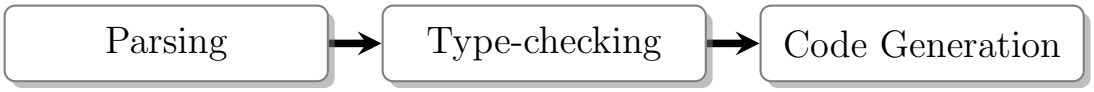
Aly Developer's Manual

Aly-Lang/NorthernL1ghts

September 28, 2025

Overview

Currently, `Aly` has three stages (or phases, or levels, anything to that regard; we'll call them stages).



The parser converts `Aly` source code into a data structure that represents the program in it's entirety.

The type-checker then validates this data structure, ensuring that the semantics of types are being followed properly.

The code generation stage creates a new data structure (an in-memory intermediate representation) that follows static single assignment form. This intermediate representation gets converted by each backend into architecture specific implementations (machine code, asm, etc).

Parsing

The parsing stage validates and begins to understand the syntax of `Aly` source code.

The parsing stage can be thought of as two systems that work together: the *lexer* and the *parser*. The parser drives the parsing stage, and calls to the lexer as needed.

The parser keeps track of all state (how much has been parsed so far, etc). The lexer is a simple tool to increment the state of the parser. The parsing state state consists of a lexeme, it's length in bytes, and a pointer to the end of the lexeme (where we will begin lexing from next). This pointer starts at the beginning of our source code, and makes it's way towards the end as the parser advances the parsing state.

Lexer

The lexer's job is to split up the input source code into understandable chunks (called lexemes); after all, the file is read as one, large, arbitrary sequence of bytes. The lexer splits these bytes into understandable pieces, and also strips off parts of the file that aren't needed, like whitespace and comments.

The lexer must not lex more than one lexeme at a time. It is the job of the parser to understand sequences of lexemes.

There are two main actions the lexer can make:

advance Get the lexeme directly after the given position, updating position to end of new lexeme.

expect Get the lexeme directly after the given position, only updating position if the given lexeme was found.

Given the following source code:

```
;; Aly Simple Lexing Example
a : integer
a := 69
```

The lexer would produce the following lexemes (line separated) from the above source code, on subsequent calls.

```
a
:
integer
a
:
=
69
```

Parser

The parser is the main component of the parsing stage, and does most of the heavy lifting.

The job of the parser is to create an understanding of the program, from the compiler's point of view. To do this, the program must be unambigous, and contain only sequences of lexemes that are understood.

One thing you may be wondering: how does a computer program (the compiler) *know* something? How can you make it *understand*? The idea is that by defining a data structure in our compiler that can store the meaning of programs, as well as any necessary data along with it, the compiler can construct one of these structures and end up with an understood program, because the data structure is already understood.

There are two usual choices for data structures that can fully define the semantics of a program, while also storing data along with it.

One, the most common, is called an *abstract syntax tree*, or AST. The AST is a fancy name for a tree that houses only the bare minimum data of what is needed for the program moving forward. For example, parentheses aren't needed in an AST, because they don't actually do anything. They are only for the parser to understand what order to parse things in; since we don't need them after that (the AST is already constructed in the right order), they don't get included in the AST itself. This is why it is an *abstract* syntax tree; the tree doesn't map one-to-one with the source code syntax. Some compilers do build what's called a parse tree that does map one-to-one with the source code.

Another, still fairly common, is called a *directed acyclic graph*, or DAG. A DAG is much like an AST, except that it is not a tree structure, but a graph. This means any node can be connected to any node, and lots of complicated group and category theory ensues. Because a single node can be referenced by any other node, it does mean that it takes less memory to store the same program, and is likely more efficient overall. The problem lies within it's complexity: DAGs are hard to manage, and when it goes wrong, it goes very, very wrong. However, it is clear that a DAG is better in almost every way than an AST, if you are willing to deal with the headaches.

For the sake of simplicity and understanding, `Aly` uses an AST, or abstract syntax tree, and generates it from recognized sequences of lexemes.

