Compiler Construction - Supervision 1

Nandor Licker

October 2019

1 Past Papers

- 2001 Paper 4 Question 4
- 2007 Paper 4 Question 4

2 Practical

You are given a basic compiler for a very simple language, which supports integers and closures. Start by forking the following repository: `https://github.com/nandor/mini-lambda`. If you want me to comment on your code, please point me to your fork on GitHub!

2.1 Getting Started

Before implementing any features, ensure the compiler builds and runs correctly. If you encounter any issues, please contact me as soon as possible - or file an issue on GitHub.

- Install dependencies, starting with *opam2*: `https://opam.ocaml.org/doc/Install.html`
- Install *dune*, the build system: `opam install dune`
- Install *menhir*, the parser generator: `opam install menhir`
- Build the project: `dune build main.exe`
- Compile a file: `./lambda tests/full.lambda`
- Read the comments inside the source files in the project

The `lambda` script is written in Bash - it invokes the compiler to generate an assembly source from an input written in the lambda language. The assembly file is compiled with the system assembler (usually *gas* on Linux, `clang` on macOS) and linked with the runtime, which provides some builtin methods performing I/O.

3 The Language

Presently, lambda supports functions, closures, integers and the addition operator. Functions which have no body, such as `print_int` in the example, are external methods implemented in assembly language by the runtime. Some examples can be found in the `tests` directory inside the project. Currently, there is no explicit syntax for type annotations: all types are inferred using the Hindley-Milner algorithm. The language is type-safe, with the exception of external methods (which you will fix!).

Exercise: Study the type checker in `typing.ml` and explain the similarities between the grouping of strongly connected components and `let rec ... and` in OCaml. Name an advantage and a disadvantage.

The compiler currently supports two targets: *ARMv7* and *x86_64*. The compiled code is not particularly efficient - the instructions emulate a stack machine. On *x86_64*, the target which you will probably be using, the stack machine is implemented using the following registers:

- %rax stores the environment pointer of closures
- %rbp is the base pointer on the stack. The value located at the location pointed to by %rbp is the base pointer of the previous frame - this chain can be used to traverse all the stack frames. The value on the stack on top of the base pointer is the return address. Arguments follow: the ith argument is located at %rbp + 16 + 8 × i. Local variables are stored on the stack below %rbp: they can be accessed at %rbp - (i + 8) × 8.

  Exercise Your first task is to draw a diagram of the stack layout , after studying the code generator.
- %rsp is the stack pointer: points to the bottom of the stack.
- %rcx and %rdx store temporary values for various computations. %rcx also stores the return value from a function call.

4 The Compiler

Compilation proceeds through the following stages, implemented in the files mentioned below:

- lexer.mll: Lexical analysis
- parsing.mly: The parser constructs the AST defined in ast.ml
- typing.ml: Implements type checking and produces a typed AST, defined in typed_ast.ml
- ir_lowering.ml: Generates the stack-based IR defined in ir.ml
- backend_*.ml: Lowers the IR to machine code

5 The Task

Your task is to build up the features required to add conditional and looping constructs.

5.1 Subtraction

Due to the finite number of hours available in a day, the language seems to be lacking a subtraction operator. The first exercise involves implementing it, following the steps below:

1. Define a token in the parser, name it MINUS
2. Provide a rule in the lexer to consume the minus sign and generate the correct token
3. Define an AST node for subtraction
4. Add a rule to the parser to construct that AST node, similarly to the addition operator. If at any point the OCaml compiler complains about unimplemented patterns in match statements, stub them out with failwith
5. Add a type checking rule - should be a straightforward adaptation of the addition rule
6. Add an instruction to the IR in ir.ml to execute subtraction
7. Lower subtraction to the IR
8. Generate code - the instruction on x86_64 is subq. Stub out the ARM version or implement it in case you have access to a board you can test on.

The previous steps highlight all the changes required to the compiler - make sure you get the hang of it as the following exercises will be less detailed.

