Department of Mathematics & Computer Science
University of Southern Denmark | IMADA
Sunday 14th May, 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, named Panda for no particular reason. The compiler is simple in the sense that some decisions have been made to ease the process, 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` (the project's main file). Moreover, a `README.md` has been written as a quick-start guide.

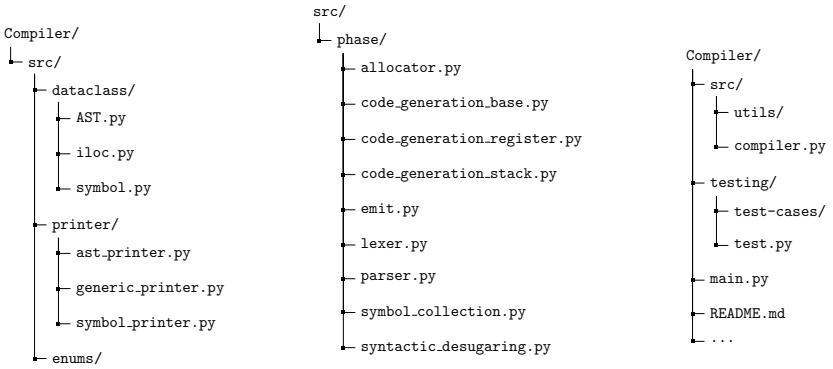

Figure 2.1: Project file tree.

2.2 Python Lex-Yacc

PLY is a native Python tool, relying on reflection, used to automatically generate scanners and LALR(1) parsers. 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. It has been chosen to use PLY in order to reserve more time for the compiler itself, though studies show that most compilers use hand-coded scanners (Copper and Torczon 2022, p. 69). However, tool-generated parsers are more common than hand-coded parsers (ibid., p. 85).

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: **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, though crucial 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 accommodated in the AST data class, since any subtree is a valid tree and all nodes are AST nodes. 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. Other examples apply, of course, as every class must accommodate every principle. This was just a quick review.

One of the big decisions regarding behavioral design patterns has been whether to use Visitors, just like in the well-known SCIL compiler from the DM565 course. It was decided not to use visitors, because Python 3.10 comes with a new cool feature, namely match statements, which makes it possible to exploit the benefits of structural pattern matching. Although it is nice to let the data structure decide its iteration, I still prefer having everything written explicitly when learning to write a compiler. Using match statements requires one to repeat the iteration logic for every new operation, but that somewhat also makes it easier to implement new logic, as one does not have to remember the visitor pattern. Another useful creational design pattern is the Singleton pattern used in `label_generator.py`. This makes it possible to retrieve the label generator in any class, while preserving the state on `count`, without having to pass the object through all the phases manually.

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 because it adds a lot of user-friendliness to the compiler. `argparse` provides the opportunity to query the compiler usage in the terminal, thus showing 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] [-s]

Compiles source code to assembly

options:
    -h, --help            show this help message and exit
    -o OUTPUT, --output OUTPUT
                          Name of assembly output file
    -c, --compile         Compile output with gcc
    -d, --debug           Debugging information, i.e., ILOC and Graphviz
    -f FILE, --file FILE  Path to input file; default is stdin.
    -t, --runTests        Run tests
    -r, --run             Run compilled program
    -s, --stack           Use stack only; default is registers
```

This informs the user that one can specify an input file or provide input directly in the command line. 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. All pip requirements needed for running `main.py` can be installed using `pip install -r requirements.txt` – file located in the root of the project.

```
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

One aspect that is important to consider is time. Keeping track of when various things happen is hard.

> *"Some decisions are made when the compiler is designed, at design time. Some algorithms run when the compiler is built, at build time. Many activities take place when the compiler itself runs, at compile time. Finally, the compiled code can execute multiple times, at runtime."*
>
> — Copper and Torczon 2022, p. 8

Clearly most time is spent on design time, because it is at design time the compiler has been created in the development environment. This includes time spent on designing an interface and writing the code. Compile time is equally important, as it can be coded using more or less efficient algorithms. Perhaps the compiler depends on certain packages being available on the machine, just as various python packages are used in this project. Some code has a very short lifetime in compile time before spending the rest of its life in runtime. It is therefore worth spending substantially more time during compile time in order to perform code analysis and subsequent optimization, such that the code running can be more efficient in some metric. The code could for example be optimized to run faster or save power. Fast programs are wanted when analyzing big data or fast response is needed. Focusing on power saving programs is key when targeting portable devices.

A typical three-phase compiler is designed as shown in Figure 3.1.

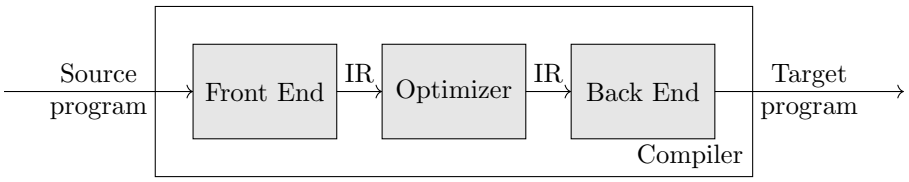

Figure 3.1: Three-phase compiler.

There is no restriction on what the individual stages can contain, as it depends entirely on architecture and program needs. The front end's responsibility is to understand the input program, such that it is possible to generate correct and meaningful code in the back end. The breakdown of Panda is presented in Figure 3.2.

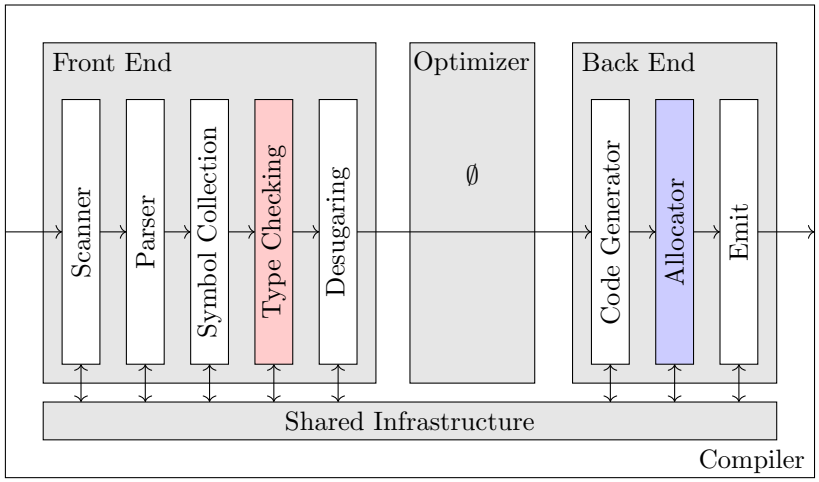

Figure 3.2: Internal structure of Panda.

Type checking has been colored red because it has not been implemented yet, though it would be preferable to have functionality to refrain users from doing something nonsensical, for example, assigning a function to an integer variable.

```
1 int a; int main(){} a = main;
```

This error will cause the compiler to raise a value error during code generation, as the match statement for case `AST.StatementAssignment` is designed to do exactly that for any symbol with `NameCategory` not parameter or variable – function is neither. It is wanted that the compiler stops compiling on such error, but it should never crash because of an uncaught exception. The user must always be informed properly about what went wrong, without having to deal with an indifferent stack trace.

