Abstract

Compilers is a practical course. Its goal is to build a real compiler, which compiles a high-level language down to the actual x86-64 machine code and produces an executable that runs on student’s laptops. The source language is Tiger*: a procedural language in the spirit of Pascal – or C with arbitrarily nested functions. The compiler itself is to be developed in OCaml.

The characteristic of the course is an iterative, incremental development: we start with the most trivial source language, develop the full compiler for it, and then keep extending the source language and the compiler in small steps, reusing the earlier work as much as possible. At each iteration, we build the complete, end-to-end compiler producing runnable and testable executables, for a (progressively larger) subset of the source language.

Another characteristic is the extensive use of tagless-final style, taking the full advantage of extensibility afforded by it. Extensibility here means reuse – of type-checked and compiled artifacts from the previous increment – rather than copy-paste. The compiler is hence structured as a stack of domain-specific languages, with parsing at the bottom and assembly at the top. The languages are extended by adding new operations here and there (and only occasionally by redirection).

We cover all standard material for the compiler course, from parsing and type-checking to analyses, optimizations, calling conventions and assembly generation – but in a quite non-traditional fashion.

Contents

1 Introduction ............................................................. 2
  1.1 Prerequisites ....................................................... 2
  1.2 Development environment ..................................... 3
    1.2.1 Operating system and environment ..................... 3
    1.2.2 OCaml .......................................................... 3
    1.2.3 Git ............................................................... 4
  1.3 Grading .............................................................. 4

2 What is a compiler? .................................................. 5
  2.1 Incremental approach ........................................... 8
3 Introduction to tagless-final style and OCaml reminder

4 Making the compiler
   4.1 The simplest source language ........................................ 13
   4.2 Unary expressions ...................................................... 17
      4.2.1 Booleans ............................................................ 19
   4.3 Challenge of binary operations ...................................... 21
      4.3.1 Migrating to ocamllex/ocamlyacc ............................. 24
   4.4 Local variables .......................................................... 26
   4.5 Full expression language ............................................ 30
   4.6 Dummy expression and sequencing ................................... 31
   4.7 Mutable variables, assignment, function calls ................... 32
   4.8 Strings and string operations ...................................... 34
   4.9 Comparisons and conditionals ...................................... 35
   4.10 Loops ................................................................. 37
   4.11 Function declarations, local functions and procedures ........ 38
      4.11.1 Type-checking .................................................. 38
      4.11.2 Complexity of local functions ............................... 41
   4.12 Optimizations .......................................................... 43
   4.13 Going Further .......................................................... 44
   4.14 Final exercises ........................................................ 44

5 Conclusions and Reflections ........................................... 45
   5.1 The place of Tiger’ in the current landscape ....................... 46
   5.2 Compilers are more fun than you may think ........................ 46

1 Introduction

1.1 Prerequisites
   + Familiarity with OCaml or the closely related F#
   + Light familiarity with C: We make the compiled code compatible with
     C so we may write initialization and support code in C and reuse the C
     standard library
   + Basic data structures (tuples, variants, lists) and algorithms on them
     – No requirement to know beforehand anything about the x86-64 assembly
       language beyond the knowledge of the computer organization
     – No experience is assumed with parsing, type-checking, code-generation,
       etc.
1.2 Development environment

This is a practical course. Besides building a complete, realistic compiler, one of its goals is to give a taste of modern software development: test-driven development; pervasive use of version control; stress on reading, comprehending and extending code rather than writing from scratch; build pipelines, explicitly spelled-out requirements; etc. Thus, the first task is to set up and test the development environment.

1.2.1 Operating system and environment

Windows Install WSL2. After installing WSL2, you obtain the Unix/Linux environment. Therefore, when installing OCaml later, use that environment and follow the directions for ‘Unix installation’ (rather than for ‘Windows installation’). Confirm that you have gcc (at least v11), git and GNU make installed. (If they are somehow missing, install them. You won’t be able to install OCaml without gcc anyway.)

