Compiler Design

by dcamenisch

A compiler translates one programming language to another. The simplified compiler has the following structure:

- Lexical Analysis: Source Code → Token Stream
- Parsing: Token Stream → AST
- Intermediate Code Generation: AST → Intermediate Code
- Code Generation: Intermediate Code → Target Code

The first two steps are the frontend and machine independent, the last step is the backend and machine dependent.

x86lite

x86lite memory consists of 2^{64} bytes numbered `0x00000000` through `0x0xffffffff`, split into 8-byte quadwords (has to be quadword-aligned).

The stack grows from high addresses to low addresses, `rsp` points to the top of the stack, `rbp` points to the bottom of the current stack frame.

The stack sits at the top of memory space, at the bottom we have code and data followed by the heap.

Registers: `rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, rip, r08 - r15`

Flags: `OF` overflow/underflow, `SF` sign (1 = negative), `ZF` zero

Condition Codes:

Code	Condition
e (equality)	`ZF`
ne (not equals)	not `ZF`
g (strictly greater)	not `ZF` and `SF` = `OF`
l (strictly less)	`SF` ≠ `OF`
ge (greater or equal)	`SF` = `OF`
le (less or equal)	`SF` ≠ `OF` or `ZF`

Instructions: `INSTR SRC DEST` (AT&T syntax), prefix register with % and immediate values with $. Note that `subq` is `DEST - SRC`.

Operands:

- `Imm`: 64-bit literal signed integer
- `Lbl`: label representing a machine address
- `Reg`: one of the registers, the value is its content
- `Ind`: machine address

`Ind` is `offset(base, index)` is calculated `base + index * 8 + offset`.

Thus, `%rax` refers to the contents of the register, while `(%rax)` refers to either the memory address or the contents of the memory address, depending on whether its used as location or value.

x86 assembly is organized into labeled blocks, indicating code locations used by jumps, etc. Program begins execution at designated label (`main`).

Calling Conventions

- Setup Stack Frame: `pushq %rbp  movq %rsp, %rbp`
- Teardown: `popq %rbp`
- Caller Save - freely usable by the called code.
- Callee Save - must be restored by the called code (`rbp, rsp, rbx, r12-15`).
- Arguments: In `rdi, rsi, rdx, rcx, r08, r09` and starting with $n = 7$ in $(n-7)+2)*8+$`rbp`
- Return value in `rax`.
- 128 byte "red zone" - scratch pad for the callee (beyond `rsp`), this means a function can use up to 128 byte without allocationg a stack frame..

Intermediate Representations

Direct translation is bad as it is hard to optimize the resulting assembly code. The representation is too concrete, as it already committed to using certain registers etc. Further retargeting the compiler to a new architecture is hard. Finally control-flow is not structured, arbitrary jumps from one code block to another. Implicit fall-through makes sequences non-modular.

Using a universal IR means that for p programming languages and q ISA's, we only need $p+q$ compilers instead of $p*q$.

IR's allow machine independent code generation and optimization.

Multiple IR's: get program closer to machine code without losing the information needed to do analysis and optimizations (high / mid / low level IR).

Good IR: Easy translation target, easy to translate, narrow interface (fewer constructs means simpler phases / optimizations).

Basic Blocks are a sequence of instructions that are always executed from the first to last instruction. They start with a label and end with a control-flow instruction (no other control-low instruction or label).

Basic blocks can be arranged into a control-flow graph (CFG): Nodes are basic blocks - directed edges represent potential jumps.

LLVM (Low Level Virtual Machine)

Storage Types: local variable `%uid`, global variable `@gid`, abstract locations (stack-allocated with `alloca`), heap-allocated structures (`malloc`).

Each `%uid` appears on the left-hand side of an assignment only once in the entire control flow graph (SSA).

The entry block of the CFG does not have to be labeled, the last instruction of a block is called the terminator.

Example Program:

```
@s = global i32 42

declare void @use (i64)

define i64 @foo(i64 %a, i64* %b) {
   %sum = add nsw i64 %a, 42
   %cond = icmp sgt i64 %sum, 100
   br i1 %cond , label %then , label %else

then:
   call void @use(i64 %sum)
   ret i64 %sum

else:
   store i64 %sum, i64* %b
   ret i64 %sum
}
```

GEP

LLVM supports structured data with the use of `types`, e.g.:

```
%struct.Node = type {i64, %struct.Node*}
```

To computer pointer values of structs or index into arrays, LLVM provides the `getelementptr` instruction. Given a pointer and a path through the structured data pointed to by that pointer, GEP computes an address - analog of LEA.

```
getelementptr <ty>* <ptrval> {, <ty> <idx>}*
```

GEP never dereferences the address it is calculating.

