Course Script

INF 5110: Compiler construction

INF5110, spring 2024

Martin Steffen

Contents

10 Code generation **1**
10.1 Introduction . . . 1
10.1.1 General overview and issues for code generation . . . 1
10.1.2 Code generation . . . 8
10.2 2AC and costs of instructions . . . 11
10.2.1 Cost model . . . 13
10.3 Basic blocks and CFGs . . . 17
10.3.1 From 3AC to CFG: “partitioning algorithm” . . . 18
10.3.2 Levels of analysis . . . 21
10.3.3 Loops in control-flow graphs . . . 22
10.3.4 Data flow analysis in general . . . 25
10.4 Liveness analysis (general) and a variation (def/use) . . . 27
10.4.1 Variation on the topic: def/use analysis . . . 31
10.5 Local liveness: dead or alive . . . 33
10.6 Local liveness^{++} . . . 36
10.7 Global analysis . . . 45
10.7.1 Basic blocks and live-in and live-out . . . 49
10.8 Code generation algorithm . . . 53
10.8.1 A simple code generation algorithm . . . 55

Chapter 10 Code generation

What is it about?

Learning Targets of this Chapter

1. 2AC
2. cost model
3. register allocation
4. control-flow graph
5. local liveness analysis (data flow analysis)
6. "global" liveness analysis

Contents

10.1 Introduction 1
10.2 2AC and costs of instructions 11
10.3 Basic blocks and CFGs 17
10.4 Liveness analysis (general) and a variation (def/use) . . . 27
10.5 Local liveness: dead or alive . 33
10.6 Local liveness^{++} 36
10.7 Global analysis 45
10.8 Code generation algorithm . . 53

10.1 Introduction

10.1.1 General overview and issues for code generation

This chapter covers the last step, the "real" code generation. Much of the material is based on the (old) *dragon book* [2]. There is also a newer edition [1]. The book is a classic in compiler construction. The principles and issues of code generation from [2] are still fine. Technically, the code generation is done for two-address machine code, i.e., the code generation will go **from 3AIC to 2AC**, i.e., to an architecture with 2A instruction set, instructions with a 2-address format. For *intermediate* code, the two-address format (which we did not cover), is typically not used. If one does not use a "stack-oriented" virtual machine architecture, 3AIC is more convenient, especially when it comes to analysis (on the intermediate code level).

For **hardware** architectures, 2AC and 3AC have different strengths and weaknesses, it's also a question of the technological state-of-the-art. There are both RISC and CISC-style designs based on 2AC as well as 3AC. Also whether the processor uses 32-bit or 64-bit instructions plays a role: 32-bit instructions may simply be too small to accommodate for 3 addresses. These questions, how to design an instruction set that fits to the current state or generation of chip or processor technology for some specific application domain belongs to the field of *computer architecture*. We assume a instruction set as given, and base the code generation on a 2AC instruction set, following Aho et al. [2]. In the newer edition of the dragon book [1], the corresponding chapter has been "ported" to cover code generation for 3AC. But the core principles and challenges are the same.

Register allocation

One core problem is register allocation, and the general issues discussed in that chapter would not change, if one would do it for a 3A instruction set. Of course, details would change. The register allocation we will do is, on the one hand, actually pretty simple. Simple in the sense that the code generator does not make a huge effort of optimization. One focus will be on code generation of "straight-line intermediate code", i.e. code *inside* one node of a control-flow graph. Those code-blocks are also known as **basic blocks**. Anyway, the register allocation method walks through one basic block, keeping track of which variable and which temporary currently contains which value, resp. keeping track for values, in which variables and/or register they reside. This book-keeping is done via so-called **register descriptors** and **address descriptors**. As said, the allocation is conceptually simple:

It focuses on not very aggressive allocation inside one basic block.

The presentation also ignores the more complex addressing modes discussed in the previous chapter. Still, the details will look, well, already detailed. Those details would, obviously change, if we used a 3AC instruction set, but the notions of address and register descriptors would remain. Also the way, the code is generated, walking through the instructions of the basic block, could remain. The way it's done is "analogous" on a very high level to what had been called **static simulation** in the previous chapter. "Mentally", the code generator goes line by line through the code, and keeps track of where is what (using address and register descriptors). That information is useful to make use of registers, i.e., generating instructions that, when executed, reuse registers, etc.

That also includes making "decisions" which registers to reuse. We don't go much into that one (like asking: if a register is "full", is it profitable to swap out the value?). By swapping, I mean, saving back the value to main memory, and loading another value to the register. If the new value is more "popular" in the future, needed more often or sooner etc., and the previous value maybe less so, then it is a good idea to swap them out, in case all registers are filled already. An extreme case of being used less often and not so soon is to be never used in the future. The corresponding variable is then called dead (at the current point), or rather *not live*, and the corresponding analysis is called liveness analysis.

Optimization (and "super-optimization"), local and global aspects

Focusing on straight-line code, we are dealing with a finite problem, so there is no issue with non-termination and undecidability. One could try therefore to make an "absolutely optimal" translation of the IC, the one with an optimal use of registers. The chapter will discuss some measures how to estimate the quality of the code in the form of a simple

cost model in Section 10.2. One could use that cost model or other, more refined ones, to define what optimal means, and then produce optimal code for that. Optimizations that are ambitious in that way are sometimes called **super-optimization** [4] and compiler phases that do that are super-optimizers. Super-optimization may not only target register usage or cost-models like the one used here, it's a general (but slightly weird) terminology for transforming code into one which genuinely and demonstrably optimal (according to a given criterion). In general, that's of course fundamentally impossible, but for straight-line code it can be done.

The code generation here does *not* do that. Actually, super-optimization is not often attempted. One reason should be clear: it's costly. For long pieces of straight-line code (i.e., big basic blocks) it may take too much time. There is also the effect of decreasing marginal utility. A relatively modest and simple "optimization" may lead to initially drastic improvements, compared to not doing anything at all. However, to get the last 5% of speed-up or improvement pushes up the effort disproportionally.

A related reason is: super-optimization can be achieved anyway only for parts of the code (like straight-line code and basic blocks). One can genealize that and push the boundaries, as long as it remains a finite problem, e.g., allowing branching, but leaving out loops. That will make the problem more complicated and targets larger chunk of code, which drives up the effort, as well, but still remains a finite problem.

But there are limits of what can be done. If we stick to our setting, where we currently generate code *per basic block*, super-optimization may be costly but doable. But it would be **only locally optimal**, per *one block*. Especially when having a code, where local blocks are small, that would have the positive effect that locally super-optimized code may be done without too much effort. But what good would that do, if the *non-local* quality is bad? Focusing optimization effort onto the local blocks and ignoring the global situation may be an unbalanced use of resources. It may be better to do a decent (but not super-optimal) local optimization that achieves already drastic improvements, and *also* invest in a simple global analysis and optimization, to also reap there low-effort but good initial gains.

That's also the route the lecture takes: now we are doing a simple register allocation, without much optimization or strategy to find the best register usage (and we discuss also one global aspect of program, across the boundaries of one elementary block). That global aspect will be global **live variable analysis**. It will come later (in Section 10.7), because first we discuss local live variable analysis which is used for the local code generation. We can remark already here, that live variable analysis can be done locally and globally; the generation just uses live variable information for its task, whether that information is local or global. So the code generation is, in that way, **independent** from whether one invests in local or in global live variable analysis. It's just produces better code, i.e., makes better use of registers, when being based on better information (like using live variable information coming from a global live variable analysis). Indeed, the code generation would produce semantically *correct* code, without *any* live variable analysis! In that way, the analysis and the code generation are separate problems but not independent, as the register allocation in the code generation makes use of the information from live variable analysis.

Concerning the "degree of locality" of the code generation: The algorithm later will work "super-locally", insofar that it generates 2AC and makes decisions on registers line by line: every line of 3AIC is translated onto 2 (or sometimes one) line of 2AC. There is no attempt afterwards to go through the 2AC again, getting some more global perspective and them optimize it further, for instance rearranging the lines, or obtaining a register usage better that the one that had been arranged for by the line-by-line code generation. The code generation steps through the 3AC line-by-line but is not completely local, it does some book-keeping about registers used, i.e., allocated in the past. And, not to forget, the code generator has access to liveness information, which is information about the *future* use of registers. In the previous chapter, the **macro expansion** was really line-by-line local, where 3AIC was translated to 1AIC (i.e., p-code): each 3AIC line was expanded into some lines of p-case in a completely "context-free" manner, focusing on each individual, line independent from where the line is used. That simplistic expansion ignored the *past*, i.e., what happened before, and it ignored the future, i.e., what will happen afterwards. The code generation here takes care for both aspects:

> What has happened in the **past** is kept tracked by the **register and address descriptors.** Aspects of the **future** are taken care of by the **liveness** analysis.

Depending on whether one does as block-local liveness analysis or a global analysis just changes how "far into the future" the analysis looks. As far as the past in concerned: that one is (in our presentation) just block-local. The book-keeping with the register and address descriptor starts fresh with each block, there is here *no* memory of what potentially had happened in some earlier block.

Live variable analysis

Now, what is live variable analysis anyway, if we mention it here already, and what role does it play here?

> A variable is live (at some point in a program) if it "will be used" (= read) in the future.

One could dually also say, a non-live variable at a given point is dead, but it's still live variable analysis since "death analysis" would not sound appealing.... It's important information, especially for register allocation: if it so happens that the value of a variable is stored in a register and if one figures additionally out, that the variable is dead (i.e., not used in the future), the register may be used otherwise. We elaborate on that further below, in first approximation we can think that the register is simply "free" and can just be used when needed otherwise.

Now, the above characterization about liveness is a bit *imprecise*, and we wrote "will be used" in quotation marks. What's the problem? The problem is that the future may be unknown, and in general it's impossible to know the exact future, for different reasons. One is, that in general, undecidability may prevent the the future behavior to be known. There can be actually another reason, namely if one analyzes not a global program but only a fragment (maybe one basic block, one loop body, one procedure body). The program fragment being analyzed is **"open"** insofar its behavior may depend on data coming from outside. In particular, the program fragment's behavior depends on that outside data or "input", when conditionals or conditional jumps are used. Even if the possible input is finite, it may influence the behavior. One behavior where, at a given point, a variable *will* be used, and another behavior, where that variable will *not* be used: In one future behavior, the variable is live, in the other future, it is dead. Without knowing the input or context, one cannot say that the variable "will" be used or not, it simply depends.

Coming back to the "definition" of liveness. The long discussion hopefully clarified, that in a general setting, when analyzing a (piece of) program it cannot be about whether a variable *will* be used. Liveness of a variable is used here to see whether a register that contains (the value of) variable can be considered free. That leads to the following interpretation:

If there **exists a possible future** where the variable stored in a register **may** be used, then the code generator cannot risk reusing the register and the variable is considered (statically) live.

That means, the notion of (static) liveness is a question of a condition that "may-in-the-future" apply. There are other interesting conditions of that sort. Some would be characterized by "must" instead of "may". And some refer to the past, not the future. That would lead to the area of **data-flow analysis** (or more ambitiously, abstract interpretation). We won't go deep there, we stick to live-variable analysis (for the purpose of code generation). However, if one understands live variable analysis, especially the *global* live variable analysis covered later, one has understood core principles of many other flavors of data flow analysis (may or must, forward or backward).

Talking about conditions applying to the "past", perhaps we should defuse a possible misconception. Liveness of a variable refers to the future, and we said, there are situations when the future is unknown. So one may come to believe that analyzing the past would not face the same problems. When running a (closed) program that may be true: we cannot know the future, but we can record the past ("logging"), so the past is known. But here we are still inside the compiler, doing *static* analysis and we may deal with open program fragments. For concreteness' sake, let's use some particular question for illustration: nil-pointer analysis. That refers to figuring out whether a variable is properly initialized or not, perhaps containing a nil-pointer, i.e., there was a point in the past run where the variable was initialized or not. Statically, a compiler warning about "uninitialized variables" typically means, the variable is potentially uninitialized ("may"), namely there may *exist* a run, where there is no initialization of a variable. Or dually, a variable is properly initialized at some point, when *for all* pasts that lead to that point the variable has been initialized. But for open programs (and/or working with abstractions), there may statically be more than one possible past and we cannot be sure which one will concretely be taken. Maybe indeed all or some of them will be taken at run time, when the code fragment being under scrutiny is executed more than once. That is the case when the analyzed code is part of a loop, or correspond to a function body called variously with different arguments. In summary, the distinction between "may" and "must" applies also to statically analyzing properties concerning the past.

Reusing and “freeing” a register

The liveness status of variables is crucial for register usage: the value for a variable being dead does not need to occupy precious register space. We promised in the previous paragraph to elaborate on that a bit, as it involves some fine points that we will see in the algorithm later, which may not be immediately obvious from reading the code.

First of all, as far as the hardware is concerned, there is no such thing as a full, non-free, resp. empty or free register. A register is just some fast and small piece of specific memory in hardware in some physical state, which corresponds to a bit pattern or binary representation. The latter one is a simplification or abstraction, insofar the registers may be in some “intermediate, unstable” state in (very short) periods of time between “ticks” of the hardware clock. So, the binary illusion is an abstraction maintained typically with the help of a clock, and compilers rely on that: registers contain bit strings or words consisting of bits. Of course, `00000000` does not “mean” empty. But when is a register empty then? As said, as far as the hardware is concerned, fullness and emptiness of registers simply does not exist. Those concepts only exist conceptually in the context of the semantics of the implemented programming language, in particular in the code generator, which has to keep track of the status and tracking the status of registers as full or empty. If the code generator wants to use a register (in that it generates a command that loads the relevant piece of data into the chosen register), generally “empty” ones are preferred, for instance one that so far has not been used at all. Initially, it will rate all registers as empty (though certainly some bit pattern is contained in them in electric form, so to say). Now in case a register contains the value for a variable, but the variable is known to be dead, doesn’t that qualify for the register being free? So isn’t it as easy as the following?

> A register is free if it contains dead data (or “no data” insofar as the register has not been used before)?

Sure enough, that’s indeed why liveness analysis is so crucial for register allocation. One has, however, to keep in mind another aspect. Just because the value of a register is connected to a variable which is dead does not mean one can “forget” about it and, by reusing the register, overwrite it. There are two reasons why that’s not enough. One is, that the content of a variable is kept in **two copies**, one in main memory and one in the register. And it may well be the case that the one in main memory “**is out of sync**”. After all, the code generator loaded the variable to register to do faster manipulations on the “variable”, therefore it is a good sign, so to say, that it’s out of sync. Keeping main memory and registers always 100% in sync is meaningless; then we would be better off without registers. Still, if the variable is really dead, what does this inconsistency matter? That’s the second point we need to consider: the concrete code generator later will do a block-**local** life analysis, only. So it can only know what’s going on locally and whether *in the current block* the variable is life or dead (respectively, all variables are assumed to be live at the end of a block). That’s different from temporaries, that are assumed to be dead at the end of the block . That means, “one” has to store the value back to main memory. Actually, “one” needs to store that value back, if “one” suspects the values disagrees, if there is an *inconsistency* between them. Who is the “one” that needs to store them value back? Of course that’s the code generator, that has to generate, in case of need, a corresponding store command, and it has to consult the register and address descriptors to make the right decision. After "synchronizing" the register with the main memory, the register can be considered as free.

Local liveness analysis here

That was a slightly panoramic view about topics we will touch upon in this chapter, but only slightly panoramic, as register allocation in general is a complex and much addressed problem. But the chapter will be more focused and concrete: code generation from 3AIC to 2AC, making use of liveness analysis which is mainly done *locally, per basic block.* We so far discussed live variable analysis and problems broader than we actually need for what is called *local* analysis here (local in the sense per basic block local). For basic blocks, which is straight-line code, there is neither looping (via jumps) nor is there branching (which would lead to "don't-know" non-determinism in the way described). Therefore we use techniques similar to what has been earlier called "static simulation": The live variable analyzer steps through the code line by line, and that may be called simulation.

There are two aspects worth noting in that context. One is, when talking about "simulation" it's not that the analysis procedure does *exactly* what the program will do. Since we are doing local analysis of only a *fragment* of a program, a basic block, we don't know the concrete values, that's not easily done (one could do it symbolically, though). But we don't need to pre-calculate the outcome, as we are not interested in what the program exactly does, we are interested in **one particular aspect** of the program, namely the question of the liveness-status of variables. In other words, we can get away with working with an **abstraction** of the actual program behavior. In the setting here, for local liveness, even given the fact that the basic block is "open", that allows *exact* analysis, in particular we know exactly whether the variable *is* live or *is not.* So the "may" aspect we discussed above is irrelevant, locally. The fact that we don't use the exact values of the variables (coming potentially from "outside" the basic block under consideration) does not influence the question of liveness, it's independent from concrete values. If we would have conditionals, that would change, because values would influence the control-flow. So, in that way it's not a "static simulation" of actual behavior, it's more simulating stepping through the program but working with an abstract representation of the involved data. As said, the concrete values can be abstracted away, in this case without loosing precision.

There is a second aspect we like to mention when calling the analysis some form of "static simulation". Actually, the live analysis that comes before the code generation, steps through the program in a **backward** manner. In that sense, the term "simulation" may be dubious (actually, the term static simulation is not widely used anyway, as mentioned earlier). But actually, in a more general setting of general data flow analysis, there are many useful backward analyses (live variable analysis being one prominent example) as well as many useful forward analyses (undefined variable analysis would be one).

With the liveness information at hand, the code generation will "step" though the 3AIC in a *forward* manner, generating 2AIC, keeping track of book-keeping information known as register descriptors and address descriptors. In that process, the code generation makes use of information whether a variable *is* locally live or *is not* locally live (or on whether a variable *may* be globally live or not when having global liveness info at hand). That means, prior to the code generation, there is a liveness analysis phase, which works *backwardly*.

Side remark 10.1.1 (Exactness of local liveness analysis (some finer points))**.** To avoid saying something incorrect when interpreted literally, let's qualify the claim from above that stipulated: for straight-line 3AIC, *exact* liveness calculation is possible (and that what we will do). That's pretty close to the truth...

However, we look at the code generation ignoring **complicating factors**, like more complex addressing modes, and "pointers". We stated above: liveness status of a variable does not depend on the actual value in the variable, and that's the reason why exact calculation can be done. Unfortunately, in the presence of pointers, aliasing enter the picture, and the actual content of the pointer variable plays a role. Similar complications for other more complex addressing modes. We don't cover those complications though. We focus on the most basic 3AIC instructions, but when dealing with a more advanced addressing modes (as done in realistic settings), the exact future liveness status would be known, not even for straight-line code. [2] covers also that, but it's left-out from the slides and the pensum.