Optimizations like instruction selection, instruction scheduling and peephole etc. could have been interesting to implement, but this is unfortunately beyond the scope of this project, since time did not allow. The stage is therefore skipped.

Panda can either produce stack machine code or register allocation code. The allocator phase is colored blue because it is clearly only used upon compiling source code to assembler code utilizing CPU registers.

The individual phases presented in Figure 3.2 will be reviewed in detail in the subsequent sections of this chapter. As a consequence of having studied the SCIL compiler, it will be clear that several things are done similarly, or at least inspired by SCIL to some extent.

3.1 Scanner

The scanner, or lexical analyzer, is the first phase of the compiler's front end. The scanner reads a stream of characters and produces a stream of words by aggregating the characters. For each word, it determines if the word is valid in the source language, and each valid word is assigned a syntactic category (corresponding to terminal symbols in the language's grammar) used by the parser.

The scanner is the only phase in the compiler that touches every character of the source program. Because grouping characters is a simple task, scanners lend themselves to fast implementations. The Lex part of PLY is an automatic scanner generation tool. Lex requires a token list and specification of token values in order to produce a recognizer. The recognizer is based on a mathematical description of the language's lexical syntax in form of regular expressions and finite automatons. What method Lex use to produce the scanner is unknown, but it can be achieved using Kleene's construction as presented in Copper and Torczon 2022, p. 45.

The documentation for Lex can be found in section 4 Beazley, David M. 2018, and the guide is almost identical to the steps implemented in SCIL. The author of the tool tried to stay faithful to the way in which traditional Lex/Yacc tools work, so there was nothing surprising about how it should be done.

First, it is necessary to make a list of reserved words, as otherwise it will not be possible to separate them from regular identifiers, because reserved words are a subset of legal identifiers. The reserved words are something like `if`, `else`, `return` and so on. These are specified in Python using a dictionary with word as key and token as value.

```
reserved = {
    'print': 'PRINT',
    'return': 'RETURN',
    ...
}
```

The immutable list of tokens is the union of tokens and reserved words. The list constitutes all valid words of the source program.

```
tokens = (
    'IDENT', 'INT', 'FLOAT',
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE',
    ...
) + tuple(reserved.values())
```

Rules for individual tokens are specified with a raw regex string as shown below:

```
t_PLUS = r'\+'
t_MINUS = r'-'
```

The prefix `t_` is used to indicate that it defines a token. The string must be compatible with Python's `re` module.

A token can also be specified as a function if some kind of action needs to be performed. This is particularly useful in connection with separating identifiers from reserved words. In that case, it will be possible to lookup a given identifier in reserved words. If the relevant key is not found, then it can be concluded that the token in question is just a regular identifier; otherwise, the found reserved token will be returned.

```
def t_IDENT(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'IDENT')
    return t
```

This approach is recommended as it greatly reduces the number of regular expression rules.

3.2 Parsing

Parsing is the second phase of the compilers front end. The parser's task is to determine whether a stream of tokenized words produced by the scanner constitutes a valid sentence in the programming language. The parser uses a context-free grammar to derive a syntactic structure for the program, fitting the tokenized words into the grammatical model. If the parser determines that the tokenized program is a valid program, it builds an intermediate representation (shortened IR). The resulting IR of this phase is a graphical IR, acyclic parse tree, because it encodes the program structure well.[1] Choosing the correct IR has major consequences on what information that can be encoded. It is necessary to have information encoded explicitly, as it is not possible to modify the implicit meaning of some structure.

The Yacc part of PLY uses a LALR(1) parser, which is a bottom up parser, meaning that the parser will attempt to build a derivation bottom up. Though we do not have to deal with the parser implementation, it is worth mentioning that top-down parsers are usually more intuitive, but bottom up parsing is more practical and efficient for complex languages.

The parser is used as a pull parser, meaning that the parser will request a new token whenever it is ready for more input. This is set up by letting the parser control both the input program and the lexer.

```
src.phase.parser.parser.parse(
    user_program,
    lexer=src.phase.lexer.lexer
)
```

[1] We will consider the properties of a linear IR in section 3.5.

The parsing result is provided by side effect, which is why the graphical IR can be retrieved by reading the variable called `interfacing_program` located in `utils/interfacing_parser.py`.

The parser implementation has been done following the guidelines presented in section 6 Beazley, David M. 2018. Moreover, the grammar implemented is heavily inspired by how it was done in SCIL.

The most prominently used idea is to store the entire program as a function that can be executed by the operating system by calling `main` – the programs single access point. In this way, the idea of stack frames is already established, and the content of the program is thereby just the function body, which in turn can contain other function calls. The starting grammar rule is stated below:

```
def p_program(t):
    'program : body'
    interfacing_parser.the_program = AST.Function(
        "?main", None, t[1], t.lexer.lineno)
```

Since the source program must be structured as specified by `Body`, it could be interesting to see what fields are available.

```
@dataclass
class Body(AstNode):
    decls: DeclarationList
    stm_list: StatementList
    lineno: int
```

The `Body` class is annotated with `dataclass`. Python data classes are specially structured class optimized for storage and representation, and it allows for concise class declaration since the constructor is generated automatically.

To make the language semantics easy to understand, it has been decided to constraint that declarations of variables must come before any usage. Declarations and statements are both optional (body can be empty), though useless without statements. Mixing declarations and statements can be confusing to handle, and it will be elaborated in section 3.3 why it encourages trouble.

```
def p_body(t):
    'body : optional_declarations optional_statement_list'
    t[0] = AST.Body(t[1], t[2], t.lexer.lineno)
```

Associativity of tokens can be encoded directly in the grammar, but can also be specified as precedence directives for the parser. It makes the grammar simpler when one can ignore this kind of trouble. The precedence directives stated below specify that the parser must reduce when encountering `TIMES`, `DIVIDE` etc. and shift when encountering `EQ`. The directives go from lowest to highest precedence. The `nonassoc` directive used on comparison based operators are very useful, because it refrains the user from specifying some like `3 < x < 7`. The semantic

of this statement is powerful, although difficult to express directly in code, as it is syntactic sugar for two subsequent statements. The `nonassoc` directive will cause the parser to throw an error whenever it encounters such case.

```
precedence = (
    ('nonassoc', 'NEQ', 'LT', 'GT', 'LTE', 'GTE'),
    ('right', 'EQ'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE')
)
```

Parsing is a relatively large operation, but it was a representative sample.

To gain better insight into the resulting IR of the parsing phase, one can advantageously use the printer class to create a graphical representation of the DAG data structure. We will consider the input program stated in Figure 3.3.

```
int j = 5;
for(int i = 1; i < 5; i = i + 1){}
```

Figure 3.3: Example program.

The resulting abstract syntax tree is given in Figure 3.4.

