Little Languages

Lecture 10:

Dynamic Memory and Recursive Types in Rust

Before lecture: **Start VM and pull 590 materials from upstream.**
Then...

```
$ cd 590-material-<you>
$ git pull upstream master
$ cd 590-material-<you>/lecture/<today>
```

Announcements

- Midterm on Wednesday 2/20 to accommodate hacking or volunteering at Pearl Hacks!
- thdc Part 2 update: Division by 0 behavior:

```
$ thdc
9 0 / f
thdc: divide by zero
0
9
```

Little Languages for CS Diagramming

- Visualizations are frequently useful in computer science
  - For example, it's helpful to illustrate graphs and trees visually
- There is a long history of little languages to describe visualizations
  - In fact, Bentley's '88 paper where "Little Languages" was coined was a case study in Brian Kernighan's PIC language ('82)
- DOT is a diagramming language commonly used today
  - Graphviz ('91-) is a package of tools that processes DOT notation
  - Full Grammar: https://www.graphviz.org/doc/info/lang.html

The PIC Language

If you're talking about compilers, you might want to depict their behavior with a picture:

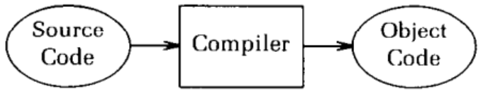
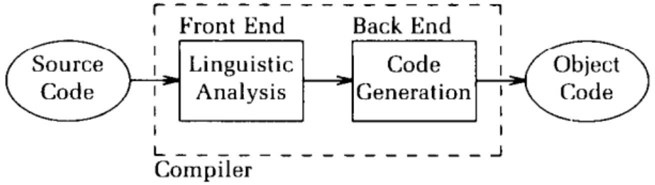


FIGURE 1. A Simple View of a Compiler

(This diagram is genuine PIC output; we'll see its input description shortly.) Or you may desire a little more detail about the internal structure:



UNIX is a trademark of AT&T Bell Laboratories.

DOT Grammar (simplified)

```
graph       -> "digraph {" stmt_list '}'
stmt_list   -> stmt (';' stmt_list)?
stmt        -> node_stmt | edge_stmt
node_stmt   -> node_id attr_list?
attr_list   -> '[' a_list ']'
a_list      -> ID '=' STRING (',' a_list)?
edge_stmt   -> node_id "->" node_id
node_id     -> ID (:port)?
```

Today we'll assume:

- *node IDs* are in the form of **n<#>**
- *a_lists* are either:
  - **label="<name of node>"**
  - **shape="record"** *(for interior nodes which have descendants)*

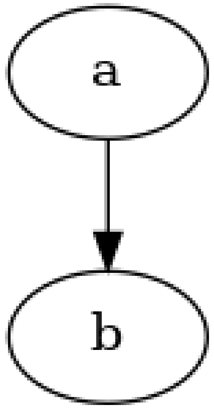
DOT Graph Example

```
digraph {

    n0 [label="a"];
    n1 [label="b"];

    n0 -> n1;

}
```




```
graph        -> "digraph {" stmt_list '}'
stmt_list    -> stmt ';' stmt_list?
stmt         -> node_stmt | edge_stmt
node_stmt    -> node_id attr_list?
attr_list    -> '[' a_list ']'
a_list       -> ID'='STRING(',' a_list)?
edge_stmt    -> node_id "->" node_id
node_id      -> ID (:port)?
```

- The DOT string above produces the simple directed graph (digraph) shown.
- Using the example above, let's relate the tokens with the grammar.

Hands-on: Produce the Graphic Right

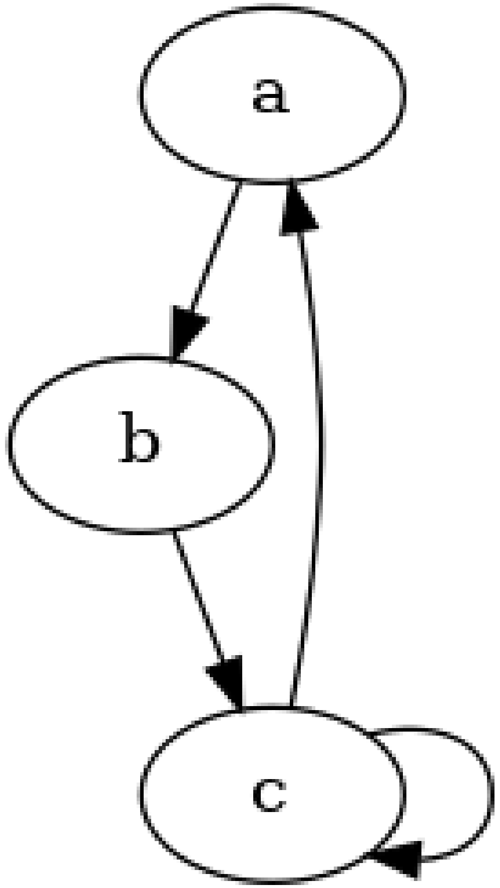
- Change directories to today's lecture and then into the 00_dot directory. Open 00_digraph.dot in vim.

