1 Optimization (Continued)

1.1 Optimization: Register Allocation

1.1.1 Restrictions

One of the main restrictions of the algorithm for register allocation is simply temporary values: what do we do with register allocation for temporary values that don't have any names?

1.2 Intermediate Representation

Let's consider the following program:

```
(+ (- 5 input) (* input (if (> 0 input) 1 -1)))
```

Below is a transformed version of the program where every nested expression that would have introduced a temporary is now a `let`-bound variable.

```
(let (tmp 1 (- t input))
    (let (tmp 2 (> 0 input))
        (let (tmp3 (if tmp2 1 -1))
            (let (tmp4 (* input tmp3))
                (+ tmp1 tmp4)
            )
        )
    )
)
```

This makes the order of operations very explicit. Notice that it's very clear that we're doing left-to-right operation. This process also makes code generation for operations a lot easier, since everything is already stored on the stack or in a register.

(Example.) Perform the transformation on the function

```
(fun (sumsquares x y)
    (+ (* x x) (* y y)))
```

As mentioned, we want to break our complex expression into much simpler types. We can accomplish this by defining any computations into `let`-bindings. This gives us

```
(fun (sumsquares x y)
    (let (val (* x x))
        (let (val2 (* y y))
            (+ val val2)
        )
    )
)
```

This transformation is fairly common, and there is a fairly standard algorithm for this transformation that takes every non-trivial or non-atomic (basically, everything that's not a literal value) and puts them in a `let`-binding.

1.2.1 Different Grammar Forms

There are two types of grammars we want to consider.

- **A-Normal Form:** This is essentially our grammar as is. This cares about scope, binding, order or evaluations, and so on.

```
<expr> := <number> | <id> | true | false | nil
    | (+ <expr> <expr>)
    | (- <expr> <expr>)
    | (if <expr> <expr> <expr>)
    | (break <expr>)
    | ...
    | (let (<id> <expr>) <expr>)
    | ...
```

- **ANF-Restricted Grammar:** This grammar is effectively the transformed grammar; that is, given our A-Normal Form, we can transform it into a more restricted version. We'll denote this as `AExpr`.

```
<val> := <number> | <id> | true | false | nil
<expr> := (+ <val> <val>)
    | (pair <val> <val>)
    | (if <val> <block> <block>)
    | (break <val>)
<block> := (let (<id> <expr>) <block>)
    | (loop <block>)
    | (break <block>+)
    | ...
    | <expr>
    | <val>
```

  This is restricted in the sense that the grammar is broken up into three different groups (productions). You have expressions (`<expr>`) that form blocks (`<block>`). Blocks have expressions in them that can perform calculations; so, we can think of `<expr>` as something that performs a calculation of some type (e.g., binary operations, creating a new pair, and so on). All the arguments to these expressions that do calculations must be primitive values or identifiers.

Remark: Loops are an interesting case to think about here.

1.2.2 Going from Normal to Restricted

How do we create an algorithm that transforms code written under one grammar to code written in the restricted grammar? Note that this is a very standard algorithm, so we'll mainly gain some intuition. One choice to make is whether we want to introduce a new `enum` for ANF expressions. For example, is it worth it to introduce a bunch of new `enum`s like shown below?

```
enum Val {
    VNum,
    VId(String),
    ...
}

enum AExpr {
    APlus(...),
    APair(...),
    ...
}

enum Block {
    BLet(...),
    ...
}
```

In any case, we can write a few functions to facilitate the conversion process.

- `anf_to_val(e: &Expr) -> Val`: Converts an A-Normal Expression Form to a literal value under the ANF-Restricted Expression Form.
- `anf_to_expr(e: &Expr) -> AExpr`: Converts an A-Normal Expression Form to a computation expression under the ANF-Restricted Expression Form.
- `anf_to_block(e: &Expr) -> Block`: Converts an A-Normal Expression Form to a block expression under the ANF-Restricted Expression Form.

1.2.3 ANF to Value

Let's suppose we want to implement

```
fn anf_to_val(e: &Expr) -> Val {
    match e {
        ...
    }
}
```

- For an expression `e` like `Number(n)`, we can trivially return `VNum(n)`.
- For an expression `e` like `Plus(e1, e2)`, this becomes more complicated. This will involve a nested plus expression, and we should return an identifier that we can use later. Note that `e1` and `e2` may be complicated expressions, but we want them to be `Val`s as well. So, we'll probably need to do some recursive calls to ideally break `e1` and `e2` down into `let`-bindings.

  ```
  Plus(e1, e2) => {
      let (v1, b1) = anf_to_val(e1);
      let (v2, b2) = anf_to_val(e2);
      // Assume new_label() is a function that returns a new identifier.
      let new_name = new_label();
      // This isn't Rust syntax, but basically we want all the bindings from
      // b1, all the bindings from b2, and our new binding in this vector.
      (VId(new_name), vec![...b1, ...b2, (new_name, APlus(v1, v2))])
  }
  ```

  Notice how we're returning a tuple. The first element is the identifier that will store the result of the evaluation of this expression. However, we also need to return list of `let`-bindings we need to eventually stick in front of *this* identifier in order for it to work (otherwise, the identifier won't be bound). So, we should modify the function return type:

  ```
  fn anf_to_val(e: &Expr) -> (Val, Vec<(String, AExpr)>) { ... }
  ```