Department of Mathematics & Computer Science
University of Southern Denmark | IMADA
Sunday 16th April, 2023

Compiler for Panda
BADM500: Bachelor Project

Author
KIAN BANKE LARSEN
kilar20@student.sdu.dk

Supervisor
KIM SKAK LARSEN
Professor

Abstract

English This is my very good abstract

Danish Et fantastisk abstract

Contents

1 Introduction

This report examines how to make a simple compiler in Python. The compiler is simple in the sense that some decisions have been made to ease the process, despite the choice, although the decisions are not necessarily optimal. The aim is to learn different compiler techniques and get a hands-on feel for the different compiler phases by actually implementing a working compiler, targeting X86 assembler, from scratch, using a Flex/Bison equivalent package such as `PLY` for scanning and parsing.

The language to be compiled is a subset of the imperative language C. This has been chosen because of its simpler syntax and easy to read curly bracket enclosed static scopes. In this project, we are interested in making a language having integers, Booleans and preferably some kind of floats. The language must have control flow constructs in form of `if-else` statements and functions, and iterative constructs such as `for`- and `while`-loops.

A modern compiler is, as is well known, divided into phases. These phases relate to lexical and syntactic analysis, resulting in an abstract syntax tree. Subsequent phases analyze and adorn the abstract syntax tree, building a symbol table and finally generating assembler code.

The main focus in regard to advanced techniques will be local register allocation, using techniques described in Copper and Torczon 2022. Handling this efficiently requires data flow analysis via control flow-graph, construction of interference graph, graph coloring and translation back to instructions using a combination of the registers and the stack, when the available registers do not suffice.

Initially, a stack machine will be prepared, which will form the basis for developing a compiler that uses CPU registers. We will take advantage of the split phases property when replacing the stack code generation phase in benefit for one that uses register allocation. This allows us to only worry about ensuring that subsequent phases cope with the changes made in the former phases.

When adding extra complexity such as register allocation, it is important to document the benefit of this choice. Performance of the stack machine and the register machine will therefore be constructively compared.

2 Project Basics

This section is reserved for articulating some choices made at the very beginning of the project that defined the framework for how to develop and use the compiler.

2.1 Project Structure

The Compiler module uses different python packages – things that belong together must be together. It has been desired to make a clear division between the different phases, and this has been achieved by creating the Python package `phase`. Likewise, `dataclass` is a package that contains internal data structures, in other words, classes used to hold data. `Printer` is a package that is used primarily for debugging, but sometimes it is just nice to consider data structures graphically. `Testing` is placed on the same level as `src` because it has nothing to do with the compiler's implementation, it is just a QA tool that makes it easier to verify correctness. `compiler.py` takes care of summarizing all functionality, but the module requires arguments from the command line, and those arguments (as well as testing) are handled in `main.py`, which is why `main.py` can be considered the project's main file.

```
Compiler/
└─ src/
   ├─ dataclass/
   │  ├─ AST.py
   │  ├─ iloc.py
   │  └─ symbol.py
   └─ printer/
      ├─ ast_printer.py
      ├─ generic_printer.py
      └─ symbol_printer.py
```

```
Compiler/
└─ src/
   ├─ phase/
   │  ├─ code_generation_stack.py
   │  ├─ emit.py
   │  ├─ lexer.py
   │  ├─ parser.py
   │  ├─ symbol_collection.py
   │  └─ syntactic_desugaring.py
   └─ enums/
```

```
Compiler/
├─ src/
│  ├─ utils/
│  └─ compiler.py
├─ testing/
│  ├─ test-cases/
│  └─ test.py
├─ main.py
├─ README.md
└─ ...
```

Figure 2.1: Project file tree.

2.2 Python Lex-Yacc

PLY is used to automatically generate a lexer and LALR(1) parser. The package is well documented at the following source: Beazley, David M. 2018. Usage of the package will be described when reviewing the compiler phases in isolation.

2.3 Design Principles & Patterns

When starting a new project, it is important to make some basic thoughts about the architecture. Sensible choices at the beginning can increase code readability and make maintainability easier. It is particularly important to consider design principles and design patterns, as this will have a big effect on, i.e., how data structures are traversed and code testability. In this context, design principles refer to SOLID, and design patterns refer to Gang of Four's 23 design patterns.

