Intermediate Code Generation (ICG)

Last phase of front end.

Generates intermediate code (IC):
- separates front end (FE) from back end (BE), so FE and BE are independent of each other.
- intermediate level of abstraction between high level parse tree and low level microprocessor instructions.

IC is internal to compiler, so each compiler can have its own kind of IC. Possible types of IC: syntax trees, static single assignment (SSA), three address code (3AC).

We use 3AC:
- x = y op z (binary operator)
- x = op y (unary op)
- x = y (copy)
- goto L (unconditional goto, L is label of destination)
- if x relational_op y then goto L (conditional goto)
- one op and three addresses (variable names) at most for each 3AC

We already saw how to go from parse tree to syntax tree, plus going from syntax tree to 3AC is easy: generate one 3AC instruction for each interior node of syntax tree. Here we go from parse tree directly to 3AC.

We need two synthesized attributes:
- E.name stores the name of the variable containing the value of E
- E.code stores the sequences of 3AC that computes the value of E

We generate temporary names for intermediate results as necessary using function newtemp(). '||' in semantic rules means "followed by".

Grammar productions and associated semantic rules:

```
S -> id = E
S.code := E.code || id.name = E.name
(no S.name because statements don't compute a value)

E1 -> E2 + E3
E1.name := newtemp()
E1.code := E2.code || E3.code || E1.name = E2.name + E3.name

E1 -> E2 * E3
E1.name := newtemp()
E1.code := E2.code || E3.code || E1.name = E2.name * E3.name

E1 -> - E2
E1.name := newtemp()
E1.code := E2.code || E1.name = - E2.name

E1 -> ( E2 )
E1.name := E2.name
E1.code := E2.code

E -> id
E.name := id.name (intrinsic)
E.code := '' (nothing to compute)
```

Example: source code 'a = b + c * d' gives the sequence of 3AC:

```
t0 = c * d
t1 = b + t0
a = t1
```

For control flow, need to introduce new attributes for labels and generate new labels as necessary using function newlabel().

```
S1 -> while E do S2
S1.begin := newlabel()
S1.after := newlabel()
```

```
S1.code := S1.begin: ||
             E.code ||
             if E.name == 0 then goto S1.after ||
               S2.code ||
             goto S1.begin ||
           S1.after:
```

Example: source code ’while a + b do x = y * z’ gives the sequence of 3AC:

```
L0:
  t0 = a + b
  if t0 == 0 then goto L1
    t1 = y * z
    x = t1
  goto L0
L1:
```

```
S1 -> if E then S2 else S3
S1.else := newlabel()
S1.after := newlabel()
S1.code := E.code ||
           if E.name == 0 then goto S1.else ||
             S2.code ||
             goto S1.after ||
           S1.else: ||
             S3.code ||
           S1.after:
```

Example: source code ’if a + b then x = y * z else u = v - w’ gives the sequence of 3AC:

```
t0 = a + b
if t0 == 0 then goto L0
  t1 = y * z
  x = t1
  goto L1
L0:
  t2 = v - w
  u = t2
L1:
```

Same idea for for loop, switch statement, etc.

Variable definitions: we need to compute the type (as in the type checking lecture notes) but also compute the amount of memory necessary to allocate for each variable => we need to compute the size of each type. We also need to decide where the memory will be allocated at runtime => assume that all variables are global (allocated from static area of program) and use an offset counter which is initialized to be the start address of the static area of the program and which is incremented by the size of a variable every time a new variable is allocated in the static area (i.e. offset is always a pointer to the first free memory byte in the static area).

```
P -> { offset := start_of_static_area } D
```

(we need to initialize offset before we compute the attributes in the parse tree for D: this is a translation scheme (the only one we will ever use in this course), not a syntax directed definition...)

```
D -> D ; D
```

(no attribute to compute)

```
D -> T id
addtype(id.entry, T.type, T.size, offset)
offset := offset + T.size
```

(allocate a variable of size T.size at address offset, then increase offset for next variable)

```
T -> int
T.type := integer
T.size := 4
```