There is another fine point. The assumption that in straight-line code, we know that each line is executed **exactly once** is actually not true! In case our instruction set would contain operations like division, there may be division-by-zero **exceptions** raised by the (floating point) hardware. Similarly, there may be overflows or underflows or other hardware-triggered errors. Whether or not such an exception occurs **depends** on the concrete data. So, it's not strictly true that we know whether a variable **is live or is not**. It may be, that an exception derails the control flow, and, from the point of the exception, the code execution in that block stops (something else may continue to happen, but at least not in this block). One may say: if such a low-level error occurs, probably trashing the program, who cares if the live variable analysis was not predicting the exact future 100%? That's a standpoint, but a better one is: the analysis actually did not do anything incorrect. The liveness analysis is a "may" analysis, and that even applies to straight-line code. The analysis says a variable in that block may be used in the future, but in the unlikely event of some intervening catastrophe, it actually may not be used. And that's fine: considering a variable live, when in fact it turns out not to be the case

> means to **err on the safe side**.

Unacceptable would would be the opposite case: an exception would trick the code generator to rate variables as dead, when, in an exception, they are not. But fortunately that's not the case, so all is fine. □

10.1.2 Code generation

In this section we work with 2AC as machine code (as from the older, classical "dragon book"). An alternative would be 3AC also on code level (not just for intermediate code); details would change, but the principles could be comparable. Note: the message of the chapter is *not*: in the last translation and code generation step, one has to find a way to translate 3-address code two 2-address code. If one assumed machine code in a 3-address format, code generation would face similar problems. The core of the code generation is the treatment of *registers*. The code generation and register allocation presented here is rather straightforward; it may look "detailed" and "complicated", but it's not very complex in that the optimization puts very much computational effort into the code generation. One optimization done is based on liveness analysis. An occurrence of a variable is "dead", if the variable will not be read in the future (unless it's first overwritten). The opposite concept is that the occurrence of a variable is live. It should be obvious that this is essential for making good decisions for register allocation. The general problem there is: we have typically less registers than variables and temps. So the compiler must make a selection: which data should be in a register and which not not?

A scheme like "the first variables in, say, alphabetical order, should be in registers as long as there is space, the others not" is not worth being called optimization... First-come-first-serve like "if I need a variable, I load it to a registers, if there is still some free, otherwise not" is not much better. Basically, what is missing is taking into account information when a variable is no longer used (when no longer live), thereby figuring out, at which point a register can be considered *free again*. Note that we are not talking about run-time, we are talking about code generation, i.e., compile time. The code generator must generate instructions that loads variables to registers it has figured out to be free (again). The code generator therefore needs to keep track over the free and occupied registers; more precisely, it needs to keep track of which variable is contained in which register, resp. which register contains which variable. Actually, in the code generation later, it can even happen that one register contains the values of *more* than one variable (in case two variables at some point are known to contain the same value). Based on such a book-keeping the code generation must also make decisions like the following: if a value needs to be read from main memory and is intended to be in a register but all of them are full, which register should be "purged". For that question, the lecture will not drill deep. We will concentrate on liveness analysis and we will do that in two stages: a block-local one and a global one in a later section. the local one concentrates on one basic block, i.e., one block of straight-line code. That makes the code generation kind of like what had been called "static simulation" before. In particular, the liveness information is *precise* (inside the block): the code generator knows at each point which variables are live (i.e., will be used in the rest of the block) and which not (but remember the remarks at the beginning of the chapter, spelling out in which way that this may not be a 100% true statement). When going to a *global* liveness analysis, that precision is no longer doable, and one goes for an approximate approach. The treatment there is *typical* for data flow analysis. There are *many* data flow analyses, for different purposes, but we only have a look at liveness analysis with the purpose of optimizing register allocation.

The **goals** of code generation is to produce **efficient** code. Small code size is also desirable, but less important. But the primary goal is, of course, **correct** code!

When not said otherwise: efficiency refers in the following to efficiency (or quality) of the *generated* code. Fastness of compilation, or with a limited memory footprint may be important, as well (likewise may the code size of the compiler itself be an issue, as opposed to the size of the generated code). Obviously, there are trade-offs to be made. But note:

even if we compile *for* a memory-restricted platform, it does not mean that we have to compile *on* that platform and therefore need a "small" compiler. One can, of course, do cross-compilation.

Correctness this is non-negotiable and a "binary" goal, the code generation is either correct or is not. Of course, compilers are complex, and bugs exists (and if found hopefully repaired) but no self-respecting compiler writer would describe a compiler with some known cases where the generated code does something erroneous as an "almost correct" compiler. Optimization is different, it's an important goal, and correct compilers can be more or less "optimal". Often there are also *conflicting* goals. We also mentioned that optimization can be (and is) done in different stages of a compiler (and in different ways). But later stages, like code generation is a *prime arena* for achieving *efficiency*. As also mentioned earlier, **optimal code** undecidable anyhow, and even if decidable (perhaps restricting oneself for straight-line code) it may be *intractable*)[1].

Besides that, one should also not forget that there are trade-offs, one has to agree on a measure of how "good" the produced code is. Later we will introduce a (simple) *cost-model* for that. Even if one agrees on a measure, the word optimization is not meant as producing an *optimal* piece of code in the conventional sense of being optimal. Its interpreted as techniques to achieve "good code".

Due to the importance of optimization at code generation time, it's often then time to bring out the "heavy artillery". So far, all techniques (parsing, lexing, even sometimes type checking) are computationally "easy" or made easy. Sure, one could invest in a parser that is computationally not easy, like defining a syntax that is ambiguous and handle that with a parser not only with unbounded look-ahead, but even using back-tracking. But what for? But at the later stages like code generation and optimization, even taking the platform into account: that's the time when an *investment* in aggressive, computationally complex and advanced techniques may be worth it. And indeed **many** different techniques are being used.

Concerning type checking, the situation is a bit different from parsing. Type checking (in particular in the way we presented it) can be simple. But that's not always the case. Type inference, aka type reconstruction, is typically computationally heavy, at least in the worst case and in languages not too simple. There are indeed technically advanced type systems around (including undecidable ones, like the one for $\mathrm{C}^{++}\ldots$). Nonetheless, it's often a valuable goal not to spend too much time in type checking and furthermore, as far as later optimization is concerned one could give the user the option how much time he is willing to invest and consequently, how aggressive the optimization is done. For our coverage of type systems in the lecture and the oblig: that one is rather simple and elementary, and poses no problems wrt. efficiency.

[1]The word "intractable" refers to computational complexity; intractable problems are those for which there is no *efficient* algorithm to solve them. Tractable refers conventionally to *polynomial time* efficiency. Note that it does not say how "bad" the polynomial is, so being tractable in that sense still might not mean practically useful. For non-tractable problems, it's often guaranteed that they don't scale.

10.2 2AC and costs of instructions

Here we look at the instruction set of the 2AC, resp. a *small subset* of an instruction set. We look at it from the perspective of a *cost model*. Later, we want to at least get a feeling that the code we are generating is "good" but then we need to what the cost is of the generated code, i.e., the cost of instructions.

Two-address code

When talking about 2AC, it's actually not a concrete instruction set of a concrete platform. Concrete chips have complicated inststruction sets, so it's more that we focus on a (very small) subset of what could be an instruction set of a 2A platform. Now, isn't that another "intermediate code"? We will see that the code now (independent from the fact that its 2AC) is more low-level than before. In that way, it could be a real instruction set of some hardware. The intermediate code from before could not. One could tell the same story we are doing here, translating from 3AIC to 2AC also by doing a translation from 3AIC to 3AC. Still that would pose equivalent problems (register allocation, cost model, etc.), but the presentation here happens to make use of a 2AC.

The not too many op-codes we cover are "typical" **two-address op-codes**, but not meant representing a particular *concrete* machine.

The **2-address instructions format** looks as follows

$$\texttt{OP}\ \mathit{source}\ \mathit{dest} \tag{10.1}$$

dest is not just the **destination** for a binary instruction, it's **also** a source for binary operations. The format describes the most general form of instruction (and that's what we will focus on), but also instructions with less arguments may be supported. Also instructions, where the *dest* is really just the destination, not *also* as source, an in the general case. Note the *order* of arguments here (esp. for minus). The order is not really per se important, nor is it a law of nature that the *source* must be mentioned first. But to be able to read later code snippets of generated code and to understand the code generator, one has to keep that order in mind.

As said, it's not only the number of allowed arguments which is a crucial difference between 3AIC and 2AC, and reducing 3 arguments into to is not the crucial challenge for this chapter. The important difference is that there are **restrictions** on *source* and *target* arguments:

Source and target arguments can only be refer to register or memory cell. The source can additionally be a target.

Besides the shown instruction format, there are of course further opcodes for conditional jumps, procedure calls . . . .

Here is a simple code snippet containing a few lines of numerical operations, followed by some jump.

```
ADD a b   // b := b + a
SUB a b   // b := b − a
MUL a b   // b := b ∗ a
GOTO i    // unconditional jump
```

Remark 10.2.1 (3A machine code as alternative and restrictions on operands)**.** Following [2], we base the presentation on 2AC. Note, machine code is **not** lower-level or closer to HW because it has one argument less than 3AIC. Instead of the format from equation 10.1, 3AC instructions could look as follows:

$$\texttt{OP}\ \textit{source1}\ \ \textit{source2}\ \ \textit{dest} \tag{10.2}$$

That resembles 3AIC instruction. The fact that the corresponding command would have been written like *dest* = *source1* `OP` *source2* is more a syntactic issue or an issue of readability. Of course, being basically at HW level now, the *actual* format of instructions is no longer a syntactical issue, it has become a *"physical"* issue. Instructions is a sequence of words, loaded by the processor on the chip over some bus, the single bits traveling as electrons over parallel lines of conductivity, so the format of the instruction (10.2) also reflects the design of the underlying hardware. And that will load the bits for the operator `OP` **first** etc., so the sequence of words in the format also reflects the way the bits are arranged in memory and the order in which they are handled by the hardware. See also Figure 10.1 for our 2AC.

Of course, if one programs in assembler code, the assembler editor may hide this and arrange the display for the progammer in a visually different way (perhaps using infix). At any rate, the order of how the individual instructions are written "notationally" is also not the main distinguishing point between intermediate code and actual code.

But what's then in general the *difference* of typical 2 (or 3) address **machine** code to 3A **intermediate** code? A main differerence is often a **restriction** on the **operands**. Hardware may for instance impose one or more of the following restrictions.

- only *one* of the arguments is allowed to be a memory access (or even: all arguments must be registers).
- no *fancy addressing* modes (indirect, indexed ... see later) for memory cells, only for registers.
- not "too much" memory-register traffic back and forth per machine instruction.

For instance

$$\&\texttt{x} = \&\texttt{y} + *\texttt{z}$$

may be 3A-intermediate code, but not 3A-machine code. Perhaps the machine code can do things corresponding to `*z` only on registers (maybe even so some specialized registers) and cannot have two operations like `&` and `*` in one instruction.

As we said, the code generation could analogously be done for 3AC instead of 2AC. But what's the difference then between 3AIC and 3AC, would the translation not be trivial?

Not quite, there is a gap between intermediate code and code using the instruction set. The most important difference is the use of registers. Related to that, 3AC instructions typically impose restrictions on the operands of the instructions. In the purest form, one may allow instructions only of the form `r1 := r2 + r3` (here addition as an example), where all arguments, sources and target, must all be in registers. That would result in a pure **load-store architecture**: before doing any operation at all, the code generator must issue appropriate load-commands, and the result needs to be stored back explicitly. That obviously leads at least to longer machine code, measured in number of instructions (but perhaps the instructions themselvelse may be represented shorter). Analogous restrictions may concern the indirect addressing modes. Instruction sets with a load-store design are often used in RISC architectures. □

10.2.1 Cost model

To speak about optimization, we need some well-fined **measure of quality** of the produced code. That's the cost model. It captures the fact that some instructions takes more time than others, without being exact in giving realistic timings for various reasons.

As for **cost-factors**. One is the **"size"** of the instruction, which is the base cost. On top of that comes the influence of different **address modes** as *additional costs* (see later). I.e. there are different (additional) costs for register access vs. main memory access vs. constants. Or also direct vs. indirect vs. indexed access.

How does the size of the instruction influences the time? Instructions need to be **loaded** into the processor. I.e., longer instructions may need longer, perhaps it takes 2 or more machine cycles to load the needed operator and operands into the processor.

The cost model (like the one here) is intended to model relevant aspects of the code, that influence the efficiency, in a proper and useful manner. The goal is not a 100% realistic representation of the timings of the processor. It will be based on assigning rule-of-thumb numerical costs to different instructions. Actually, the cost model captures not much more than the following very simple and obvious fact:

Accessing a register is "very much" faster than accessing main memory. And loading (and processing) longer instructions requires more cycles than shorter ones.

But the model does not use realistic figures (maybe by consulting the specs of the machine or doing measurements). Indeed, "main memory" access may not have a uniform access cost (in terms of access time). There are factors outside the control of the code generation, which have to do with the memory hierarchy. The code is generated as if there are only two levels: registers and main memory. But, of course, that's not realistic: there is caching (actually a whole hierarchy of caches may be used). Furthermore, data may even be stored in the background memory, being swapped in and out under the control of an operating system. Being not under the control of the code generator, those are stochastic influences. The compiler is not completely helpless facing caches and other memory hierarchy effects. Based on assumptions how caching and paging typically works, the code generator could try to generate code that has good characterisics concerning "locality" of data. Locality means that in general it's a good idea to store data items "than belong together" in close vicinity, and not sprinkle them randomly across the address space (whatever "belonging together" means). That's because the designer of the code generator knows that this suits chaching or swapping algorithms, that perhaps swap out cache lines, banks of adjacent addresses, whole memory pages etc. As far as caches are concerned, that's simply a rational hardware design. But one can also turn the argument around: hardware designers know, that it's "natural" that data structures coming from a high-level data structure of a structured programming language (and which contain conceptually data "that belongs together) will be generated in a way being "localized". Even if the compiler writer has never thought of efficiency and memory hierarchies, it's simply natural to place different fields of a record side by side. Also for more complex, dynamic data structures, such principles are often observed: the nodes of a tree are all placed into the same area and not randomly. More tricky maybe the the presence of a garbage collector, that could mess that up, if done mindlessness. But also the garbage collector can make an effort to preserve locality. So, in a way, it all hangs together: well-designed memory placement will be rewarded by standard ways managing the memory hierarchy, and well-designed memory management will run standard memory layout by compilers faster. It's almost a situation of co-evolution.

But all that is more a topic for how the compiler arranges memory (beyond the general principles we discussed in connection with memory layout and the run-time environments). Here we are looking more focused on the code generation and trying to attribute a round-about costs **on individual instructions**. So questions of locality cannot be considered, as they are about the global arrangement, neither are questions of caching, etc., as one individual instruction and the instruction set is not aware of caching, let alone the influence of the operating system.

So, how can we express the very rough above observation "registers are very much faster than memory accesses"? That's easy, register access costs "nothing", it will have a zero costs. Main memory accesses will have cost of 1. Also, the cost of 0 vs. the cost of 1, it's about time **additional** to the load and execution time to the operation. So doing `ADD r1 r2`, an addition involving 2 registers, costs 1 (say, one load cycle), only the register accesses don't add to the costs, their access and carrying out the addition are done within one single load cycle. Even if we had realistic figures from somewhere (via profiling and measuring average execution times under typical conditions or similar), the use would be limited: as stressed a few times, genuine and absolute optimal performance is (and cannot be) the goal (super-optimization aside). The goal is getting good or excellent performance with a decent amount of effort. We are content to use the cost model as a guideline (for the code generator) on decisions like

> when translating *one* line of 3AIC, shall I use a register right now or rather not?

We will see that this is the way the code generator will work. One might not even call it "optimization": it's *not* that first some not-so-good code is generated which afterwards is improved and optimized. The code generator takes the cost model into account on-the-fly, while spitting out the code. It does not even consult the cost model (by invoking a function, comparing different alternatives for the next lines, and then choosing the best). It simply compiles line after line, and the decisions are plausible, and one can convince oneself of the plausibility by looking at the cost model. Actually, one can convince oneself of the plausibility even without looking at the cost model, just preferring values from registers over main memory when possible. And that simple fact is one of two important pieces of common knowledge the cost model captures.

What's the second piece then? The other piece is that executing one command costs also something. So, each "line" costs 1. In that sense, the 0-costs of register access is realistic, insofar registers access is typically done in one processor cycle, i.e., in the same time slice than the loading and executing the instruction as a whole. So, in that sense, register accesses really don't cost anything additional. Other accesses incur additional costs, and since we don't aim at absolute realism, all the non-register accesses costs 1.

After all that background information, let's get more concrete. Figure 10.1 describes the **instruction format** in terms of bytes and their layout

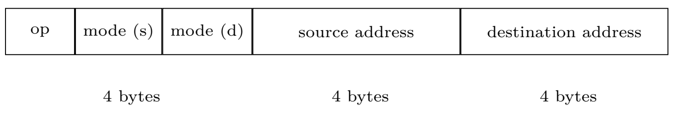


Figure 10.1: Instruction format

mode	abbr.	address	**added cost**	
absolute	M	M	1	
register	R	R	0	
indexed	c(R)	$c + cont(R)$	1	
indirect register	*R	$cont(R)$	0	
indirect indexed	*c(R)	$cont(c + cont(R))$	1	
literal	$\#M$	the *value* M	1	only for source

Table 10.1: Address modes and cost model

In the most general case, for our 2AC and for instructions with 2 arguments, an instruction is split into 3 parts each 4 bytes or 4 octets long. 4 bytes are also called one **word** in that architecture. The first word represents the op-code including information how to interpret the following two words, namely the source and the destination address; as mentioned, the destination address is also the address of a source, in a binary operation. The content of the source and destination arguments can be interpreted differently, that called their **mode** in the corresponding op-code. The mode of the two instruction arguments can be specified independantly and the various modes are summarized in Table 10.1 (in the right-hand column).

The modes called **indirect** and **indexed** in the table correspond to those discussed in connection with intermediate code (in connection with accessing slots in an array or fields of a struct). Note that those more advanced access modes assumes that their arguments reside in registers already.

We see that there are no real restictions when and when not memory access are allowed and when registers. Earlier we mentioned something like "load-store" architectures, where binary operations may only work on registers, or other restrictions. That is not the case here.

