Compiler Design Lab 1
Summer 2025

Instructor: André Platzer
TAs: Enguerrand Prebet, Hannes Greule, Darius Schefer, Julian Wachter

Start: 05.05.2025
Tests due: 12.05.2025
Due: 19.05.2025

Introduction

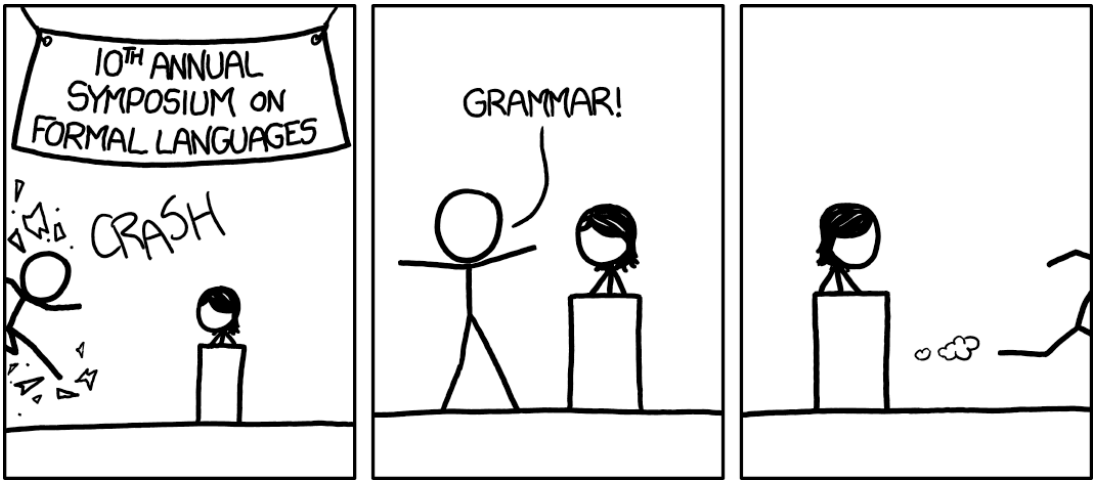
In this course, we will build not just one compiler, but several! Each compiler will build on the previous one, so careful thought and design are very important in the first labs. You *will* be re-using and re-writing code.

Your task in this lab is to translate a small programming language, called **L1**, into executable binaries, targeting x86-64 assembly. To help you get started and give you a scaffold to work on, we provide you with an unfinished compiler for this language. This starter code targets a very simple "abstract assembly" language with an infinite number of registers and a very simple instruction set, just including arithmetic instructions. You do not have to use our starter code — you do not even have to use any of our supported languages — but if you haven't dealt with compilers before, we highly recommend using it. Beware though: Using the starter code is no substitute for *understanding* the starter code, i.e. the basic structure of a typical compiler. This is essential, because you will be editing every stage of the compiler in future labs, which will include any starter code that you're depending on.

The main focus in this lab is to implement instruction selection and register allocation. If you don't use the starter code, you also need to implement the previous steps (lexing, parsing, semantic analysis, ...). You also need to assemble and link your assembly output, e.g. using `gcc`.

And now welcome to a short organization paragraph! If you are reading this, you have already found the course page and moodle. You can work on your compiler either on your own or in teams of two. The compiler submission is managed by `crow` and explained in more detail in section Project Requirements (p. 4) .

To emphasize again, *all the projects in this course are cumulative.* Therefore, falling behind in Lab 1 would be disastrous. Please get an early start, and remember that we're here to help. We have also created a forum in the moodle course for any further questions.



xkcd 1090: Formal Languages

L1 Syntax

The syntax of **L1** is defined by the context-free grammar shown in Listing 1. The precedence of unary and binary operators is given in Table 1.

Lexical Tokens

The concrete Syntax of **L1** is based on ASCII character encoding.

Whitespace and Token Delimiting

In **L1**, whitespace is either a space, horizontal tab (`\t`), carriage return (`\r`), or linefeed (`\n`) character in ASCII encoding. Whitespace is ignored, except that it terminates tokens. For example, `+=` is one token, while `+ =` is two tokens.

Comments

L1 source programs may contain C-style comments of the form `/* ... */` for multi-line comments and `//` for single-line comments. Multi-line comments may be nested (and of course the delimiters must be balanced).

