ANF provenance information without positional data

Table 3.2: Options for the transformation process

can then upload this file. The project has an interactive interface that allows users to select elements in the ANF code to visualize its appearance within the original code, providing a direct link between both representation forms.

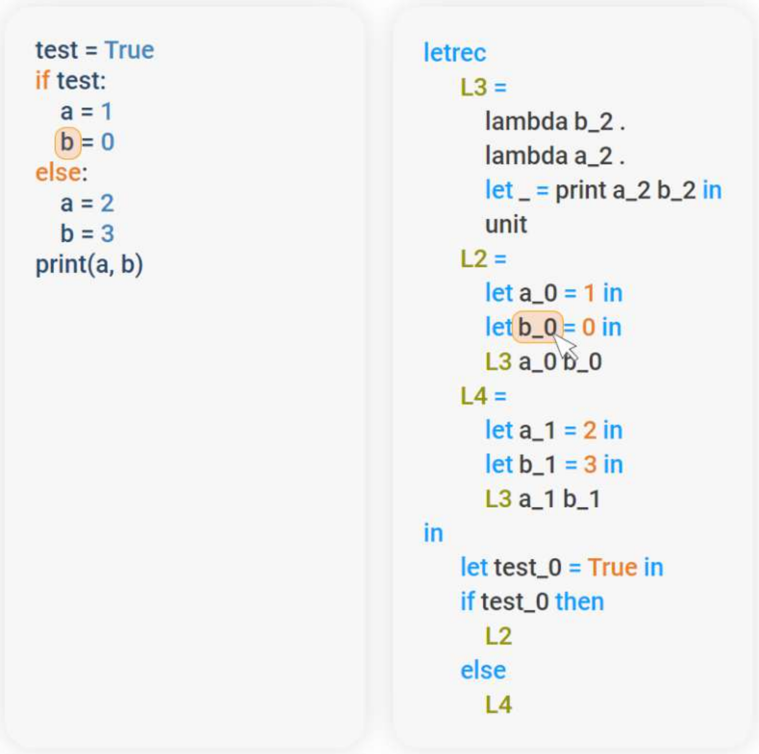
An application is shown in Figure 3.12, which illustrates this highlighting, using the previous example from Chapter 2. By selecting b_0 in the ANF section, the mapping to the corresponding variable b in the original source becomes visible. It demonstrates the properties of the SSA, which serves as an intermediate step within the transformation pipeline, showing the requirement of unique and atomic definitions.

3.4.2 Preprocessing

Referring to the transformation pipeline depicted in Figure 3.1, the preprocessing serves as the initial step, aimed at simplifying the complex structures within the imperative Python source code, such as **ListComprehension** or **IfExp**. The procedure, described in Section 3.3.1, is required for the transformation, whereby we begin by generating AST of the provided input code.

We proceed by systematically iterating over all nodes within the tree until we come across for one of the specified nodes of interest, as defined in Figure 3.6. After identifying the node, we determine the line of code in which it is located and replace it with a simplified equivalent. This iterative process is continued with the regenerated AST until no further

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



```
test = True
if test:
  a = 1
  b = 0
else:
  a = 2
  b = 3
print(a, b)
```

```
letrec
   L3 =
      lambda b_2 .
      lambda a_2 .
      let _ = print a_2 b_2 in
      unit
   L2 =
      let a_0 = 1 in
      let b_0 = 0 in
      L3 a_0 b_0
   L4 =
      let a_1 = 2 in
      let b_1 = 3 in
      L3 a_1 b_1
in
   let test_0 = True in
   if test_0 then
      L2
   else
      L4
```

Figure 3.12: Visualization of the transformation process with the web-project

modifications are necessary.

For clarity, the following example illustrates the simplification process within the code using the provided exemplary node with the designation `ast.ListComp`:

Algorithm 2 ListComp:

```
x = [1,2,3,4,5]
y = [i + y for i in x for y in x]
print(y)
```

```
x = [1,2,3,4,5]
__ssa__buffer__0 = []
for i in x:
    for y in x:
        __ssa__buffer__0.append(i + y)
y = __ssa__buffer__0
print(y)
```

As a result, the original Python code is simplified to a streamlined version and passed on to the next step in our pipeline.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3.4.3 Simplified Python to SSA

Before we can start converting our code into the SSA representation, we first need to convert into the CFG of the code. Therefore, we visit every node in the AST and depending on the type of node detected, we either add new blocks to the CFG or append statements as well as exit information to the currently active block.

For the subsequent steps, we utilize the Scalpel[6] library, which includes a basic implementation of the SSA transformation algorithm as defined by [Cytron et al., 1991]. This library provides meta-information regarding the way the variables used in each statement should be named, typically involving appending a sub-scripted number that increments with each new assignment. Additionally, it informs us which φ-statements need to be inserted into which blocks.

While this process can be executed with a single CFG, our library extends this capability by managing global mappings of the used variables and their latest indices. Consequently, SSA information is gathered not only for the main CFG, but also for its function CFGs, nested function CFGs, and so on.

Adaptions

It is essential to highlight that the Scalpel library has been extensively adapted in order to fulfill our specific requirements. Although it remains operational, for our purpose there are some minor inconsistencies regarding the SSA parser that must be addressed before it can be used. Before integrating it into our workflow, additional functionalities are required to effectively deal with various code examples.

Primarily, modifications were made to ensure compatibility along with our tracked global variable mapping and to prevent variables within different CFGs from being renamed to the identical name as before. In addition, some bugs in the library were fixed as a part of this project:

- If an assignment occurs to a variable while the same variable is present on the right side of this assignment, Scalpel currently renames the loaded variable on the right side using the same index name that was used to allocate the new variable on the left side, due to the order in which this process is implemented. Instead, the assignment process should begin by renaming the right-hand side of the assignment. Subsequently, the newly assigned variable must be relabeled.
- φ-assignments are only added if a variable is assigned in both parent blocks. Instead, parents should be searched recursively to find the corresponding assignation to the same variable.
- Scalpel incorrectly utilizes the library networkx to build the immediate dominators. The resulting dominator list includes the starting node as its own immediate domi-

[6]https://github.com/SMAT-Lab/Scalpel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

nator, which is incorrect. To address this issue, when constructing the dominator tree, a ghost node connected to the first node in the graph should be added to prevent this behavior.
- Scalpel fails to introduce any φ-assignments for recursive behavior found in loops such as `for` or `while`. However, this is an essential aspect for our transformation. To handle loop behavior in a CFG where a predecessor block might not be parsed concerning its SSA information when handling the root node of a loop, we start to initialize only the left-hand side of all φ-assignments. Subsequently, after parsing all blocks and extracting their SSA information, we iterate over each block again in a second phase to update the right-hand side of all φ-assignments by recursively examining predecessor blocks that are already initialized.

