

Assignment 12: register allocation

ECEN 4553 & 5013, CSCI 4555 & 5525

Prof. Jeremy G. Siek

December 12, 2007

This week we allocate some local variables to live in the general purpose registers of the x86 and the rest of the local variables to live in locations on the stack. Figure 1 shows the new passes that we'll be adding to the compiler. The register allocator is made up of four passes that iterate until all local variables have been assigned a home. The pass "Introduce Spill Code" may introduce new local variables, which is why we must iterate the register allocation process. Note that the removal of structure control statements (ifs and whiles) has been moved to after the register allocation passes. The reason for this is that the register allocator needs to understand the control flow and it is easier to understand structured control flow than to analyze unstructured control flow. The later requires building a control flow graph, which is not altogether difficult but is not necessary in our setting. Make sure that your if and while statements do not have complex operators/operands in their condition expression before you get to the register allocator, i.e., remove them in "Remove Complex Opera*".

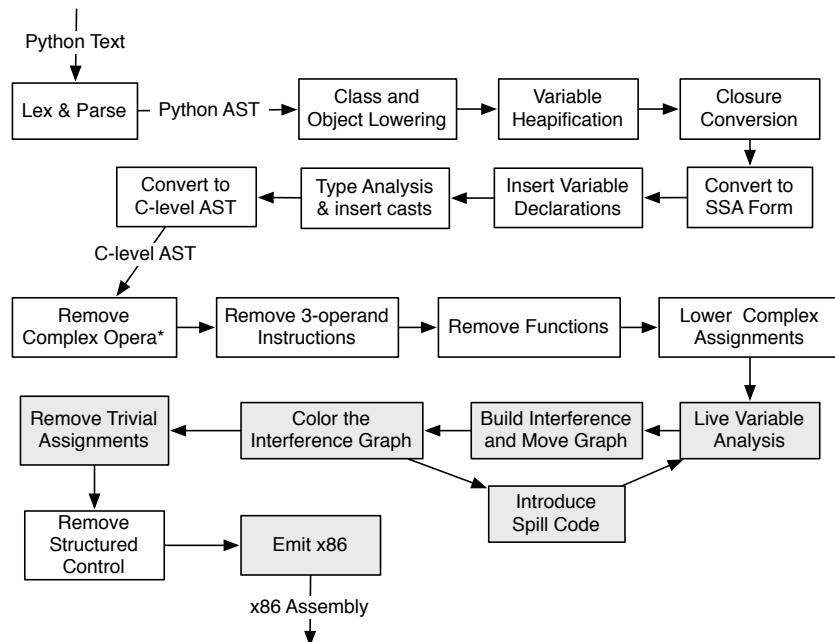


Figure 1: Structure of the compiler.

1 Introduction to Register Allocation

```
int x, z, y, w;

z = 4;    // live: {}
w = 0;    // live: {}
z = 1;    // live: {w}
x = w + z; // live: {w,z}
y = x + 1; // live: {x}
w = y;    // live: {y}
```

The job of register allocation is to map the possibly very many local variables of a function to the few general purpose machine registers. Of course, it is not always possible to fit all the variables into registers, but often times we do not need all of the variables all of the time, so we can have many variables take turns using a register. Consider the program fragment on the left. After the variable z is used in $x = w + z$ 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. A variable whose current

value is needed later on in the program is called *live*. It is impossible to determine whether every variable is live or not in every program. You would have to be able to predict what the program is going to do, which is undecidable. Instead, we make a conservative approximation based on a flow insensitive analysis and compute whether a variable *may be* live.

Definition 1.1. 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 later condition, consider variable z in the program above. It is not live immediately after the assignment $z = 4$ because the later uses of z get their value instead from the assignment $z = 1$. The variable z is live between this later assignment and its use in $x = w + z$. We have annotated each assignment statement with the set of live variables when the right-hand side is being evaluated (just before the actual assignment is executed).

We say that two variables interfere with one another if they are both live at the same time. In particular, a variable u *interferes* with v if v is in the live set when u is assigned to. In the above, x , z , and w interfere with one another and x and y interfere, but y and w do not interfere and neither do y and z . This information is recorded in an undirected graph, called the *interference graph*, as shown in Figure 2.

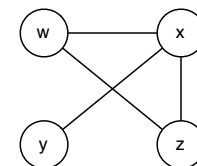


Figure 2: Interference graph

In this program, we can choose to assign y and w to the same register or we can choose to assign y and z to the same register. Which option is better? If we assign y and w to the same register we can eliminate the assignment $w = y$, so that is the better choice. To bias the register allocation in this way, we need to keep track of which variables are related by an assignment. The *move graph* connects two variables u and v if there is an assignment statement from u to v or vice versa. Figure 3 shows the move graph for the above program.

In the next sections we look in more detail at the new compilation passes for this week.