Lexing

Lexing is the process of taking the source code as an input and producing a token stream as output. The problem is to precisely define tokens and matching tokens simultaneously.

One way of implementing a lexer is, using regular expressions. Regex rules precisely describe a sets of strings. But regex alone can be ambiguous if we have multiple matching rules. Most languages therefore choose the longest match or have another specified order.

Regex can be implemented by forming an NFA and then transforming it to a DFA.

Parsing

In this part we take the token stream and generate an abstract syntax tree (AST). Parsing itself does not check things such as variable scoping, type agreement etc.

Parsing uses a more powerful tool than regex - context free grammars (CFG).

Chomsky Hierarchy:

- Regular - Productions have at most one nonterminal and it is at the start or end of the word
- Context-Free (CFG) - LHS of productions only have a single nonterminal
- Context-Sensitive
- Recursively Enumerable

An example for a non CFG would be $a^n b^m c^n d^m$. This corresponds to methods having matching parameters.

A CFG consists of a set of terminals, a set of nonterminals, a start symbol and a set of productions. A production consists of a single nonterminal LHS and an arbitrary RHS.

Derivation Orders - Productions can be applied in any order, however they will all lead to the same parse tree. There are two standard orders:

- Leftmost derivation: Find the left-most nonterminal and apply a production to it
- Rightmost derivation: Find the right-most nonterminal and apply a production there

A grammar is **ambiguous** if there are multiple derivation trees for the same word. This can be a problem for associative operators.

In CFGs ambiguity can (often) be removed by adding nonterminals and allowing recursion only on one side. For example:

```
S -> S + S | S * S | (S) | n
```

Becomes:

```
S_0 -> S_0 + S_1 | S_1
S_1 -> S_2 * S_1
S_2 -> n | (S_0)
```

LL Grammars and Top-Down Parsing

When parsing a grammar **top-down**, we can encounter the problem of multiple productions being possible.

LL(1) means **L**eft-to-right scanning, **L**eft-most derivation, **1** lookahead symbol.

Left-factoring a grammar can make it LL(1): If there is a common prefix we can add a new non-terminal at the decision point. We also need to eliminate left-recursion:

```
S -> S a_1 | ... | S a_n | b_1 | ... | b_m
```

Becomes:

```
S -> b_1 S‘ | ... | b_m S‘
S‘ -> a_1 S‘ | ... | a_n S‘ | epsilon$
```

To actually use these grammars, we need to translate them into a **parsing table**:

For a given production $A \to \gamma$:

- Construct the **first set** of A, this set contains all terminals that begin strings derivable from the nonterminal. For each nonterminal of the first set, add the corresponding production to the table.
- Construct the **follow set** of A, this set contains all terminals that can appear immediately to the right of the given nonterminal. If ϵ is derivable by the production, add the corresponding production to the table.

First(T) = First(S)
First(S) = First(E)
First(S′) = { +, ε }
First(E) = { number, ‘(‘ }

Follow(S′) = Follow(S)
Follow(S) = { $, ‘)’ } ∪ Follow(S′)

Note: we want the *least* solution to this system of set equations… a *fixpoint* computation. More on these later in the course.

T ⟼ S$
S ⟼ ES′
S′ ⟼ ε
S′ ⟼ + S
E ⟼ number | (S)

	number	+	(	)	$ (EOF)
T	⟼ S$		⟼S$		
S	⟼ E S′		⟼E S′		
S′		⟼ + S		⟼ ε	⟼ ε
E	⟼ number		⟼ (S)		

This can be extended to LL(k) grammars by generating a bigger table.

LR Grammars and Bottom-Up Parsing

LR grammars are more expressive than LL grammars. They can handle left-recursive and right-recursive grammars. However error reporting is poorer.

Bottom-up parsing is a sequence of **shift** and **reduce** operations:

- Shift: Move look-ahead token to stack.
- Reduce: Replace symbols γ at the top of the stack with nonterminal X such that $X \to \gamma$ is a production. Pop γ, push X.

