Compiler Design Lab 4
Summer 2025

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

Start: 30.06.2025
Tests due: 10.07.2025
Due: 17.07.2025

Introduction

In this lab you will upgrade your **L3** compiler to a compiler for the language **L4**. After dealing with only the stack for now, **L4** can allocate memory on the heap! While this is already plenty interesting for our existing primitive types, we also introduce arrays and structs which come with their own implementation challenges. While we're at it, we finally give meaning to some of the reserved keywords that have been unused so far. Using pointers means we also have to fight the dreaded `NULL`, however, we make sure to do so in a type-safe way.

What this means for you is that you will have to touch all phases of your existing compiler, one last time.

xkcd 138: Pointers

L4 Syntax

The concrete syntax of **L4** is based on ASCII character encoding of source code.

Lexical Tokens

The reserved keywords in **L4** are the same as in **L3**.

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

L4 brings new tokens `->`, `.`, and `*`, to access struct members and to denote and to dereference pointers. Additionally, **L4** uses `[` and `]` for array types and accesses.

Whitespace and Token Delimiting

In **L4**, 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

L4 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 **L4** 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 before: we attribute the `else` to the closest `if` without an `else`.

Operator	Associates	Meaning
`*`	n/a	pointer dereference
`() [] -> .`	n/a	explicit parentheses, array subscript, field dereference, field select
`! ~ -`	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 **L4**, from highest to lowest.

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

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

⟨param⟩ ::= ⟨type⟩ **ident**

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

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

⟨struct⟩ ::= **struct** **ident** **{** ⟨field-list⟩ **}** **;**

⟨field⟩ ::= ⟨type⟩ **ident** **;**

⟨field-list⟩ ::= ε | ⟨field⟩ ⟨field-list⟩

⟨block⟩ ::= **{** ⟨stmts⟩ **}**

⟨type⟩ ::= **int** | **bool** | **struct** **ident** | ⟨type⟩ ***** | ⟨type⟩ **[** **]**

⟨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⟩ **)**
| ⟨lvalue⟩ **.** **ident** | ⟨lvalue⟩ **->** **ident** | ***** ⟨lvalue⟩ | ⟨lvalue⟩ **[** ⟨exp⟩ **]**

⟨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⟩
| **alloc** **(** ⟨type⟩ **)** | **alloc_array** **(** ⟨type⟩ **,** ⟨exp⟩ **)**
| ⟨ident⟩ ⟨arg-list⟩

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

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

⟨exp⟩ ::= **true** | **false** | **ident** | **NULL**
| **(** ⟨exp⟩ **)** | ⟨intconst⟩
| ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ | ⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ **?** ⟨exp⟩ **:** ⟨exp⟩ | ⟨call⟩
| ⟨exp⟩ **.** **ident** | ⟨exp⟩ **->** **ident**
| ***** ⟨exp⟩ | ⟨exp⟩ **[** ⟨exp⟩ **]**

⟨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 **L4**, nonterminals in ⟨brackets⟩, terminals in **bold**.

One of the fun difficulties in parsing `C` is that parsing needs to know *type-information*. Consider `foo* bar;`. This is obviously a variable definition for `bar` of type `foo*`, right? But what happens when we add some inconsequential whitespace? `foo * bar;` now looks much more like an expression statement multiplying `foo` and `bar` together. However, this whitespace is not required by the grammar — both are valid ways to write the same expressions. Therefore, the parser needs to know information about `foo` to continue parsing: either it is already known as a type, in which case you parse it as a declaration, or it is already known as a variable, in which case you parse it as a multiplication.

In our languages, we have *separate namespaces* for types, functions, and variables, which means that this distinction is not easily possible. Instead, we sidestep this issue in a different way — **L4** has no `typedefs` and no expression statements. This disambiguates `struct foo*bar;` to `struct foo* bar;`.

L4 Static Semantics

In **L4**, we consider functions, structs, and variables to reside in separate namespaces. Therefore, functions, structs, and variables can have the same name; it is never ambiguous. Function names are also unique, i.e., there is no function overloading in **L4**. The same applies to structs, so two struct definitions with the same name are not allowed.

Additionally, structs create their own nested namespace. This means that all field names inside a struct must be unique, but different structs can have fields with the same name.

Like functions, struct definitions are visible even before their definition, and definitions imply their declaration. This also means that the definition order of functions and structs is irrelevant.

Within **L4**, `foo->bar` is syntactic sugar for `(*foo).bar`, but quite useful in practice.
The dereference operator `*` can only be applied to *pointers*, not arrays or any other primitive type.

If `x` is a value of the array type τ`[]`, `x[i]` is the `i`-th element of `x`, and is of type τ. Arrays are indexed beginning with `0`.

A well-formed **L4** 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.

The other existing language constructs inherit their semantics from **L3**.

Type Checking

In **L4** we distinguish between small types (that can fit into a register) and large types (that have to be stored in memory). Refer to the lecture notes on "Program Semantics and Analysis"[1] for a more detailed explanation. **L4** programs have to adhere to the following typing rules:

Local variables, function parameters, and return types must be of small types. Lvalues (values appearing on the left side of an assignment expression) must be of small types as well. Do not get this mixed up with the $\langle\text{lvalue}\rangle$ production rule in the grammar, which operates on a syntactical level. Everything that might be assigned to is derived by this production rule in the grammar, but our semantic definition of *lvalue* is more restrictive: An *lvalue* is a place in memory, something to which we can assign.

