PLD COMP
Engineering a compiler back-end in 16 hours

Florent de Dinechin

Contents

CHAPTER I

The big picture and the data structures

Our input language is a subset of C. Our output language is assembly code. We want the latter to faithfully represent the former.

I.1 Syntax versus semantics

I.1.1 Definition

From Wikipedia: *The semantics of a language describes the processes a computer follows when executing a program in that specific language. This can be shown by describing the relationship between the input and output of a program, or an explanation of how the program will execute on a certain platform.*

In short: the semantics of a language describes the meaning of syntactically valid programs.

The semantics should be as formal as possible, but most of the times it provided as textual documents, not mathematics. Example: the C language standard, 2018 version [ISO18].

I.1.2 What is a correct program

First level of correctness is syntactic: a program that is not accepted by the parser cannot be correct.

However, a program can be syntactically correct but have no meaning, for instance

- because it enters an infinite loop,
- or because it computes out of values that are not initialized,
- or ... (insert here the last bug you spent more than one hour on).

The halting problem (see wikipedia if you don't know yet what that means) entails that is impossible to automatically check/verify the semantics of arbitrary programs (whereas it is possible to automatically check the syntax). Still, a formal semantics enables to formally prove, for instance, that two programs are equivalent. This is exactly what we want for a compiler: we want the output program to be equivalent to the input program.

I.1.3 Operational semantics

The semantics of imperative languages is operational: *the semantics of a program is a function that takes the universe in a certain state and transforms it into a new state*.

$$[\![P]\!] : \text{States} \to \text{States}$$

$$[\![P]\!](S) = S'$$

For programs with no input/output, the state is simply the state of the computer, including the content of its memory (hence what appears on its screen).

For instance, the semantics of the program `x=1;` is a function that takes a state of the universes S and returns a state S' that is it almost identical to S, except for the value of variable `x` which is 1 in S'.

Unfortunately, interesting programs have inputs (your joystick) and perform some output (your screen), and then the definition of the universe gets a bit more complicated.

Actually, what is complicated is the definition of a semantics that abstracts away irrelevant parts of the universe, such as you the user.

In this overview, we will limit our universe to the computer state.

I.1.4 Composition

A semantics is compositional if the semantics of a compound program is a (simple) function of the semantics of the parts.

For instance, sequential composition is compositional: for two programs P and Q that each have a semantics, I can write a new program by writing P then Q, and its semantics is of course:

$$[\![P;Q]\!](S) = [\![Q]\!]([\![P]\!](S))$$

Therefore,

$$[\![P;Q]\!] = [\![Q]\!] \circ [\![P]\!] \quad .$$

The semantics of

$$\text{while}(e)\{P\}$$

is also a function of the semantics of e (an expression) and the semantics of P (a program). There is a formula defining this semantics, but it is beyond the scope of this course (it involves a *fix point operator*).

I.1.5 Other semantics

Denotational semantics describes the meaning of a procedure as a mathematical function. For instance we like to think that the meaning of the C function
`double square (double x) { return x*x; }`
is the mathematical function $x \mapsto x^2$.

Denotational semantics is natural for functional programs. It is also naturally compositional. It becomes clumsy for programs that can perform side effects (writing to a file or a screen, etc).

It is also the natural semantics of *expressions*, even in imperative languages.

I.1.6 Memory-based semantics in C and his family

To illustrate how semantics differs from syntax, we will briefly overview the semantics of an assignation such as `delta=b*b-4*a*c;`

Here is a short but fundamental bit of the C semantics: **A variable in C represents a memory location.** For instance, in C, you can always recover the address of a variable with the & operator.

Therefore, in an assignation

- all the variables represent memory locations
- the right-hand-side (RHS) expression is evaluated to a value (sometimes called *rvalue*),
- the left-hand-side (LHS) expression is evaluated to an address (sometimes called *lvalue*),
- and the = operator stores the rvalue at the lvalue.

It is important to insist on the fact that **the semantics of the LHS is an address, while the semantics of the RHS is a value**.

Different semantics for the same syntax Let us now take two examples.

```
a = a +1;
```

Here the `a` on the left represents the address of the variable `a`, while the `a` on the right represents the content of this address. Same syntax, different semantics!

```
a[i] = a[i] + 1;
```

Again, the expression `a[i]` on the left represents the address of the i-th element of array `a`. Again, the same expression on the right represents the content of this address. However there is a subtlety: the symbols `a` and `i` both represent variables. The value of `i` is an integer, but the value of `a` is an adress. The address of `a[i]` is defined as: value(a) + value(i) $\times$ sizeof(type(a)).

Therefore, to compute the *address* on the LHS, we will need to evaluate an expression combining *values* of variables.

This is why we mentioned *left-hand side expressions* above.

I.1.7 Semantics and optimization

A variable may be kept in a register as an optimization "if nobody will notice". But this is an optimization!

And (in the real C) it is non-trivial to ensure that "nobody will notice": with the memory-based semantics, you must be sure that there is no pointer to the variable, for instance.

The formal definition of "nobody will notice" is: *there exists no program to which, according to the semantics*[1]*, it makes a difference*.

[1]Note that the semantics has the notion of "undefined behaviour". For instance, what happens when you attempt to recover the value of an array outside of its declared index range is undefined in C (it raises an exception in Java). So in practice somebody can notice if a variable is kept in a register: it is no longer accessible by an illegal array access. But since this behaviour is undefined anyway, it doesn't matter.

In this project, we will favor **safe semantics** over **dubious optimizations**.

I.1.8 Evaluation order: a case study of semantics-preserving optimisation

The addition operator has two inputs. This is (hopefully) reflected in the addition node of your AST. How should `a+b+c+d` be evaluated?

The C semantics mandates: left to right.

Why does it matter?

arithmetic optimizations

If `a` is an int32_t variable, is it allowed to optimize `1+a+2+a` into `2*a+3`?

Answer is yes:

- The semantics of `int32_t` addition is an addition modulo 2^{32}.
- This addition is associative and commutative
- Therefore "nobody will notice".

Important remark Answer would be no for floating-point types, whose addition is **not** associative.

Side effects

Is it allowed to optimize `f(a)+f(a)` into `2*f(a)`?

Answer: yes only if it can be proven that f has no side effect: consider what happens if

- `f` increments a global variable `x`, or
- `f` prints something to screen

One silly remark: If I understand correctly, in `f(a)+g(a)+h(a)` C mandates that the addition be performed left to right, but does **not** mandate that the order of evaluation is `f(a)` then `g(a)` then `h(a)`.

I.1.9 Implicit type casts

```
uint32_t a,b,c,d;
uint64_t x;
x=a*b+c;
d=a*b+c;
```

Is the + a 32-bit + or a 64-bit one? Does this decision depend on the type of the LHS?

Remark this question can be completely managed by the parser. What is the boundary between syntax and semantics?

Personal opinion: I call it a syntax issue if it is managed by a *grammar*. All the rest is semantics. However, this definition is not perfect. For instance, the storage class of a variable is essentially a syntactic issue (it depends on its lexical scope) with deep semantic implications.

There has been quite a lot of research trying to capture more of the semantics in the grammar: look up "attribute grammars".

Modern compiler framework (ANTLR) also blur this distinction, in the name of pragmatic/efficient compiler design.

I.2 The back-end's input

What the back-end receives from the front-end is a data structure consisting of

- an AST, and
- a table of symbols.

I.2.1 The abstract syntax tree, or AST

... is not my problem, it is Eric's.

I.2.2 The table of symbols

Each symbol is associated to information relevant to this symbol:

- its syntactic class (variable name, type name, function name, etc)
- for a variable name,
  - its type
  - its storage class (related to its scope): a variable may be global, local to a function, an input variable, etc.
- for a function name,
  - its return type
  - the number and types of parameters

The support of named types (structs, unions) is optional in this project. Such named types would also be included in the table of symbols.