(32 bits machine => integer is 4 bytes)

```
T -> char
T.type := character
T.size = 1

T1 -> T2 [ num ]
T1.type := array(0...(num.val - 1), T2.type)
T1.size := num.val * T2.size
(array size is size of one array element * number of elements)

T1 -> T2 *
T1.type := pointer(T2.type)
T1.size := 4
(type of pointer nowadays is usually the same as size of integer => 4
bytes on 32 bits machine)

If we want to allocate the memory for a variable from the stack
instead of from the static area, then we just need to initialize
offset to be the current stack frame pointer (which points at the top
of the current stack frame in the stack, see your operating system
lecture notes).

The attribute T.size is also how the compiler computes the result of the
special sizeof operator.  This operator is special because it takes a type
as argument (not a runtime value made of bits) and its result is computed
at compile-time using T.size, it is not computed at runtime (because types
do not exist at runtime, only bits exist at runtime).


Code Optimization
-----------------

Make code better: faster, or smaller, etc.  Here we only look at
optimizations to make code faster.

Optimizations must:
- preserve the meaning of the program (preserve results and preserve
  exception) => be careful about some optimization: 10^20 - 10^20 + 1
  is equal to 1, but 10^20 + 1 - 10^20 is 0 (because 10^20 is too big
  compared to 1 so 10^20 + 1 is the same as 10^20) => optimizations
  always have to be conservative (in theory, in practice compiler
  implementors often take shortcuts and introduce subtle bugs...)

- should improve code (on average).  Not always easy to be sure that
  code will be faster, because optimizations also have effects on data
  placement in the microprocessor's cache, on pressure on registers
  during register allocation, on instruction selection and scheduling,
  etc.  Most simple optimizations below always improve the code, but
  some optimizations only improve code on average: out of 100 programs
  90 will run faster but 10 will run slower => people who create
  optimizations spend a lot of time testing them on a lot of code...

- should have reasonable cost (i.e. take seconds, not days).  In
  practice code follows the 90/10 rule (or 80/20 rule, depending on
  people): 90% of the execution time of a program is spent executing
  only 10% of the code => spend time optimizing only the 10% of code
  that matters (e.g. code inside loops) and forget about optimizing
  the remaining 90% of code => optimization much faster (this is the
  idea behind just-in-time compilation, see your operating system
  lecture notes).

Optimizations take 3AC as input and produce (faster) 3AC as output =>
can chain optimizations in a sequence, or even in a loop: very common
for a compiler to repeat the same optimizations several times in a
loop until code is good (generating optimum code is NP-complete
though: finding the best possible code requires trying all the
possible combinations of 3AC, so in practice optimizations only do
their best).

Optimizing 3AC is best because optimizations are independent of source
language and target language of compiler.  It is also possible to have
language-specific optimizations (optimizing the dynamic dispatch of
```

methods in Java, for example) by optimizing the parse tree after the semantic phase and before 3AC generation, or to have microprocessor-specific optimizations (using special microprocessor instructions, for example) by optimizing the assembly language after the code generation, but here we only look at optimization for 3AC.

Examples of Simple Optimizations

Suppose you have three integer arrays a, b, and c, and want to add the content of a to the content of b and store it in c => put the source code 'c[i] = a[i] + b[i]' inside a loop. Corresponding 3AC is:

```
t0 = 4 * i (array indexing as to take size of array elements into account)
t1 = a[t0]
t2 = 4 * i
t3 = b[t2]
t4 = t1 + t3
t5 = 4 * i
c[t5] = t4
```

Common Subexpression Elimination (CSE): replace redundant computations with the corresponding name. In the code above 4 * i is computed multiple times => compute it only once:

```
t0 = 4 * i
t1 = a[t0]
t2 = t0        (changed)
t3 = b[t2]
t4 = t1 + t3
t5 = t0        (changed)
c[t5] = t4
```

Copy Propagation (CP): if a variable is just a copy of another variable, replace that variable with the name of the original variable. In the code above t2 and t5 are copies of t0 => replace them with t0:

```
t0 = 4 * i
t1 = a[t0]
t2 = t0
t3 = b[t0]     (changed)
t4 = t1 + t3
t5 = t0
c[t0] = t4     (changed)
```

Dead Code Elimination (DCE): remove the definition of any variable which is not used. In the code above t2 and t5 are never used => remove their definitions:

```
t0 = 4 * i
t1 = a[t0]
t3 = b[t0]
t4 = t1 + t3
c[t0] = t4
```

Note: DCE is more powerful than that: it can remove any code which is never used. For example, DCE can transform the code 'if true then S1 else S2' into just the code 'S1' since the test of the if statement is always true.

Constant Folding (CF): compute at compile time all the operations that have constant operands. For example, replace:

```
t0 = 2 * 3.14
c = t0 * r
```

with:

```
t0 = 6.28       (changed)
c = t0 * r
```

Constant Propagation (CP): replace variables with constant values by their value. In the code above t0 is constant, so we replace t0 by its value in the computation of c:

```
t0 = 6.28
c = 6.28 * r    (changed)
```

If we then do DCE we get just the code:
`c = 6.28 * r`
since the variable t0 is not used anymore. So doing CF, CP, and DCE in sequence transformed the original code:

```
t0 = 2 * 3.14
c = t0 * r
```

into just:

```
c = 6.28 * r
```

Each optimization is very simple, but by combining them in a sequence (and repeating them in a loop) we can get very good results.