Now that **L4** has pointers, we also need to define what equality means for them: comparing pointers is only allowed if they both refer to the same type and only performs an identity comparison (do the two pointers point to the same memory address), not a deep equality (are all fields recursively the same).

Structs can have other structs and arrays as members, and arrays of structs are also possible. In order to prevent types with infinite size, a struct may not appear as a member of itself, either directly or indirectly. Therefore, the following program is invalid:

```
struct Foo {
  struct Foo field;
};
```

However, using a pointer to itself is allowed, as a pointer always has a fixed size:

```
struct LinkedList {
  struct LinkedList* next;
};
```

[1] https://symbolaris.com/course/Compilers/12-semantics.pdf

While `NULL` is a valid value of every pointer type, we disallow directly dereferencing `NULL` to simplify type checking. That means, `NULL->a` and `*NULL` are not allowed and should result in a semantic error.
Expressions such as `NULL == NULL` are still allowed, however.

Built-in functions

There are five built-in functions that you have to consider when analyzing code:

Name	Return Types	Parameter Type	Return Value
`print`	`int`	`int`	`0` (constant)
`read`	`int`	-	`-1` if no input is available, `0-255` otherwise
`flush`	`int`	-	`0` (constant)
`alloc`	*type**	*type*	A pointer to a slice of memory large enough to hold a *type*
`alloc_array`	*type*`[]`	*type*, `int`	A slice of memory large enough to hold `int` *type* values.

Table 2: The five built-in functions.

Note that `alloc` and `alloc_array` have to be treated as special cases in the parser, since an *expression* like `int` wouldn't make much sense elsewhere.

The `alloc` function takes a type τ as its only parameter and should allocate memory on the heap, large enough to hold a value of the provided type, and return a pointer of type $\tau *$ to the newly allocated memory.

The `alloc_array` function expects both a type τ and an integer `n` and should allocate enough memory on the heap to hold `n` values of type τ. It returns a pointer of type $\tau *$ to the allocated memory. Attempting to allocate an array with a negative size should result in a semantic error.

Both `alloc` and `alloc_array` should never return `NULL`, except in the case that the system has run out of memory to allocate. The latter restriction is only to allow you to use `libc` functions, *please do not test against it*. Very few userspace programs are able to adequately handle out-of-memory errors and nothing you will want to write in **L4** fits that category. If you have good ideas for how to handle out-of-memory situations in Linux, the LKML is likely happy to receive them 🐞.

The functions `print`, `read`, and `flush` behave just like they did in **L3**.

L4 Dynamic Semantics

L4 adds even more features where order of operation is crucial. Besides new exceptions when reading from unallocated memory addresses, you also need to consider the order of load/store operations.

If `p` is a pointer of type $\tau *$, accessing the pointer via `*p` evaluates `p` to a memory address and returns the contents of the memory at that address. Unsurprisingly, dereferencing a pointer with the value `NULL` should raise a `SIGSEGV` exception.

Memory allocated via `alloc` is zero-initialized. That means:
- `int` and `bool` values are initialized with `0` and `false` respectively,
- pointer values are `NULL`,
- values of array types point to address `0`, like a `NULL` pointer,
- struct members are zero-initialized recursively.

The elements of an array allocated using `alloc_array` are all zero-initialized.

We also want to make sure we only touch memory we allocated, so when indexing into an array, the program should exit with a `SIGABRT` exception if the index lies outside the bounds of the array (either less than 0 or greater or equal to the array's length).

For an in-depth explanation of the dynamic semantics of memory accesses once again refer to the lecture notes mentioned in Type Checking (p. 4) .

Project Requirements

For this project, you are required to hand in a complete working compiler for **L4** 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. 7) for details.

Repository setup

You should have a working setup already from the previous labs. 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 **L4** 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 crash`	`Abort`	Your **binary** crashes with signal `SIGABRT`.
`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.
    `Should crash > Segmentation fault` in `crow`
  - `SIGABRT` indicates an assertion failure, like out-of-bounds array access.
    `Should crash > Program aborted` 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: 10.07.)
  - 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: 17.07.)
  - 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 **L4** programs.

For allocating memory on the heap, you can have a look at the `calloc` function from `libc`. If you decide to use `malloc` instead, be sure to `memset` the memory! It might work correctly on first-allocation (as the Linux kernel only hands out zeroed memory to processes) but will fail once `malloc` reuses memory. This might not happen at all, or only in specific circumstances, as **L4** has no `free`. You do not want to debug this.

As arrays are subject to runtime bounds checks, it is a very good idea to also store the array's length in the allocated memory.
The alternative to explicit runtime bounds checks is range-analysis driven omission. This allows the compiler to reason about the possible values of index variables and elide the check when it must always fall into the array's length. Maybe unintuitively, this typically speeds up programs quite a bit, as it often happens inside tight loops. It is also the cause for *numerous* arbitrary code executions vulnerabilities in production JIT compilers, such as V8.

For both `alloc` and `alloc_array`, you probably want to have the returned pointer pointing to the *beginning* of the allocated memory, with struct fields and array accesses implemented via positive offsets from the base address. The negative offsets are sometimes used for meta information, like a vtables or class information in languages like Haskell. There is no need for either in **L4**, however, so you are free to make up your own memory layout.

Happy Coding!