A tree, in computer programming, is a data structure that can branch into multiple data structures. An AST happens to be a recursive tree, in that each of the branches of the tree can have branches, and those branches can have branches, and so on. A lot of compilers use a binary tree, or btree, which is a tree with a recursive left hand side and right hand side. The problem you see in using binary trees is that it is rather difficult to represent anything with more than two children (as one might expect). This is quite a common case in programming language ASTs; for example, when calling a function, there may be any amount of arguments. Most compilers go the

LISP route, and have these sort of lists implemented as recursive pairs, where as long as the lhs is another pair, the list continues. I feel like this is a messy implementation, and results in code that is quite hard to read, due to chains of dereferences, i.e. `lhs->lhs->rhs->value`.

The compiler includes a text-debug format to print the abstract syntax tree that is produced, and can be printed out for any program by adding the verbose command line flag: `-v`.

Let's take a look at an example. Given the following source code:

```
;; Aly Parsing Example
fact : integer (n : integer) = integer (n : integer) {
  if n < 2 {
    1
  } else {
    n * fact(n - 1)
  }
}

fact(5)
```

The parser would produce the following abstract syntax tree:

```
PROGRAM
  VARIABLE DECLARATION
    SYM:"fact"          <- colon separates node type from node value
  VARIABLE REASSIGNMENT
    SYM:"fact"
    FUNCTION
      SYM:"integer" (0) <- refers to level of pointer indirection
      NONE              <- empty node to store list of parameter types
        SYM:"integer"
      NONE              <- empty node to store list of body expressions
        IF
          BINARY OPERATOR:"<"
            VARIABLE ACCESS:"n"
            INT:2
          NONE          <- stores 'then' body
            INT:1
          NONE          <- stores 'otherwise' body
            BINARY OPERATOR:"*"
              VARIABLE ACCESS:"n"
              FUNCTION CALL
                SYM:"fact"
                BINARY OPERATOR:"-"
                  VARIABLE ACCESS:"n"
                  INT:1
  FUNCTION CALL
    SYM:"fact"
    INT:5
```

As you can see, this is a complete representation of the program. It is able to understand high level concepts that the programmer construed in the source code, like conditional control flow, variable accesses, function calls with arguments, and more.

Type-checking

Type-checking refers to the process of ensuring the types of expressions are what they are expected to be.

This is only relevant due to `Aly` being statically typed. This means the type of a variable is known at compile-time, due to the programmer declaring it. This declared type is associated with the variable, and any expressions assigned to it must return a compatible type. It also comes into play with binary expressions: they operate on two objects, each of some type. It wouldn't make any sense to try to add a function to an array, would it? Function calls' arguments must match a function's declared parameters' types. A dereference may only operate on a pointer. All of these semantics of the language are enforced by the type-checker.

Code Generation

Finally, we arrive at code generation. This is definitely the most complicated part, and is where compiler developers get lost for decades. While `Aly` started as a very simple, x86_64-only compiler, it is slowly growing to become a compiler that supports many backends, each of which cross platform themselves.

Our compiler has an intermediate representation.

If you are like me, you may wonder why this IR is even needed; it seems (at least to me) like it is of no use, and everything it represents could be stored in the AST. If you are not like me, that's good. You're probably smart :P.

Here is a list of reasons that were used to convince myself of using an IR: - There are none.

Seriously, you can do everything with just an AST, I promise. If you don't want an IR, don't use one. In the early versions of the compiler (that I wrote), there was no IR and codegen happened straight from AST. It wasn't always optimal but through peephole optimizations I'm fairly sure we could get it very good. In any case, I won't turn down a chance to learn something I don't know.

First of all, in the intermediate representation, that doesn't conform to any hardware standard, there are some assumptions that can be made. If there are no hardware limitations, then every value could just be stored in a new register; why wouldn't there be infinite? With this, we would never have to worry about slow-downs from RAM access or an appalling disk IO situation: it would simply always be register to register movement, and only when the user dereferences something would we begin to have memory accesses generated. It's like a code optimizers wet dream: everything in a register, and only explicit memory accesses.

```
;; Aly Intermediate Representation Example
;; Expressions are annotated with numbers for easy referencing.

;; Let us assume that input_condition and if_condition are somehow
;; different every time the program runs, like parameters.
input_condition : integer           ; 0
if_condition : integer              ; 1

a : integer                         ; 2
b : integer = 420                   ; 3
c : integer = 69                    ; 4
d : integer                         ; 5
e : integer                         ; 6
f : integer                         ; 7

while 1 {                           ; 8, 9

  a := b + c                        ; 10
  d := a * -1                       ; 11
  e := d + f                        ; 12

  if input_condition {              ; 13, 14
    f := 2 * e                      ; 15
  } else {
    b := d + e                      ; 16
    e := e - 1                      ; 17
  }

  b := f + c                        ; 18
}
```