MacOS Install XCode, then Homebrew, and then make and git. Confirm the version of make (should be GNU make). As an alternative to installing Homebrew, one can use a Virtual Machine running Linux (the particular distribution does not matter).

You may use any editor/IDE you like. If you have not used any programming editor yet, you may want to try VS Code. It is a modern editor with a good support for OCaml, among other languages.

1.2.2 OCaml

OCaml is the language we use to write the Tiger’ compiler. OCaml is actually very good for writing compilers: the first Rust compiler was written in OCaml; the reference Wasm implementation is in OCaml; Meta’s Hack language is developed in OCaml. The quite well-known theorem prover Coq is written in OCaml (and its close relative, Isabelle/HOL is written in Standard ML: OCaml’s close relative).

installation https://ocaml.org/docs/install.html, or https://pl.cs.jhu.edu/fpse/coding.html (you do not need to install any extra OPAM packages listed on that page, although ocaml-lsp-server, merlin and utop can be very useful.)

If you are working on Windows and installed WSL2 as recommended, open the WSL2 terminal and install OCaml as for Unix/Linux (not Windows!)

Install OCaml version at least 4.14.1

For build, we use Makefiles at the beginning, but then switch to a custom build pipeline.

1.2.3 Git

Git is the de facto standard of software development. We will be using it extensively in this course. First, install git and learn its basics: at the very least, learn how to use `git add`, `git pull`, `git commit --a`, `git push`. Also useful are `git diff` and `git log --p --n`. (I will not use branches, submodules and more advanced features.)

Second,

- make an account for yourself at https://bitbucket.org/ (Also register SSH keys: see the documentation on the site; otherwise, you have to create an APP password.)
- make a private repository for yourself (include your name or student ID in the repository name). This will be your development repository for your compiler
- share that repository with me: e-mail address: oleg@okmij.org
  Give me the write access to your repo.

Once I receive your invitation, I will share with you the class repo, which contains the code for the class and these notes. Both will be extended as the class progresses. Therefore, you may want to ‘watch’ that repository (that is, get notified on updates by e-mail): set-up via bitbucket.

When you receive the invitation to join the class repo, reply to it, clone the class repo and copy its directories (and also files `.gitignore` and `Makefile.common`) into your repo. To test the setup, go to the `scratch` directory and enter `make` there (on a Mac, enter `make mac`), which will try to build the code in §2. If the `make` finished successfully and the built program `sample` works, your set-up is done.

1.3 Grading

This is a practical course. Each week there are 1-3 homework assignments. They are graded on the scale 0–10, as a rule. Bonus points may be given for particularly clever or impressive solutions. The course grade is determined from the points you earn from these assignments. There is no final exam.

Submission deadline  Homework assigned on week $N$ must be submitted by the start of the $N + 1$-week class.
Submission guideline Each assignment will ask you to develop some code or tests. All the development has to be done in your private bitbucket repository that you have shared with me. I also share with you the class repository, which contains the code covered in the class. Many assignments ask you to improve that code.

As the first step, copy that code in your repo and immediately commit it. Then start improving, as described in the assignment.

Overall, your answer to a homework should be in files or directories with the specified names, committed into your bitbucket repository.

At the deadline, I clone your repo, compile your code and run it on my tests. **If the code fails to compile or fails the tests, you get 0 points.**

The grade and the comments, if any, are reported in the file grade.txt that I commit to your private repository, in the same directory as the submitted homework.

**Important** *Read the assignment closely: at least two times.* If the submission does not satisfy the assignment, you get 0 points. “But your know what I mean” excuses are not accepted. This is a course about programming. Programming is talking with a computer. A computer does not know what you mean: It only knows what you entered. Even a one letter mis-spell is fatal. (One-letter mistakes may create big problems also in real life: if you buy a plane ticket online and misspell your name, even by one latter, you will be denied boarding the plane.)