The mentioned wrong/missing implementations were fixed/implemented in the process of this work and submitted for review to the authors of the library Scalpel.

After we have collected the parsed SSA data, we use this information along with the CFG of the code in order to generate an internal representation of the SSA code which corresponds to a class structure resembling the AST tree, except that this time it represents a SSA tree. Mainly, we apply the definition from Section 3.3.2 to create the new SSA blocks and their terms. Throughout this transformation, we establish the SSA nodes of the tree according to the terms specified in the syntax definition in Figure 3.3. Each node corresponds to a procedure, a block, a scoped block, an assignment, a variable, and so on.

One aspect that has not been clarified so far is the renaming and placement of φ-assignments. Whenever we encounter a variable, regardless of whether it is used inside a `load` or `store` node within the AST, we rename it according to the information we have extracted from its SSA information. In addition, at the beginning of each block, as defined in Figure 3.8, line (3.6), we introduce the φ-assignments that we have tracked for the respective block. We also integrate the position information of the AST nodes into the internal SSA objects. In the end, we have produced a code that adheres to our SSA syntax, featuring renamed variables and φ-assignments placed at all the necessary locations.

The next step involves transforming the code into ANF.

3.4.4 SSA to ANF

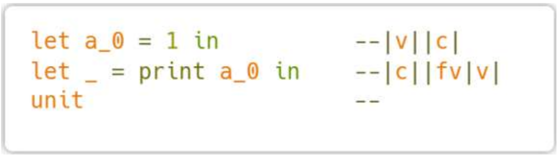
The procedure for transforming SSA to ANF is basically similar to the previous method. This time, we utilize the definition outlined in Figure 3.10 and apply it onto the existing SSA tree representation. An important step in this process is the handling of function calls: We first extract the parameters to potentially generate temporary variables, if these parameters are sub-expressions themselves. Secondly, we need to translate blocks on the same level within a scope in reverse order to ensure the correct mapping and order

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

of definitions.

Throughout the transformation, nodes from the SSA tree are now mapped to new nodes of the ANF tree and nested inside each other in order to build the tree structure. Once the transformation is completed, depending on our needs, we have the option to either print this code with or without positional information.

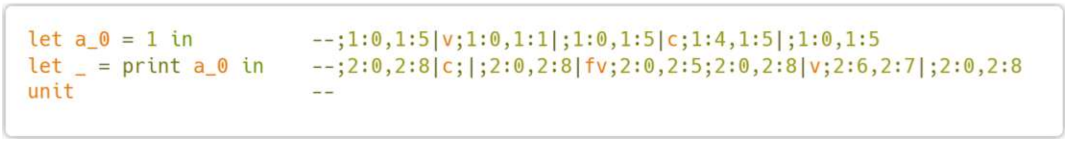
An example of such an ANF code, without position details, is displayed in Figure 3.13.



```
let a_0 = 1 in             --|v||c|
let _ = print a_0 in       --|c||fv|v|
unit                       --
```

Figure 3.13: ANF code without position information

If the position information is included, a possible scenario is obtained in Figure 3.14.



```
let a_0 = 1 in          --;1:0,1:5|v;1:0,1:1|;1:0,1:5|c;1:4,1:5|;1:0,1:5
let _ = print a_0 in    --;2:0,2:8|c;|;2:0,2:8|fv;2:0,2:5;2:0,2:8|v;2:6,2:7|;2:0,2:8
unit                    --
```

Figure 3.14: ANF code with position information

Since preprocessing is a mandatory step, on which the AST tree is built up, we only get the position information for the simplified code. This segment could be expanded, as mentioned later in Section 6. Both examples indicate that variables are denoted with 'v', constants with 'c' and function calls with 'fv', as can be seen on the right hand side of the figures and allocated according to Section 3.3.3. Each piece of information is separated by a separator character |.

Any other standard words (such as 'let', 'in', etc.) and characters (such as '=') do not provide any additional information, resulting in a blank space within the provenance information between |, except for their positional data in the second example Figure 3.14. At this stage, it is possible to modify the code using optimization algorithms and other relevant concepts. The backwards transformation represents the next and final part of the process.

3.4.5 Backward Transformation from ANF to Python

Upon completing the translation of the code in both directions, our primary objective is to maintain the original form as accurately as possible. As elaborated in the previously mentioned sections, the cumulative loss of information across all transformation steps includes code complexity, comments, code formatting, variable names, positional information, block information and variable types.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

With the exception of code formatting, which is changed to a standard format using the AST tree, all other data is considered relevant and should therefore be retained. To prevent this loss and ensure that the imperative source code remains consistent, we utilize the provenance information printed in Section 3.4.4 (the appended keys as a suffix to each ANF line). This gives us the capability to parse a given ANF code back into our internal representation of an ANF tree. If modifications have been made to the code, it must adhere to the same provenance information as applied by the library in previous steps. However, in cases where changes are applied to the ANF printed code, there might not be a reference to an original code, thus positional information might be omitted. Therefore, when modifications are made, the library is employed without printing the positional information. With the parsed ANF tree, we possess the capability to directly convert it back into Python code. This conversion process is executed individually for each ANF object, which delineates one term, such as a `let`-binding, application, variable, etc., depending on its type. We adhere to the transformation guidelines specified in Figure 3.11.

As already mentioned within the definition of the transformation (Section 3.3.4), we use functions as FM in (3.83), which serves to link the given input v back to a Python function. The transformation is executed using the given (and thus known) parameters in $\overline{v}$. Thereby, the outcome consists of a formatted string, where %s placeholders are substituted with the provided parameters, effectively restoring the original appearance of the function. The mapping of the functions is facilitated through Table 3.3, which contains the associated function names and their respective formats. Here we can see that based on the AST definitions[7], the transformation table contains an entry for each `boolop`, `operator`, `unaryop`, `cmpop` and custom created functions, generated within the preprocessing step.