- To generate the graphic file, run the command in vim
  - :! ./make_digraph

- On your host machine, open the folder of your VM and look for the file lec11_dot_output - drag this file into a web browser.

- Try editing the file, saving, rerunning the command above, and refreshing your browser until you've reproduced the diagram right.

- Check-in on PollEv.com/compunc when complete

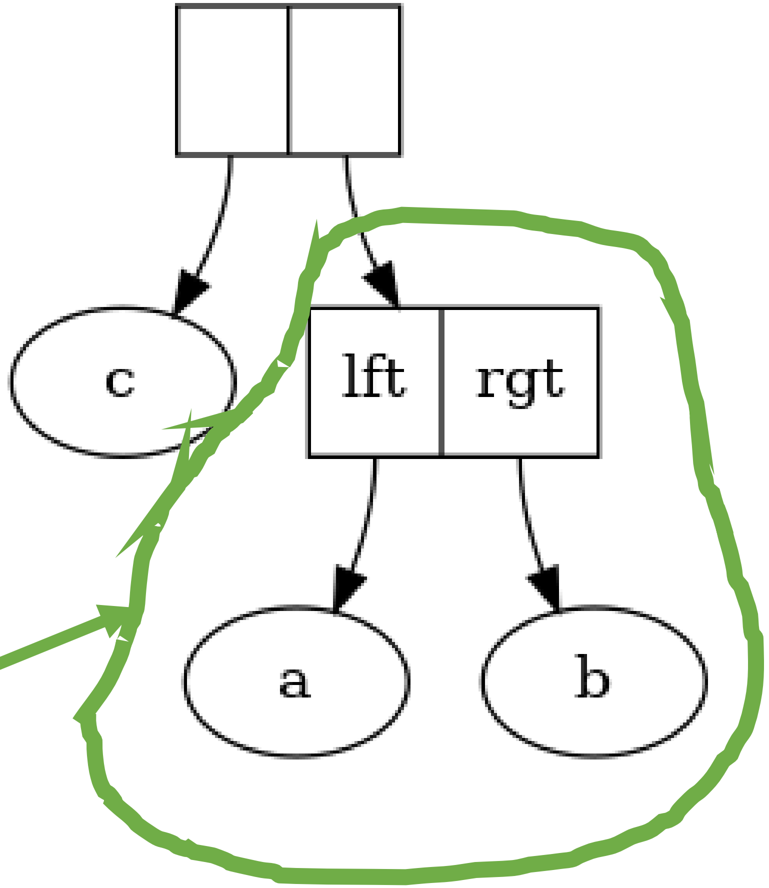


Follow Along: The **record** Shape and "Ports"

- Having "records" with cells is often useful in diagramming.
- DOT's label strings for the record shape have their own *little language* for adding "ports" via <port_name> separated by '|'s
- You can then connect edges from or to a "port" by adding :<port_name> after the node id as shown below.
- Let's try extending the 01_record.dot file to produce the visualization right.

```
digraph {
    n0 [label="<l>lft|<r>rgt" shape="record"];
    n1 [label="a"];
    n2 [label="b"];
    n0:l -> n1;
    n0:r -> n2;
}
```