The parser state is made up of a stack of nonterminals and terminals, as well as the so far unconsumed input.

Action Selection Problem:

- Given a stack σ and a lookahead symbol b, should the parser **shift** b onto the stack (new stack is σb) , or **reduce** a production $X \mapsto \gamma$, assuming that $\sigma = \alpha\gamma$?
- Sometimes the parser can reduce, but should not, sometimes the stack can be reduced in different ways.

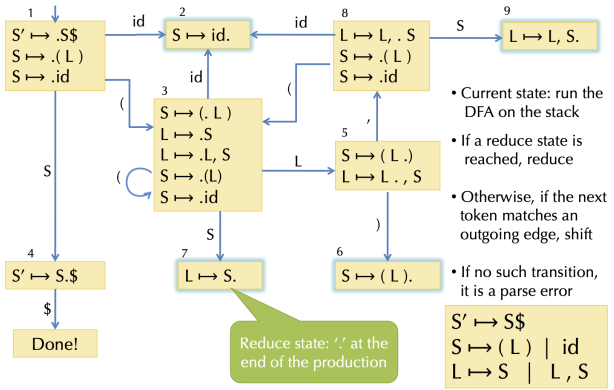
We want to decide based on a prefix α of the stack and the look-ahead.

In LR(0) we have states: items to track progress on possible upcoming reductions. An item is a production with an extra separator ”.” in the RHS.

The idea is that the stuff before the ”.” is already on the stack and the rest is what might be seen next.

Constructing the DFA:

- Add new production: $S' \to S\$$, this is the start of the DFA.
- Add all productions whose LHS occurs in an item in the state just after the dot. Note that these items can cause more items to be added until a fixpoint is reached.
- Add transitions for each possible next (non-)terminal. Shift the dot by one in each of those states.
- Every state that ends in a dot is a reduce state.



The parser then runs the DFA.

Instead of running the DFA from start for each step, we can store the state with each symbol on the stack - representing the DFA as a table of shape `state` × (`terminals` + `nonterminals`).

An LR(0) machine only works if states with reduce actions have a single reduce action else we will encounter shift/reduce or reduce/reduce conflicts (use LR(1) grammar).

In LR(1), each item is an LR(0) item plus a set of look-ahead symbols $A \to \alpha.\beta,\ \mathcal{L}$.

To form the LR(1) closure, we first do the same as for LR(0). Additionally for each item $C \to .\gamma$ we add due to a rule $A \to \beta.C\gamma,\ \mathcal{L}$, we compute its look-ahead set $\mathcal{M}$ including FIRST(γ) and if γ can derive ϵ also $\mathcal{L}$.

S′ ⟼ S$
S ⟼ E + S | E
E ⟼ number | (S)

Start item: S′ ⟼ .S$, {}

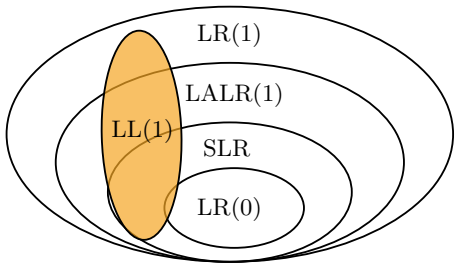
Since S is to the right of a ‘.’, add
S ⟼ .E + S , {$} Note: {$} is FIRST($)
S ⟼ .E , {$}

Need to keep closing, since E appears to the right of a ‘.’ in ‘.E + S’
E ⟼ .number , {+} Note: + added for reason 1
E ⟼ .(S) , {+} FIRST(+ S) = {+}

Because E also appears to the right of ‘.’ in ‘.E’ we get:
E ⟼ .number , {$} Note: $ added for reason 2
E ⟼ .(S) , {$} δ is ε

All items are distinct, so we’re done

For LR(1) we have a shift-reduce conflict if the shifted token is contained in the follow set of the reduction.



