Compiler Design Lab 2
Summer 2025

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

Start: 20.05.2025
Tests due: 27.05.2025
Due: 16.06.2025

Introduction

In this lab you will expand your compiler for **L1** to a compiler for the language **L2**. This language expands upon **L1** by adding support for conditionals, loops, and some additional operators. To implement this, you will likely have to touch all phases of your existing compiler. Making use of these new features, we can write some interesting iterative programs over integers.

xkcd 1421: Future Self

L2 Syntax

The concrete syntax of **L2** is based on ASCII character encoding of source code.

Lexical Tokens

L2 has some additional unary and binary operators; you can find them as terminals in the `asnop`, `unop`, `binop` productions of Listing 1.

The reserved keywords in **L2** are the same as in **L1**:

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

Whitespace and Token Delimiting

In **L2**, 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

L2 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 **L2** 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 occurs in `if` chains terminating with a single else, like `if (a) if (b) else`. This code can be parsed in two ways:

```
if (a) {
  if (b) {
  } else {
  }
}
```

```
if (a) {
  if (b) {
  }
} else {
}
```

In **L2**, we attribute the `else` to the closest `if` without an `else`, i.e. the first option on the left.

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	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 **L2**, from highest to lowest.

⟨program⟩ ::= **int** **main** **(** **)** ⟨block⟩
⟨block⟩ ::= **{** ⟨stmts⟩ **}**
⟨type⟩ ::= **int** | **bool**
⟨decl⟩ ::= ⟨type⟩ **ident**
| ⟨type⟩ **ident** **=** ⟨exp⟩
⟨stmts⟩ ::= ε | ⟨stmt⟩ ⟨stmts⟩
⟨stmt⟩ ::= ⟨simp⟩ **;** | ⟨control⟩ | ⟨block⟩
⟨simp⟩ ::= ⟨lvalue⟩ ⟨asnop⟩ ⟨exp⟩
| ⟨decl⟩
⟨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⟩ **;**
⟨exp⟩ ::= **true** | **false** | **ident**
| **(** ⟨exp⟩ **)** | ⟨intconst⟩
| ⟨exp⟩ ⟨binop⟩ ⟨exp⟩
| ⟨unop⟩ ⟨exp⟩
| ⟨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 **L2**, nonterminals in ⟨brackets⟩, terminals in **bold**.

L2 Elaboration

As the name intended to suggest, Elaboration is the process of elaborating the grammar that we have to one that is simpler and more well behaved – the abstract syntax. We describe elaboration separately because it is logically intended as a separate pass of compilation that happens immediately after parsing. Unfortunately, justifying the way we choose to elaborate a language may depend on an intuitive understanding of the operational behavior of the concrete language. Therefore, we recommend that you check what you see in this section against the description of the static and dynamic semantics of **L2** that appear in the next two sections.

Your implementation may of course employ a different elaboration strategy — but we will rely on the following elaboration strategy extensively in the description of the static semantics, and your implementation must behave in an equivalent manner. As always, document any design decisions you make.

We propose the following tree structure as the abstract syntax for statements s:

$$
\begin{aligned}
s ::= &\ | \ \mathbf{assign}(x, e) && | \ \mathbf{if}(e, s, s) \\
&| \ \mathbf{while}(e, s) && | \ \mathbf{for}(s, e, s, s) \\
&| \ \mathbf{continue} && | \ \mathbf{break} \\
&| \ \mathbf{return}(e) && | \ \mathbf{seq}(s, s) \\
&| \ \mathbf{declare}(x, \tau, s) && | \ \mathbf{nop}
\end{aligned}
$$

where e stands for an expression, x for an identifier, and τ for a type. Do not be confused by the fact that this looks like a grammar: the terms on the right hand side describe trees, not strings. The whole program is represented here as a single statement $\mathbf{seq}(s_1, \mathbf{seq}(s_2, ...))$. In an implementation it may be more convenient to use lists explicitly.