Another example: start with the code:

```
x = 3
y = x + 4
```

After CP:

```
x = 3
y = 3 + 4
```

After CF:

```
x = 3
y = 7
```

After DCE:

```
y = 7
```

Or starting from the same code again:

```
x = 3
y = x + 4
```

After CP:

```
x = 3
y = 3 + 4
```

After DCE:

```
y = 3 + 4
```

After CF:

```
y = 7
```

So some optimizations can be sequenced in different orders and still get the same result (that’s what we expect since optimizations have to preserve the meaning of the code).

Compilers also often use algebraic identities to simplify code:

`x + 0 = x = 0 + x`

`x - 0 = x (but x != 0 - x)`

`x * 1 = x = 1 * x`

x * 0 = 0 = 0 * x

x / 1 = x (but x != 1 / x)

2 * x = x + x (addition is often faster than multiplication)

2 * x = x << 1 (left shift by one bit, works only for integers)

x / 2 = 0.5 * x (multiplication is often faster than division, works only for floating point numbers, since integer division does extra rounding)

x / 2 = x >> 1 (right shift by one bit, works only for integers)

Optimization for Loops

Code Motion: move constant computation outside of loops.
For example, in this code:

```
t0 = ...
L0:
  t1 = t0 + 5
  ... (value of t0 never changes inside the loop) ...
  goto L0
```

the value of t0 does not change inside the loop, therefore the value of t1 never changes inside the loop, so, instead of computing the same t1 during each loop iteration, we can just compute t1 once:

```
t0 = ...
t1 = t0 + 5
L0:
  ... (value of t0 never changes inside the loop) ...
  goto L0
```

Loop Unrolling: copy the body of a loop a certain number of times to decrease the number of instructions to execute.

For example, transform this for loop:

```
for(i = 0; i < 3; i++){
  sum += a[i];
}
```

into this code:

```
sum += a[0];
sum += a[1];
sum += a[2];
```

The first version of the code does:
- initialize i to zero: 1 operation
- test i against 3: 4 operations (the test succeeds three times and fails one time to terminate the loop)
- increment i by 1: 3 operations (once per loop iteration)
- add a[i] to the sum: 3 operations (once per loop iteration)
- goto back to the top of the loop: 3 operations (once per loop iteration)

total: 14 operations.

The second version of the code only does three additions, so a total of 3 operations, so it is much faster than the first version of the code.

What about this loop:

```
for(i = 0; i < 1000; i++){
  sum += a[i];
}
```

Here we do not want to completely unroll the loop because the resulting code would be very big and would completely fill the

microprocessor’s cache, which means the code would then run more slowly!

Even in such a case we can still do some partial loop unrolling:

```
for(i = 0; i < 1000; i += 2){
  sum += a[i];
  sum += a[i + 1];
}
```

The first version of the code does:
- initialize i to zero: 1 operation
- test i against 1000: 1001 operations (the test succeeds 1000 times and fails 1 time to terminate the loop)
- increment i by 1: 1000 operations (once per loop iteration)
- add a[i] to the sum: 1000 operations (once per loop iteration)
- goto back to the top of the loop: 1000 operations (once per loop iteration)

total: 4002 operations.

The second version of the code does:
- initialize i to zero: 1 operation
- test i against 1000: 501 operations (since i is incremented in steps of 2, the test succeeds 500 times and fails 1 time to terminate the loop)
- increment i by 2: 500 operations (once per loop iteration)
- add a[i] to the sum: 500 operations (once per loop iteration)
- compute i + 1: 500 operations (once per loop iteration)
- add a[i + 1] to the sum: 500 operations (once per loop iteration)
- goto back to the top of the loop: 500 operations (once per loop iteration)

total: 3002 operations.

So the second version is faster than the first one. The same loop can be unrolled again:

```
for(i = 0; i < 1000; i += 4){
  sum += a[i];
  sum += a[i + 1];
  sum += a[i + 2];
  sum += a[i + 3];
}
```

This third version of the code does:
- initialize i to zero: 1 operation
- test i against 1000: 251 operations (since i is incremented in steps of 4, the test succeeds 250 times and fails 1 time to terminate the loop)
- increment i by 4: 250 operations (once per loop iteration)
- add a[i] to the sum: 250 operations (once per loop iteration)
- compute i + 1: 250 operations (once per loop iteration)
- add a[i + 1] to the sum: 250 operations (once per loop iteration)
- compute i + 2: 250 operations (once per loop iteration)
- add a[i + 2] to the sum: 250 operations (once per loop iteration)
- compute i + 3: 250 operations (once per loop iteration)
- add a[i + 3] to the sum: 250 operations (once per loop iteration)
- goto back to the top of the loop: 250 operations (once per loop iteration)

total: 2502 operations.