Remark 10.2.2 (Instruction format, HW architecture and comparison to byte code formats)**.** The format here corresponds to a 32-bit architecture, which is a popular format (actually, it's pretty old, there had been 32-bit machines early on, likewise also 64-bit (not micro-processors at that time). There are 16-bit microprocessors (in the past), and there are 64-bit processors as well. Of course, having 4 bytes for the op-code does not mean all bit patterns are used for actual instructions (that would be way too many). But we have to keep in mind (or at least in the back of our mind, as that's no longer the concern of a compiler writer): the instructions need to be handled by the given hardware with a given size of the "bus", there is no longer the freedom and flexibility of software. In particular, it's not "**byte code**" (more like 4-bytes code. . .) It's nice to think of a binary opcode as to represent "addition" or "jump", but the 0 and 1's in the code actually are connected to hardware, the slots in the 32-bit word are "wired up" connecting them to logical gates that open and close and trigger other bits/electrons to flow from here to there that ultimatly result in another bit pattern that can be interpreted as that an addition has happened (on our level of abstraction). So the actual bit-codes for the logical machine instructions are are "sparcely" distributed, and most bit-patterns are not simply unused ("undefined"). If used they would open and close the "logic gates" of the chip in a weird, meaningless manner. As said, all that is not the concern of a compiler writer, who can see an `add`-code as addition, but it's interesting that the story does not end there, there are complex layers of abstraction below that and also, that we are leaving the world of "anything goes" of software: the compiler writer can design any form of intermediate representations in intermediate codes and translate between them etc. But below that, things get more restricted by physics and the laws of nature. □

Example 10.2.3 (Cost model example `a := b + c`). The following examples are not breathtakingly interesting. They show different possible translations and their costs. The first two Listings 10.1 and 10.2 show two equivalent ways of translating the given assignment, one operating directly on the main memory, one partly loading the arguments to a register and then using that. Both versions have the same cost, in our cost model (despite the fact that the first program executes 3 commands and the second only 2).

The other two examples from Listing 10.3 and 10.4 represent the same command, but under a different assumption, namely: the arguments are already loaded in some registers. Listing 10.3 assumes that `R0`, `R1`, and `R2` contain **addresses** for `a`, `b`, and `c`. In Listing 10.4 in contrast, the assumption is that `R1` and `R2` contain **values** for `b`, and `c`. Either way, that drives down the costs. But that should be pretty clear, that's why one has registers, after all.

We also see that to profit from the use of registers, the code generator needs to **know** which variables are stored in the registers already. That will be done later by so-called *address descriptors* and *register destriptors.*

Also, especially the second example shows, that sometime the generated code is a bit strange: Since we have only 2AC, one argument is source, the other one is source *and* destination. That means, 2AC like addition "destroys" one argument. That means, in general we need to temporarily copy that argument somewhere else, otherwise it gets lost.

In the second example, since `a` is updated anyway, the first step uses `a` for that temporary copy of `b`. That's a **general pattern** of this form of code.

```
MOV b, R0   // R0 = b
ADD c, R0   // R0 = c + R0
MOV R0, a   // a = R0
```

Listing 10.1: Using registers, costs=6

```
MOV b, a    // a = b
ADD c, a    // a = c + a
```

Listing 10.2: Memory-memory ops, costs=6

```
MOV *R1, *R0 // *R0 = *R1
ADD *R2, *R0 // *R0 = *R2 + *R0
```

Listing 10.3: Addresses in registers, costs=2

```
ADD R2, R1   // R1 = R2 + R1
MOV R1, a    // a = R1
```

Listing 10.4: Storing back to memory, costs=3

□

10.3 Basic blocks and control-flow graphs

In the introductory overview of this chapter and elsewhere, we have mentioned the concepts of basic blocks and control-flow graphs already. The notion of control-flow graph is used, in this lecture, at the level of IC (maybe 3AIC), resp. also machine code. The notion of CFG makes also sense on highler levels of abstractions, i.e., one can do a control-flow graph also for abstract syntax. In our setting, there would be not much difference between to control-flow graphs from intermediate code and machine code. Both representations make use of jumps and conditional jumps and labels (resp. addresses), and that determines the edges of the graph.

In this section, we work with 3AIC, generated from some AST probably with higher-level control-flow constructs like two-armed conditionals and loops. Now we "reconstruct" a more high-level representation of the code by figuring out the CFG (at that level). It is not uncommon extract a CFG first,[2] and *use* the CFG assisting in the (intermediate) code generation. In such a settings, the control-flow graphs are and explicit data structure, as another intermediate representation.

Anyway, the general concept of CFG works analogously at different levels, same for basic blocks, at least when working with a standard procedural language.

Basic blocks are largest possible sequences of straight-line code. A **control-flow graph** is basically **graph** with basic blocks as nodes and jumps, resp. conditional jumps and fall-throughs as **edges.**

The characterization of control-flow graph does not cover procedure calls. In its basic form, a control-flow graph represents the control flow of the body of *one* procedure. One basic block, corresponding to a node in the control-flow graph, is (a largest possible) sequence of instructions without **jumps in or out**. It's also the elementary unit of code analysis/optimization.

[2]See also the written exam 2016.

When saying, a CFG is “basically” a graph, we mean that, apart from some fundamentals which makes them graphs, details may vary. In particular, it may well be the case in a compiler, that cfg’s are some accessible **intermediate representation,** i.e., a specific concrete data structure, with concrete choices for representation. For example, we present here control-flow graphs as *directed* graphs: nodes are connected to other nodes via edges (depicted as arrows), which represent potential successors in terms of the control flow of the program. Concretely, the data structure may additionally (for reasons of efficiency) also represent arrows from successor nodes to predecessor nodes, similar to the way, that linked lists may be implemented in a **doubly-linked** fashion. Such a represantation would be useful when dealing with data flow analyses that work “backwards”. As a matter of fact: the one data flow analysis we cover in this lecture (live variable analysis) is of that “backward” kind. Other bells and whistles may be part of the concrete representation, like dedicated start and end nodes. For the purpose of the lecture, when don’t go into much concrete details, for us, cfg’s are: nodes (corresponding to basic blocks) and edges. This general setting is the most conventional view of cfg’s.

10.3.1 From 3AC to CFG: “partitioning algorithm”

As said, control-flow graphs are **reconstructed** here from a linear representation, said 3AIC. Actually, the fact that we use 3AIC as starting point for the extraction of the graph is not really important. It would also work the same way for our p-code, or any comparable linear instruction formal. All what’s needed is that the code supports jumps and conditional jumps and label, and that’s characteristic from linear intermediate codes and also machine code. Actually, even if the code does not officially have labels, as for instance in plain machine code and addresses as jump targets, one can do the same with addresses instead of symbolic labels (or the graph extraction at the same time also introduces symbolc labels for the relevant adresses, which at the same time serve as name of the control flow graph).

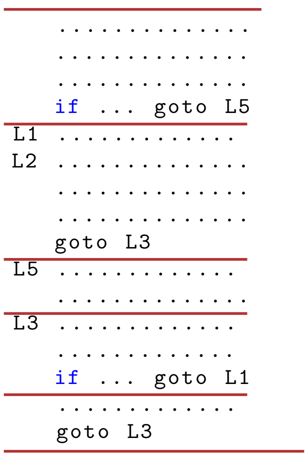
The CFG is determined by something that is called here **partitioning algorithm**. That’s a big name for something pretty simple. We have learned in the context of *minimization* of DFAs the so-called partitioning refinement approach, which is a clever thing. The partitioning here is really not fancy at all. The task is to find in the linear code largest stretches of straight-line code, which will be the nodes of the CFG. Those blockes are **demarkated by labels and gotos** (and of course the overall beginning and end of the code.) There is only one small addition to that: an unused label, i.e., a label not used as target of some jump, does not demarkate the border between to blocks, obviously. An unused label might as well be not there, anyway.

The construction is often described making use of the concept of a **leader** of a basic block. That’s a fancy name for the first line of a basic block.

If the code contains a statement **GOTO** i, then line labelled i is a leader. Instruction *after* a **GOTO** or a conditional goto is a leader. And the instruction sequence from (and including) one leader to (but excluding) the next leader or to the end of code forms one **basic block**.

In case the labels are not “before” a line, but as pseudo instructions occupying a line themselves, the leader would be the first real instruction in the basic block. Either way, it’s the same thing, pseudo instructions are not really there anyway, they as are just used to give a name to real instructions.

An example for a partitioning of a piece of code is illustrated in Listing 10.2. The red lines show the demarcations between the code of the basic blocks. The lines at the same time correspond to what we called *leaders*: the leaders are the lines *following* the the red lines and they indicate the first line of a basic block. Threre is one exception, that’s the red line at the end of the program. That one, obviously, does not correspond to a leader or the beginning of some bacic block. It demarcates, however, the end of the last basic block. Note also that the line labelled L_2 is *not* a leader. The reason is that in the sketched program, this label is not used as jump target, unconditional or otherwise.



```
     ..............
     ..............
     ..............
     if ... goto L5
L1   .............
L2   ..............
     ..............
     ..............
     goto L3
L5   .............
     ..............
L3   .............
     .............
     if ... goto L1
     .............
     goto L3
```

Figure 10.2: Partitioning (illustration)

It is worth thinking about what would happen if we considered L_2 a header nonetheless. In that case, the basic blocks would no longer be the *largest* sequences of straight-line code. In our example, we would end up with 6 basic blocks instead of 5. That should, however, **not affect the correctness** of the generated code. As mentioned, basic blocks are the elementary levels of optimizations and code generation. Cutting the basic blocks smaller than necessary will lead to smaller stretches of code targeted by local analysis. An example would be the local liveness analysis covered later. If one uses liveness analysis only on the local level, i.e., only inside basic blocks, then the smaller than necessary basic blocks would lead to a less precise analysis. Liveness analysis (like others) can be precise within basic block, but typically resorts to approximation more globally, like doing analysis for a whole control-flow graph. In Section 10.7, we will look into that kind of global analysis. But when doing only a local one, the analysis ignores what happens outside the current basic block, and thus, to play it safe and assumes variables at the end of a basic block potentially used later. It assumes a variable at the end of a block to be *live*, though a global analysis may reveal that it is not. This safe overapproximation is typical for many forms of analyses, in particular data flow analyses, but also type checking. As a consequence, unnessessarily small blocks of straight-line code lead **loss of precision**, an approximation which is still safe safe but needlessly approximative.

Indirectly therefore, also register allocation is affected by a too finegrained block structure.

As long as the lifeness approximation is correct or safe, the register allocator will lead to correct code as well, though presumably slower code compared to more precises liveness information.

This describes what happens to liveness analysis and register allocation if the straight-line code blocks are needlessly small, if one assumes local analysis only. The situation for other kinds of analyses would be similar.

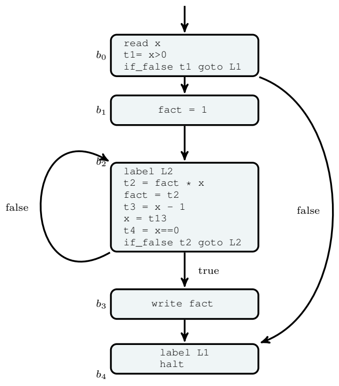
What would happen using small straight-line blocks if one employed a **global analysis?** In this case, one typically would *not* loose precision. The global analysis anyway looks at the whole control-flow graph. Unlike for local liveness analysis, to stick with this form of analysis, a global analysis will not of course *assume* that a variable at the end of a basic block is live, just to be safe. It will investigate to figure out if the variable may be used in the future or if it's sure it's dead. That's what the global analysis is good for, after all.

Does it mean, if one is doing a global analysis anyway, the size of the blocks of straight-line code does not matter? In a way, yes. As said, one will not loose precision by being to finegrained. In an extremal case, one could use every instruction line as one elementary block. Why would one still like to work with the largest possible stretches of straigh-line code, i.e., with basic blocks of the form introduced?

The reason is mostly that the global analysis can be done if not more precisely, but more **efficiently.** Global analysis typically involves the analysis of loops or cycles, something that, by definition, is not needed for straight-line code. The analysis of cycles in the control-flow graph entails that one does analysis steps *repeatedly* for nodes participating in a cycle. If one has a large basic block as part of a cycle, one can analyse relevant information (for instance concerning the liveness status of variables) one in summarized form. For instance, let's assume the first usage of a a variable, say x in a given basic block is that it's assigned to like in a line of the form $x := e$, where e does not refer to x. That means, x actually is dead at the beginning of said basic block. A local analysis of the block will find that out, and one can use the information in summarizing corresponding information for the basic block for all variables. Resp. one could do that summary information for all basic blocks, which form the nodes of the control flow graph. What good would that do? The local analysis, needed for the summary information steps through the lines of the basic block. As we will see, one single pass through the lines is enough. Actually, it should be even intuitively clear, that one pass should be enough to see locally, for each variable, if it's used or not. Anyway, the basic block, as said, may be part of a cycle in the graph, and cycles need repeated treatment. But with the summary information precomputed by local analyses, one at least need not step through the individual ones over and over again, to (re-)discover the liveness status of the involved variables, like rediscovering that x is dead at the entrance of the basic block; that information is remembered in the summary. It's in a way like memoization of the the local liveness situation for the basic blocks. In this way, the analysis may become faster.

Let's have a look at some more concrete example. Listing **??** shows 3AIC for the factorial function from the previous chapter, and Figure 10.3 shows the corresponding control-flow graph. The code contains 5 basic block and thus the illustration of the control-flow graph 5 nodes. The first line in each node is the corresponding header. Unlike in the schematic example from Figure 10.2, all labels in the code are jump targets. Typically, the (intermediate) code generator would not generate labels not being used as jump-target, though they are not “harmful”; the partitioning algo does not treat them as leaders and the label instructions from the 3AC are *pseudo-intructions*, i.e., the don’t correspond ultimately lead actual machine-code instructions.

Example 10.3.1 (Control-flow graph of the factorial function). Let’s use as example the 3AIC for the factorial function. We had encountered the example already for intermediate code generation. The corresponding CFG is shown in Figure 10.3.



```
read x
t1= x>0
if_false t1 goto L1
```

```
fact = 1
```

```
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t13
t4 = x==0
if_false t2 goto L2
```

```
write fact
```

```
label L1
halt
```

Figure 10.3: Control-flow graph (factorial)

We see that gotos and conditional goto never *inside* a block, but not every block ends in a goto or starts with a label. So it’s not that labels and blocks are in exact correspondance (and in the picture, the blocks are named b_i). □

10.3.2 Levels of analysis

Figure 10.3 contains the control-flow graph of the factorial procedured, resp. the graph for a main procedure of a program that realizes the factorial function. Note that the program does not do procedure calls; the factorial is calculated using a while-look in the source language and it’s not the recursive factorial solution, one often finds. Anyway, the control-flow graph and the analysis one can do on that graph is **intra-procedural**. That refers to notions and analysis “inside” one procedure, not across

More general is **inter-procedural** analysis. Inter-procedural analyses are harder, resp. require more effort than *intra*-procedural. In this lecture, we don’t cover inter-procedural considerations. Except that call sequences and parameter passing has to do of course with relating different procedures and in that case deal with inter-procedural aspects. But that was in connection with run-time environments, not what to do now in connection with analysis, register allocation, or optimization. So, in this lecture resp. this chapter, “local” refers to inside one basic block, “global” refers to across many blocks (but inside one procedure). Later, we have a short look at global liveness analysis. As mentioned, we don’t cover analyses across procedures, in the terminology used here, they would be even more global. Actually, in the more general literature, global program analysis would typically refer to analysis spanning more than one procedure. Indeed, one should avoid talking about local analysis without further qualifications; it’s better to speak of block-local analysis, procedure-local, method-local, or thread-local, to make clear which level of locality is addressed. We are doing block-local analysis resp. procedure-local analysis and will call the latter "global". In general, the more global, the more *costly* the analysis and especially the more costly the corresponding optimizations.

10.3.3 Loops in control-flow graphs

Next we comment on so-called loops in control-flow graph, without going into much detail. Loops in programs are thankful places for optimizations. That's sometimes called *loop optimization.* That's not necessarily specific analysis or optimizations working only for loops. For instance, liveness analysis the way presented here work for control-flow graphs with or without cycles in the graph. But a good or aggressive analysis and register allocation for loops, specially for smalls ones that are taken very often during execution may greatly improve the performance.

Programming language looping constructs —while-loop, for-loop etc.—- leads to cycles in the graph. However, **not all cycles in a cfg are loops.** In other words, the concept of loops in control flow graphs is **not** identical with **cycles** in a graph.

All **loops** are graph **cycles** but not vice versa.

Intuitively, loops are cycles originating from source-level looping constructs, like while. Gotos, on the other hand, may lead to non-loop cycles in the CFG.

Why does one even bother to make that distinction? Actually, sometimes one does not. For instance, later we will look into global liveness analysis, a form of data flow analysis. For that we sketch an algorithms, that works for general control flow graphs, with loops or cycles (and also for control-flow graphs without cycles . . .). We won't show in detail how to tune the data flow analysis algorithm for efficency, for instance relying on particular graph traversal strategies.

But that's the point: loops as a restricted form of cycles would allow particular traversal strategies that perform better, but they can't be applied on general cycles. Of course, with no cycles at all and the control-flow graph as directed acyclic graph, even better strategies are possible. As a matter of fact, the *local* liveness analysis we look at first deals with basic blocks, i.e., straight-line code, and that is *even more* restricted than just being acyclic. And thus the liveness analysis become even more simple and efficient.

Besides that fact that loops have have better properties compared to non-loop cycles when it comes to analysing the CFG, they are also more well-behaved when considering certain **optimizations** or code transformations (based on analyzing the code). That's a slightly different thing. Certain analyses may get slower for non-loops, but certain code transformations don't work for non-loops. We will come back to that a bit later after defining in more detail what a loop actually is, not just stating that it comes from a source-code looping construct.

The analysis, that figures out which cycles in a graph are actually loops and which not is sometimes called **loop discovery**. However, in modern lanuages, it's no longer an important analysis; in the absence of gotos or similar commands, all cycles are loops.

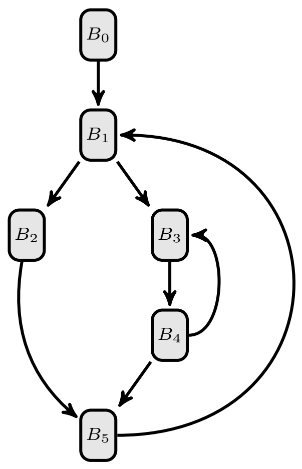


Figure 10.4: Loop example

Cycles in a graph are well-known. As said, loops, while closely related, are *not* identical with cycles. So, loop-detection is not the same as cycle-detection. Otherwise there'd be no much point discussing it, since cycle detection in graphs is well known, for instance covered in standard algorithms and data structures courses like INF2220/IN2010.

Definition 10.3.2 (Loop in a (control-flow) graph)**.** A **loop** L in a CFG is a set of nodes, including **header node** $h \in L$ such that

1. for any node in L, there is a non-empty path *inside* L **to** h, and there is
2. a path inside L **from** h to any node in L. Finally,
3. there is no edge in the graph that goes into a node of L **other** than h **from the outside** of L.

