Basic Blocks

We can decompose any program into a control flow graph of basic blocks. A basic block is a contiguous series of instructions such that:

- Entry into the basic block only happens at the start of the basic block
- Exit out of the basic block only happens at the end of the basic block

Because the instructions in a basic block are always executed together, it's much simpler to perform optimizations within basic blocks.

Value Numbering within Basic Blocks

First, we write the original code we have into a form where every calculated value is placed in a different virtual register. Let's look at an example block.

```
v1 := a
v2 := a
v3 := 17
v4 := v1 + v3
v5 := v2 + v3
a := v5
```

We keep track of a dictionary and process the block line by line.

Value Numbering within Basic Blocks

```
v1 := a
v2 := a
v3 := 17
v4 := v1 + v3
v5 := v2 + v3
a := v5
```

key	value
a	v1

a isn't available anywhere so we have to load it.

```
v1 := a
```

Value Numbering within Basic Blocks

```
v1 := a
v2 := a
v3 := 17
v4 := v1 + v3
v5 := v2 + v3
a := v5
```

key	value
a	v1
v2	v1

a is already available in v1.
We can replace all future instances of v2 with v1.

No instruction to be generated here.

```
v1 := a
```

Value Numbering within Basic Blocks

```
v1 := a
v2 := a
v3 := 17
v4 := v1 + v3
v5 := v2 + v3
a := v5
```

key	value
a	v1
v2	v1
v3	17

We can replace all future uses of v3 with the constant 17 directly.

```
v1 := a
```

Value Numbering within Basic Blocks

```
v1 := a
v2 := a
v3 := 17
v4 := v1 + v3
v5 := v2 + v3
a := v5
```

key	value
a	v1
v2	v1
v3	17
v1 + 17	v4

```
v1 := a
v4 := v1 + 17
```

First, we consult the dictionary to process v1 + v3. v3 is already in there, so we replace it with 17.

Since v1 + 17 wasn't computed before, we make note that we can find that value in v4.

Value Numbering within Basic Blocks

```
v1 := a
v2 := a
v3 := 17
v4 := v1 + v3
v5 := v2 + v3
a := v5
```

key	value
a	v1
v2	v1
v3	17
v1 + 17	v4
v5	v4

```
v1 := a
v4 := v1 + 17
```

Now, we consult the dictionary to process v2 + v3. v2 and v3 are already in there, so we replace it with v1 + 17.

v1 + 17 was computed before in v4! We can replace all instances of v5 with v4.

Value Numbering within Basic Blocks

```
v1 := a
v2 := a
v3 := 17
v4 := v1 + v3
v5 := v2 + v3
a := v5
```

key	value
~~a~~	~~v1~~
v2	v1
v3	17
v1 + 17	v4
v5	v4
a	v4

```
v1 := a
v4 := v1 + 17
a := v4
```

We are storing the value of virtual register v5 (whose value is found in v4) into a. We need to invalidate the old entry saying that the value of a can be found in v1, and place a new entry.

Records in Dynamic Languages

- Assumption: all values have a size of one word
- Simplest approach is a tuple/fixed-length array "object"
  - Stores N "slots" of equal size
  - Read/write access is by index
  - Tuples of various sizes can be created, so the runtime stores N at the beginning of the memory block to check bounds
  - Total size: N+1 words
- Need to know what type is in each slot
  - Typical approach: values are tagged, so self-describing
  - If values aren't tagged, then you need a tag for each slot (does it store an int, bool, or object?)
    - Naively uses 2N+1 words, but can compress to 1 byte or a couple of bits per slot

Records in Dynamic Languages

- More complicated approach:
  - Stores N fields
  - Layout: first the size (N) then an array of pairs of words
    - The first element is a pointer to the field name (typically a string, but a compiler could assign numbers if all field names are known)
    - The second element is the actual value
  - Size used: 2N+1 words
  - Same tagging issues apply, could result in using more space