So this third version is faster then the second one (but the difference is less than between the first and second version). We can repeat the same process until the body of the loop becomes too big to fit inside the microprocessor’s cache, at which point unrolling the loop even more will start to make the code slower.

In practice there are dozens of optimizations for loops (Fortran compilers, for example, are very good at optimizing loops, because Fortran is quite specialized for scientific computations where the programs manipulate big arrays using lots of nested loops) and hundreds of optimizations in general. Most compiler implement the most simple ones (like CSE, DCE, loop unrolling etc.) because these optimization are easy to implement, give a good speedup to the code,

and have a small cost in terms of extra compilation time.

Conclusion: do not waste time trying to optimize the details of your code, the compiler will most likely do it for you anyway. If you want to make your code run much faster then use faster algorithms, don't try to manually optimize small details that the compiler can take care of.

Code Generation

Last phase in the compiler: takes optimized 3AC from the optimization phase and generates assembly code (which will later be transformed into relocatable object code by an assembler, see the first lecture notes in this course).

To generate assembly code, we need to do three things:
- instruction selection
- instruction scheduling
- register allocation

As we have seen in the first lecture notes, all these three tasks are NP-complete: the only way to generate the best assembly code is to try all the possible combinations of assembly code and test them one by one. Since there is an exponential number of possible combinations, testing them all is not feasible in practice, so the code generation phase of a compiler only tries its best to generate code that is good enough, but there is not guarantee that the generated code will be the best one: the compiler generates only an approximation of the best code (hopefully). To simplify the problem, here we only present very simple algorithms to solve the three tasks listed above.

Before we see how to generate code, we first have to talk about the kind of assembly code we want to generate. Assembly instructions:
- ADDI R0, R1 integer addition between register, result goes implicitly into R1
- ADDI a, b integer addition between memory locations, result goes implicitly into the memory for b
  This is only available for CISC machines, not RISC machines
- MOV a, R0 copies the value of a from memory to register R0
- MOV R0, a copies the value in register R0 into the memory for a

We assume that every instruction has a basic cost of 1 (the compiler needs to compute a cost to know which instruction is the best one) This is a big approximation, since in practice the cost of an instruction depends on instruction itself, the amount of pipelining for that instruction done in the microprocessor, etc.

We also assume that there is an extra cost depending on the kind of addressing mode used by the instruction (we assume an extra cost of 1 for each extra word of memory necessary to specify the whole instruction):

Mode	Form	Address	Extra Cost
absolute	a	memory	1 (for the memory address)
register	R	register	0
indexed	c(R)	c+contents(R)	1 (for c)
indirect register	*R	contents(R)	0
indirect indexed	*c(R)	contents(c+contents(R))	1 (for c)

The indexed mode is useful to access array elements: R contains the address of the first element of the array and c is the offset of the element within the array. It's the hardware equivalent of the C language's [] syntax for array accesses.

The indirect register mode is useful when using pointer: R contains the value of the pointer (address of the value pointed at) and *R is then the value pointed at. It's the hardware equivalent of C's * syntax for pointer dereference.

The indirect indexed mode is useful when manipulating dynamic data structure like linked lists, trees, graphs, etc. For example R contains the address of a linked list node, then c+contents(R) is the address of one of the member of the node, and *c(R) therefore is the value of that member inside the node. It's somewhat the equivalent of C's -> operator for pointers (i.e. a pointer dereference followed by indexing into a data structure, in other words, the combination of the * and the . syntaxes).

Of course in a RISC machine only the register mode is allowed (except for the MOV instruction which can also use one absolute mode (and only one)). That's why it's called a RISC machine (Reduce Instruction Set Computer, i.e. the number of modes is reduced compared to a CISC machine) In a CISC machine all modes are allowed, which means that the microprocessor has to be much more complicated to implement all these possible mode. That's why RISC microprocessors can run faster: their hardware is conceptually simpler and so can be better optimized for speed.

As you can also see from the table, we want to maximize the use of register, since they have an extra cost of zero. In practice using registers is good too, because hardware operations on registers are much faster than operations on memory.