In control-flow graphs, one often has an additional assumption. It's actually not so much a restriction on loops, it's more an assumption on control-flow graphs. It's assumed that the "initial node of a control flow graph for a procedure is *not* itself an entry or header node of a loop. Same for cycles: the entry node should not be part of a cycles, should the control-flow graph have cycles. That's mostly for convenience, making certain analysis a tiny bit more straightforward (avoiding to take care of some corner cases.[3]

The definition is best understood in a small example.

Example 10.3.3 (Loop example)*.* See the control-flow graph from Figure 10.4. The graph contains the following two loops

$$\{B_3, B_4\} \quad \text{and} \quad \{B_4, B_3, B_1, B_5, B_2\}$$

The first loop is a *nested* loop. The unique entry for the loops is marked in red. The set

$$\{B_1, B_2, B_5\}$$

[3] We had comparable and actually related restriction for context-free grammars, where it also helped avoiding said corner cases. There we sometimes assumed that the start-symbol should never occur on the right-hand side of a production. That would mean it would occur in a cycles. To "repair" that, one could add an addition start symbol, say S' instead of S. But the fact that context-free grammars and control-flow graph are both abbreviated as CFG, that's of course coincidence ....

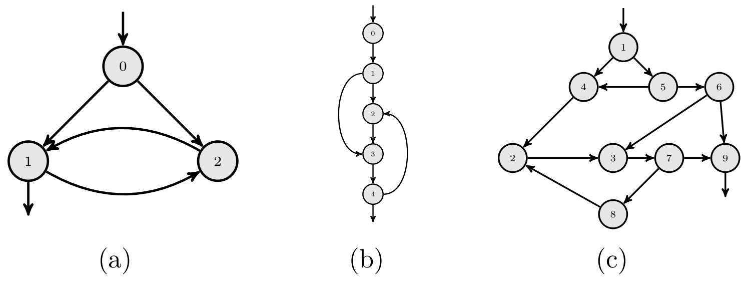


Figure 10.5: Non-loops

is not a loop.

Figure 10.5 shows some graphs without loops, but cycles. □

The first two points from the definition make the nodes of a loop **strongly connected**: all nodes in a loop can reach each other. From the algorithm and data structure lecture INF2220/IN2010, one may have encountered the notion of *strongly connected component* and the corresponding algorithm. A strongly connected component adds "maximality" to the requirement of being strongly connected; i.e., a strongly connected component is a maximal set of strongly connected nodes. This maximality is not required for loops.

For instance in the graph from Figure 10.4, the inner loop $\{B_3, B_4\}$ is strongly connected (and a loop), but it's not a strongly connected component, since one could enlarge it to $\{B_1, B_2, \ldots, B_5\}$ and still stay strongly connected. The notion of stronly connnectd components corresponds to outermost loops (including all nested loops within).

Let's look at a small example for optimization opportunities in connection with loops. We don't look at a control-flow graph representation, as an analyser and optimizer might do, but illustrate optimizations at source code level (see Listing 10.5).

```
while (i < n) {i++; A[i] = 3*k }
```

Listing 10.5: Loops as fertile ground for optimizations

One possible optimization is to move the computation `3*k` out of the loop. In the control-flow graph, it would be placed right in front of the header node.[4] That this is possible of course rests on the observation, that `3*k` is constant inside the body of the loop. That's easy enough to see here in the example, but that needs to be established by a particular analysis. Another form of optimization in connection with loops is to put variables used repeatedly in the loop into registers (like i here.)

We don't need to explore loops further, actually for the way we do global analysis later (in the form of global liveness analysis) that will work for non-loop cycles ("unstructured" programs) as well as for loop-only graphs, in the version we present it. If one knows that there are loops-only, one could improve the analysis (and others). Not in making the result of the analysis more precise, but making the analysis algorithm more efficient. That could be done by exploiting the structure of the graph better, for instance exploiting that loops are *nested*, targeting inner-loops first. In the examples from Figure 10.5, such strategy

[4] That's one of the motivations for unique entries.

would not apply, as cycles are not necessarily nested. Since we don't exploit the presence of loops, we don't dig deeper here. It should be noted that the definition of loops (with unique entry points) is a classical concept for CFGs and program analysis. One may, however, find material ignoring the subtle, traditional difference between cycles and loops and where the two notions are used interchangably.

One is interested in loops not necessarily as a concept in itself, but in the larger context of optimization. We called loops a fertile ground of optimizations, which is of course also true for general cycles: both involve (potential) repetition of code snippets, and shaving off execution time there, that's a good idea. Often, the optimization is about moving things outside of the loop, typically "in front" of the loop. That's when a unique entrance of a loop comes in handy (sometimes called a loop-header). The non-loop examples don't have a single loop-header.

In the more or less distant past, loop detection (and cycle detection) would be a task a compiler would engage in. Now, that most programs are written following structured progamming, there a no non-loop cycles. Additionally, when compiled from source code, the program structure contains all the information where the loops are, so there not need to do an analysis (for instance for the intermediate code) to (re-)discover them at the lower level. However, the partitioning algorithm we discussed is a bit in that spirit. The control flow structure is (re-)discovered from (intermediate) code, in the form of the control flow graph.

10.3.4 Data flow analysis in general

Data flow analysis is a **general** analysis technique (or a class of analysis techniques) working on control flow graphs. There are *many* different concrete forms of such analyses. Often analyses work hand in hand with a particular **optimization**. For a particular optimization, one needs particular pieces of information, for instance about variables, or about values at different points in the control-flow graph. For instance, for optimizations like the one sketched in Listing 10.5, one needs to figure out if variables change their values in the body of the loop (`k` in the example).

In our lecture, the optimization goal is to obtain a decent register allocation, and the corresponding information is the *liveness* status of variables and the data flow analysis we use is *liveness analysis.*

Not all data flow analyses are done with optimization in mind. For instance, the user could be interested to know, whether there are potentially unititialized variables in the program or potential nil-pointer exceptions. etc. Such questions can be analyzed by tailor-made data-flow analyses, but there are no optimizations connected to it. The only results are warnings to the user about potential sources of trouble, and it's left to the user to deal with it.

Why are such analyses called *data flow analyses*, resp. what's data-flow anyway? And while at it: what's *control flow?* After all we are doing data-flow analysis on a control-flow graph. The words refer to the fundamental separation between two aspects of a program:[5] There is the **data** on the one hand, stored in memory/temporaries/registers etc. On

[5] At least in procedural languages, in functional languages the line is blurred.

the other hand, there is the **control**. That's the code. The current point of control is a particular point in the control-flow graph. Ultimately, in the running program it's represented by *instruction pointer*, pointing to the address of the next instruction to be executed in the code segment. So the "control" of a running program moves by changing the instruction pointer. Inside elementary blocks, the instruction pointer is incremented, moving though instruction after instruction, and (conditional) jumping corresponds to setting the instruction pointer to the target address of the jumps (and that corresponds to an edge in the CFG). The distinction between "data" and "control" is also visible in the *memory layout* discussed in connection with run-time environments, where the code in the code segment, and all the data placed separately.

Side remark 10.3.4 (Control-flow analysis)**.** We have data-flow and control-flow and we are discussing data-flow analysis on control-flow graphs, in particular later in the form of liveness analysis. Is there also something like **control-flow analysis?**

Indeed there is. Earlier we discussed a way to extract the control-flow graph from a given linear code representation (say, 3AIC or similar). That was called a bit bombastically a *partitioning algorithm.* In effect, it can be called a **control-flow analysis**. Actually, most would not bother to call it control-flow analysis, because also that is too bombastic for the simple thing it does. "Proper" control-flow analysis is reserved for complex situations, when the control-flow situation is not as obvious as in our case. It is obvious in our setting since the jumps go to fixed labels.

Things get more complicated, in particular in connection with functions. Function calls involve jumps (as part of the call sequences), and in complex enough languages, the called function may not be statically known. That's the case for languages with references to functions or function variables, in particular also for language with higher-order function. Also for late-bound methods, the jump-target as part of the call is not known statically. In such situations, the compiler may invest in an analysis to narrow down potential jump-targets. That's normally what would be called control-flow analysis, though calling our partitioning algorithm is certainly also a control-flow analysis, though a very trivial one, and one that does not need techniques similar to data flow. □

Data flowing from (a) to (b): Given the control flow of the program, normally as control-flow graph, the question is: is it *possible* or is it *guaranteed* that some "data" originating at one control-flow point (a) reaches control-flow point (b). ("may" vs. "must" analysis).

The characterization of data flow may sound plausible: some data is "created" at some point of origin and then "flows" through the graph. In case of branching, one does not know if the data "flows left" or "flows right", so one **approximates** by taking both cases into account. The "origin" of data seems also clear, for instance, an assignment "creates" or defines some piece of data (as l-value), and one may ask if that piece of data is (potentially or necessarily) used someplace else (as r-value), without knowing resp. being interesting in its exact value that is being used. This is sometimes also called def-use analysis. Later we will discuss definitions and uses. Another illustration of that picture may be the following question: assuming one has an data-base program with user interaction. The user can interact by inputting data via some (web)-interface or similar. That information is then processed and forwarded to some SQL-data base. Now, the inputs are points of origin, and one may ask if this data may reach the SQL database without being "sanitized" first (i.e., checked for compliance and whether the user did inject into the input some escapes and SQL-commands).

Anyway, this picture of (user) data originating somewhere in a CFG and then flowing through it is plausible and not swrong per se, but is too narrow in some way. It sounds as if every data flow analysis traces (in an abstract, approximative manner) the flow of pieces of data through the graph.

Not all data flow analyses are like that. Actually, the live variable analysis will be an example for that. So more generally, it's more like that "information of interest" is traced through the graph. Since the information of interests may not be an abstract version of real data, it may also not necessarily be traced in a **forward** manner. For liveness analysis, one is interested in whether a variable may be used in the future. So the information of interest is the locations of usage. That are the points of origin of that information one is interested in. And from those points on, the information is traced **backwards** through the graph. So, this is an example of a *backward analysis* (there are others). Of course, when the program runs, real data *always* "flows" forwardly, as the program runs forwardly: first data orignates and later is may be consumed. But for some analysis, like liveness analysis, one changes perspective: instead of asking: where will information originating here (potentially or necessarily) flows to in the future, one asks:

> where did information or data arriving here orignate from (potentially or necessarily) in the past.

Let's also comment on the treatment of **basic blocks**. Basic blocks are maximal sequences of straight-line code. We encountered a treatment of straight-line code also in the chapter about *intermediate* code generation. The technique there was called *static simulation* (or simple symbolic execution). *Static simulation* was done for basic blocks only and for the purpose of *translation*. The translation of course needs to be **exact, non-approximative**. Symbolic evaluation also exist (also for other purposes) in more general forms, especially also working on conditionals.

In summary, the general message is: for SLC and basic blocks, *exact* analyses are possible, it's for the global analysis, when one (necessarily) resorts to overapproximation and abstraction. After a general discussion of liveness analysis in Section 10.4, we cover liveness analysis in basic blocks in Section 10.5 and global liveness analysis in Section 10.7.

10.4 Liveness analysis (general) and a variation (def/use)

Liveness analysis is a classical data flow analysis. The introductory remarks of this chapter already introduced what liveness of a variable means, and why that is important for register allocation. There are many different data flow analyses, and liveness anaysis is only one typical example of that form of semantics analysis (but a very important one). It's typical insofar that the ideas and technique for liveness analysis apply analogously to other data flow analyses. By the underlying principles, we mean mostly the way the *global* analysis is approached in Section 10.7 and the way the approximation is done in an iterative manner.

Let's first think about how to tackle **basic blocks** (more details in Section 10.4). The question is to figure out at each point in a given block, whether a variable is live or not. Live at least as far as the current block is concerned. Focusing locally one single block means, the analysis does not have information about what will happen after that block. As a consequence, the analysis *assumes* that variables are live at the *end* of a basic block. This assumption is done in the spirit of safe approximation. If, seen globally, a variable would actually not be live (statically or dynamically), the register allocation would at least make no error. On the other hand, if a variable would be judged dead in contrast to the real situation, that could lead to wrong code in that the content of the variable may get lost, even if it still needed in the future.

In the local live variable analysis, variables and temporaries (i.e., temporary variables) are treated analogously, with one exception, an that the mentioned assumption at the of a block: proper variables are assumed live, as explained. For temporaries, the liveness analysis exploits knowledge about how temporaries have been generated in the intermediate code generated. For each intermediate results, for instance for compound expressions, a new temporary variable is created to hold that intermediate value *temporarily*. The way that works also implies that temporary variables are never (re-)used across the boundaries of a basic block. So that means, at the end of a basic block, temporaries are assumed to be dead, and that is more than an assumption; it reflects reality, at least as long as the intermediate code is generated the way described.

In the following, when we say "variable" we mean proper variables as well as temporary variables.

The question whether a variable is live or not refers to "control points" in the program. So it's not about "is variable x live or dead?", it's about "is variable x live at this point". Obviously, at some points in the program, x may be used in the future, and, at other points, it may no longer be used. If held in a register, when a variable's status turns from live to dead, the register allocator may decide to re-use the register, which may involve to save the register back to main memory. But that will the register allocator's task in Section 10.8, here we are just figuring out the liveness information the allocator can make use of.

The "points" in the program here refers to "lines" in the straight-line code. Actually, it's not actually that variable live in a given line, that is too imprecise. It's actually the question

whether a variable is live right before a given line, or right after it.

One has to make that distinction, since obviously, the liveness status of a given variable can change at a given line. For example, for a statement `x := 4`, variable `x` is definitely dead before that statement, but may well be live afterwards.

Of course, the liveness status right after line number n is identical to the live status right in front of line $n+1$. This distinction between "right-in-front-of" and "right-afterwards" can also be applied to whole basic blocks. One can figure out, what is the liveness status of a variable right in from of a basic block, which means right in front of its first line, and right afterwards. One cannot do that for a single basic block in isolation. For the same reason, one cannot for instance, figure out for a single line inside a basic block in isolation, say for `x := 4` from above, whether `x` is live or not afterwards inside. Neither can one figure out whether `y` is live in front (assuming `x` and `y` are different variables). That's also the reason why for local blocks and local liveness analysis only, proper variables are *assumed* live at the end; when doing only local analysis, one simply does not know what is the case. So when lifting the "right-in-front-of" and "right-afterwards" considerations to the level to whole basic blocks, that will be done for the global level, analysing a complete control-flow graph in Section 10.7. The corresponding information will be called *inLive* and *outLive*.

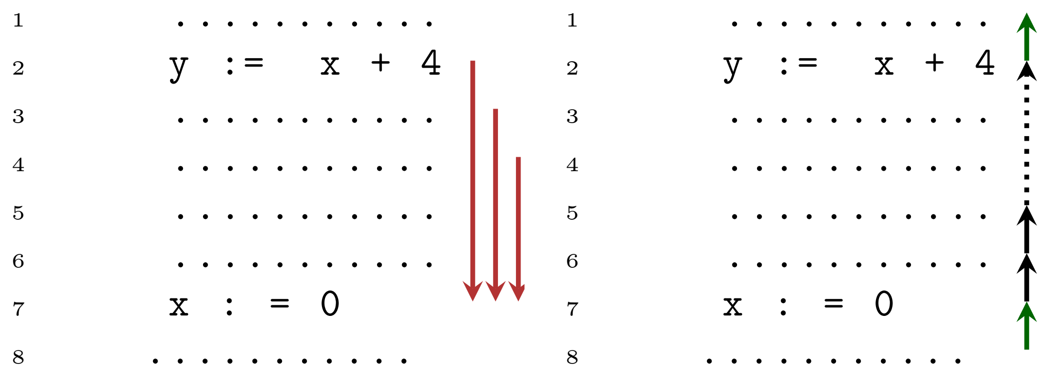
Coming back to the local analysis: it should be intuitively clear that it's quite straightforward to do the liveness analysis, i.e. to determine the liveness status for each each variables and for each point after resp. before each line in block. A variable is live at a given point, if it is used later, but without being overwritten in the meantime. The situation where a proper variable is neither used nor overwritten for the rest of the block is not much different.

So to check if a variable, say `x` is live at a given point, one may be tempted to proceed forwardly and check for the following lines if `x` will be used in the future inside this block without being overwritten first (then it's live), or the first thing that happens in the future is being overwritten (then it's dead), or nothing happens the future inside this block, neither reading from it nor overwriting it, in which case the variable is assumed live resp. dead, depening on whether we are speaking about a proper variable or a temporary variable.

One can do that for all points in the straight-line code and for all variables, and that considerations shows that the determining the lifeness status inside a basic block is decidable. However looking for future uses of variables in the sketched way, checking each point in the program *independently* is absurdly inefficient.

The reason why it's inefficient is that an independent checking partly obtains the same information over and over again. As part of the problem, one needs to determine the liveness status for a variable at a point say at beginning of line n (or for all variables at that point, it does not matter for the argument). For instance in Figure 10.6a, assume we want to determine the liveness status for x for all lines. For instance, if one wants to determine it at the end of line 2, one can search forward. Assuming that the lines of the form `.....` have nothing to do with x, then the first use of x is discover in line 7, at which point it becomes clear that x is not live at the end of line 2.

To do local liveness analysis means we need to do similar considerations for all points in the basic block. For instance for the end of 3 and line 4 etc, as illustrated by the other two arrows in the figure. Of course, figuring out the liveness information for the end of line 3 and 4 simply repeats the search done for the end of line 2 already. So, the different lines are better not treated as independent problems; one better reuses the information obtained for one line when doing another line. And the best way to do that is to proceed *backwardly*



(a) Forward (b) Backward analysis

Figure 10.6: Local liveness: forward vs. backward

That is illustrated in Figure 10.6b, again for variable x. Of course, the analysis that steps through the program in the sketched backward manner will treat all variables at the same time, not doing the same backward scan over and over for each variable. Anyway, proceeding backwards means, the analysis starts at the very *end* of the basic block. Assuming that x is a proper variable, it means, the x is *assumed* live at the end of the block. That's indicated by the green arrow. Stepping thought the code backwards, it remembers right in front of the assignment in line 7, resp. at the end of line 6, that variable x is dead now, indicated by the black arrows. Continuing the backward scan, this information is propagated though the lines with decreasing number, since nothing happens wrt. to variable x, until at the beginning of line 2 resp. end of line 1. At this point, the variable is *live* again, indicated the green arrow. The being live information is then propaged further on backwards, in the example till the beginning of the program. Actually, the liveness algo later will not just propagate the binary liveness information live vs. dead (here green vs. black), but it also indicates for live variable, the location of the next use.

That's an information that could be exploited by the code generator resp. register allocator. When it has to make the decision which of two live variables to keep in a register, preference could go to the one whose next use is nearer in the future. The actual code generator we look at in Section 10.8, does not actually make use of that information, resp. we don't go so deep into the details of the decision making process of the code generator to see in which way that next-use information of life variables can be used.