In situations where the number of parameters is variable, the function FM_2 in (3.83) is employed as an alternative. Within this context, the function mapping is denoted in Table 3.4. Here, we create a string containing all parameters, each separated by a colon. This string is then substituted into the discovered format string within the given table. Notably, the left side (the key of the map) is expressed as a regex, as the number of elements required by the function is unknown. Consequently, it is added as a suffix to its name, resulting in a dynamic function name that necessitates mapping.

In case of a dictionary, which is a special case, it must be handled differently. Here, an additional function (3.94) is used, which builds the string that requires the structure of a key and value list for the dictionary.

[7]https://docs.python.org/3/library/ast.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Function Name	Format String	Function Name	Format String
_And	(%s and %s)	_Gt	(%s > %s)
_Or	(%s or %s)	_GtE	(%s >= %s)
_Add	(%s + %s)	_Is	(%s is %s)
_Sub	(%s - %s)	_IsNot	(%s is not %s)
_Mult	(%s * %s)	_In	(%s in %s)
_MatMult	unknown	_NotIn	(%s not in %s)
_Div	(%s / %s)	_LSD_Get	%s[%s]
_Mod	(%s \% %s)	_Raise	raise %s
_Pow	(%s ** %s)	_Raise_2	raise %s from %s
_LShift	(%s << %s)	_Assert	assert (%s)
_RShift	(%s >> %s)	_Assert_2	assert (%s, %s)
_BitOr	(%s | %s)	_Pass	pass
_BitXor	(%s \^ %s)	_Break	break
_BitAnd	(%s \& %s)	_Continue	continue
_FloorDiv	(%s // %s)	_List_Slice_L	%s[%s:]
_Invert	(~%s)	_List_Slice_U	%s[:%s]
_Not	(not %s)	_List_Slice_US	%s [:%s:%s]
_UAdd	(+%s)	_List_Slice_LS	%s[%s::%s]
_USub	(-%s)	_List_Slice_LU	%s[%s:%s]
_Eq	(%s == %s)	_List_Slice_LUS	%s[%s:%s:%s]
_NotEq	(%s != %s)	_List_Slice_S	%s[::%s]
_Lt	(%s < %s)	_Tuple_Get	%s[%s]
_LtE	(%s <= %s)		

Table 3.3: Function names and format strings

Regular Expression	Format String
^_new_list_([0-9])+$	[%s]
^_Delete_([0-9])+$	del %s
^_new_tuple_([0-9])+$	(%s)
^_new_set_([0-9])+$	{%s}
^_new_dict_([0-9])+$	**f_dict**

Table 3.4: Regular expressions and format strings

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

f_dict:

```
lambda params :  '{' + ','.join(
  [str(p) if i % 2 == 0 else (':  '  + str(p)) for i, p in enumerate(params)]
  ).replace(',:', ':')  + '}'                                          (3.94)
```

Through the utilization of a custom method implemented across all types of nodes within the ANF tree, we can seamlessly invoke the transformation on the topmost node, resulting in a fully parsed string of simplified Python code.

Upon examining the result, we encounter the issue that we now possess only the simplified version of the original code, due to the preprocessing done at the beginning. Consequently, we must execute a postprocessing step, which essentially involves reversing the preprocessing procedure. It entails analyzing the code line by line, as described in Section 3.3.4. The step can be accomplished because in all preprocessing cases, we add a comment marker in the line before (e.g.`#-SSA-ListComp`). This allows us to identify where transformations are needed, such as converting a `for`-loop into a list comprehension. The process is repeated for all markers found, iteratively making the code more similar to the original one.

After we have completed this step until no more markers are visible, we parse and unparse the resulting code again using the AST library to achieve a standard formatting of the resulting code. Finally, the result is written to a file and displayed on the console. This already answers **RQ1** and **RQ2**, since we now have the relevant data and have established a method for retrieving it back.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Evaluation

In this chapter, we evaluate our approach, based on the definitions and developments in the previous chapters. The implemented library undergoes a comprehensive validation, consisting of two evaluations. The first involves verifying the implemented AST nodes from Table A.2 using unit tests, while the second encompasses a total of 20 tests, by applying the library to real-world examples from GitHub[1] open source projects. The Python files have been selected based on the implemented nodes, listed in Table A.2.

The second evaluation is performed and analyzed according to the criteria, whether the transformation functionality is working and whether the source code is still the same after backwards transformation, or to what extent it is different.

In addition, threats to validity of the approach are conceptualized. This analysis provides important insights for the further development and optimization of the library in order to continue to enhance its performance and potential uses.

4.1 Unit Tests

Each tested AST node is listed in Table 4.1, using a separate test file for every node. Multiple tests are arranged for each node per file, with the total amount of tests specified in the „Total“ column. The „Result“ column indicates whether all tests have passed, otherwise the number of successfully parsed tests is displayed. With the exception of four cases, all nodes operate successfully. The reasons for these failures, which occurred for *FunctionDef* and *Subscript*, are explained below, while the *Class* and *Import* nodes are not being considered (translated as comments). Failed test cases occurred due to the following problems:

[1]https://github.com/

- *Subscript* - Arises when it is invoked on the left side of an assignment, as it has not been implemented yet. (e.g. `data[1:2]=data2`)
- *FunctionDef* - The correct order of code and function is not preserved after the backwards transformation. This discrepancy arises at the SSA transformation step, where functions are globally defined as `procs`, leading to global elements being prioritized. Additionally according to the node implementation of *Starred*, a parameter $*arg$ is replaced by *_Starred*(arg). The problem arises if this argument is defined after a named argument (name = value), as a named argument may only be followed by either another named argument or a starred argument, but not an unnamed argument within a function call. Also within function definitions `kwarg`(**args) and `vararg`(*args) are not implemented and those test also fail. Hence, there is a discrepancy in the order during the backwards transformation. Also, default parameters are not processed within functions and therefore get lost throughout the transformation process, leading to a wrong behaviour of the code when being executed.

All of these limitations are regarded as constraints of the library, suggesting areas for potential future enhancement.

