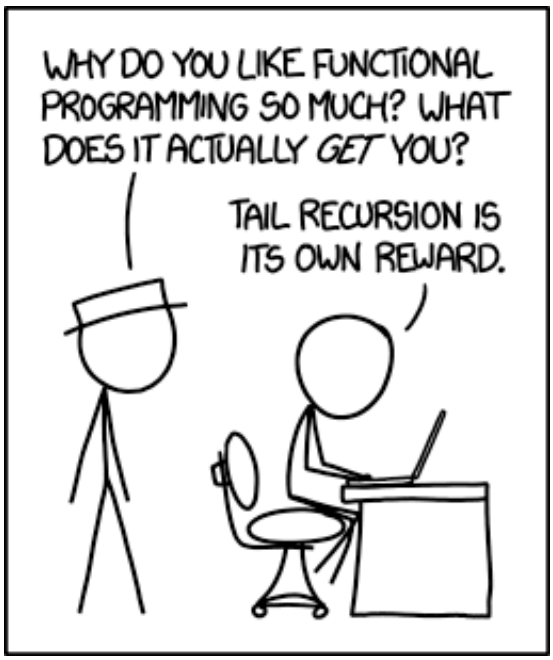
Compiler Design Lab 3
Summer 2025

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

Start: 16.06.2025
Tests due: 23.06.2025
Due: 30.06.2025

Introduction

In this lab you will upgrade your **L2** compiler to a compiler for the language **L3**, which supports defining and calling functions. To implement this, you will once again have to touch all phases of your existing compiler. With the newly gained functions, you can write more elaborate and interesting code, especially using recursion[1]. Some reserved keywords, notably `print`, now also make another appearance: As built-in functions they allow you to communicate with the outside world.



xkcd 1270: Functional

[1] See 1 or here

L3 Syntax

The concrete syntax of **L3** is based on ASCII character encoding of source code.

Lexical Tokens

The reserved keywords in **L3** are almost the same as in **L2**, with only `flush` added:

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

L3 also introduces the terminal `,` which is used in parameter lists and argument lists.

Whitespace and Token Delimiting

In **L3**, 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. Note that whitespace in not a *requirement* to terminate a token. For instance, `()` should be tokenized into a left parenthesis followed by a right parenthesis according to the given lexical specification. The lexer should produce the longest valid token possible. Therefore, `+=` is one token while `+ =` is two tokens.

Comments

L3 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.)

Grammar

The syntax of **L3** is defined by the context-free grammar in Listing 1. Ambiguities in this grammar are resolved according to the operator precedence table in Table 1. There is one leftover ambiguity in this grammar, commonly referred to as the "dangling-else" problem. It is handled just like in **L2**: we attribute the `else` to the closest `if` without an `else`.

Operator	Associates	Meaning
`()`	n/a	explicit parentheses
`! ~ -`	right	logical not, bitwise not, unary minus
`* / %`	left	integer times, divide, modulo
`+ -`	left	integer plus, minus
`<< >>`	left	(arithmetic) shift left, right
`< <= > >=`	left	integer comparison
`== !=`	left	overloaded equality, disequality
`&`	left	bitwise and
`^`	left	bitwise exclusive or
`|`	left	bitwise or
`&&`	left	logical and
`||`	left	logical or
`? :`	right	conditional expression
`= += -= *= /= %= &= ^= |= <<= >>=`	right	assignment operators

Table 1: Precedence of unary and binary operators in **L3**, from highest to lowest.