*Test before submission:* submit only tested code. If the submitted code fails to compile, you get 0 points. You also get 0 points if the submitted code fails tests: ends up in an exception or infinite loop although given valid input.

## 2 What is a compiler?

First of all, what is a computer? I hope I don’t need to elaborate: everybody knows. Everybody also knows that a computer has the CPU to execute programs, memory to store programs and data, and some sort of IO devices. Programs are sequences of instructions for the CPU. What are they?

Here is an example.

```
48 83ec28e8
 00000000 48894424 0848c744 24180000
000048c7 44241001 000000eb 18e80000
00004889 0424488b 04244801 44241848
83442410 01488b44 2410483b 4424087e
dc488b44 24184889 c7e80000 00004883
c428c3
```

The numbers you see are the instructions, for the modern Intel/AMD CPU, called x86-64 architecture. (See [CS107(2021b), CS107(2021a)] for introduction
and [Cloutier (Ed.)(2022)] for complete reference.) It is currently the most widely used architecture for desktop and laptop computers.¹

The numbers are hexadecimal numbers (do you know what are they?) and represent instructions. Can anyone tell what the instructions are and what does this program do?

For example, the number 4889C7 here instructs the CPU take data from the register `rax` and put into the register `rdi`. The next instruction, starting with `e8`, is the function call. We will talk about these instructions in more detail in §4.1. In old times, people indeed programmed computers by manually entering these numbers into memory, using switches – as was the case for Altair, the first commercial Personal Computer in the world (Altair 8800 debuted in 1974).

I vouch for that: I myself programmed that way, for a different computer, when I was a student. I could look at such list of hexadecimal numbers and see the program and understand what it does. It is not as difficult when you learn and get used to it. Still, there are lots of bothersome things, like offsets in jump instructions and figuring out the target of a jump. (see EB18 at the 3d line near the end; there 18 is the offset).

To help with such tedious tasks, and also to make the program more readable, assembly language was invented. Here is the same program in assembly.²

```assembly
.globl ti_main
.type ti_main, @function
ti_main:
    subq $40, %rsp
    call read_int
    movq %rax, 8(%rsp)
```

¹If you own a recent Mac, you are using a different, ARM architecture. Still, it has the x86-64 emulation mode.

²The listing uses the so-called AT&T notation (also called GAS notation) common on Unix (including MacOS) and Linux. There is another, nearly opposite, x86-64 assembly notation called Intel or MASM. It is typically used on Windows and in Intel documentation. In this class we stick to the GAS notation.
It is more readable, isn’t it? An assembler is a program that translates code in this notation to the numbers we have seen earlier. The translation is straightforward: using the dictionary that relates a string such as `movq %rax, %rdi` to the corresponding number, `4889C7` in this case. Jumps like `jmp` and `jle` interrupt the sequential, instruction-after-instruction execution and transfer control to some other place in the instruction sequence. In assembly, the target of a jump in assembly is denoted by a label. The corresponding instruction needs a distance (offset), which the assembler also computes. This is very welcome, since it is very tedious to do by hand (I did it, and I still remember the tediousness).

Still, this assembly code, although quite more readable than numbers, is rather difficult to comprehend. Anyone can tell what the program does? It is also difficult to write such assembly code, because it is so low-level. One have to think of so many details: which registers to use and when to reuse, what register or stack location like `24(%rsp)` means what, figure out how much space for temporary data the program needs and reserve it at the beginning and free at the end (see `subq` and `addq` instructions, etc.)

And so were invented higher-level languages, to make programs easier to comprehend and to write. The first higher-level language was FORTRAN, which means FORMula Translator. (FORTRAN was the first programming language I learned, in high school in late 1970s). The idea was to write formulas in a conventional math notation. Many, many more programming languages were developed since FORTRAN. In this class, we will be dealing with a language called Tiger*: a dialect of Tiger developed in [Appel(1998)]. It is a high-level procedural language in the spirit of Pascal or C. You can think of it as C with nested functions, and with keywords instead of curly braces. Our running example looks in Tiger* as

```
let
```

val n := read_int()
var sum := 0
in
  for i:=1 to n do
    let val v := read_int()
    in sum := sum + v end
done;
print_int(sum)
end

Has it become easier to understand? Anyone can tell what the program does?