Reserved Keywords

The following are reserved keywords and cannot appear as a valid token in any place not explicitly mentioned as a terminal in the grammar:

```
struct if else while for continue break return assert true false NULL print read alloc alloc_array int bool void char string
```

Many of these keywords are unused in **L1**. However, the specification treats these as keywords to maintain forward compatibility with future labs.

$$
\begin{array}{lcl}
\langle \text{program} \rangle & ::= & \textbf{int} \quad \textbf{main} \quad () \quad \{ \ \langle \text{stmts} \rangle \ \} \\
\langle \text{stmts} \rangle & ::= & \varepsilon \\
& | & \langle \text{stmt} \rangle \ \langle \text{stmts} \rangle \\
\langle \text{stmt} \rangle & ::= & \langle \text{decl} \rangle \ ; \\
& | & \langle \text{simp} \rangle \ ; \\
& | & \textbf{return} \ \langle \text{exp} \rangle \ ; \\
\langle \text{decl} \rangle & ::= & \textbf{int} \quad \textbf{ident} \\
& | & \textbf{int} \quad \textbf{ident} \ = \langle \text{exp} \rangle \\
\langle \text{simp} \rangle & ::= & \langle \text{lvalue} \rangle \ \langle \text{asnop} \rangle \ \langle \text{exp} \rangle \\
\langle \text{lvalue} \rangle & ::= & \textbf{ident} \\
& | & (\ \langle \text{lvalue} \rangle \ ) \\
\langle \text{exp} \rangle & ::= & (\ \langle \text{exp} \rangle \ ) \\
& | & \langle \text{intconst} \rangle \\
& | & \textbf{ident} \\
& | & \langle \text{exp} \rangle \ \langle \text{binop} \rangle \ \langle \text{exp} \rangle \\
& | & \langle \text{unop} \rangle \ \langle \text{exp} \rangle \\
\langle \text{intconst} \rangle & ::= & \textbf{decnum} \\
& | & \textbf{hexnum} \\
\langle \text{unop} \rangle & ::= & \texttt{-} \\
\langle \text{asnop} \rangle & ::= & \texttt{=} \ | \ \texttt{+=} \ | \ \texttt{-=} \ | \ \texttt{*=} \ | \ \texttt{/=} \ | \ \texttt{\%=} \\
\langle \text{binop} \rangle & ::= & \texttt{+} \ | \ \texttt{-} \ | \ \texttt{*} \ | \ \texttt{/} \ | \ \texttt{\%} \\
\textbf{ident} & ::= & \texttt{[A-Za-z\_] [A-Za-z0-9\_]*} \\
\textbf{decnum} & ::= & \texttt{0} \ | \ \texttt{[1-9] [0-9]*} \\
\textbf{hexnum} & ::= & \texttt{0 [xX] [A-Fa-f0-9]+}
\end{array}
$$

Listing 1: Grammar of **L1**, nonterminals in ⟨brackets⟩, terminals in **bold**

Operator	Associates	Class	Meaning
`-`	right	unary	unary negation
`* / %`	left	binary	integer multiplication, division, modulo
`+ -`	left	binary	integer addition, subtraction
`= += -= *= /= %=`	right	binary	assignment

Table 1: Precedece of operators, from highest to lowest

L1 Static Semantics

The **L1** language does not have a very interesting type system. Most constraints imposed by the type system are for the time being imposed by the grammar instead.

Declarations

Though declarations are a bit redundant in a language with only one type and no interesting control flow constructs, we require every variable in the function to be declared (with the correct type, in this case `int`) before being used, although statements and declarations can be mixed. We do this to ensure that the valid **L1** programs are forward compatible with respect to future labs.

Initialization Checking

Programs that attempt to reference a variable before assigning to it should cause the compiler to generate a compile-time error message.

Return Checking

To maintain forward compatibility, we require that **L1** programs contain a return statement, but not necessarily only once or at the last statement.

Constants

Decimal constants `c` in a program must be in the range $0 \leq c \leq 2^{31}$ where $2^{31} = -2^{31}$ according to modular arithmetic. Hexadecimal constants must fit into 32 bits (`0x00000000` - `0xffffffff`).

L1 Dynamic Semantics

Statements have the obvious operational semantics, although there are subtleties regarding the evaluation of expressions. Each statement is executed in turn. To execute a statement, the expression on the right-hand side of the assignment operator is evaluated, and then the result is assigned to the variable on the left-hand side, according to the type of assignment operator. The meanings of the special assignment operators are given in Listing 2, where `x` stands for any identifier and `e` for any expression.

```
x += e  ≡  x = x + e
x -= e  ≡  x = x - e
x *= e  ≡  x = x * e
x /= e  ≡  x = x / e
x %= e  ≡  x = x % e
```

Listing 2: Assignment operations in **L1**

The result of executing an **L1** program is the value of the expression in the program's `return` statement. As in `C`, the returned value is used as the exit code of the program and on Linux only the least significant byte is used[1].