⟨program⟩ ::= ε | ⟨function⟩ ⟨program⟩

⟨function⟩ ::= ⟨type⟩ **ident** ⟨param-list⟩ ⟨block⟩

⟨param⟩ ::= ⟨type⟩ **ident**

⟨param-list-follow⟩ ::= ε | **,** ⟨param⟩ ⟨param-list-follow⟩

⟨param-list⟩ ::= **(** **)** | **(** ⟨param⟩ ⟨param-list-follow⟩ **)**

⟨block⟩ ::= **{** ⟨stmts⟩ **}**

⟨type⟩ ::= **int** | **bool**

⟨decl⟩ ::= ⟨type⟩ **ident**

| ⟨type⟩ **ident** **=** ⟨exp⟩

⟨stmts⟩ ::= ε | ⟨stmt⟩ ⟨stmts⟩

⟨stmt⟩ ::= ⟨simp⟩ **;** | ⟨control⟩ | ⟨block⟩

⟨simp⟩ ::= ⟨lvalue⟩ ⟨asnop⟩ ⟨exp⟩

| ⟨decl⟩ | ⟨call⟩

⟨simpopt⟩ ::= ε | ⟨simp⟩

⟨lvalue⟩ ::= **ident** | **(** ⟨lvalue⟩ **)**

⟨elseopt⟩ ::= ε | **else** ⟨stmt⟩

⟨control⟩ ::= **if** **(** ⟨exp⟩ **)** ⟨stmt⟩ ⟨elseopt⟩

| **while** **(** ⟨exp⟩ **)** ⟨stmt⟩

| **for** **(** ⟨simpopt⟩ **;** ⟨exp⟩ **;** ⟨simpopt⟩ **)** ⟨stmt⟩

| **continue** **;** | **break** **;** | **return** ⟨exp⟩ **;**

⟨call⟩ ::= **print** ⟨arg-list⟩ | **read** ⟨arg-list⟩ | **flush** ⟨arg-list⟩

| ⟨ident⟩ ⟨arg-list⟩

⟨arg-list-follow⟩ ::= ε | **,** ⟨exp⟩ ⟨arg-list-follow⟩

⟨arg-list⟩ ::= **(** **)** | **(** ⟨exp⟩ ⟨arg-list-follow⟩ **)**

⟨exp⟩ ::= **true** | **false** | **ident**

| **(** ⟨exp⟩ **)** | ⟨intconst⟩

| ⟨exp⟩ ⟨binop⟩ ⟨exp⟩

| ⟨unop⟩ ⟨exp⟩

| ⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩

| ⟨call⟩

⟨intconst⟩ ::= **decnum** | **hexnum**

⟨unop⟩ ::= **!** | **~** | **-**

⟨asnop⟩ ::= **=** | **+=** | **-=** | ***=** | **/=** | **%=**

| **&=** | **^=** | **|=** | **<<=** | **>>=**

⟨binop⟩ ::= **+** | **-** | ***** | **/** | **%** | **<** | **<=** | **>** | **>=**

| **==** | **!=** | **&&** | **||** | **&** | **^** | **|** | **<<** | **>>**

ident ::= `[A-Za-z_] [A-Za-z0-9_]*`

decnum ::= `0` | `[1-9] [0-9]*`

hexnum ::= `0 [xX] [A-Fa-f0-9]+`

Listing 1: Grammar of **L3**, nonterminals in ⟨brackets⟩, terminals in **bold**.

L3 Elaboration

In the following sections we present elaborations for the two main new constructs, functions and function calls.

Now that we have multiple functions in one program, we need to expand our abstract representation from **L2**. The easiest way to do so is to consider each function on its own. The only interaction between functions, function calls, can be dealt with by first collecting all available functions and their signatures (their name and type), before elaborating any individual function. This ensures we always know what functions exist and can resolve calls to functions defined later in the file. Built-in functions are manually added to this list, ensuring type-checking and analysis passes have all necessary information.

L3 Static Semantics

The static semantics of **L3** expand mostly naturally. One major departure from C is that we only have function definitions and no function declarations (a definition implies the declaration). As a consequence, the order of function definitions does not matter; a function call can refer to a function defined later.

In **L3**, we consider functions and variables to reside in two separate namespaces. Therefore, variables and functions can have the same name; it is never ambiguous. Function names are also unique, i.e., there is no function overloading in **L3**.

A well-formed **L3** program has to contain a function named `main` which doesn't take any arguments and returns an `int`, the program's exit code. This is the entry point, so execution of your program starts here.

Type Checking

In addition to variable types, **L3** introduces function types. A function type is an n-tuple consisting of the function's return type and zero or more parameter types. To type-check function calls, you need to verify that all arguments have the same type as their corresponding parameters. If a function is used inside an expression, you also need to validate that the function's return type matches what the outer expression expects.

Parameters

We treat function parameters like local variables defined at the beginning of the function. In particular, you can not re-declare them, but you can write to them to update their value inside the function, even if that is discouraged by any sane style guide.

Built-in functions

There are three built-in functions that you have to consider when analyzing code.

Name	Return Type	Parameter Type	Return Value
`print`	`int`	`int`	`0` (constant)
`read`	`int`	-	`-1` if no input is available, `0-255` otherwise
`flush`	`int`	-	`0` (constant)

Table 2: The three built-in functions.

L3 Dynamic Semantics

In **L3**, the correct ordering of operations becomes more important. As **L2** programs could only throw exceptions and were otherwise side-effect free, the compiler had great flexibility in ordering blocks of code. In **L3** though, the `print` built-in function allows side effects wherever the program author pleases. For example, reordering a division before a `print` might make the `print` unreachable in some cases, changing the output.

The arguments of a function call are evaluated left-to-right, before the function is called. Functions are call-by-value, i.e. you need to *copy* the argument into the called function:

```
int main() {
  int localVar = 21;
  foo(localVar);
  return localVar; // returns 21
}
```

```
int foo(int a) {
  a = 42;
  return 0;
}
```

The three built-in functions, `print`, `read` and `flush` mostly mirror the semantics of the corresponding `libc` library functions and operate on the `stdout` and `stdin` streams respectively. We suggest you implement them as wrapper functions delegating to `putchar` for `print`, `getchar` for `read` and `fflush(stdout)` for `flush`. Make sure you carefully read what the `libc` functions return and adjust it according to the specification of **L3**. It could be helpful to read Calling Conventions (p. 6).

