

Compilers: Course Notes

Oleg Kiselyov

January 26, 2026

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; which runs 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 other goal of the course is to give a taste of modern software development, specifically: test-driven development, version control, and the stress on reading, comprehending and extending code rather than writing from scratch.

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 a *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 the extensibility afforded by it. The 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).

Yet another feature is the attention given to names, or 'variables', and associating attributes to them. Our approach, which readily permits adding attributes at will and analyzing variable usage, may remind some of algebraic effects.

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

I	Introduction	5
1	Course Overview	5
1.1	Prerequisites	5
1.2	Development environment	5
1.2.1	Operating system and environment	5
1.2.2	OCaml	6
1.2.3	Git	7
1.3	Exercises and Grading	8
2	What is a compiler?	9
2.1	Incremental approach	13
3	Introduction to tagless-final style and OCaml reminder	14
3.1	A simple embedded DSL and its interpreters	15
3.2	DSL transformations	19
3.3	Separate compilation: making a project	22
3.4	Extensibility with modules	25
II	Making the compiler	27
4	The simplest source language	27
4.1	Printing an integer, in Assembly	27
4.2	Assembly as a DSL and writing a simple but real compiler	30
4.3	Testing a compiler	31
4.4	Build system	33
4.5	What is a bug?	36
5	Unary expressions	36
5.1	Languages, Grammars, Parsing	37
5.2	LL(1) grammars and recursive descent parsing	39
5.3	Tokenizing	41
5.4	Compiling unary expressions	43
5.5	Testing a compiler, more thoroughly	47
5.6	Types	50
5.6.1	Negative View	50
5.6.2	Positive View	52
5.6.3	Types and grammars	53
5.7	Compiling integer and boolean unary expressions	53
6	Challenge of binary operations	56
6.1	Migrating to ocamllex/ocamlyacc	61

7	Local variables	62
7.1	Shadowing	66
7.2	What is a variable	66
7.3	A variable is a hole in an expression	67
7.4	Question-answer semantics	68
7.5	Type checking with variables	71
7.6	Why memory allocation	72
7.7	Stack	73
7.8	Stack-based allocation	74
8	Full expression language	74
8.1	Atomic binding optimization	76
9	Better Memory Allocation	78
10	Dummy expression and sequencing	78
11	Mutable variables, assignment, function calls	79
11.1	Side-effects are not simple	81
11.2	Function calls	81
12	Strings	82
12.1	Data Section	84
13	Integer Arrays	86
14	Comparisons and conditionals	88
15	Loops	90
15.1	for-Loops	91
16	Local Functions	92
16.1	Type-checking	93
16.2	Compiling local functions	97
16.3	Accessing ancestors' variables: Lexical and dynamic binding	99
16.4	Lambda-lifting	101
16.5	Accessing an ancestor frame directly	102
16.6	Accessing mutable ancestor variables	102
17	Final exercises	103
III	Conclusions and Reflections	103
18	Future Work	103
19	Reflections on Language Design	104

20 The place of Tiger' in the current landscape	106
21 Compilers are more fun than you may think	107
22 Projects	108

Part I

Introduction

1 Course Overview

Compilers is a practical course: building a compiler from a high-level procedural language Tiger' to x86-64 assembly. The compiler is to run on students' laptops and produce executables that run on the laptops.

Besides building a complete, realistic compiler, the course aims to give *a taste of modern software development*:

- test-driven development;
- pervasive use of version control; build pipelines;
- stress on reading, comprehending and extending code rather than writing from scratch;
- explicitly spelled-out requirements.

1.1 Prerequisites

- + Familiarity with OCaml or the closely related F#, or the willingness to learn (see Resources in §1.2.2).
- + Light familiarity with C: Tiger' compiled code follows the C application binary interface (ABI), to allow the reuse of the C standard library. Tiger' minimal standard library is to be written in C as well.
- + Programming experience
- *No* requirement to know the x86-64 assembly language beyond the basics of computer organization.
- *No* experience is assumed with parsing, type-checking, code-generation.

1.2 Development environment

The first task is to set up and test the development environment.

1.2.1 Operating system and environment

Windows If you use Windows, first install WSL2, which gives you the Unix (Linux) environment – the development environment for the course. In particular, use that environment when installing OCaml as described later. Confirm that you have `gcc` (at least v11), `git` and GNU `make` installed. (If they are somehow missing, install them)

MacOS, Intel On an Intel Mac, first 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 or container running Linux (distribution does not matter).

MacOS, Mac M (Apple Silicon) On Apple Silicon, there are two choices. One is to install Rosetta. Then install XCode, then Homebrew, and then `make` and `git`. All toolchain (especially the compiler, `gcc` or `clang`) has to be for the `x86_64 architecture`. One way to ensure it is to run all install commands prefixing them with `arch -x86_64`. A better way is to open the terminal in the rosetta mode: `arch -x86_64 alacrity` and run all commands from there. You will use the same rosetta mode terminal to build the compiler, run it and run the compiled executable – basically do all the work in the class.¹ Our compiler (and development) will be under MacOS, using its tools, but for the `x86_64` CPU architecture.

As an alternative, install UTM <https://mac.getutm.app/>, using, as a VM image, `debian-12-nocloud-amd64.qcow2` (or newer) from <https://cdimage.debian.org/cdimage/cloud/bookworm/latest/> (other distributions would also work). In that case, our compiler and development will be under Unix/Linux.

You may use any editor or 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 our compiler. OCaml is truly very good for writing compilers: the first Rust compiler was developed in OCaml; the reference Wasm implementation is in OCaml; and so is Meta’s Hack language. The well-known theorem prover Rocq is written in OCaml (another well-known theorem prover Isabelle/HOL is written in Standard ML: OCaml’s close relative).

For this course we need OCaml version at least 5.3. We will only use the standard library; no extra packages are necessary. For build, we use Makefiles at the very beginning, but then switch to a custom build pipeline. Therefore, `dune` will not be needed.

installation (Windows) <https://tarides.com/blog/2024-05-08-how-to-setup-ocaml-on-windows-with-wsl/>

To re-iterate, first install WSL2, open the WSL2 terminal and install OCaml following the directions for *Unix installation* (rather than for ‘Windows installation’).

¹For configuring VS code, see <https://stackoverflow.com/questions/70217885/configure-m1-vscode-arm-but-with-a-rosetta-terminal>

installation (Mac, Linux) <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.

reference <https://ocaml.org/manual/index.html>

books There are quite a few, of various levels; in particular: See [Clarkson \[2022\]](#), [Whittington \[2013\]](#)

other resources The ocaml.org web site links to many OCaml tutorials, libraries, projects and other resources. See also <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.² First, install `git` and learn its basics. There are many, many `git` tutorial available on the internet; there are also several books. We will be using only the very basic `git` features (no branches, submodules, etc.). Specifically, you need to be familiar with `git status`, `git pull`, `git add`, `git commit -a`, `git push`. Also useful are `git diff` and `git log -p -n`. To make a repo, use `git init` or `git clone`.

For the course,

1. make an account for yourself at <https://bitbucket.org/> To access it from your computer, you need either to create an APP password (click on Help and read the documentation³) or register SSH keys. If you set up the APP password, *be sure to save it somewhere*: you need to enter it every time you pull or push to the bitbucket. (SSH keys are quite more convenient.)
2. make a *private* repository for yourself, including your name or student ID in the repository name. This will be your development repository for your compiler.
3. 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`).

²Adventurous may consider `jujutsu`, which is compatible with `git`: <https://v5.chriskrycho.com/essays/jj-init/>

³<https://support.atlassian.com/bitbucket-cloud/docs/create-an-app-password/>

`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 Exercises and Grading

This is a practical course. Each week there are 1-3 homework assignments: exercises. Usually, one exercise is to write tests for a new compiler feature, and the other is to extend the current compiler code to implement the feature and make the tests pass.

The purpose of the exercises is to deepen the understanding of the compiler code, to encourage thinking of implementation strategies – and to learn and practice modern software development. *There is no single right answer!*

Besides the mandatory there are also optional exercises, often a bit more challenging. Optional exercises do not have to be submitted. If submitted, they will be graded as regular ones. There is also a possibility of projects, for those looking for a challenge.

The exercises are graded on the scale of 0 to 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 midterm or final exam.

Submission deadline Homework assigned on week N must be submitted by the start of the $N + 1$ -week class.

Submission procedure 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.

1. Copy the directories from the class repository to your repository (on your laptop), as specified in the assignment.
2. Extend or add to that code, as described in the assignment. Your homework answers should be in the files and directories with the specified names: check the assignment.
3. *Test* your code before submission, to make sure it compiles, runs, and passes the test cases.
4. By the submission deadline, *commit* your code from your laptop into your bitbucket private repository.
5. At the deadline, I clone your private repo, compile your code and run it on your, and perhaps also mine, tests. *If the code fails to compile or fails your own 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* A submission that does not satisfy the assignment cannot be accepted. After all, this course is about programming: talking to a computer. A computer does not know what you mean: It only knows what you entered. Even a one letter mis-spell in a program is often fatal. Also think how forgiving is *your compiler* of mistakes. (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 letter, 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. No points are given if the submitted code fails tests: ends up in an exception or infinite loop although given valid input.

Submitting code that fails to compile or fails the tests is called ‘breaking the build’ and is regarded as one of the cardinal sins in software development. You might want to search the net or YouTube for ‘breaking the build’ to see how companies, or colleagues, treat those who break the build. Getting zero points is a comparatively minor punishment.