Figure 10.6b sketches how the "data" is propagated through the lines of the basic block, resp. information of interest *about* the data. The real data, integers in the example, is handled in the programs via assigning it to variables ("defs") an reading the variable later ("uses"). In this way, the data "flows" forward in an execution. After all, an execution does so in a forward manner. Here, the information of interested is not the data itself, but information about when corresponding variables are assigned to resp. read. This (information about the) data flows backwards. For straight-line code as in this section, that leads to a single pass through the code. In that sense the information "flows" through the code exactly ones, here backwardly (which corresponds to the fact, that the lines of a piece of straight-line code *in isolation* execute exactly onces, as well, though in a forward manner, of course.)

Going beyond straight-line code, there will be edges of the control flow graph to be considered. In case of multiple edges connected to a node, the information of the analysis will flow equally "both ways" (or more in case of more than two edges). In the more general setting, the basic blocks are then part of a control flow graph, which typically contains cycles. Thus, a single-pass of analysis is no longer sufficient, and the "data flow " *circulates* through the graph. That will be covered in Section 10.7.

This treatment, single pass for straight-line code resp. circulating data flow in a whole control-flow graph is characteristic for data-flow analysis. What is *not* characteristic for all of them is that the analysis data flows *backwards* through the straight-line code as in 10.6b, resp. backwards through the graph as discussed later. Liveness is information about the *future*, i.e. whether there will be (or might be) a place where a variable is used. As explained, instead of seaching forward as illustrated in Figure 10.6a, one arranges for a backward propagation of the relevant information. In other situations, one is interested in information about the past instead. For instance, analyzing wheher all variables have been properly initialized previously. This reverses the picture, and the corresponding analysis works by *forward* data flow.

Definition 10.4.1. A "variable" is **live** at a given control-flow point if there *exists* an execution starting from there (given the level of abstraction), where the variable is *used* in the future [Make more precise]

The notion of liveness given in the slides corresponds to **static liveness**, he notion that (static) liveness analysis deals with. That is hidden in the condition "given the level of abstraction" for example, using the given control-flow graph. A variable in a given *concrete* actual execution of a program is *dynamically live* if in the future, it is still needed (or, for non-deterministic programs: if there exists a future, where it's still used.) Dynamic liveness is undecidable, obviously. We are concerned here with static liveness, if one wants to be very precise.

10.4.1 Variation on the topic: def/use analysis

We should have a good impression of what liveness analysis is supposed to do. It's about analyzing whether, at a given point in the program, a variable will (potentially) be used in the future.[6] Thinking about it, that formulation is factually not really precise. A variable being dead at some point may well be used in the future! The more precise formulation is that liveness is the question whether for a variable at a given point the current **value** at that point will be used in the future. Despite that, conventionally one talks about (static) liveness of variables. With this clarification, it's also clear, a dead variable variable may well be used in the future, being dead just means it's current value will no longer be used. If

[6]When saying variable, we also mean temporary variables.

the next thing that happens in the code is that the variable is *assigned to*, and afterwards, the new value is read, then in the span between now and the point of assignment, the variable is dead, but after the assignment it becomes live again.

With this clarified: we see important points in the "life time" of a variable (resp. the values stored in a variable) are points where the variables are read, resp. when they are assigned to. Those are important points simply by the fact that those are the points where the liveness status of the variable can change. When reading a variable, the status may change from live to dead. That would happen when the reading is the last time the current value is read. When writing to a variable, the status may change from dead to live. That will in general be the case, because writing to a variable where afterwards the variable is dead would mean, one has written a value to the variable that's never used and the value was not needed at all, which should happen not too often.

Writing to and reading from a variable is often referred to as places of **definition** resp. **use** of the variable. The corresponding analysis is also known as **def/use** analysis. For each definition of a variable, it shows the (potential) uses. In case there is no use, the variable is dead. It should be clear that it can be seen as a variation or generalization on the theme live-variable analysis: instead of just a boolean dead/live information, def/use-analysis answers the following question: given a "use", determine all possible corresponding "definitions".[7]

When doing block-local analysis, each each line has has exactly one place where a given variable has been defined, i.e., assigned to last (or else the variable is not assigned to in the block at all). In other words, for a use-def style of analysis, each use is connected to at most definition. For def-use, in contrast, one definition can be connected to *more* than one use. After all, a variable can be read multiple times.

Example 10.4.2 (Defs, uses, and liveness). Let's have a look at the CFG of Figure 10.7.

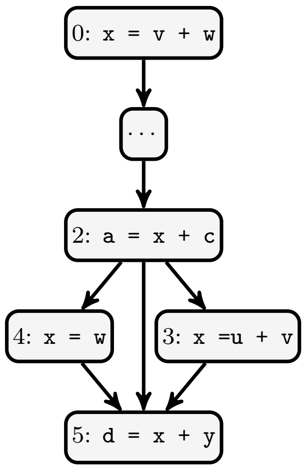


Figure 10.7: Defs and uses in a CFG

[7]There is also the converse analysis called **use-def** analysis: given a "use": determine all possible "definitions".

- x is “defined” (= assigned to) in 0, 3, and 4.
- the definition of x in 0 is used in 2 and potentially in 5 (assuming that the “...” in the non-labeled node don’t mess with x ...). The picture does not contain a graphical representation connecting the defs and the uses.
- u is **live** at the end of block 2, as it *may* be *used* in 3.

Note: In this simple illustration, we consider liveness across block-boundaries, i.e. global liveness, but each block contains only one instruction here. □

10.5 Local liveness: dead or alive

We have in general terms discussed different aspects concerning data-flow analysis, including aspects of global analysis which comes later. Let’s turn back to the more focused task of local liveness analysis. We start first with a plain version. The algorithm calculates, for each point in a basic block and for each variable, a binary piece of information whether the variable at a given point is variables is live or not, dead or alive, so to say.

That’s straightforward and should be reasonably clear already from the informal discussion so far, in particular in connection with Figure 10.6b, motivating the backward-scan idea. Later, we extend that version with information about next-use information about variables, and introduce the concept of the so-called **dependence graph** in Section 10.6.

Inside one block, local liveness is mostly about optimizing the (register) use for temporaries. Of course also for user variables, but user variables, unlike temporaries, have a life span exceeding basic blocks, the local liveness analysis cannot really do them justice, or at least only in a limited way.

Temporaries are symbolic representations to hold intermediate results, generated on request. Conceptually there is an unbounded reservoir reservoir of temporaries, but of course only a restricted, fixed number of registers, the exact number is platform dependent.

In the liveness analysis, we rely on the following **assumptions** about temporaries and variables. Temporaries **don’t transfer data** across blocks (unlike program variables). As a consequence

Temporaries are **dead** at the end of a block. In contrast, variables are **assumed** live at the end of a block.

The assumption that variables are to be assumed live is a consequence of doing a block-local analysis. If it’s not know whether a variable will be used or not, one has to do a conservative, safe abstraction, which means to consider them live. That temporaries can be assumed dead must be guaranteed by the way the intermediate code generator works. Indeed, our code generator always creates fresh temporaries. Another observation concerning how temporaries are created is: the first occurrence of a temporary in a local block is *always* on the left-hand side of an assignment. Inside a block, temporaries are *never* used first, they are always “defined” first. As a consequence, temporaries are **dead** at the beginning of a block.

Example 10.5.1 (3AIC code). We use the 3AIC from Listing 10.6 to illustrate the backward scan more conretely. The example will be reused also when extending later in Section 10.6 the current liveness algo.

```
t1 := a - b
t2 := t1 * a
a  := t1 * t2
t1 := t1 - c
a  := t1 * a
```

Listing 10.6: 3AIC example

□

Side remark 10.5.2. In intermediate code generated the way disucssed in the previous chapter: temporaries are *always* generated new for each intermediate result, so t_1 in the example is "unrealistic" for generated intermediate code. But also that is not important for liveness analysis. It works for variables and temporaries alike, re-assigned or not. □

Let's call variables or temporaries on the right-hand side of an instruction **operands**. Note: 3AIC also allows literal constants as operator arguments; they don't play a role for liveness analysis.

We said, liveness is about to determine for each variable, whether its current content it will (or may) be used in the future at each given (control) point in the code. The control-points correspond to lines in the code, but we have look more precisely, as mentioned: It makes a difference whether the control is in front of a line or directly after. As far as liveness analysis is concered, the liveness status of variables or temporaries can change when executing one instruction in a block. Immediately in front of the line, the temporary may be live, for instance because it is used right in that line as operand. But if that's the last use of the temporary in that block, it will be dead afterwards.

Consider the statement

$$x_1 := x_2 \ op \ x_3 \tag{10.3}$$

For that, the liveness situation for variables is characterized as follows:

A variable x is live at the **beginning** of $x_1 := x_2 \ op \ x_3$, if

1. if x is x_2 or x_3, or
2. if x live at its *end*, if x and x_1 are different variables

A variable x is live at the **end** of an instruction,

- if it's live at *beginning of the next* instruction
- if **no next** instruction:
  - all temporaries are dead
  - user-level variables are (assumed) live.

The definition explains for each line, how the liveness status in front of the live **depends on** the lifeness status at the end of the line. It does so for lines of the given form $x_1 := x_2 \ op \ x_3$; for other forms of lines, like $x_1 := op \ x_2$, the definition needs to be adapted in the obivous manner. Of course, statemens like `jump 3` need *not* to be considered for a block-local analysis. Jumps transfer control between different basic blocks.

Back to the 3 address assignment statement: as said, the definition explains the dependence of the flow information before the statement on the status of the information at the end of the statement. This dependence is a *function* from the exit of the statement to the entry of it. This functional dependency is called the **transfer function** (of that line, resp. the statement in a given line). Note that the liveness information of the *entry* point of a line is expressed as function of the corresponding information at the *exit*, not the other way around. This is of course characteristic for **backwards** analyses like liveness analysis.

We the transforfunction for one line (in the form of equation (10.3), it's easy to formulate the local-liveness as a simple backward scan of a given basic block. See Listing 10.7.

```
// ——— initialise T ——————————————————————
  for all entries: T[i,x] := D
  except: for all variables a // but not temps
              T[n,a] := L,
//———— backward pass ————————————————————
for instruction i = n−1 down to 0
   let current instruction at i+1: $x := y\ op\ z$;
      T[i,o] := T[i+1,o] (for all other vars o)
      T[i,x] := D // note order; x can ``equal'' y or z
      T[i,y] := L
      T[i,z] := L
end
```

Listing 10.7: Local liveness (dead or alive)

The table is a two dimensional, there is one slot per variable and per line. Each line can change the liveness information for one or more variables (that what conteptually the transfer function is doing) so the liveness information at the end of each line is different from that in front of a line. The entries in the table or two-dimensional array represent the information **at the end** of the corresponding line. That's of course the same as at the beginning of the next line. It's assumed that the line numbers go from 1 to n (not from 0 to n). The loop steps down to determine the effect of all lines numbered n to 1: note that what is called "current instruction" in the loop refers to the line with $i + 1$ in the code. Even of there is no instruction with line number 0, the corresponding entry representing the "end of line 0" respresents the liveness information at the beginning of the first line, i.e., at the beginning of the whole block.

Earlier we mentioned in passing the notion of *transfer functions*, without going into details. The code of Listing 10.6, stepping backwards through the lines does not *explicitly* make use of a separately defined transfer function. Implicitly, the transfer function is executed in the body of the loop, updating the entries of the table.

Example 10.5.3 (Local liveness). Revisiting Example 10.5.1, the result of the run of the liveness algorithms for the 3AIC code from Listing 10.6 is given in Table 10.2.

The analysis operaters with binary information and thus the table contains binary information (dead or alive). Later in Section 10.6 we will extend that information and (mildly) extend the algorithm. Revisiting the same example means that we get a (mildly) extended version of this table. The extension is the following: for live variables, one does not report the fact that the variable is live, but also point to the line where it is used next. □

line	a	b	c	t_1	t_2
[0]	L	L	L	D	D
1	L	L	L	L	D
2	D	L	L	L	L
3	L	L	L	L	D
4	L	L	L	L	D
5	L	L	L	D	D

Table 10.2: Liveness analysis example: result of the analysis

10.6 Local liveness^{++}: Next-use information and dependence graph

In this section we revisit the dead-or-alive algorithm from Listing 10.7. The previous version was binary in that it determined the liveness status for each variable. It needs only a *minor extension* to obtain better information. Instead of determining whether a variable is dead or alive inside the current block (resp. *assumed* live in case of a proper variable at the end of the block), one can determine, where resp. when they are used in the future, if not dead. This is done below in Listing 10.8, a mild extension indeed of the one from Listing 10.7.

So, the extended algo keeps track of where the **next use** for each variable will be. That's done here by tracking the **line number** of the next use. That's usually precise enough. One might also track where a variable is used *inside* a line, as the first argument of a operation or the second argument (or both). That's normally seen as overkill, so we track the line only. The analysis works not with the binary information L and D, but for liveness, the information is $L(n)$, where n is the line number where the variable or temporary in question is used next. The number refers to a line **inside** the basic block. As explained, proper variables are *assumed* live at the end of a block (unlike temporaries, which are rated as dead). Of course, in such a situation, the analysis can't determine the line of the next use. We are currently doing a block-local analysis, so we have no information about subsequent blocks; that's why we just *assume* variables to be potentially live. Besides that, on a more global level, it makes no real sense of talking about **the** next use of a variable. Due to branching, there may be multiple next uses. If one wanted a next-use information, in would be a set of next-use points in the general case.

Anyway, in the situation here, a variable **assumed** live is captured by the notation $L(\bot)$.

```
// ——— initialise T ————————————————————————
  for all entries: T[i,x] := $D$
  except: for all variables a // but not temps
              T[n,a] := $L(\bot)$,
//———— backward pass ——————————————————————
for instruction i = n−1 down to 0
   let current instruction at i+1: $x := y\ op\ z$;
      T[i,o] := T[i+1,o] (for all other vars o)
      T[i,x] := $D$ // note order; x can ``equal'' y or z
      T[i,y] := $L(i+1)$
      T[i,z] := $L(i+1)$
```

```
end
```

Listing 10.8: Local liveness (with next use information)

Example 10.6.1. Let's revisit Example 10.5.3 and the 3AI code of Listing 10.6. The result of applying the algorithm on the code is shown in Table 10.3. Since the algorithm is a straightforward generalization of the previous binary version, the new table is a straightforward generalization of the previous Table 10.2 (on the same example). □

line	a	b	c	t_1	t_2
[0]	$L(1)$	$L(1)$	$L(4)$	D	D
1	$L(2)$	$L(\perp)$	$L(4)$	$L(2)$	D
2	D	$L(\perp)$	$L(4)$	$L(3)$	$L(3)$
3	$L(5)$	$L(\perp)$	$L(4)$	$L(4)$	D
4	$L(5)$	$L(\perp)$	$L(\perp)$	$L(5)$	D
5	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	D

Table 10.3: Liveness analysis example: result of the analysis

So, we see the next-use extension is really straighforward. We mentioned already in the discussions at the beginning of Section 10.4, in which way register allocation can profit from the extra next-use information and we could leave it at that. However, the next-use information that one can calculate by doing liveness analysis is closely related to another important concept resp. another intermediate representation. So we take the opportunity to discuss that shortly here as well.

All next usages and dependence graph

We extend the next-use information one step further, not just tracking *the* next-use, but **all next uses.** That results in an intermediate representation is known as **dependence graph**. We will (shortly) discuss it for basic blocks since we are currently doing *local* liveness. One can also generalize it to whole control-flow graph analogous to the fact that one can do liveness on a whole control-flow graph. But let's stick to the local level.

Let's assume we have a 3A code, like the 3AIC from example from Listing 10.6. 3A code instead of intermediate code would work similarly; likewise one could do a dependence analysis for forms of 2-address codes.

At any rate: a typical line in the 3-address code consists of a left-hand side and a right-hand side, as in instructions $x_1 := x_2 \ op \ x_3$. In such a line, x_1 on the left-hand side is "defined" and x_2 and x_3 are "used". The next-use form of liveness analysis figures out where inside a block for each point and for each variable, the next use will occur (resp. assumed somewhere outside the block). As said, we now generalize that a bit more, and **track all next usages** in the future (inside the block, resp. somewhere outside the block). That's not a big extension. A clever register allocator may also profit from this more detailed information about the use of variables; not ours though, it will look at the binary dead-or-alive information only. But something else is more important.

If we track all future usages of a variable at different points in the basic block, we in particular have information in a line of like $x_1 := x_2\ op\ x_3$ about the next usages of the variable *defined* in that line, i.e. x_1. This information thus connects the definition of x_1 with all its future uses. Normally, one contents oneself to connect definitions and usages per line (as we did with the next-use information). There is anyway only *one* variable on the left-hand side of each line, and whether this variable "definition" is later used as first operand or second operand in some line is not really important.

Keeping it on the per-line level, it means one connects a line like $x_1 := x_2\ op\ x_3$ with all lines later that *use* x_1 (without that x_1 is being overwritten in the meantime, of course). The later lines (resp. the uses in that later lines) then are said to (directly) **depend** on said line (resp. depend on the *definition* of x_1 on the right-hand side of that line).

Direct dependence is a **def-use** situation, and an analysis that figures it out is a **def-use analysis**.

We mentioned earlier, that def-use analysis is closely related to liveness analysis, and here we see more clearly how.

If one tracks the dependencies of the described kind (i.e., def-use connections) that results in a graph, the mentioned **dependence graph**. This is another well-known intermediate representation (different from control-flow graphs, ASTs, 3AIC, etc.).

Inside a basic block, the dependence graph is **acyclic**. In other words, the corresponding graph is a directed acyclic graph (**DAG**) with the lines as nodes and the (direct) dependencies as edges. Instead of viewing the lines and dependencies as DAG, one can equivalently view the lines as **partially ordered** (each DAG corresponds to a partial order, and vice versa).

Why is def-use information, i.e. the dependence graph relevant? It expresses ordering constraints, making clear which lines of the code (here 3AIC) needs to be executed *before* others, since the latter depend on the former. This dependence is only a partial order on the lines of a basic block, not a total or linear order and some lines are **independent:** there is no dependence directly or indirectly in either direction. Being independent means, the order or execution is irrelevant. In other words, the line-wise linearization in the 3AIC (or later 3AC or 2AC etc.) is a *particular* linear arrangement, more strict than actually necessary given the partial order of dependencies.

A **dependence analysis** could reveal the looser partial order and thus reveal whether 2 statements need necessarly be executed in the order as listed in the code, the latter one depending or the earlier one, or whether the compiler could *reorder* then.

That is known as **out-of-order** execution. Figuring out a good order of execution, for independent instructions is known as **instruction scheduling**.