And the generated AST of the program:

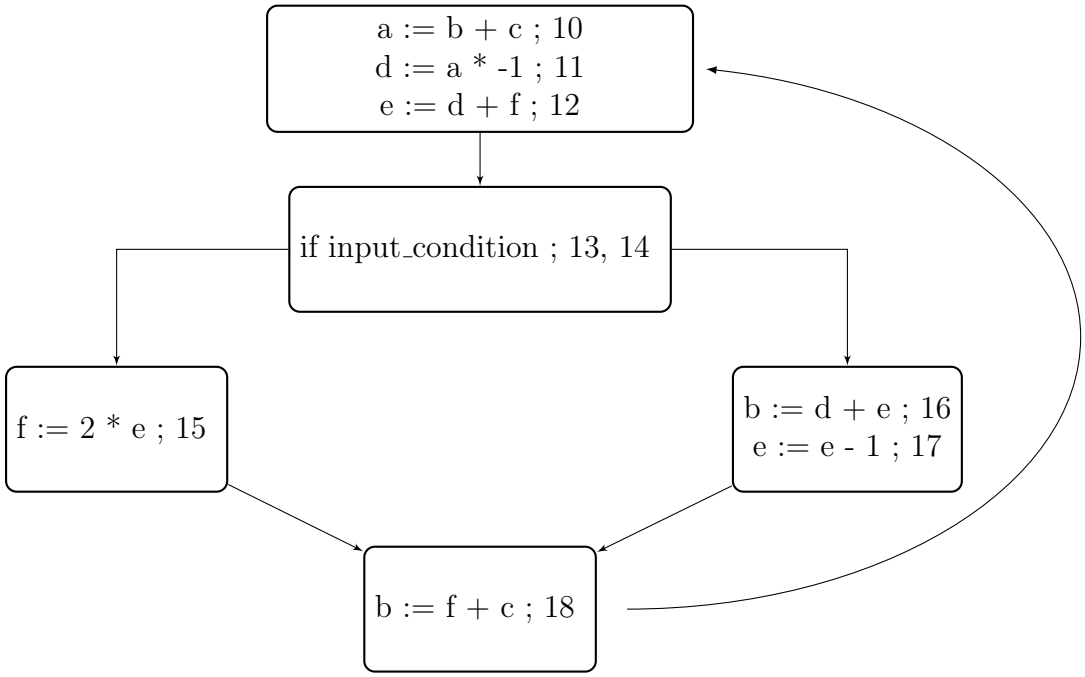
```
PROGRAM
  VARIABLE DECLARATION:"input_condition"    ; 0
  VARIABLE DECLARATION:"if_condition"       ; 1
  VARIABLE DECLARATION:"a"                  ; 2
  VARIABLE DECLARATION:"b"                  ; 3
  VARIABLE ASSIGNMENT
    SYM:"b"
    INT:420
  VARIABLE DECLARATION:"c"                  ; 4
  VARIABLE ASSIGNMENT
    SYM:"c"
    INT:69
  VARIABLE DECLARATION:"d"                  ; 5
  VARIABLE DECLARATION:"e"                  ; 6
```

```
VARIABLE DECLARATION:"f"                    ; 7
WHILE                                       ; 8
  INT:1                                     ; 9
  NONE
    VARIABLE ASSIGNMENT                     ; 10
      SYM:"a"
      BINARY OPERATOR:"+"
        VARIABLE ACCESS:"b"
        VARIABLE ACCESS:"c"
    VARIABLE ASSIGNMENT                     ; 11
      SYM:"d"
      BINARY OPERATOR:"*"
        VARIABLE ACCESS:"a"
        INT:-1
    VARIABLE ASSIGNMENT                     ; 12
      SYM:"e"
      BINARY OPERATOR:"+"
        VARIABLE ACCESS:"d"
        VARIABLE ACCESS:"f"
    IF                                      ; 13
      VARIABLE ACCESS:"input_condition"     ; 14
      NONE
        VARIABLE ASSIGNMENT                 ; 15
          SYM:"f"
          BINARY OPERATOR:"*"
            INT:2
            VARIABLE ACCESS:"e"
      NONE
        VARIABLE ASSIGNMENT                 ; 16
          SYM:"b"
          BINARY OPERATOR:"+"
            VARIABLE ACCESS:"d"
            VARIABLE ACCESS:"e"
        VARIABLE ASSIGNMENT                 ; 17
          SYM:"e"
          BINARY OPERATOR:"-"
            VARIABLE ACCESS:"e"
            INT:1
    VARIABLE ASSIGNMENT                     ; 18
      SYM:"b"
      BINARY OPERATOR:"+"
        VARIABLE ACCESS:"f"
        VARIABLE ACCESS:"c"
```