AST NODE	TOTAL	SUCCESS	FAILURE	
Starred	6	6	0	✓
FunctionDef	12	7	5	✗
Function_NamedParams	3	3	0	✓
Return	3	3	0	✓
Delete	4	4	0	✓
AugAssign	12	12	0	✓
AnnAssign	2	2	0	✓
For	5	5	0	✓
While	3	3	0	✓
If	4	4	0	✓
Raise	2	2	0	✓
Assert	2	2	0	✓
Pass	2	2	0	✓
Break	3	3	0	✓
Continue	3	3	0	✓
BinOp	12	12	0	✓
UnaryOp	2	2	0	✓

4.2 Real-World Examples

The second evaluation is comprised of a collection of Python files obtained from public repositories on GitHub. The selection process involved identifying files with the extension ".py" that do not contain any of the unsupported AST nodes listed in Appendix A. This

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

AST NODE	TOTAL	SUCCESS	FAILURE	
Lambda	2	2	0	✓
IfExp	3	3	0	✓
Dict	5	5	0	✓
Set	4	4	0	✓
ListComp	2	2	0	✓
Compare	10	10	0	✓
Constant	3	3	0	✓
Attribute	4	4	0	✓
Subscript	4	3	1	✗
Name	5	5	0	✓
List	4	4	0	✓
Tuple	5	5	0	✓
Slice	8	8	0	✓
JoinedStr	3	3	0	✓
FormattedValue	3	3	0	✓
NamedExpr	2	2	0	✓
SetComp	2	6	0	✓
DictComp	2	2	0	✓
Comment	9	9	0	✓
Class	2	2	0	⊖
Import	2	2	0	⊖
BoopOp	2	2	0	✓

Table 4.1: Results of AST Nodes (unity tests)

process utilized GitHub's search function, whereby the top files that met these criteria were added to the set. The distribution of AST nodes used in these files is displayed in Figure 4.1, showing 72 different AST nodes in total. This number exceeds the amount of supported nodes in Table 4.1, as certain nodes, such as `BinOp`, encompass subtypes including `Add`, `Sub`, `Mult`, `MatMult` and more as described in the Python definition of AST[2].

The results of each test run are summarized in Table 4.2, highlighting whether the parsing process was successful and whether the transformation was exact. Any discrepancies between the original source code and the code generated by the backwards transformation are analyzed in detail.

This examination covers the minor deviations, so-called syntactic sugars, which do not change the behavior of the code. Various syntactic sugar elements are taken into account, such as new lines or additional spaces, as these are accepted and mainly arise due to the

[2]https://docs.python.org/3/library/ast.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

autoformatting feature of Python AST. Reference is made to the corresponding legend to facilitate better understanding of the found syntactic sugar. In addition, the link of each source code file is listed in Table A.1 together with the corresponding test ID assignment.

Legend:

- **Comment indent** - Comment indentation: adjustment of comment alignment
- **Exp. notation** - Exponential function notation E instead of e after transformation
- **Function pos.** - Change of function position
- **Inline Comments** - Inline comments shifted to next line, if initially on the same line
- **Multiline text** - Multiline text format changed from using ””” to ‘
- **New line** - New line structure altered
- **Number repr.** - Number representation adjustment: integer instead of hexadecimal
- **Parenthesis** - Adjustment of parenthesis: additional or reduced
- **Quoting** - Use of single quote instead of double quotes
- **Spaces** - Adjustment of spaces: additional or reduced
- **Typehints** - Removal of type hints
- **Trailing zeros** - Removal of trailing zeros

Additional problems, apart from syntactic sugar, can cause changes in the code’s behavior, as arising in test cases 4, 9, and 16:

- Test case 4: The star in front of the function’s args variable (variable length list of variable) is lost, see example: (`def a(*args):...`).
- Test cases 9, 16: The default values in the function definitions are lost, see example: (`def a(x=1):...`).

As indicated in the evaluation table, certain test cases failed during parsing:

- Test cases 1, 17, 18: The inability to parse a Starred AST node after named parameters leads to failure, see example: (`turn(x=1, *pos)`).
- Test case 5: A subscript AST node is defined on the left hand side of an assignment, see example: (`a[1:3]=b`).

The most frequent change caused through backwards transformation is the position of the function, as can be seen in Figure 4.2. This issue has already been discussed within the node *FunctionDef* in Section 4.1, but requires further investigation to determine if the behavior may change in other source codes.

These differences highlight what the library is incapable of preserving in terms of syntactic sugar and will be further discussed in the Section 4.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

