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.
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.


**books** See Clarkson [2022], Whitington [2013]

**other resources** [https://batsov.com/articles/2022/08/29/ocaml-at-first-glance/](https://batsov.com/articles/2022/08/29/ocaml-at-first-glance/)

### 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/](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 [2021a,b] for introduction and Cloutier [Ed.] 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, staring 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.\(^2\)

```
.globl ti_main  
.type ti_main, @function  
ti_main:  
.subq $40, %rsp
```

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

\(^2\)The 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

```java
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.

```
Tiger’ Assembly Object code Executable
compiler assembler linker
```

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 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 \textit{Lang}. It has only integer literals and the addition operation. Here are a few sentences, or expressions, of \textit{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, \textit{Lang}’s language definition can be stated in the form of a context-free grammar:

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

Let’s embed \textit{Lang} in OCaml: that is, represent its sentences in the form of OCaml expressions. OCaml is a functional language, so the fundamental

\footnote{Actually, F# started as a dialect of OCaml, but later diverged.}
operation is application. It seems appropriate then to represent Lang’s sentences as OCaml applications. Assume a function int. Then the application int 1 can 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 into OCaml’s top level, we immediately get an error that add is not defined. Saying to ourselves ‘assume add exists’ is not enough: we have to say it to OCaml. As the first step, we have to consider 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 must also have some type. At this point, of describing the language, we do not care what exactly it is. Therefore, we leave 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, the sample Lang sentence is represented by the following OCaml expression:  

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

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).

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

\[\text{It may be surprising that a very similar expression appears in the paper that introduced ML, the predecessor of OCaml, back in 1978 Gordon \textit{et al.} [1978]. Perhaps one should not be too surprised: after all, ML was designed for representing languages. That is what ML stood for: Meta Language.}\]
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 `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, `L` is the name of some implementation of the signature `Lang`: that is, some `Lang` interpreter. The expression `open L` brings the operations it defines – `int` and `add` – into scope, so that we may use them (without needing to attach `L.` prefix all the time).

To evaluate that expression we need an implementation of the `Lang` signature. Here is one:

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

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

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

which gives the OCaml value 2, the meaning of `Ex1` in `Eval` as an OCaml integer.

`Lang` hence may also be viewed as the signature of the DSL interpreters, which give a particular meaning to DSL expressions. `Eval` is not the only possible interpreter of `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 = “(“ ˆ x ˆ “+” ˆ y ˆ “)”
  end
```

---

5In the graduate school you may learn that a `Lang` implementation specifies a denotational semantics for our language: `repr` defines the domain, and `int` and `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.
Interpreting the same \texttt{Ex1} using \texttt{Pp}, as

\begin{verbatim}
let module M = Ex1(Pp) in M.res
\end{verbatim}

now gives the string "\((4\omega+\omega)\omega+(\omega+\omega+1))".

Besides evaluating DSL expressions we may also want to transform them. A simple example is negation. The intent is that evaluating a DSL expression in the ‘negated language’ should give the opposite result. This transformation can also be written as an interpreter: after all, interpreting is the only thing we can do with tagless-final expressions.

\begin{verbatim}
module Neg(F:Lang) = struct
  type repr = F.repr
  let int x = F.int (\(-x\))
  let add e1 e2 = F.add e1 e2
end
\end{verbatim}

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

\begin{verbatim}
let module M = Ex1(Neg(Eval)) in M.res
let module M = Ex1(Neg(Pp)) in M.res
\end{verbatim}

The result is easy to imagine. We also confirm that

\begin{verbatim}
let module M = Ex1(Neg(Eval)) in M.res
let module M = Ex1(Eval) in M.res
\end{verbatim}

indeed give the opposite results.