We will start by considering design principles. SOLID is a mnemonic acronym for at set of design principles concerning software development in object-oriented languages. The principles are concerning: **S**ingle Responsibility, **O**pen Closed, **L**iskov's Substitution, **I**nterface Segregation and **D**ependency Inversion. The principles are in many ways obvious when rehearsed, but not necessarily followed as it requires active consideration. Single responsibility is particularly expressed in the project by the sharp division of phases and their interfaces between them. Open/closed is not particularly used in the project, as inheritance cases sparsely appear, but it is crucial is in the printer package. It will never be necessary to edit the generic printer because functionality to print a specific data structure is first implemented upon extension. Liskov's substitution principle is used in the AST data class, since any subtree is a valid tree and all nodes are an AST node. The code will work regardless of the structure of the tree, however, the nodes themselves limit what associations can be created. Dependency inversion principle simply means that you must program against an interface and not an implementation. There is actually an incident where the project does not live up to this principle, and that is when using the hidden method `_value2member_map_` on `Enum` in the parsing phase.

kk

2.4 Compiler Usage

The main file handles the instantiation and therefore also the running of the `PandaCompiler` class. Python `argparse` is used to take command line arguments. Python `argparse` adds a lot of user-friendliness to the compiler, because one get the opportunity to query the use of the compiler in the terminal, thus see what options are available. Doing so yields the result stated below:

```
Compiler$ python3.10 main.py --help

usage: Compiler for Panda [-h] [-o OUTPUT] [-c] [-d] [-f FILE] [-t] [-r]

Compiles source code to assembly

options:
    -h, --help            show this help message and exit
    -o OUTPUT, --output OUTPUT
                        Specify name of assembly output file
    -c, --compile         Set this flag if the output file should be compilled
    ↪  with gcc
    -d, --debug           Set this flag for debugging information, i.e., ILOC and
    ↪  Graphviz
    -f FILE, --file FILE  Path to input file, otherwise stdin will be used
    -t, --runTests        Run tests
    -r, --run             Run compilled program
```

This informs the user that one can specify an input file or provide input directly in the command line, and specifying the name of the output file is optional. Furthermore, one can control whether the file should be automatically compiled with `gcc` and directly run on the system. Many of the arguments act as flags to the compiler, such as `--debug` or `--runTests`, which are flags that specify whether a certain piece of code should be executed. Much of the setup of `argparse` is omitted, in this example, but the essential part of the functionality is listed below:

```python
args = argparser.parse_args()

if args.runTests:
    runner = unittest.TextTestRunner(verbosity=2)
    result = runner.run(testing.test.load_tests(args))
    ...
else:
    PandaCompiler(args).compile()
```

If the `--runTests` flag is set, then it will take priority over the regular compiler functionality. However, it is still possible to specify `--debug`, as debugging information may be useful in case some tests fail. Debugging information is information such as graphical representation of data structures and pretty printed ILOC code – sequential assembly IR. Testing will be explained in depth later.