FirstClass Functions

Consider the Lambda Calculus. It has variables, functions and function application. The only values are (closed) functions. Instead of `(fun x -> e)` we write: $\lambda x.e$

x{v/x}	= v	*(replace the free x by v)*
y{v/x}	= y	*(assuming y ≠ x)*
(fun x -> exp){v/x}	= (fun x -> exp)	*(x is bound in exp)*
(fun y -> exp){v/x}	= (fun y -> exp{v/x})	*(assuming y ≠ x)*
(e_1 e_2){v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

Function application is interpreted by substitution. In `fun y -> x + y`, `x` is said to be free and `y` is bound by `fun y`.

A term without free variables is **closed**, else it is **open**.
Two terms that differ only by consistent renaming of bound variables are alpha equivalent.

To avoid accidently capturing a free variable by a substitution $e_1\{e_2/x\}$, we first pick an alpha equivalent version of e_1 such that the bound variables do not mention the free variables of e_2.

Some special Lambda Calculus terms:

- Omega Term (infinite loop):
$$(\lambda x.\ x\ x)(\lambda x.\ x\ x)$$
- Y-Combinator (computes fixed point so $Yg = g(Yg)$):
$$\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Operational Semantics is a way to give meaning to a program (interpreter) using inference rules. $exp \Downarrow v$ means exp evaluates to v.

Inference rules are of the form $G; L \vdash e : t$. This means in the global environment G and local environment L the expression e is of type t. Sometimes we include a third symbol on the LHS referring to the return type.

With this we can build up derivation or proof trees. Leaves of the tree are axioms.

Typing

Applying a set of inference rules allows for type checking of a program. For simply typed lambda calculus this implies termination, for well-typed expressions we are a bit more general.

A well-typed program either terminates in a well-defined way, or it continues computing forever.

If we view types as sets of values, there is a natural inclusion relation `Pos` $\subseteq$ `Int`. This gives rise to a subtype relation `P` $<:$ `Int` and to a subtyping hierarchy.

The LUB (least upper bound) is defined for two types $T_1 \vee T_2$.

A subtyping rule is sound if it approximates the underlying subset relation, i.e. if $T_1 <: T_2$ implies $[[T_1]] \subseteq [[T_2]]$. It follows that $[[T_1]] \cup [[T_2]] \subseteq [[LUB(T_1, T_2)]]$.

Argument type is contravariant (it is okay if a function takes more arguments), output type is covariant (it is okay if a function returns less arguments).

$$\frac{S_1 <: T_1 \qquad T_2 <: S_2}{(T_1 -> T_2) <: (S_1 -> S_2)}$$

For records, we have to decide between **width** and **depth subtyping**. In width subtyping, a record is a subtype of another record if it has more (or the same number of) fields (order matters!). In depth subtyping a record is a subtype of another if every elements type is a subtype of the others.

Mutable structure need to be invariant - else one can break typesafety. Thus, T ref $<: S$ ref $\implies T = S$.

OAT Type System

Primitive (non-reference) types: `int, bool`

Definitely non-null reference types: `R` (named) mutable structs with width subtyping, `strings`, `arrays`

Possibly-null reference types: `R?`

Subtyping: `R` $<:$ `R?`

Compiling Objects

The dispatch problem occurs when the same interface is implemented by multiple classes. In the client program, it may be necessary to dynamically choose witch implementation to use. In order to do this, object contain a pointer to a **dispatch vector** (vtable) with pointer to method code.

For extension / inheritance, the dispatch vector gets extended at the end.

For multiple inheritance there are different approaches:

- Allow multiple DV tables (C++), choose which DV to use based on static types, casting requires runtime operations.
- Use a level of indirection: Map method identifiers to code pointers using a hash table, search up through the class hierarchy.
- Give up separate compilation: Use sparse dispatch vectors or binary decision trees.

Multiple Dispatch Vectors: Objects may have multiple entry points with individual DVs, casts change entry point of a variable

Optimizations

There are different kinds of optimization: Power, Space, Time.

- **Constant Folding**: If operands are statically known, compute value at compile-time. More general algebraic simplification: Use mathematical identities.
- **Constant Propagation**: If x is a constant replace its uses by the constant.
- **Copy Propagation**: For $x = y$ replace uses of x with y
- **Dead Code Elimination**: If side-effect free code can never be observed, safe to eliminate it.
- **Inlining**: Replace a function call with the body of the function (arguments are rewritten to local variables).
- **Code Specialization**: Create Specialized versions of a function that is called form different places with different arguments.
- **Common Subexpression Elimination**: It is the opposite of inlining, fold redundant computations together.
- **Loop Optimizations**
  - Hot spots often occur in loops (esp. inner loops)
  - Loop Invariant Code Motion (hoist outside)
  - Strength Reduction (replace expensive ops by cheap ones by creating a dependent induction variable)
  - Loop Unrolling

Dataflow Analysis

Almost every dataflow analysis is a variation of the following algorithm.