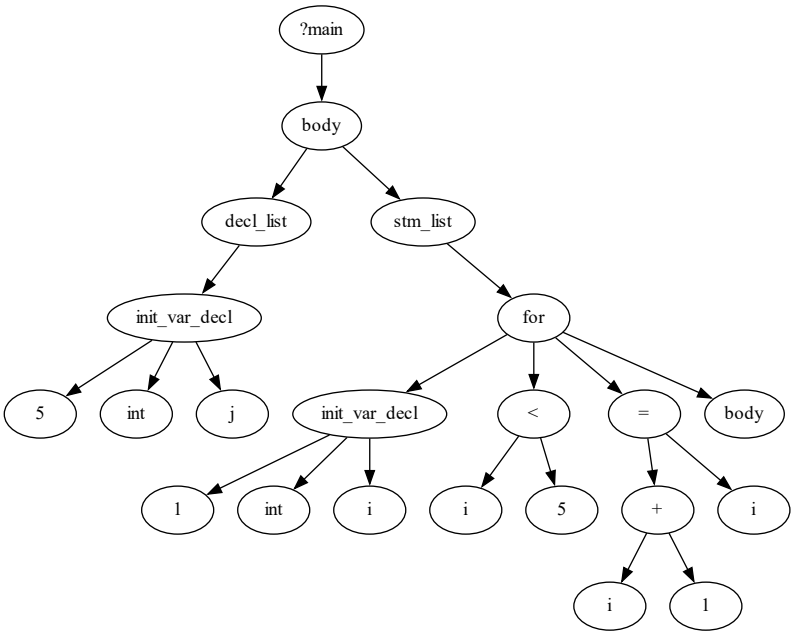

Figure 3.4: Abstract Syntax Tree.

The input program stated in Figure 3.3 is a good example because it is a small program with an associated small parse tree, although the parse tree is more verbose than it necessarily should be. Furthermore, there are two different uses of the `init_var_decl` node – this node contains syntax such as `int i = 1;`. All usages of that node is syntactic sugar for a declaration and simultaneous assignment. The iterative `for` construction uses it as a parameter, so that can just be transformed directly when visited. The story is a little more complicated for other usages of this node, since code for declarations is first generated after all statements belonging to a `body` node have been visited. It has therefore been necessary to add a desugaring phase to transform the AST, as we will see later in section 3.4.

Panda is designed in such a way that every curly bracket enclosed environment creates a new scope. The default scope is global scope. Every scope consists of a single `body` node – the grammar is `'new_scope : LCURL body RCURL'`. Recall that a `body` node can contain optional declarations followed by optional statements. Just to mention boolean values, the convention used is that the integer value 0 is `false`; otherwise `true`.

The syntax is very C-like, but below are some examples of the different language constructions. Starting with a `for` loop:

```
for(int i = 1; i < 5; i = i + 1){...}
```

Then a `while` loops follows naturally:

```
while(...){...}
```

The `if-else` flow control statement:

```
if(...){...} else{...}
```

The classical `print` statement:

```
print(3 + 5);
```

And finally function calls:[2]

```
int a; int b = 1;
int fee(int arg){... return ...;}
void foo(){...}

a = fee(b);
foo();
```

More examples can be found in the `/testing/test-cases/` folder.

[2] Remark: it is not valid to assign `void` function calls to any variable.

3.3 Symbol Collection

Symbol collection is the third phase of the compilers front end. The compiler discovers the names and properties of many entities during parsing. For each name used in the program, the compiler needs a variety of information before it can generate code to manipulate that entity. The Symbol Collection phase implements a recursive traversal of the AST in order to collect symbols into symbol tables.

Each scope has its own symbol table and each symbol within that scope is stored in that symbol table. The sole purpose of scopes is to encapsulate symbols so that logically separate parts of the program do not interfere. The introduction of scopes makes it possible to reuse symbol names, which is convenient, and a necessity when using recursion. The scoping mechanism implemented in Panda is static nested scoping. "Static" because it can be determined where a symbol is accessible and visible by reasoning about the lexical structure of the source program – this is often a cognitive comprehensive task when reading programs based on dynamic scoping. "Nested" refer to the hierarchical separation of scopes. The primary purpose of a symbol table is to resolve names. When the compiler finds a symbol, it needs a mechanism that maps that symbol back to its declaration; otherwise we would not be able to locate the entity in memory.

The compiler for Panda only supports simple 64 bit scalar variables and functions, and some additional information such as source register (`SR`) and `escaping` which will be elaborated in the code generation phase (section 3.5). The `Symbol` class is defined as stated in Figure 3.5.

```
@dataclass
class Symbol:
    type: str
    kind: NameCategory
    info: int
    SR: int = None
    escaping: bool = False
```

Figure 3.5: Symbol to be stored in symbol table.

Each symbol encountered is stored in a symbol table associated with information about its `type`, `kind` (variable, parameter or function) and `info` (offset such that the variable can be located on the stack). The following has been inserted, to provide insight into what happens when encountering a function declaration:

```
case AST.DeclarationFunction(type, func, lineno):
    symval = Symbol(type, NameCategory.FUNCTION, func)
    self._current_scope.insert(func.name, symval, lineno)
    self._current_scope = SymbolTable(self._current_scope)
    self._build_symbol_table(func)
case ...
```

In this case, `_current_scope` is the active symbol table. It is thereby possible to instantiate a `Symbol` with information about the function and insert it into the table. Before recursively collecting symbols within that function, a new symbol table is instantiated for that scope.

The way the symbol table is declared is shown below:

```
@dataclass(init=False)
class SymbolTable:
    level: int
    parent: SymbolTable
    _tab: dict

    def __init__(self, parent: SymbolTable) -> SymbolTable:
        self._tab = {}
        self.level = parent.level + 1 if parent else 0
        self.parent = parent
```

The symbol table itself takes care of keeping track of its lexical `level`, its parent, and the actual symbol table named `_tab`. The `parent` field makes it possible to search upwards in the structure towards the global scope, if the symbol searched for is not at the current level.

Scopes create a tree structure, but where the leafs point to the root and not the other way around. Considering the input program stated in Figure 3.3 again, we see, in Figure 3.6, that `j` is declared in the global scope (root, level 0) and `i` is declared at level 1 immediately incident to global scope.

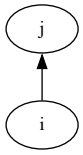

Figure 3.6: Symbol collection.

Because the example just presented is relatively minimal, a more exciting input program is considered below:

```
int a;
if(3<4){int b;} else{int c; if(3){int c;} else{int d;}}
while(4){int d;}
```

It is here clearly shown how to quickly introduce some scopes by using `if-else` statements, since all curly bracket enclosed environments constitute a new scope. Furthermore, it is also noted that two versions of d are accessible in two scopes respectively.

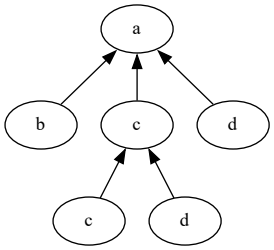

Figure 3.7: Another symbol collection example.

The symbol collection figures are automatically generated using `printer/-symbol_printer.py`. The feature is only provided when executing the compiler with the debug option.

To return to the issue of mixing declarations and statements, it is quickly realized that this places greater demands on parser flexibility, since it will no longer be possible to make a discrete segregation of these groups. As such, it makes no difference to the symbol collection phase, as we already have to recurse through both declarations and statements. I will though make code generation more complicated, because code for functions declared within the scope will be generated simultaneously with statements, if this is not handled with some special case logic. Besides, the mixing also makes it confusing what variable that accessed whenever, since some declaration can appear near the end of the scope – the declaration is valid for the entire scope and not just after the line it appears – this is a hypothetical implementation, open for discussion, of course.