We also need a program to translate such easier to understand code to the assembly. This program is the compiler. The overall flow is hence as follows.

\[
\text{Tiger' \xrightarrow{\text{compiler}} \text{Assembly} \xrightarrow{\text{linker}} \text{Object code} \xrightarrow{\text{assembler}} \text{Executable}}
\]

The linker, not mentioned earlier, is needed to find and pull in the code for library functions (like `read_int`) and put their address into the corresponding call instructions. The complete executable also needs start-up code, which the linker also arranges for.

Creating a compiler from Tiger’ to x86-64 is the goal of this class. We shall indeed compile the above Tiger’ code and obtain the assembly, very similar to the one shown before, with identical functionality.

### 2.1 Incremental approach

The characteristic of this course is incremental development, in many small steps. As pointed out by Ghuloum in [Ghuloum(2006)], traditional compiler courses teach a compiler one pass at a time; “many of the issues that a compiler writer has to be aware of are solved beforehand and only the final solution is presented. The reader is not engaged in the process of developing the compiler.” There is too much focus on individual passes and not enough focus on the “big picture”.

Like [Ghuloum(2006)], this course is different. Our development is by extending the complete, working compiler one small step at a time. At each step we end up with the working compiler, for a subset of the source language. Specifically, the methodology ([Ghuloum(2006), §2.6]):

1. choose a small subset of the source language that is easy to directly compile to assembly
2. Write the extensive test cases
3. Write a compiler for the chosen subset to the assembly language. Perhaps optimize
4. Run all the tests
5. Base on the experience, refactor. Make sure the tests still pass

6. Enlarge the subset of the source language and extend the compiler correspondingly, refactoring as needed.

7. Repeat from 2

In contrast to [Ghuloum(2006)], we rely on the tagless-final approach [Kiselyov(2022)], which makes extensibility easy. We hence use the motivation of Ghuloum, but apply it diametrically differently. (Our source language is also different, Tiger' rather than Scheme.)

3 Introduction to tagless-final style and OCaml reminder

OCaml is the language used in this course to write the compiler and keep extending it. OCaml is similar to F# with which you should be familiar from earlier classes. This section is a brief reminder, stressing the module system (quite more powerful, compared to F#), which we will be using modules extensively. This section also introduces the so-called tagless-final style [Kiselyov(2022)] of embedding domain-specific languages (DSL) and writing their interpreters and transformers. The characteristic of the tagless-final style is extensibility: the ability to add features to the DSL one-by-one at a later stage, reusing the already written code. As we shall later see, our Tiger’ compiler is one of the interpreters of the Tiger’s DSL. The extensibility of the interpreter is particularly valuable in the incremental approach.

We start with the simplest DSL (whose extension will be the homework assignment). Let’s call it \texttt{Lang}. It has only integer literals and the addition operation. Here are a few sentences, or expressions, of \texttt{Lang}: each on a separate line in a column.

\begin{align*}
1 & \quad 0 \\
-1 & \quad (4 + 0) \\
((4 + 0) + -1) & \quad (-1 + -1) \\
((4 + 0) + (-1 + -1)) & \quad (((4 + 0) + -1) + -1)
\end{align*}

In other words: (i) an integer is an expression; (ii) connecting two existing expression with the plus sign (and putting parentheses around) makes a new expression. With fewer words, \texttt{Lang}’s language definition can be stated in the form of a context-free grammar:

\begin{align*}
S \rightarrow & \text{ integer} \\
S \rightarrow & \text{ (S + S)}
\end{align*}