Actually, also the processor can have facilities of out-of-order execution of machine code instructions, but that's outside the control of the compiler. However, knowing the rules

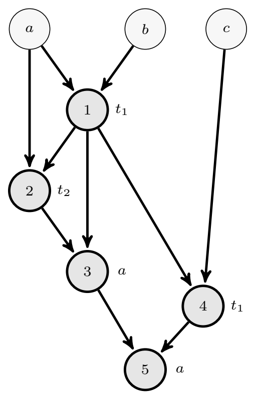


Figure 10.8: Dependence graph for the 3AIC code block

and conditions under which a processor does out-of-order or overlapping execution could be exploited by a code generator "scheduling" the instructions in a way that suits well to the platform's corresponding capabilities. For that, the compiler needs to figure out dependencies of the data (and registers, etc.).

This explanation analyzes the linear IR of, say, 3AC or 3AIC to determine which orders in the sequential arrangment of lines of instructions are real and which are spurious. With that help, one can rearrange the linear code, if deemed profitable. An alternative view is **not** to take the linear instruction sequence as primary intermediate representation, for instance inside a basic block. One could use DAG-based representation instead, i.e., the dependence graph and linearize the DAG in a subsequent stage. These alternatives are comparable to the situation with control-flow graphs. One can use them as intermediate representation to a lower level intermediate representation. Or, analyze a lower-level representation like the machine code or intermediate code to "reconstruct" from the linear representation the control-flow graph (as done by the simple partitioning algorithm in Section 10.3.1.

So much about motivating dependence graphs and what they code be used for (our code generator won't make use of them).

Example 10.6.2 (Dependence graph). Let's look at Listing 10.6 from Example 10.5.1 again. The result of the next-use analysis from Table 10.3 can be generalized to the dependence graph of Figure 10.8.

We see that the temporary defined in line 1 has three uses, namely in the lines 2, 3, and 4. In the linear code arrangement, the next use of the definition t_1 in line 1 is by the subsequent line 2. The DAG also makes clear that those lines 2, 3, and 4 are independent and could be executed in any order (or in parallel). These three edges correspond to the fact that in Table 10.3, the definition or assignment to t_1 in line one is marked as $L(2)$, $L(3)$, $L(4)$ in lines 1, 2, and 3.

In line 4, t_1 is "re-defined", i.e., assigned to again. Therefore, the entry $L(5)$ in the table does not refer to the first assignment to t_1 in line one, but the assignment in line 4. Remember, in Table 10.3, the stored information corresponds to the next-use or liveness information at the *end* of the corresponding code line. □

ASTs and DAGs

Let's have a last look at dependence graphs and explore a connection with ASTs. In principle, this is not new information, we have introduced both concepts, but perhaps it worthwile to spell it out more explicitly, looking at a few more examples.

Example 10.6.3. Let's start with the 3AIC for the source code assignment

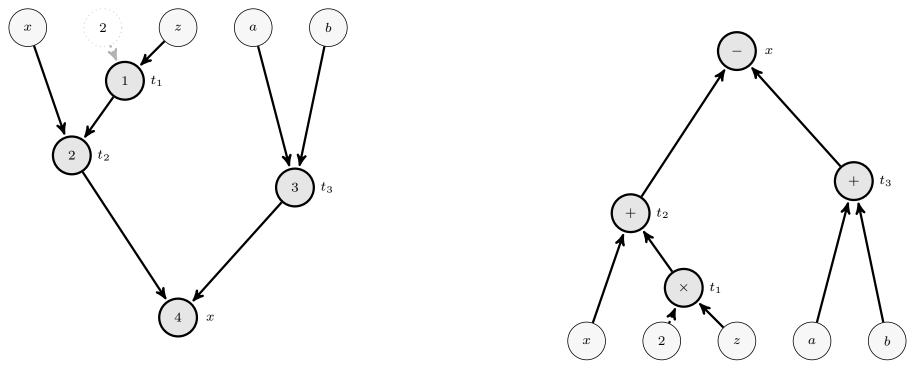
$$x := (x + 2 * z) - (a + b)$$

as shown in Listing 10.9.

```
t1  :=   2  *  z
t2  :=   x  +  t1
t3  :=   a  +  b
 x  :=  t2  -  t3
```

Listing 10.9: 3AIC for $x := (x + 2 * z) - (a + b)$

The corresponding dependence graph is shown in Figure 10.9a. As inner nodes of the DAG, we use the line numbers from 1 to 4. The inner nodes in the picture are also labelled with the variable or temporary being *"defined"* in the node. The first three lines calculate the **side-effect free** expression on the right-hand side of source code assignment, and the "numbers" of the temporaries $t_1, \ldots, t_3$ correspond to the line number; that's the way the intermediate codegenerator works (if we assume "numbered" temporaries starting from say 1). The DAG shows also a node correspond to the constant 2. Normally one would not bother to include the node and the corresponding edge into a DAG. The outcome or value of t_1, which is "defined" by the right-hand side $2 \times z$ does not depend on 2 insofar that it is a constant anyway. □



(a) DAG of the right-hand side

(b) The AST of right-hand side

Figure 10.9

It might not come as a complete surprise that the dependence graph from Figure 10.9a is basically nothing else than the **abstract syntax tree** of the right-hand side $(x+2*z) - (a+b)$ upside down; Normally, ASTs are written, like trees often are, with the root node on top. For the DAG, I choose to write it the other way around, so that the def-nodes come before the use-node, i.e., higher-up, as in the code and the dependence edges go down. An AST is shown in Figure 10.9b.

So, determining the DAG from a piece of 3AIC reconstructs in some way the AST. At least conceptually and in this example. Of course, the AST as concrete data structure is most probably represented differently from the dependence graph inside the compiler, and the two structures, if a compiler uses them both, serve different purposes. That the AST and the DAG are basically the same in this example is also caused by the fact that the code example is very simple: just an assignment to a variable with a pure, **side-effect free** expression on the right-hand side. As seen in the corresponding chapter, the intermediate code generator simply traverses the abstract syntax tree, generates a fresh temporary for each intermediate result, i.e. or each inner node of the AST. So we obtain three temporaries t_1, t_2, t_3, which are *defined* i.e., assigned-to, in the corresponding lines of the 3AIC in Listing 10.9. These lines or temporaries are the source nodes of the DAG, the soucre node of an edge is always the "def", the target node(s) are the "use(s)". In the simple example, each defined temporary has exactly one use, which correspond to the parent node. That makes the DAG in the example a **tree** (the syntax tree of the expression on the right-hand side of the assignment).

The connection between ASTs and dependence graphs is less close in more general situations, like the DAG example from earlier for a basic block and even less so for a complete procedure; actually in the presence of loops in the code or cycles in the CFG, also the def-use dependencies may no longer be *acyclic*, i.e., the dependence graph is no longer a DAG.

In the simple situation of the DAG from Figure 10.9a, the tree structure illustrates a fact which we knew all along: in a pure, side-effect free expression, it does not matter **in which order subexpressions are executed**. The nodes or the different subtrees are indepdendent in the DAG, i.e., unconnected by dependence edges.

In general, the DAG for the dependence graph may not not resemble the AST, of an expression, not even in a simple, side-effect situation like here. Here, for expressions and basic blocks, the connection between AST and code, here in 3AIC, is very direct and the connection is so direct and close, that we can effectively revert the translation, so to say, **decompile** the code from Listing 10.9 back into the AST of the expression.

The connection may not be so direct. The 3AIC may have undergone some optimization. Same already for the AST from the source code. If we look not at 3AIC, but at machine code (for which one could likewise do a dependence analysis), the distance to the source code ast is even larger, and additional optimization may have done. The compiler resp. compiler related tools could even make effort to make decompilation harder resp. harder to understand the result of a decompilation. Often, that is done, however, at the source-code level. In that case, it's known as **source-code obsfuscation** or **code hardening** or **encryption.**

To conclude, let's have a look at another, slightly more complex example.

Example 10.6.4. Assume the following expression

$$(x := x + 3) + 4 \; .$$

. The corresponding code is given Listing 10.10. We have seen a quite similar example and the code already in the chapter about intermediate code generation. It's an "expression" containing side effects.

```
t1  = x + 3
x   = t1
t2  = t1 + x
```

Listing 10.10: 3AIC for $(x := x + 3) + x$

The corresponding dependence graph is shown in Figure 10.10. This time, it's not a tree, but still a DAG. Note also, the order of evaluation in the expression now *does* matter, unlike before. The generated code assumes that the arguments of a binary operator like $+$ are to be evaluated from left to right and the generated 3AIC does exactly that. In the top-level addition, the x on the right-hand side uses the incremented value of x. That's also visible in the dependence graph in the edge from 2 to 3. The lines of the code cannot be reordered of course. □

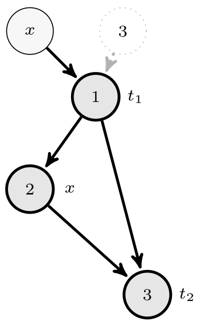


Figure 10.10: DAG of an expression with side effects

Connection to SSA

Starting from liveness analysis, we took the opportunity to introduce next-use information resp. introduced def-use analysis and the notion of dependence graphs, here DAGs. We take another opportunity and discuss to some extent another important concept used in intermediate representations, the notion of **static single assignment** format. We can stratch only the surface of it, in particular, as we focus on basic blocks. If generalizing to whole control flow graphs, additional complications would enter the picture, which we don't cover here.

Nonetheless, the conceptual core of single-assignment format can be understood here. Under this restriction, straight-line code it's almost trivial, and is also connected to the def-use analysis and thus live variable analysis. So here is a good place to introduce some ideas behind the single assignment format.

The section header mentioned **static** single assignment or SSA. For straightline code, there is no difference btween static single assignment and (general) single assignment.

Statically, a variable is "single-assignment" in a piece of code, if there is at most one assginment to it mentioned in the code.

In the terminology here, there is statically at most one definition of a variable. We can exclude the degenerated case of variables that are defined but not used. Those are useless, they are never live at all. If we exclude that, the requirement for static single assignment is that all variables are defined exactly once i.e. assigned to exactly one. Of course there are in general more than one use per definition.

In the presence of cycles and procedures, the fact that there is exactly one place in the code where a variable is defined, that does not guarantee that at **at-runtime**, the variable is assigned-to exactly once. A program that assigns every variable only once at run-time is single-assignment, which is (much) more restrictive than to be in static single assignment or SSA format. For blocks of straight-line code, like we are discussing here, there is no difference in SSA and single-assignment. Being defined textually once in a basic block means, it's assigned-to exactly once per execution of that block (if no "exceptions" derail the execution and prevent the assignment). Actually, the same could be said about acyclic control-flow graphs; those could originate from a program using conditionals, but not loops. See for instance the one from Figure 10.7. Also for code of that shape, assigning every variable once implies single assignment. But let's stick to basic blocks.

Looking at Listing 10.6, that one is **not** in single-assignment format. Both variable a and temporary t_1 are assigned to twice. As mentioned earlier, the code from that listing cannot be the result of the intermediate code generator we discussed, at least not directly. Maybe indirectly via some optimization or other, or manually given, but it does not matter: it's anyway a good idea that liveness analysis or dependence analysis works not just for some particular way of generating (intermediate) code.

But, the code generator would indeed *not* reuse t_1 in the second assignment but would use t_3 instead. Of course, the use in the last line of what is now t_1 would then refer consistely to t_3 when using the definition of t_3 in line 5. In other words, the intermediate code generator already generates temporaries that follow the single-assignment pattern. If a global counter is used for the temporaries, it's even in single-assignment format globally; in particular and additionally, temporaries don't transfer data between blocks.

So far so good, but what about proper variables, not temporaries. In the code example also the proper variable a is assigned to more than once. Such a "re-definition" would come from situations, where at source code level, a variable is assigned to more than once in one piece of straight-like code, and the code generator dutifully generates code that does the anologous thing at (intermediate) code level.

But it is straightforward to obtain intermediate code that avoids that. Instead of reusing a, one simply uses different variable as shown in Listing 10.11. Typically that's done by first generating code without not in single assignment format, which is then translated into the format afterward. That is done by consistently renaming or "re-indexing" variables like a in the example. In the code, we assume that a_0, b_0 and c_0 refer to the versions resp. the values of the three variables coming from *at the beginning*. It's like the *input* values to the basic block.

```
t1 := a0 − b0
t2 := t1 * a0
a1 := t1 * t2
t3 := t1 − c0
a2 := t3 * a1
```

Listing 10.11: 3AIC code example (single assignment)

For basic blocks, one can easily achieve that in one pass. The reason why it's mostly done in a 2-stage manner is, that, while for straight-line code, (static) single assignment is trivial, the generalization to branching code (including code with loops) requires additional insight and tricks. Only then one would even say "the compiler uses SSA as intermediate format".

Terminology aside, the appropriately renamed code leads to the dependence graph of Figure 10.11.

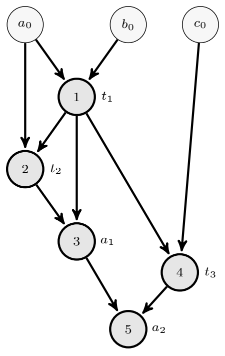


Figure 10.11: DAG for the 3AIC code block

One could get the impression, that's all fine and good, and actually pretty straightforward So, what's the big deal with SSA as intermediate format?

In a way, it's another angle or elaboration of the liveness-analysis, next-use analysis, and dependence or def-use analysis. After transforming a block into single assignment format, the uses of a variable being defined at some point are directly visible from the code: it's all places that mention the particular variable (which must be after the point of definition). Analogously, the **life span** of a variable is directly visible in the code: from the point of being defined until the last mentioning. Inside a basic block, as we *assume* variables to be live at the end, the local live span of each variables starts at the point of their definition and lasts till the end of the block. That should be plausible: if variables are never overwritten because of single-assignment, they never become dead (if we assume them live at the end). Basically, local liveness becomes pretty trivial: for variables "defined" in a local block, they are dead before the (unique) defining line, and live afterwards. For variables defined (or assumed defined) outside the block, like the a_0, b_0, and c_0 in the last single-assignment example, they are live throughout the block.

Also in a more global setting of control-flow graphs, the connection of defs and uses of variables is clear from the names of the variables and temporaries: all other mentionings of a variable defined in one line must be uses. So, the variables carry the def-use information and most of the (global) liveness information in their name.

Of course, nice as it is, it's not for free. As discussed, transforming code into SSA actually *requires* to do basically something like liveness analysis, not in it's binary form we started out with, but more refined, and additionally following a proper renaming scheme. And, as said, for the global level, there are additional complications, beyond straightforwardly generalizing the binary liveness information and some easy renaming, that's where the "real" SSA starts, but we don't go there in this lecture.

To have SSA as a format that makes liveness analysis quite simple does not in itself breathtakingly useful. After all, one could do liveness analysis straightforwardly without doing that intermediate format (which, as said, gives additional challenges to overcome). The importance and popularity of SSA comes from the following: the format does **not only** help for live variable analysis. As hinted at, there are many data flow analyses one might want to do, which serve different purposes and using similar techniques as liveness analysis, and many of them similarly profit from that format. So the effort invested **to transform the code into SSA may pay off multiple times**, in case the compiler employs multiple analyses, not just liveness analysis, on the given level of abstraction.

10.7 Global analysis

We have discussed general ideas behind liveness analysis earlier and covered local liveness in Sections 10.4 and 10.5. Additionally, in Section 10.6, we discussed closely related concepts, like next-use analysis, def-use analysis and the concept of dependence graphs. We focussed mostly on the local level, i.e., on basic blocks, only hinting at that some things get more involved when doing analysis for a whole control-flow graphs. Here we fill in a few more details on this, without also covering extensions like dependence analysis again.

Going from block-local analysis to an analysis of a whole CFG, there are basically 2 complications: **branching** of the control flow and **cycles**. Both originate from control-flow constructs in the source code, like conditionals and loops.

Branching is conceptually the simpler problem, though if one has loops in the source language one invariably also has to face branching. On the 3A(I)C level, there are no loops, there are just jumps and conditional jumps. Conditional jumps obviously lead to branching, the true-case and the fall-through alternative, so a block with a conditional jump at the end has two successor blocks or successor nodes in the control-flow graph- But also unconditional jumps may involve branching: the block jumped to, which starts with the corresponding jump label, may be entered also otherwise, for instance with by a fall-through or being the jump target from different sites. though in a "backward" manner: the node starting with a jump label may have more then one predecessor node in the control-flow graph.

Let's start with branching and discuss it for liveness of variables. Conceptually, with branching, one does no longer try to find out exactly if a variable, at a point *is* live or not. We mentioned earlier that **approximation** is characteristic and crucial for all kinds of semantic analyses, and in particular for data-flow analyses, like live variable analysis.

Dynamically, i.e., at run-time, at one given point in a program execution, the liveness information is still binary: a variable will be used in the future, or it will be not. At least it’s binary when we are dealing with *deterministic* programs or with a given run or trace of a program. Whether or not a variable (or address) will be used in the future at a given point in the executing is the question of **dynamic liveness**. That’s of course undecidable in general and that’s not what data-flow analysis aims at.

It gives an approximative answer, using the control-flow graph as level of abstraction. As far as branching is concered, data-flow does not try to find out which branch is taken. Not knowing which branch is actually taken, the approximation explores both, combining the information. Indeed, a running program will often explore both branches, though at each given point in time, the program either turns left or else turns right. That’s the common behavior for loops: for some number of iterations, the body is entered, and when the termination criterion finally is satified, the exit-edge is followed (unless the loop runs forever or when never entered).

For static liveness, we have to keep in mind what we need the information for. The intention is to support register allocation. In particular the code generator will consider a register containing a dead variable’s value as “free” and reusable for other values. That means, if we mistake a live variable for being dead, that will lead to erroneous code; the compiler is incorrect. The opposite mistake, rating an actually dead variable to be live may lead to a missed oportunity of reusing a corresponding register, but that’s not an error. The code may just been slower than it could have been without making that misjudgment.

For live variable analysis it means, when in doubt count a variable as live. We did the same at the local level, when judging variables live at the end of a block (since locally one does not know what actually is the case). Same principle here: when facing a situation in a graph with two alternatives, one where the variables is used in the future, further down the graph, and another where it’s not used, the variable needs to be rated as live.

So, the approximation for liveness is about whether the variable **may** be used in the future, not that it’s guaranteed that it will be used (**must**). There are different data flow analyses that work with a must-kind of approximation instead of may-type. It’s the difference between over-approximation and under-approximation. Which one is the right choice depends on what the compiler does with this information, the intended usage. For live variable analysis used for register allocation, it has to be over-approximative (may be live). We don’t look at other data flow analyses for other purposes, so we don’t cover in the lecture must analyses, though if one knows how to do a may analysis like liveness, it’s straightforward to do a must analysis.