To give an example of what problems may arise with this implementation, please consider the code fragment below. At first sight, one will assume that the input program is valid, but as it is considered, one will realize that `a` is not defined before `print(a)` is called. This would have been harder to spot of if the distance between print and declaration of a spanned multiple lines.

```
1 void main(){print(a); int a = 8;}
2 a = 5;
3 main();
4 int a;
```

3.4 Desugar

Desugar is the fourth, and last, phase of the compiles front end. Desugar is a phase that has been included because it was realized that the way nodes were visited during code generation was inexpedient, although necessary. The problem is only associated with the grammar:

```
variable_init_declaration :  type IDENT ASSIGN expression SEMICOL
```

Code for statements is produced before producing code for declarations. This is because we want to strangulate declared functions such that they do not appear within the function executed by the operating system. The functions would otherwise execute without being called by the user.

The way the problem has been solved is by traversing declarations and collecting all nodes of this type. The nodes are then transformed to `StatementAssignment` nodes and inserted, in the beginning and in the same order, as assignment statements too. The transformed nodes could have been advantageously removed from the tree, to minimize the IR, but since they do nothing, it has been decided to leave them as they are. The algorithmic approach is stated below:

```
1  case AST.Body(decls):
2      self._desugar_AST(decls)
3      decl_var_init_list = self._collect_decl_var_init(decls)
4      assignments = self._trans_var_init_list_to_stm(decl_var_init_list)
5      self._insert_stm(assignments, ast_node)
6      self._desugar_AST(ast_node.stm_list)
```

Reusing the input program from Figure 3.3, we can observe that the yellow nodes have been inserted, in Figure 3.8, as a result of the desugaring phase.

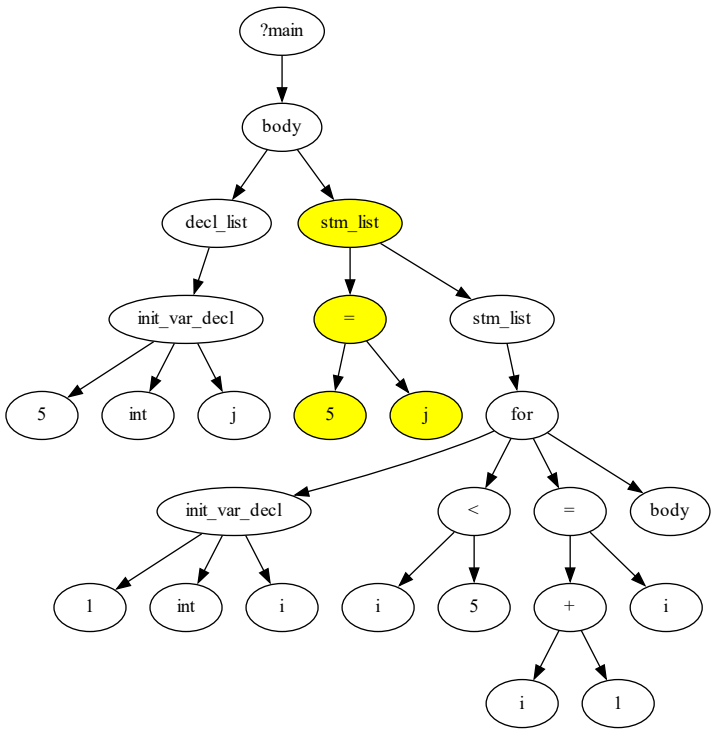

Figure 3.8: Desugaring tree.

3.5 Code Generation

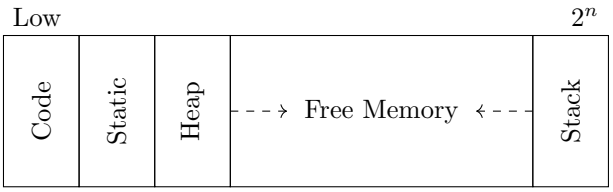

Figure 3.9: Virtual address-space layout.

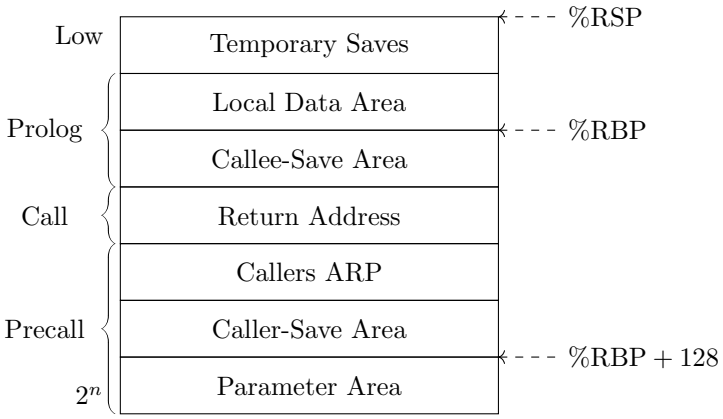

Figure 3.10: Stack Frame.

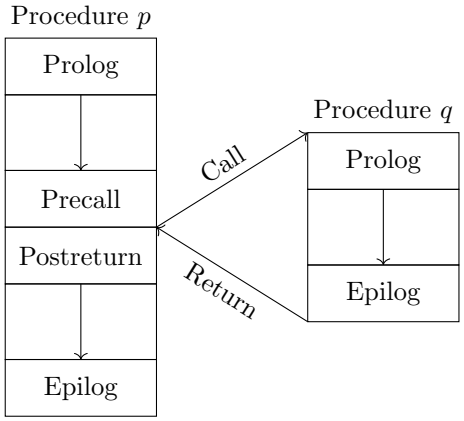

Figure 3.11: Procedure linkage.

```
1  @dataclass(init=False)
2  class Instruction:
3      opcode: Op
```

```
    args: list(Operand)

    def __init__(self, *args) -> Instruction:
        self.opcode = args[0]
        self.args = args[1:]
```

```
@dataclass
class Operand:
    target: Target
    addressing: Mode
```

```
[
    Instruction(opcode=<Op.LABEL: '7'>,
        args=(Operand(target=Target(spec=<T.MEM: 2>, val='main'),
                    addressing=Mode(mode=<M.DIR: 1>, offset=None)),)),
    Instruction(opcode=<Op.META: '8'>, args=(<Meta.PROLOG: 2>,)),
    ...
]
```

```
def _follow_static_link(self, symbol_level: int) -> int:
    level_difference = self._current_scope.level - symbol_level

    self._append_instruction(
        Instruction(Op.MOVE,
                    Operand(Target(T.RBP), Mode(M.DIR)),
                    Operand(Target(T.RSL), Mode(M.DIR)))
    )
    self._get_code_block_to_extend().extend(
        [Instruction(Op.MOVE,
                        Operand(Target(T.RSL), Mode(M.IRL, -7)),
                        Operand(Target(T.RSL), Mode(M.DIR)))
            for _ in range(level_difference)]
    )

    return level_difference
```

```
    _, symbol_level = self._current_scope.lookup(func.name)
    level_difference = self._current_scope.level - symbol_level

    self._get_code_block_to_extend().extend(
        [Instruction(Op.MOVE,
                     Operand(Target(T.RBP), Mode(M.IRL, -7)),
                     Operand(Target(T.RBP), Mode(M.DIR)))
            for _ in range(level_difference-1)]
    )
```