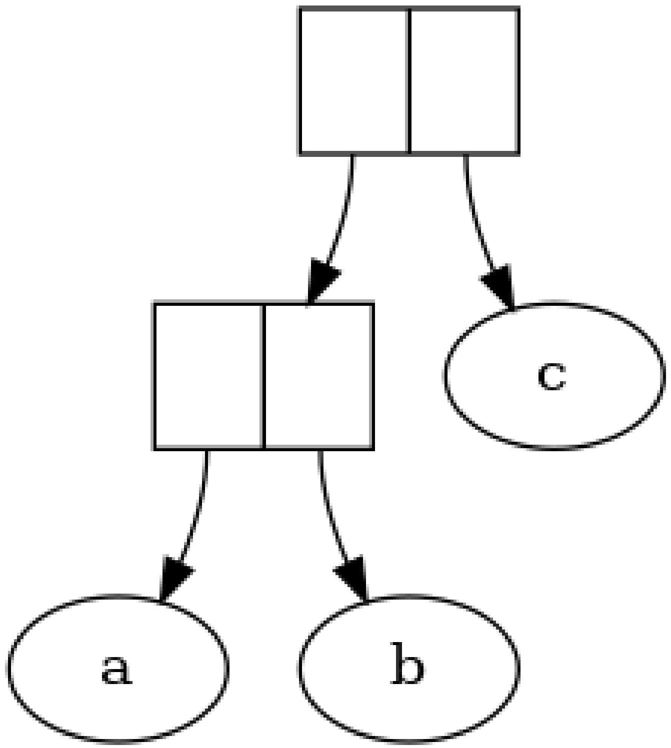
Produces



Visualizing LISP-like Data Structures in DOT

```
enum Value {
    Char(char),
    Pair(Box<Value>, Box<Value>),
}
```

- Suppose every **`Value`** is defined as above.
- Assume *cons* is a function that produces `Value::Pairs` by boxing its arguments.
- We want to produce the diagram right given the Value produced with cons below:




```
cons(cons(Char('a'), Char('b')), Char('c'))
```

Emitting DOT Code Programmatically

- Our goal is to take a data structure in our program (produced above) as input
- And **emit** (produce) the DOT code right programmatically.
- What challenges do we face?
    - How might we do this algorithmically?

```
cons(cons(Char('a'), Char('b')), Char('c'))
```

```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

DotGen - Helper Struct for our DOT Problem

To simplify some of the book keeping for emitting DOT file strings, I've setup a DotGen helper struct with some methods to emit code.

```
fn emit_pair(&mut self) -> usize
```

Emits a Pair node (record) and returns its ID#

```
fn emit_char(&mut self, label: char) -> usize
```

Emits a Char node (ellipse) and returns its ID#

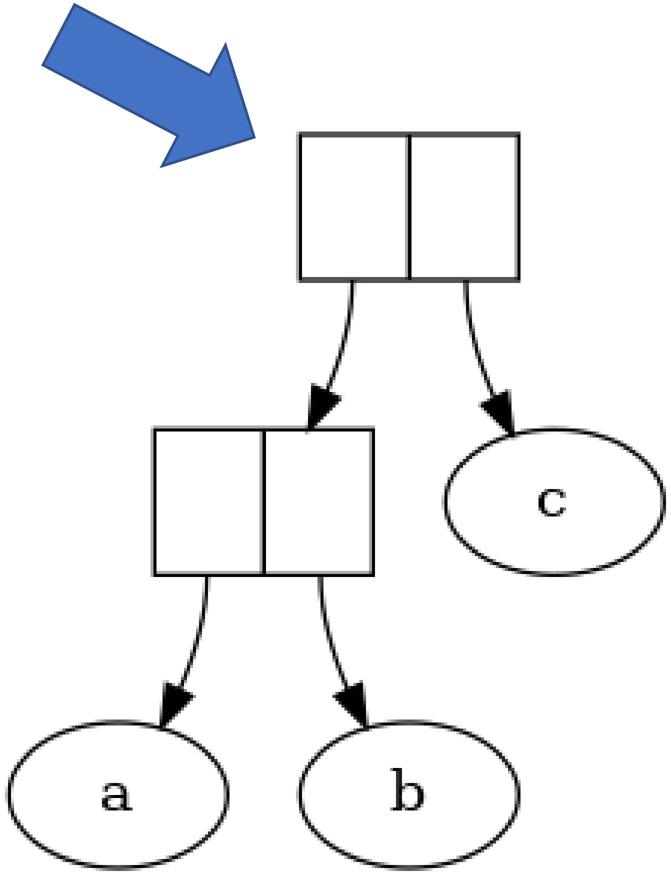
```
fn emit_edges(&mut self, pair: usize, lhs: usize, rhs: usize)
```

Emits edges to connect pair ID to lhs and rhs IDs.