Let’s embed \texttt{Lang} in OCaml: that is, represent its sentences in the form of OCaml expressions. OCaml is a functional language, so the fundamental operation is application. It seems appropriate then to represent \texttt{Lang}’s sentences as OCaml applications. Assume a function \texttt{int}. Then the application \texttt{int 1} can

\footnote{Actually, F# started as a dialect of OCaml, but later diverged.}
be used to represent the sentence 1 of Lang. Assume a two-argument function add. Then the application

\[
\text{add (add (int 4) (int 0)) (add (int (−1)) (int (−1)))}
\]

could represent the sentence \((4 + 0) + (−1 + −1)\), which is our running example.

If we just enter the above expression onto Ocaml’s top level, we immediately get an error that add is not define. Just saying to ourselves ‘assume add exists’ is not enough: we have to say it to OCaml. Let’s first think of the types of int and add. Lang’s sentences are represented as OCaml’s applicative expressions. All expressions in OCaml (that is, the ones accepted by OCaml (compiler)) have a type. OCaml expressions representing Lang hence must also have some type. At this point, of describing a language, we do not actually care what it is. Therefore, we make it abstract, call repr. The functions int and add then have the types

\[
\text{int: int → repr} \\
\text{add: repr → repr → repr}
\]

To formally declare to OCaml that int and add are assumptions, we make them function arguments. After all, a function is an implication: given the value of its arguments it produces the value of its body. Hence in full, a sample Lang sentence is represented by the following OCaml expression:

\[
\text{fun \((\text{type repr})\ (\text{int:int→repr})\ (\text{add:repr→repr→repr}) \to} \\
\text{add (add (int 4) (int 0)) (add (int (−1)) (int (−1)))}
\]

This is a tagless-final embedding of Lang – in a preliminary form at least.

The obvious drawback is the need to enumerate the constructor functions int and add all the time (more realistic languages have quite more constructors) and remember their argument order, which is not really important. It would also be useful to group int and add with their types, and to finally be able to attach the name Lang. OCaml’s module signatures is the facility to do exactly such grouping.

Here is Lang’s definition in the form of the module signature, which lists the language operations (sentence constructors) and their types (i.e., arity).

\[
\text{module type Lang = sig} \\
\text{type repr} \\
\text{val int: int → repr} \\
\text{val add: repr → repr → repr} \\
\text{end}
\]

The abstract type repr stands for some representation of language expressions. The type of int says that we can make DSL expressions from OCaml integers, like int 4. The type of add says that given two DSL expressions (remember, they are represented as values of the type repr), we can make a new DSL expression: their sum. Note how closely Lang matches the context-free grammar of the language shown earlier. One may say therefore that Lang defines the syntax of our DSL.
Using the signature, the sample \texttt{Lang} expression is written as

```ocaml
module Ex1(L:Lang) = struct
  open L
  let res = add (add (int 4) (int 0))
  (add (int (-1)) (int (-1)))
end
```

Here, \texttt{L} is the name of some implementation of the signature \texttt{Lang}: that is, some \texttt{Lang} interpreter. The expression \texttt{open L} brings the operations it defines – \texttt{int} and \texttt{add} – into scope, so that we may use them (without needing to attach \texttt{L}. prefix all the time).

To evaluate that expression we need an implementation of the \texttt{Lang} signature. Here is one:

```ocaml
module Eval = struct
  type repr = int
  let int x = x
  let add = (+)
end
```

It is an interpreter of our DSL, interpreting its expressions as familiar addition expressions over integers. Hence the representation type is \texttt{int}: the value of DSL expressions in \emph{this} interpretation. The \texttt{Eval} interpreter maps DSL operations directly to the corresponding OCaml operations: \texttt{Eval} is a so-called \textit{meta-circular} interpreter for the tiny subset of OCaml. To evaluate the sample \texttt{Ex1} we interpret it with the \texttt{Eval} interpreter:

```ocaml
let module M = Ex1(Eval) in M.res
```

which gives the OCaml value 2, the meaning of \texttt{Ex1} in \texttt{Eval} as an OCaml integer.