Implementing allocation

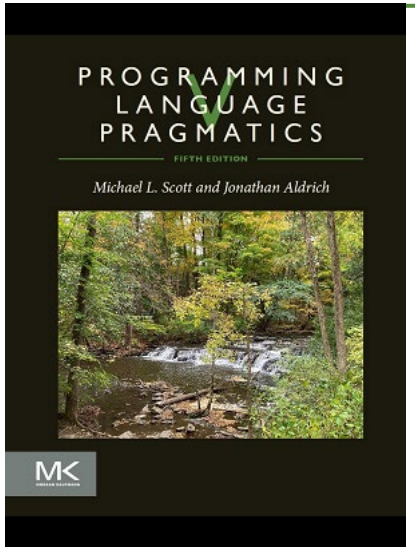
Simple "bump allocator" approach

1. Get a big chunk of memory from the operating system
   - Or from Rust, in this class
2. Pick a register for allocation, initialize it to the start of the chunk
   - Callee-save registers are good choices, if you call an external function they won't get messed up.
3. When allocating:
   a. Make sure there is enough room left in the chunk to allocate. If not, throw an error, or garbage collect (see below)
   b. Copy the allocation register to the new pointer
   c. Increment (bump) the allocation register by the specified amount

There are many more sophisticated approaches!

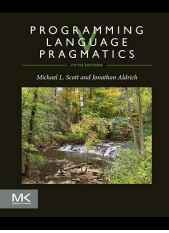
- We'll discuss garbage collection and reference counting in the next lecture
  - guest lecture by my postdoc, whose research focuses on GC

Section 8.5 part 2: Recursive Types



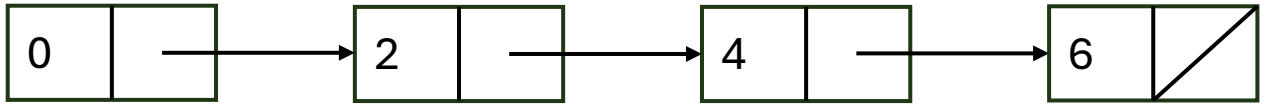
Programming Language Pragmatics, Fifth Edition

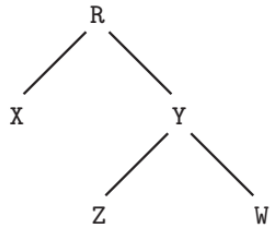
Michael L. Scott and Jonathan Aldrich



Recursive types

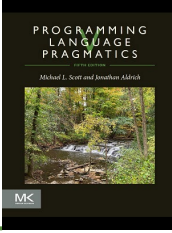
- Definition: a recursive type is one whose objects may contain one or more references to other objects of the type
- Support common heap data structures such as lists, trees