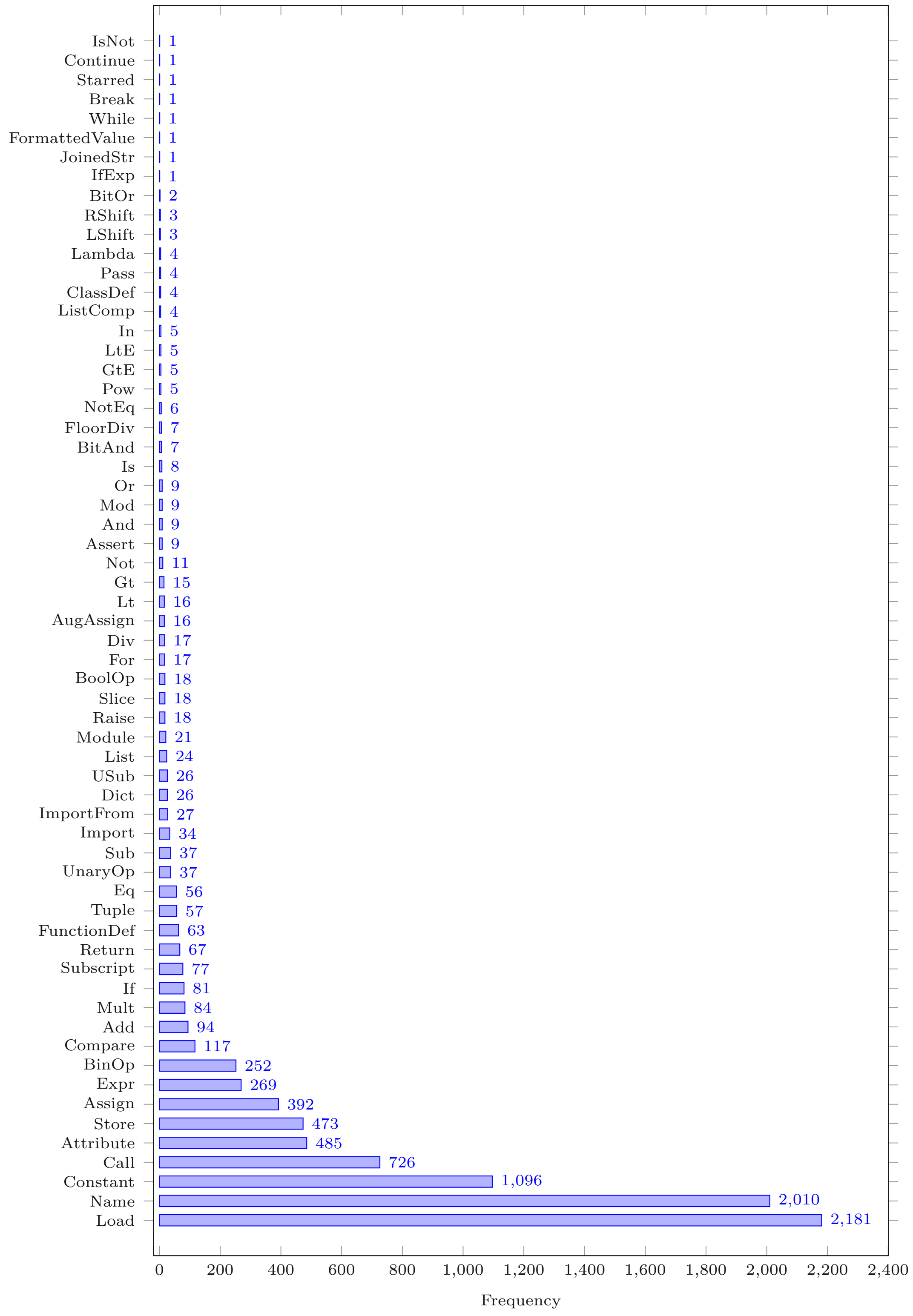


Figure 4.1: Frequency of AST nodes in Test-set 2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

	Results		Differences											
Project Name	Parsed	Exact	Comment indent	Exp. notation	Function pos.	Inline Comments	Multiline text	New line	Number repr.	Parenthesis	Quoting	Spaces	Typehints	Trailing zeros
1 - python-nvd3	✗	✗	-	-	-	-	-	-	-	-	-	-	-	-
2 - vector-datasource	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓
3 - AstroBuild	✓	✓	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗
4 - django-autofixture	✓	✗	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓
5 - kaggle_pbr	✗	✗	-	-	-	-	-	-	-	-	-	-	-	-
6 - unirest-python	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓
7 - 30-Days-Of-Python	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓
8 - FPN_Tensorflow	✓	✓	✗	✓	✗	✓	✗	✗	✓	✗	✗	✓	✓	✓
9 - kaggletils	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✗
10 - Python	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓	✗	✓
11 - HyperFace-with-...	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✓	✓
12 - john	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
13 - 30-Days-Of-Python	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓
14 - Tautulli	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗	✗	✓	✗	✓
15 - asuswrt-merlin.ng	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓	✓
16 - Download-and-...	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓
17 - samba	✗	✗	-	-	-	-	-	-	-	-	-	-	-	-
18 - asuswrt-merlin.ng	✗	✗	-	-	-	-	-	-	-	-	-	-	-	-
19 - asuswrt-merlin.ng	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓	✓
20 - asuswrt-merlin.ng	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓	✓

Table 4.2: Test Results of Test-set 2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

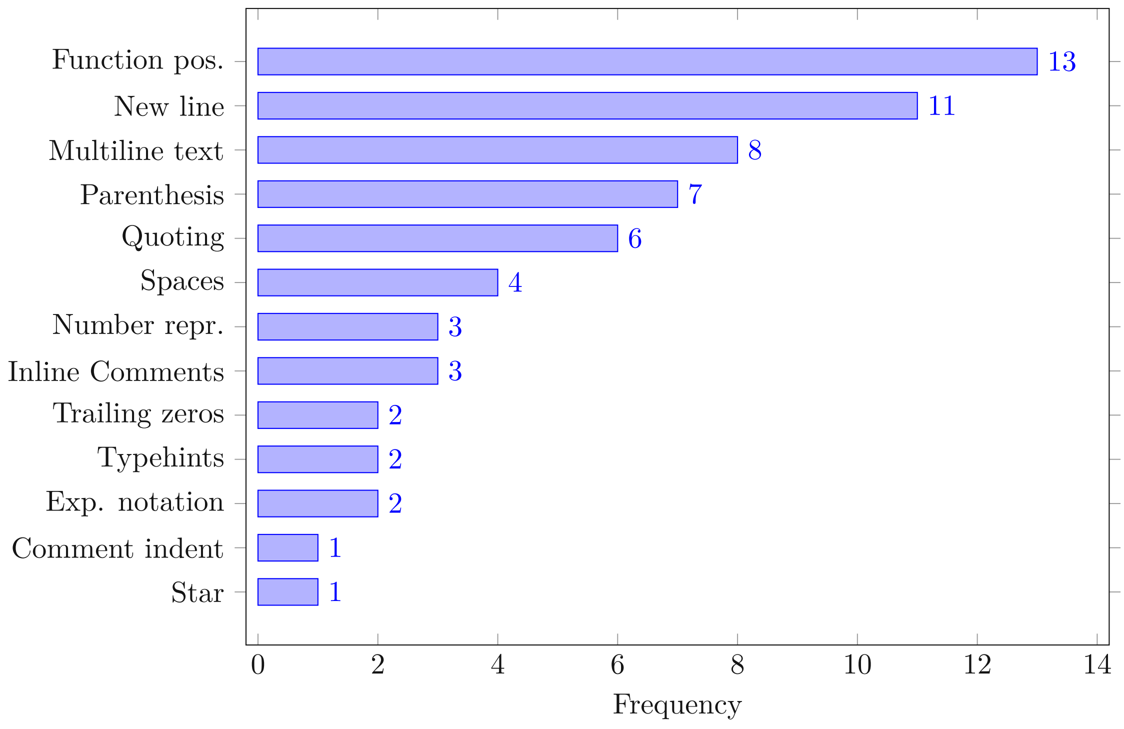


Figure 4.2: Frequency of types of differences from Table 4.2

4.3 Threats to Validity

4.3.1 External Validity

Scope of Test Set: The external validity of our study is affected by the scope of our test set, which is limited to supported nodes (see Table A.2). However, it is important to consider potential implementations of these nodes within unsupported constructs, such as classes, which are not covered in our test set.