There is no single right answer On the other hand, if the submitted code satisfies all conditions of the assignment, it will get the full 10 points – or maybe more, if particularly clever or impressive.

2 What is a compiler?

Everybody must know what is a computer. Everybody also knows that a computer has the CPU to execute programs, memory to store programs and data, and some sort of IO devices. At its basic, 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
```

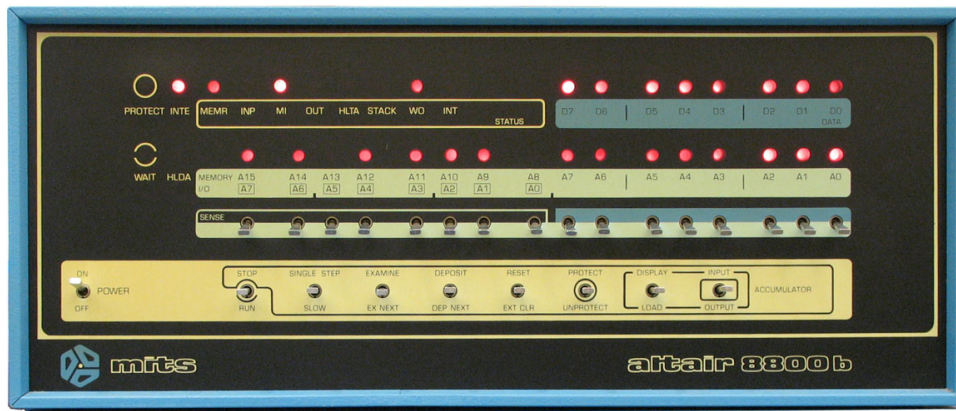
The numbers you see are the instructions, for the modern Intel/AMD CPU, called x86-64 architecture. (See [CS107 \[2021a,b\]](#) for introduction and [Cloutier \[Ed.\]](#) for complete reference.) It is currently the most widely used architecture for desktop and laptop computers.⁴

⁴If you own a recent Mac, you are using a different, ARM architecture. Still, it has the x86-64 emulation mode.

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 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.

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).



The original Altair had no display or keyboard. What you see is the entire user interface. One enters a program using switches, and reads the results by displaying memory bytes (bits) on the indicator lamps.

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, inconveniences are many, 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.⁵

```

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

```

⁵The 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.

```

    movq    $0, 24(%rsp)
    movq    $1, 16(%rsp)
    jmp     .L5
.L6:
    call   read_int
    movq   %rax, (%rsp)
    movq   (%rsp), %rax
    addq   %rax, 24(%rsp)
    addq   $1, 16(%rsp)
.L5:
    movq   16(%rsp), %rax
    cmpq   8(%rsp), %rax
    jle    .L6
    movq   24(%rsp), %rax
    movq   %rax, %rdi
    call   print_int
    addq   $40, %rsp
    ret

```

This is a good place to recall how a CPU executes the instructions: the main CPU loop:

1. fetch the instruction pointed out by the instruction pointer (rip);
2. decode the instruction and increment rip;
3. execute the instruction, modifying registers, memory or CPU flags; repeat. A branch instructions changes rip, by loading a new address or adding/subtracting an offset.
4. repeat.

For more details, see the simple Intel x86-64 emulator in [scratch/emulator.ml](#) of the class repository.

The assembly code 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, how much temporary space to

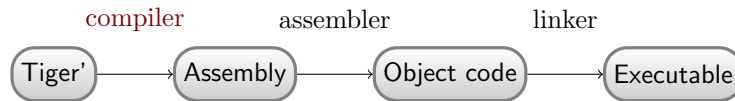
allocate (see `subq` and `addq` instructions), what a stack location such as `24(%rsp)` means (at a given program point), etc.

And so were invented higher-level languages, to make programs easier to comprehend and to write. The first higher-level language was FORTRAN, which stands for FORmula TRANslator. (FORTRAN was the first programming language I learned, in high school in late 1970s). Its motivation was to let the programmers write formulas in the 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

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

Has it become easier to understand? Anyone can tell what the program does? One may have perhaps noticed the distinction between `val` and `var`, and between `'=`' and `':='`.

The program to translate such easier to understand code to the assembly is a *compiler*. There are further steps to the final 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.

Before we begin, here is one more example of a Tiger' program, featuring arrays and nested functions. Indeed, Tiger' is more than just C.

```
// N-Queens problem
function nqueens(n:int) = begin
  array row[n]
  array col[n]
  array diag1[(2 * n) - 1]
  array diag2[(2 * n) - 1]
```

```

function print_board() = begin
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      print(if col[i] = j then "_0" else "_." fi)
    done; print_line()
  done; print_line()
end

function try(c: int) =
  if c = n then print_board() else
  for r = 0 to n - 1 do
    val rc1 = r + c
    val rc2 = r - c + n - 1
    in
    if row[r] = 0 & diag1[rc1] = 0 & diag2[rc2] = 0 then
      row[r] := 1;
      diag1[rc1] := 1;
      diag2[rc2] := 1;
      col[c] := r;
      try(c + 1); // recur
      row[r] := 0; // restore arrays, to try other solutions
      diag1[rc1] := 0;
      diag2[rc2] := 0
    fi
  done
fi

in try(0)
end

in
nqueens(8)

```

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;
4. Run all the tests;
5. Base on the experience, refactor or optimize. 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 [2024], 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.⁶ This section is a brief overview, stressing the module system (quite more powerful, compared to F#), which we will be using extensively.

The characteristic of the course is treating a compiler as a stack of DSLs. DSL – Domain-Specific Language – is a relatively small language designed for a specific task and hence has the vocabulary, syntax, idioms, semantics tailored for that task Hudak [1996]. Examples include TeX and Matlab. Broadly speaking, DSLs are *conlangs* – at least that is how we would look at them. DSLs can be either standalone (e.g., TeX and Matlab) or embedded. The latter are DSLs incorporated into a larger, general-purpose language – called host language or metalanguage – and piggy-back on its its syntax and some vocabulary, but still possessing their own semantics and specialized vocabulary. One may say that an embedded DSL is a *jargon* of its host language.

This section also introduces the so-called tagless-final style Kiselyov [2024] 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 a DSL one-by-one, reusing the already written code. As we shall later see, we treat Tiger’ as a DSL, and the Tiger’ compiler as one of its interpreters. The assembly and the intermediate languages are also treated as (embedded) DSLs. The extensibility of interpreters is particularly valuable in the incremental approach.

⁶F# started as a dialect of OCaml, but later diverged.

3.1 A simple embedded DSL and its interpreters

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

1	x
-1	$(4 + x)$
$((4 + x) + -1)$	$(-1 + -1)$
$((4 + x) + (-1 + x))$	$((4 + x) + -1) + x$

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

$$\begin{aligned} S &\rightarrow \text{integer} \\ S &\rightarrow 'x' \\ S &\rightarrow (S + S) \end{aligned}$$

Let's embed **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 **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

```
add (add (int 4) x) (add (int (-1)) x)
```

could represent the sentence $((4 + x) + (-1 + x))$, which is to be our running example.

If we enter the above expression at the OCaml's toplevel prompt,

```
add (add (int 4) x) (add (int (-1)) x)
~> Error: Unbound value "add"
```

we immediately get the error that `add` is not defined. (Here, `~>` shows the response of the OCaml top level). Saying to ourselves 'assume `add` exists' is not enough: we have to say it to OCaml. To formally declare to OCaml that `int`, `x`, and `add` are assumptions, we make them function arguments. After all, a function is an implication: given (assuming) the values of its arguments it produces the value of its body.

```
fun int add x →
  add (add (int 4) x) (add (int (-1)) x)
~> - : (int → α) → (α → α → α) → α → α = <fun>
```

It is worth taking a moment to think of the types of `int`, `x` and `add`, using as a hint the type inferred by OCaml for our example. **Lang**'s sentences are represented as OCaml's applicative expressions. All expressions in OCaml (that is, the ones accepted by the 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 variable `x`, which is an expression by itself, has hence the type `repr`, and the functions `int` and `add` then have the types

```
int: int → repr
add: repr → repr → repr
```

Hence the sample `Lang` sentence, now with explicit type annotations, is represented by the following OCaml expression (we also gave it a name, for easy reference):⁷

```
let e1 = fun (type repr) (int:int→repr) (add:repr→repr→repr) (x:repr) →
  add (add (int 4) x) (add (int (-1)) x)
```

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` as well as `x` 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`, `x` 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.⁸

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).

```
module type Lang = sig
  type repr
  val x: repr
  val int: int → repr
  val add: repr → repr → repr
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. And `x` is an expression by itself. Note how closely `Lang` matches the context-free grammar of the language, repeated below:

$$\begin{aligned} S &\rightarrow \text{integer} \\ S &\rightarrow \text{'x'} \\ S &\rightarrow (S + S) \end{aligned}$$

One may say therefore that `Lang` defines the *syntax* of our DSL.

Using the signature, the sample `Lang` expression is written as

⁷It may be surprising that a very similar expression appears in the paper that introduced ML, the predecessor of OCaml, back in 1978 [Gordon 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.