Integer Operations

The integers of this language are in two's complement representation with a word size of 32 bits. Through sheer luck, the definitions of **L1**'s operators align with their counterparts in x86-64 assembly. Specifically, addition, subtraction, multiplication, and negation wrap around and never raise an overflow exception. In contrast, division `i / k` and modulus `i % k` are required to raise a `divide` exception if either `k = 0` or the result is too large or too small to fit into a 32 bit word in two's complement representation.

[1] See https://www.man7.org/linux/man-pages/man2/wait.2.html, `WEXITSTATUS`

The division `i / k` returns the truncated quotient of the division of `i` by `k`, dropping any fractional part. This means it always rounds towards zero.

The modulus `i % k` returns the remainder of the division of `i` by `k`. The modulus has either the same sign as `i` or is 0, and therefore `(i / k) * k + (i % k) = i`.

Project Requirements

For this project, you are required to hand in a complete working compiler for **L1** that produces correct and executable target programs for x86-64 Linux machines.

Especially for your sake, we also recommend that you document your code. You do not need to reinvent the wheel; we encourage the use of existing tools such as lexer and parser generators, as well as collection or graph implementations. Should you use publicly available libraries, you are required to indicate their use and link to their source in the README file. If you are unsure whether it is appropriate to use a specific tool or library, please discuss it with course staff.

Your compiler and test programs must be formatted and handed in via `crow` as specified below. For this lab, you must also write and hand in at least ten test programs. Please refer to What to Turn in (p. 5) for details.

Repository setup

For feedback and grading, your compiler is automatically built by the submission system `crow` at https://compiler.vads.kastel.kit.edu. To ensure this works correctly, you should set up a public git repository. A good start is the starter code template, if you want to use our starter code. If you want to keep your repository private, please contact us and we will provide you with an SSH key to add to your repository. `crow` has a dedicated GitHub integration using a GitHub app, which automatically synchronizes the `crow` test results with commits on GitHub. This integration can be enabled in the `Repository` tab of `crow`. We recommend you either use this integration, or build your own using the provided tokens, to automatically test every pull request and commit you make against `crow`'s test corpus. This will help you prevent accidental regressions and shows you where you stand directly in your IDE or repository.

In order for `crow` to execute your code, your repository needs to adhere to a common standard for building your compiler and executing it. In particular, we require the following files to be present *and executable*[2]:

- `/build.sh`, an executable file (we recommend a `bash` or `sh` script with a "shebang") that builds your compiler. This command will automatically be executed by `crow` after cloning your repository. Its output is persisted for all tests, so everything you create here will still be available when `crow` calls `/run.sh`.
- `/run.sh`, an executable file (we recommend a `bash` or `sh` script with a "shebang") that runs your compiler. The first positional argument of this executable is the *input file*. The last positional argument of this executable is the desired *output file*. Your compiler must place its generated artifact at this location, if it was tasked with generating a binary. For example, running `/run.sh foo.c foo` compiles the file `foo.c` and stores the result in an *executable* file `foo`.

Test Files

Test files can either be created directly in `crow` or managed locally as markdown files. `crow` provides a command line client, `crow-client`, which allows interfacing with `crow`. The client currently allows you to download all tests, run them locally and upload new or existing ones. We suggest that you create at least your first test in the online editor of `crow` (found on the tests page) to find out what the markdown format looks like.

As writing tests needs to be relatively flexible, `crow` uses general *test modifiers*. These modifiers can either configure the execution of your compiler, the compiled binary, or verify its result. If the validation of your compiler invocation fails, `crow` will return an error and skip executing the binary. Otherwise, if any binary modifier is configured, `crow` will continue on executing the compiled binary. If the binary also passes validation, the test is successful.

`crow` currently knows the invocation modifiers outlined in Table 2.