5.2 Booleans

It is a good idea to be able to use boolean values as conditions to `if` statements and while loops. Add the following constants and operators to the language:

- `true`
- `false`
- `==`
- `!=`
- `&&`
- `||`

This time you will also need to add a new type, alongside `TyInt`. Unification should stay unchanged.

5.3 Assignment

What does the following bit of code achieve? How many copies of x will there b?

```
x <- 1;
x <- 2;
```

This is not particularly useful in an imperative language. Modify the compiler such that the second 'bind' to x rewrites the initial value instead of creating a new local.

5.4 If Statements

The language should support if statements with optional else branches. Implement them. In the IR and assembly, you will need at least the following 3 constructs:

- Labels: to denote a point in the program, used as target for jumps.
- Unconditional jumps: `jmp` in assembly.
- Conditional jumps: All jumps should take a boolean argument. Assuming one is on top of the stack prior to the jump, it can be popped into a register. Afterwards, `tstq %rdx, %rdx` can be used (with almost any register!) to check if the value is zero, followed by `jz` or `jnz`.

6 Additional Questions

1. Consider the following example from some awful programming language which shall remain unnamed:

```
function test(a, b, c) {
  return a + b / c.length - 123.0;
}
function some_call(c, d) {
  z = c + c;
  return test(3, z, "\"hello" + d);
}
some_call(5, "world\"");
```

   (a) Enumerate all the tokens, grouping them by their type.

   (b) Draw the automatons which accept the following tokens: numbers, identifiers and strings.

(c) Provide regular expressions which match numbers, identifiers and strings.

(d) Constructs the NFAs of the previous regular expressions and reduce them to DFAs.

(e) Draw the automaton of the entire lexer, with states to accept all tokens of the language.

(f) Implement the lexer in `ocamllex`.

2. In some languages, floating point numbers can be defined in multiple ways. The following are examples of valid numeric constants in C++11:

   - `0b101`
   - `0x123`
   - `0233`
   - `1234`
   - `-123.5`
   - `123.5f`
   - `1.3e-20`
   - `2.3e+10`
   - `1e1f`

   (a) Provide a regular expression for the following: binary, decimal, hexadecimal, float and double.

   (b) Draw an automaton with accepting states for the following tokens, capable of parsing numbers similar to the ones shown before, with 3 accepting states: `INT`, `FLOAT`, `DOUBLE`.

   (c) Can you design a regex to only accept 32-bit signed integers? Sketch a solution. How about 32-bit floats? Explain why this might not be practical. How would you handle this in `ocamllex`?

   (d) At what stage in the compiler should an error be emitted when a numeric constant is too large? Discuss the following languages: C/C++, Java, OCaml and JavaScript.

3. This question concerns parsing and representations.

   (a) Define the Abstract Syntax Tree (AST) of the awful language mentioned before in OCaml, using records and variants. What are the options of representing function bodies and statements? Make sure the AST can represent first-class functions and closures.

   (b) Draw the parse tree and the AST of the test function:

   ```
   function test(a, b, c) {
     return a + b / c.length - 123.0;
   }
   ```

   (c) Provide the EBNF grammar for a parser.

   (d) Sketch the code of a recursive descent parser for this grammar.

4. The awful language mentioned before also supports regular expressions with the following syntax:

   ```
   function is_aaa(str) {
     return /aaa/.match(str);
   }
   ```

   (a) Write a function which contains the string `/aaa/`, but does not contain any regexes.

   (b) Can you tokenize the stream by simply adding a rule to the previous lexer? Explain why not.

   (c) Show how the recursive descent parser would interact with a modified lexer to parse this language.

5. Provide the small-step semantics of the language, with eager evaluation, passing arguments from left-to-right. Discuss any other assumptions. Show how the following example reduces to a single value:

```
function f(x) {
  return function(z) {
    return x + z;
  };
};
f(2)(3)
```