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

\begin{verbatim}
module Ex1Neg(F:Lang) = Ex1(Neg(F))
\end{verbatim}

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

\begin{verbatim}
let module M = Ex1Neg(Eval) in M.res
-2
let module M = Ex1Neg(Pp) in M.res
"((-4\omega+\omega)\omega+(\omega+\omega))"
let module M = Ex1Neg(Neg(Eval)) in M.res
2
\end{verbatim}
We now ‘modularize’ our DSL development, arranging the DSL definition, interpreters, transformers, and the testing script each in a separate file. Look in the directory tfintro of 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. The file lang.mli contains:

```ocaml
(** Two operations of the language *)

val int : int -> repr
val add : repr -> repr -> repr
```

which is the content of module type Lang introduced earlier. In OCaml, a file with the .mli extension such as lang.mli is treated as a module type (signature) declaration: as if module type Lang = sig ... end were wrapped around it. That is, an .mli file is a signature declaration that can be separately compiled (the compiled file has the .cmo suffix). The signature name is derived from the file name by capitalizing its first letter. After compiling lang.mli, the signature it contains can then be referred to as module type of Lang.

Likewise, the file eval.ml contains:

```ocaml
let int x = x
let add e1 e2 = F.add e1 e2
```

which is the content of the earlier module Eval. In OCaml, a file with the .ml extension such as eval.ml is treated as a module declaration: as if module Eval = struct ... end were wrapped around it. The module name is the file name with the first letter capitalized. After compiling eval.ml (which gives the eval.cmo file), we can refer to its content as Eval.int, Eval.add, etc. (The compiled eval.cmo have to be linked in into the executable – or, if using the top-level interpreter, loaded by the #load directive.)

Alas, no straightforward wrapping exists for functors. The contents of neg.ml is:

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

which is the module Neg containing the functor also named Neg. To refer to the functor, we have to say Neg.Neg.

Please pay particular attention to ex1.ml:
module type Lang = module type of Lang

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

let x = let module M = Ex1(Eval) in M.res
let _ = assert (x=2)

let x = let module M = Ex1(Pp) in M.res
let _ = assert (x = "((4+0)+((-1+(-1)))")

module Ex1Neg(F:Lang) = Ex1(Neg.Neg(F))

let x = let module M = Ex1Neg(Eval) in M.res
let _ = assert (x = -2)

let () = print_endline "All Done"

(The first line introduces the abbreviation Lang for module type of Lang.) The file 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 running it finishes normally, printing “All Done”, there is some confidence things work as expected. If it fails with an error, we have to investigate.

Exercise 1. Extend the language with some other operation, and correspondingly extend the interpreters Eval, Pp and Neg. Write an example that uses the old and the added operations, and try interpreting it in the extended interpreters.

In more detail: Copy the tfintro directory into your repo. Add new files: lang2.mli (language extended by you), eval2.ml (extended Eval interpreter), pp2.ml (extended Pp interpreter), neg2.ml (extended Neg transformer), ex2.ml (with the tests that cover your added feature, and also the tests checking that the earlier features are not broken by your additions). Confirm on a sample example(s) that in the extended language as well, evaluating using Neg2(Eval2) gives the result opposite to that of the Eval2 evaluation. Add the ex2 target to Makefile and update README.dr correspondingly.

By the submission deadline, the tfintro directory in your repository must contain the above mentioned files (in addition to the files copied from the class repo’s tfintro) and the updated Makefile, README.dr. Make sure that the regression test ex2.ml passes.
4 Making the compiler

We now build the compiler for Tiger, in many small steps.

4.1 The simplest source language

All the code for this section is in the directory step1 in the class repo. Furthermore, the directory util contains commonly used utility code. Copy both directories into your private repository.

As in §3, we start with the simplest language: in fact, even simpler. Our first language to compile has only integer literals. To be precise, see Spec. 1.

**Specification 1.** Our compiler is to read a source file that contains a single signed 64-bit integer. If the source file does not contain only a single integer, or contains a signed integer that does not fit within 64-bits, the compiler must report an error. Otherwise, it is to produce an assembly code and, eventually, an executable that, when run, prints the integer that was in the source file.

Although the source language cannot be any simpler, its compilation, albeit trivial, has lots of minute, bothersome details. Building a running executable does take a bit of work.