```
fn to_string(&mut self) -> String
```

Returns a complete DOT file String containing all pairs, chars, & edges emitted.

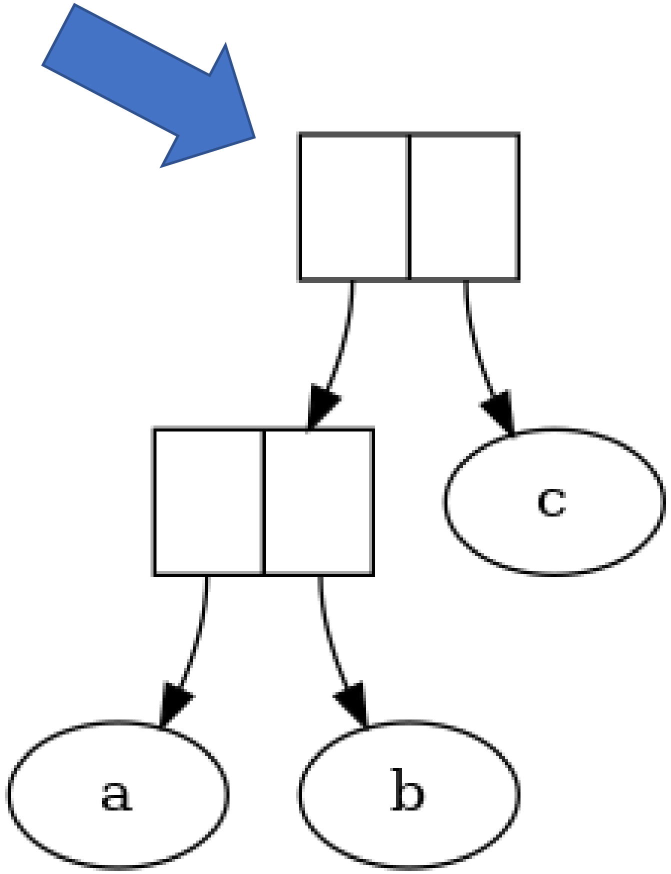
Walking our structure recursively




```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

"Walk this way." -Aerosmith

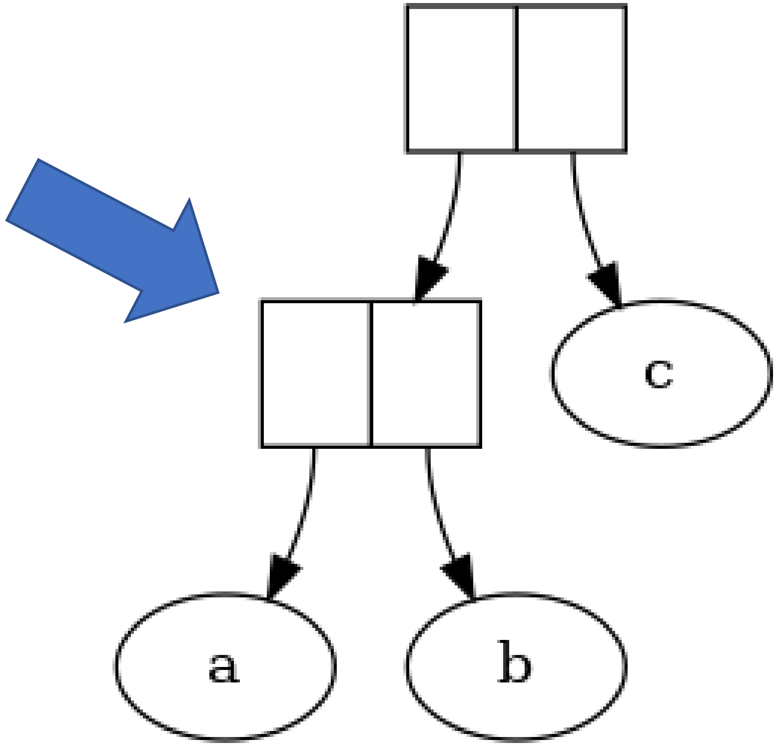
Visiting a Pair: Emit a Pair Node (Record)




```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

Then go visit the left hand side.

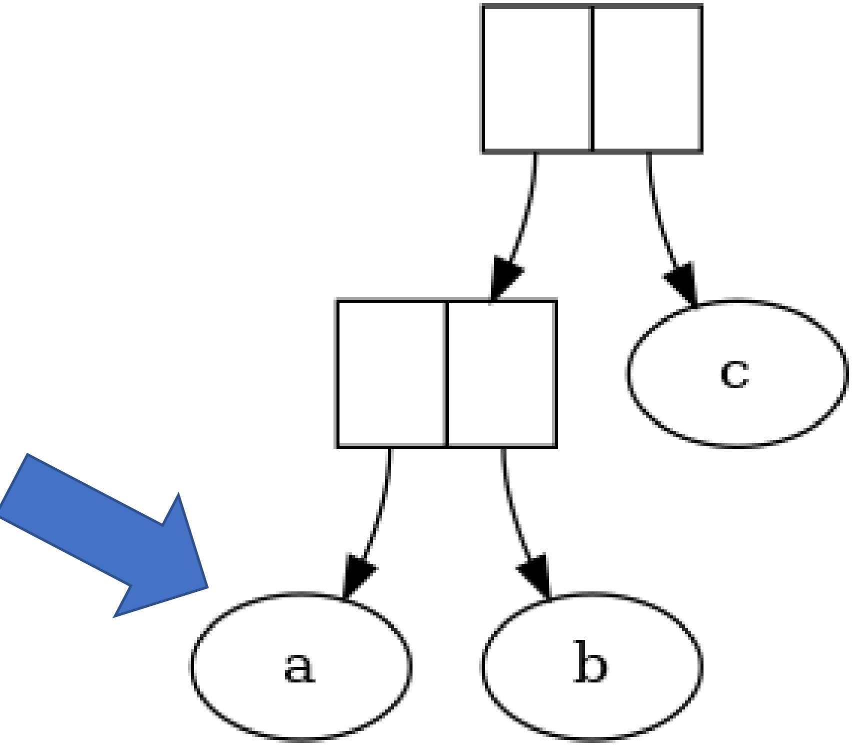
Visiting a Pair: Emit a Pair Node (Record)