The compiler will add information to the table of symbols as compilation progresses. For instance, the table of symbols will eventually include

- for each variable name, its actual memory location
- for each function name, the actual assembly code of this function, along with its entry point (the address in memory to jump to in order to call this function)

```
#include <stdint.h>

int32_t fact(int32_t a)
{
  if(a==1)
    return 1;
  else {
    int32_t b;
    b = a * fact(a-1);
    return b;
  }
}
```

```
Programme {
 DefFun {
   int32
   Ident(fact)
   Param {
     int32
   }
   ListInstr {
     If {
       == {
         Ident(a)
         Const(1)
       }
       Ret {
         Const(1)
       }
       Bloc {
         ListInstr {
           Decl {
             int32
             ListeVar {
               Ident(b)
             }
           }
           Aff {
             Ident(b)
             * {
               Ident(a)
               Call {
                 Ident(fact)
                 Param {
                   - {
                     Ident(a)
                     Const(1)
                   }
                 }
               }
             }
           }
           Ret {
             Ident(b)
           }
         }
       }
     }
   }
  }
}
```

Figure I.1: Exercise: draw the AST whose textual representation is on the right

- ...

It is therefore important to understand that the table of symbols will be a dynamic structure. It is initialized by the front-end, then progressively enriched by the compiler until all the information needed for code generation is there.

A final remark is that the table of symbols will also include *temporary variables* that were not declared in the code (see below).

I.3 The back-end's output

A real assembly language running on a real machine.

Remark: we still rely on the GNU assembler and linker.

- the assembler, because it is too easy for you. Why is it easy? Because assembly code can be compiled line-by-line, without any context information. An assembler essentially iterates a dictionnary on the lines of the assembly program. The only subtlety is the management of *labels* but it is an easy one. *If this is not clear to you, please discuss it with us in lab hours.*
- the linker, because it is too difficult for you (in any case it is for me).

I.3.1 Main target

The documentation of the target assembly language can be found in the 2198 pages of *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z* [int16].

An alternative to reading these 2198 pages is to use `gcc -O0 -S` to observe the generated code, starting with simple programs that just perform one characteristic action.

The `-O0` option is important to prevent the compiler from optimizing anything. *What is expected from you in this project is essentially a -O0 compiler.* An example is given below:

```
int main() {
  for (int i=0; i<17; i++) {
  }

}
```

gives

```
movl $0, -4(%rbp)
jmp .L2
.L3:
addl $1, -4(%rbp)
.L2:
cmpl $16, -4(%rbp)
jle .L3
```

Exercise: try it yourself, starting with an empty program. You will notice that the assembly code above is wrapped within other instructions. These will be explained soon.

Keep the instruction set reference at hand to understand the instructions whose meaning is not obvious at first read.

Remark Each instruction of ISA can also be described in terms of syntax and semantics.

I.3.2 Optional target

ARM code is simpler in the sense that the manual is smaller, but the same remarks apply.

MSP430 code is even simpler, and you know the assembly language already.

I.4 Code generation by AST walk

It is possible to emit assembly code directly in one traversal of the AST. Actually it would be possible to do so directly in the parser: early compilers did so.

We try to give the intuitions here, a more formal algorithm comes in next lecture.

- The declarations are used to reserve the memory for the variables; Technically:
  - The main task is to number all the variables; in other words, each variable is associated an index (stored in the table of symbols).
  - A "memory pool" will be reserved for the function. This is simply a chunk of consecutive memory.
  - The OS will provide to our code a pointer called BP (base pointer) to the beginning of this memory pool
  - The address of each variable will be BP+index.
- Expressions, which are trees, are linearized using a variation of the following depth-first traversal: a function `emitCodeForExpr(node n)` generates code for `n`, and returns the memory location where it has stored its result. It is defined by induction on the node type. For instance, if node `n` is `Add(n1, n2)`, the function is called recursively on `n1`, then on `n2`. Then a few assembly instructions are emitted to add the contents of the two locations and store the result to a new location. This result location is returned.

  In the previous, "location" may be the location of a variable, or temporary locations in the memory pool.
- A C assignation first generates the code for its RHS expressions using the previous function. Then it generates the code that computes the address of the LHS. Finally, an instruction that stores the former into the latter is emitted.
- sequences of C statements are transformed in corresponding concatenations of code blocks;
- `if(condition) then A else B` is transformed as follows:
  - first a code block evaluating the `condition`
  - then a conditional branch to `labelB`
  - then the code of `A`, ended with a (inconditional) jump to an `endif` label
  - then the code of `B`
  - then the `endif` label
- `while(condition) A` is transformed into a similar arrangement of code blocks (with a backward jump).
- Exercise: show how a `for` loop is compiled on the assembly above.

Example (black board): code generation for `a=x-2*y;`

It should be noted that in this naive version, an arbitrary number of memory locations may be allocated when compiling expressions. Consider it a minor problem for now: Once we have finished the evaluation of the expression, all these locations can be recycled. And expressions themselves, in typical code, are not that large.

Remark: after all the program has been processed, the total number of needed memory locations is known. This means that the compiler knows the size of the needed memory pool. It will be exploited in chapter III to allocate this memory.

I.5 The intermediate representation

I.5.1 Motivations

The code generated by the previous algorithm has several inefficiencies:

- intermediate results are stored from a register to a memory location at the end of a recursive call, then read back immediately after. They would be best kept in processor registers. Here is the kind of things you get with `gcc -O0` (this is not x86 code, and destination is on the left)

```
1c0:    59 ef f0 ff     st.w [%a14]-16,%d15
1c4:    19 ef dc ff     ld.w %d15,[%a14]-36
1c8:    59 ef c4 ff     st.w [%a14]-60,%d15
1cc:    19 ef f0 ff     ld.w %d15,[%a14]-16
1d0:    59 ef c0 ff     st.w [%a14]-64,%d15
1d4:    19 ef c4 ff     ld.w %d15,[%a14]-60
1d8:    0b f0 00 48     mov %e4,%d15
```

- registers are not used
- ... see the chapter on optimization.

To fix this, the compiler needs to perform several passes on the code being generated. But then it is better to keep it in a data structure, not to emit it directly to a file.

For his purpose, most compilers define an "intermediate representation" (IR) that is an abstraction of the assembly code.

There are many other motivations for using an IR, the main one being that it will allow **retargetable compilers**: A compiler like GCC supports many input languages (using different parsers), and many target processors (using various IR-to-assembly translators). The architecture of such a retargetable compiler is shown on Figure I.2. If we have n input languages and m target processors, this approach will allow you to obtain nm compilers with just $n + m$ components (parsers and IR-to-assembly).

This assumes that a lot of the work, in particular most optimizations, can be performed on the IR itself, and the IR has to be designed for this purpose. In modern compilers, the bulk of the development effort is in the *Optimizer* box of Figure I.2 (which actually includes tens of optimization steps). Comparatively, writing a parser or a processor back-end is simple.

In terms of project management, one benefit of having an IR is that you can split the work between the TreeWalk algorithm (AST to IR), and the processor backend (IR to actual assembly code). Both steps will be much simpler than a single-step compiler. However, the total amount of work is slightly larger, since you will have to manage the IR.

To sum up, a good IR is close to assembly code, but a few steps back to abstraction. Let us detail this.

I.5.2 Straight-line code in the IR

The IR is composed of elementary instructions just like assembly code. If you remember your computer architecture lectures, elementary instructions in actual processors can have from 0 to 4 operands. The best choice for an IR is a 3-operand instruction format (two sources and one destination):

- It matches well the AST structure, where most arithmetic operations have two operands.
- It matches well modern RISC processors.

The idea is that each instruction should be mapped relatively easily to any target machine:

- sometimes an IR instruction will require a few target instructions
- sometimes several IR instructions can be merged into one target instruction

but in both cases the process is relatively local and relatively straightforward.