How to write an assembly code that prints an integer? Especially if one does not know any assembly? Let’s ask something that does know: for example, the C compiler. Let’s compile the simplest representative C program:

```c
#include <stdint.h>

extern void print_int(int64_t);

void ti_main(void) {
    print_int(-42LL);
}
```

and look at the produced assembly code. The cleaned and commented assembly code is as follows (see the file clean_int.s):

```assembly
# Beginning of the code
.text

# The name (label) defined below is global (visible outside this file)
.globl ti_main

# This name is a function name (that is, it points to code)
.type ti_main, @function

# The declaration of the name itself
ti_main:

# x86-64 ABI, stack alignment: see the text
.subq $8, %rsp

# move -42 to the argument register
```

---

6The listing looks a bit different on a Mac, since MacOS uses a different executable file format and hence different conventions for naming sections and global identifiers such. We will have to adapt to it later.
movq  $-42, %rdi
# call the external function
call  print_int
# restore the stack
addq  $8, %rsp
# return
ret

Most of the code is self-explanatory, except for \texttt{subq} $8, %rsp. It comes from the stack alignment rule of the x86-64 Application Binary Interface (ABI): before any function call, the stack must be aligned at a 16-byte boundary. Therefore, upon a function entry \texttt{rsp} is 8 (mod 16), because the 8-byte–long returned address has been pushed onto the stack. If the function contains calls to other functions, it has to re-align the stack at a 16-byte boundary. The easiest way to accomplish that is to subtract an odd number of 8s (see Fog [2022]).

With the above assembly code as a template, we write our first compiler: file \texttt{compiler0.ml}, which takes an input stream and produces the assembly code file. I insist on writing type signatures of all top-level functions (except those extremely small), in one of the two styles (see the code). The code takes the ‘template-base code generation’ to the heart. It is really just the template substitution, using \texttt{printf}. It is very similar to the familiar C \texttt{printf}. In particular, characters ‘\%’ in the format string that are meant to be printed literally have to be doubled. Unlike C however, if you forget it, you get a type error.

\begin{verbatim}
let compile (ich:in_channel) (och:out_channel) : unit =
  Scanf.fscanf ich "\%d" @@ fun n ->
    Printf.fprintf och {q|..|...|q}
    .text
    .globl ti_main
    .type  ti_main, @function
ti_main:
    subq  $8, %rsp
    movq  $%d, %rdi
    call  print_int
    addq  $8, %rsp
    ret
|q} n
\end{verbatim}

Here, \{q|...|q\} is the alternative OCaml syntax for string literals, suitable for multi-line strings. This code is not only simple but also can be tested right away, at the top level: \texttt{compile stdin stdout}. Its drawback is that it is too simple: too specific and hard to generalize. And generalize we must, already, to account for MacOS, whose executable format is a bit different.

Let’s look at the assembly as a DSL. Which operations do we need from it? We need to build instructions (function calls, moving a number into %rdi); compose them into a sequence; and to turn an instruction sequence to the code for the \texttt{ti\_main} function, wrapping into the suitable prologue and epilogue. And we need to write the complete code into a file. Hence we come to the following interface (see \texttt{asm.mli}).
Using this interface, the compiler code becomes

```ocaml
let compile (ich:in_channel) (och:out_channel) : unit =
  let n = scanf.(Scanf.bscanf (Scanf.Scanning.from_channel ich) "%d" Fun.id in
  let open Asm in
    (movq (imm n) (reg rdi) @ call "print_int") |> make_function "ti_main" |> write_file och
```

We can try to compile it (see the target `test_compiler`).

Obviously, we need an implementation of the Asm interface: see `asm.ml`. It realizes the abstract types `instr`, `register` and `operand` as mere strings, and takes some care to print the code prettily. The implementation is a good place to take care of the differences between Unix/Linux/WSL and MacOS platforms. These platforms have different executable code formats (ELF and Mach-O, resp.) and slightly different ABI. The main differences are: the MacOS assembler does not accept the `.type` directive; the names of global symbols must begin with an underscore. These differences are isolated in `asm.ml`; the rest of the compiler does not need to know about them.

How to find what platform we are compiling for? We may try to discover via a run-time test (e.g., `uname -s`). The easiest, however, is to set this information at the configuration time – as done in all compilers to my knowledge. To this end, we add the file `config.ml` with the relevant configuration data, prepared by a configuration tool: at present, by the compiler writer.