\texttt{Lang} hence may also be viewed as the signature of the DSL interpreters, which give a particular meaning to DSL expressions.\footnote{In the graduate school you may learn that a \texttt{Lang} implementation specifies a denotational semantics for our language: \texttt{repr} defines the domain, and \texttt{int} and \texttt{add} give the meaning to DSL integer literals and the addition in this domain. The denotation for complex expressions is determined compositionally, from the denotations of their sub-expressions.} \texttt{Eval} is not the only possible interpreter of \texttt{Lang}. We may also interpret DSL expressions as strings, so to display them. The meaning for an expression is hence its printed representation:

```ocaml
module Pp = struct
  type repr = string
  let int = string_of_int
  let add x y = "(" ^ x ^ " + " ^ y ^ ")"
end
```

Interpreting the same \texttt{Ex1} using \texttt{Pp}, as

```ocaml
let module M = Ex1(Pp) in M.res
```

now gives the string "((4 + 0) + (−1 + −1))".

Besides evaluating DSL expressions we may also want to transform them. A simple example is negation. This transformation can also be written as an
interpreter: after all, interpreting is the only thing we can do with tagless-final expressions.

```ocaml
module Neg(F:Lang) =
  struct
    type repr = F.repr
    let int x = F.int (-x)
    let add e1 e2 = F.add e1 e2
  end
```

Neg interprets the DSL in terms of another interpreter, \( F \) (from ‘From’). We used the fact that \((-x + y) = (-x) + (-y)\): addition means (for evaluation) the same thing in the original and the negated languages. On the surface of it, \( \text{Neg} \) is an interpreter transformer: it takes one implementation of \( \text{Lang} \) and produced another implementation. We may hence transform the earlier \( \text{Eval} \) and \( \text{Pp} \) implementations and use the result to interpret the same \( \text{Ex1} \), for example,

```ocaml
let module M = Ex1(Neg(Eval)) in M.res
let module M = Ex1(Neg(Pp)) in M.res
```

The result is easy to imagine.

Dually, \( \text{Neg} \) may also be regarded as an expression transformer. Here is the \( \text{Neg} \)-transformed \( \text{Ex1} \):

```ocaml
module Ex1Neg(F:Lang) = Ex1(Neg(F))
```

\( \text{Ex1Neg} \) has the same type as the original \( \text{Ex1} \): given an interpreter \( \text{Lang} \) it computes the meaning of \( \text{res} \) in that interpreter. That is, \( \text{Ex1Neg} \) is a tagless-final representation of a DSL expression – namely, the negated one. It can be interpreted with the existing \( \text{Eval} \) and \( \text{Pp} \) interpreters, or even the \( \text{Neg} \)-transformed interpreters:

```ocaml
let module M = Ex1Neg(Eval) in M.res
−2
let module M = Ex1Neg(Pp) in M.res
"((-4+.0)+.0+(1+.1))"
let module M = Ex1Neg(Neg(Eval)) in M.res
2
```

(The evaluation result is shown, indented, underneath each expression.)

We now 'modularize' our DSL development, arranging the DSL definition, interpreters, transformers, and the testing script each in a separate file. Look at the directory `tfintro` in the class repo. It has the typical organization for our projects. It always has the file `README.dr`, which describes the project and explains the other files there. There is also `Makefile`, which tells how to make the project.

Let’s look at `README.dr` and examine the files mentioned therein. Please pay particular attention to `ex1.ml`: it is not just an example of using the DSL. It also contains `assert` statements, that check the results match expectations. In case of mismatch, `assert` will crash the program. Therefore, `ex1.ml` is also a regression test: if we run it and it finishes normally, there is some confidence things work as expected. If it fails with an error, we have to investigate.
References