The IR instruction set also includes instructions to access memory from/to register.

The IR also abstracts the jumps of the program. This is detailed in next section.

Finally, a fundamental feature of the IR instruction set is that it has an infinite number of registers. Another point of view is that it is an assembly language that can work directly on the variables and temporaries of the program[2] (it is not obvious that it is another point of view...).

Mapping this infinite number of registers to the finite number of registers of the target machine is actually a difficult optimization problem (to be discussed in Chapter IV). However, there also exist simple, non-optimal techniques that can be used in this project.

I.5.3 Control flow graph

The program could be represented in this IR as a single list of IR instructions, just like a `.s` file. However, we have an opportunity to abstract the execution flow of the program as a graph, the *control flow graph*.

In this graph, the nodes are *basic blocks* and the edges represent possible jumps.

[2]The current trend is actually to abstract away completely registers and memory locations: In a 3-operand IR instruction, each source register is best replaced with a pointer to the instruction that produced it. This is the philosophy behind Static Single Assignemnt (SSA). If you are passionate about compiler technology, you should definitely have a look at it. However, we definitely do not suggest that it can be implemented within this project: setting up SSA properly requires a deep understanding of compilers in the first place. Besides, the main benefit of the SSA form is in designing optimisations, therefore you should first wait for Chapter IV.

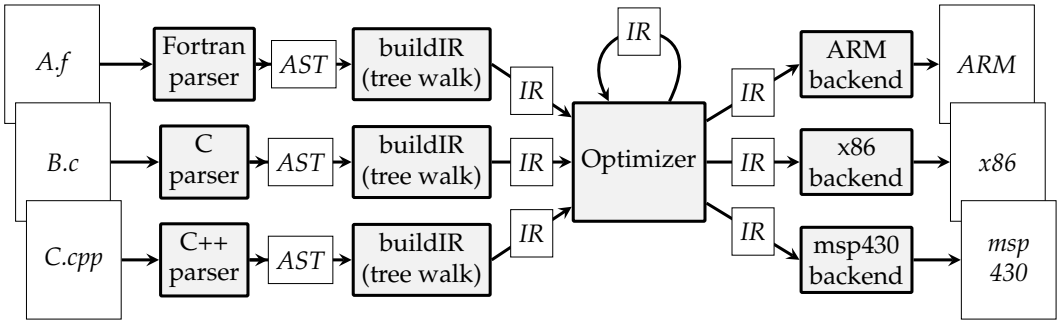


Figure I.2: Architecture of a retargetable, optimizing compiler

Definition: Basic Block A basic block is a sequence of instructions that begins with a label and ends either with one jump, or two conditional jumps (with two opposite conditions). There is no jump destination (no label) inside a basic block. There is no jump inside a basic block, except at the end. In other words, inside the basic block, all the instructions are always executed sequentially.

Definition: Control flow graph The edges, in the control flow graph, are directed and represent possible jumps. There can be an arbitrary number of edges entering a basic block, but at most two exiting edges.

This graph structure clearly separates the control flow and the computations, which will vastly simplify program analysis. It also abstracts away arbitrary code placement decisions (e.g. should the "then" branch come before the "else" branch). The best code placement becomes an optimisation problem that can be decided in due time.

```
int gcd(a,b) {
  while (a!=b) {
    if(a>b)
      a = a-b;
    else
     b = b-a;
  }
  return a
}
```

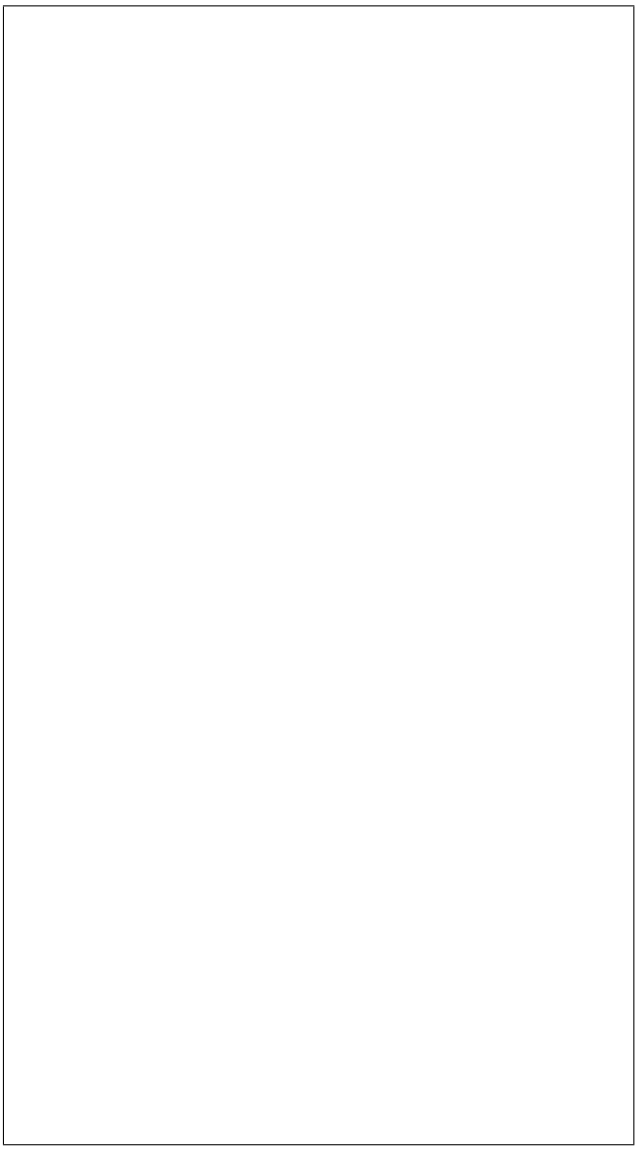


Figure I.3: Exercise: draw the CFG for Euclid's algorithm

I.6 Don't panic: specification of the IR

In this project, we made the choice of providing the IR specification, and it will be introduced in details in the next chapters. However, you are free to improve it (contrary to the other parts of the specifications, which are the C language and the x86 assembly language).

As we have already mentioned, the IR is almost assembly code, but retains some abstraction from the input high-level language.

I.6.1 What looks like assembly code:

- The IR only knows the same base types as a typical processor (int64, int32, int8, fp64, fp32). In our case we will start with int64 only, and support the other ones some day, maybe.

  In particular, *addresses* are integers (int64 for us because our PCs are 64-bit machines). Access to arrays have been transformed into computations on addresses.

- We have one special variable, `!bp`, that holds the base pointer (%rbp on the X86 target). Addresses of local variables are defined by offsets relative to `!bp`.

- IR instructions only have 1 or 2 source operands (except `call`)

I.6.2 What looks like C:

- The IR variables are

  - all the variables declared in the program (with the names they have in C, possibly extended with lexical scope information
  - the base pointer `!bp`,
  - an arbitrary number of temporary variables `!tmp1` to `!tmp`∞ which are created by the compiler.

    There is a method for creating a new temporary variable during IR construction. All these temp variables will be mapped to as many new symbols in the symbol table, and therefore as many memory locations in the activation record. Actually, it is helpful for debugging to number them with the offset they have in the activation record. This is the convention we use in this poly.

    The names of these variables begin with a `!` to avoid conflicts with the names of the C variables, since they live in the same symbol table.

- The `call` instruction has an arbitrary number of parameters, but these parameters are simple variables.

Here is the minimal instruction set to use in this project (see `IR.h`). Here the destination is on the left, and dest, op1, op2 etc are variables.

Mnemonic	Description
`copy` *dest op1*	copy *op1* to *dest*
`ldconst` *dest const*	load a constant in a variable
`add` *dest op1 op2*	binary operation (also `sub`, `mul`)
`cmp_eq` *dest op1 op2*	comparison (also `cmp_lt` and `cmp_le`)
`call` *dest label* (*op1*, *op2*, *op3*...)	subroutine call, *dest* is the returned value
`rmem` *dest addr*	memory read: the content of address *addr* is copied in variable *dest*
`wmem` *addr var*	memory write: the value of variable *var* is written at address *addr*

Remark that the result of a comparison is in a variable, not in flags as in usual processors.