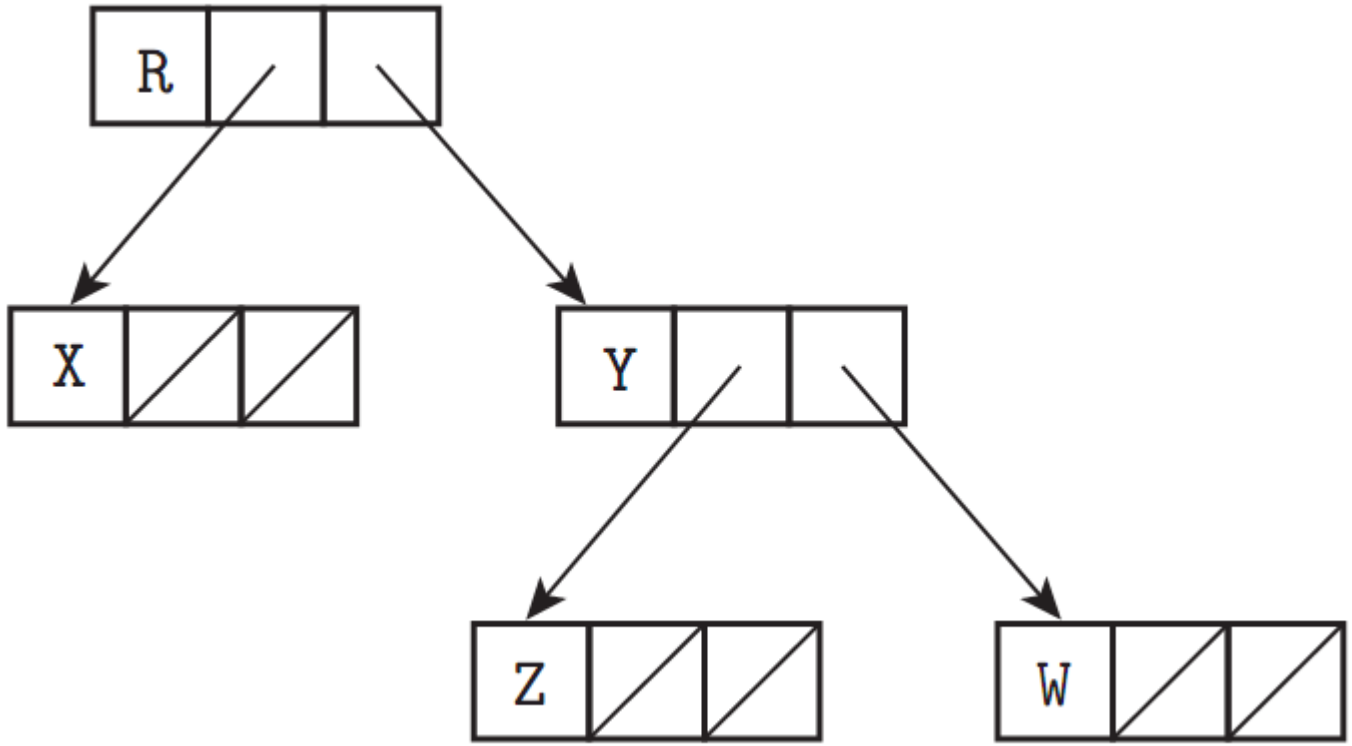


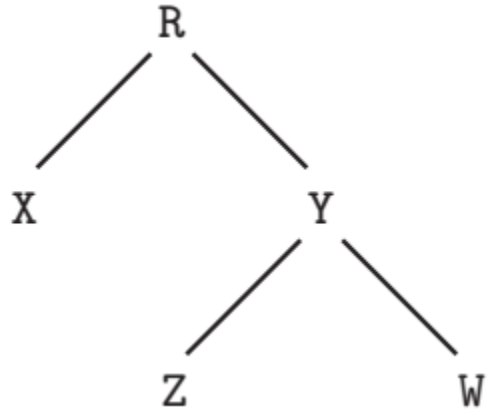
- Recursive types are usually combined with a record (and maybe union) type
  - The type typically needs to hold something in addition to the recursive reference
  - There needs to be a non-recursive variant of the type as the base case for recursion
- Examples
  - Classes are unions, records, and recursive types
  - ML datatypes are unions and recursive types
    - records are separate but often used with datatypes
  - Languages with null can encode simple unions by using a null pointer/reference
    - null represents an empty base case, so an explicit union construct may not be needed

Recursive tree data structure in a language with pointers



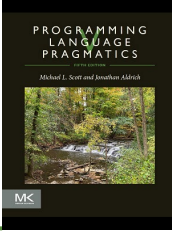
- Each node in the data structure refers to left and right nodes of the same type (or null, denoted with a slash)