Forward Must Dataflow Analysis

```
for all n, in[n] = T, out[n] = T
repeat until no change in 'in' or 'out'
  for all n
    in[n] = intersect out[n'] for all n' in pred[n]
    out[n] = gen[n] union (in[n] \ kill[n])
```

Backward: swap `in` and `out` and `pred` with `succ`.

May: swap $\top$ with $\bot$ or $\emptyset$ and replace $\cup$ with $\cap$.

For each dataflow analysis we only need to define the set `gen`, `kill` as well as the domain of dataflow values $\mathcal{L}$ and a combining operator $\cup$ or $\cap$.

Liveness (Backward, May)

We can use the same registers for multiple `%uids` if they are not alive at the same time. We define `gen[s]` as all the variables used and `kill[s]` as all the variables defined by statement s. $\mathcal{L}$ corresponds to the variables and the combination operator to the set union.

It holds: in$[n] \supseteq$ gen$[n]$, in$[n] \supseteq$ out$[n] \setminus$ kill$[n]$ and out$[n] \supseteq$ in$[n']$ if $n' \in$ succ$[n]$.

Reaching Definition (Forward, May)

What variable definitions reach a particular use of a variable? Used for constant and copy propagation. `in / out` is the set of nodes defining some variable such that the definition may reach the beginning resp. end of the current node. `gen` is the current statement, if it defines a variable, and `kill` is all the other definitions of the defined variable.

Available Expressions (Forward, Must)

Used for common subexpression elimination. `in / out` are the set of nodes whose values are available on entry / exit of the current node. `gen` is the current statement and `kill` is the set of all expressions that use the newly modified variable.

Very Busy (Backward, Must)

An expression is very busy at location p, if every path from p must evaluate the expression before any variable is redefined. It is used for hoisting expressions.

Dominators (Forward, Must)

Define `dom[n]` as the set of all nodes that dominate `n`, i.e. `dom[n] = out[n]`, `gen` is the singelton set of the node itself, `kill` is the empty set.

The iterative solution computes the ideal meet-over-path solution if the flow function distributes over $\cap$. Most of the problems that express properties on how the program computes are distributive and compute the MOP solution, analyses of what the program computes do not. Our analyses also always terminate, as the flow function (`out[n] = ...`) is monotonic.

Register Allocation

Linear-Scan Register Allocation

Compute liveness information and then scan through the program, for each instruction try to find an available register, else spill it on the stack.

Graph Coloring

Compute liveness information for each temp, create an inference graph (nodes are temps and there is an edge if they are alive at the same time), try to color the graph.

Kempe's Algorithm:

- Find a node with degree $< k$ and cut it out of the graph
- Recursively k-color the remaining subgraph
- When remaining graph is colored, there must be at least one free color available for the deleted node.
- If the graph cannot be colored we spill a node and try again.

This can be improve by adding `move` related edges (temps used in a move should have the same color). More aggresively, we may coalesce two move-related nodes into one. This may increase the degree of a node, so we need to be careful.

Brigg's strategy is to only coalesce if the resulting node has fewer than k neighbors with degree $\geq k$.

George's strategs is to only coalesce if for every neighbor t of one of the coalescing nodes x, t also interferes with the other coalescing node or t has degree $< k$.

Precolored Nodes: Certain variables must be pre-assigned to registers (`call`, `imul`, caller-save registers)

Dominator Trees

We want to identify loops in a CFG. For that we use domination. A dominates B (A dom B), if the only way to reach B from start node is via A. This relation is transitive, reflexive and anti-symmetric. This can be computed as forward must dataflow analysis.

The Hasse diagram of the dominates relation is called the dominator tree.

A loop is a set of nodes in the CFG, with a distinguished entry, the header, and exit nodes. It is a strongly connected component (SSC), every node is reachable from every other node and vice versa.

A loop contains at least 1 back edge (back edge = target dominates the source).