Examples:

Instruction	Meaning	Total Cost
MOV R0, R1	contents(R0) -> R1	1
MOV R0, a	contents(R0) -> a (in memory)	1 + 1 (for a)
ADD #1, R3	1 -> R3 (#1 is an integer literal)	1 + 1 (for #1)
SUB 4(R0),*12(R1)	contents(contents(12+contents(R1))) - contents(4+contents(R0)) -> contents(12+contents(R1))	1 + 2 (for 4 and 12)

Suppose we have the 3AC 'a = b + c' then we can implement it in various ways with different costs:

```
MOV b, R0
ADDI c, R0   (CISC machine only)
MOV R0, a
```

Total cost: 6 (3 instructions + 3 memory addresses a, b, and c)

```
MOV b, a     (CISC machine only)
ADDI c, a    (CISC machine only)
```

Total cost: 6 (2 instructions + 4 addresses)

```
MOV b, R0
MOV c, R1
ADDI R0, R1
MOV R1, a
```

This code works for both CISC and RISC machines.
Total cost: 7 (4 instructions + 3 addresses)

Even though it looks like a CISC machine can use assembly code with a lower cost (6) compared to the cost of the assembly code for a RISC machine (cost of 7), in practice the code for the RISC machine will end up being faster: in the RISC code, the addition ADDI is done between registers so it is very fast, while the ADDI addition in the CISC code uses at least one (or even two) memory addresses, which means that in the CISC case the speed of the addition will be limited by the speed of the memory, which in computers is much slower than the speed of the microprocessor. So even though a RISC computer will have to execute more instructions the total time will in fact be lower, because each instruction on the RISC machine can be executed much faster than on a CISC machine. I.e. using a big number of simple instructions is better than using a small number of complicated instructions, since with simple instructions the microprocessor can run at a much higher speed.

In practice of course the cost of an instruction depends a lot on how the instruction is implemented in hardware with how many pipeline stages, it depends on the size of the caches, the number of registers, etc. The back end of a good compiler therefore needs to have a lot of knowledge about the exact architecture of the computer (and particularly of the microprocessor) in order to generate good code. That's why the best compiler for a given microprocessor often comes from the manufacturer of the microprocessors itself (e.g. Intel) since the manufacturer has access to all the information required to implement a good code generator. Here our cost model is extremely simplified.

Basic Blocks

To generate assembly code starting from 3AC, we first split the 3AC into basic blocks, then generate the assembly code for each basic block one by one, then glue back together all the pieces of assembly code to get the final target program.

A basic block (BB) is a sequence of 3AC such that:
- the flow of control enters at the top (first 3AC in BB)
- the flow of control leaves at the bottom (last 3AC in BB)
- there is no goto in or out of the middle of the basic black
In short a basic block is a "flat" sequence of 3AC that you can only enter at the top and leave at the bottom.

To compute BBs we first compute leaders:
- the very first 3AC is a leader
- the target of a goto is a leader
- the 3AC after a goto is a leader

For example:
```
1 t0 = ...
2 t1 = ...
3 L0:
4 t2 = ...
5 t3 = ...
6 goto L0
7 t4 = ...
8 t5 = ...
```

Here the 3AC number 1 is a leader (first 3AC in the code), the 3AC number 3 is a leader (target of a goto), and the 3AC number 7 is a leader (3AC after a goto).

Once we have computed leaders, then a BB is a leader plus all the 3AC after the leader until (but not including) the next leader.

For example, in the code above there are three leaders: 3AC number 1, 3, and 7. Therefore there are three basic blocks:

```
1 t0 = ...
2 t1 = ...

3 L0:
4 t2 = ...
5 t3 = ...
6 goto L0

7 t4 = ...
8 t5 = ...
```

The flow of control only enters a BB at its top and only leaves at its bottom. There are no goto in or out of each BB directly in the middle of the BB: by definition gotos can only appear at the end of a BB, and only jump to the beginning of a BB.

Once we have BBs, we then do instruction selection, instruction scheduling, and register allocation for each BB independently of the other ones. To simplify the problem we assume that all registers are

empty at the start of each BB. Once we have generated the assembly code for each BB, we glue all the pieces of assembly code for each BB back together using (assembly instructions for) gotos.

Instruction Selection

We use a very simple method: to determine which assembly instruction to use, we only look at the 3AC we want to implement and the types of the operands, using a table. So to implement some operation 'x op y' we use a table like this:

op	x	y	OP
+	int	int	ADDI
+	float	float	ADDF
-	int	int	SUBI
-	float	float	SUBF
*	int	int	MULI
*	float	float	MULF

etc.