⁸Many languages have a facility to group definitions, or just their types (prototypes), under a single name: cf. Go interfaces, Rust and Scala’s traits, virtual/abstract classes or namespaces in C++.

```

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

```

It takes the form of a *module* (a named group of definitions – in our case, a single definition of `res`) parameterized by another: in our case, an implementation of the signature `Lang`. The formal OCaml name for such a parameterized module is *functor*. To be able to refer to the `Lang` implementation within `Ex1`, it is given a name, `L` in our case. Since explicitly qualifying DSL operations by `L` is cumbersome, one may use `open L` to bring the operations it defines – `int`, `x` and `add` – into scope, so that we may use them without needing to attach the `L` prefix all the time.

```

module Ex1(L:Lang) = struct
  open L
  let res = add (add (int 4) x)
              (add (int (-1)) x)
end
~> module Ex1 : (L : Lang) -> sig val res : L.repr end

```

The inferred type looks like a ‘module function’: *given a `Lang` implementation, `Ex1` produces the interpretation of our sample DSL expression in that implementation.*

Just like ordinary functions, the only thing we can do with `Ex1` is to apply it – to an implementation of `Lang`. We need, therefore, an implementation of `Lang`. The sample DSL expression `Ex1` looks like an arithmetic expression; it is tempting to treat it as such – that is, consider DSL integer literals as OCaml literals, and DSL addition as the OCaml addition:

```

module Eval = struct
  type repr = int
  let x = 42
  let int (i:int) = i
  let add = (+)
end
~> module Eval :
sig
  type repr = int
  val x : repr
  val int : repr -> repr
  val add : repr -> repr -> repr
end

```

The module signature for `Eval`, inferred and reported by the OCaml top level, matches `Lang`. (Actually, it is an instance of `Lang`, with `repr` now a concrete type: `int`). Therefore, `Eval` is an implementation of `Lang`: 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 the 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. That is, we apply `Ex1` to `Eval` – in effect, obtaining the value of `Ex1.res` with `int`, `x` and `add` as defined in `Eval`:

```
module M = Ex1(Eval)
```

and see what `res` has evaluated to:

```
M.res
~~ 87
```

The result `87` is hence the meaning of `Ex1.res` in `Eval`: an OCaml integer. More compactly – as a single line and single expression – the interpretation of `Ex1` can be written as

```
let module M = Ex1(Eval) in M.res
~~ 87
```

One may say that a DSL interpreter like `Eval` gives *a* meaning to a DSL expression:⁹

Whereas the `Lang` signature is the syntax for the DSL, an implementation of this signature – the DSL interpreter – is the DSL *semantics*.

`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:

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

Interpreting the same `Ex1` using `Pp` now gives the string.

```
let module M = Ex1(Pp) in M.res
~~ - : string = "((4 + x) + (-1 + x))"
```

Remark 1. If we enter the result of pretty-printing `Ex1.res` at the OCaml prompt (ensuring `x` is defined appropriately)

```
let x = 42;;
((4 + x) + (-1 + x));;
~~ - : int = 87
```

⁹In graduate school you may learn that a `Lang` implementation specifies a denotational semantics for our language: `repr` stands for a 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, only from the denotations of their sub-expressions.

we obtain the same result as evaluating `Ex1.res` with `Eval`: that is, `Eval` and `Pp` give coherent interpretations – at least in this example. One may wonder how to ensure DSL interpreters are always coherent. After all, nothing would have prevented us from defining the `Pp` interpreter in which `int` and `add` always produce "0".

To make sure `Eval` and `Pp` are coherent, we merely need to examine their implementations of the DSL primitives: concretely, that the result of `Pp.int`, that is, `string_of_int i` corresponds to `Eval.int i` $\equiv i$ for all integer i . Likewise, the results of `Pp.add` and `Eval.add` correspond assuming their arguments do. The verification does involve some work: a manual or a computer-assisted formal proof. The proofs are typically simple and modular: one DSL operation at a time. Once we verified that each DSL operation, taken in isolation, is coherently implemented in DSL interpreters, we are certain any DSL expression, no matter how complex, shall be coherently interpreted.

3.2 DSL transformations

Besides evaluating DSL expressions we may also want to transform them. A DSL transformation inevitably brings to mind examining DSL expressions: that is, deconstructing, or pattern-matching on them. However, there seems to be no way to pattern-match on tagless-final expressions.

First of all, there is.¹⁰ Second, there is a different view on expression transformations, closely related to a so-called *normalization-by-evaluation* – which is particularly suitable to compiler optimizations and, so happens, to the tagless-final style. This section is a very simple introduction.

As a sample transformation, let's re-associate our DSL expressions to the left; applied to our running example:

$$((4 + x) + (-1 + x)) \mapsto (((4 + x) + -1) + x)$$

A left-associated expression is much easier to evaluate, using a single accumulator. It is, hence, much easier to compile, as we shall see later.

The most common, textbook, implementation of this transformation is by re-writing: apply the re-writing rule

$$(e_1 + (e_2 + e_3)) \mapsto ((e_1 + e_2) + e_3)$$

repeatedly, until it no longer applies. The rule is a particular form of the associative law of the addition. The re-writing certainly a deconstruction, or pattern-match, on the expression: in fact, e_1 , e_2 and e_3 are the pattern variables, which stand for an arbitrary expression.

Such a re-writing transformation immediately raises a few questions. First, which particular form of an associative law to use for re-writing? After all, an equational law is an equality, and hence 'unoriented'. Does this repeated re-writing actually terminate? If it does, is the result as we want? In case there

¹⁰<https://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html> and <http://okmij.org/ftp/tagless-final/datatypes.html>

are several re-writing rules: does the order matter? And if it does (as often happens), how to find the ‘right’ order of applying the re-writing rules? There is also the question of efficiency: each application of a re-writing rule traverses through the whole expression to some degree, and re-writes it, to an expression of the same size. The whole transformation would generally involve many traversals and re-writings (especially for an initially right-associated expression).

The alternative transformation (optimization) method – normalization-by-evaluation, or NbE – has none of these problems, as we are about to see. The termination and correctness are straightforward, and the result is obtained by a single traversal through the expression.

The first step in applying NbE is to decide what exactly do we want: what sort of expression we want as the result. In other words, we have to define the desired expressions (a subset of all expressions) – so-called *normal form*. In the re-writing-based approach, normal form is what we *end up* with when the re-writing terminates because no rule applies. In contrast, in NbE we *start* by defining it. What is a normal form is hence a matter of definition, and varies depending on the task at hand. For our task, the desired subset of `Lang` expressions are those in which addition is left-associated, e.g., $((4 + x) + -1) + x$. Such expressions follow the grammar:

$$\begin{aligned} S &\rightarrow A \\ S &\rightarrow (S + A) \\ A &\rightarrow \text{integer} \\ A &\rightarrow \text{'x'} \end{aligned}$$

One notices that left-associated expressions `S` are *essentially* sequences (lists) of ‘atoms’ `A`. Therefore, we use the list of atoms as the definition of our normal form. (‘Essentially’ does the heavy lifting. The power of NbE is this considerable freedom to chose the representation: anything that could be eventually converted to the real normal form – and that easy to compute with.)

The second step in NbE is deciding how to compute the normal form. That is, for each operation find a way to produce the normal form assuming the operands are in normal form.¹¹ In the tagless-final context, it is defining a DSL implementation that interprets DSL expressions as such normal forms:

```
module LAssoc(L:Lang) = struct
  type repr = L.repr list
  let x = [L.x]
  let int i = [L.int i]
  let add x y = x @ y
  type obs = L.repr
  let obs : repr → L.repr =
    function (h::t) → List.fold_left L.add h t
    | _ → assert false (* impossible *)
end
```

¹¹ And ensuring this way of producing the normal form is coherent with evaluation: Rem. 1.

The interpreter `LAssoc` interprets `Lang` expressions as normal forms, of `– Lang`. That other, target language `Lang` is arbitrary. That is why `LAssoc` is parameterized by `Lang`. When reading `LAssoc` code, read `repr`, `x`, `int`, `add` as pertaining to the language to transform (the source language). On the other hand, `L.repr`, `L.x`, `L.int`, `L.add` pertain to the target language. The source language treats a DSL expression as a non-empty list of target DSL expressions. As it becomes clear, those target expressions are atomic: produced either by `L.x` or `L.int`. The source language `x` is interpreted as a single-element list of the (atomic) target expression: `L.x`. Integer literals are handled similarly. Finally, the addition concatenates the lists.

We also need an observation function, here `obs`, to transform what is an *essentially* a target `Lang` expression to the real target `Lang` expression. In our case, we transform the list of atoms to the left-associated addition of these atoms. Such observation, or ‘read-out’, is the third step of NbE.

That the transformation is terminating is obvious: none of the primitives are interpreted recursively. The correctness is a clear consequence of associativity. As to the performance: The implementation of `add` relies on list append, which is expensive. One may use a better sequence implementation than a single-linked (OCaml) list, with the constant-time concatenation. In fact, we shall be using such an implementation in the course: see `sq.mli` and `sq.ml` in the directory `util`.

`LAssoc` is a parameterized interpreter: it interprets `Lang` expressions in terms of another interpreter, `L`. On the surface of it, `LAssoc` is an interpreter transformer: it takes one implementation of `Lang` and produces another. We may hence transform the earlier `Eval` and `Pp` implementations and use the result to interpret the same `Ex1`, for example,

```
let module E = LAssoc(Eval) in let module M = Ex1(E) in E.obs M.res
↪ - : int = 87
let module E = LAssoc(Pp) in let module M = Ex1(E) in E.obs M.res
↪ - : string = "(((4+_x)_+_1)_+_x)"
```

The `Eval` result is unchanged, as expected: an optimization ought to preserve the evaluation meaning. The pretty-printed result is left-associated, as desired.