The **dominance frontier** of a node A is the set of all CFG nodes γ such that A dominates a predecessor of γ, but does not strictly dominate γ. Intuitively: starting at A, there is a path to γ, but there is another route that does not go through A.

Single Static Assignment (SSA)

Each LLVM IR `%uid` can be assigned only once. When coming from an `if-else` branch or similar, we might not know which `%uid` to take. That's where we introduce ϕ-nodes.

A ϕ-node picks the version of a variable depending on the label from which the ϕ-node was entered. It even allows usage of later-defined `%uid`s.

```
%uid = phi <type> v1, <label1>, ..., vn, <labeln>
```

Converting to SSA:

- Start with CFG with `alloca`s, identify promotable `alloca`s
- Compute dominator tree information
- Calculate `def` / `use` information for each variable
- Insert ϕ-nodes at necessary join points
- Replace `load` / `stores` with freshly generated `%uid`s
- Eliminate unneeded `alloca`s

Some `alloca`s are needed, either if the address of the variable is taken or the address escapes by being passed to a function. If neither condition holds, it is promotable.

Necessary join points are defined as the transitive closure of the dominance frontier of all nodes where a variable x is defined or modified. Then we just need to pick the value of x depending on the predecessors of the node where we just inserted the ϕ-node.

To place ϕ-nodes without breaking SSA, we insert `load`s at the end of each block, and insert `store`s after ϕ-nodes. We can then optimize load after stores (LAS) by substituting all uses of the load by the value stored and remove the load itself. Then, we can eliminate dead `stores` and dead `alloca`s. At the very end, we can eliminate ϕ-nodes with only a single value, or identical values from each predecessor.

Garbage Collection

An object `x` is reachable iff a register contains a pointer to `x` or another reachable object `y` contains a pointer to `x` (we also consider the stack as a source for pointers!). If an object is not reachable, we might want to consider it as garbage.

Reachable objects can be found by starting from registers and following all pointers.

Mark and Sweep

When memory runs out, GC executes two phases: mark phase: trace reachable objects; sweep phase: collects garbage objects (extra bit reserved for memory management)

One problem is that it only runs when we are out of memory - yet we need to keep track of our todo-list (not yet checked pointers). The solution to this is pointer reversal. We keep only a backward and forward pointer. The forward pointer points to the next object to be examined and the backward pointer to the one we just handled. Whenever we follow a pointer, we update that pointer to point to the back pointer, the back pointer points to the current object we followed to it.

Once we checked every object, we go in reverse. We let the backpointed-object point to where the forward pointer is while following its pointer with the back pointer and update the forward pointer to the current object.

Pros: Objects stay in place, no need to update pointers. **Cons**: Fragmentation.

Stop and Copy

Memory is organized into two areas: Old space (used for allocation), new space (use as a reserve for GC).

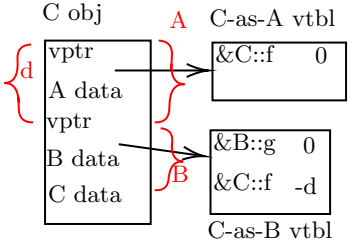
When old space is full all reachable objects are moved to the new space and the roles of the spaces are swapped. To avoid copying twice, a already copied object is replaced by a forwarding pointer to the new copy.

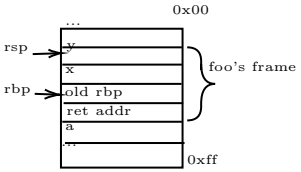
To achieve this without extra space, we divide the new space in three regions: copied and scanned, a scan pointer followed by the copied region, and the alloc pointer followed by the empty region. We copy the objects pointed to by roots, then, as long as scan hasn't caught up to alloc: Find each object pointed to by the object at scan, if it's a forwarding pointer update our current pointer, if not, copy the pointed-to object to the new space, update alloc pointer, and update our current pointer. Increment the scan pointer.

Despite having to update many pointers, stop and copy is generally the fastest GC technique, as allocation and collection is relatively cheap when there's lots of garbage. However, we need to tell objects and its contents apart, which is not possible in languages like C/C++.

Reference Counting

Store number of references in the object itself, assignments modify that number. If the reference count is zero, free the object. Cannot collect circular structures and updating the reference count on each assignment is slow.




0x00
...
rsp
y
x
rbp
old rbp
ret addr
a
...
0xff
foo's frame