

Compiler Design

by dcamenisch

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

- Lexical Analysis: Source Code \rightarrow Token Stream
- Parsing: Token Stream \rightarrow AST
- Intermediate Code Generation: AST \rightarrow Intermediate Code
- Code Generation: Intermediate Code \rightarrow 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 0xffffffff, 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.

Register: **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 = 0F
l (strictly less)	SF \neq 0F
ge (greater or equal)	SF = 0F
le (less or equal)	SF \neq 0F 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 $(n - 7) * 8 + \text{rbp}$
- 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 allocating 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 compute 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, we want + to be left associative, * right associative and * has the higher precedence:

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

Becomes:

```
S_0 -> S_0 + S_1 | S_1
S_1 -> S_2 * S_1 | S_2
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' | $epsilon$
```


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).

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

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 type-safety. Thus, $T \text{ ref } <: S \text{ ref} \implies T = S$.

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: $R <: R?$

Note that `string[]` is not a subtype of `string?[]` but `string[] <: string[]?` holds.

Compiling Objects

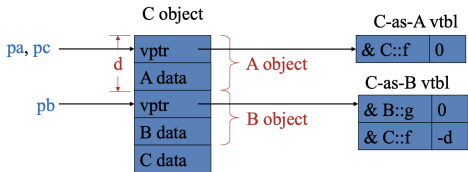
The dispatch problem occurs when the same interface is implemented by multiple classes. In the client program, it may be necessary to dynamically choose witch implementation to use. In order to do this, object contain a pointer to a **dispatch vector** (vtable) with pointer to method code.

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 \perp or \emptyset and replace `intersect` with `union`.

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 (`live[n]` = uids used before end/reassign). We define `gen[s]` as all the variables used (RHS) and `kill[s]` as all the variables defined by statement s (LHS). \mathcal{L} corresponds to the variables and the combination operator to the set union.

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

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. **in/out/gen allow for expressions ($x+y$, ...)**

`gen[B]` = {expr; expr a op b is evaluated in B, neither a nor b are subsequently redefined in B }

`kill[B]` = {expr; a or b of expr a op b are defined in B and a op b is not subsequently evaluated in B}

Dominators (Forward, Must)

Define `dom[n]` as the set of all nodes that dominate `n`, i.e. `dom[n]` = `out[n]`, `gen` is the singleton 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 (e.g. constant propagation). Our analyses also always terminate, as the flow function (`out[n] = ...`) is monotonic.

Soundness is defined as an under approximation of the set of variables.

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 aggressively, 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 ($A \text{ dom } B$), 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. A strictly dominate B , if $A \neq B$ and $A \text{ dom } B$.

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. For a back edge $n \rightarrow h$, h is the header. Loops with the same header can be merged (i.e. $\{A, B, C\}$, $\{A, D\}$ can be merged to $\{A, B, C, D\}$).

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 . It is the set of nodes where A 's dominance stops.

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 %uids.

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

- Converting to SSA:
- Start with CFG with allocas, identify promotable allocas
 - Compute dominator tree information
 - Calculate def / use information for each variable
 - Insert ϕ-nodes at necessary join points
 - Replace load / stores with freshly generated %uids
 - Eliminate unneeded allocas

Some allocas 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 loads at the end of each block, and insert stores 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 allocas. 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**. Pointer reversal enables a depth first traversal of the reachable objects without using additional memory. 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. Stop and Copy moves objects around. Pointers to those objects must also be updated. However, objects (structs) in C and C++ do not carry any metadata that identify which members are pointers and therefore it is impossible to properly update them.

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.

Exercises

- Ex. In parsing, ...
- ☐ LL(1) parsers are more powerful than LR(0) parsers.
 - ☒ LR(1) parsers are more powerful than LL(1) parsers.
 - ☒ LALR(1) may introduce new reduce/reduce conflicts compared to LR(1) when parsing the same grammar.
 - ☐ LALR(1) may introduce new shift/reduce conflicts compared to LR(1) when parsing the same grammar.

- Ex. For context free grammars, ...
- ☒ LR parsers can handle left-recursive grammars.
 - ☒ LR parsers can handle right-recursive grammars.
 - ☐ LL parsers can handle left-recursive grammars.

- Ex. A left-recursive grammar cannot be implemented by an LL(k) parser for any k.
- ☒ True
 - ☐ False

- Ex. LR(k) grammars cannot be right recursive.
- ☐ True
 - ☒ False

- Ex. There is no such thing as a shift/shift conflict for a LR parser.
- ☒ True
 - ☐ False

- Ex. Calling conventions, ...
- ☒ specify where arguments and return values should be stored.
 - ☐ specify the starting address of stack and heap.
 - ☒ can be disregarded by the compiler for functions that are not exposed to external callers as an optimization.

- Ex. Nominal subtyping, ...
- ☒ requires us to explicitly declare subtyping relationships.
 - ☐ is used by OAT for struct subtyping.
 - ☐ is a subcategory of structural subtyping.
 - ☒ is used by Java for subtyping.

- Ex. A basic block, ...
- ☒ starts with a label.

- ☐ can contain more than one control-flow instruction.
- ☒ is always executed starting from the basic block’s first instruction.

Ex. Consider the following dominance frontier DF of a graph: $DF[U] = \{Y\}$, $DF[V] = \{Z, W, Y\}$, $DF[W] = \{V, Y\}$, $DF[X] = \{\}$, $DF[Y] = \{V\}$, $DF[Z] = \{\}$. There is a variable x that is modified at nodes $N = \{X, Y, Z\}$. Determine all nodes where ϕ functions for x have to be inserted, i.e., the join points for N with respect to x. Determine the least fixed point of the sequence: $J[N] = DF_k[N]$ where $DF_0[N] = DF[N]$; $DF_{i+1}[N] = DF[DF_i[N] \cup \{N\}]$

$$\begin{aligned} DF_0[\{X, Y, Z\}] &= \{V\}, \\ DF_1[\{X, Y, Z, V\}] &= \{V, W, Z, Y\}, \\ DF_1[\{V, W, Z, Y\}] &= \{V, W, Z, Y\} = J[\{X, Y, Z\}] \end{aligned}$$