Here are the suggested inference rules to elaborate blocks to trees:

$$\frac{}{\{\} \rightsquigarrow \mathbf{nop}} \qquad \frac{\text{statement} \rightsquigarrow s \qquad \{\text{remaining body}\} \rightsquigarrow s'}{\{\text{statement; remaining body}\} \rightsquigarrow \mathbf{seq}(s, s')}$$

$$\frac{\{\text{remaining body}\} \rightsquigarrow s'}{\{\tau\ x;\ \text{remaining body}\} \rightsquigarrow \mathbf{declare}(x, \tau, s')}$$

Note that for the code: $\tau\ x = e$ one may be tempted to write the following elaborate rule:

$$\frac{e \rightsquigarrow e' \qquad \{\text{remaining body}\} \rightsquigarrow s'}{\{\tau\ x = e;\ \text{remaining body}\} \rightsquigarrow \mathbf{declare}(x, \tau, \mathbf{seq}(\mathbf{assign}(x, e'), s'))}$$

But this form of elaboration is subtly wrong! This states that the variable x should have type τ in the expression e, but semantically that doesn't quite make sense. The language specification states that variables are only available in the statements following their initialization. This will become more apparent in lab 3 when functions are added, but for the meantime, please check these elaboration rules against the scoping rules given in the static semantics.

As can be seen from the syntax, `if`-statements always have both a `then` branch and an `else` branch represented by the two statements in that order. `for`-loops have the *initializer* statement, the conditional expression, the *step* statement, and the loop body in that order. Elaborating `if`, `else`, `while`, `for`, `break`, `continue`, and `return` should be fairly straightforward, and we do not give any rules for them.

As in **L1** assignment statements of the form `a op= b` are elaborated to be equivalent to `a = a op b`.

Unlike statements, expressions are already in a compact and well-behaved representation. We only elaborate the logical operators. `a && b` elaborates `a ? b : false`, and `a || b` elaborates to `a ? true : b`.

L2 Static Semantics

The following section describes the static semantics of **L2**.

Type Checking

Since our grammar and our type system have become a little more interesting, we now give some rules for type-checking. These rules are fairly informal.

Type-checking Statements v. Type-checking Expressions

Our grammar allows expressions to appear within statements, but there is no way to embed statements within expressions. As a consequence it is meaningful to have a judgement of the form "$e : \tau$" to convey that an expression e has the type τ, but the same is meaningless when discussing statements. Therefore, for statements, we have the judgement "$s\ valid$".

Variable Declarations and Contexts

As in **L1**, variables need to be declared with their types before they can be used. Unlike in **L1**, declarations can appear in any block. The declaration of a variable is not visible outside the block of its declaration. Note that variables declared in inner blocks do *not* shadow variables declared in enclosing scopes. Therefore, multiple declarations of the same identifier may be present in the body of `main` if and only if no two of them are visible within the same block.

The abstract syntax for declarations, i.e. $\mathbf{declare}(x, \tau, s)$ is very convenient for capturing all these phenomena and specifying the rules of type-checking. Here, the declaration of x of type τ is visible only to statement s, and your elaborator should take care to elaborate statements while correctly preserving the lexical scope of declarations. Making the scope explicit in the abstract syntax makes it easy for you to build up a context of variable declarations available while type-checking any particular statement or expression.

We say "$\Gamma \vdash e : \tau$" when an expression e is type-checked under the context Γ, which keeps track of all variable declarations and their types. The following inference rules demonstrate the function of the context.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{x : \tau' \notin \Gamma \text{ for any } \tau' \quad \Gamma, x : \tau \vdash s \ valid}{\Gamma \vdash \mathbf{declare}(x, \tau, s) \ valid}$$

Here, x stands for any identifier.

The Types

In **L2**, `int` is no longer the only type. It is joined by `bool`, which is inhabited by `true` and `false`. **L2** does not allow implicit or explicit coercion between integral and boolean values. This is a major point of departure from C and other (in)famous languages.