2 Liveness Analysis

Liveness analysis is a data-flow analysis, and as such is similar to the type analysis pass that we've already implemented. This time, instead of computing types for variables we are computing whether they might be live or not. In general, data-flow analyses can be categorized as either

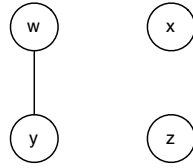


Figure 3: Move graph

forward or backward depending on how the information flows through the program. The type analysis was a forward analysis, whereas the liveness analysis is backwards. The way that we compute the set of live variables is to start at the end of the program and work backwards. When we see uses of variables, we add them to the set of live variables. Let's look again at the following program.

```

int x, z, y, w;

z = 4;    // live: {}
w = 0;    // live: {}
z = 1;    // live: {w}
x = w + z; // live: {w,z}
y = x + 1; // live: {x}
w = y;    // live: {y}
  
```

Starting from the back we have a reference to y , so $\{y\}$ is the set of live variables. Moving up, the set of live variables just before the assignment $y = x + 1$ is the same as the set for the subsequent statement, minus the variable assigned to (in this case y), plus the new variables that are referenced on the right hand side (x). So the set live variables at $y = x + 1$ is $\{x\}$.

We annotate assignment statements and function calls with a new attribute, called `lives`, that is the list of live variables at that assignment or function call.

If statements The two branches are analyzed back to front starting with the set of live variables from just after the `if` statement. The set of live variables to use for the statement that precedes the `if` statement is the union of the live variables from the beginning of the “then” and “else” branches plus any variables referenced in the condition of the `if`. The following example shows how to annotate a program with `if` statement with liveness information.

```

x = 1;    // live: {}
y = 3;    // live: {x}
w = 1;    // live: {x,y}
z = 0;    // live: {w,x,y}
if (x)
    w = 4;    // live: {y}
else
    y = w + 2; // live: {w}
    z = y + w; // live: {w,y}
  
```

While statements The body of the `while` loop is analyzed front to back, starting with the set of live variables from just after the `while` statement. The live variables to start for the statement that precedes the `while` loop is the union of the live variables from after the loop with the live variables from the top of the body, plus any variables referred to in the loop condition. The following is an example.

```

x = 1; // live: {}
y = 2; // live: {x}
  
```

```

w = 3; // live: {x,y}
z = 4; // live: {w,x,y}
while (x)
    w = z; // live: {y,z}
    z = y + w; // live: {w,y}

```

3 Build the Interference and Move Graph

Once the liveness analysis is complete we can make another pass over the program to build the interference and move graphs. We generate one interference and move graph per function. We have several cases to consider.

1. $x = c$, an assignment of a constant, or $x += e$ an arithmetic or logical operation. Let $\{v_1, \dots, v_n\}$ be the live variables at this point. Then add edges (x, v_i) to the interference graph for all $i \in \{1, \dots, n\}$.
2. $x = y$ an assignment between two variables. Let $\{v_1, \dots, v_n\}$ be the live variables at this point. Then add edges (x, v_i) to the interference graph for all $i \in \{1, \dots, n\}$ so long as $v_i \neq y$. Add the edge (x, y) to the move graph.
3. `call label` Let $\{v_1, \dots, v_n\}$ be the live variables at this point. The called function may write to the caller-save registers, so they all interfere with the live variables at this point. So we add edges $(\%eax, v_i), (\%ecx, v_i), (\%edx, v_i)$ for all $i \in \{1, \dots, n\}$.

4 Color the Interference Graph

Graph coloring is a recursive algorithm that operates on a list of nodes (variables) to be colored (assigned registers or stack locations). We process each function separately with a call to the graph coloring algorithm. Initially, the list of nodes contains all the local variables for a function. The algorithm returns a dictionary that maps each variable to its home. The algorithm proceeds as follows.

1. If the list of variables is empty, return an empty dictionary.
2. Pick a node u whose degree (number of neighbors) is less than the number of registers. If there is no such node, pick any high degree node u that is spillable. (Initially, all local variables are spillable but when we "Introduce Spill Code" some new locals are marked as unspillable.) When computing the degree, disregard neighbors that are not in the current list of variables.
3. Remove the node u from the list of variables and recursively process the list, getting back a dictionary that provides homes for all the other variables.
4. Select a home for u . The first choice is a register that does not interfere with u and that has not already been assigned to a variable that interferes with u . Prefer registers that are move-related to u or that have been assigned variables that are move related to u . If there is no compatible register, look for a compatible stack location. The stack location must be of the

appropriate size and it must not already be assigned a variable that interferes with u . Again, prefer move-related locations to non-move related locations. If there are no compatible stack locations, create a new one (take the lowest base-pointer offset so far and subtract the size of the variable). Once a home h has been found, return the dictionary with the additional assignment of u to h .

5 Introduce Spill Code

In this pass we need to adjust the code to jive with our decisions regarding the locations of the local variables. Any instruction where more than one operand is variable assigned to a location on the stack. For instance, suppose we have an assignment $x = y$; where x and y are assigned to different locations on the stack. Replace this assignment with $\text{tmp} = y$; $x = \text{tmp}$; where tmp is a new local variable that is marked as unspillable. The type of tmp should be the same as the type of y .

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 spill code introduction. When you start the new iteration, remember the homes for variables that have been spilled and assigned to the stack, but discard the assignments to registers.

6 Remove Trivial Assignments

Wherever you have an assignment between two variables, such as

$$x = y$$

where x and y have been assigned to the same location (register or stack offset), remove the assignment.

Exercise 6.1. Implement the four new register allocation passes and update your “Emit x86” pass to deal with the lower-level AST that is produced by register allocation.