I.6.3 What looks like nothing:

the control flow graph.

- Instructions are grouped in linear sequences called *basic block* or BB. In a BB, all the instructions will execute in sequence.
- The CFG of a function is composed of
  - an input BB (it will generate the prologue)
  - an output BB (it will generate the epilogue)
  - an arbitrary number of other BBs.
- A BB is identified by its label (a branch target), which is the label of the first instruction of the BB.
- A BB may have
  - zero successors in the CFG, in which case it is the output BB,
  - one successor in the CFG, in which case it ends with an unconditional jump to this successor,
  - two successors in the CFG, in which cases it ends with a conditional jump. The successor is selected according to the value computed by the last instruction of the BB (usually, this instruction will be one of the `cmp`s)

Note that the IR doesn't include jump/branch instructions: these are implicitely encoded in the successors of the CFG.

I.6.4 Remarks on the choices made

Compared to the textbook

In the Cooper-Torczon book,

- there is no `call` instruction, and no detail on how calls are supported
- IR has conditional jumps (i.e. is closer to a real processor)

- all the binary operations may have a constant as second operand.
- There is also an addressing mode that allows to specify an address as (reg+constant). We will see soon that it is useful.

This shows that there is a bit of engineering trade-offs in the definition of an IR... just like for the definition of an ISA (see IF3-AO).

In the PLD COMP the choice is deliberately on a minimal IR. The optimization step that fuses several IR instructions into a single machine instruction is well researched.

No flags?

In the IR, conditional execution is captured by BBs with two successors. But most processors use flags for conditional execution (not all! RiscV, for instance, has no flags). Here is a simple way to manage this.

- All the BBs emit an inconditional jump after they have emited the code for their instructions.
- Code generation for a comparison instruction, *if it is the last instruction of the BB*, emits one assembly comparison instruction (which will set the flags), then a conditional jump to the 'false' BB. Otherwise, a comparison instruction is just a normal binary operation.
- A BB with two successors that doesn't end with a cmp must a/ emit the code that tests the boolean value of the result variable of the last instruction, and b/ implement the two corresponding jumps.

I.6.5 The IR data structures in details

See the provided `IR.h`. We have 3 classes there, `IRInstr`, `BasicBlock` and `CFG`.

CHAPTER II

Compiling the body

Here we ignore everything related to function calls: it will be the subject of next chapter. We do have a memory pool reserved for the function. We have a Base Pointer (BP) to the beginning of this memory pool. The front end has assigned a memory location to each variable of the program, in the form of an integer index in this memory pool. If we need temporary variables, we can allocate them as new indices in this memory pool. And we don't worry about keeping variables in registers, or any other optimizations: we first want something that work.

II.1 Big picture of IR generation

The code generation is a recursive walk of the AST of the function body. Each node of the AST has a method `buildIR(CFG* cfg)`.

The pointer to the CFG allows in particular access to

- the symbol table, which belongs to the CFG
- the `string createNewVar()` function that creates a new temorary IR variable, also allocates its place in the memory pool, and returns its name. Of course this grows the memory pool of the function.
- a `currentBB` pointer: this is where generated code goes.
- an `addInstruction()` method that adds an IR instruction at the end of the current basic block
- other useful information that we will introduce in due time

Initially the CFG consists of two basic blocks (entry and exit, responsible respectively of the prologue and epilogue). The `currentBB` pointer points to the entry block. Some day it will be drawn here:

II.2 Compiling expressions

We assume that all the AST expression nodes inherit from an Expr class that provides the virtual method `string buildIR(CFG* cfg)`.

The specification of this method is to

- generates the IR code that evaluates the expression,
- then return the name of the IR temporary variable that holds the result of the evaluation.

We present it as overloaded by the subclasses of Expr, but old-style switch statements work as well in this case.

II.2.1 Constants

```
string ExprConstant::buildIR(CFG* cfg)
{
    string var = createNewVar();
    cfg→addInstruction(ldconst var constant);
    return var;
}
```

II.2.2 Variable as RValue

Since all the local variables are also variables of the IR, this is relatively trivial. We assume that the AST node has a field varName:

```
string ExprVarRvalue::buildIR(CFG* cfg)
{
    return varName;
}
```

II.2.3 Binary operation

Assuming that this node has 3 fields op (which is one of the mnemonics of our IR), leftChild and rightChild:

```
string ExprBinOp::buildIR(CFG* cfg)
{
    string var1 = leftChild→buildIR(cfg);
    string var2 = rightChild→buildIR(cfg);
    string var3 = createNewVar();
    cfg→addInstruction(op var3 var1 var2);
    return var3;
}
```

Remark: If one of var1 or var2 is a temporary created by the compiler (i.e. not a variable/parameter of the initial C), then one could be tempted to save the creation of var3, and recycle this tmp instead.

In general, you may be tempted to attempt to "free" variables once you are sure that they are no longer useful.

It is a false good idea. The good place to perform this optimisation at the level of the full CFG, where it can give much better results. See "register allocation" later in the course.

However, since you will probably not have the time to implement a proper register allocator, you are encouraged to go for trivial and effective optimisations such as the one above.

The simplest one is that all the temporary variables created in a C statement may all be recycled for the next statement. In other words, you may reset the counter used by createNewVar() at the beginning of the buildIR() of a statement.

II.2.4 Variable as LValue

If you remember, we need to get the address of the variable. All the needed information is available in the symbol table through the CFG.

```
string ExprVarLvalue::buildIR(CFG* cfg)
{
    string var = createNewVar();
    int offset = cfg→getOffsetFromSymbolTable(varName);
    cfg→addInstruction(ldconst var offset );
    cfg→addInstruction(add var !bp var);
    return var;
}
```

II.2.5 Function calls

... will be implemented only after going through Chapter III.

II.3 Assignment

Now we have all it takes to compile `delta = b*b+4*a*c;`

We assume the AST node for assignment has two pointers, called lValue and rValue, to the left and right expressions.

```
string ExprAssignment::buildIR(CFG* cfg)
{
    string right = rValue→buildIR(cfg);
    string left = lValue→buildIR(cfg);
    addInstruction( wmem left right );
    return right;
}
```

Remark This returns the rvalue. It is one of the ugly features of C.

Let us spend some time executing all this code on `delta = b*b+4*a*c;`.

One easy optimization The resulting code is quite inefficient. One easy optimization (that you are actually encouraged to implement first, because it will be enough as long as you don't want to manage arrays) is to test if lValue is a ExprVar node. In this case, instead of calling lValue→buildIR(cfg), it is possible to simply emit one instruction: `copy` left right.

But we still need to implement the default code for the case when the lvalue is an array access (or a pointer expression in real C).

II.4 Back-end code generation inside a basic block

Let us now show how simple it is to generate actual assembly code out of the IR. We encourage you to generate comments (as below) in your generated assembly code, for instance expliciting the C variable name corresponding to each offset.