Dually, `LAssoc` may also be regarded as an *expression* transformer. Here is the `LAssoc`-transformed `Ex1`:

```
module Ex1LA(L:Lang) = struct
  let res =
    let module E = LAssoc(L) in
    let module M = Ex1(E) in
    E.obs M.res
end
↪ module Ex1LA : (L : Lang) → sig val res : L.repr end
```

`Ex1LA` has the same type as the original `Ex1`: given an interpreter `Lang` it computes the meaning of `res` in that interpreter. That is, `Ex1LA` is a tagless-final

representation of a `Lang` expression – namely, the expression in which additions are left-associated. `LAssoc` hence accomplishes an expression transformation:

Transforming interpreters \Leftrightarrow Transforming DSL expressions

The transformed expression can be interpreted with the existing `Eval` and `Pp` interpreters, or even the `LAssoc`-transformed interpreters:

```
let module M = Ex1LA(Eval) in M.res
~> - : int = 87
let module M = Ex1LA(Pp) in M.res
~> - : string = "(((4_+_x)_+_1)_+_x)"
let module E = LAssoc(Pp) in let module M = Ex1LA(E) in E.obs M.res
~> - : string = "(((4_+_x)_+_1)_+_x)"
```

The last result shows that left-associating an already left-associated expression leaves it as is: our transformation is *idempotent*, as expected of normalization.

Our DSL has been trivial to the extreme. It is, however, a variation of a DSL for a simple version of Tiger’, and we shall come to see it again. That time, we will be compiling it to x86-64 machine code. The compiler is a DSL interpreter – along the lines of interpreters we have just considered.

The well-known and often recommended normalization-by-evaluation tutorial is [Dybjer and Filinski \[2002\]](#). For an elementary tutorial, see [Kiselyov \[2020\]](#). Normalization-by-evaluation in tagless-final style has been used extensively in practice: optimizing database queries, very efficient compilation of stream pipelines, etc.; see [Kiselyov \[2024\]](#) for references.

3.3 Separate compilation: making a project

We further ‘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 `OREADME.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 `OREADME.dr` and examine the files mentioned therein. The file `lang.mli` contains:

```
type repr (* representation type (abstract) *)

(* Operations of the language *)
val x: repr
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 `.cmi` 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:

```
type repr = int

let x = 42
let int x = x
let add = (+)
```

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` has 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 `assoc.ml` is:

```
module LAssoc(L:module type of Lang) = struct
  type repr = L.repr list
  let x = [L.x]
  let int i = [L.int i]
  let add x y = x @ y
  type obs = L.repr
  let obs : repr → L.repr =
    function (h::t) → List.fold_left L.add h t
    | _ → assert false (* impossible *)
end
```

which is the module `Assoc` containing the functor named `LAssoc`. To refer to the functor, we have to say `Assoc.LAssoc`.

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) x)
              (add (int (-1)) x)
end

let x = let module M = Ex1(Eval) in M.res
let _ = assert (x=87)

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

```
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 `Lang` with some other operation, and correspondingly extend the interpreters `Eval`, `Pp` and `LAssoc`. Write an example that uses the old and the added operation, and try interpreting it in the extended interpreters.

In more detail:

- Copy the `tfintro` directory into your repo.
- Add new files: `lang2.mli` (the language extended by you), `eval2.ml` (the extended `Eval` interpreter), `pp2.ml` (extended `Pp` interpreter), `assoc2.ml` (the extended or re-written `LAssoc` transformer), `ex2.ml` (the example of using the extended language, with the tests that cover your added feature, and also the tests checking that the earlier features are not broken by your additions).
- The optimization of your extended language, and its implementation in `assoc2.ml`, is up to you. That is, what is to be the normal form is your choice. Be sure to check, however, that your normalization `Assoc2.LAssoc` preserves the result of the `Evaluation` and that it is idempotent.
- Make sure that the regression test `ex2.ml` passes.
- Add the `ex2` target to `Makefile` and update `OREADME.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`, `OREADME.dr`.

Hint: this exercise could be very easy or very difficult, depending on how you choose the new operation and the normal form. The level of challenge is up to you.

Exercise 2 (Optional). Think of a more optimal normal form for the original `Lang` expressions, taking advantage of the commutativity of addition and performing arithmetic operations as much as possible (that is, on integer literals). Implement this optimization by normalization-by-evaluation.

Copy `tfintro` from the class repository into `tfintro3` in your repo, and make all the changes there. Don’t forget tests, and making sure they pass – and don’t forget to commit the `tfintro3` directory by the deadline.

Part II

Making the compiler

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

4 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. Compile the utility code by typing `make while` in `util` directory.

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 file contents is something other than a single integer, or the contained signed integer does not fit within 64-bits, the compiler must report an error. Otherwise, it is to produce 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.

4.1 Printing an integer, in Assembly

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. Here is a simple C program that prints a 64-bit integer, `-42`.

```
#include <stdio.h>
#include <stdint.h>

int main(void) {
    int64_t x = -42LL; // A sample integer
    printf("%lld", (long long int)x);
    printf("\nDone\n");
    return 0;
}
```

Our compiler has to create a similar code, but in Assembly. Looking at the code closely, we clearly see the finalization part (the last three lines). The first `printf`, for printing a 64-bit integer, probably could be separated in its own function; we only need to generate the call to it. All in all, we can partition the code into the part that deals with setting up and clean up, and contains the useful utilities like `print_int`.

```

#include <stdio.h>
#include <stdint.h>

// Main Tiger' function
extern void ti_main(void);

// Tiger' standard library
void print_int(int64_t x) {
    printf("%lld", (long long int)x);
}

int main(void) {
    ti_main();
    printf("\nDone\n");
    return 0;
}

```

One can expect this code to be roughly the same no matter what *Tiger*' program we are to compile. This part may remain in C: see `init.c` in the repo.

The part that our compiler is really responsible for can be represented by the following C function. Let's call it `ti_main`. It is invoked by the set-up/clean-up code `init.c`.

```

#include <stdint.h>

extern void print_int(int64_t);

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

```

Our compiler has to produce something like that, but in Assembly. To learn what assembly code to generate, let's make a C compiler generate it for us (by passing the flag `-S`) and look at the result. The cleaned and commented assembly code is as follows (see the file `clean_int.s`):¹²

```

# 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
    movq    $-42, %rdi

```

¹²The 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 *sigh*. We will have to adapt to it later.

```

# call the external function
    call    print_int
# restore the stack
    addq   $8, %rsp
# return
    ret

```

Most of the code is self-explanatory: see [CS107 \[2021a,b\]](#) for a brief but sufficient reference, and [Cloutier \[Ed.\]](#) for the complete reference. It is essentially the invocation of `print_int`, passing it the number to print. According to the x86-64 Application Binary Interface (ABI), the first argument to a function is passed in the register `rdi` (see [Fog \[2022\]](#)). Therefore, our `ti_main` loads the number into the register and invokes `print_int`. We also see the mysterious `subq` and `addq` instructions, adding and subtracting 8 from `rsp`, seemingly for no reason. We shall see the reason later, when we talk about the stack in more detail. For now, let's just accept that it is a part of function entry/exit code, again dictated by the ABI conventions.

With the above assembly code as a template, we write our first compiler: file `compiler0.ml`, which takes an input stream and produces the assembly code file.

I insist on writing type signatures of all top-level functions (perhaps except those extremely small), in one of the two styles (see the code).

The code takes the ‘template-based code generation’ to the heart. It is really just the template substitution, using `printf`. It is very similar to the familiar C `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.

```

let compile (ich:in_channel) (och:out_channel) : unit =
  Scanf.bscanf (Scanf.Scanning.from_channel ich) "%d" @@ fun n →
  Printf.fprintf och {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

```

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: `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.

4.2 Assembly as a DSL and writing a simple but real compiler

Let's look at the assembly as a DSL: as a simple language (see §3, DSL as a jargon). What words and phrases do we need to be able to write the simple `ti_main` assembly code in the previous section? 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 `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 `asm.mli`).

```
type instr                                (* abstract *)
val (@) : instr → instr → instr          (* concatenate instructions *)

val call : name → instr

type register                             (* abstract *)
val rdi : register

type operand                             (* abstract *)
val imm : int → operand                  (* immediate value *)
val reg : register → operand

val movq : operand → operand → instr

val make_function : name → instr → instr

val write_file : out_channel → instr → unit
```

Using this interface, the compiler code becomes

```
let compile (ich:in_channel) (och:out_channel) : unit =
  let n = 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 `comp_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.

Finally, we need to arrange to invoke `ti_main` and to implement `print_int`. That is the job of the ‘run-time’ system, so to speak, the file `init.c` that we talked about in §4.1. 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` (OCaml does this too: it is a popular technique in general):

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

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 §4.4) links everything together and builds the compiler: `Build/tigerc`. If we write the string “-42” into a file `prog1.tg` and invoke

```
Build/tigerc prog1.tg
```

we obtain the executable file `Build/a.out`, executing which prints `-42`.

The produced `a.out` executable is 20,584 bytes long (on Linux): 20KB of code just to print an integer.¹³ If we examine the produced `a.out` using `nm a.out` or `objdump -x -d 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.

4.3 Testing a compiler

Needless to say, we must test the compiler. We need both positive tests (correct programs should be accepted, compiled and run correctly) and negative tests (wrong programs should be rejected with an appropriate error message).

The first thing to test is the compiler invocation itself. The compiler, the `tigerc` command, expects one argument, the file name with the source program. Therefore, `tigerc` should complain if invoked without arguments (or if invoked with a non-existent file, for example). Let’s try: `Build/tigerc`.

Mostly we will be testing compilation. For example, the program `0` is correct and should compile. To verify, we create a temporary file, say, `/tmp/a1.tg` containing `0` and pass it to the compiler:

```
Build/tigerc /tmp/a1.tg
```

¹³Since it is a dynamically-linked executable, against the C standard library `libc`, the total code size is quite larger.

The command should finish without errors, producing `Build/a.out`, running which should print `0` followed by `Done`. On the other hand, the program `xxx` should clearly be rejected.

Running the tests by hand is very cumbersome. The file `util/expect.ml` implements a simple testing framework, in OCaml, with four main functions:

test_command *cmd* Execute the command *cmd* and return its output or the failure indicator, to be analyzed by `expect` or `expect_fail` below. Here, *cmd* is a list of strings: the head is the command itself (the file name), followed by arguments, if any.

test_it *inp cmdf* Execute the given command on the given input and return its its output or the failure indicator, to be analyzed by `expect` or `expect_fail` below. Here, *cmdf* is a function that accepts a file name and returns a list of strings: the command to run and its arguments; *inp* is a string. The function `test_it` creates a temporary file, writes *inp* to it and passes the file name to *cmdf*. It then runs the resulting command (specified as a list of string) as `test_command` above.

expect *expected res* Check that *res*, which should be the result of `test_it` or `test_command`, indicates the successful completion of the command, and check that its output matches the string *expected*.

expect_fail *res* Check that *res*, which is the result of `test_it` or `test_command`, indicates the failure of the command.

The functions `expect` and `expect_failure` crash the program if their check fails.

The earlier check that `0` is the correct program is hence automated as follows:

```
let tigerc = Filename.concat "Build" "tigerc"
let execf = Filename.concat "Build" "a.out"
let cmd fname = [tigerc; fname]
let _ =
  expect "" @@ test_it "0" cmd;
  expect "0\nDone" @@ test_command [execf]
```

This OCaml code does exactly what we did before by hand: creates a temporary file; writes the string `"0"` into it; runs `Build/tigerc` passing the file name as the argument; tests that command finishes successfully with no output; runs the command `Build/a.out` with no arguments; checks that it finishes successfully and outputs `0` and `Done`.

The following OCaml code verifies that the program `xxx` is indeed rejected:

```
let _ =
  expect_fail @@ test_it "xxx" cmd
```

The file `test_script.ml` is the test script containing the above tests (and a few more). It is executed by the build target `test` (see below). It should finish without errors, printing `"All Done"` at the end.

4.4 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).

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

For example, the former rule 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 `ocamllex`, `ocamlyacc` (for lexer and parser generation, resp.), `cc` for compiling C code and `link` for linking. For example,

```
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

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

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

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

is the rule that runs the listed test scripts, invoked in 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.

5.1 Languages, Grammars, Parsing

Let's start with a bit of terminology.

Language (programming or natural) A set of sentences (usually a countably infinite set)

Sentence A sequence, whose elements are called *characters* or *words*

Grammar A definition of a language, which can be viewed either as:¹⁵

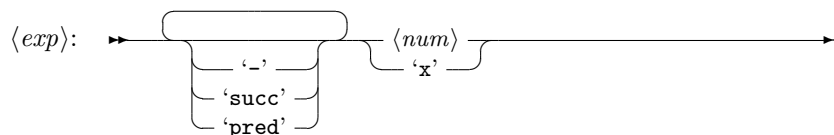
- A method to generate sentences of a language one-by-one; or
- A way to decide if an arbitrary sentence belongs to the language – is a member of the set of language sentences, – or does not. If a sentence belongs to the language, it is called *well-formed* (for that language); otherwise, it is *ill-formed*.

Remark 3. A bit of history. Describing a set of sequences (formulas, sentences, etc.) by the way of constructing them through applying so-called production rules to already constructed sequences (including the initial set: axioms) is called *Post System* – conceived by the mathematician and logician Emil Post in 1920s, but published only in 1941 [Urquhart \[2009\]](#). Type-checking rules we encounter in §16.1 are an instance of Post System. Context-Free Grammars (CFG) is another instance – proposed by the now famous linguist (at that time, just finished the graduate school) Noam Chomsky [Chomsky \[1956\]](#) for natural languages in 1956 based on Post work, which he likely learned from a textbook on logic. Chomsky's work caught the attention of Donald Knuth (who even took Chomsky's book "Syntactic structures" to read on his honeymoon in 1961); hence started Knuth's long line of research on CFG parsing and attribute grammars.

There are many notations for a grammar. We will be using so-called "railway tracks" – introduced by Niklaus Wirth, to describe Pascal language that he designed and developed [Jensen and Wirth \[1975\]](#).

The programming language we will be compiling in the present section is an extension of the language of §4 with unary operations: 'succ', 'pred' and unary minus. To make the language more interesting, we also add a single variable 'x', whose value is to come from the argument given to the compiled executable. This language is thus defined by the following railway-tracks grammar:

Specification 2 (Unary Language).



where $\langle num \rangle$ is a sequence of digits denoting a non-negative 64-bit integer.

¹⁵If we can also generate sentences *not* in the language, the two viewpoints become equivalent.

Think of a train that goes on the track from the starting point on the left to the final point on the right; ‘x’, ‘succ’, etc. are the ‘stations’. A sequence of stations it passes along the way is a sentence of the language. We obtain thus a way to generate the language. Here are a few sample sentences:

10 -10 -x - succ x

In short, our language has integer literals and the variable ‘x’, optionally preceded by any number of prefix operations ‘-’, ‘succ’ and ‘pred’. The language appears a bit odd, permitting expressions such as `pred succ - - succ x`. Parentheses would sure help. We shall get to it soon.

Parsing is determining how a language sentence, such as `-succ x`, may be obtained (or, ‘generated’) by the grammar. Concretely, for a railway track grammar: given a sequence of stations passed by, infer the exact path the train must have taken – or determine that the train could not have possibly passed these stations in this sequence.

This ‘exact path of the train’ is formally called a *parsing derivation*, or *parse tree*.

The obvious inference, or parsing, algorithm for generative grammars is to get the grammar to generate sentences of the language one by one, and see if one of them matches the given sentence. It is also obvious that this method may be very slow, or may even fail to terminate if the given sentence does not belong to the language and it is not possible to generate sentences in order of the increasing length. For very complex grammars, however, this generate-and-test method may be the only option. Fortunately, for simple grammars, much more efficient algorithms are possible, as we see next.

To get the computer to do parsing, we first need to find a way to represent parsing derivations – train paths – on computer. We may use the tagless-final style again, similarly as explained in §3. For example (see [parsing_intro.ml](#)):

```
module type Lang = sig
  type repr          (* the type of paths,
                     from some point through the end *)

  val int  : int64 → repr
  val varx : repr

  val neg  : repr → repr
  val succ : repr → repr
  val pred : repr → repr

  val observe : repr → unit          (* explained in the text *)
end
```

An OCaml value of type `repr` represents a path from a point that is directly reachable from the starting point – through the end. For example, `int 10!` represents a path that goes straight through the station ‘10’, without veering into side tracks. Likewise, `varx` represents the path that passes only by ‘x’. A turn around ‘succ’ is represented as `succ p`, where `p` stands for the path after the turn (which the train could have taken directly). Incidentally, the argument type of the `int` operation betrays one answer to the last homework. One should keep in mind that even in C, `int` is not necessarily 64 bits, even on a 64-bit platform. 64-bit integer data type has to be specified explicitly – in C, and in OCaml.

The `Lang` signature hence defines a DSL: the DSL of all paths from the beginning through the end of Spec. 2 – the paths that represent all and only well-formed sentences of our language (well-formed programs). Like the railroad track diagram, the `Lang` signature, hence, is also a grammar of our language: a ‘machine-readable’ grammar, so to speak.

Given a path, one may want to know which stations it passes through. In other words, given a parse tree, find the sentence it is the parse tree of – find the sentence generated by that derivation. Such a sentence is called the *yield* of a parsing derivation.

```
module Yield : Lang = struct
  type repr = string
  let int : int64 → repr = Int64.to_string
  let varx : repr = "x"
  let neg : repr → repr = fun p → "-" ^ p
  let succ : repr → repr = fun p → "succ" ^ p
  let pred : repr → repr = fun p → "pred" ^ p

  let observe : repr → unit = print_endline
end
```

Its module type is `Lang`, with `repr` kept abstract. We still need a way to observe the result: that’s why we added `observe` to the signature, which `Yield` implements (see also `observe` in §3.2).

A sample expression `neg (neg varx)` of `Lang` – the path that loops around ‘neg’ twice before going to ‘x’ and ending there – is interpreted and observed by `Yield` as the string `--x`. *Parsing is the converse*: read the string `--x` and build the expression `neg (neg varx)`, which can then be interpreted by various `Lang` interpreters (such as `Yield`). We shall soon see that a compiler is also a `Lang` interpreter.

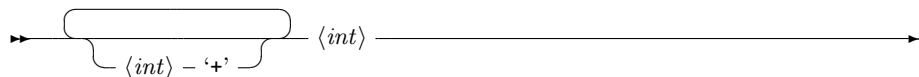
Parsing is the converse of yield-ing:
given a string, reconstruct a `Lang` expression that yields it.

5.2 LL(1) grammars and recursive descent parsing

What makes parsing complex are the many ‘forks’: the choices the train can make. Turn-arounds (repetitions) multiply the choices indefinitely.

