CHAPTER 3

Register allocation

In chapter 1 we simplified the generation of x86 assembly by placing all variables on the stack. We can improve the performance of the generated code considerably if we instead try to place as many variables as possible into registers. The CPU can access a register in a single cycle, whereas accessing the stack can take from several cycles (to go to cache) to hundreds of cycles (to go to main memory). Figure 1 shows a program fragment that we'll use as a running example. The program is almost in x86 assembly but not quite; it still contains variables instead of stack locations or registers.

The goal of register allocation is to fit as many variables into registers as possible. It is often the case that we have more variables than registers, so we can't naively map each variable to a register. Fortunately, it is also common for different variables to be needed during different periods of time, and in such cases the variables can be mapped to the same register. Consider variables y and z in Figure 1. After the variable z is used in `addl z, x` it is no longer needed. Variable y, on the other hand, is only used after this point, so z and y could share the same register.

what is a live variable at a location ℓ?

```
movl $4, z
movl $0, w
movl $1, z
movl w, x
addl z, x
movl w, y
addl x, y
movl y, w
addl x, w
```

$\{\}$

$\{\}$

$\{w\}$

$\{w, z\}$

$\{w, z, x\}$

$\{w, x\}$

I

$\{x, y\}$

$\{y, x\}$

$\{x, w\}$

$L_{in} = read(I) \cup (L_{out} - write(I))$

L_{out}

At ℓ, variable x is live iff it assigned before ℓ and is read/used after ℓ (without some intervening assignment after ℓ before that read

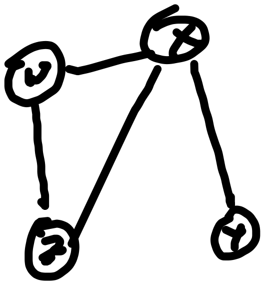


FIGURE 1. An example program in pseudo assembly code. The program still uses variables instead of registers and stack locations.

Liveness is a backwards (dataflow) analysis

3.1. Liveness analysis

A variable whose current value is needed later on in the program is called *live*.

DEFINITION 3.1. A variable is ***live*** if the variable is used at some later point in the program and there is not an intervening assignment to the variable.

To understand the latter condition, consider variable `z` in Figure 1. It is not live immediately after the instruction `movl $4, z` because the later uses of `z` get their value instead from the instruction `movl $1, z`. The variable `z` is live between `z = 1` and its use in `addl z, x`. We have annotated the program with the set of variables that are live between each instruction.

The live variables can be computed by traversing the instruction sequence back to front (i.e., backwards in execution order). Let $I_1, \ldots, I_n$ be the instruction sequence. We write $L_{\mathsf{after}}(k)$ for the set of live variables after instruction I_k and $L_{\mathsf{before}}(k)$ for the set of live variables before instruction I_k. The live variables after an instruction is always equal to the live variables before the next instruction.

$$L_{\mathsf{after}}(k) = L_{\mathsf{before}}(k+1)$$

To start things off, there are no live variables after the last instruction, so we have

$$L_{\mathsf{after}}(n) = \emptyset$$

We then apply the following rule repeatedly, traversing the instruction sequence back to front.

$$L_{\mathtt{before}}(k) = (L_{\mathtt{after}}(k) - W(k)) \cup R(k),$$

where $W(k)$ are the variables written to by instruction I_k and $R(k)$ are the variables read by instruction I_k. Figure 2 shows the results of live variables analysis for the example program from Figure 1.

Implementing the live variable analysis in Python is straightforward thanks to the built-in support for sets. You can construct a set by first creating a list and then passing it to the `set` function. The following creates an empty set:

```
>>> set([])
set([])
```

You can take the union of two sets with the `|` operator:

```
>>> set([1,2,3]) | set([3,4,5])
set([1, 2, 3, 4, 5])
```

To take the difference of two sets, use the `-` operator:

```
movl $4, z
              {}
movl $0, w
              {w}
movl $1, z
              {w, z}
movl w, x
              {x, w, z}
addl z, x
              {x, w}
movl w, y
              {x, y}
addl x, y
              {x, y}
movl y, w
              {w, x}
addl x, w
              {}
```

FIGURE 2. The example program annotated with the set of live variables between each instruction.

```
>>> set([1,2,3]) - set([3,4,5])
set([1, 2])
```

Also, just like lists, you can use Python's `for` loop to iterate through the elements of a set:

```
>>> for x in set([1,2,2,3]):
...   print x
1
2
3
```

3.2. Building the interference graph

Based on the liveness analysis, we know the program regions where each variable is needed. However, during register allocation, we'll need to answer questions of the specific form: are variables u and v ever live at the same time? (And therefore can't be assigned to the same register.) To make this question easier to answer, we create an explicit data structure, an *interference graph*. An interference graph is an undirected graph that has an edge between two variables if they are live at the same time, that is, if they interfere with each other.