II.4.1 Expression code for x86

C Code	IR	x86		
a+1+b	ldconst !tmp24 1	movq $1,	-24(%rbp)	
	add !tmp32 a !tmp24	movq -16(%rbp),	%rax	# offset of a is -16
		addq -24(%rbp),	%rax	
		movq %rax,	-32(%rbp)	
	add !tmp40 !tmp32 b	movq -32(%rbp),	%rax	
		addq -8(%rbp),	%rax	# offset of b is -8
		movq %rax,	-40(%rbp)	

II.4.2 Simple assignment code when the lvalue is a scalar variable

C Code	IR	x86		
a = exp;	(...)	(...)		# exp val in !tmp64
	copy a !tmp64	movq -64(%rbp),	%rax	
		movq %rax	-16(%rbp)	# offset of a is -16

II.4.3 Generic Lvalue-based assignment code

C Code	IR	x86		
a = exp;	(...)	(...)		# exp val in !tmp64
	ldconst !tmp72 -16	movq $-16,	-72(%rbp)	# offset of a is -16
	add !tmp72 !bp !tmp72	movq %rbp,	%rax	
		addq -72(%rbp),	%rax	
		movq %rax,	-72(%rbp)	
	wmem !tmp72 !tmp64	movq -72(%rbp),	%rax	
		movq -64(%rbp),	%r10	
		movq %r10,	(%rax)	

II.4.4 Straight-line code for a 3-operand RISC (ARM, Power)

... everything is easier (closer to the IR instruction set) and we also need two registers at most.

II.4.5 Management of branches

- A basic block with no successor generates the epilogue
- A basic block with one successor generates an inconditional jump
- A basic block with two successors generate a conditional jump then an inconditional jump. The conditional jump generation may be delegated to the comparison instructions: indeed, in `if(expr1 < expr2)`, the test basic block will end with a comparison instruction.

II.5 If Then Else

Here is the pseudo-code for an If-Then-Else which has three attributes:

- a pointer test to the test expression
- a pointer trueCode to the "then" instruction list
- a pointer falseCode to the "false" instruction list

```
InstructionIF::buildIR(CFG* cfg)
{
    test→buildIR(cfg);                          ▷ returns a variable name but we don't use it
    testBB = cfg→currentBB
    thenBB = new BasicBlock(cfg, trueCode)  ▷ this constructor also generates the code
    elseBB = new BasicBlock(cfg, falseCode)
    afterIfBB = new BasicBlock(cfg)                 ▷ constructor of an empty basic block
    afterIfBB→exitTrue = testBB→exitTrue                                  ▷ pointer stitching
    afterIfBB→exitFalse = testBB→exitFalse                                ▷ pointer stitching
    testBB→exitTrue = thenBB                                              ▷ pointer stitching
    testBB→exitFalse = elseBB                                             ▷ pointer stitching
    thenBB→exitTrue = afterIfBB                                           ▷ pointer stitching
    thenBB→exitFalse = NULL                                           ▷ unconditional exit
    elseBB→exitTrue = afterIfBB                                           ▷ pointer stitching
    elseBB→exitFalse = NULL                                           ▷ unconditional exit
    cfg→currentBB = afterIfBB
}
```

Figure II.1 illustrates this.

Exercise: if the else branch is empty, what should be done?

Figure II.1: Code generation for the if-then-else

II.6 Two ways to compile boolean expressions

Consider `if (expr1 && expr2) { body }`. It could be evaluated in two ways:

- as a binary operation: evaluate expr1, then evaluate expr2, then compute the logical AND.
- as a sequence of tests: evaluate expr1, and only if is true evaluate expr2. In other words, as `if (expr1) { if (expr2) { body } }`
  The advantage is that expr2 is only evaluated when needed. The inconvenient is that there are more BBs and more jumps.

The C semantics actually mandates the second option, probably so that people may write dangerous code such as `if(ptr && ptr->toto=3) ...`.

On the other hand you may get rid of the `&&` and `||` by their conversion to if in the AST.

II.7 While

Once the if-then-else is working in your project (end-to-end, including code generation), you will be able to implement the while in a few minutes, because it does not require anything new.

Here is the pseudo-code for `while(test) { body }`. The AST node has 2 attributes:

- test, a pointer to the test expression
- body, a pointer to the instruction list of the body of the while

```
InstructionWhile::buildIR(CFG* cfg)
{
    beforeWhileBB = cfg→currentBB
    bodyBB = new BasicBlock(cfg)                                    ▷ it is empty
    testBB = new BasicBlock(cfg)                                    ▷ it is empty
    cfg→currentBB = testBB                                          ▷ so we fill it
    test→buildIR(cfg)
    bodyBB = new BasicBlock(cfg, body)      ▷ this constructor also generates the code
    afterWhileBB = new BasicBlock(cfg)          ▷ constructor of an empty basic block
    afterWhileBB→exitTrue = beforeWhileBB→exitTrue                  ▷ pointer stitching
    afterWhileBB→exitFalse = beforeWhileBB→exitFalse                ▷ pointer stitching
    testBB→exitTrue = bodyBB                                        ▷ pointer stitching
    testBB→exitFalse = afterWhileBB                                 ▷ pointer stitching
    bodyBB→exitTrue = testBB                                        ▷ pointer stitching
    bodyBB→exitFalse = NULL                                         ▷ unconditional exit
    cfg→currentBB = afterWhileBB
}
```

Execution of this code is illustrated by Figure II.2.

Figure II.2: Code generation for the While

II.8 For loops

... are left as an exercise.

CHAPTER III

Functions and activation records

In this chapter we show how the compiler is able to allocate the memory pool for each function on a stack.

First, as usual, there is a long reference document:
System V Application Binary Interface, AMD64 Architecture Processor Supplement

Let us compile the following program with `gcc -S -O0`:

```
#include<inttypes.h>
void toto() {
  int64_t x=1;
  int64_t y=2;
  int64_t z=3;
  putchar('a');
}
```

Here is what we get (after I remove things that are beyond my understanding, and insert a few bank lines to structure the code):

Generated by `gcc`

```
  .file  "3_parameters.c"
  .text
  .globl  toto
  .type  toto, @function

toto:
  pushq  %rbp
  movq  %rsp, %rbp
  subq  $32, %rsp

  movq  $1, -24(%rbp)
  movq  $2, -16(%rbp)
  movq  $3, -8(%rbp)
  movl  $97, %edi
  call  putchar
  nop

  leave
  ret
```

Generated by `clang`

```
.text
  .file  "3_parameters.c"
  .globl  toto
  .type  toto,@function

toto:
  pushq  %rbp
  movq  %rsp, %rbp
  subq  $32, %rsp

  movl  $97, %edi
  movq  $1, -8(%rbp)
  movq  $2, -16(%rbp)
  movq  $3, -24(%rbp)
  movb  $0, %al
  callq  putchar
  movl  %eax, -28(%rbp)

  addq  $32, %rsp
  popq  %rbp
  retq
```

Exercise

- Identify the assembler and linker *directives*, beginning with a dot. I removed some of them.
- Identify the labels (I removed some, too).
- Show the prologue, the body and the epilogue.
- Determine, for the instructions of the body, if the destination operand is the first or the second one. Write it here somewhere to remember it.

The purpose of this chapter is to understand what is behind these prologues and epilogues.

III.1 The activation record

The "activation record" is what we called "memory pool" so far. It is the space in memory where all the local variables of a procedure or function are allocated. (from now on we will just use the term "function" for "procedure or function")

This space is allocated when a function begins execution. It is freed when the function exits. It is active in between, hence the name.

When a function calls another one, it creates a new activation record for the callee, but it does not destroy that of the caller: several activation records may be active at the same

time. This will be managed as a stack. When the callee exits, its space is freed and the execution resumes for the caller.

When a function is recursive, each time the procedure calls itself, a new set of fresh local variables are allocated, i.e. a new activation record. It is important to understand that in this case, only one copy of the code exists. The activation record is really about the variables of the program, not about its code.

During compilation, each local variable is assigned one position in the activation record. This position is defined by a constant offset relative to a base pointer that we call BP (it is called ARP for *activation record pointer* in the Cooper&Torczon).

This way, the same code can run with its local variables in different ARs: you change the BP, you change the AR.

Exercise In our AMD64 ABI, the base pointer is held in the register `%rbp`. The addressing mode `12(%rbp)` is a stupid way of specifying the content of the memory location at address `12+%rbp`. In C, it would be written `*(12+%rbp)`.

- Look for values 1, 2, 3 in the generated assembly code.
- What are the offsets of the C variables `x`, `y`, `z`?
- Did GCC and Clang chose the same offsets?