Let's look however at our simple diagram, and check the first character of a station in every possible turn: '-', 's', 'p', 'x' or a digit. They are all different. Hence just by looking at the input string, such as --x, we can tell right away which turn the train has made – without any guessing. Our grammar is hence *predictive*: a choice to make can be predicted just by looking at the current symbol of the input string, without any trial-and-error. Technically, the grammar is called LL(1). It is the easiest to understand, unambiguous, grammar, which is also easy to parse.

Remark 4. Alas, not all grammars are LL(1): in fact, most programming language grammars are not. Take our grammar: if we replace $\langle num \rangle$ with $\langle int \rangle$, which may be a negative number, the grammar becomes ambiguous and no longer LL(1). A simple change can break an LL(1) grammar. Another example of a non-LL(1) grammar (in fact, not an LL-grammar at all) is the following grammar for the language of numbers and addition, considered in §3:



Let's look at parser1 in [parsing_intro.ml](#), which closely follows the track diagram: like the train driver looking at the map. LL(1) grammars are simple because they can be parsed using what is technically called *recursive descent*: mutually recursive functions, one for each part in the diagram.

```
let parser1 (module I:Lang) (ic: in_channel) : unit =
  let getc () = In_channel.input_char ic in
  let rec loop () = match getc () with
  | Some '-' → loop () ▷ I.neg
  | Some 's' → check "ucc"; loop () ▷ I.succ
  | Some 'p' → check "red"; loop () ▷ I.pred
  | Some 'x' → if getc () ≠ None then fail "junk_after_x"; I.varx
  | Some ('0'..'9' as d) → digit_to_int d ▷ read_num ▷ I.int
  | Some x → fail "bad_char:_%c" x
  | None → fail "unexpected_EOF"
  and
  check : string → unit =
    String.iter (fun c → if Some c ≠ getc () then fail "unexpected_char")
  and
  digit_to_int (d:char) : int64 = Char.(code d - code '0') ▷ Int64.of_int
  and
  read_num (accum: int64) = match getc () with
  | Some ('0'..'9' as d) → Int64.(add (mul accum 10L) (digit_to_int d)) ▷ read_num
  | None → accum
  | Some _ → fail "not_a_digit"
  in
  loop () ▷ I.observe
```

As we said, the parser is to take a string and return a Lang expression, like neg (neg x). How to return a tagless-final (that is, an OCaml expression), precisely? Recall that the only thing we may do with a tagless-final expression

is to interpret it, in some interpreter. Therefore, returning an expression is equivalent to receiving an interpreter (here, the first-class module `l`) and return the result of interpreting and observing the expression. In our case, we have the main track and something we called $\langle num \rangle$. They correspond to `loop` and `read_num`, resp. In a LL(1) grammar, when facing a turn we can always determine which to take by looking at the current symbol. Therefore, we never need backtracking.

Although the parser seems to work, it also has problems: it accepts numbers that are too large to fit into 64-bits, accepts the input like `succpred-x` that should probably be rejected, and does not permit leading, trailing or intervening spaces, which should probably be allowed (spaces increase readability). The file [parsing_intro.ml](#) illustrates all these problems.

5.3 Tokenizing

The problems come from a mis-understanding of the railroad diagram. The diagram deals with *tokens* (words, or ‘station names’) rather than individual characters. The input stream, or ‘file’, is a stream of characters however. Therefore, it first have to be converted to a stream of tokens, or tokenized. (The word ‘first’ may be misleading: tokenizing can be done on-demand rather than ahead-of-time, given a good enough implementation of streams.) The tokenizer, also called lexer, is in the file [tokenizer.ml](#). For our tokens, it can be implemented particularly elegantly, using the `groupBy` operation on streams.

This two-stage process – lexing (or tokenizing) followed by parsing proper – is the norm in compilers. The first reason is that separating out lexing makes parsers simpler. The parser proper does not have to deal with spaces and newlines, does not have to bother with maintaining the line count (used in error messages), etc. In fact, it may well be that the grammar on the character stream is not LL(1), but the grammar on the token streams is. Another reason for tokenizing is the quality of error messages: errors 12x or invalid characters can be reported right away and precisely.

The overall flow of data is represented by the picture Fig. 1. The input is a stream of characters, such as `'-', 's', 'u', 'c', 'c', ' ', 'x'`, which the lexer (tokenizer) converts it to a stream of tokens (words): `'-', 'succ', 'x'`. The parser turns that stream to the expression `neg (succ varx)` of the Lang DSL. We have so far one interpreter of Lang: `pretty-printer`.

The test cases (file [test_script.ml](#)) has the test cases for parsing.

We now extend the language with boolean literals (`'true'` and `'false'`) and the unary operations `'not'` and `'is_zero'`. The grammar thus becomes:

Specification 3 (Unary Language with Booleans).

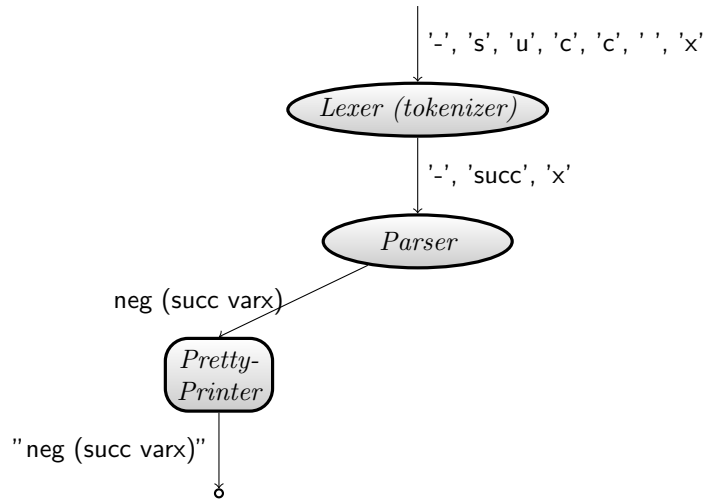
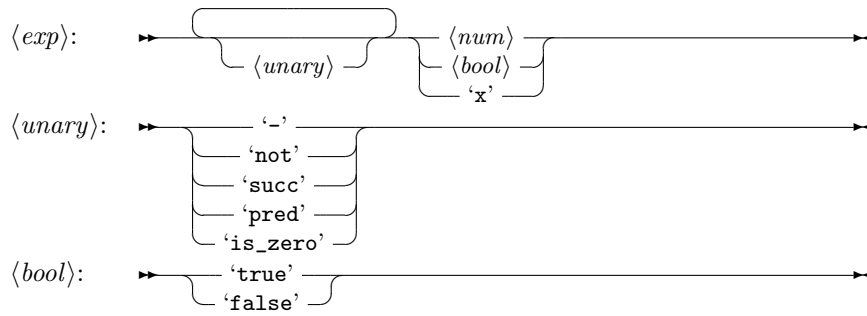


Figure 1: The overall data flow in the compiler (parsing only). Rounded-corner rectangles are interpreters.



Here is a sample expression: not is_zero x.

5.6 Types

In Ex. 8, you must have faced two problems. First, how to print the result, which could be either a number or `true/false`. How to choose which format to use? Second, what to do about Tiger’ programs like `- true, not 1, - is_zero x?` They are all well-formed according to the grammar, can be parsed and pretty-printed. Evaluating or compiling them, however, is puzzling. Do they even make sense? To be sure, one may try to give them some sense (e.g., to treat `true` as 1 and `false` as 0, as C or APL do). On the other hand, second-guessing the programmer tends to produce surprising results and sweep bugs under the rug, where they are quite more difficult to find.¹⁷ It is not for nothing that most programming languages do distinguish booleans and integers.

5.6.1 Negative View

Sentences that are well-formed yet meaningless are common in natural languages as well. Everyone probably had an experience in high school of a teacher marking a sentence as not making any sense, despite being grammatically correct. The famous example is “Colorless green ideas sleep furiously” by Chomsky. Another is the famous so-called liar’s paradox: “This sentence is false.” A sentence may also be meaningless because it refers to something that has not been introduced. ‘Making sense’ is a tough topic; in a Computing Theory class you will learn that there is no, and cannot be, a definite procedure (algorithm) to decide if sentences or programs make sense (except for very simple languages).

What we are left with, it seems, is to try to evaluate a given sentence anyway – perform its computation. If we encounter a roadblock such as trying to apply an operation to something outside of its range – e.g., dividing by zero, applying an integer operation to not-an-integer – report an error and exit. Needless to say, a meaningless program may as well finish normally, with a plausibly-looking result – but no connection to the truth. When a road-block does occur, however, letting the program crash and then restart it (reboot) is often an uncannily effective strategy. There are many situations however where it is detrimental, harmful or even disastrous. An impressive example of the latter is the first test flight of the Ariane 5 rocket. The guidance computer obtained a result that did not fit within allocated 16-bits and raised an overflow exception. The system was rebooted. When the rocket is ascending with engines at full thrust, it is not a good time for rebooting the guidance system.¹⁸

Therefore, it becomes important to *anticipate* roadblocks and take counter-measures *before starting the execution*. At the beginning of the 20th century, Bertrand Russell faced a similar problem with some logical formulas which, although well-formed, were paradoxical, such as the liar’s paradox. He wanted to find a way to rule out such a formula by just ‘looking over it’, without getting bogged down in its possible meaning. In 1903 (with further development in

¹⁷For a concrete, and rather scary example, see the entry DWIM in *The Jargon File* <http://www.catb.org/~esr/jargon/html/D/DWIM.html>.