Figure 3.12: Tear down multiple stack frames if needed.

```
case AST.Function(body=body):
    ...
    self._ensure_labels(ast_node)

    self._append_instruction(
        Instruction(Op.LABEL,
                    Operand(Target(T.MEM, ast_node.start_label), Mode(M.DIR)))
    )
    self._prolog(body)
    self._generate_code(body.stm_list)
    self._append_instruction(
        Instruction(Op.LABEL,
                    Operand(Target(T.MEM, ast_node.end_label), Mode(M.DIR)))
    )
    self._epilog()
    self._append_instruction(
        Instruction(Op.META, Meta.RET)
    )
    self._generate_code(body.decls)
    ...
```

3.5.1 Stack Machine

```
case AST.ExpressionBinop(binop, lhs, rhs) if binop in [Op.ADD, Op.SUB, Op.DIV,
↪  Op.MUL]:
    self._generate_code(lhs)
    self._generate_code(rhs)

    self._append_instruction(
        Instruction(Op.POP,
                    Operand(Target(T.REG, 1), Mode(M.DIR)))
    )
    self._append_instruction(
        Instruction(Op.POP,
                    Operand(Target(T.REG, 2), Mode(M.DIR)))
    )
    self._append_instruction(
        Instruction(binop,
                    Operand(Target(T.REG, 1), Mode(M.DIR)),
                    Operand(Target(T.REG, 2), Mode(M.DIR)))
    )
    self._append_instruction(
        Instruction(Op.PUSH,
                    Operand(Target(T.REG, 2), Mode(M.DIR)))
    )
```

3.5.2 Register Machine

Appel 1997

```
_reg_count: int = 0
_reg_stack: list[int] = field(default_factory=list)
```

```
case AST.ExpressionBinop(binop, lhs, rhs) if binop in [Op.ADD, Op.SUB, Op.DIV,
↪  Op.MUL]:
    self._generate_code(lhs)
    self._generate_code(rhs)

    reg1 = self._reg_stack_pop()
    reg2 = self._reg_stack_pop()

    self._push_new_reg_count()

    self._append_instruction(
        Instruction(Op.MOVE,
                    Operand(Target(T.REG, reg2), Mode(M.DIR)),
                    Operand(Target(T.REG, self._reg_count), Mode(M.DIR)))
    )
    self._append_instruction(
        Instruction(binop,
                    Operand(Target(T.REG, reg1), Mode(M.DIR)),
                    Operand(Target(T.REG, self._reg_count), Mode(M.DIR)))
    )
```

```
[
    instruction_1,
    instruction_2,
    [
        instruction_3,
        ...,
        instruction_n
    ],
    instruction_n+1
]
```

3.6 Allocator

```
def perform_register_allocation(self, code: list[Instruction]) ->
↪ list[Instruction]:
    code = copy.deepcopy(code)

    self._find_labels(code)
    self._control_flow(code)
    self._liveness_analysis(code)
    graph = self._build_graph(code)
    colors = self._color_graph(graph)
    code = self._flatmap(code)
    self._rename_registers(colors, code)

    return code
```

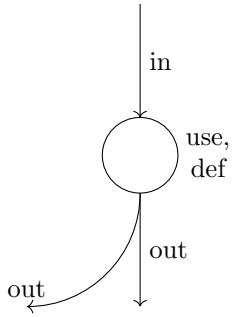

Figure 3.13: Abstract flow graph.

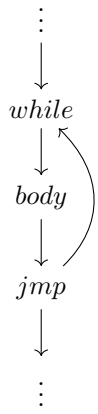

Figure 3.14: Loop construction flow graph.

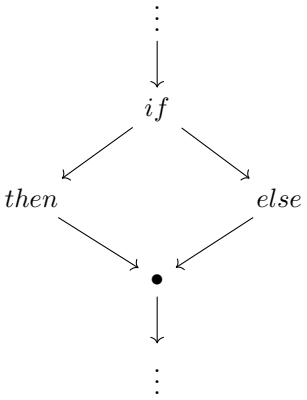

Figure 3.15: `if` construction flow graph.

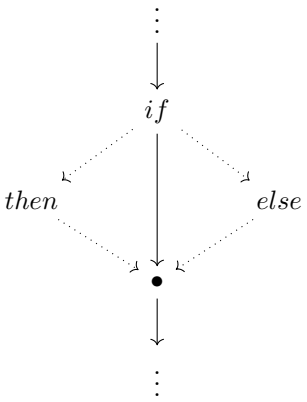

Figure 3.16: Flow restriction due to scope implementation.

```
1 # Example code
```

```
1 # Show in sets
```

```
1 # Show graph
```

```
1 # Show colors
```

This is the colored graph:

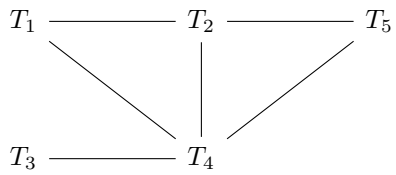

Figure 3.17: Interference graph (TODO: do real example).

3.7 Emit

The emit phase is the last phase in the compiler's back end. It is in this phase that the internal representation introduced in the code generation is made into actual assembler instructions, which can be compiled into machine code using `gcc`. Luckily it has been possible to develop an interface such that this class could be used for both the stack machine and register machine.

The mapping from the internal representation is almost 1:1, however there is small deviations such as the use of `%RSL`, which is a virtual register used to calculate links, and that register does not exist nativelty on the x86 architecture. In addition, it is necessary to handle a handful of `Meta` instructions, because it made sense to assemble these into such composite instructions, and then let the emit phase handle the actual code for this, since it is the same every time – these instructions could as well have been represented in the linear IR. The `Meta` instructions are:

```
self._enum_to_method_map = {
    Meta.CALL_PRINTF: self._call_printf,
    Meta.PROLOG: self._prolog,
    Meta.EPILOG: self._epilog,
    Meta.PRECALL: self._precall,
    Meta.POSTRETURN: self._postreturn,
    Meta.RET: self._ret
}
```

The `Meta` instructions is thus simply mapped using a dictionary in python, mapping the Enum to a function implemented within the Emit class.

The majority of these functions are functions used during procedure linkage, which is why it makes sense to declare exactly which registers are callee-save and caller-save registers:

```
self._callee_save_reg: list[str] = ["rbx", "r12", "r13", "r14", "r15", "rbp"]
self._calleer_save_reg: list[str] = ["rcx", "rdx", "rsi", "rdi", "r8", "r9",
↪  "r10", "r11"]
```

It might also have been advantageous to have specified these fields in another class, so that variable offsets did not have to be hardcoded in the code generation, but could instead have been implemented as a function on the two lists. Adding extra callee-save and caller-save would then not require futher intervention.