Statements

We have already explained how declarations work. Here are the remaining significant rules:

$$\frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash s_1 \ valid \quad \Gamma \vdash s_2 \ valid}{\Gamma \vdash \mathbf{if}(e, s_1, s_2) \ valid} \qquad \frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash s \ valid}{\Gamma \vdash \mathbf{while}(e, s) \ valid}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \mathbf{bool} \quad \Gamma, x : \tau \vdash s_2 \ valid \quad \Gamma, x : \tau \vdash s_3 \ valid \quad x \notin \Gamma}{\Gamma \vdash \mathbf{for}(\tau \ x = e, e_1, s_2, s_3) \ valid}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \mathbf{bool} \quad \Gamma, x : \tau \vdash s_2 \ valid \quad \Gamma, x : \tau \vdash s_3 \ valid \quad x \notin \Gamma}{\Gamma \vdash \mathbf{for}(\tau \ x, e_1, s_2, s_3) \ valid}$$

$$\frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash s_1 \ valid \quad \Gamma \vdash s_2 \ valid \quad \Gamma \vdash s_3 \ valid}{\Gamma \vdash \mathbf{for}(s_1, e, s_2, s_3) \ valid}$$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{assign}(x, e) \ valid} \qquad \frac{\Gamma \vdash s_1 \ valid \quad \Gamma \vdash s_2 \ valid}{\Gamma \vdash \mathbf{seq}(s_1, s_2) \ valid} \qquad \frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash \mathbf{return}(e) \ valid}$$

The rule for `return` statements is still very rudimentary because we have only one function in the program, and it is required to return an `int`.

Expressions

The following are the rules to check expressions for type correctness:

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{intconst} : \mathbf{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \text{relop} \in \{<, \leq, >, \geq\}}{\Gamma \vdash e_1 \text{ relop } e_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \text{polyeq} \in \{==, !=\}}{\Gamma \vdash e_1 \text{ polyeq } e_2 : \mathbf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool} \quad \text{logop} \in \{\&\&, ||\}}{\Gamma \vdash e_1 \text{ logop } e_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash !e : \mathbf{bool}}$$

Note that in a type theoretic sense, equality and disequality are *overloaded operators*, not polymorphic operators. See the dynamic semantics to see how the implementation is affected.

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \text{ binop } e_2 : \mathbf{int}} \qquad \frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash \text{unop } e : \mathbf{int}}$$

Here, binop and unop are all the remaining binary and unary operators in the grammar not covered by the rules for booleans.

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (e_1 \,?\, e_2 : e_3) : \tau}$$

Control Flow

Regarding control flow, several properties must be checked:
- Each (finite) control flow path through the program must terminate with an explicit `return` statement. This ensures that the program does not terminate with an undefined value.
- Each `break` or `continue` statement must occur within a `while` or `for` loop.

Regarding variables, we need the following:
- On each control flow path through the program, each variable must be defined by an assignment before it is used. This ensures that there will be no references to uninitialized variables.
- The *step* statement in a for loop may not be a declaration.

We define these checks more rigorously on the abstract syntax as follows.

Checking Proper Returns

We check that all finite control flow path through a program end with an explicit `return` statement. We say that s *returns* if every execution of s that terminates will always end with a return statement. Overall, we want to ensure that the whole program, represented as a single statement s, returns according to this definition. If not, the compiler must signal a semantic analysis error.

$\mathbf{declare}(x, \tau, s)$	returns if s returns
$\mathbf{assign}(x, e)$	does not return
$\mathbf{if}(e, s_1, s_2)$	returns if both s_1 and s_2 return
$\mathbf{while}(e, s)$	does not return
$\mathbf{for}(s_1, e, s_2, s_3)$	does not return
$\mathbf{return}(e)$	returns
$\mathbf{nop}$	does not return
$\mathbf{seq}(s_1, s_2)$	returns if either s_1 returns (and therefore s_2 is dead code) or s_2 returns.