```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

Then go visit the left hand side.

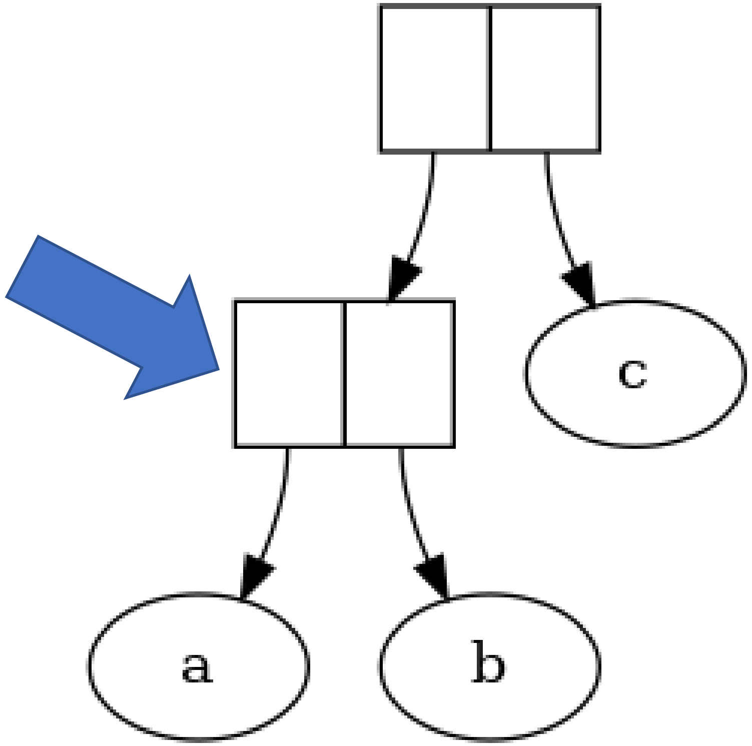
Visiting a Char: Emit a Char Node




```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

Return your ID back to parent.

Completed Left Hand Side Visit: Record lhs_id

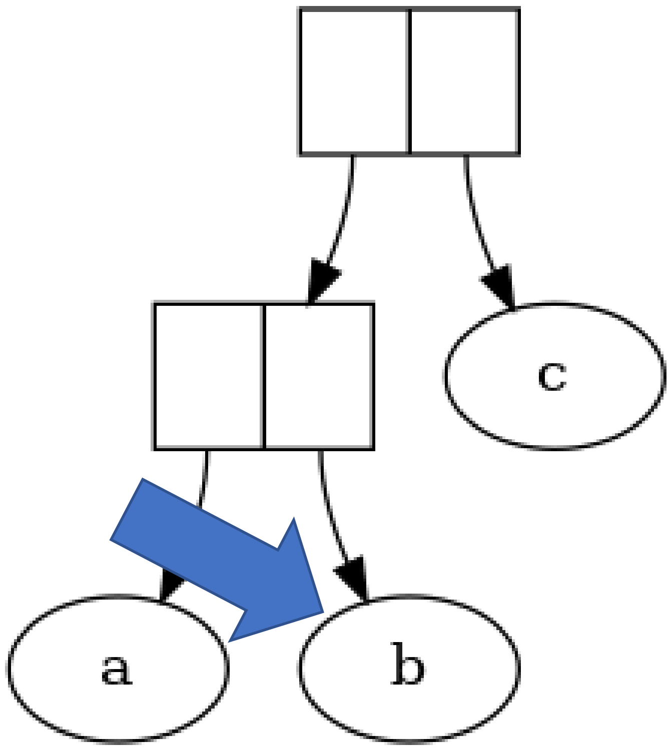



```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

lhs: n2

Then go do the same with right hand side.