So, for example, if x and y are integers (the compilers knows the types of x and y since it stored these types in the symbol table when doing type checking) then, based on the table, we use the ADDI assembly instruction to implement 'x + y'.

Note: the table does not contain entries like:

op	x	y	OP
+	int	float	?

That's because microprocessors do not have hardware (and therefore no instruction) to do additions between mixed types like int and float. In fact that's why the compiler automatically insert type conversions in the parse tree during type checking, so that the back end of the compiler never has to generate code for 'x + y' when x and y have different types. So the table above that we use to select assembly instructions does not contain entries for mixed types.

Of course using such a table means we are not always generating the best possible code. For example, if x and y are integers, then 'x * y' will always be implemented using the MULI instructions, even though the code might end up being faster in the case where x is always 2 (in which case we could use an ADDI instruction to simulate 'y + y', which would be faster, but our simple table does not allow this).

Instruction Scheduling

Once we have used instruction selection to decide which assembly instruction to use, we have to decide on an order in which to use them. Here again we use a very simple solution: we use for the assembly instructions the exact same order as for the 3AC. This is not always optimal (since there might dependencies between various computations in the 3AC, which might trigger pipeline stalls inside the microprocessor, see the first lecture notes for detail) but it is a simply solution that is guaranteed to always work since we know that the 3AC was generated bottom-up from the parse tree in the intermediate code generation phase, and we know any bottom-up order will work.

Register Allocation

We now have to decide in which register we are going to store which variable, and when, during the computation.

For CISC computers, the simplest register allocation method is to do no register allocation at all. We know that CISC instructions always work even when using memory locations only, so we do not need to move values into register before or after doing any computation. For example, for the 3AC 'x = y + z', we can simply use the following assembly code:

```
MOV z, x
ADDI y, x
```

This code works and does not use any register. Of course if we do this then the resulting target program will run extremely slowly, since the speed of all computations will be limited by the speed of the memory, which is much slower than the microprocessor. But at least it's a simple to avoid doing register allocation and it always works

If we want to produce CISC code which is a bit faster, or if we are using a RISC computer, then we do not have the choice: we have to do some register allocation. The problem is that microprocessor have few registers (usually a few dozens, rarely more than 50 because of electrical limitations) and the program we are compiling can contain thousands of variables (plus thousands of temporary variables that were created during the intermediate code generation). So registers are scarce and variables are plenty. Deciding which variable should go into which register, when, and for how long, is therefore not going to be easy.

To help with the task we are going to keep two extra tables of information:

- a table of register descriptors (RD) (one row in the table for each register in the microprocessor) that indicates which variables are currently stored in which register (a register can store several variables if all the variables have the same value);

- a table of address descriptors (AD) (one row in the table for each variable in the program) that indicates which locations (memory and / or register) currently stores the value of the variable (a variable can be stored in multiple locations if it is copied many times). Since we need one address descriptor for each variable, we can actually store the address descriptor in the symbol table entry for the variable, since the symbol table has one entry for each variable in the program (which already stores the type of the variables, its size, etc.)

Initially, when we start generating the code for the current basic block, we assume that all registers are empty, so that generating the code for one basic block is completely independent from generating the code for another basic block. While this makes the code generation algorithm below much simpler, it means that the assembly code generated by the algorithm will not be the fastest one because variables used by multiple basic blocks will be stored in memory between basic blocks instead of being stored in registers. Real compilers use more complex code generation algorithms that keep track of which variables are used in which basic blocks, and try to keep variables used in multiple basic blocks in registers between basic blocks.

The algorithm to do register allocation is then as follows: for each 3AC of the form 'x = y op z' in the current basic block, do:

- call the getreg() function (see below) to determine the location L for the result x of the operation:
  - on a CISC machine L can be either a memory location or a register (which is preferred for speed reasons)
  - on a RISC machine L has to be a register

- look at the AD for y and select one of the current locations y' of y (register preferred)
  - if y' is not already in L:
    - output the assembly instruction: MOV y', L
    - update the RD and AD for y and L

- look at the AD for z and select one of the current locations z' of z (register preferred)
  - if z' is already in a register R, do nothing

  - if z’ is not already in a register:
    - call getreg() to get a location R for z’
      - on a CISC machine R can be either a memory location or a register (which is preferred for speed reasons)
      - on a RISC machine R has to be a register
    - output the assembly instruction: MOV z’, R
    - update the RD and AD for z and R

- do instruction selection based on ’op’ and the types of y and z
  - output the assembly instruction: OP R, L
  - remove L from the AD of y, since y has now been overwritten in L by x

- remove x from all RDs (since we just computed a new x, all the old values of x are now obsolete) and add x to the RD for L (if L is a register)

- clear the AD for x and add L to AD for x

- if y and / or z are not used again later and are in registers, then clear the RDs for them and update their ADs

For example, for a 3AC operation like ’x = y + z’, the algorithm above will generate code like this (assuming y and z are not yet in registers):

```
MOV y’, L
MOV z’, R
ADDI R, L
```

At that point L contains the value of x. Later, once the end of the basic block is reached, we need to save all the (useful) values currently stored in the registers back into memory, so that the registers are empty again when we start generating code for the next basic block.