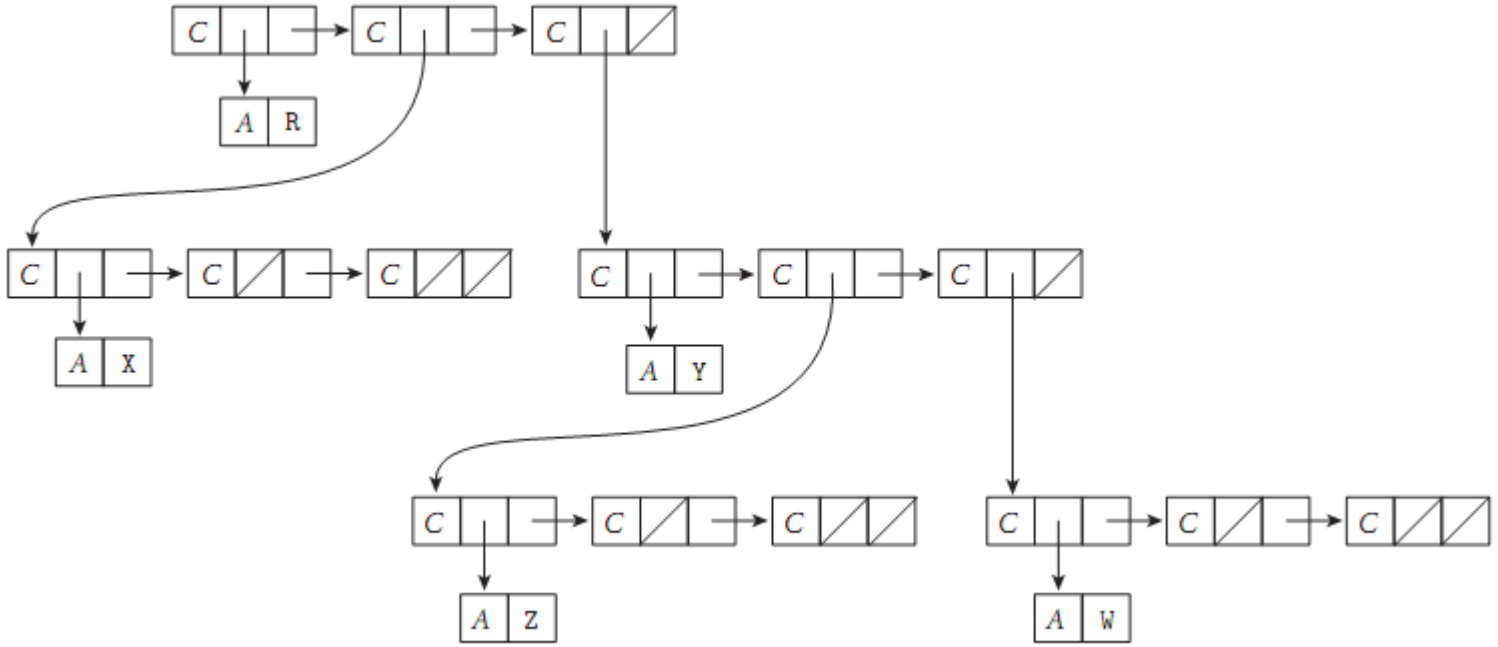


Abstract (conceptual) tree

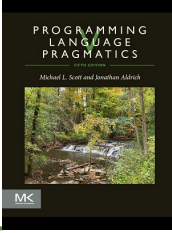
Recursive tree data structure in Lisp



- Pointers are implicit as Lisp uses a reference model of variables
- Recursive data structures are typically built of cons cells
- The tag (C or A) distinguishes between a cons cell and an atom



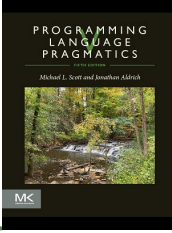
Recursive datatype declarations in OCaml



- The following declares an integer list type in OCaml:

```
type IntList =
    Cons of { value:int, next:IntList }
|   Nil
```

- This is a recursive type because `IntList` is used within its type definition (for the `next` field)
- The | denotes a union type, and { } denote a record type
  - As mentioned earlier, these are often combined with recursive types



Modeling recursive types

- To understand recursive types better, let's identify their essence separately from records and unions
- We'll add a recursive type **`rec`** $T.\tau$ to our grammar.
  - T is the name of the recursive type
  - τ is the type's definition
  - T can appear in τ

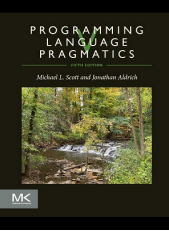
$$\tau \quad ::= \quad \ldots \mid \textbf{rec}\ T.\tau \mid T$$

- Now we can model lists as follows:

```
rec List . union { Cons: { val:int, next:List }, Nil:unit }
```

  - We use a recursive types, a union type, and a record type
  - `unit` is a type that contains no data – conceptually a record with no fields

Unfolding recursive types

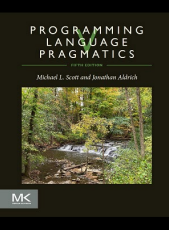


- A recursive type is conceptually equivalent to its definition
  - We can ***unfold*** a recursive type by replacing the type name with its definition

$$unfold(\textbf{rec}\ \ T.\tau) = [\textbf{rec}\ \ T.\tau/T]\tau$$

  - The notation [**rec** *T*.τ/*T*]τ means τ, but with **rec** *T* substituted for *T* wherever it appears

Unfolding recursive types



- A recursive type is conceptually equivalent to its definition
  - We can ***unfold*** a recursive type by replacing the type name with its definition

$$unfold(\textbf{rec}\ \ T.\tau) = [\textbf{rec}\ \ T.\tau/T]\tau$$

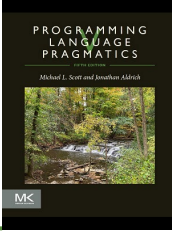
- Here's an example:

```
rec List . union { Cons: { val:int, next:List }, Nil:unit }
```


```
union { Cons: { val:int,
                next:rec List . union { Cons: { val:int, next:List },
                                        Nil:  unit } },
        Nil:  unit }
```

- A symmetric ***fold*** operation replaces the body of a recursive type with the recursive type itself

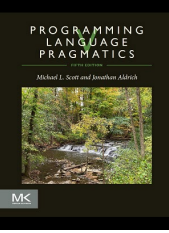
How do we know when to fold/unfold?