Generating assembly is only a part of the job. We also need to implement the function `print_int`, and we need to arrange to invoke `ti_main`. Here C helps: see the file `init.c`. We also have to assemble (invoke assembler on) our emitted assembly code, and link it with `init.o` and the code to initialize the C standard library that we are using. It is a lot of work. Luckily, we can leave all of that to gcc:

```
gcc compiled_code.s init.o
```

19
Finally, we need the driver.ml that puts everything together: accepts arguments from the command line, opens needed files, invokes the compile function and arranges the build of the final executable. It has lots of error handling: when the input file is not given, when the input file does not exist, when a syntax error is detected, when the file to output the assembly code could not be opened, or when the build failed for some reason.

The target tigerc in the build script (see §4.1.1) links everything together and builds the compiler: Build/tigerc. If we write the string ”−42” into a file /tmp/prog1.tg and invoke

Build/tigerc prog1.tg

we obtain the executable file Build/a.out, running which prints −42.

If we examine the produced a.out using nm a.out, we see ti_main that we generated. We also see a lot of other stuff, needed to make the program to run. Note _start, which is invoked by the OS kernel to start the program. After a lot of initialization, it eventually calls main, which calls our ti_main. Luckily all this other stuff can be entrusted to gcc (the toolchain), and we only have to concentrate on generating the assembly code. Most of other compilers take a similar approach.

The file test_script.ml is the test script, using the testing framework defined in the directory util. The build target test runs the test suite. It should finish without errors, printing “All Done” at the end.

4.1.1 Build system

In this class we will be using a dedicated build system, which one may think of as a version of make: with all the needed facilities (such as build directory, dealing with the source code spread around many directories, etc.) but without their complexities. One may read the rationale in the file util/build.ml, which is the implementation.

To use the build system, first compile all the code in the util directory.

In our build system, all the built artifacts – compiled code such as .cmi and .cmo files, tigerc itself, etc. – are stored in a special build directory, called Build.

Check that your copy of the compiler code (e.g., step1 directory) contains Build as a subdirectory. If not, make it.

The build script is an OCaml script build.ml. It is meant to be invoked as is: ./build.ml. On some platforms, invoking the script like this may cause errors. In this case, run this script as ocaml build.ml.

The interface is the same as make: the script accepts the list of targets to make, and makes them in the specified sequence. For example:

./build.ml tigerc test
or, on some platforms
ocaml build.ml tigerc test

The list of targets may be empty, in which case the default target is built (which is usually tigerc). In this class, we typically use three targets: tigerc to build
the compiler and the run-time system, test to run the tests, and clean to clean everything up.

The build script **build.ml** is a regular OCaml file, and can be edited as such. One should particularly note three definitions. First, **compiler_manifest** lists all the source files needed to build the compiler, and how to build them. *The order is important.* (In fact, the order is the linking order of the files).

```ocaml
let compiler_manifest = build_all [ 
  existent "../util/util.cmo";
  ocaml "config.ml";
  ocaml "asm.mli";
  ocaml "asm.ml";
  ocaml "compiler.ml";
  ocaml "driver.ml";
]
```

The argument of **build_all** is a list of rules. The rule **existent** checks to see that the file (**../util/util.cmo** in the above example) already exist, reporting an error if does not. The rule **ocaml** is to compile the given .ml or .mli file by the OCaml compiler. Later we shall see that the rule has the optional argument "rename", used as:

```ocaml
  ocaml "../step21/lang.mli" "rename:"lang_21"
  ocaml "../step21/pp_ast.ml" "rename:"pp_ast_21"
```

For example, the former first renames **..
/step21/lang.mli** to **Build/lang_21.mli** before compiling it (which produces **Build/lang_21.cmi**: all built artifacts go to the **Build** directory). Further rules include **ocamlex, ocamlyacc** (for lexer and parser generation, resp.), **cc** for compiling C code and **link** for linking. For example,

```ocaml
let tigerc = link "out:"tigerc" compiler_manifest
```

is the rule to link all files built by **compiler_manifest** into the executable named **tigerc**. The earlier **build_all** is also a rule: collecting several rules into one.

Another important definition in the build script is

```ocaml
let runtime_manifest = build_all [ 
  cc "init.c";
]
```

which is the rule to compile the files that make the Tiger’ run-time system. Finally,

```ocaml
let tests = [ 
  "test_script.ml";
]
```

lists the tests to run (when building the target **test**). As we extend the compiler, we add new files to **compiler_manifest, runtime_manifest**, and **tests** lists. Typically, one would not modify anything else in the **build.ml** script.
References


