Translator Design - Week 1

1 Introduction

Over the course of these labs, you will be extending the Imp language into a usable, imperative, interpreted programming language. The source code with the bare-bones interpreter you will be working with is at:

`https://github.com/nandor/utcn-imp`

Fork or clone the repository using `git`, ensuring you can keep track of changes you make. After each lab, you are expected to submit a text or PDF file with the written answers, as well as a *diff* (`git diff -w` or `git show`), highlighting all changes relative to the previous state of the project. Please ensure that the *diff* is human-readable, ignoring white space and excluding binary files.

In case you find bugs and issues with the code, please file a bug report or submit a pull request to the GitHub repository. Note that many features are missing by design, left to be implemented by the reader. If you notice some questions or bits of documentation that are ambiguous or lack clarity, do notify the author.

1.1 Grading

Each week, you will be assigned a series of theoretical questions and practical exercises. Some of the exercises will be discussed during the lab hours and a full submission is expected within exactly a week.

- Up to 40% of the available marks will be given to the written answers to the questions, split between problems weighted by their difficulty.
- Up to an additional 40% will be assigned to a correct C++ implementation of the practical exercises. Each submission *must* include at least one test case in the Imp language covering all the newly added features to the language. If the test case is absent, no marks will be awarded for this section.
- Finally, the last 20% can be obtained by implementing any additional language features not explicitly mentioned in the exercises. For example, an alternative control flow construct (for, do-while) or arithmetic operator (modulo). The extension is to be briefly documented and followed by a test.

2 The Imp Language

In principle, Imp is an imperative, interpreted and statically-typed programming language. As such, the design of the language itself is intertwined with its compiler, the bytecode it translates to and the interpreter executing it. In its current form, only a very small number of features are implemented, enough to showcase the compilation pipeline and provide a platform for the remainder of the language constructs to be defined. This section briefly documents the language, although the source code implementing the compiler should also be consulted prior to attempting the exercises.

2.1 Syntax

Syntactically, Imp aims to be simple and easy to parse with a hand-written parser, sharing vague similarities with Go. A program consists of a series of function definitions and top-level statements, all executed upon the initialisation of the program in the order they are present in the source file. Currently, the language can

be used to define and call functions computing simple arithmetic expressions. Most notably, local and global variable definitions are missing, to be defined later on.

A sample program is illustrated below. The first two statements specify the prototypes of two external functions: these methods are not defined in the language itself, instead they point to special named functions provided by the runtime and implemented in `runtime.cc`. These two specific methods are helpers performing I/O operations, reading and writing integers. Such primitives are used quite often in practice, including in languages such as OCaml and Haskell. The `test` function is defined in Imp, introduced through the FUNC keyword. Between parentheses, the names of the arguments are explicitly specified, separated by a colon. After the argument list, a single return type is provided - currently, a type must be specified for all functions. Afterwards, between braces, a series of statements specify the behaviour of the function, in this instance simply returning the sum of the arguments. The last line of the program is an statement consisting of a single expression, calling the methods defined earlier in order to print the sum of two user-provided arguments. The parser and the code generator also allow `while` loops to be defined, although without support for local mutable variables such constructs are not particularly useful.

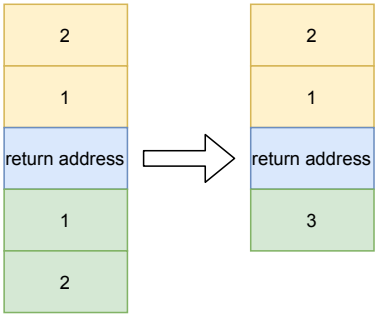
```
func print_int(a: int): int = "print_int"
func read_int(): int = "read_int"

func test(a: int, b: int): int {
  return a + b
}

print_int(test(read_int(), read_int()))
```