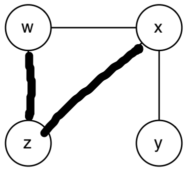


FIGURE 3. Interference graph for the example program.

x,y live at the same time but don't interfere

① they have the same value

The most obvious way to compute the interference graph is to look at the set of live variables between each statement in the program, and add an edge to the graph for every pair of variables in the same set. This approach is less than ideal for two reasons. First, it can be rather expensive because it takes $O(n^2)$ time to look at every pair in a set of n live variables. Second, there's a special case in which two variables that are live at the same time don't actually interfere with each other: when they both contain the same value.

A better way to compute the edges of the intereference graph is given by the following rules.

if $t \in L_{after}(k)$ then

- If instruction I_k is a move: `movl` s, t, then add the edge (t, v) for every $v \in L_{\mathsf{after}}(k)$ unless $v = t$ or $v = s$.
- If instruction I_k is not a move but some other arithmetic instruction such as `addl` s, t, then add the edge (t, v) for every $v \in L_{\mathsf{after}}(k)$ unless $v = t$.
- If instruction I_k is of the form `call` *label*, then add an edge (r, v) for every caller-save register r and every variable $v \in L_{\mathsf{after}}(k)$. (The caller-save registers are `eax`, `ecx`, and `edx`.)

t — v

$v \in L_{after}(k$

Working from the top to bottom of Figure 2, `z` interferes with `w` and `x`, `w` interferes with `x`, and `y` interferes with `x`. In the second to last statement, we see that `w` interferes with `x`, but we already know that. The resulting interference graph is shown in Figure 3.

t — x

In Python, a convenient representation for graphs is to use a dictionary that maps nodes to a set of adjacent nodes. So for the interference graph, the dictionary would map variable names to sets of variable names.

%eax

Consider the first instruction in Figure 2, `movl $4, z`. The L_{after} set for this instruction is the empty set, in particular, z is not live which means that this `movl` is useless, or in compiler lingo, it's dead. It is tempting to remove dead instructions during the construction of the interference graph, but in general, it's better to keep each pass focused on just one job, making it easier to debug and maintain the code. A more principled approach is to insert a pass after liveness analysis that removes dead instructions. Also, removing dead instructions may cause more instructions to become dead, so one could iterate the liveness analysis and dead-code removal passes until there is no more dead code to remove. However, to really do dead-code elimination right, one should combine it with several other optimizations, such as constant propagation, constant folding, and procedure inlining [**22**]. This combination of optimizations is a good choice of topic for your final project.

3.3. Color the interference graph by playing Sudoku

We now come to the main event, mapping variables to registers (or to stack locations in the event that we run out of registers). We need to make sure not to map two variables to the same register if the two variables interfere with each other. In terms of the interference graph, this means we cannot map adjacent nodes to the same register. If we think of registers as colors, the register allocation problem becomes the widely-studied graph coloring problem [**1, 17**].

The reader may actually be more familar with the graph coloring problem then he or she realizes; the popular game of Sudoku is an instance of graph coloring. The following describes how to build a graph out of a Sudoku board.

- There is one node in the graph for each Sudoku square.
- There is an edge between two nodes if the corresponding squares are in the same row or column, or if the squares are in the same 3×3 region.
- Choose nine colors to correspond to the numbers 1 to 9.
- Based on the initial assignment of numbers to squares in the Sudoku board, assign the corresponding colors to the corresponding nodes in the graph.

If you can color the remaining nodes in the graph with the nine colors, then you've also solved the corresponding game of Sudoku.