¹⁸<https://www.bugsnap.com/blog/bug-day-ariane-5-disaster/>



Figure 3: Ariane 501: Ariane 5 first flight. Debris raining down after the self destruction, 4 June 1996. European Space Agency (ESA) Historic Photo Archive.

1908) he introduced a formalism for so-called “range of significance” of logical predicates and for deciding, at a glance, when a predicate is applied outside of its range. He called this “range of significance” *type*.

It is important to realize that ‘anticipation’ is rarely perfect. An anticipated rain may not happen, or rain may still fall with a sunny forecast. Types are designed to eliminate the latter error – continuing our analogy, if no rain is anticipated, no rain shall fall.

In a well-typed program, particular (explicitly specified) classes of errors shall not occur, *no matter what the input*:

“Well-typed programs do no wrong.”

The cost however is conservativity: some programs that make sense (or could make sense on some inputs) are judged problematic.¹⁹

Types hence play a *negative role*: rejecting programs that make no sense – or, to be more precise, rejecting programs that could not be judged meaningful and safe. One comes across many complaints about a type system restricting

¹⁹Therefore, common type systems may be called ‘pessimistic’: they err on the side of rejecting programs. If a program can “do wrong” even on a single input, it is judged ill-typed. There are also ‘optimistic’ type systems, quite less common. Somehow they are limited to logic programming and Erlang.

programmer’s flexibility and ‘freedom’. This is like complaining about a fire alarm being annoying: annoying is what a fire alarm is designed to be.

5.6.2 Positive View

Types play also a *positive role*. “Well-typed programs do no wrong” is a strong positive statement, and hence very valuable: Since we are sure particular errors shall not occur in our program once it passed the type check, we do not have to guard against such errors, and do not have to think about recovery. For example, the code generator does not have to worry about compiling successor on booleans: such case is simply impossible. The code generator may also assume that a register that contains a boolean has only two possible pre-defined bit patterns, and generate optimal instruction sequences accordingly. Types hence lead to simpler, and more efficient code.

Returning to Ex. 8: the result of an integer Tiger’ program is to be printed as an integer; a boolean result is to be printed as “true” or “false”. If the choice of which function to call at the end of the program – `print_int` or `print_bool` – were to be done at run-time, we would have to distinguish booleans and integers at run time and add code to each operation to check and convert the operands. It would greatly help the performance if all these checks and choices were done at compiler time instead. Types help make this choice. Using types to guide the selection of instructions in a compiler is the reason types were first introduced in programming languages: see [Martini \[2016\]](#).

It is also the reason types were introduced into C: “For all these reasons, it seemed that a typing scheme was necessary to cope with characters and byte addressing, and to prepare for the coming floating-point hardware. Other issues, particularly type safety and interface checking, did not seem as important then as they became later.” [Ritchie \[1993\]](#). The floating-point hardware – an upgrade to the PDP computer used to develop C and Unix – made possible floating-point computations. There was a temptation however to use the same syntax, like `x*3 + 1`, for both integer and floating-point arithmetic expressions. Since machine instructions for integer and floating-point operations differ, there has to be some way to compile such *overloaded* expressions and select appropriate instructions. For more detail on overloading, see Rem. §8.

To emphasize, types are used at compile time, to guide code generation and reject programs deemed unsafe or meaningless. They are not represented at run-time.²⁰ Unsafe may mean liable to raise a run-time exception – or interfering with another program or misusing data. This broadly-speaking “security” point of view is expressed in the Reynolds slogan [Reynolds \[1974\]](#)

types are syntactical discipline for enforcing level of abstraction

See [Martini \[2016\]](#) for the development of this point of view, mainly due to J.H.Morris.

²⁰Formally, this is called “type erasure”.

Remark 5. It took quite a while to realize that the classification of values to select instructions, and the rejection of meaningless/unsafe programs are two aspects of the same notion: types. It is rather surprising therefore that there was a person who recognized these connections, back in late 1940s: logician H.B. Curry. The first technical use of “type” in programming, in 1949, was also due to him [Curry \[1949\]](#). Deeply unfortunately, his papers seem to have been forgotten. It is not until nine years later, in 1958, that the term ‘type’ as classes of values started to be used [Martini \[2016\]](#). The connection to logic had to wait another decade to be rediscovered.

Still, the paper that begat the Hindley-Milner type system mentioned Curry on many occasions. It seems its author, J. Roger Hindley, was greatly influenced by Curry. One may say that there is a sort of connection from Curry’s early insights to the modern type systems.

5.6.3 Types and grammars

Types are similar to grammar: both give a procedure to judge and accept a sentence, or reject it as being ill-formed (or ill-typed). Crucially, in both cases the procedure is ‘superficial’: the judgement is done only by looking at the shape or form of sentences without deeply considering their meaning. In principle, type judgments may be incorporated into a grammar. It is rarely a good idea, because the resulting grammar becomes very difficult to understand, and also too slow to parse.²¹ ²² Just like separating lexing from parsing, it makes sense to perform type checking as a separate step.

5.7 Compiling integer and boolean unary expressions

All the code for this section is in the directory [step22](#) in the class repo.

Overall, compilation now looks as in [Fig. 4](#). We have added the compiler phase: which is another interpreter for `Lang`, performing type-checking, and then passing the parsed expression to the code generator.

Type checking can be performed as a sort of evaluation: Abstract Interpretation [Jones and Nielson \[1994\]](#). The type checker (‘compiler’ in [Fig. 4](#)) is hence written as an interpreter, of `Lang`. Actually, the compiler of [Fig. 4](#) is a product²³ of two interpreters: the type-checking interpreter and code-generation interpreter, or the higher-level assembly (what used to be called ‘compiler’ in [Fig. 1](#).) They interpret the same expression in ‘parallel’ so to speak (or, in lockstep). The compiler code [`compiler.ml`](#) shows the two roles of types clearly: the type checking interpreter checks that operations receive the arguments in their ‘range of significance’. The result, the type of the whole program, is used

²¹Programming language grammars are usually context-free or mostly context-free. Type systems, however, are often context-sensitive.

²²For example, one may compare for equality two integer or two boolean expressions, but not an integer with a boolean. Expressing this constraint in a grammar leads to the explosion of non-terminals.

²³that is, a pair

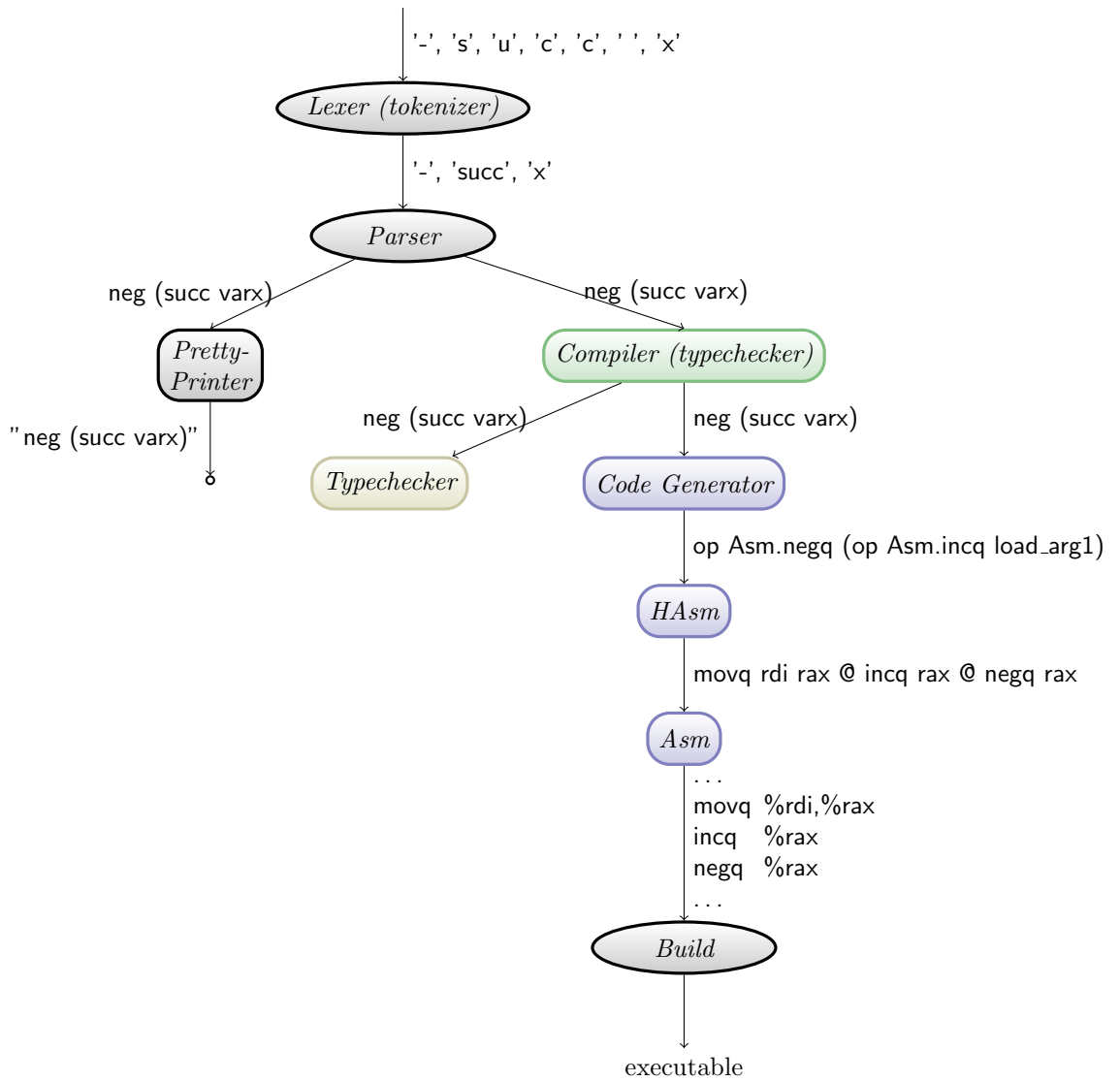


Figure 4: Data flow after adding type checker

at the end, in observe, to select the function to print the result: either `print_int` or `print_bool`.