Previously, this AST would be used to generate code by the code generation backend (the frontend delegates to different backends based on defaults or configuration at the command line). However, this now gets converted into

the following IR (excluding initial variable declarations, only for the sake of brevity in the initial basic block):

Here is the while loop body from our current program, in terms of `Aly` 's intermediate representation, or IR.



Each box with code within it is referred to as a *basic block*.
The diagram above illustrates the basic blocks of the "main" *function* in our example program and their relation to one another.

Basic Block A flat list of IR instructions that must end in a branch.

Function A flat list of basic blocks with at least an entry and a return block specified, although they may be the same.

However, this diagram does not actually conform to `Aly` 's intermediate representation; there is one more step to complete before it matches one-to-one: conversion into static single assignment form, or SSA form for short.

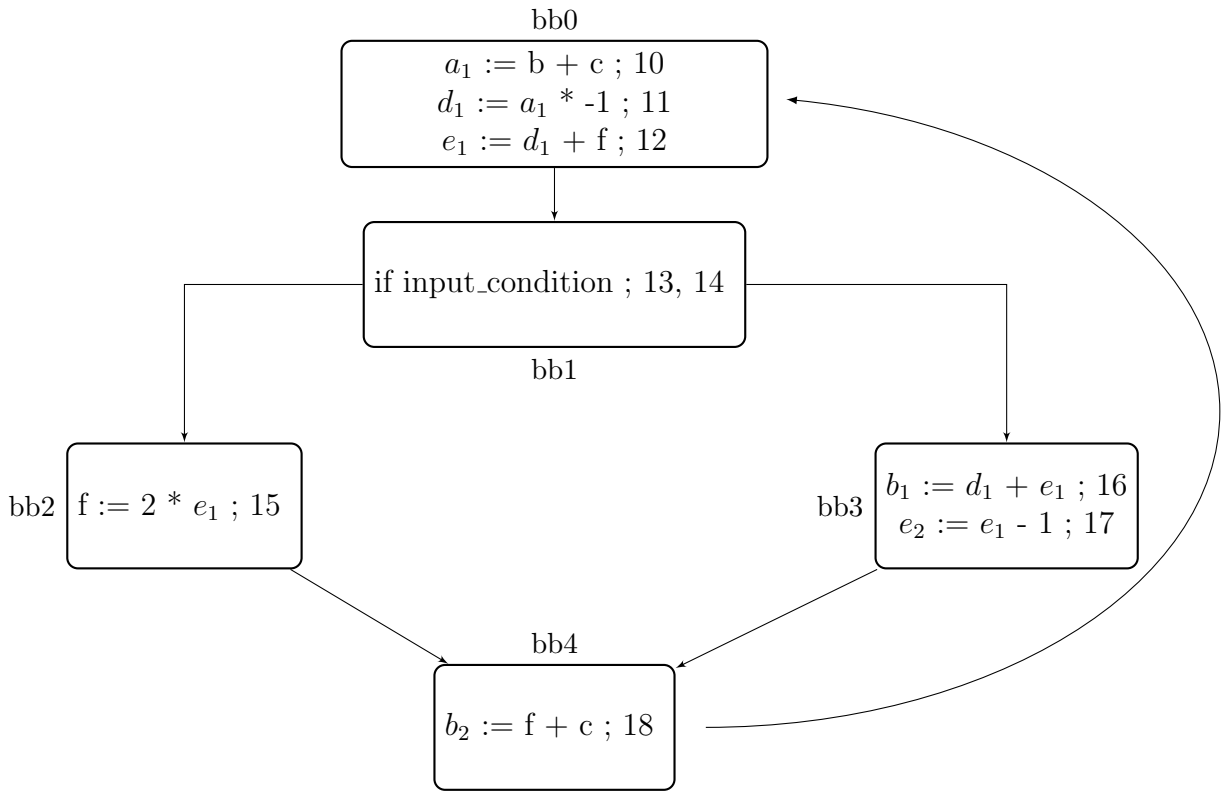
Static Single Assignment Form

A variable has different values at different times throughout the program, all with different lifetimes... They almost sound like different variables! Wait... what if they were?

Each assignment of a variable after the first creating a new variable is the entire idea behind *SSA*, or static single assignment, form. SSA form is a

form of program in which each variable does not get assigned more than one time.

Let's take a look at the example above, converted into SSA form.



The above diagram is not what the actual data structure appears as, so let's take a look at a lower level representation of the intermediate representation. How about that?

% A temporary value, like a register on a CPU.

bb A basic block within a function.

fun A function with a list of basic blocks.

```
fun main:
  bb0:
    %r0 = global_load("b")
    %r1 = global_load("c")
    %r2 = add(%r0, %r1)
    global_store(%r2, "a") ;;  10
    %r3 = global_load("a")
    %r4 = immediate(-1)
    %r5 = multiply(%r3, %r4)
    global_store(%r5, "d") ;; 11
    ;; ...
```

Variable Liveness

WARNING: The following section contains vocabulary regarding topics of life and death, due to this being what it has been called in many, many learning resources. We hope to move away from these possibly offensive terms, and towards something we can all discuss without bad memories being brought up.

Live A temporary is considered live after it has been assigned and up until the last use of this temporary.