The activation record is most of the times allocated on the stack. An activation record on the stack is often called a *frame*[1]. Therefore the base pointer is sometimes also called *frame pointer*.

- Allocating and freeing a lot of space on the stack is as fast as an addition on the stack pointer (SP).
- The stack perfectly captures recursive function history. Actually recursion is deeply linked to the notion of stack.

SP versus BP

But then if the AR is on the top of stack, why do wee need a base pointer? We could use the stack pointer as BP.

This is absolutely true. Even when the compiler generates push operations (for example to pass parameters, see below), it knows how many it has generated and could still access its local variables by updating the offset.

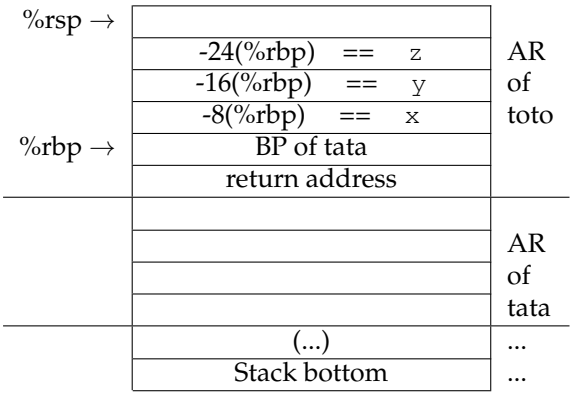
To see the need for a BP that is different from the SP, one has to think of something more exceptional, for instance exception recovery.

[1] In the actual assembly generated by GCC or Clang, you will find a lot of directives beginning with `cfi` or `cfa`, for *call frame information* or *call frame address*. According to Stack Overflow it is used for exception handling and debugging. We will ignore it.

III.2 Dissection of a stack frame

First, since we are dealing with a stack, we have to read in the manual that this stack is **descending in memory, with stack pointer pointing the last full entry.**

The figure below is a snapshot of the stack during the execution of procedure `toto` above. We see that the AR of `toto` contains the 3 local variables `x`, `y`, and `z`. Below `toto`'s AR, we see the AR of the procedure that called it, here `tata`.



It would have been enough to reserve 24 bytes. We have an empty location at the top due a 16-byte alignment constraint. Precisely, the ABI states that *the value (%rsp + 8) is always a multiple of 16 when control is transferred to the function entry point*. It doesn't say why: any ideas?

III.3 The prologue

The prologue first saves the BP of the caller by pushing it on the stack. Then it allocates, in one single addition, all the space required for the local variables.

See slides.

III.4 The epilogue

Its work can be summarized as: **please leave the stack in the same state as you found it when entering.**

3 options:

```
leave
ret
```

```
movq %rbp, %rsp
pop %rbp
ret
```

```
addq $32, %rsp
pop %rbp
ret
```

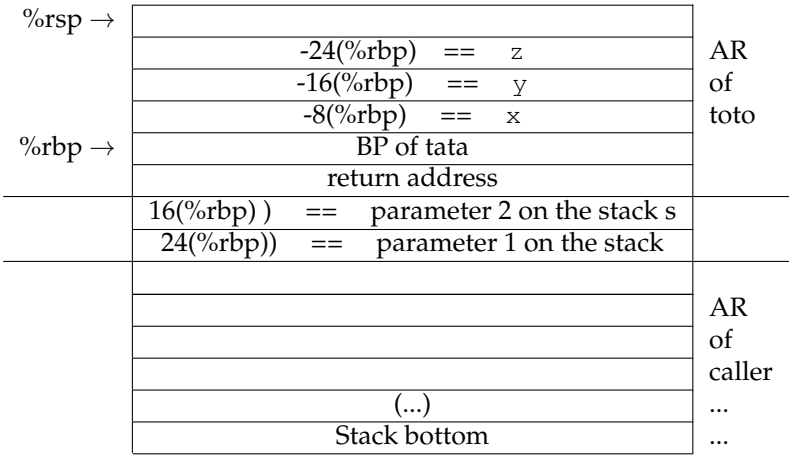
The third is the most explicit but takes more space (requires to encode the constant 32).

The description of the `leave` instruction in the ISA reference manual is: *Set RSP to RBP, then pop RBP*. So if you follow closely, you should understand that the three epilogues do the same thing.

(And `leave` is a one-byte instruction. That's CISC for you.)

III.5 Parameter passing

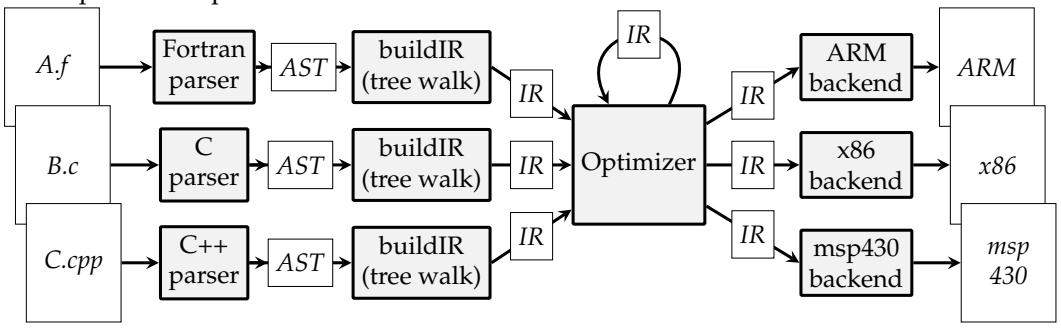
On all modern ABIs, function/procedure arguments are passed through registers if possible. This is faster than passing them on memory. However, there is only a limited number of registers. Arguments may also be passed on the stack. They will be between the AR of the caller and the AR of the callee. Both can access them by a fix offset with respect to the BP, as shown on the example below (during the execution of toto).



An ABI typically specifies what to do when a function/procedure has many arguments. For instance, on the AMD64 ABI, the 6 first parameters of a function are passed in a predefined set of registers, and the remainding ones are pushed on the stack. The details are ugly and you will discover them in the project.

III.6 Wrap up

And a final exercise for this chapter: on this beautiful figure of mine, where are the prologue and epilogue generation? Hint: they are completely ABI-specific, and an ABI is by definition processor-specific.



Another exercise for those interested in optimization: look what the prologue, epilogue and body have become when you compile the same program with `gcc -O2`.

CHAPTER IV

Introduction to optimization

Figure IV.1 shows an overview of the optimizations performed in modern compilers.

On this figure, **A** denotes source-level optimizations; **B** and **C** can be performed on the AST or on the IR; **D** are target-dependent optimizations; **E** are performed at link time.

This chapter overviews a few of these optimizations.

IV.1 Peep-hole optimization

This optimization is performed on the target code. The code generated by the techniques seen so far often does stupid things such as

- a jump to the address following immediately, as

```
  (...)
  jle BB_2_after_while
  jmp BB_1_while_body
BB_1_while_body:
  (...)
```

- writing something to memory and reading it back immediately after, as

```
  (...)
  movq  %rax, -24(%rbp)
  movq -24(%rbp), %rax
  (...)
```

The useless code can easily be detected and removed by a pass on the generated code. This is easier if we have a data-structure for generated code.