3 Phases

3.1 Lexical Analysis

3.2 Parsing

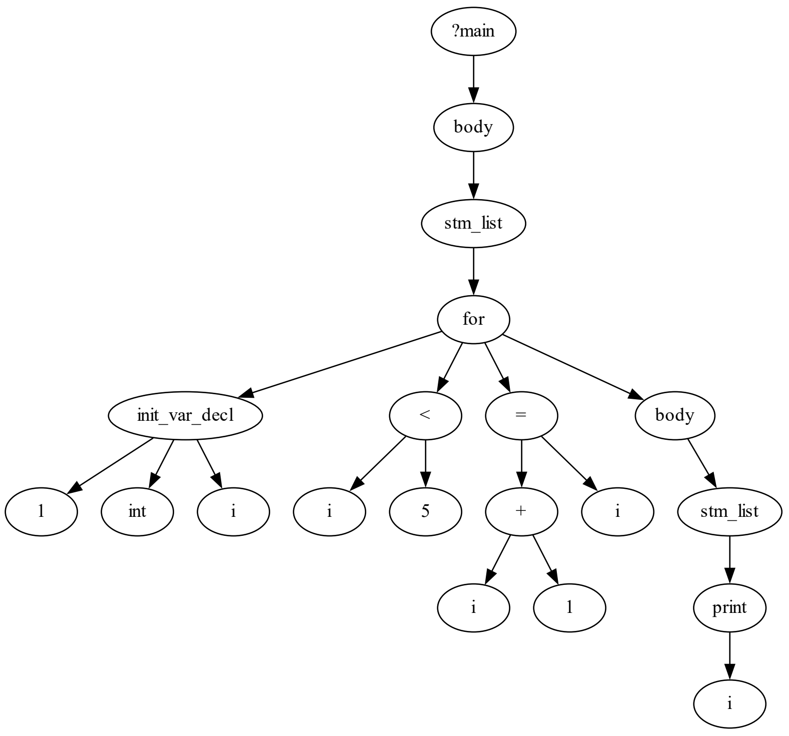


Figure 3.1: Abstract Syntax Tree.

3.3 Symbol Collection

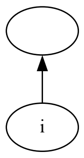


Figure 3.2: Symbol collection.

3.4 Desugaring

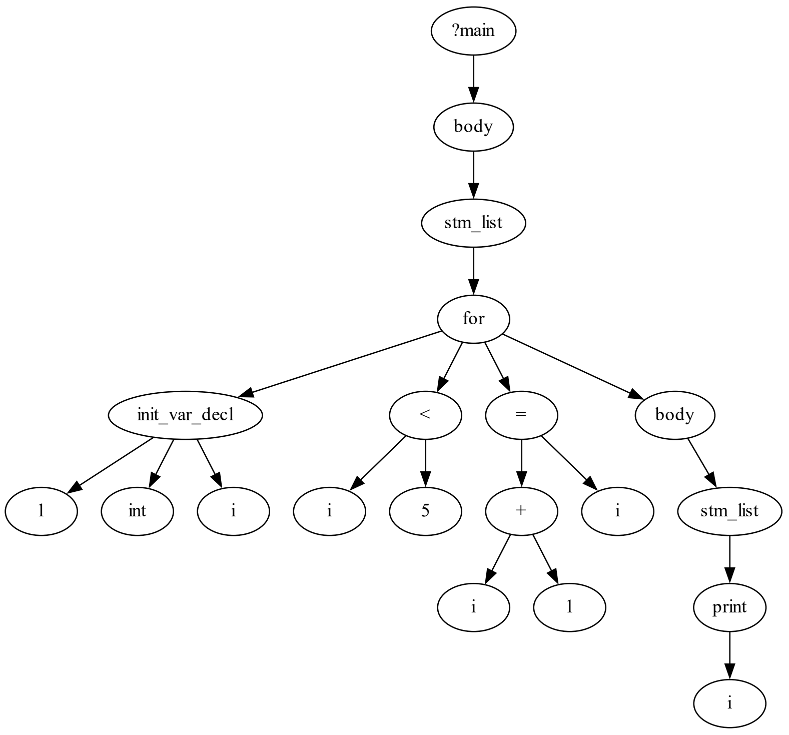


Figure 3.3: Desugaring tree.