Given that Sudoku is graph coloring, one can use Sudoku strategies to come up with an algorithm for allocating registers. For example, one of the basic techniques for Sudoku is Pencil Marks. The idea is that you use a process of elimination to determine what numbers still make sense for a square, and write down those numbers in the square (writing very small). At first, each number might be a possibility, but as the board fills up, more and more of the possibilities are crossed off (or erased). For example, if the number 1 is assigned to a square, then by process of elimination, you can cross off the 1

pencil mark from all the squares in the same row, column, and region. Many Sudoku computer games provide automatic support for Pencil Marks. This heuristic also reduces the degree of branching in the search tree.

The Pencil Marks technique corresponds to the notion of color *saturation* due to Brélaz [**3**]. The saturation of a node, in Sudoku terms, is the number of possibilities that have been crossed off using the process of elimination mentioned above. In graph terminology, we have the following definition:

$$\mathrm{saturation}(u) = |\{c \mid \exists v. v \in \mathrm{Adj}(u) \text{ and } \mathrm{color}(v) = c\}|$$

where $\mathrm{Adj}(u)$ is the set of nodes adjacent to u and the notation $|S|$ stands for the size of the set S.

Using the Pencil Marks technique leads to a simple strategy for filling in numbers: if there is a square with only one possible number left, then write down that number! But what if there aren't any squares with only one possibility left? One brute-force approach is to just make a guess. If that guess ultimately leads to a solution, great. If not, backtrack to the guess and make a different guess. Of course, this is horribly time consuming. One standard way to reduce the amount of backtracking is to use the most-constrained-first heuristic. That is, when making a guess, always choose a square with the fewest possibilities left (the node with the highest saturation). The idea is that choosing highly constrained squares earlier rather than later is better because later there may not be any possibilities left.

In some sense, register allocation is easier than Sudoku because we can always cheat and add more numbers by spilling variables to the stack. Also, we'd like to minimize the time needed to color the graph, and backtracking is expensive. Thus, it makes sense to keep the most-constrained-first heuristic but drop the backtracking in favor of greedy search (guess and just keep going). Figure 4 gives the pseudo-code for this simple greedy algorithm for register allocation based on saturation and the most-constrained-first heuristic, which is roughly equivalent to the DSATUR algorithm of Brélaz [**3**] (also known as saturation degree ordering (SDO) [**7, 15**]). Just as in Sudoku, the algorithm represents colors with integers, with the first k colors corresponding to the k registers in a given machine and the rest of the integers corresponding to stack locations.

3.4. Generate spill code

In this pass we need to adjust the program to take into account our decisions regarding the locations of the local variables. Recall

```
Algorithm: DSATUR
Input: the inference graph $G$
Output: an assignment $\text{color}(v)$ for each node $v \in G$

$W \leftarrow vertices(G)$
while $W \neq \emptyset$ do
   pick a node $u$ from $W$ with the highest saturation,
      breaking ties randomly
   find the lowest color $c$ that is not in $\{\text{color}(v) \mid v \in \text{Adj}(v)\}$
   $\text{color}(u) = c$
   $W \leftarrow W - \{u\}$
```

FIGURE 4. Saturation-based greedy graph coloring algorithm.

that x86 assembly only allows one operand per instruction to be a memory access. For instance, suppose we have a move `movl y, x` where `x` and `y` are assigned to different memory locations on the stack. We need to replace this instruction with two instructions, one that moves the contents of `y` into a register and then another instruction that moves the register's contents into `x`. But what register? We could reserve a register for this purpose, and use the same register for every move between two stack locations. However, that would decrease the number of registers available for other uses, sometimes requiring the allocator to spill more variables.

• x is going to be spilled

x ↦ -4(%ebp)
y ↦ -8(%ebp)

Instead, we recommend creating a new temporary variable (not yet assigned to a register or stack location) and rerunning the register allocator on the new program, where `movl y, x` is replaced by

```
movl y, tmp0
movl tmp0, x
```

× movl -4(%ebp), -8(%ebp)

The `tmp0` variable will have a very short live range, so it does not make the overall graph coloring problem harder to solve. However, to prevent `tmp0` from being spilled and then needing yet another temporary, we recommend marking `tmp0` as "unspillable" and changing the graph coloring algorithm with respect to how it picks the next node. Instead of breaking ties randomly between nodes with equal saturation, give preference to nodes marked as unspillable.

If you did not need to introduce any new temporaries, then register allocation is complete. Otherwise, you need to go back and do another iteration of live variable analysis, graph building, graph coloring, and generating spill code. When you start the next iteration, do not start from scratch; keep the spill decisions, that is, which variables are spilled and their assigned stack locations, but redo the allocation for the new temporaries and the variables that were assigned to registers.