Programming Language Extension: Our implementation is specifically designed for Python. However, the use of Python AST libraries facilitate the possible extension to other programming languages. The underlying principles of the transformation remain consistent across languages due to the use of ASTs. This adaptability strengthens the internal validity of our study, as the consistent principles remain general beyond Python.

Subexpression Extraction: The extraction of sub-expressions, which is essential for ANF, could already be performed during the initial transition from imperative languages to SSA. However, in order to increase external validity, we have integrated this procedure into a later point of our pipeline, namely the transformation from SSA to ANF. This extension is designed to improve the adaptability of our approach for different programming languages.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4.3.2 Internal Validity

Impact of Function Position: Possible impacts of the transformation process regarding the positioning of functions must be considered, as our approach changes the position of the functions. Such changes could result in variations in behavior and outcomes. Specifically, if a function shares its name with a variable and their definitions are incorrectly ordered, there's a risk of the entire program breaking. Although this is not certain in all cases, this risk exists in particular scenarios depending on the places the variable is used.

Unforeseen errors: Acknowledging the limitations of the developed library, it is possible that unforeseen errors may occur due to oversight or inherent difficulties in detection.

Known Limitations and Missing Functionality: Taking into account the presence of unimplemented nodes within Python files, the frequency of these occurrences prompts inquiries into the broader applicability and effectiveness of our transformation tool. Additionally, recognizing limitations and missing functionalities within our implementation invoke constraints, that play an important role in shaping the interpretation and generalization of our findings.

- Function placement after backwards transformation
- When using (*args) as a function argument, it fails if a named argument is infront of it
- When using (*args) as function parameters, the * is lost throughout the transformation
- Default parameters fail to function when they are non-trivial, such as attributes, as in the example `def a(x=a.b)`
- Double backslashes in strings, such as a = ”$C:\backslash\backslash test.py$”, and a = ”$C:\backslash\backslash$”, cause issues

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Related Work

Intermediate Representations: IRs are used in a wide range of applications, such as predicting future trajectories in urban driving scenarios [Srikanth et al., 2019], the semantic segmentation in video understanding [Kangaspunta et al., 2021], neural semantic parsers that aim to close the semantic gap between natural and formal languages [Nie et al., 2022], and many more. But they are also used in terms of data transformation for abstract visualizations [Hamaza et al., 2020] or the conversion into a grammatically consistent form of IR in order to establish a basis in the variety of program languages for static analyses [Spanier and Mahoney, 2023].

Referring to our work, the public available Scalpel framework Li et al. [2022] provides fundamental static analysis functions (e.g., call graph constructions, control-flow graph constructions, alias analysis, etc.), which builds the basis for our thesis. Although, the framework lacks in the available amount of functionalities, which we enlarged for our purposes, such as the introduction of φ-assignments, the utilization of the networkx library and the renaming of newly assigned variables. The approach in [Click and Paleczny, 1995] demonstrates the use of CFG and SSA for program execution and optimization. Here, control dependencies are defined as edges, and control information as nodes. Similar to us, the CFG is used to simplify the data structure, although the implementation is done with C++. In their work [Cytron et al., 1991], the authors propose an efficient method for constructing data structures using CFG and SSA form, aiming for enhanced efficiency and program optimization. However, a drawback is the potential increase in program size. The transformation from SSA to ANF, as proposed in [Chakravarty et al., 2004], aims to enhance reasoning about SSA-based optimization algorithms. But it leads to the constraint of restricting SSA to include only atomic function parameters, thereby limiting flexibility in implementation and ease of enhancement to other languages. The approach also introduces the possibility of increased program size.

Transformations: The transformation from continuation passing style to SSA, detailed in [Kelsey, 1995], underscores its helpfulness in compiling functional programs. This approach allows for direct transformation, eliminating the need for flow analysis and enabling the analysis of recursively expressed loops. Complementary, [Sreedhar et al., 1999] delves into SSA programs featuring φ instructions, which often require nontrivial translations for native instructions such as copy propagation. The framework developed in this study facilitates the translation out of SSA through copy placement, streamlining the process while removing redundant copy operands. In [Appel, 1998b], the focus shifts to representation forms in SSA and lambda calculus. The direct transformation presented here demonstrates its efficacy in optimizing imperative and functional language compilers, highlighting that both representations yield identical results despite differences in notation. The same process of generating the SSA is used within the Scalpel library and is thus important for our work. Expanding on these transformations, [Jaramillo et al.] investigates the effects of code transformation. A technique for automatically identifying statement instance correspondences between untransformed and transformed code is introduced, enabling seamless mapping when code-improving transformations are applied. This approach supports loop optimizations, symbolic debugging, and adjustments in statement positioning and ordering. Transitioning to functional programming, [Buszka and Biernacki, 2021] discusses the transformation into ANF, emphasizing automated functional correspondence. This entails describing the evaluator in a functional meta-language to facilitate program analysis before selectively translating to continuation-passing style and defunctionalization. This process is significant for our methods, despite our source language is imperative.

In [Dig et al., 2009], a distinct category of code transformation is discussed, focusing on object-oriented languages and code refactoring to modify code and improve software quality, while also enabling backwards transformations. These tools encompass various applications, plugins, and libraries tailored for refactoring across multiple programming languages. For instance, ReLooper automates the execution of running loops in parallel, although necessitating safe loop iterations. Similarly, the syntax-preserving algorithm for program slicing, detailed in [Marinov, 2020], offers comparable functionality by refactoring code while maintaining high design quality, particularly in the context of SSA. More broadly, refactoring tools integrated into IDEs such as Eclipse[1] or IntelliJ IDEA[2] are available. These tools enable actions such as method inlining or function and variable renaming. However, they primarily operate at higher code levels like the CFG and only apply predefined changes during the backwards transformation, limiting user intervention.

Provenance Information: To the best of my knowledge, there hasn't been any prior work in the field of program analysis that explicitly formulates translations with provenance.

[1]https://www.eclipse.org/
[2]https://www.jetbrains.com/idea/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Conclusion and Outlook

The entire work builds upon the research questions initially posed, which have been addressed throughout this study. All implementations, definitions, and developments are grounded in them.