Visiting a Char: Emit a Char Node

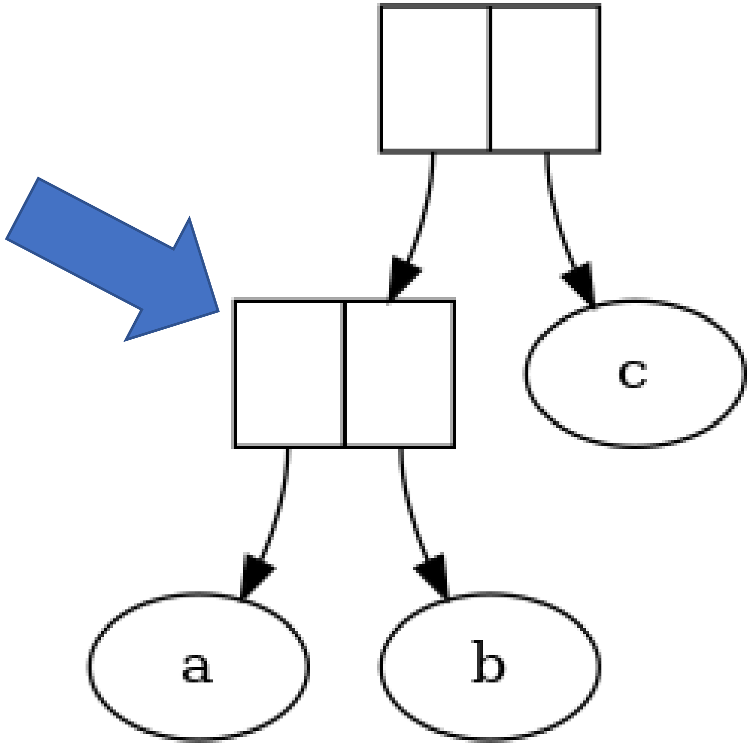



```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

lhs: n2

Return your ID back to parent.

Completed Right Hand Side Visit: Emit Edges

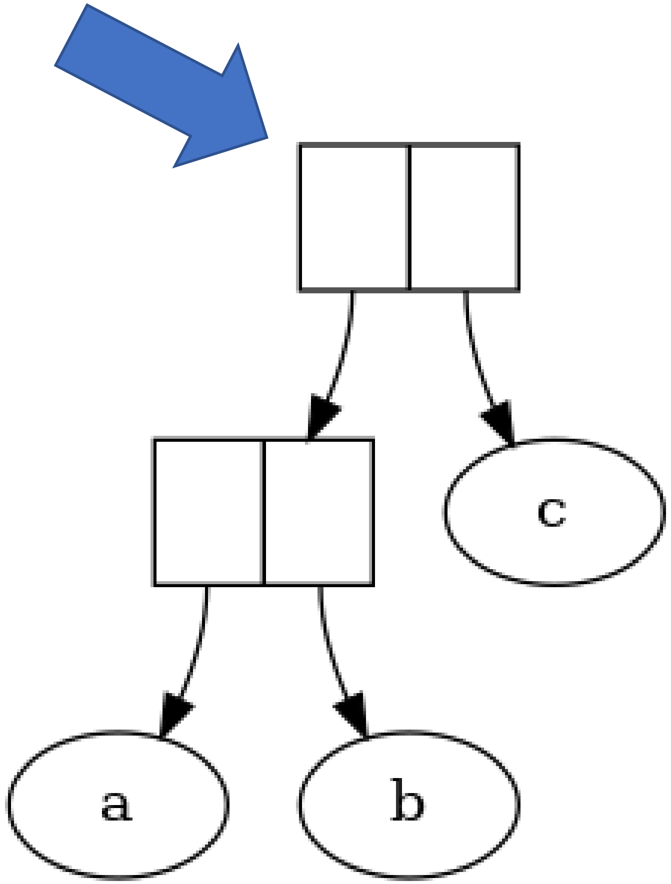