`Meta.CALL_PRINTF` uses labels to ensure correct memory alignment, and we therefore want to reuse `Labels` instantiated during code generations. This is achieved using the property that the class is declared singleton, and Python will therefore just return the same instance as used in code generation, without manually having to pass that instance to code emition:

```
self._labels = Labels()
```

It is important that the counter within `Labels` is not reset, since this could, in the worst case, cause clashing labels, meaning that the program would be incorrect and in turn prevent `gcc` from being able to compile it, since labels are no longer ensured to be unique.

The code snippet given below is responsible for "dispatching" any ILOC instruction to an assembly instruction. Because the ILOC instructions are provided as a list, one can simply iterate through the list and translate the individual instruction as intended. The native function `map` is used though this is not the intended use of it, as `_dispatch` accumulates its result in a field called `_code`. However, it is nice to think of it as a mapping, why it is done that way. Also, map is a concise way to do iteration even if the resulting list just contains None.

```
def emit(self, iloc_ir: list[iloc.Instruction]) -> str:
    self._program_prologue()
    list(map(self._dispatch, iloc_ir))
    self._code.append("\n")
    return "\n".join(self._code)
```

The final result of `emit` is a single string containing all the generated assembler code, including newlines. The output string can therefore just be inserted directly into any file using Python's built-in I/O library.

```
def _handle_reg_spill(self, instruction: iloc.Instruction) ->
↪  list[iloc.Instruction]:
    if instructions := self._check_register_dependent_operations(instruction):
        return instructions

    instructions.extend(
        self._check_if_spill_is_needed(instruction)
    )
    instructions.append(
        self._make_reference_to_spilled_registers(instruction)
    )

    return instructions
```

4 Testing

The main benefit of testing is the identification and subsequent mitigation of errors. It is to be expected to find errors in larger projects, which is why there must be a way to quickly and dynamically run tests during development. Testing helps software developers compare expected and actual output in order to improve quality. In this case, only system testing is carried out, although a higher granularity of testing would only enrich the project. There are many tools available to solve this task, which is why the big challenge lies in choosing the tool that best suits the problem. The most advanced tool is not always the best solution. The chosen tools in this project are the testing framework `unittest` and the coverage tool `coverage.py`. In addition, SonarLint is used to perform static code analysis within the IDE to provide best-practice hints, cognitive complexity metrics and more – not important but worth mentioning.

4.1 System Testing

The most common choice would have been to use `pytest` because it is significantly more popular than `unittest`, but for the task `unittest` seemed like the right choice. I am not able to judge whether the same could have been achieved using `pytest` with parametrized tests, but setting up `unittest` was straight forward.

I wanted a solution where it is possible to load tests dynamically, based on test files in a given folder. Hence, it would not be necessary to write more code just because tests are added to the test suite – writing test cases must not be a burden; otherwise it will not be done. The idea is as follows (in pseudo Python):

```
1 class TestCase(unittest.TestCase):
2     def __init__(self, <args>) -> TestCase:
3         super().__init__()
4 
5         def runTest(self):
6             pass
```

First, A class inheriting from `unittest.TestCase` is declared, which makes it possible to define a test case with the required interface. A single test case can then be created by creating an instance of `TestCase`. A test case is created for every pair of `file.panda` and `file.eop` by walking the directory `test-cases`.

```
def load_tests(args: argparse.Namespace) -> unittest.TestSuite:
    test_cases = unittest.TestSuite()

    for src, res in test-cases:
        test_cases.addTest(TestCase(src, res))
```

The method named `runTest` will be executed when running the test, unless otherwise specified. `runTest` has the responsibility to report whether an individual test succeeds or fails. My `runTest` does the following:

1. Compile source code;
2. Execute compiled code and pipe `std:out` to file;
3. Assert whether output and expected output is identical.

Of course, some exception handling needs to be done, but that is the basics. An interesting exception handling is when running a test developed to fail, as the exception handling in the compiler will execute `os.exit(1)` (it does not make sense to continue), so that exception must be caught in the running test in order to retrieve whatever is printed to `std:err`.

The test suite can be run as follows when the desired tests have been added:

```
runner = unittest.TextTestRunner(verbosity=2)
result = runner.run(testing.test.load_tests(args))
```

The above logic is contained within `main.py` and wrapped in an `if-else` statement. Running the tests therefore requires calling main.py with option `--runTests`, or simply `-t`, as shown below:

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

runTest (testing.test.TestCase)
Testing testing/test-cases/fibonacci_classic.panda ... ok
runTest (testing.test.TestCase)
Testing testing/test-cases/static_nested_scope.panda ... ok
...
----------------------------------------------------------------------
Ran 21 tests in 2.261s

OK
```

The test suite is automatically run on every pull request or push to GitHub as part of the CI workflow using GitHub Actions. Configuration can be found in

`/.github/workflows/unittest.yml`. A `ValueError` is raised when a test fails, otherwise the GitHub Actions workflow will not detect this.

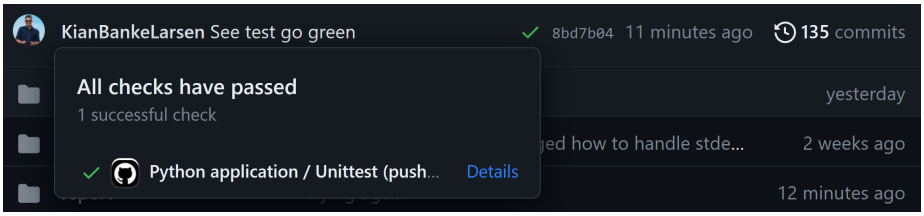

Figure 4.1: GitHub Actions for system testing.

That way, one know exactly which push or pull request caused the tests to fail, and associated code is directly available from the workflow via commit ID. History for workflow is available too.

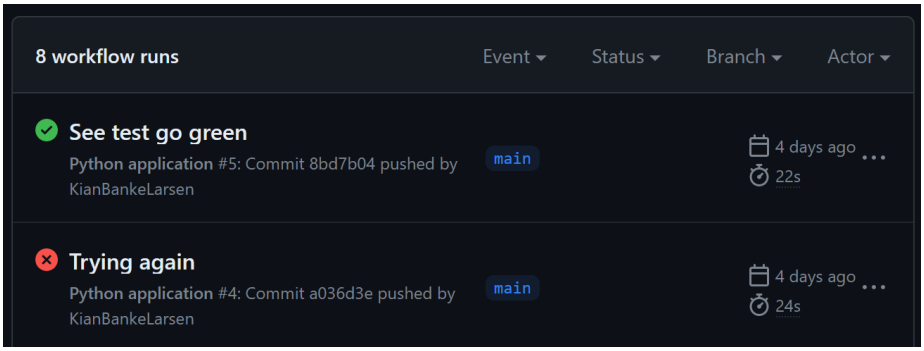

Figure 4.2: Workflow history.

A workflow build consists of the stages stated in Figure 4.3. Note that, in this example, the stack machine test passed, but the register version did not. Upon build failure, an email is sent to whom introduced the issue.

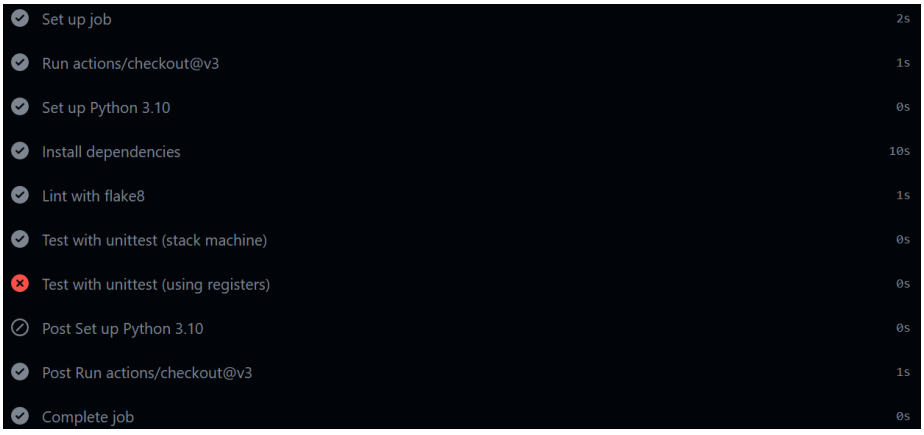

Figure 4.3: Workflow run. Red indicates failure.

The list of tests is too long to mention each one, instead two tests that caused particularly many problems have been chosen. The tests are representative because they have a relatively high complexity, as the input programs utilize recursion, loops and nested scopes. For this to work properly, it requires that the constructs and static link pointer works correctly. In addition, it must be possible to tear down several stack frames at once, since every scope introduces a new stack frame. Failing to remove a stack frame properly causes problems in the postreturn and epilog phase, and in worst case segmentation fault.

The test below is borrowed from SCIL. The test is named summers.panda.