To briefly recap these questions: They inquire into how provenance information can be retained during the transformation into ANF and whether a bijective relation is feasible. These questions have been answered through an examination of the aspects of transforming high-level imperative languages into ANF code and vice versa.

The feasibility of a bijective relation can clearly be affirmed, as the novel method developed maintains the capability to preserve provenance information, enabling the backwards transformation.

The defined pipeline, encompassing multiple IRs, including the AST, the CFG and the SSA to transform to the targeting ANF, alongside with the novel method, retains the provenance information.

All necessary syntax definitions, transformation rules and guidelines have been outlined within this work. Each development has been encapsulated and merged within the Python library we have constructed. This library serves for code conversion and can be utilized for various analyses and optimizations.

Additionally, a web application has been implemented as part of this thesis, which, with the assistance of accompanying positional information from the library, visually illustrates which code fragments on the ANF side correspond to which parts of the simplified Python code.

Despite the numerous advantages and innovations that have been discovered, there are still limitations and constraints that could be of interest for future projects.

- All limitations, that are referred to in Section 4.3, could be rectified.
- In order to make the library applicable to a wider range of applications, it might be of interest to implement additional AST nodes that were not considered in this thesis. Once all nodes are supported, the library is ready to be tested in bulk.
- The preprocessing step was implemented in order to be able to use a more general code in the transformation phase to SSA and to facilitate the extension of this library to other languages. However, if more emphasis is placed on a complete visualization, this step could be shifted to the SSA phase so that the position information comes from the original code and not from the simplified Python code.

In summary, the developed Python library effectively addresses the specified research questions and can offer favorable solutions through its implementation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX A

Appendix

TEST ID	LINK
1	https://github.com/areski/python-nvd3/blob/develop/examples/lineChart.py
2	https://github.com/tilezen/vector-datasource/blob/master/scripts/all_the_kinds.py
3	https://github.com/lhartikk/AstroBuild/blob/master/astro_build.py
4	https://github.com/gregmuellegger/django-autofixture/blob/master/runtests.py
5	https://github.com/emanuele/kaggle_pbr/blob/master/blend.py
6	https://github.com/Kong/unirest-python/blob/master/unirest/utils.py
7	https://github.com/Asabeneh/30-Days-Of-Python/blob/master/03_Day_Operators/day-3.py
8	https://github.com/yangxue0827/FPN_Tensorflow/blob/master/help_utils/help_utils.py
9	https://github.com/Far0n/kaggletils/blob/master/kaggletils/math.py
10	https://github.com/TheAlgorithms/Python/blob/c6ca1942e14a6e88c7ea1b96ef3a6d17ca843f52/maths/abs.py#L5
11	https://github.com/sourabhvora/HyperFace-with-SqueezeNet/blob/master/hyperface.py
12	https://github.com/openwall/john/blob/f55f42067431c0e8f67e600768cd8a3ad8439818/run/dns/tsigkeyring.py#L25
13	https://github.com/Asabeneh/30-Days-Of-Python/blob/master/04_Day_Strings/day_4.py
14	https://github.com/Tautulli/Tautulli/blob/d019efcf911b4806618761c2da48bab7d04031ec/lib/dns/grange.py#L24
15	https://github.com/RMerl/asuswrt-merlin.ng/blob/bc3c8c32858492818c2be50e2ea95522bd342f5e/release/src/router/samba-3.6.x_opwrt/source/lib/dnspython/dns/opcode.py#L45
16	https://github.com/hemathulsidhos/Download-and-Extract-Structural-Metadata-from-Islandora/blob/main/download_rels_ext_2.0.py
17	https://github.com/amitay/samba/blob/68ef3c48fc6df2396381af622140fbc2023bd81c/lib/dnspython/dns/rdtypes/IN/IPSECKEY.py#L76
18	https://github.com/RMerl/asuswrt-merlin.ng/blob/bc3c8c32858492818c2be50e2ea95522bd342f5e/release/src/router/samba-3.6.x_opwrt/source/lib/dnspython/dns/rdtypes/ANY/SSHFP.py#L49
19	https://github.com/RMerl/asuswrt-merlin.ng/blob/bc3c8c32858492818c2be50e2ea95522bd342f5e/release/src/router/samba-3.6.x_opwrt/source/lib/dnspython/dns/rcode.py#L61
20	https://github.com/RMerl/asuswrt-merlin.ng/blob/bc3c8c32858492818c2be50e2ea95522bd342f5e/release/src/router/samba-3.6.x_opwrt/source/lib/dnspython/dns/flags.py#L82

Table A.1: Link to the source code files of test-set 2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

AST Node	Example	Supported
Statements		
FunctionDef	`def a():`	✓
Return	`return 1`	✓
Delete	`del a[0]`	✓
Assign	`a = 1`	✓
AugAssign	`a += 1`	✓
AnnAssign	`a:int`	✓
For	`for i in []:`	✓
While	`while (true):`	✓
If	`if a: ...`	✓
Raise	`raise ”`	✓
Assert	`assert a == 2, ”`	✓
Expr	**Expression**	✓
Pass	`pass`	✓
Break	`break`	✓
Continue	`continue`	✓
Expressions		
BoolOp	`a or b`	✓
BinOp	`a + b`	✓
UnaryOp	`not a`	✓
Lambda	`lambda a: a + a`	✓
IfExp	`a if x else b`	✓
Dict	`’a’: 1`	✓
Set	`(1, 2)`	✓
ListComp	`[i for i in x]`	✓
SetComp	`(i for i in x)`	✓
DictComp	`{i:i for i in x}`	✓
Compare	`a < b`	✓
Call	`a()`	✓
FormattedValue	`f’text{a}’`	✓
JoinedStr	`f’text{a}{b}’`	✓
Constant	`10`	✓
Attribute	`a.b()`	✓
Subscript	`a[1,2]`	✓
Name	`a`	✓
List	`[1,2,3]`	✓
Tuple	`(1,2,3)`	✓
Slice	`a[1:2]`	✓
NamedExpr	`(b := 10)`	✓

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

AST Node	Example	Supported
Asynchronous nodes		
AsyncFunctionDef	`async def a():`	⊗
AsyncFor	**For**	⊗
Await	`await a()`	⊗
Yield	`yield a`	⊗
YieldFrom	`yield from a`	⊗
With	`with a:...`	⊗
AsyncWith	**With**	⊗
Exception handling		
Try	`try:... except e:...`	⊗
TryStar	`try:... except* e:...`	⊗
Others		
Starred	`*a`	⊖
Global	`global a,b`	⊗
Nonlocal	`nonlocal a,b`	⊗
Match	`match x:case 1:...`	⊗
GeneratorExp	`i for i in x`	⊗
TypeAlias	`type Alias=int`	⊗
Kept as Comments		
ClassDef	`class Name:...`	⊖
Import	`import re`	⊖
ImportFrom	`from a import b`	⊖

Table A.2: Supported AST nodes with representative examples

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88*, POPL '88. ACM Press, 1988. doi: 10.1145/73560.73561. URL http://dx.doi.org/10.1145/73560.73561.