```
digraph {
	n0 [label="<l>|<r>", shape="record"];
	n1 [label="<l>|<r>", shape="record"];
	n2 [label="a"];
	n3 [label="b"];
	n1:l -> n2;
	n1:r -> n3;
	n4 [label="c"];
	n0:l -> n1;
	n0:r -> n4;
}
```

lhs: n2   rhs: n3

Connect from current Pair node to two children based on their generated IDs.

Completed Pair: Return Pair ID to Parent

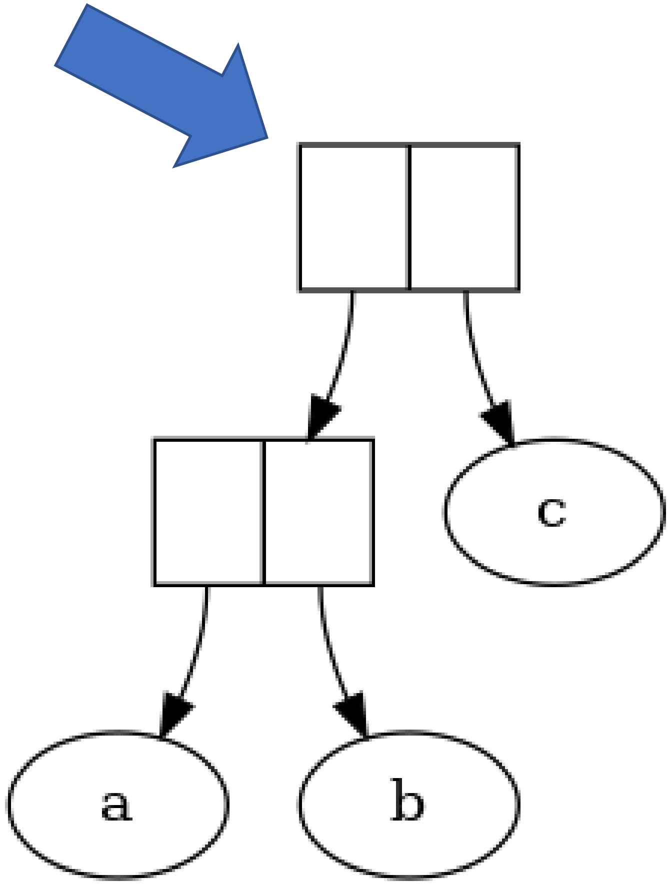



```
digraph {
	n0 [label="<l>|<r>", shape="record"];
	n1 [label="<l>|<r>", shape="record"];
	n2 [label="a"];
	n3 [label="b"];
	n1:l -> n2;
	n1:r -> n3;
	n4 [label="c"];
	n0:l -> n1;
	n0:r -> n4;
}
```

lhs: n1

Now that we've completed the left of the root node, we record its lhs_id as n1.

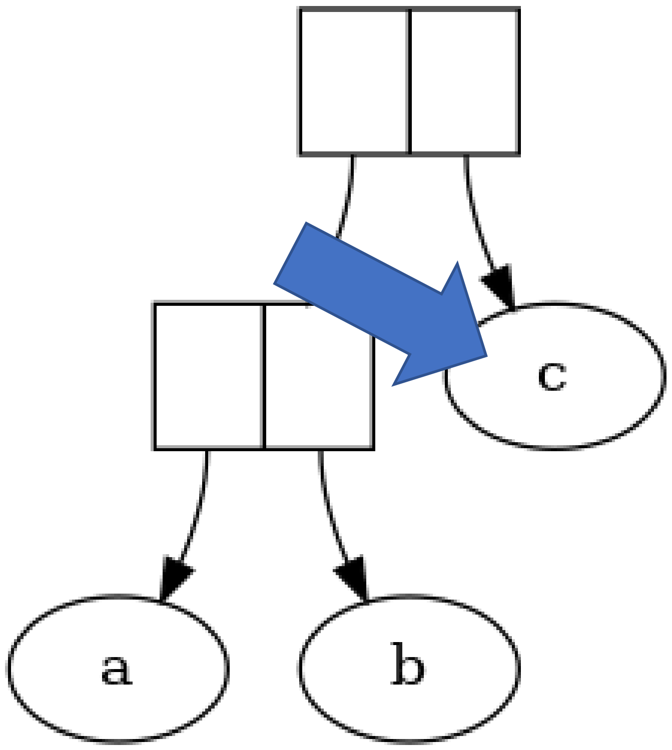
Visit Right Hand Side




```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

lhs: n1

Visiting a Char: Emit a Char Node

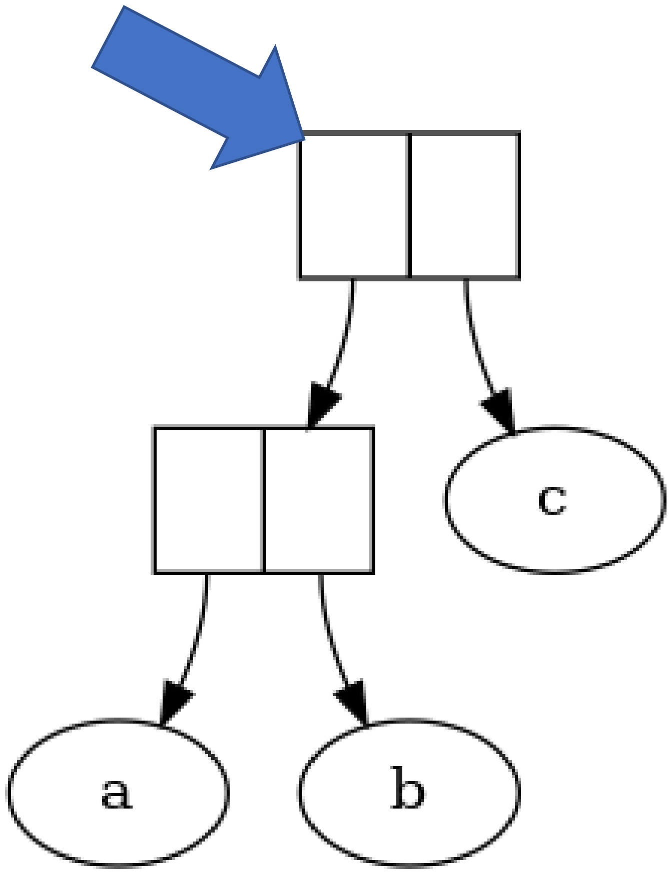



```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

lhs: n1

Return your ID back to parent.

Completed Right Hand Side Visit: Emit Edges




