Compiler Front-End for Translating Python Subset Into Bril (CS265 Final Project)

Gabriel Raulet

December 18, 2024

1 Introduction

Bril [1] is an intermediate language designed to teach PhD students about the middle-end stage of compilers. The middle-end stage of a compiler is focused on code-improving optimizations, which are typically measured in terms of instruction count, optimized usage of the memory hierarchy, full use of hardware (for example, data-level parallelism), and so on. Bril is a typed, three-address code linear intermediate representation (IR), meaning that all instructions involve at most three operands v_0, v_1 and v_2, with the most typical instruction assigning the result of a binary operation: $v_0 \leftarrow v_1 \oplus v_2$. Because Bril was developed specifically with pedagogical intentions, performance considerations are eschewed in favor of ease of use for the lay person. In particular, Bril programs are fully specifiable as JSON objects, and this is the preferred representation.

For a student new to compilers, focusing on the middle end stage of compilation can feel a bit like putting the cart before the horse. Learning non-trivial optimizations involving data-flow analysis, value numbering, and loop optimizations can feel particularly daunting when one is dealing with an intermediate representation that is already somewhat foreign to the kinds of programming languages we are usually most familiar. When working on optimization tasks, I found myself wanting to see how different changes to a higher-level program translated into more or less efficient uses of the IR optimizations we were learning about.

The goal for this project is therefore to develop a simple tool that I believe will help students in such a position, as it is the kind of tool I would I have wanted when working through the class assignments. It also is an exercise in learning the front-end of the compilation process, a phase which is quite different in flavor yet more immediately accessible.

In this project I developed the front-end of a compiler that translates a useful subset of Python into the Bril IR. The compiler can handle a large enough subset in order to implement non-trivial integer-based arithmetic algorithms that involve a mix of loops, if-elif-else statements, recursion, and arbitrarily complicated arithmetic and logical expressions. The two basic Bril types, integers and booleans, are also the two basic types implemented in my front-end. Function definitions are implemented that support the basic types as well. All in all, the compiler is a helpful tool for converting simple (but expressive) Python programs into Bril.

I relied extensively on the ChocoPy implementation guide [2] to help understand what grammar subset I should implement. `https://github.com/gabe-raulet/chocopy2bril`

2 Implementation

The compiler is broken into three main phases: token generation (lexer), syntax analysis (parser), and code generation. The code generation phase does type checking on the fly as it generates Bril instructions.

2.1 Lexing

The purpose of lexical analysis is, essentially, to convert a string of characters (an input program text file) into a stream of tokens. A token represents the basic lexical unit of a language. Each token is represented by a pair $\langle name, value \rangle$ where the token name represents the *category* of token, and the token value represents

Figure 1: Implemented grammar

```
    program ::= {func_def}*
   func_def ::= "def" ID "(" {typed_var {"," typed_var}*}? ")" {"->" type}? ":" NEWLINE INDENT func_body DEDENT
  func_body ::= {var_def}* stmt+
  typed_var ::= ID ":" type
    var_def ::= typed_var "=" literal NEWLINE
       stmt ::= simple_stmt NEWLINE | "if" expr ":" block {"elif" expr ":" block}* {"else" : block}? | "while" expr
simple_stmt ::= "pass" | "expr" | "return" expr | ID "=" expr | "print" "(" expr ")"
      block ::= NEWLINE INDENT stmt+ DEDENT
    literal ::= "True" | "False" | NUM
       expr ::= cexpr | "not" expr | expr {"and" | "or"} expr
      cexpr ::= ID | literal | "(" expr ")" | ID "(" {expr {"," expr}*}? ")" | cexpr binop cexpr | "-" cexpr
      binop ::= "+" | "-" | "*" | "//" | "%" | "==" | "!=" | "<=" | ">=" | "<" | ">"
       type ::= "bool" | "int"

         ID ::= [a-zA-Z_][a-zA-Z_0-9]*
        NUM ::= [0-9]+
```

a specific string that matches that category. My compiler defines a token pattern recognition class whose job is to match certain patterns (defined as regular expressions) to their corresponding token category.