As the compiler author, you need to ensure that `stdout` is flushed before a compiled program exits if it exits without exception. You can e.g. adjust your assembly template, or introduce synthetic calls to `flush` in `main`.

Project Requirements

For this project, you are required to hand in a complete working compiler for **L3** that produces correct and executable target programs for x86-64 Linux machines.

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. 6) for details.

Repository setup

You should have a working setup already from Lab 1 and 2. If not, refer to Lab 1 for more details on how to set up your source code repository.

Test Files

Tests are still handled by `crow`, for detailed instructions on how the testing system works, take a look at Lab 1. Test modifiers available for **L3** tests are listed in Table 3.

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 🐞.

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.
`Expected output`	*long string*	The output printed to the standard output stream (“`stdout`”) by 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`.
`Should timeout`		Your **binary** does not terminate (within a few seconds).
`Exit code`	`0-255`	Your **binary** exits with the given exit code, i.e. returns this value from `main`.

Table 3: The currently implemented test modifiers in crow.

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: 23.06.)
  - 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: 30.06.)
  - 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

The following section contains some additional hints that might be of use while you upgrade your compiler to accept **L3** programs.

Calling Conventions

When calling `libc` library functions, your code must strictly adhere to the x86-64 System-V AMD64 ABI calling convention. We want to draw special emphasis to "The stack is 16-byte aligned just before the call instruction is executed." 16-byte alignment means that your stack pointer (`rsp`) is divisible by 16 without a remainder.

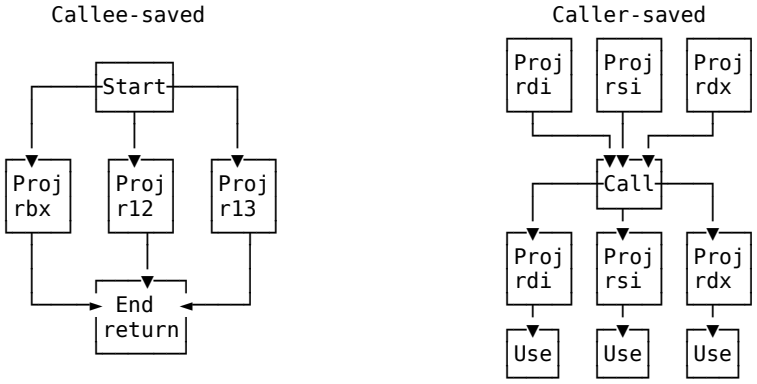
The calling convention is highly relevant for your register allocator. When another function is called, it too will likely have local variables loaded into registers, clobbering whatever was in there before. The System-V AMD64 ABI therefore splits registers into two groups: call*er*-saved and call*ee*-saved. Caller-saved registers can be freely mutated by any function, so you need to spill to the stack or other registers if you still need a value after a call. Complementary, callee-saved registers must survive across a call – if you write to those you need to save and restore their original value before returning.

For **L3** function calls you have absolute power: You compile the calls as well as the callees, after all. Still, sticking to the same native calling convention allows existing debuggers like `gdb` to work correctly. It also means you don't have to special-case the `libc` calls in your register allocator.

Handling caller and callee-saved registers can be done by simply saving all callee-saved registers on the stack at the beginning of the method and restoring them at the end. A slightly smarter solution would only save the registers that are actually *written to* inside the method.
Caller-saved registers can be saved on the stack before each call and restored afterwards. A slightly smarter compiler only saves the registers allocated to values that actually live across the call.

There also are some more advanced tricks for handling callee- and caller-saved registers in a unified fashion, without any larger special casing. **These can quickly become a bit unwieldy – and surprisingly difficult – if done to their fullest extent, so we recommend starting with the simple solution above.**
For callee-saved registers, you project one value per callee-saved register out of the start node, limited to that register, and use it again as an input for every `return` statement. If your register allocation algorithm is sufficiently well behaved, this will automatically ensure callee-saved registers are restored when needed: Whenever the register allocator assigns to a callee-saved register inside the method, the lifetimes of the artificial projection and the new value will collide, necessitating a stack spill and reload.

The same pattern applies to caller-saved registers. You take all caller-saved registers as input for the call (or a "Permutation" node directly before the call, which also allows you to shuffle them around) and project them back out. All later usages of these registers are rewired to their projection after the call.



Parameters in SSA

You might notice that – in contrast to normal local variables – parameters don't have a definition as we know it (unless reassigned). When using a graph-based SSA representation, you can define a new node type that has the start node as its input, and holds the value of a parameter.

Happy Coding!