```
digraph {
        n0 [label="<l>|<r>", shape="record"];
        n1 [label="<l>|<r>", shape="record"];
        n2 [label="a"];
        n3 [label="b"];
        n1:l -> n2;
        n1:r -> n3;
        n4 [label="c"];
        n0:l -> n1;
        n0:r -> n4;
}
```

lhs: n1 rhs: n4

Connect from current Pair node to two children based on their generated IDs. Fin.

Follow Along: Recursive Walk

- Let's implement a *visit* function to recursively walk the tree and emit DOT constructs for *any* Value. We'll do our work in `<lec11>/01_cons/src/main.rs`

- Algorithm Overview:
  - Base Case - We're visiting a Char node. Emit the char and return node id.
  - Recursive Case - We're visiting a Pair node.
    1. Emit a Pair record, record its returned id.
    2. Recursively visit the left-hand side. Record its returned id.
    3. Recursivley visit the right-hand side. Record its returned id.
    4. Emit edges from pair id to lhs and rhs ids.
    5. Return the pair id.

- Intuition: Each visit to a Value is responsible for emitting itself, visiting its descendants, and returning its own id.

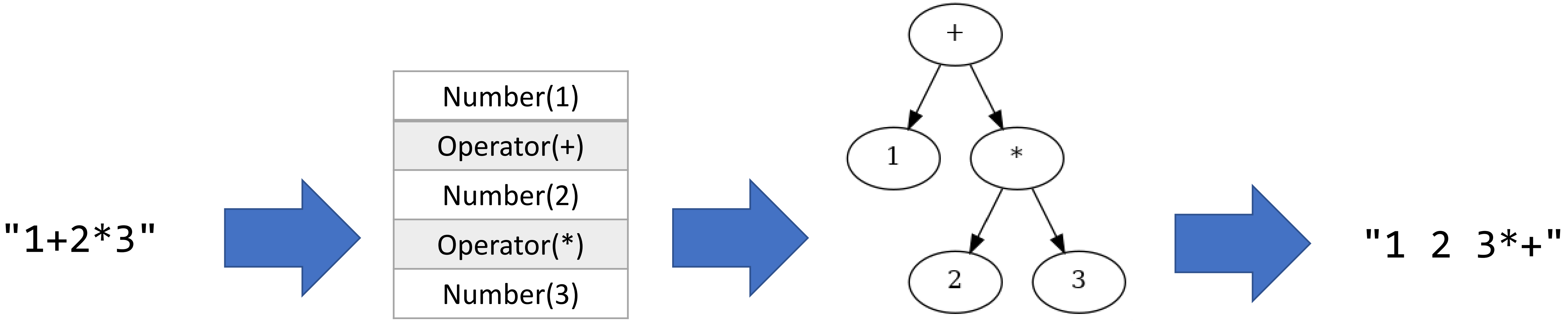
- We can use the script `./make_diagram` to run our program and generate the graphic.

Visit Solution

- Notice how cleanly the overview of the algorithm is able to translate into respective code

```
match val {
    Char(c) => dot.emit_char(c),
    Pair(lhs, rhs) => {
        let pair_id = dot.emit_pair();
        let lhs_id = visit(dot, *lhs);
        let rhs_id = visit(dot, *rhs);
        dot.emit_edges(pair_id, lhs_id, rhs_id);
        pair_id
    }
}
```

What's the big picture?



Tokenization

Input characters are transformed into meaningful tokens.
(Part 1 of thdc.)

Parsing

Data structures are built-up to represent the relationships between tokens.
(We're doing this next.)

Code Generation

Finally, an algorithm visits the hierarchy to generate some alternative representation.
(What we did today.)

This is effectively how compilers read your programs and emit machine code!