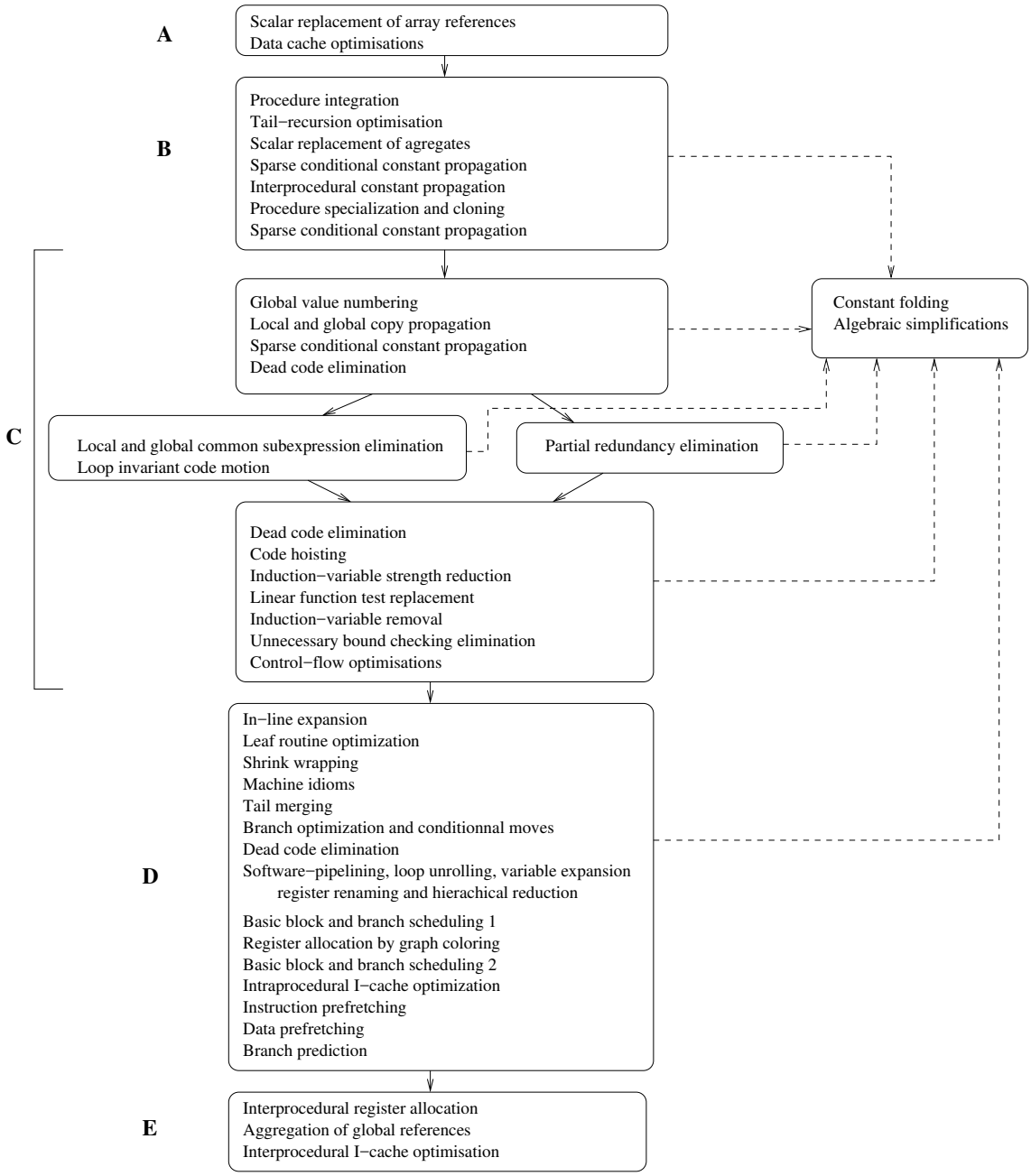


Figure IV.1: Compiler optimisations, from the book *Advanced Compiler Design and Implementation* by S. Muchnick

IV.2 Liveness analysis and register allocation

If you look back to Section II.4, it is easy to see that the generated code uses at most 2 registers, plus the base pointer. All the variables (including temporaries) is in memory. But memory access is slow, while modern ISAs provide many registers (typically 16-32).

The purpose of register allocation is therefore to use these registers to reduce the number of memory accesses in the code generated. A secondary benefit will be to use fewer temporary variables in the activation record, hence reducing its size.

We assume we have R registers, and we want to assign a register to each IR variables. We also want to keep the program variables in the registers.

IV.2.1 Detailed control flow graph

We now consider the CFG before assembly code generation, and we define the detailed CFG (DCFG) as the graph of IR instructions it represents.

No need for a new data structure: the DCFG is simply the CFG where the BB nodes are replaced with the corresponding linear subgraph of IR instruction – this graph is linear by definition of the BB.

IV.2.2 Liveness

A variable is alive on an edge of the DCFG if 1/ it may have been defined before, and 2/ it may be used later.

In other words, a variable is alive on an edge e of the DCFG if e is on a path from a definition of this variable to a use of this variable.

Why this notion? Because if two variables are live at the same point, they cannot share the same register. On the other hand, if two variables are never live together, they may share the same register.

Liveness is a property of *variables* that we attach to *edges* of the DCFG. What we will compute is, for each edge, the set of live variables.

And now for a few definitions. For a DCFG node n (an IR instruction),

- *Def*(n) is the set of variables defined by n. if n is a binary op, *Def*(n) is the singleton of the destination variable.
- *Use*(n) is the set of variables used in the node. Up to two variables in our case, except for call nodes which may use an arbitrary number of variables.
- *LiveIn*(n) is the set of variables alive when execution of n begins.
- *LiveOut*(n) is the set of variables that are alive just after the execution of n.

Not that if our IR had flags and instructions that consume them, the vivacity of the flags should be computed as well.

The sets *Def*(n) and *Use*(n) are defined for each node. The sets *LiveIn*(n) and *LiveOut*(n) are defined by the two following mutually recursive equations:

$$\mathit{LiveIn}(n) = \mathit{Use}(n) \cup (\mathit{LiveOut}(n) \setminus \mathit{Def}(n)) \tag{IV.1}$$

$$\mathit{LiveOut}(n) = \bigcup_{s \in succ(n)} \mathit{LiveIn}(s) \tag{IV.2}$$

Spend some time expressing these equations in natural language. The first defines what happens within a BB. The second is trivial within a BB, and useful between BBs.

One can see that the liveness information is computed on the DCFG in the backward direction, from the exit node to the input nodes.

These are *dataflow equations*. The sets may actually computed by iterating them over the CFG:

- start with the output node, whose *LiveOut* is the empty set;
- follow the CFG edges backwards, applying equations (IV.1) and (IV.2).

Since there may be loops in the CFG, there must be a termination condition. This condition is: iterate until a fixed point is reached, *i.e.* until applying the equations doesn't change the sets anymore.

Are we sure that this will terminate? The answer is yes, and the sketch of the proof is:

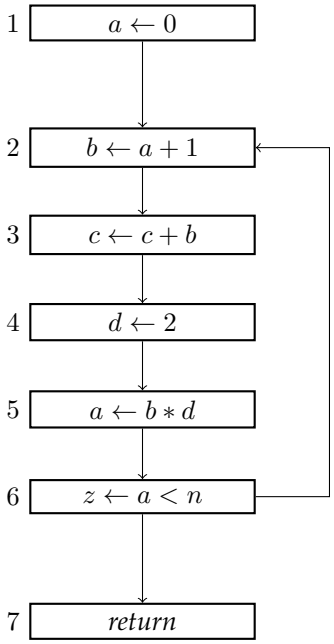
- Equation (IV.2) only grows the sets $\mathit{LiveOut}(n)$.
- Equation (IV.1) also only grows the sets $\mathit{LiveIn}(n)$, since the set of variables removed by the $\setminus \mathit{Def}(n)$ part is constant for each iterations.
- These are sets of variables and there is a finite number of variables, so they cannot grow forever.

There is even a convergence theorem that shows that whatever the iteration order chosen, it converges to the same fix point.

IV.2.3 Example

You should have understood that *LiveIn* and *LiveOut* are the node-centric point of view of the iseful *Live* live information that its attached to edges.

