Compiler for Panda
BADM500: Bachelor Project

Author

KIAN BANKE LARSEN

kilar20@student.sdu.dk

Supervisor

KIM SKAK LARSEN

Professor

SDU

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
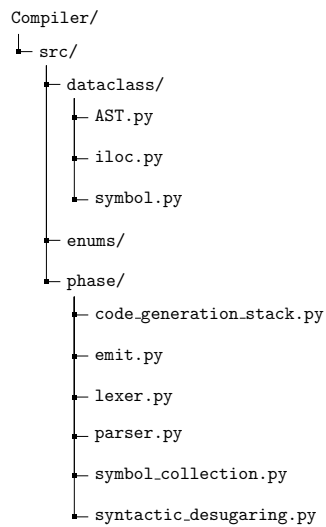
this is a test

2.1 Project Structure

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

Figure 2.1: File tree (first half).

```
Compiler/
└── src/
    ├── printer/
    │   ├── ast_printer.py
    │   ├── generic_printer.py
    │   └── symbol_printer.py
    ├── utils/
    └── compiler.py
├── testing/
│   ├── test-cases/
│   └── test.py
├── main.py
├── README.md
└── ...
```
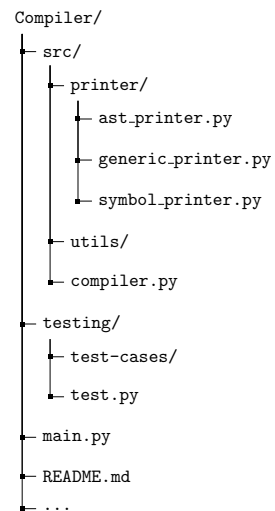
Figure 2.2: File tree (second half).

2.2 Compiler Usage

```
1  Compiler$ python3.10 main.py --help
2
3  usage: Compiler for Panda [-h] [-o OUTPUT] [-c] [-d] [-f FILE] [-t] [-r]
4
5  Compiles source code to assembly
```

```
 6
 7  options:
 8      -h, --help             show this help message and exit
 9      -o OUTPUT, --output OUTPUT
10                             Specify name of assembly output file
11      -c, --compile          Set this flag if the output file should be compilled
   ↪   with gcc
12      -d, --debug            Set this flag for debugging information, i.e., ILOC and
   ↪   Graphviz
13      -f FILE, --file FILE  Path to input file, otherwise stdin will be used
14      -t, --runTests         Run tests
15      -r, --run              Run compilled program
```

2.3   Design Patterns

Phases

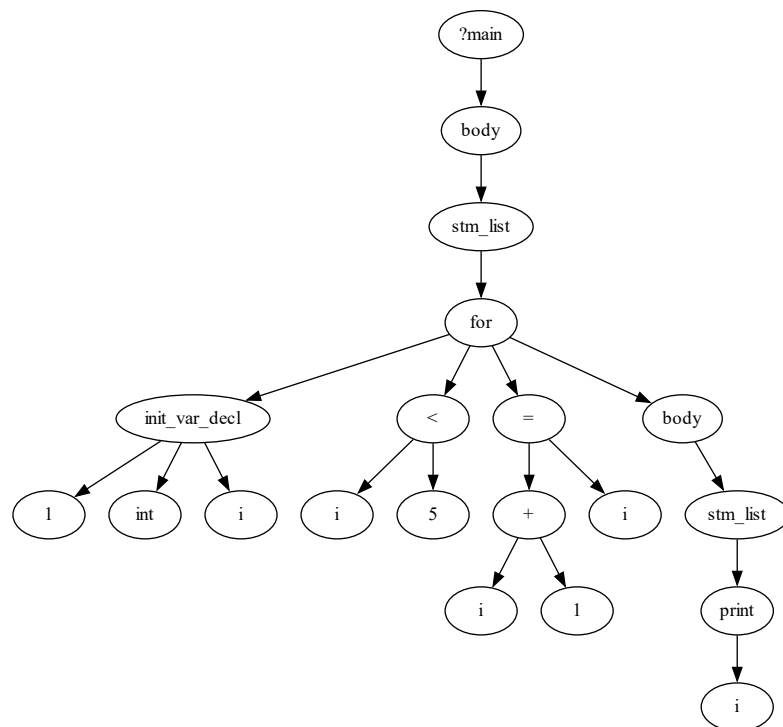## 3.1 Lexical Analysis

3.2 Parsing

Figure 3.1: Abstract Syntax Tree.