Live Range The range of expressions in a basic block in which a temporary is live.

Liveness analysis is done by keeping track of every use of some calculated value in a linked list of `Use` structures (pronounce like the noun, the 'S' is voiceless). This is done during intermediate representation generation. Effectively, we build our IR with live ranges built-in.

If a temporary is live during the definition of another temporary, the two temporaries are said to *interfere*.

Excerpt from demonstration IR above:

```
%r0 = global_load("b")   ;; a
%r1 = global_load("c")   ;; b
%r2 = add(%r0, %r1)      ;; c
global_store(%r2, "a")   ;; d
%r3 = global_load("a")   ;; e
%r4 = immediate(-1)      ;; f
%r5 = multiply(%r3, %r4) ;; g
global_store(%r5, "d")   ;; h
```

Live ranges from above excerpt:

```
  r0 r1 r2 r3 r4 r5
a |
b |  |
c |  |  |
d       |
e          |
f          |  |
g          |  |  |
h                |
```

Register Allocation

Φ (Phi) The Φ IR instruction represents a merging of multiple temporaries into a new temporary. The temporaries to-be-merged are referred to as Φ's arguments.

Web A list of temporaries that ***must*** be stored in the same hardware register.

At this point, we will build our webs. To build webs, there are only a few simple rules.

For two IR instructions A and B, they *must* share the same register iff any one of the following are true:

- A is a Φ instruction and B is an argument of A.
- B is a Φ instruction and A is an argument of B.
- A and B are both arguments of the same Φ node.
- A and B are both arguments of a COPY instruction.

Conceptually, this all tracks. If two temporaries are merged, they must be stored in the same hardware register, otherwise, further accesses would require run-time knowledge during compile-time. If two temporaries are copied, why not just use the same register to refer to both? That way no actual hardware copy has to happen; the copy can just use the originally calculated temporary.

Webs from above excerpt:

```
w0: {r0}
w1: {r1}
w2: {r2}
w3: {r3}
w4: {r4}
w5: {r5}
```

These webs each contain a list of temporaries that *must* be stored in the same register. Keep in mind that these webs (and corresponding temporaries) *may* still be stored in the same register, if they do not interfere (calculated later).

What is web interference? Two webs, A and B are said to interfere iff any one of the temporaries within A inteferes with any one of the temporaries within B.

When two webs interfere, they *must* not be stored in the same register.