One may even flip the live-variable analysis around and formulate it as must-analysis. If we consider that what we are after as a **dead-variable analysis** instead of a live-variable analysis, and looking for situations where variables are dead, then, what we need are situations when it’s *guaranteed* that the variable is dead. In that sense, it’s a matter of perspective.

Now, approximation of relevant information, like “may the variable be used in the future” is the way to deal with **branching.** As described earlier, for straight-line code, the way the liveness algorithm propagates this information is **backwards**. That will also be done for the global liveness analysis, i.e., the information flows in the reverse direction of the edges of control-flow graph: from suceesor nodes to predecessors.

Now, what about **cycles** in a graph? That is a different problem and makes the problem harder. That refers the complexity of the problem, but also the required theoretical background. We don't go into the latter here, we just hint at why it's harder and what to do about it, without explainging why it actually works, resp. under which circumstances it works. For us it's enough to know that for live variable analysis the sketched approach does indeed work.

As explained, the local liveness analysis "walks" through the code in a single pass, namely in a backward manner. The same can be done if one had a branching structure without loop, like for instance in the CFG from Figure 10.7. The only thing that is mildly more complex compared to local analysis is that one has to treat the liveness information approximative. In the graph of that figure, that would concern the treatment of node 2, where the three "flows" coming from below merge.

In the presences of cycles, one cannot expect to propagate the information one time along each edge and through each node and be done. Information that propagates through the graph, say, in tendency "upwards" in a picture, will in a loop also be propagaged back down again to a place already explored.

In that way the information, for instance about liveness status of variables, **circulates** through the graph, sometimes "going" through a cycle multiple times.

That directly makes that task computationally more complex than the single-pass approach that suffices to deal with acyclic structures. Also termination of the data-flow analysis may be of concern, though actually for live variable analysis in our setting (and similar data flow analysis) termination is guaranteed. It's only not 100% immediate as for acyclic structures, where the analysis stops after having treated evey line or node exactly once.

The characteristics of the core data flow algorithm (for liveness and others) is captured by fixing 3 aspects:

Initialization: firstly, with wich (liveness) information should the data flow start? **Loop:** secondly, how to repeatedly propagate the (liveness) information during the analysis? There will be a loop whose body "lets the data flow", by propagating it through the control-flow graph. **Termination:** Finally, when to terminate, i.e., exit the loop.

These three aspects shape the general skeleton of the data flow algorithm. There will be an initialization phase, a loop, and the loop has an exit condition.

Actually, already the simpler liveness analysis for basic blocks from Listing 10.7 is of that shape. One difference to the global setting is that now the termination condition is more complex. The straight-line code version simply stops after having treated the first line in its single backward pass through the code. The second difference is that the traversal for the graph is not so rigid, like backwards. In the presence of loops, when following the edges, there is no single possible plausible strategy, and one generally has to treat the nodes of the cycle multiple times anyway.

So far the general picture of how the algorithm is shaped. We won't give pseudo-code for it, we mainly explain by way of examples how it works. But still we fill in some details of the skeleton, sketching what actually is done during initialization, what step(s) are iterated, and when actually to stop.

We discuss it without making a difference between temporaries and variables, which are treated analogously. That's different than what we did in the local anlysis, hich assumed temporaries dead at the end of the block. We don't need to assume that here, like we don't need to assume that variables are live at the end of a block. With the control-flow graph at hand, the global analysis (approximatively) figures out which variables are statically live and which not, and for the temporaries it will figure out that indeed they are all dead (for temporaries generated the way described). No need to "assume" anything. At the very end of the whole program, all variables are dead for sure, proper variables and temporaries alike.

initialisation: minimal information all variables are assumed dead.

increase repeatedly the body of the loop treats one element of the graph, like one node or one edge and updates the current liveness information. The update works in a **monotone, increasing** manner: a variable previously still considered dead is flipped to be rated live, but never in the other direction.

termination by stabilization when no more information can be added, so no more live variables at some places are detected, the algorithm stops.

We can picture it as follow: the algorithms starts with *no knowledge* about liveness status, and considers for a start all variables dead at all places. The same was done for the liveness analysis for basic blocks from Listing 10.7 (except that for the last line, the proper variables were assumed live). We can see that starting point as absolute *minimal* information and discovering more variables at more places live during the analysis can be considered to *increase* the information. It's not so much that the sets (of live variables) grow larger (though they do). It's more that the amount of confirmed information about the liveness status at different places increases. If, at some place, the current status in the algo of a variable switches from dead to live means that the algo has explored parts of the graph deep enough and confirmed that there is a potential path to a future use of the variables. Once established, further exploration may find further statically live variables at other places, but the liveness information established and added right now never has to be reversed back by new information discovery later. In that sense, the liveness information steadily grows and never shrinks, i.e., monotonously increases during the iteration of the algo. This is crucial and **characteristic** for data flow analyses.

It also makes clear when to stop, namely when the information cannot be increased any more. The algorithm reaches stabilization or it saturates or a "closure" or a fix-point.

Indeed, one finds this "monotonously adding information until stabilization" idea not just for data flow analyses. For instance, the first- and follow-set calculations worked similarly. Also there, one basically explored a graph, namely the way the non-terminals of the grammars hang together. Since context-free grammars use recursive definitions, the corresponding "graph" contains cycles (we never drew the grammars as graphs, we noted them in BNF...) So, for instance, the set of terminals (or ϵ) confirmed to be in the first-set of various non-terminals increases until no more such information is added, at which point the first-set saturation procedure terminates.

10.7.1 Basic blocks and live-in and live-out

Before presenting the liveness mainly by way of examples, we explain a practical aspect, namely the *efficient treatment of basic blocks*. We mentioned that in passing earlier. For **efficiency**, it's best to analyse basic blocks, the nodes of the control flow graph first and "summarize" the corresponding information. That will allow the global analysis to treat them effectively as if they were single lines.

This summarizing local analysis is not identical to the local liveness information we did, but it works the same (like steping backwards through the lines). Here, it's not (only) about whether a variable is (assumed or factually) live of dead in the different lines of the basic block. For some variables one cannot determine that locally (for some one can). It's about **changes** to the liveness status of all the variables.

It's much like looking to one line as in the local analysis. Consider one single line of the form $x_1 := x_2 \ op \ x_3$ as they were treated in the local analysis and let's assume all three variables are all different). For that line, it's clear that at the beginning of the line, x_1 is dead and x_2 and x_3 are live. Considering that as a change of information from the situation after the line to that before that line, it's as following. The situation for x_2 and x_3 will be set to live, and x_1 to dead, independent from how it there status is (currently) afterwards. For all other variables, the information from after the line is left unchanged.

Basically, one can do the same for basic blocks, only more than just 3 variables may change their status. We will not show code how do calculate that, it's easy enough.

At any rate, relevant for the global analysis is not what happens line by line in the basic blocks, relevant is only the situation right in front of each basic block, resp. right afterwards. This is also called *inLive* (in front) and *outLive* (afterward). What the global analysis needs to know how the *inLive* and *outLive* information hang together. Doing a backward analysis, in particular how, for a given block, the *inLive* information is calculated for a given *outLive* information.

And therein lies the improvement: having **precomputed** the block-local effect on the flow of liveness information per block, the global analysis can just use that, and avoid stepping through the individual lines of the bloks over and over. When occuring as part of a cycles in a graph, blocks will have to be evaluated more than once in general, and it's to be expected that the precomputation will make the analysis more efficient. Blocks not occurring in a loop would not profit from a pre-computation.

Anyway, pre-computing the effect or not will not influence the result of the analysis, only the running time.

In the examples later, we don't precompute anything explicitly, the figures illustrated the *inLive* and *outLive* information and whether this is the result of being smart and having precomputed some bits or whether one does it over and over again, is not visible, and the outcome, as said, is the same anyway.

At the end of the algorithm, *inLive* and *outLive* contain (in an approximative manner) sets of live variables[8] at the *beginning* and *end* per basic block.

The basic step when calculating that is how the different *inLive* and *outLive* information depend on each other.

> *inLive* of a block depends on the *outLive* of that block and on the straight-line code inside that block. That dependence corresponds to the **transfer function** of the block. The *outLive* of a block depends on *inLive* of the *successor* blocks. It about the dependence of the situation of the successors (not the other way around), because liveness analysis is a *backward* analysis.

Calculating the information approximatively has the goal **to err on the safe side**. judging a variable (statically) live is always *safe*. Judging wrongly a variable *dead* which actually will be used), that is **unsafe**.

> The goal is to calculate the **smallest** (but still **safe**) possible sets for *outLive* (and *inLive*)

Example 10.7.1 (Factorial CFG). Let's revisit the control-flow graph from Example 10.3.1 resp. Figure 10.3, the factorial function. Figure 10.12 here shows the CFG again, but with highlighting the *inLive* and *outLive* in the graph, at least the places where*inLive* and *outLive*.

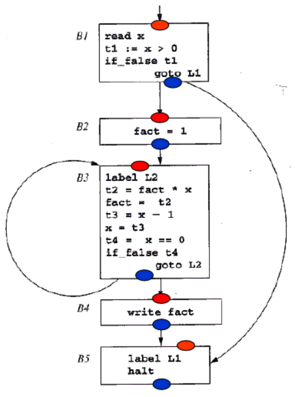


Figure 10.12: CFG and *inLive* and *outLive*

The picture shows the arrows from nodes to successor node. Since we are doing a backward analysis, the analysis need to follow those arrows in reverse. If we consider an implementation of graphs where edges between nodes are represented as pointer or references, the graph should have pointers in that directions. Indeed, if one implements a CFG as intermediate representation for instance with pointers, a doubly linked structure would be useful (with pointer "forward" as well as "backward". This way one can efficiently perform forward as well as backward style analysis on the representation.

[8]To stress "approximation": *inLive* and *outLive* contain sets of *statically* live variables. If those are dynamically live or not is undecidable.

node/block	predecessors
B_1	$\emptyset$
B_2	$\{B_1\}$
B_3	$\{B_2, B_3\}$
B_4	$\{B_3\}$
B_5	$\{B_1, B_4\}$

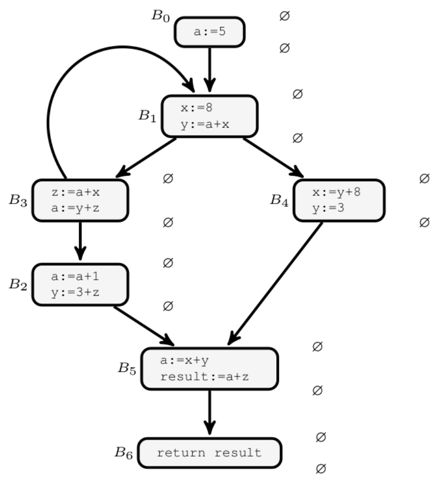
□

Example 10.7.2. Let's use the code from Listing 10.12 as example.

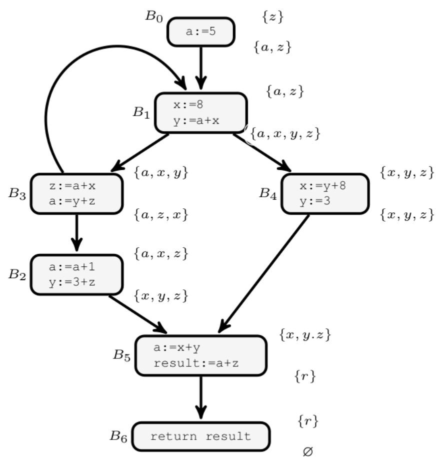
```
    a := 5
L1: x := 8
    y := a + x
    if_true x=0 goto L4
    z := a + x          // B3
    a := y + z
    if_false a=0  goto L1
    a := a + 1          // B2
    y := 3 + x
L5  a := x + y
    result := a + z
    return result        // B6
L4: a := y + 8
    y := 3
    goto L5
```

Listing 10.12: Sample code for global liveness

The slides show the working of the algorithm more "dynamically" with overlays. Here, we just show two particular situations: The one after initialization and the one at the end (Figures 10.13a and 10.13b) but not intermediate stages.



(a) Initialization



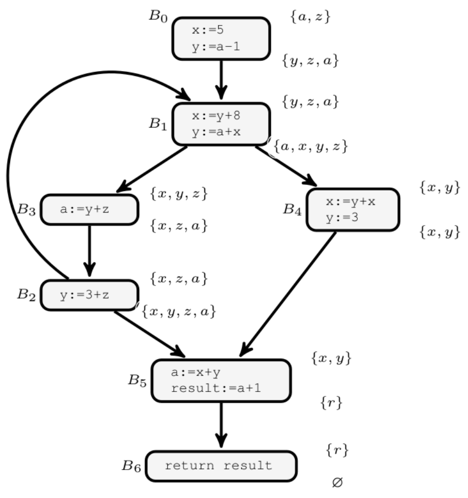
(b) liveness: Result of the run

Figure 10.13: Initialization

At the beginning, *inLive* and *outLive* are *initialized* to $\emptyset$ everywere, i.e, we start with a *unsafe* estimation.

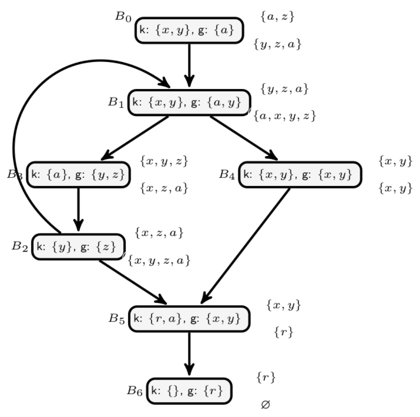
The overlays show how the information "flows" through the graph. The shown traversal strategy is (cleverly) backwards. The example contains a loop, but actually with chosen strategy, it terminates quickly. □

Example 10.7.3 (Another, more interesting, example). See the following graph



Again the slides will show in overlays, how the information flows through the graph. This time the loop is treated multiple times, resp. that some of the information is increased more than once before it stabilizes. □

Example 10.7.4 (Precompuation: kill and genenerate).



10.8 Code generation algorithm

Finally, we cover the code generation proper. We focus on generating code for straight-line code and register allocation. Indeed, translating jumps and conditional jumps from 3AIC is not very complicated. The intermediate code is already a linear code form, and one can expect that the (conditional) jumps have a direct correspondence in machine code. Ultimately, one will have to get rid of the labels. They serve in the intermediate code as symbolic addresses, and need to be replaced by real addresses, in a first stage say, *relocatable* addresses. That involves calculating with the actual byte sizes of the commands for the instruction set of the target platform. For instance like the size shown in Figure 10.1 earlier.

Properly accounting with the concrete instruction sizes to obtain concrete (relocatable) addresses has nothing to do with register usage and is an indepedent problem. As said, we focus on register allocation for basic blocks, making use of liveness information. This focus does not mean that the code generator uses only local liveness information. Also liveness analysis is an independent problem, and the code generator will be correct as long as the liveness information is correct, i.e., a safe over-approximation of the actual future use of variables.

The code generation will proceed line by line through the 3AIC, and it will **forwardly**. The generator will make decisions concerning register usage **on the fly**, i.e., while generating the code. To do so, it obviously needs to keep track of which registers are currently in use and for which variables (resp. which registers will be in use at the different points in the code once the program will be run, of course.) To do that kind of book-keeping, the code generator will maintain specific data structures. They are called **register descriptors** and **address descriptors**. This way the code generator has an overview which variable is stored in which register(s) (if any) and at which address it resides in main memory. That's the address descriptors. The register descriptor information records at each point for each register, which "variable(s)" it contains if any. More precisely, to which value of which variable(s) the current register content corresponds, if any.

In principle, the address descriptor information would be enough: with that information for all variables, the code generator could figure out the register usage by searching through the corresponding table, in some form of *reverse look-up*. That is of course inefficiecent, and the code generate is better off keeping track of the relevant information in two tables.

Aside: Graph coloring register allocation

As said, the code generator generates 2AC instructions **on-the-fly** including making decisions on which registers to use. That's not the only way one can do register allocation. A well-known and widely used approach is known as register allocation by **graph coloring**. We don't cover that, but since it's a standard approach, it's well worth mentioning. The idea is actually simple and elegant: first get an overview over the *live-spans* of variables. Live spans are particular simple for code in SSA format, insofar a particular variable becomes live at some point, stays live for some while, and then becomes dead. For non-SSA usage, a variable may switch from dead to live and back multiple times.

Be it as it may, knowing the live spans is important insofar: two variables with an *overlapping* live time cannot occupy the same register, they need to be in different ones. Of course one could try to keep a variable in one register for some time, kick it out from the register for a short while and store it back to main memory, to make room for another one for a short while, perhaps there is only as short period of overlap, and then, later load it back again into the same register (or a different one). But there is only so much sophistication one can do, and juggling values back and forth between registers and memory is costly at any rate.

So a clear and useful arrangement is the following: if it's decided to place a particular variable in a register, the association is fixed. The variable is put into the designated register at the start of its live span, it's keep there it there during the live span, and will be saved back to main memory at the end of the live span, if the value in the register has changed in the meantime. That is a particularly clear strategy for SSA-style code.

In the sketched strategy, having an overlap in live times has the mentioned consequence: the register allocator has to select two different registers for the two variables.

Variables with an overlap in live spans are said to be **in conflict**. One can represent then the conflict situation via an undirected **graph**. Variables are the nodes of the graph, and conflicts are the edges.

The number of nodes in the graph corresponds to the number of variables we need to take care of. The register allocation task is to color the nodes with registers in such a way that two neighboring nodes don't reside in the same register, i.e. have different colors. Typically, the number of nodes in the graph exceeds the number of available registers, otherwise the problem would be trivial.

This is a particular graph coloring problem, trying to color the graph with the given colors so that no neighboring nodes (representing conflicting variables) carry the same color.[9] The registers correspond to "colors" Solving it is a problem of high computational **complexity**, i.e., finding an answer to the question:

Can the given graph be colored with the given colors (and if so, how)?

Observing the analogy of register allocation and graph coloring is interesting and actually straightforward, and so one could consider the problem solved: color the graph, and that gives the required association from variables to registers. But there is more to it. For a start, register allocation is more than just a binary graph coloring problem. It's not the question "can this program be compiled or not using the given registers". It **has** to be compiled given the registers, and more often than not, there are not enough registers for the given variables, and the graph coloring problem has no solution. Still the compiler cannot refuse to generate code, just because the graph-coloring fails. Furthermore, graph coloring is a **compuationally hard** problem. Trying to find a graph coloring for a given conflict graph may well not be worth the effort, especially given the fact that the attempt will often fail anyway, as there are not enough registers.

[9]Graph theory has studied flavors of graph coloring problems, but that's a very basic and common one.