The type-checking interpreter evaluates an expression as type (`trepr` – type representation – in the code.) Here is an example of type checking as a (non-standard) evaluation:

```
not is_zero succ x
```

```
~~ not is_zero succ int
~~ not is_zero int
~~ not bool
~~ bool
```

Incidentally, we cannot actually evaluate this expression in the standard way, since `x` is not known at this point. Later on, in §16.1, we see another, conventional notation for expressing type systems.

We have also introduces HAsm: a higher-level assembly, meant to abstract over the assembly details, in particular, how exactly the generated code is arranged and where the expression result really is. Before, we generated code keeping the result in the register `rax`. We still do that, but we keep the `rax` and such details from the code generator. The code generator still uses Assembly instructions. It should not use `movq` however, since the exact content of the registers is abstracted out. One can think of HAsm as a sort of a simple virtual machine.

22 Projects

1. Take a signature like `Lang` (actually, the compiled `lang.cmi` file) and automatically build a parser for that language. The signature needs to be annotated with the names of terminals. This is what yacc grammar essentially does. One needs to draw the distinction between concrete and abstract syntaxes. See §5.1.
2. Change `asm.ml` to output the assembly code in Intel format
3. Currently we represent a boolean value as a 64-bit word: 0 meaning false and 1 meaning true. GCC on x86-64 uses a slightly different representation: still a 64-bit word, but only the least significant 8 bits are taken into account. Do you see the advantage of it? Change the compiler to use this representation. All earlier tests should pass. As an extra challenge, think of (micro- and medium-size) benchmarks to test how much of a difference this new representation makes.
4. Python has popularized *generators*: functions that yeild. Such generators, however, first appeared in the programming language CLU (where they were called iterators) in mid 1970s [Liskov, 1992, §3.10]. They were then borrowed in some versions of C, notably MetaWare’s High C Compiler:³⁷

```
// This is C, with High C extensions
void Upto(int Lo, int Hi) → (int) { // Generator Upto
  int i = Lo;
  while(i ≤ Hi)
    yield (i++);
}
for i ← Upto(1,10) do
  printf("%d_squared_is_%d\n", i, i*i);
```

Implement the generators, by desugaring into ordinary function calls. For hints, see Liskov’s paper on CLU [Liskov [1992]] or the article on High C referenced above (or, better, try to find the book mentioned in the article).

5. Much better memory allocation. One particular direction: notice the similarity between unique variable renaming (Ex. 15) and allocating memory for those variables.
6. A better representation for strings, such as Phil Bagwell’s persistent data structures. See §12.
7. Functions returning multiple arguments
8. Implement the lambda-lifting technique. See §16.3.

³⁷<https://duriansoftware.com/joe/the-lost-language-extensions-of-metaware's-high-c-compiler>

9. Implement various micro-optimizations. First, when loading an immediate value into a register, say, `rax`, and that value happens to be zero, instead of `movq $0,%rax` emit `xorq %rax,%rax`, which is *far* more efficient. Intel CPU recognizes such instruction already during decoding (it is not even need to be executed, strictly speaking). Second, multiplication by constants is often done without `imul` instruction: see, for example, how GCC compiles `x * (-7)`. Division by a constant is always done without using the division instruction, which is slow and quite awkward to use. Implement such optimizations. As a further challenge, discover other micro-optimizations (e.g., by analyzing GCC assembly output or search the Web) and implement them. Ideally you will come up with a framework which makes implementing (and adding) micro-optimizations easy.

References

- Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN 0-521-52745-1. doi: 10.1017/CBO9780511811449. URL <http://www.cs.princeton.edu/~appel/modern/ml/>.
- Noam Chomsky. Three models for the description of language. *IRE Trans. Inf. Theory*, 2(3):113–124, 1956. doi: 10.1109/TIT.1956.1056813.
- Michael R. Clarkson. OCaml programming: Correct + Efficient + Beautiful. <https://cs3110.github.io/textbook/cover.html>, 2022. Cornell U. course; videos available.
- Félix Cloutier (Ed.). x86 and amd64 instruction reference. <https://www.felixcloutier.com/x86/index.html>, 9 September 2022.
- staff CS107. CS107 x86-64 reference sheet. <https://web.stanford.edu/class/cs107/resources/x86-64-reference.pdf>, 2021a.
- staff CS107. Guide to x86-64. <https://web.stanford.edu/class/archive/cs/cs107/cs107.1224/guide/x86-64.html>, 2021b.
- Haskell B. Curry. On the composition of programs for automatic computing. Technical Report Memorandum 10337, Naval Ordnance Laboratory, 1949.
- Nikita Danilenko. And... Action! - Monoid actions and (Pre)orders. In Michael Hanus and Ricardo Rocha, editors, *Declarative Programming and Knowledge Management - Declarative Programming Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, September 11-13, 2013, Revised Selected Papers*, volume 8439 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2013. doi: 10.1007/978-3-319-08909-6_6.
- Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM 2000: International Summer School on Applied Semantics, Advanced Lectures*, number 2395 in *Lecture Notes in Computer Science*, pages 137–192. Springer, 2002.
- Andrei Petrovich Ershov. On programming arithmetic operators. *Doklady Akademii Nauk*, 118(3):427–430, 1958a.
- Andrei Petrovich Ershov. On programming of arithmetic operations. *Comm. ACM*, 1(8):3–6, August 1958b.
- Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 47–58. ACM Press, May 22-25 1995. doi: 10.1145/223784.223789.

- Agner Fog. Calling conventions: for different C++ compilers and operating systems. http://agner.org/optimize/calling_conventions.pdf, 5 August 2022.
- Abdulaziz Ghuloum. An incremental approach to compiler construction. In Robert Bruce Findler, editor, *Proceedings of the 7th Workshop on Scheme and Functional Programming*, number TR-2006-06 in Tech. Rep., pages 27–37. University of Chicago, 17 September 2006. URL <http://scheme2006.cs.uchicago.edu/>.
- Michael Gordon, Robin Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, Tucson, Arizona, January 23–25, 1978. ACM SIGACT-SIGPLAN. URL <http://www-public.tem-tsp.eu/~gibson/Teaching/CSC4504/ReadingMaterial/GordonMMNW78.pdf>.
- C. A. R. Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Department of Computer Science, Stanford University, December 1973. URL <http://flint.cs.yale.edu/cs428/doc/HintsPL.pdf>. <https://purl.stanford.edu/vr655dy1064>.
- Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, December 1996.
- K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer-Verlag, Berlin, 1975.
- Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, pages 527–629. Oxford University Press, 1994. <http://www.diku.dk/forskning/topps/bibliography/1994.html#D-58>.
- Oleg Kiselyov. Elementary tutorial on Normalization-by-Evaluation. <http://okmij.org/ftp/tagless-final/NBE.html>, August 2020.
- Oleg Kiselyov. Free variable as effect, in practice, 27 December 2023. URL <http://arxiv.org/abs/2312.16446>. HOPE Workshop 2023.
- Oleg Kiselyov. Tagless-final style. <https://okmij.org/ftp/tagless-final/>, 2024.
- Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.
- Charles H. Lindsey. A history of ALGOL 68. In John A. N. Lee and Jean E. Sammet, editors, *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*, pages 97–132. ACM, 1993. doi: 10.1145/154766.155365.

- Barbara Liskov. A history of CLU. Technical Report 561, MIT LCS, April 1992. URL <http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-561.pdf>.
- Simone Martini. Several types of types in programming languages. In *History and Philosophy of Computing, HAPOC 2015*, volume 487 of *IFIP Advances in Information and Communication Technology*, pages 216–227. Springer, 2016. URL <https://arxiv.org/abs/1510.03726>.
- Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.
- Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21th Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Somerset, NJ, 1983. Association for Computational Linguistics. URL <http://www.aclweb.org/anthology/P83-1021>.
- John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Programming Symposium: Proceedings, Colloque sur la Programmation*, number 19 in *Lecture Notes in Computer Science*, pages 408–425, Berlin, 9–11 April 1974. Springer.
- Dennis Ritchie. The development of the C language. In John A. N. Lee and Jean E. Sammet, editors, *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*, pages 201–208. ACM, 1993. doi: 10.1145/154766.155580. URL <http://csapp.cs.cmu.edu/3e/docs/chistory.html>.
- Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995. URL <http://arxiv.org/abs/cmp-lg/9404008><http://www.eecs.harvard.edu/~shieber/Distrib/Sources/deductive-parser/>.
- Alasdair Urquhart. Emil Post. In Dov M. Gabbay and John Woods, editors, *Logic from Russell to Church*, volume 5 of *Handbook of the History of Logic*, pages 617–666. Elsevier, 2009. doi: 10.1016/S1874-5857(09)70016-0.
- John Whittington. *OCaml from the Very Beginning*. Coherent Press, 2013. ISBN ISBN-13: 978-0957671102; ISBN-10: 0957671105. URL <https://ocaml-book.com/>.
- Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi: 10.1145/1993498.1993532.