```
int sum_recurse(int n) {
    if(n == 1) {return 1;} else {
        return n + sum_recurse(n - 1);
    }
}
int sum_loop(int n) {
    int sum, i;
    i = 1; sum = 0;
    while(i <= n) {
        sum = sum + i;
        i = i + 1;
    }
    return sum;
}

print(sum_recurse(9)); print(sum_loop(9));
print(sum_recurse(42)); print(sum_loop(42));
```

The test below does not calculate anything meaningful or difficult, but it checks that the functionality implemented in Figure 3.12 works as expected.

```
int i = 5;
int main(){
    if(1){
        if(1){
            if(1){
                i = i + 1;
                return i;
            }
        }
    }
}
print(main());
```

The goal has been to test all legal semantics thoroughly. This includes assignment to variables declared outside the current scope. Assignment to formal parameters, recursion, return from static nested scopes, comments, sugared declaration with assignment, binary operations (comparisons and arithmetic) and constructs like `while`, `for`, `if` and `print`.

this is a test

```
1 print(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+()))))))))))))));
```

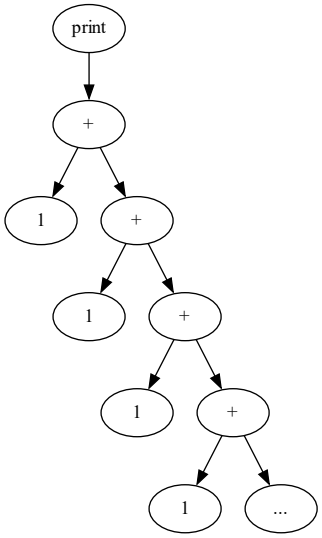

Figure 4.4: Unbalanced expression.

4.2 Coverage

Coverage cannot and should not be a measure of test quality, because the order in which functions and flow are executed has a great influence on the result. However, code coverage gives a direct warning if subsets of the code have not been tested at all. Thus, it is clear that more tests need to be written.

The code coverage measuring tool `coverage.py` is used to perform code coverage statistics on this project. Code coverage is performed in the following way:

```
Compiler$ python3.10 -m coverage erase
    && python3.10 -m coverage run -a main.py -td
    && python3.10 -m coverage run -a main.py -tsd
    && python3.10 -m coverage report