3.5. Assign homes and remove trivial moves

Once the register allocation algorithm has settled on a coloring, update the program by replacing variables with their homes: registers or stack locations. In addition, delete trivial moves. That is, wherever you have a move between two variables, such as

```
movl y, x
```

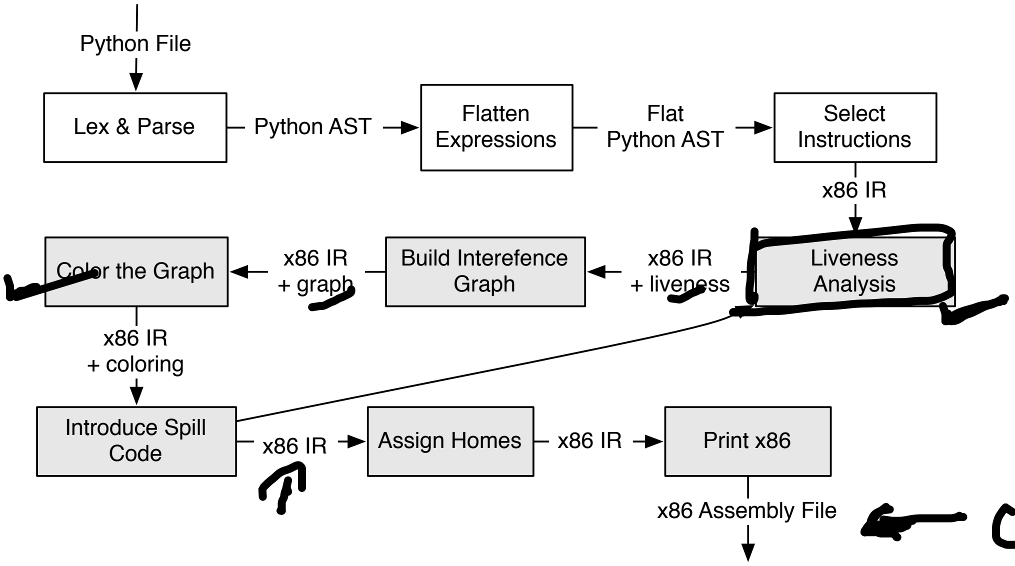
where x and y have been assigned to the same location (register or stack location), delete the move instruction.

EXERCISE 3.1. Update your compiler to perform register allocation. Test your updated compiler on your suite of test cases to make sure you haven't introduced bugs. The suggested organization of your compiler is shown in Figure 7. What is the time complexity of your register allocation algorithm? If it is greater than $O(n \log n)$, find a way to make it $O(n \log n)$, where n is the number of variables in the program.

3.6. Read more about register allocation

The general graph coloring problem is NP-complete [**6**], so finding an optimal coloring (fewest colors) takes exponential time (for example, by using a backtracking algorithm). However, there are many algorithms for finding good colorings and the search for even better algorithms is an ongoing area of research. The most widely used coloring algorithm for register allocation is the classic algorithm of Chaitin [**5**]. Briggs describes several improvements to the classic algorithm [**4**]. Numerous others have also made refinements and proposed alternative algorithms. The interested reader can google "register allocation".

More recently, researchers have noticed that the interference graphs that arise in compilers using static single-assignment form have a special property, they are chordal graphs. This property allows a simple algorithm to find optimal colorings [**8**]. Furthermore, even if the compiler does not use static single-assignment form, many interference graphs are either chordal or nearly chordal [**16**].



TDD

COG
HW1-3
(30)

FIGURE 5. Suggested organization of the compiler.

The chordal graph coloring algorithm consists of putting two standard algorithms together. The first algorithm orders the nodes so that that the next node in the sequence is always the node that is adjacent to the most nodes further back in the sequence. This algorithm is called the maximum cardinality search algorithm (MCS) [**18**]. The second algorithm is the greedy coloring algorithm, which simply goes through the sequence of nodes produced by MCS and assigns a color to each node. The ordering produced by the MCS is similar to the most-constrained-first heuristic: if you've already colored many of the neighbors of a node, then that node likely does not have many possibilities left. The saturation based algorithm presented in Section 3.3 takes this idea a bit further, basing the choice of the next vertex on how many colors have been ruled out for each vertex.