In this compiler there are, broadly speaking, three kinds of tokens. First, there are the so-called "exact-matching" tokens. These are the tokens whose category corresponds to a single string. Tokens in this category include arithmetic, logical and relational operators, as well as punctuation and separators (e.g. commas, parentheses, arrows, colons, etc.). These kinds of tokens are essential for telling the parser which syntax productions *can* be chosen and which productions *should* be chosen based on the surrounding context. As the name implies, these tokens are represented by their own token pattern which consist solely of the corresponding representative string. The second kind of token has a pattern defined by a finite set of valid matching strings. Due to the simplicity of the language subset implemented here, there are just three tokens here: keywords, types, and bools. Finally there are the tokens whose pattern is defined by regular expression corresponding to a potentially infinite number of matching strings. These are the identifiers and integer literals.

Generally speaking, lexical analysis is the most straightforward part of any compiler. It is self-contained, and has little complexity. In Python, there are some minor idiosyncrasies that complicate this claim a little bit. Unlike C-style languages, newline and whitespace indentation have syntactical significance and therefore should be handled with care. In particular, newlines are used to mark the end of program statements, and combinations of newlines and indentation are used to mark the endpoints of control-flow blocks. We therefore read through a single line of program text at a time (dropping out text that follows the comment delimiter '#'), generating the tokens for that line followed by the generation of a newline token. We maintain a stack storing the current indentation level. The last number on the stack represents the current indentation level. Whenever a new line is processed, we compute the amount of whitespace at the front and compare to the last value on the stack. If it is larger, we emit a indentation token and push the new level on to the stack. If it is smaller, we pop the stack until we get a value that matches the current level, emitting a "dedentation" token for each value popped. The remaining aspects of the lexer are quite simple: we use the regular expressions of each token pattern to search for a valid token, match it, and move on to the remaining text.

2.2 Parsing

The parser is responsible for the syntax analysis phase. In this phase, the stream of tokens emitted from the lexer are converted into a syntax tree representing valid program constructs. Learning how to build a parser from scratch is an incredibly good way to learn how programming languages work and why they are so powerful. It is also the most complicated part to get right, so care must be taken to make good designed decisions early on before expanding.

I wrote a recursive descent parser by hand (without a parser generator) which implemented the grammar

Figure 2: Lexer loop (lexer.py)

```
import re

def lex_text(text: str):

    stack = [0] # indentation level stack
    # each iteration is for a new program line
    while True:
        # find the first ignorable character on current line
        s = re.search(r"[ \t]*(#.*)?\n", text)
        if not s: break # nothing left, we're done
        line = text[:s.start()] # line of interest
        text = text[s.end():] # remaining text
        if not line: continue # empty line, go to next one
        # match leading whitespace
        front = re.match(r"[ ]*", line)
        l = len(front.group())
        if l > stack[-1]:
            stack.append(l) # more whitespace than last line means we are indenting
            yield Token("INDENT")
        elif l < stack[-1]: # less whitespace than last line means we are dedenting
            while l < stack[-1]:
                stack.pop()
                yield Token("DEDENT")
            assert l == stack[-1]
        # tokenize the line
        result = Token.match(line)
        while result:
            token, line = result
            yield token
            result = Token.match(line)
        # end of line token
        yield Token("NEWLINE")
    # handle remaining dedents
    while stack and stack[-1] > 0:
        stack.pop()
        yield Token("DEDENT")
```

listed in Figure 1. The parser is implemented as class which is initialized with the full sequence of tokens and a pointer to the current token. Arbitrary look-ahead is therefore possible (because we have all the tokens), but in practice I only ever look-ahead a single token for this grammar. There are basically three kinds of functions implemented for the parser class. Match-checking functions are implemented for most grammar productions which check whether the current token (and potentially its predecessor) match the correct sequence needed for the start of that given production. Matching functions actually match tokens and advance the token pointer. Get functions are called when we want to construct a syntax object, usually a tree. In Figure 3, an example of a get function for a call expression is shown at the end. Most of the get functions are similar in flavor. It asserts that the current tokens match the beginning of a call expression, and then gets the called function's identifier and any potential arguments, which would be a list of expression objects. It then returns a myast.CallExpr object with the name and arguments. I will discuss these kinds of objects in the next subsection.