We do not look inside loops (even though the bodies may contain `return` statements) because the body might not be executed at all, or might terminate by a `break`, not a `return`. Because we do not look inside loops, we do not need rules for `break` or `continue`.

Checking Variable Initialization

We wish to give a well-formed deterministic dynamic semantics to **L2**. A program should either return a value, raise a `divide` exception, or fail to terminate. In order to do this, our static semantics must enforce the following necessary condition: we need to check that along all control flow paths, any variable is defined before use. Since the language allows the nesting of scopes of declaration, we define the variable initialization property as a local property of a scope which is actually stronger than the guarantee we need to make in order to give a well-formed dynamic semantics to the entire program.

To further streamline the specification, we transform `for` loops as follows:

$$\mathbf{for}(\mathrm{initializer}, e, \mathrm{step}, \mathrm{body}) \rightsquigarrow \mathbf{seq}\big(\mathrm{initializer}, \mathbf{while}\big(e, \mathrm{body}_{\mathrm{step}}\big)\big)$$

Here $\mathrm{body}_{\mathrm{step}}$ is body with step inserted before every occurrence of `continue` that doesn't fall within the scope of an inner loop, and at the very end of the loop body. Check the dynamic semantics and make sure you understand why this transformation preserves those semantics. You may also find it an interesting exercise to consider what would go wrong if you were to include this transformation in the elaborator.

Make sure that your elaboration for the case when $\mathrm{initializer}$ contains a variable definition is very careful, so that variable initialization checking works on variables introduced in for loops. Your implementation may be more efficient if you convert `while` loops to `for` loops instead–a similar analysis will apply in this case.

First, we specify when a statement s *defines* a variable x. We read this as: Whenever s finishes normally, it will have defined x. This excludes cases where s returns, executes a `break` or `continue` statement, or does not terminate.

$\mathbf{declare}(x, \tau, s)$	defines no variable
$\mathbf{assign}(x, e)$	defines only x
$\mathbf{if}(e, s_1, s_2)$	defines x if both s_1 and s_2 define x
$\mathbf{while}(e, s)$	defines no x (because the loop body may not be executed)
$\mathbf{break}$	defines all x within scope (because it transfers control out of the scope)
$\mathbf{continue}$	defines all x within scope (because it transfers control out of the scope)
$\mathbf{return}(e)$	defines all x within scope (because it transfers control out of the scope)
$\mathbf{nop}$	defines no x
$\mathbf{seq}(s_1, s_2)$	defines x if either s_1 or s_2 does

We also say that an expression e *uses* a variable x if x occurs in e. In our language, e may have logical operators which will not necessarily evaluate all their arguments, but we still say that a variable occurring in such an argument is used, because it might be.

We now define which variables are *live* in a statement s, that is, their value may be used in the execution of s.

y is live in $\mathbf{declare}(x, \tau, s)$	if y is live in s and y is not the same as x
y is live in $\mathbf{assign}(x, e)$	if y is used in e
y is live in $\mathbf{if}(e, s_1, s_2)$	if y is used in e or live in s_1 or s_2
y is live in $\mathbf{while}(e, s)$	if y is used in e or live in s
y is live in $\mathbf{break}$	never (the jump target is accounted for elsewhere)
y is live in $\mathbf{continue}$	never (the jump target is accounted for elsewhere)
y is live in $\mathbf{return}(e)$	if y is used in e
y is live in $\mathbf{nop}$	never
y is live in $\mathbf{seq}(s_1, s_2)$	if y is live in s_1 or y is live in s_2 and not defined in s_1

Since scopes are encoded as declare statements, the given strategy has also told us what variables are live at the beginning of a scope, i.e. not initialized before its first use. Static analysis should reject a program if for any $\mathbf{declare}(x, \tau, s)$, the variable x is live in s.