Name                                       Stmts   Miss  Cover
------------------------------------------------------------
main.py                                       24      2    92%
src/compiler.py                               70      3    96%
src/dataclass/AST.py                         125      0   100%
src/dataclass/iloc.py                         22      0   100%
src/dataclass/symbol.py                       36      2    94%
src/enums/code_generation_enum.py             38      0   100%
src/enums/symbols_enum.py                      5      0   100%
src/phase/allocator.py                       213      5    98%
src/phase/code_generation_base.py             58      3    95%
src/phase/code_generation_register.py        262      9    97%
src/phase/code_generation_stack.py           199      5    97%
src/phase/emit.py                            199     17    91%
src/phase/lexer.py                            44      8    82%
src/phase/parser.py                          101      2    98%
src/phase/parsetab.py                         18      0   100%
src/phase/symbol_collection.py               117      2    98%
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                   7      0   100%
src/utils/x86_instruction_enum_dict.py         2      0   100%
testing/test.py                               75      0   100%
------------------------------------------------------------
TOTAL                                       1884     61    97%
Wrote HTML report to htmlcov/index.html
```

Based on this output, it can be assessed that the prepared tests in the `test-cases` folder cover the code well. Note that code coverage is run with the debug flag, `-d`. This is because code in the printer files are only executed when debug is desired.

It is possible to convert the coverage data to an HTML report with the command

`python3.10 -m coverage html`. The advantage of having the data in report form is that it is possible to see which lines have been executed and which are missing, as shown in the figure below:

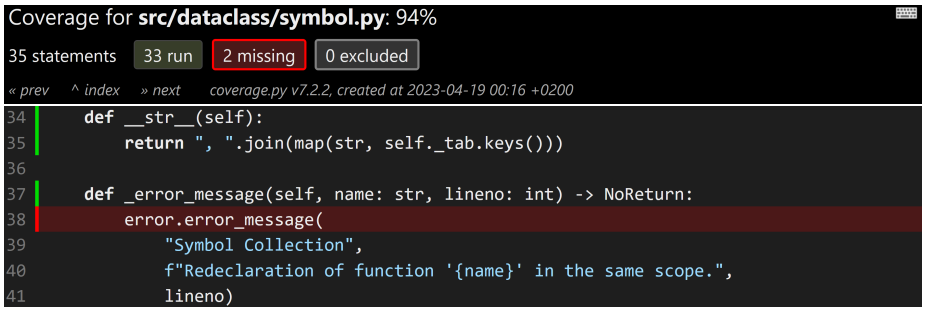

Figure 4.5: Coverage HTML report.

The HTML report also makes it possible to carry out filtering etc. in the file overview table, and thus get a nicer and more user-friendly interface.

5 Performance Comparison

introduction...

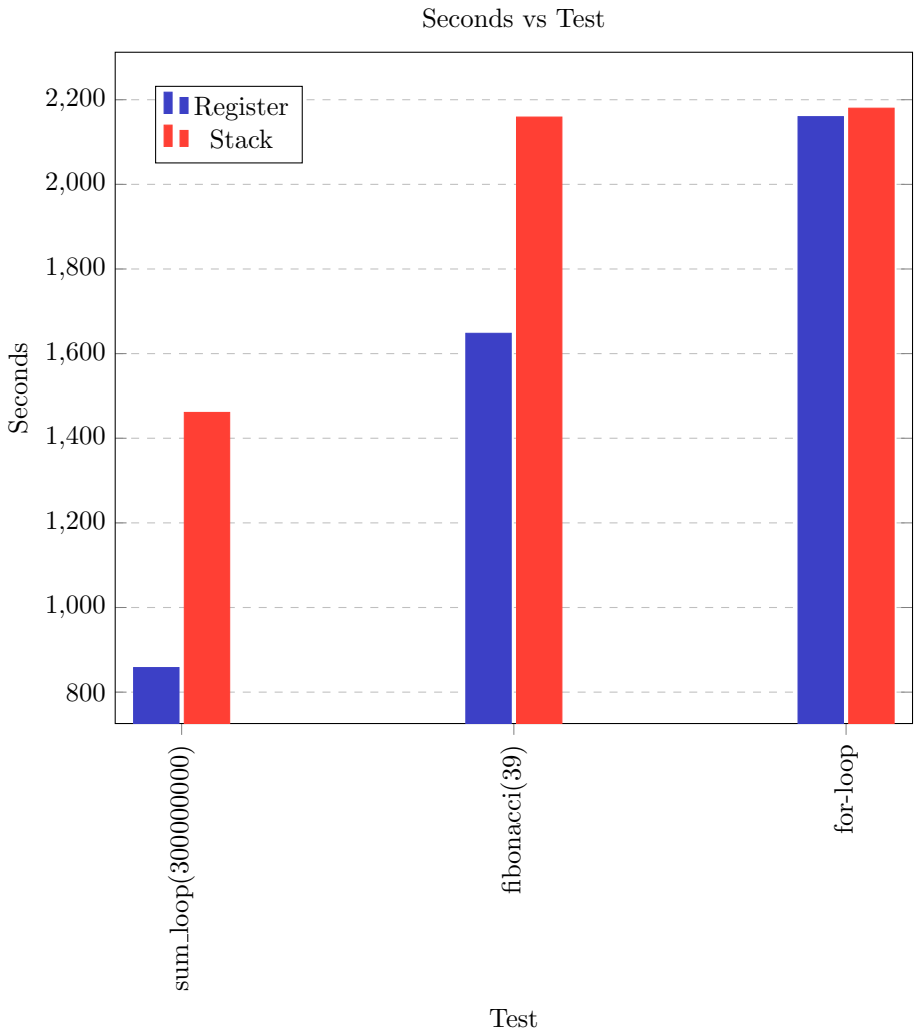

Figure 5.1: Stack Machine vs Register Machine.

second page...

6 Evaluation

This chapter is devoted to reflecting on the learning through the project, as well as describing some issues that could be interesting to work on if the project had had a longer duration.

6.1 Retrospect

There are two fundamental principles that must be observed when building a compiler. The first principle is inviolable:

> *"The compiler must preserve the meaning of the input program."*
>
> — Copper and Torczon 2022, p. 6

The second principle has a more practical aspect:

> *"The compiler must discernibly improve the input program."*
>
> — ibid., p. 7

It has been a core value throughout the project to make a well-defined compiler implementation that does not do anything inappropriate. Furthermore, the goal has been to create a clear, consistent, easily understandable grammar that a user would be able to pick up with ease. It is assessed that the first principle has been complied with. In this project, no special effort has been made in optimization apart from the preparation of an implementation that uses registers, which is why the second principle must be trivially respected. An example of an optimization that could challenge this would be instruction scheduling, since instructions are literally rescheduled after the invariant based linear IR has been generated.

The approach to the project has been academically based on literature, in order to gain an overview of new as well as old knowledge. Although this has given a confident approach to the project, it must be recognized that it has dulled the process unnecessarily. When coding a project such as a compiler that consists of relatively many lines of code, and many hours of debugging, it is necessary to

purposefully choose a theme and then learn as needed, otherwise many hours are lost on something that is not directly applicable. The hours are of course not wasted, but not beneficial for the project either.

It has always been desired that it should be possible to declare new variables in any scope, which is why it was decided to let `body` (syntactic category from parsing) be what constituted a scope. The choice later made it clear, during code generation phase, that this imposed too many constraints on how the stack should be set up, since a simple generalized procedure linkage was wanted. This resulted in suddenly having to handle all scopes as procedure calls, though it had nothing to with a procedure call. One also encounter problems when doing local register allocation, because flow is only considered locally for a scope (normally local to functions). This consequence limits the analysis to the individual scope, making the content of a `then` block, of an `if`-statement, a black box for the surrounding scope. Thus, the analysis of control flow becomes less effective, which contributes to overall poorer performance of the register machine compiler. The easiest solution, and most likely most effective, would be to redo the scope implementation. Global register allocation would be the hard option, though it would not solve the underlying problem besides patching it.

6.2 Further Development

6.3 Single Point of Contact

```
theme: jekyll-theme-slate
title: Panda Compiler
description: This is a bachelor project, Southern University of Denmark
show_downloads: true
```

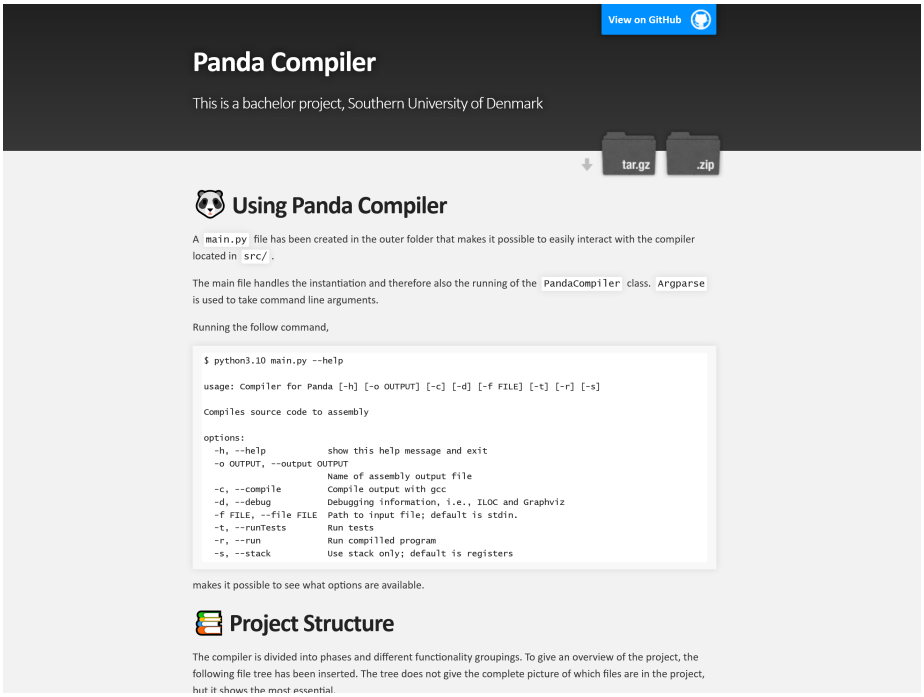

Figure 6.1: Automatically generated Jekyll Slate webpage hosted on GitHub Pages: https://kianbankelarsen.github.io/Compiler/.

7 Conclusion

Bibliography

Appel, Andrew W. (1997). *Modern Compiler Implementation in C*. English. Combridge University Press. ISBN: 978-0-52-158390-9.

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*. English. 3. Morgan Kaufmann Publishers. ISBN: 978-0-12-815412-0.