2.2 Bytecode

The Imp programs are mapped to bytecode instructions by the code generator. As shown in Figure 1a, the instructions consist of an opcode identifying the operation (`PEEK`), as well as an optional list of arguments. The instructions operate on a virtual stack managed by the interpreter, pushing and popping values. The instruction to be executed is identified through a program counter, which is either automatically incremented to fetch the next instruction and its operands or altered through jump instructions implementing control flow. For example, the operation of the `ADD` instruction is illustrated in Figure 1b. At the point of its execution, the `PEEK` instructions will have already fetched both arguments and placed them on top of the stack, to be both popped by `ADD` which also pushes the result in their place. On exit from the function, the RET instruction accesses both the return value and the return address, removing the latter from the stack and transferring control to that location in the program, leaving the value on top.

```
test: PEEK 1
      PEEK 3
      ADD
      RET
```

(a) Bytecode of `test`



(b) Stack at the execution of Add

Figure 1: Compiling `test` to bytecode

Stack-based machines are preferred for interpreters as they are easy targets for code generation. Unlike register-based machines, which are ubiquitous among hardware targets, stack machines do not require an

expensive register allocation step, allowing bytecode to be directly generated from the AST. Naturally, simplicity comes at the cost of performance, as the execution of each instruction requires multiple memory accesses to read its operands from the stack. Presently, the interpreter supports a small set of instructions, to be later extended through the exercises:

PUSH_FUNC f**:** Pushes the address of f on top of the stack.

PUSH_PROTO f**:** Pushes the address of the runtime method f.

PEEK n**:** Reads the value n elements from the top (0 is top) and pushes it onto the stack. PEEK 0 is often called DUP in other interpreters, as it duplicates the value on top.

POP: Discards the value from the top of the stack.

CALL: Pops a value, which should be the address of a function, transferring control to it. The location after the call opcode is pushed onto the stack, marking the address where return should jump back.

JUMP *addr***:** Continues execution at *addr*.

JUMP_FALSE *addr***:** Pops a value from the stack. Jumps to *addr* if it is zero.

STOP: Stops the execution of the program.

ADD: Pops two values from the stack and pushes their sum.

2.3 Compiler

The IMP compiler consists of a lexer splitting the stream into a series of tokens, a parser constructing the Abstract Syntax Tree (AST) from the tokens, followed by the code generator mapping the AST to bytecode. Here the compiler is described briefly, more information is available among the sources of the project. To illustrate its operation, consider the following code fragment:

```
while (read_int()) {
  print_int(read_int() + read_int())
}
```

2.3.1 Lexer

The lexer traverses the sources character-by-character, splitting it into a series of tokens while also keeping track of the location of the tokens (line number, character number) in the source file. Implemented in `lexer.h` and `lexer.cpp`, it primarily exposes the `Next` method, which advances the stream and returns the next parsed token. The tokens themselves can either be standalone characters (such as parantheses or operators) or can carry additional information (as is the case with strings and named identifiers). Lexical analysis skips both whitespace and comments, as they carry no information relevant to building the AST. The prior example would be tokenised as follows:

```
WHILE, LPAREN, IDENT("read_int"), LPAREN, RPAREN, RPAREN, LBRACE,
IDENT("print_int"), LPAREN, IDENT("read_int"), LPAREN, RPAREN,
PLUS, IDENT("read_int"), LPAREN, RPAREN, RBRACE
```

2.3.2 Parser

The parser, implemented in `parser.h` and `parser.cpp` is a handwritten recursive-descent parser, inspecting one token at a time and constructing the appropriate AST nodes representing the input program. The parser relies on a single look-ahead token: the stream is advanced through calls to the `Next` method of the lexer, after which the kind of the returned token is inspected in order to identify the node that should be constructed, invoking the appropriate method validating and building it. Additionally, the parser is

responsible for ensuring the syntax is valid, identifying errors and raising a descriptive `ParserError` to stop the compiler. Note that this compiler does not explicitly construct a parse tree: the stream of tokens is converted directly into AST nodes.