3.5 Code Generation

3.5.1 Stack Machine

3.5.2 Register Allocation

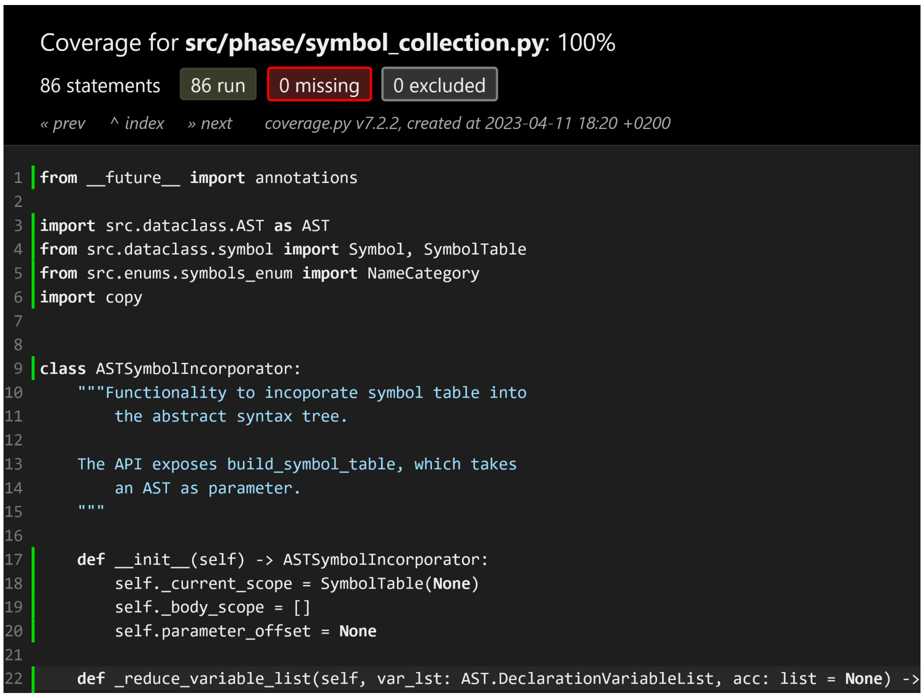
4 Testing

4.1 Unittest

```
Compiler$ python3.10 main.py --runTests

runTest (testing.test.TestCase)
Testing testing/test-cases/declaration_init_function.panda ... ok
runTest (testing.test.TestCase)
Testing testing/test-cases/fibonacci_classic.panda ... ok
...
runTest (testing.test.TestCase)
Testing testing/test-cases/statement-while.panda ... ok
runTest (testing.test.TestCase)
Testing testing/test-cases/static_nested_scope.panda ... ok
runTest (testing.test.TestCase)
Testing testing/test-cases/summers.panda ... ok

----------------------------------------------------------------------
Ran 21 tests in 2.261s

OK
```

4.2 Coverage

```
Compiler$ python3.10 -m coverage run main.py --runTests -d

Name                                     Stmts   Miss  Cover
------------------------------------------------------------
main.py                                     21      1    95%
src/compiler.py                             54      3    94%
src/dataclass/AST.py                       125      0   100%
src/dataclass/iloc.py                       22      0   100%
src/dataclass/symbol.py                     34      2    94%
src/enums/code_generation_enum.py           38      0   100%
src/enums/symbols_enum.py                    5      0   100%
src/phase/code_generation_stack.py         231      3    99%
src/phase/emit.py                          128      6    95%
src/phase/lexer.py                          44      8    82%
src/phase/parser.py                        101      3    97%
src/phase/parsetab.py                       18      0   100%
src/phase/symbol_collection.py              86      0   100%
src/phase/syntactic_desugaring.py           65      0   100%
src/printer/ast_printer.py                 141      3    98%
src/printer/generic_printer.py              17      0   100%
src/printer/symbol_printer.py               40      0   100%
src/utils/error.py                           5      0   100%
src/utils/interfacing_parser.py              1      0   100%
src/utils/label_generator.py                 9      0   100%
src/utils/x86_instruction_enum_dict.py       2      0   100%
testing/test.py                             73      0   100%
------------------------------------------------------------
TOTAL                                     1260     29    98%
Wrote HTML report to htmlcov/index.html
```



Coverage for **src/phase/symbol_collection.py**: 100%

86 statements 86 run 0 missing 0 excluded

« prev ^ index » next coverage.py v7.2.2, created at 2023-04-11 18:20 +0200

```python
from __future__ import annotations

import src.dataclass.AST as AST
from src.dataclass.symbol import Symbol, SymbolTable
from src.enums.symbols_enum import NameCategory
import copy


class ASTSymbolIncorporator:
    """Functionality to incoporate symbol table into
        the abstract syntax tree.

    The API exposes build_symbol_table, which takes
        an AST as parameter.
    """

    def __init__(self) -> ASTSymbolIncorporator:
        self._current_scope = SymbolTable(None)
        self._body_scope = []
        self.parameter_offset = None

    def _reduce_variable_list(self, var_lst: AST.DeclarationVariableList, acc: list = None) ->
```

Figure 4.1: Coverage HTML report.

5 Performance Comparison

6 Evaluation

6.1 Language Considerations

6.2 Further Development

7 Conclusion

Bibliography

Beazley, David M. (2018). *PLY (Python Lex-Yacc).* https://www.dabeaz.com/ply/ply.html. Online; accessed 16. April 2023.

Copper, Keith D. and Linda Torczon (Oct. 2022). *Engineering a Compiler.* 3. Morgan Kaufmann publishers. ISBN: 978-0-12-815412-0.