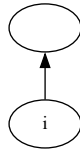## 3.3 Symbol Collection
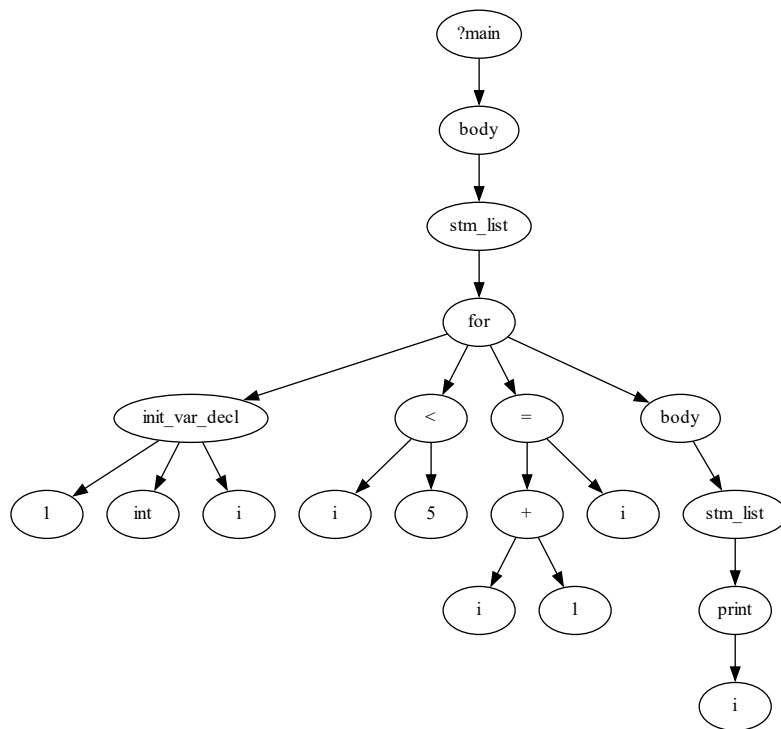

Figure 3.2: Symbol collection.

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

```
1   Compiler$ python3.10 main.py --runTests
2
3   runTest (testing.test.TestCase)
4   Testing testing/test-cases/assignment.panda ... ok
5   runTest (testing.test.TestCase)
6   Testing testing/test-cases/declaration_init_function.panda ... ok
7   runTest (testing.test.TestCase)
8   Testing testing/test-cases/fibonacci_classic.panda ... ok
9   ...
10  runTest (testing.test.TestCase)
11  Testing testing/test-cases/statement-while.panda ... ok
12  runTest (testing.test.TestCase)
13  Testing testing/test-cases/static_nested_scope.panda ... ok
14  runTest (testing.test.TestCase)
15  Testing testing/test-cases/summers.panda ... ok
16
17  ----------------------------------------------------------------------
18  Ran 21 tests in 2.261s
19
20  OK
```

4.2 Coverage

```
1   Compiler$ python3.10 -m coverage run main.py --runTests -d
2
3   Name                                     Stmts   Miss  Cover
4   ------------------------------------------------------------
5   main.py                                     21      1    95%
```

```
 6   src/compiler.py                              54    3    94%
 7   src/dataclass/AST.py                        125    0   100%
 8   src/dataclass/iloc.py                        22    0   100%
 9   src/dataclass/symbol.py                      34    2    94%
10   src/enums/code_generation_enum.py            38    0   100%
11   src/enums/symbols_enum.py                     5    0   100%
12   src/phase/code_generation_stack.py          231    3    99%
13   src/phase/emit.py                           128    6    95%
14   src/phase/lexer.py                           44    8    82%
15   src/phase/parser.py                         101    3    97%
16   src/phase/parsetab.py                        18    0   100%
17   src/phase/symbol_collection.py               86    0   100%
18   src/phase/syntactic_desugaring.py            65    0   100%
19   src/printer/ast_printer.py                  141    3    98%
20   src/printer/generic_printer.py               17    0   100%
21   src/printer/symbol_printer.py                40    0   100%
22   src/utils/error.py                            5    0   100%
23   src/utils/interfacing_parser.py               1    0   100%
24   src/utils/label_generator.py                  9    0   100%
25   src/utils/x86_instruction_enum_dict.py        2    0   100%
26   testing/test.py                              73    0   100%
27   ------------------------------------------------------------
28   TOTAL                                      1260   29    98%
29   Wrote HTML report to htmlcov/index.html
```

5

Performance Comparison

6

Evaluation

7

Conclusion

Bibliography

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