Predictably, a difficult portion of the parser to implement was the part that parses expressions into syntax trees. In my first try I attempted to follow the examples given by canonical resources like the Dragon book which first parses 'factors' whose operators are multiplication and division, followed by 'terms' whose operators are addition and subtraction, and so on. Once I tried to incorporate unary operators (unary arithmetic negation and logical 'not') as well as general logical operators, I was too confused for my own comfort level to continue. Luckily, I learned about another approach called operator-precedence parsing which significantly simplified the problem for me, and my code as well. Instead of breaking everything up into an excessive number of similar looking productions, we simply use the operator precedence (higher precedence operators associate large sub-expressions then lower precedence operators) to decide which sub-expressions are grouped together.

2.3 Code generation

Last, but certainly not least, is the code generation phase. No attempt to make optimized code was made for this compiler. Simply getting the correct answers was the goal. The best way to figure out the code generation phase, I learned through trial and error, is to start with arithmetic/logical expressions, as these are the most difficult and once they are done the others follow relatively painlessly. The key is to traverse binary operator tree nodes from left to right, generating virtual registers in post-order to hold the temporary result of each internal node. Generating new virtual registers is, in a way, the same thing as implementing static single assignment as each further computed expression gets a unique identifier. The key mechanism needed is to have an object newly created for each unique scope (in this case, I really mean function body) which stores the number of the last created virtual register. We then pass this scoped object to each recursive call to the left and right sub-trees. Once the left and right sub trees have been traversed, we are given the virtual registers storing the result of each expression. We then create a new virtual register and instruction which operates on the virtual registers for the left and right sub-tree and assigns to the new virtual register. This register then stores the correct computation for the given tree root.

Once I got this working, a similar idea follows for control-flow statements (loops and branches). Within the scope object, we also maintain a number tag for the last created label. Label names correspond to the kind of flow statement that generated them (for example, then labels correspond to basic block followed when an if condition evaluates to true), plus a unique tag. The rest is a simple matter of implementing the correct bril JSON instructions, which I won't go into here.

3 Discussion

In order to verify the correctness of my compiler, I implemented a series of algorithms which can be found in the /examples directory in the github repo. All control-flow constructs are rigorously tested throughout these examples, as well as correct implementations of recursive algorithms which confirm the correctness of the function constructs.

To see an example of the compiler's output, consider example in Figure 5. This is a simple example which finds the nth prime number starting from 2. The is_prime function is not included but the full example can be found in the repo on github. In Figure 6, the generated bril is shown.

Figure 3: Parsing class and example production (parser.py)

```
class Parser(object):

    def __init__(self, tokens):
        self.tokens = tokens
        self.size = len(tokens)
        self.pos = 0

    def error(self, msg=""):
        if msg: msg = f": {msg}"
        raise Exception(f"Syntax error{msg}")

    def token(self, peek_amt=0) -> Token | None:
        """
        Returns the token that `peek_amt` away from the current
        one, or None if it doesn't exist. `peek_amt` defaults
        to 0, so the default token is the current one.
        """
        if self.pos + peek_amt < self.size:
            return self.tokens[self.pos + peek_amt]
        else:
            return None

    def advance(self):
        """
        Advance to the next token if it exists.
        """
        if self.pos < self.size:
            self.pos += 1

    def match(self, token) -> Token:
        """
        If the current token has the same "name" (e.g. "LPAREN", "COMMA", "TYPE", etc.)
        as `token`, then we advance to the next token. Otherwise an error is raised.
        """
        matchee = self.token()
        if matchee.matches(token): self.advance()
        else: self.error(f"{token} does not match current token {matchee}")
        return matchee

# ... #

def get_call_expr(self) -> myast.CallExpr:
    assert self.matches_call_expr()
    name = self.get_id()
    args = []
    self.match(Token.LPAREN)
    if not self.token().matches(Token.RPAREN):
        args.append(self.get_expr())
        while self.token().matches(Token.COMMA):
            self.match(Token.COMMA)
            args.append(self.get_expr())
    self.match(Token.RPAREN)
    return myast.CallExpr(name, args)
```