So graph coloring register allocation does not attempt to solve the exact graph-coloring; that would cost too much time. Instead, it uses a heuristics that does a decent effort without aiming at an exact solution to start with, choosing colors or registers for the variables avoiding conflicts as long as possible. But when no non-conflicting choice is possible for the next node, not attempt is made to try again with a different coloring scheme to see if that turns out to be more successful. Instead, one simply resorts to use the main memory (“**spilling**”) for the node that cannot be colored right now, and then the allocator proceeds with coloring the rest.

Details of the register allocation may become involved for practical languages and platforms, starting already with the fact that some platforms put restrictions on what registers can or have to be used for what, and other complications and fine-tunings. However, the basic idea is elegant and straightforward and hopefully understandable from the high-level description.

Graph coloring register allocation is widely used. The original proposal is described in a software patent (by IBM), one quite early software patent. Not everyone agreed and agrees in which way or to which extent ideas like that can be patented. In this particular case, the graph coloring idea directly employs a recursive strategy described over 100 years ago, tackling in a **heuristic manner** a particular graph coloring problem.

10.8.1 A simple code generation algorithm

After taking a look at a general class of register allocation algorithms based on graph coloring, we go back to our code generation algorithms, which will employ a rather simple register allocation strategy in comparison.

Some make a distinction between register **allocation**: “should the data be held in register (and how long)” vs. register **assignment**: “which of the available registers to use for that”, but the distinction is not central for us.

Limitations

There are **limitations** of the code generation algorithm presented later. One is that we discuss only local **intra block** code generation. It’s not so much that liveness analysis is block-local only. In fact, as mentioned, code generation in general is independent from liveness analysis: code generation is correct as long as liveness analysis is a convervative approximation of liveness analysis, and investing in more precise live information, doing a more global analysis, could result in better code. However, the code generation here will do the following. At the end of the block, all *variables* kept in registers have to be stored back to main memory (at least those, where the value in memory has become out-of-sync with the register value. For temporaries, that’s not needed; it’s what makes them temporary. Analogously, at the beginning of a block, all general-purpose registers are treated as empty. That means, with this design, even if the code generate would use a global liveness analysis, it would not profit from that and doing local liveness analysis is matching the block local code generation approach.

Also, we omit code generation for more complex data structures, like arrays, pointers, etc.

The way the simple code generator happens to work for one block has consequences for the treatment of *read-only* variables. It turns out that they are never put in registers, even if variable is *repeatedly* read. This is not a "design-goal": it's a not so smart side-effect of the way the algorithm works. Due to its simplicity, the treatment of read-only variables leaves room for improvement.

Also the algorithm works only with the temporaries and variables given and does not come up with new ones or does other fancy things like trying to rearrange the code. for instance (dependency graphs could help). Finally, no attempt is made to take the **semantics** of operations into account, for instance exploiting commitativity like the fact that $a+b$ equals $b+a$. Sometimes knowing that can be exploited for optmimizations and reuse of values.

We decompose the code generation into two (or three) parts, discussed separately:

> The *code generation* itself and, afterwards a procedure called `getreg`, as auxiliary function where to store the result. One can see liveness information as third ingredient.

The liveness information calculated separately in advance (and we have discussed that part already). The code generation, though, goes through the straight-line 3AIC line-by-line and in a **forward** manner, calling repeatedly `getreg` as helper function to determine which register or memory address to use. We start by mentioning the general purpose of the `getreg` function, but postpone the realization for afterwards.

As far as the code generation may is concerned: finally there's no way around the fact that we need to translate 3-address lines of code to 2-address instructions. Since the two-address instructions have one source and the second source is, at the same time, also the destination of the instruction, one operand is "lost". So, in many cases, the code generation need to save one of its 3 arguments in a first step somewhere, to avoid that one operand is really overwritten. We have gotten a taste of that in the simple examples earlier used to illustrate the cost model. The "saving place" for the otherwise lost argument is, at the same time the place where the end result is supposed to be and it's the place determined by `getreg`.

Of course, there are situations, when the operand does not need to be moved to the "saving place". One is, obviously, when it's already there. The register and address descriptors help in determining a situation like that.

For presentational reasons, we proceed the explanation of the code generation algorithm in stages at different levels of details, first without updating the book-keeping, afterwards keeping the books in sync, and finally, also keeping *liveness information* into account. Still, even the most detailed version hide some details, for instance, if there is more than one location to choose from, which one is actually taken. The same will be the case for the `getreg` function later: some choice-points are left unresolved. It's not a big deal, it's not a question of correctness, it's more a question of how efficient the code (on average) is going to be.

Purpose and “signature” of the *getreg* function

The procedure is one *core* of the code generator, repeatedly called during the code generation process. We present, as mentioned the *getreg* function after the code generation. To understand the code generator, we need, however, an understanding what *getreg* is supposed used to do, without detailing how it does that.

The code generator will step through the 3AIC line by line. Let’s focus on lines of the form $x := y$ **op** z). It needs to make decisions on the spot (that’s one of the limitations), how to use registers for the calculation. Actually, the crucial decision it will do is

where to place the result x?

That’s what the *getreg* function calculates.

Input: TAIC-instruction $x := y$ **op** z

Output: return *location* where x is to be stored

Additionally, it the function can consult the liveness information and the register resp. address descriptors.

The **location** refers to the place where data is being found, where the result of the operation, that is descibed in 3AIC by the right-hand variable z. The locatation will be be a register, if possible, or a memory location.

In the 3AIC lines, x, y, and z can also stand for temporaries. Resp. there’s no difference anyhow, so it does not matter. Temporaries and variables are different, concerning their treatment for (local) liveness, but that information is available via the liveness information. For locations (in the 2AC level), we sometimes use l representing registers or memory addresses.

Register and address descriptors

Besides the auxiliary procedure *getreg* and access to liveness information, another ingredient for code generation are register and address descriptors, mentioned earlier. The code generator has to keep track of register contents addresses for names.

Both are data structures (like look-up tables) playing inverse roles. **register descriptors** keeps book for each register, which variable(s) it contains (if any). More precises, one should say, it keeps track per register, whether the register is free, or it contains a value that corresponds to one (or more) variables in 3AIC. Note we said that a register can contain (the values of) **more than one variable**. That has nothing do with the sizes of data types, like that it could be possible to squeeze to short pieces of data into the register. But when executing a 3AIC instruction of the form $x := y$, and storing the result in a register, that register contains, at least for now, the value for x and y. In that sense, the register represents the two variables, and the register descriptor has to keep track of that, too.

As said: at block entry, assume all registers are unused and the code generator, when steping though the 3AIC code emitting lines of 2AIC, updates the register descriptor (as well as the address descriptor table). Indeed, also the *getreg*-function, that decides on the location of the results consults and updates the tables.

Address descriptors contain the reverse information. It track per variable or temporary the location(s), where its current value. That can be a register, and/or an address in main memory (for instance the stack or some other part of the main memory). There is **more that one** location possible, That's not due to overapproximation: the code generator needs to keept track exactly, it's a question of correctness. But after for instance loading a variable content from main memory to a register, the variable content exists both in main memory and in one particular register (currently in sync).

By saying that the register descriptor is needed to track the content of a register, we don't mean to track the actual *value* (which will only be known at run-time). It's rather keeping track of the following information: the content of the register corresponds to the (current content of the following) variable(s).

Code generation algo for $x := y$ **op** z

We start with a "textual" version first (see Figure 10.14), followed by one using a little more programming/math notation.

1. determine location (preferably register) for result

   ```
   l = getreg( ``x := y op z'')
   ```

2. make sure, that the value of y is in l :
   - consult address descriptor for $y \Rightarrow$ current locations l_y for y
   - choose the best location l_y from those (preferably register)
   - if value of y *not* in l, generate

     ```
     MOV l_y , l
     ```

3. generate

   ```
   OP l_z , l // l_z: a current location of z (prefer reg's)
   ```

   - update address descriptor $[x \mapsto_\cup l]$
   - if l is a reg: update reg descriptor $l \mapsto x$
4. exploit liveness/next use info: update register descriptors

Figure 10.14: Code generation for $x := y$ **op** z

The core of the algo is sketched in pseudo-code notation in Listing 10.13. One can see the general form of the generated code. One 3AIC line is translated into 2 lines of 2AC or, if lucky, in 1 line of 2AC. The sketched code is *non-deterministic*, ignoring how to choose the locations l_z and l_y. The `let` $l_y \in \ldots$ notation is meant as pseudo-code notation for a non-deterministic choice, in this case, for location l_y from some set of possible candidates.

Note the invariant we mentioned: it's guaranteed, that y is stored *somewhere* (at least when still live), so it's guaranteed that there is at least one l_y to pick.

Also details of *getreg* hidden as well as the issue of *book-keeping* ignored, i.e. we the code don't include the name and address descriptor tables. That in particular implies that step 4 from Figure 10.14 is likewise missing.

```
l = getreg(``x:= y op z'')  // target location for x
if $l \notin T_a(y)$ then let $l_y \in T_a(y)$) in emit ("MOV $l_y$, $l$");
let $l_z \in T_a(z)$ in emit ("OP $l_z$,$l$");
```

Listing 10.13: Skeleton code generation algo for $x := y$ **op** z

Also note (again), the order of the argument in 2AC. We save `y` at some location, l. That one is mentioned as second argument in the 2AC. But the second argument, which at the same time is also the destination location may better be thought of as *first* input. For addition, it may not matter much, but for example `SUB b a` corresponds to `a - b` (with the result stored in `a`). Because of that and thhe way, the translation works also makes clear that we save `y` and not `z`.

Let's now add some missing details, including how to **exploit liveness/next use info** to **recycle registers**. See Listing 10.14. Of course register descriptors don't update themselves during code generation, once set (e.g. as $R_0 \mapsto t$), the info stays, unless reset. Thus in step 4 from Figure 10.14, if y and/or z are currently *not live* and are in *registers*, it's necessary to "wipe" the info from the corresponding register descriptors. That's done in the last 2 lines of Listing 10.14.

```
l = getreg("i: x := y op z")   // i for instructions line number/label
if $l \notin T_a(y)$
then let $l_y$ = best ($T_a(y)$)
         in  emit ("MOV $l_y$, $l$")
else skip;
let $l_z$ = best ($T_a(z)$)
in emit ("OP $l_z$,$l$");
$T_a := T_a \backslash (\_ \mapsto l)$;
$T_a := T_a[x \mapsto l]$;
if   $l$ is a register
then $T_r := T_r[l \mapsto x]$;

if   $\neg T_{live}[i, y]$ and $T_a(y) = r$ then $T_r := T_r \backslash (r \mapsto y)$
if   $\neg T_{live}[i, z]$ and $T_a(z) = r$ then $T_r := T_r \backslash (r \mapsto z)$
```

Listing 10.14: Code generation algo for $x := y$ **op** z, using liveness

Side remark 10.8.1 (Updating the address descriptor table for dead variables?)**.** We said, the code generator should consult the liveness information to update the register descriptors. If variables kept in registers are dead, the corresponding register is afterwards probably free (unless it contains also the value of at least one non-dead variable). We also said, the register descriptor table and address descriptor table contain more or less same information, but in "reverse" form. So one could ask: if one wipes the information about dead variables from the register descriptors, why not also from the address descriptors? One could do so, to keep both tables consistent, for instance satisfying

$$r \in T_a[x] \Leftrightarrow x \in T_r[r] \ .$$

However, wiping the register information form the address table for dead variable has no practical purpose, it won't make a difference if y and/or z are not-live anyhow as their address descriptor wont' be consulted further in the block. Wiping the information from the register descriptor tables does of course have a practical purpose: free registers can be recycled, that's the whole purpose of the liveness analysis. □

In the pseudo-code we make use of some math-like notation. We write T_a and T_r for the two tables. They may be implemented as arrays or look-up structures. For updating we use notations like $T_a[x \mapsto l]$. This is meant to say: after the update, x is stored in l, the old information overwritten. Variables can be stored in different locations, but updating x in such an assignment *invalidates* all other locations, they become **out-of-date** or **stale** or **out-of-sync**. The *only* place where x resides in l. By $T_a \backslash (\_ \mapsto l)$ we mean, we *remove* bindings, namely all that mention l.

Since there are situations, where one location can contain (the content of) more than variable, one may also have to suppert operations like $T_r[l \mapsto_{\cup} x]$, meaning that old information (here for l) is not overritten, but another "binding" is added: after the update, location l contains *also* (the value) of x, without forgetting the old values. This is not needed in the translation of our 3AIC instruction, but would occur when translating $x := y$ for instance, i.e., copying values.

We could also check whether x is live and do the corresponding wiping for x as well. In which case, the whole assignment is meaningless, and (as a consequence, also the liveness status of y and z could change in turn...).

As an invariant, a variable never resides in more than one register.

Determining the result location (the auxiliary function *getreg*)

We have postponed filling in details how *getreg* works. The purpose, as discussed earlier and as visible in the code generator is the following: given a 3AIC line $x := y$ **op** z, return a good location for the target variable x. Basically, starting from the cheapest possibilities, preferring registers, to move to more costly ones, if necessary. More concretely, do the following steps, in that order

1. **in place:** if x is in a register already (and if that's fine otherwise), then return the register
2. **new register:** if there's an unused register: return that
3. **purge filled register:** choose more or less cleverly a filled register and save its content, if needed, and return that register
4. **use main memory:** if all else fails

Figure 10.15 shows the procedure in a but more detail.

Not all details are filled in. For instance, it's not said which register should be purged. One could use some heuristics. One possibility is is least-urgently used, i.e., one whose

```
1. if
    • $y$ in register $R$, and
    • $R$ holds no alternative names
    • $y$ is not live and has no next use after the 3AIC instruction
    ⇒ return $R$
2. else: if there is an empty register $R'$: return $R'$
3. else: if
    • $x$ has a next use [or operator requires a register] ⇒
        – find an occupied register $R$
        – store $R$ into $M$ if needed (MOV R, M))
        – don't forget to update $M$ 's address descriptor, if needed
        – return $R$
4. else: $x$ not used in the block or no suituable occupied register can be found
    • return $x$ as location $l$
```

Figure 10.15: *getreg* algo: $x := y$ **op** z in more details

next-use is furthest away in the future. For step 3 one must not forget (i.e., the code generator must not forget): registers may contain values for > 1 variable. That means the step may involve *multiple* lines with MOV's.

We have not ingredients in place for code generation in basic block (with the limitations as mentioned earlier). All ingredience, except details like which registers will be chosen for purging, but one can for simplistity use a random strategy: if, for instance, one has no heuristics or other strategy at hand, that makes a clever choice which register to purge, one can assume it's picked at random. That may not be the most clever strategy, but the correctness of the code generation does not depend on it.

Example 10.8.2 (Code generation). Let's consider the following assignment in source code

$$\mathtt{d} := (\mathtt{a} - \mathtt{b}) + (\mathtt{a} - \mathtt{c}) + (\mathtt{a} - \mathtt{c}) \tag{10.4}$$

The corresponding three-address intermediate code is shown in Listing 10.15.[10] The table on the right hand side collects the liveness information, resp. the next-use information for the variables.

```
t := a − b
u := a − c
v := t + u
d := v + u
```

Listing 10.15: 3AIC

line	a	b	c	d	t	u	v
[0]	$L(1)$	$L(1)$	$L(2)$	D	D	D	D
1	$L(2)$	$L(\perp)$	$L(2)$	D	$L(3)$	D	D
2	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	$L(3)$	$L(3)$	D
3	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	D	$L(4)$	$L(4)$
4	$L(\perp)$	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	D	D

Let's assume we have 2 registers only. Then Table 10.16 shows two address code produced by the code generator.

[10]The attentive reader will see that the code is a result of a compilation where the source-code addition is treated *left-associative* by the parser, though that's not really important for the example here concerned with code generation.

	3AIC	2AC	reg. descr.		addr. descriptor						
			R_0	R_1	a	b	c	d	t	u	v
[0]			$\perp$	$\perp$	a	b	c	d	t	u	v
1	t := a − b	**MOV** a, R0	$[a]$		$[R_0]$						
		SUB b, R0	t		$\cancel{R_0}$				R_0		
2	u := a − c	**MOV** a, R1	·	$[a]$	$[R_0]$						
		SUB c, R1		u	$\cancel{R_0}$					R_1	
3	v := t + u	**ADD** R1, R0	v	·					$\cancel{R_0}$		R_0
4	d := v + u	**ADD** R1, R0	d					R_0			$\cancel{R_0}$
		MOV R0, d									
			R_i: unused		all var's in "home position"						

Figure 10.16: Code generation example

The code is given in the 3rd column. The table shows not all information in each slot concerning the address descriptors and the register descriptors, only focusing on changes. That means, empty slots represent the content is unchanged from the previous line. The crossed-out slots means that he information is wiped out in the book-keeping. For instance, after line 3: t is dead **dead**, currently m t resides in R_0 (and nothing else in R_0), which means one can reuse **reuse** R_0 (and the example does so). The fact that R_0 is free at that point is marked by $\cancel{R_0}$. Some of the entries are marked in [brackets]. The way the code generator works is that it has to move one of the arguments into the target location. In a 2-line 2AC translation of one 3AIC line, that done in the first of the two instructions. The second one **overwrites** that immedeatly for the result. For instance in line 1, we write $[a]$ as content for R_0 to mean, a is parked in R_0 for just one instruction, but is immediately overwritten in the next line. □

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools*. Pearson,Addison-Wesley, second edition.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

[3] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

[4] Massalin, H. (1987). Superoptimizer — a look at the smallest program. In *Proceedings of the Second Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 122–126.

Index

address descriptor, 2
analysis
    global and local, 5
    inter-procedural, 22
    intra-procedural, 22

backward analysis, 5
basic block, 2, 17, 19

C++, 10
cache, 13
code generation, 1
complexity, 10
computer architecture, 1
control-flow graph, 17
cost model, 2, 11, 13
cross-compilation, 10

DAG, 38
data flow
    backward, 31
    forward, 31
data flow analysis
    forward and backward, 5
def-use, 38
def-use analysis, 38
depdendence graph, 33
dependence graph, 37, 38
directed acyclic graph, 38

efficiency, 9

forward analysis, 5

garbarge collector, 14

hardware architecture, 1

instruction scheduling, 38
inta-procedural analysis, 22
inter-procedural analysis, 22

leader, 19
live variable, 4
liveness analysis, 4
    local, 8
loop discovery, 22

may analysis, 5
may-analysis, 5
memory hierarchy, 13
must analysis, 5
must-analysis, 5

optimization, 2

partial order, 38
partitioning, 18

register, 13
    allocation vs. assignment, 55
    free and occupied, 6
register allocation, 2, 7
register descriptor, 2

static liveness, 31
static single assignment, 42
super-optimization, 2, 3
symbolic execution, 3

tractable, 10

uninitalized variable, 5