Exercise: on the following figure, add the liveness information on the edges of the DCFG on the left. The solution is on the right.



edge	Live set
$1 \to 2$	a, c, n
$2 \to 3$	b, c, n
$3 \to 4$	b, c, n
$4 \to 5$	c, b, d, n
$5 \to 6$	a, c, n
$6 \to 7$	$\emptyset$
$6 \to 2$	a, c, n

IV.2.4 Live ranges

Once we have the liveness information, we know that two variables alive on one edge cannot share a register.

However it often happens that the same variable is reused several times within a basic block. This may happen in the source code, but also because our code generation algorithms recycle temporary variable names (see the discussion at the end of II.2.3).

What is important is therefore the notion of *live range*: the interval between the definition of a variable and its use. Obviously we don't need to associate the same register to the different live ranges of a variable.

Live ranges within a basic block are easy to define by a linear pass. In our example b is alive between instructions 2 and 5. On the full CFG, it is a bit more complicated. A live range for the variable v is a connex subset of the nodes.

Formally, two DCFG nodes a and b belong to the same live range associated to variable v if there exists a path P of the DCFG from a definition of v to a use of v without any redefinition of v which includes a sub-path from a to b.

Read the previous 3 times, then don't worry. In the chosen example, there is only one live range for each variable.

IV.2.5 Interference graph

The *interference graph* is a new graph whose nodes are the live ranges. There is an edge between two nodes when the live ranges overlap, *i.e.* there exists an edge on the DCFG where the two variables are alive together.

IV.2.6 Register allocation

... is a coloring of the interference graph such as adjacent nodes do not have the same color. Each color corresponds to a register. If we have 16 registers we look for a coloring with 16 colors.

That's it.

If such a coloring is not possible (i.e. it is not possible to keep all the variables in registers because we do not have enough registers), no problem: some variables will stay in memory.

However we have a new optimization problem to solve: which variables go to memory? The jargon term is: which variables will be spilled to memory?

I will not address this question but you find it in the textbooks.

IV.3 Dead code elimination

An instruction $d \leftarrow a \oplus b$ can be removed safely if d is not live out (unless it is a comparison at the end of a BB as in our example).

There are many more possibilities of dead code elimination.

IV.4 Sub-expression sharing

This is an optimization that is relatively easy to perform locally on the AST of an expression. The traditional technique is called *value numbering*. However, extending it to branches and loops is very difficult.

What we present here is a technique in two steps:

- a dataflow analysis that allows to detect shared subexpressions directly on the CFG;
- a code transformation step that exploit this information to replace a recomputation of an expression with a copy of the variable in which this expression was computed.

IV.4.1 Available expressions

The set $AvailIn(n)$ is the set of expressions available at the input of n. An expression is simply binary expression such as $a * b$ The idea is that if later the compiler finds that the same expression is recomputed, it can replace this computation with a copy.

However, an expression will no longer be available as soon as one of its operands is reassigned: if between the assignment to v and the assignment to v' the value of a has changed, then the asignment to v' needs to recompute the expression.

Therefore, a binary operation $d \leftarrow a \oplus b$ defines a new available expression computed in d (and possibly built out of the available expressions in a and b if they exist). However it also kills all the available expressions that used d, since the value of d changes with this instruction.

Therefore, we define two sets:

- $ExprDef(n)$ is the expression defined by n. If n is $d \leftarrow a \oplus b$ and d is neither a nor b, then $ExprDef(n) = \{a \oplus b\}$. Otherwise $ExprDef(n) = \emptyset$.

- $ExprKill(n)$ is the set of expressions killed by n, i.e. all the expressions that have d as an argument.

The dataflow equations that defines expression availability are:

$$AvailIn(n) = \bigcap_{p \in pred(n)} AvailOut(p) \tag{IV.3}$$

$$AvailOut(n) = ExprDef(n) \cup \big(AvailIn(n) \setminus ExprKill(n)\big) \tag{IV.4}$$

Here the intersection means that the expression must be available on all the edges leading to n. This information propagates forward in the DFG (since the intersection is on all the predecessors).

IV.4.2 Subexpression sharing

The algorithm is more or less the following.

- identify $n = d \leftarrow a \oplus b$ where $a \oplus b$ is available in input.
- define a new temporary w.
- replace n with $n' = d \leftarrow w$
- after each definition $t \leftarrow a \oplus b$ that reach n, add a copy $w \leftarrow t$.

Damned, how do we know which assigments defined this shared subexpression? Here we need one more analysis.

Remark: this algorithm adds many copies. They will be removed by the register allocation.

IV.4.3 Reaching definitions

This notion is related to vivacity. The purpose here is to define the set of IR instructions that depend on a given variable definition.

A *definition* of variable t is a node of the DCFG $t \leftarrow a \oplus b$.

A definition $d = t \leftarrow a \oplus b$ reaches another DCFG node n if there exists a path from d to n without any redefinition of t.

For a DCFG node n, the sets $RDIn(n)$ and $RDOut(n)$ are respectively the sets of definitions that reach the input and the output of n.

The set $VarDef(t)$ is the set of DCFG nodes that define t.

For a definition $d = t \leftarrow a \oplus b$ or $d = t \leftarrow c$, we define

$$RDDef(d) = \{d\}$$

and

$$RDKill(d) = VarDef(t) \setminus \{d\} \quad .$$

For other nodes, both sets are defined as the empty set.

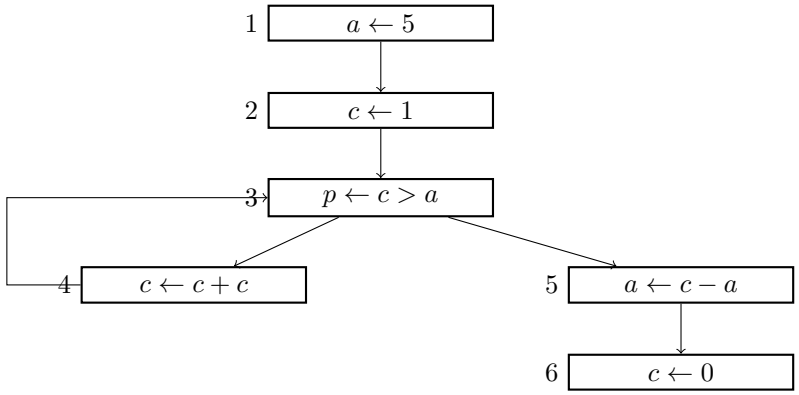
The dataflow equations for reaching definitions are:

$$RDIn(n) = \bigcup_{p \in pred(n)} RDOut(p) \tag{IV.5}$$

$$RDOut(n) = RDDef(n) \cup \big(RDIn(n) \setminus RDKill(n)\big) \tag{IV.6}$$

This information propagates forward in the DFG.

IV.4.4 Example



statement d	$RDDef(d)$	$RDKill(d)$	$RDIn(d)$	$RDOut(d)$
1				
2				
3				
4				
5				
6				

CHAPTER V

Probing further

V.1 Language features we won't need to manage

- floats, doubles: nothing new
- structs, unions: little new (we trust you would invent the good solution)
- pointers: add a lot of subtleties.
- object orientation (overloading, inheritance and name resolution). This requires more complex data structures, more complex activation records, and the generation of more code.
- pre-processing and templating: this is well beyond my understanding.

Other languages (e.g. functional and logic languages) require completely different compiler techniques: this is out of scope.

Bibliography

[App98] Andrew W. Appel. *Modern Compiler implementation bin Java*. Cambridge University press, 1998.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[CT03] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan-Kaufmann, 2003.

[GBJL00] D. Grune, H. Bal, J. H. Jacobs, and K. Langendoen. *Modern Compiler Design.* John Wiley & Sons, 2000.

[int16] *Intel 64 and IA-32 Architectures, Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z*, sep 2016.

[ISO18] ISO/IEC. *International Standard ISO/IEC ISO/IEC 9899:2018. Programming languages – C*. December 2018.

[Muc97] Steven S. Muchnick. *Compiler Design Implementation*. Morgan-Kaufmann, 1997.