Figure 4: While loop statement object (myast.py)

```
class WhileStmt(Stmt):

    def __init__(self, cond : Expr, block: list[Stmt]):
        self.cond = cond
        self.block = block

    def get_instrs(self, scope):

        label = scope.next_label()
        entry_label = f"entry.{label}"
        body_label = f"body.{label}"
        exit_label = f"exit.{label}"

        instrs = [{"label" : entry_label}]
        instrs += self.cond.get_instrs(scope)
        cond = instrs[-1]["dest"]
        instrs.append({"op" : "br", "labels" : [body_label, exit_label], "args" : [cond]})

        instrs.append({"label" : body_label})
        for stmt in self.block:
            instrs += stmt.get_instrs(scope)

        instrs.append({"op" : "jmp", "labels" : [entry_label]})
        instrs.append({"label" : exit_label})
        return instrs
```

4 Conclusion

There are a few useful features currently missing, the biggest being lists. Without lists, the number of useful algorithms one can implement is limited mostly to number-theoretic algorithms, as you will see in the implemented test cases. Assuming lists were implemented, algorithms involving sorting, linear algebra, and graphs will immediately be in grasp. Furthermore, there is relatively little work to be done to implement statically sized lists, as the lexer, parser and syntax tree code has been purposefully implemented to be easily extensible. Perhaps ironically, I believe the most useful aspect of this compiler is that it is completely unoptimized. All intermediate expressions are assigned to new virtual registers, including seemingly trivial assignments like $v_0 \leftarrow a$. This provides useful test cases for implementations of trivial dead code elimination, local value numbering, and constant folding/propagation.

References

[1] A. Sampson, "Bril intermediate representation," https://github.com/sampsyo/bril, accessed: 2024-12-17.

[2] R. Padhye, K. Sen, and P. N. Hilfinger, "Chocopy: a programming language for compilers courses," in *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, ser. SPLASH-E 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 41–45. [Online]. Available: https://doi.org/10.1145/3358711.3361627

Figure 5: Prime number example (primes.py)

```
# Get the nth prime starting from 2

def get_prime(n: int) -> int:
    candidate: int = 2
    found: int = 0
    while True:
        if is_prime(candidate):
            found = found + 1
            if found == n:
                return candidate
        candidate = candidate + 1
    return 0 # should never happen
```

Figure 6: Prime number example (primes.bril)

```
@get_prime(n: int): int {
  candidate: int = const 2;
  found: int = const 0;
.entry.1:
  v1: bool = const true;
  br v1 .body.1 .exit.1;
.body.1:
  v2: int = id candidate;
  v3: bool = call @is_prime v2;
  br v3 .then.2 .else.2;
.then.2:
  v4: int = id found;
  v5: int = const 1;
  v6: int = add v4 v5;
  found: int = id v6;
  v7: int = id found;
  v8: int = id n;
  v9: bool = eq v7 v8;
  br v9 .then.3 .else.3;
.then.3:
  v10: int = id candidate;
  ret v10;
  jmp .endif.3;
.else.3:
.endif.3:
  jmp .endif.2;
.else.2:
.endif.2:
  v11: int = id candidate;
  v12: int = const 1;
  v13: int = add v11 v12;
  candidate: int = id v13;
  jmp .entry.1;
.exit.1:
  v14: int = const 0;
  ret v14;
}
```