A. W. Appel. *Modern Compiler Implementation: In ML*. Cambridge University Press, USA, 1st edition, 1998a. ISBN 0521582741.

A. W. Appel. Ssa is functional programming. *Acm Sigplan Notices*, 33(4):17–20, 1998b.

M. Blume and A. W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 112–124, 1997.

W. J. Bowman. A low-level look at a-normal form. 2024.

M. Buszka and D. Biernacki. Automating the functional correspondence between higher-order evaluators and abstract machines. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 38–59. Springer, 2021.

M. M. Chakravarty, G. Keller, and P. Zadarnowski. A functional perspective on ssa optimisation algorithms. *Electronic Notes in Theoretical Computer Science*, 82(2):347–361, Apr. 2004. ISSN 1571-0661. doi: 10.1016/s1571-0661(05)82596-4. URL http://dx.doi.org/10.1016/S1571-0661(05)82596-4.

J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '91*, POPL '91. ACM Press, 1991. doi: 10.1145/99583.99594. URL http://dx.doi.org/10.1145/99583.99594.

C. Click and M. Paleczny. A simple graph-based intermediate representation. *ACM SIGPLAN Notices*, 30(3):35–49, Mar. 1995. ISSN 1558-1160. doi: 10.1145/202530.202534. URL http://dx.doi.org/10.1145/202530.202534.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991. ISSN 1558-4593. doi: 10.1145/115372.115320. URL `http://dx.doi.org/10.1145/115372.115320`.

D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: refactoring for loop parallelism in java. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA09. ACM, Oct. 2009. doi: 10.1145/1639950.1640018. URL `http://dx.doi.org/10.1145/1639950.1640018`.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. volume 28, page 237–247. Association for Computing Machinery (ACM), June 1993. doi: 10.1145/173262.155113. URL `http://dx.doi.org/10.1145/173262.155113`.

S. Hack, D. Grund, and G. Goos. Register allocation for programs in ssa-form. In *Compiler Construction: 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006. Proceedings 15*, pages 247–262. Springer, 2006.

S. Hamaza, I. Georgilas, G. Heredia, A. Ollero, and T. Richardson. Design, modeling, and control of an aerial manipulator for placement and retrieval of sensors in the environment. *Journal of Field Robotics*, 37(7):1224–1245, June 2020. ISSN 1556-4967. doi: 10.1002/rob.21963. URL `http://dx.doi.org/10.1002/rob.21963`.

J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical structures in computer science*, 7(5):507–541, 1997.

C. Jaramillo, R. Gupta, and M. Soffa. Capturing the effects of code improving transformations. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, PACT-98. IEEE Comput. Soc. doi: 10.1109/pact.1998.727181. URL `http://dx.doi.org/10.1109/PACT.1998.727181`.

J. Kangaspunta, A. Piergiovanni, R. Jonschkowski, M. Ryoo, and A. Angelova. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1602–1612, 2021.

R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, POPL95. ACM, Mar. 1995. doi: 10.1145/202529.202532. URL `http://dx.doi.org/10.1145/202529.202532`.

J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. volume 29, page 147–158. Association for Computing Machinery (ACM), June 1994. doi: 10.1145/773473.178256. URL `http://dx.doi.org/10.1145/773473.178256`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

L. Li, J. Wang, and H. Quan. Scalpel: The python static analysis framework. *arXiv preprint arXiv:2202.11840*, 2022.

E. S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, Jan. 1969. ISSN 1557-7317. doi: 10.1145/362835.362838. URL http://dx.doi.org/10.1145/362835.362838.

C. K. Mai, B. V. Kiranmayee, M. N. Favorskaya, S. C. Satapathy, and K. S. Raju. *Proceedings of International Conference on Advances in Computer Engineering and Communication Systems - ICACECS 2020*. Springer Nature, Singapore, 2021. ISBN 978-9-811-59293-5.

A. Marinov. *An efficient syntax-preserving slide-based algorithm for program slicing*. PhD thesis, Academic College OF Tel Aviv-Yaffo, 2020.

S. Mirliaz and D. Pichardie. A flow-insensitive-complete program representation. In B. Finkbeiner and T. Wies, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 197–218, Cham, 2022. Springer International Publishing. ISBN 978-3-030-94583-1.

L. Nie, S. Cao, J. Shi, J. Sun, Q. Tian, L. Hou, J. Li, and J. Zhai. Graphq ir: Unifying the semantic parsing of graph query languages with one intermediate representation. *arXiv preprint arXiv:2205.12078*, 2022.

A. Spanier and W. Mahoney. Static vulnerability analysis using intermediate representations: A literature review. In *European Conference on Cyber Warfare and Security*, volume 22, pages 458–465, 2023.

V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Static Analysis: 6th International Symposium, SAS'99 Venice, Italy, September 22–24, 1999 Proceedings 6*, pages 194–210. Springer, 1999.

S. Srikanth, J. A. Ansari, R. K. Ram, S. Sharma, J. K. Murthy, and K. M. Krishna. Infer: Intermediate representations for future prediction. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 942–949. IEEE, 2019.

R. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1): 62–89, 1974.

A. Tavares, B. Boissinot, F. Pereira, and F. Rastello. Parameterized construction of program representations for sparse dataflow analyses. In A. Cohen, editor, *Compiler Construction*, pages 18–39, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54807-9.

Tinman. Lexical and syntax analysis. https://tinman.cs.gsu.edu/~raj/4330/slides/c04.pdf, 2015. [Accessed 06-03-2024].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.