CSC-363 Lecture 01B
Phases of a Compiler

Colin McKinney

21 January 2026

Phases of a Compiler: Overview

- Scanner: takes source code and produces tokens
- Parser: takes tokens and produces abstract syntax tree
- Semantic Actions: takes AST and produces intermediate representation (IR)
- Optimizer: takes IR and optimizes to produce new IR
- Code generator: takes IR (or AST) and produces target code.

Some of these steps can be skipped or combined. Advantage of intermediate representations: they can be language and target agnostic, and hence more general. Can split the phases into “front end" (scan/parse/semantics) and “back end" (optimization and code generation).

Scanning

- Begins by taking source code and stripping out unnecessary characters (comments, whitespace).
- Convert groups of characters into tokens.

Source code:

```
int a;
int b;
b = a + 5;
```

As a char stream:

```
'i' 'n' 't' 'a' ';'...
```

As a token stream:

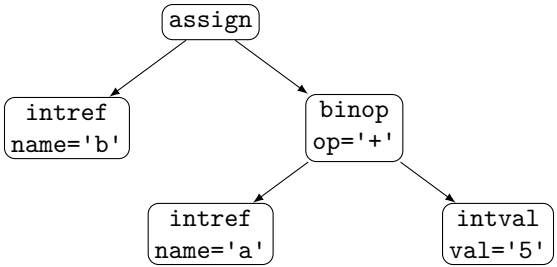
```
[int decl; name = 'a']
[int decl; name = 'b']
[int ref; name = 'b']
[assign]
[int ref; name = 'a']
[binop; op='+']
[int val; val = '5']
```

Parsing

We saw examples of parsing on Monday. We had the token stream

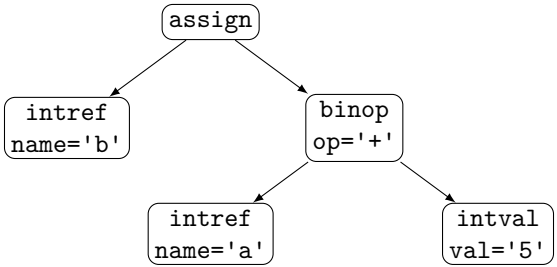
```
[intref;name='b'][assign][intref;name='a'][binop; op='+'][intval;val='5']
```

We need to parse this according to the grammar of our language (here C), to produce an AST



Semantic Analysis

We might need an auxiliary object to keep track of variables, such as a symbol table. We also might want to look at this tree before we do any code generation in order to verify that things make sense. For example, let's consider the previous parse tree



What if instead of `intval; val=5` we had `floatval; val=5.5`? Do we know that `b` has been declared? What about scope?

Intermediate Representation

Some compilers have a stage in between the AST and the code generation phase, called an intermediate representation (IR). For our example, it could look something like

```
a: int
b: int

t1 = load a
t2 = add t1, 5
store b, t2
```

This looks sort of like assembly code, but it's generic and architecture independent. Each line has (at most) three items, and is sometimes called Three Address Code (3AC).

Optimizations

```
 1  a: int
 2  b: int
 3  c: int
 4  t1 = load a
 5  t2 = add t1, 5
 6  store b, t2
 7  t3 = load b
 8  t4 = mul t3, 2
 9  store c, t4
10  t5 = load a
11  t6 = add t5, 5
12  store b, t6
13  call print, b
```

What do you notice about this code?

- Can any of it be eliminated?
- Should we analyze from bottom up or top down?
- What about if control structures are present?
- If we did passes of code like this, are we guaranteed to ever finish?

Register Allocation

Machines have a small number of registers. Suppose our machine has R0 and R1.

```
1  t1 = load a
2  t2 = load c
3  t3 = add t1, 5
4  t4 = mul t2, t3
5  store b, t4
```

How do we allocate registers? In this case we could do

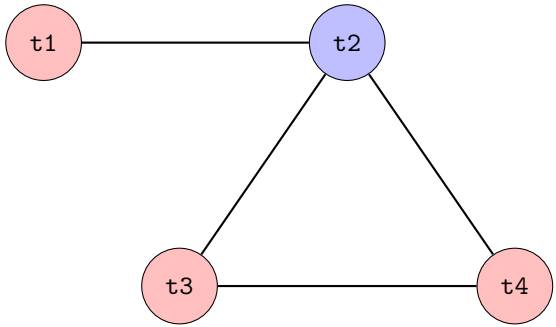
```
1  R0 = load a          ; t1: R0
2  R1 = load c          ; t2: R1
3  R0 = add R0, 5       ; t3: R0
4  R0 = mul R1, R0      ; t4: R0
5  store b, R0
```

We might have an issue if we can't load directly, e.g. we have to load a memory address and then do an indirect load: `LDR X0, =a`, `LDR X1, [X0]`.

Register Allocation and Graph Coloring

It turns out that register allocation is equivalent to graph coloring.

```
1  R0 = load a          ; t1: R0
2  R1 = load c          ; t2: R1
3  R0 = add R0, 5       ; t3: R0
4  R0 = mul R1, R0      ; t4: R0
5  store b, R0
```



Code Generation

IR:

```
1  t1 = load a
2  t2 = add  t1, 5
3  store b, t2
```

aarch64

```
1  ldr x1, =a
2  ldr w0, [x1]
3  add w0, w0, #5
4  ldr x1, =b
5  str w0, [x1]
```

RISC-V

```
1  la t1, a
2  lw t0, 0(t1)
3  addi t0, t0, 5
4  la t1, b
5  sw t0, 0(t1)
```

With our “real" compiler, we'll produce RISC-V code that uses an unlimited number of virtual `t` registers (contrast with the actual `t0`–`t5`). This will allow us to skip the register allocation step, because it can be very difficult. Reason: optimal graph coloring is NP-complete.