So when we reach the end of the current basic block, do:

- look at all the RDs to determine which registers currently contain the values of variables that we might need to save in memory

- check whether each of these variables is used again later in the program: if some variables are not used again then we can forget the values of these variables, so do nothing for those variables

- use the ADs to check whether the values of the remaining variables are already in memory or not (remember that a variable can be in multiple locations at the same time): if some variables are already somewhere in memory then there is no need to save the content of the corresponding registers, so do nothing for these variables

- output MOV instructions to save the remaining variables from registers to memory, update the AD for those variables

- clear all RDs to ensure all registers are empty before generating code for the next basic block, remove all registers from the ADs have the variables we just saved

Here is the algorithm for the getreg function, which we still need to explain. Given some variable x used an a 3AC like ’x = y op z’, the getreg function finds a place (preferably a register) where to store the value of x. The function works as follows:

- if y is already in a register L and y is not used again after the operation, then return L. In other words, if, after using y, we don’t need y anymore, then we can just re-use the register of y for the new value x.

- if there is an empty register L, then return L. In other words, if we cannot re-use a register in the previous step, then we use a new one.

- if all registers are currently full, select one register L, save its content into memory, and return L. In other word, if there is no room left in the registers, then make some. This is called doing a "spill". The hard part is to decide which register to select:
  - a register can be selected randomly, but then the generated code might not be very good, because if the value that was spilled into memory is used very often in the program, then we want to keep it into a register, not save it into memory.
  - the best possible solution is to select the register which is going to be used again the furthest away in the future, i.e. the register that contains the value that is going to be idle for the longest time in the future. The problem is that the compiler cannot predict the future and therefore cannot find the best possible solution. Instead the compiler is going to assume that looking at the past is a good way to predict the future, and, instead of trying to select the register that is going to be used again the furthest away in the future, the compiler will select the register that has not been used for the longest time in the past (i.e. the compiler assumes that, if a value in a register has not been used for a long time in the past, then it is not going to be used for a long time in the future). This is called the "least recently used" strategy, and it gives fairly good results in practice (this is the same "least recently used" algorithm we used in the operating system class when we needed to select a memory page to move from RAM to the hard disk when the memory was full).

- if no register can be spilled (i.e. all the current values in all the registers are really needed very soon) then select a memory location L for y, instead of selecting a register, and return L.

This last step is of course not allowed in a RISC computer, only in a CISC computer, since in a RISC machine the value of the variable y has to be in a register before it can be used in some computation. This means that a compiler for a RISC machine always has to find a register to store a value, so such a compiler has to be much better at doing register allocation than a compiler for a CISC machine has to be (in fact, as we said above, a compiler for a CISC machine can even entirely skip register allocation). In essence CISC and RISC computers use different trade-offs: in a CISC computer the microprocessor is complex (because it needs the be able to deal with complex addressing modes) but register allocation in the compiler can be done easily (it can even be completely ignored). In a RISC machine the microprocessor is simpler (because it needs only to be able to deal with registers) but register allocation in the compiler therefore has to do a much better job. The transition from CISC to RISC thus corresponds to moving some complexity from the hardware (microprocessor) into the software (compiler).

Example. Consider the source code ’d = (a - b) + (a - c) + (a - c)’. The corresponding 3AC basic block is then:

```
t0 = a - b
t1 = a - c
t2 = t0 + t1
d = t2 + t1
```

Assume that we have a RISC microprocessor with three registers, that initially a, b, and c are in memory, that all three variable have integer types, and that we need to compute d because we needs its value later in the program. We then generate assembly code as follows:

Code	RD	AD

At the start of the basic block all the registers are empty and all the variables in memory:

	R0 empty	a in memory
	R1 empty	b in memory
	R2 empty	c in memory

We now start generating the code for: t0 = a - b

We call getreg to determine the location L of the result t0. All registers are empty so getreg cannot re-use a register. Instead getreg returns one of the empty registers, for example R0. We look at the AD of a and then we know that a is only in memory, so a is not already in R0, therefore we output a MOV instruction to move a into R0, and we update the RD for R0 and the AD for a:

```
MOV a, R0        R0 contains a       a in memory and R0
                 R1 empty            b in memory
                 R2 empty            c in memory
```