The following example demonstrates how our static analysis is sufficient to guarantee deterministic evaluation:

```
{ int x; int y; return 1; x = y + 1 ; }
```

is valid because the statement `x = y + 1` cannot be reached along any control flow path from the beginning of the program. Formally, the statement `return 1` is taken to define all variables, including `y`, so that `y` is not live in the whole program even though it is live in the second statement.

The following example demonstrates how our static analysis is stronger than a sufficient condition to guarantee deterministic evaluation:

```
{ int x; return 1; { int y; x = y + 1; } }
```

We can still give a well-formed dynamic semantics for the program. However, static analysis will raise an error because the following block is not well-formed:

```
{ int y; x = y + 1; }
```

Make sure that you have checked that all control flow paths return and all occurrences of `break` and `continue` are inside loops *before* checking for variable initialization. Otherwise, the effect that these statements have on liveness may produce very unhelpful and cryptic error messages for ill-formed programs.

L2 Dynamic Semantics

In most cases, statements have the familiar operational semantics from C. Conditionals, `for`, and `while` loops execute as in C. `continue` skips the rest of the statements in a loop body and `break` jumps to the first statement after

a loop. As in C, when encountering a `continue` inside a `for` lop, we jump to the *step* statement. The suggested method of elaboration reflects this behavior. Both `break` and `continue` always apply to the innermost loop the occur in.

The ternary operator (`?:`), as in C, must only evaluate the branch that is actually taken. The suggested elaboration of the logical operators to the ternary operator also reflect their C-like short-circuit evaluation.

Integer Operations

Since expressions do not have effects (except for a possible `divide` exception that might be raised) the order of their evaluation is irrelevant.

The integers of this language are signed integers in two's complement representation with a word size of 32 bits. The semantics of the operations is given by modular arithmetic, as in **L1**. Recall that division by zero and division overflow must raise a runtime division exception. This is the only runtime exception that should be possible in the language for now.

As in **L1**, 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.

The left `<<` and right `>>` shift operations are arithmetic shifts. Since our numbers are signed, this means the right shift will copy the sign bit in the highest bit rather than filling with zero. Left shifts always fill the lowest bit with zero. Also, the shift quantity k will be masked to 5 bits and is interpreted positively so that a shift is always between 0 and 31, inclusively. This is also the hardware behavior of the appropriate arithmetic shift instructions on the x86-64 architecture and is consistent with C where the behavior is underspecified.

The comparison operators `<`, `<=`, `>`, and `>=`, have their standard meaning on signed integers as in the definition of C. Operators `==` and `!=` are overloaded operators that test for the equality of either a pair of `int`s or a pair of `bool`s.

Project Requirements

For this project, you are required to hand in a complete working compiler for **L2** 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. 9) for details.

Repository setup

You should have a working setup already from Lab 1. 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 **L2** tests are listed in Table 2.

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.
`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.

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: 27.05.)
  - 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: 16.06.)
  - 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 **L2** programs.

Static Checking

The specification of static checking should be implemented on abstract syntax trees, translating the rules into code. You should take some care to produce useful error messages.

It may be tempting to wait until liveness analysis on abstract assembly to see if any variables are live at the beginning of the program and signal an error then, rather than checking this directly on the abstract syntax tree. There are two reasons to avoid this: (1) it may be difficult or impossible to generate decent error messages, and (2) the intermediate representation might undergo some transformations (for example optimizations, or transforming logical operators into conditionals) which make it difficult to be sure that the check strictly conforms to the given specification.

Shift Operators

There are some tricky details on the machine instructions implementing shift operators. The instruction `sall k, D` (shift arithmetic left long) and `sarl k, D` (shift arithmetic right long) take a shift value `k` and a destination operand `D`. The shift either has to he the `%cl` register, which consists of the lowest 8 bits of `%rcx`, or can be given as an immediate of at most 8 bits. In either case, only the low 5 bits affect the shift of a 32 bit value; the other bits are masked out. The assembler will fail if an immediate of more than 8 bits is provided as an argument.

Happy Coding!