[2] If you are on Windows, you might need to use `git update-index --chmod=+x path/to/file`

Modifier	Argument	Description
`Argument string`	*short string*	An argument to the **compiler**, such as `--compile`
`Argument file`	*long string*	The text you enter is written to a file and the file name passed to the **compiler**. Passing an input `c` file to the compiler is done using this modifier.
`Program input`	*long string*	Passes input on the standard input stream (“`stdin`”) to the **binary**. This is used to mimic user interactions.
`Program output`	*long string*	The output of the **binary** must match the given string.
`Should succeed`		The **compiler** or your **binary** exits with exit code `0`.
`Should fail`	`Parsing`	The **compiler** should fail while lexing/parsing the input program.
`Should fail`	`Semantic analysis`	The semantic analysis phase of your **compiler** should fail on the input.
`Should crash`	`Floating point exception`	Your **binary** crashes with signal `SIGFPE`.
`Should crash`	`Segmentation fault`	Your **binary** crashes with signal `SIGSEGV`.
`Exit code`	`0-255`	Your **binary** exits with the given exit code, i.e. returns this value from `main`.

Table 2: The currently implemented test modifiers in crow.

Runtime Environment

`crow` uses a docker container when executing your project. Currently, it is based on the latest `archlinux` version with common development software such as `gcc`, `make`, `java` and `ghc` installed. If you use any other language and find that `crow` can not compile it yet, please report what software you are missing in the Build problems thread in the `crow` forum. Please also report any other problems you encounter that might need assistance in that forum 🐞.

Exit codes

Tests in `crow` typically assume your compiler exits *successfully*. But what does this actually mean and what does a failing invocation look like? `crow` uses the program exit code to determine success. To help us all write sensible tests, `crow` ships with a few preset exit codes imbued with meaning, which are available in the test creation page or the markdown test files.

For your **compiler** the following applies:

- exit code `0` indicates success
  `Should succeed` in `crow`
- exit code `42` indicates that the code was rejected by your lexer or parser
  `Should fail > Parsing` in `crow`
- exit code `7` indicates that the code was rejected by your semantic analysis
  `Should fail > Semantic analysis` in `crow`
- any other exit code indicates a general unexpected failure

For your **binary** the following applies:

- exit code `0` indicates success
  `Should succeed` in `crow`
- killed by signal
  - `SIGFPE` indicates a division by zero
    `Should crash > Floating point exception` in `crow`
  - `SIGSEGV` indicates a null pointer dereference. This is not yet relevant for you.
    `Should crash > Segmentation fault` in `crow`
- any other exit code indicates a non-zero return value from the binary’s main function
  `Exit code > [code]`

What to Turn in

- Test cases (deadline: 12.05.)
  - Upload at least 10 test cases to `crow`; two of which must fail to compile, two of which must generate a runtime error, and two of which must execute correctly and return an exit code.

- Your compiler (deadline: 19.05.)
  - You can either submit your code manually in `crow` or rely on its heuristics. For the heuristic, `crow` sorts your commits by `(<passing test count> DESC, <commit date> DESC)` and picks the first. You always see which commit `crow` currently selected on the home page.

Notes and Hints

For pointers on register allocation and code generation, please refer to the lecture notes. The optional textbook also contains further information.

Starter Assembly

As you need to generate assembly, we provide a most basic example that should help you get started. If you follow this template, make sure your main method in the generated assembly is named `_main` (note the underscore).

You can switch to Intel syntax by inserting `.intel_syntax noprefix` at the top and rewrite the template, and your emitted assembly, to Intel syntax.

```
.global main
.global _main
.text

main:
call _main

; move the return value into the first argument for the syscall
movq %rax, %rdi
; move the exit syscall number into rax
movq $0x3C, %rax
syscall

_main:
; your generated code here
```

Development Environment

We assume you are using Linux. If you are running Windows, consider using WSL. If you are running macOS, you can either install `gcc` using Homebrew or figure out how to get Apple's `clang` to behave.

Assembling & Linking

You can use `gcc` to get from your assembly output to an executable. You are free to invoke it directly from your compiler or from the `run.sh` script, but having it as a part of your compiler might make debugging easier. The relevant syntax is `gcc <input> -o <output>`, where `<input>` denotes your generated assembly file (with a `.s` file extension!) and `<output>` denotes the name of the executable you want to generate.

Supported Programming Languages

This course does not require you to use any specific programming language to implement your compilers. However, we cannot provide starter code for every programming language in existence and have therefore only distributed starter code for Java and Haskell. For both languages, the starter code ships with a `README.md`, which will help you understand the existing pieces. You will also find more material there about algorithms and implementations. The starter code can be found here: https://github.com/LS-Lab/Compilers-course-code-template.

If you want to use any other language, you might need to tell us how to package it in the runtime image `crow` uses. Please post a message in the `crow` forum on moodle :)

Happy Coding!