while (<cond>) <stmt>

On the prior example, the goal of the parser is to construct a node representing the `while` loop. Roughly, this node requires the presence of a keyword (`while`) and a nested expression between parenthesis specifying the loop condition. The loop body is a simple statement, optionally packed into a block. The fragment of code below shows the implementation of the method parsing while loops. The code first uses the `Check` method to ensure that the current token is indeed the `while` keyword, after which the stream is advanced asserting that the subsequent one is a parenthesis. The stream is advanced again to fetch a look-ahead token, recursively parsing an expression with the appropriate method. Since the expression parser moves the stream one token past the expression, on return the code ensures that it is followed by the appropriate closing parenthesis. Finally, the statement itself is parsed, building the node with the information gathered.

```
std::shared_ptr<WhileStmt> Parser::ParseWhileStmt()
{
  Check(Token::Kind::WHILE);
  Expect(Token::Kind::LPAREN);
  lexer_.Next();
  auto cond = ParseExpr();
  Check(Token::Kind::RPAREN);
  lexer_.Next();
  auto stmt = ParseStmt();
  return std::make_shared<WhileStmt>(cond, stmt);
}
```

2.3.3 Codegen

The code generator receives the AST from the parser and must generate a sequence of instructions specifying the behaviour of the program. In contrast with the tree structure of the AST, the opcodes are flattened and laid out sequentially in memory. Control flow is implemented using labels and conditional jumps: labels are created to identify specific points in the program, which are used as operands to instructions which transfer control to them. The methods lowering the nodes rely on a set of helpers (`Emit*`) to encode instructions.

```
void Codegen::LowerWhileStmt(const Scope &scope, const WhileStmt &whileStmt)
{
  auto entry = MakeLabel();
  auto exit = MakeLabel();

  EmitLabel(entry);
  LowerExpr(scope, whileStmt.GetCond());
  EmitJumpFalse(exit);
  LowerStmt(scope, whileStmt.GetStmt());
  EmitJump(entry);
  EmitLabel(exit);
}
```

The function above translates a while loop to bytecode. First, two labels are created: one to identify the entry point and one to point to the exit. The `EmitLabel` call pins the label to the address past the last emitted instruction, allowing other instructions to reference this location later on. The condition is then lowered, followed by a conditional jump which exits the loop when the check fails. At this point, the address of the exit label is unknown, as it has not yet been emitted: to account for forward references, the location

of the operand is recorded as a fixup to be adjusted later on, once the label is known. Following the jump, the body is generated and closed with a backwards jump back to the entry and to the next iteration of the condition check. This is a backwards jump to a known label, allowing the correct address to be emitted straight away without the need for a fixup. The sequence of bytes encoding the instructions is packed into a `Program` defined in `program.h`, to be passed on to the interpreter for execution.

3 Exercises

3.1 Language

1. Add another primitive to the interpreter, `print_newline`. Write a test script that reads two integers and prints their sum properly followed by the newline character.
2. Would a `print_char` function be better? Discuss what else should be added to the language for such a function to be useful.
3. Name and discuss in a few sentences each at least 5 different missing constructs or language features.

3.2 Lexer

1. Identify one performance issue in the lexer. Discuss in a few sentences how you would improve it.
2. Extend the lexer to accept operators for multiplication, division and subtraction.
3. Add the commonly used comparison operators to the language.
4. Modify the `Token` class to represent integers and extend the lexer to tokenise them. Argue in a few sentences why there is no need to represent negative numbers at this point. Define an arithmetic operator which can help represent negative numbers. Identify an edge case. How would you handle it?

3.3 Parser

1. Define the AST nodes to represent the operators and constants added to the lexer.
2. What is the priority of the operators defined so far? Are they left or right associative?
3. Extend the parser with syntax to allow users to control the priority of operators. Is there a need for a separate AST node?
4. Extend the parser to accept the newly added arithmetic and comparison operators, as well as the integral constants.

3.4 Interpreter & Code Generator

1. Identify, describe and fix the bugs in the PEEK and ADD opcodes.
2. Define an opcode to push a constant integer to the stack.
3. Define the opcodes for the newly added operators. Do they need any additional arguments?
4. Translate the new AST nodes to bytecode. Do not forget to add tests.