- Conceptually we can unfold as many times as we want
  - but it's not practical (the fully unfolded type would be infinite)
  - in practice, type checkers unfold when needed
  - but they need hints--this is why practical languages don't provide a recursive type by itself

- Thus, recursive types are combined with unions (and often fields)
  - ML datatypes, C structs, Java classes
  - Operations on the combined type trigger folds and unfolds
  - A fold operation is inserted when creating a class, struct, or datatype instance
  - An unfold operation is inserted when pattern matching (ML) or accessing a field (C or Java)

Recursive type syntax



- We extend our syntax with:
  - recursive types,

$$\tau \quad \longrightarrow \quad \ldots \mid \texttt{rec}\ T.\tau \mid T$$

  - fold and unfold operations,

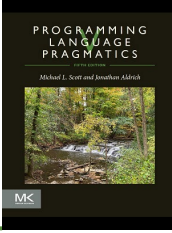
$$e \quad \longrightarrow \quad \ldots \mid \texttt{fold}_{\texttt{rec}\ T.\tau}\ e \mid \texttt{unfold}\ e$$

  - and folded values

$$v \quad \longrightarrow \quad \ldots \mid \texttt{fold}_{\texttt{rec}\ T.\tau}\ v$$

- Our fold operation keeps track of the folded recursive type, for bookkeeping

Recursive type semantics



- Folded data structures are values, so we only need to add a rule for unfolding these values to get at their contents:

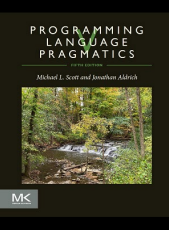
$$\frac{E \vdash e \Downarrow \texttt{fold}_{\texttt{rec}\,T.\tau}\ v}{E \vdash \texttt{unfold}\ e \Downarrow v}\ \textit{ev-unfold}$$

- The typing rules convert between folded and unfolded representations of types:

$$\frac{\Gamma \vdash e : \texttt{rec}\ T.\tau}{\Gamma \vdash \texttt{unfold}\ e : [\texttt{rec}\ T.\tau/T]\tau}\ \textit{t-unfold} \qquad \frac{\Gamma \vdash e : [\texttt{rec}\ T.\tau/T]\tau}{\Gamma \vdash \texttt{fold}_{\texttt{rec}\,T.\tau}\ e : \texttt{rec}\ T.\tau}\ \textit{t-fold}$$

- The notation [**rec** $T.\tau/T$]τ means τ, but with **rec** T substituted for T wherever it appears

Recursive type subtyping

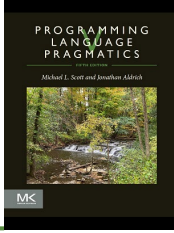


- The main rule for recursive type subtyping is analogous to depth subtyping
  - One recursive is a subtype of another if their definitions have a corresponding subtyping relationship, assuming in the check that the recursive type variables are subtypes

$$\frac{\Gamma, T <: T' \vdash \tau <: \tau'}{\Gamma \vdash \mathtt{rec}\ T.\tau <: \mathtt{rec}\ T'.\tau'}\ \textit{s-rec} \qquad \frac{T <: T' \in \Gamma}{\Gamma \vdash T <: T'}\ \textit{s-assume}$$

  - The second rule, *s-assume*, allows us to use this assumption
  - We extend the definition of Γ to allow subtyping assumptions

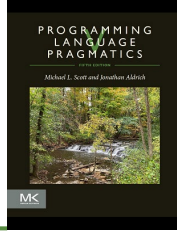
Check your understanding: recursive types



- Why are recursive types typically combined with union and record types?

- (press pause for more time)

Check your understanding: recursive types



- Why are recursive types typically combined with union and record types?

- Answer: the recursive type typically stores data, and fields are needed for that. Unions are needed to ensure the data structure has a non-recursive base case. In languages with null pointers, null can be used to represent an (empty) base case instead of an explicit union.