We look at the AD of b and then we know that b is only in memory, so we call getreg to get a register for b, getreg cannot re-use R0 since we need a again later, so getreg returns a free register, for example R1, we output a MOV instruction to move b into R1, and we update the RD for R1 and the AD for b:

```
MOV b, R1        R0 contains a       a in memory and R0
                 R1 contains b       b in memory and R1
                 R2 empty            c in memory
```

We do instruction selection using a table: the operator is - and the types of a and b are integer, so we select the instruction SUBI, we output the instruction SUBI R1, R0 (since we know from the ADs that a is in R0 and b is in R1), we then remove R0 from the AD of a (since a has been overwritten in R0 by t0), we remove t0 from all RDs (it wasn't in any RD anyway), add t0 to the RD for R0, and update the AD for t0 to indicate that t0 is now in R0:

```
SUBI R1, R0      R0 contains t0      a in memory
                 R1 contains b       b in memory and R1
                 R2 empty            c in memory
                                     t0 in R0
```

Since b is not used again later and is in register R1, we clear the RD for R1 (which becomes empty) and update the AD for b:

```
SUBI R1, R0      R0 contains t0      a in memory
                 R1 empty            b in memory
                 R2 empty            c in memory
                                     t0 in R0
```

Now we generate the code for the next 3AC instruction: t1 = a - c

We call getreg to determine the location L of the result t1. Register R0 already contains t0, and we need t0 again later, so getreg cannot re-use that register. Instead getreg returns one of the empty registers, for example R1. We look at the AD of a and then we know that a is only in memory, so a is not already in R1, therefore we output a MOV instruction to move a into R1, and we update the RD for R1 and the AD for a:

```
MOV a, R1        R0 contains t0      a in memory and R1
                 R1 contains a       b in memory
                 R2 empty            c in memory
                                     t0 in R0
```

We look at the AD of c and then we know that c is only in memory, so we call getreg to get a register for c, getreg cannot re-use R0 or R1 since we need them again later, so getreg returns the last free register R2, we output a MOV instruction to move c into R2, and we update the RD for R2 and the AD for c:

```
MOV c, R2        R0 contains t0      a in memory and R1
                 R1 contains a       b in memory
                 R2 contains c       c in memory and R2
                                     t0 in R0
```

We do instruction selection using a table: the operator is - and the types of a and c are integer, so we select the instruction SUBI, we output the instruction SUBI R2, R1 (since we know from the ADs that a is in R1 and c is in R2), we then remove R1 from the AD of a (since a has been overwritten in R1 by t1), we remove t1 from all RDs (it wasn't in any RD anyway), add t1 to the RD for R1, and update the AD for t1 to indicate that t1 is now in R1:

```
SUBI R2, R1      R0 contains t0      a in memory
                 R1 contains t1      b in memory
                 R2 contains c       c in memory and R2
                                     t0 in R0
                                     t1 in R1
```

Since c is not used again later and is in register R2, we clear the RD for R2 (which becomes empty) and update the AD for c:

```
SUBI R2, R1      R0 contains t0      a in memory
                 R1 contains t1      b in memory
                 R2 is empty         c in memory
                                     t0 in R0
                                     t1 in R1
```

Then we generate the code for the next 3AC instruction: t2 = t0 + t1

We call getreg to determine the location L of the result t2. Register R0 already contains t0 and we do not need t0 again later, so getreg can re-use the register R0 for t2. Since t0 is already in R0 there is no need to output any MOV instruction for t0.

We look at the AD of t1 and then we know that t1 is already in register R1, so there is no need to output any MOV instruction for t1.

We do instruction selection using a table: the operator is + and the types of t0 and t1 are integer, so we select the instruction ADDI, we output the instruction ADDI R1, R0 (since we know from the ADs that t1 is in R1 and t0 is in R0), we then remove R0 from the AD of t0 (since t0 has been overwritten in R0 by t2) which means that t0 has now disappeared, we remove t2 from all RDs (it wasn't in any RD anyway), add t2 to the RD for R0, and update the AD for t2 to indicate that t2 is now in R0:

```
ADDI R1, R0      R0 contains t2      a in memory
                 R1 contains t1      b in memory
                 R2 is empty         c in memory
                                     t1 in R1
                                     t2 in R0
```

Then we do the code generation for the last 3AC instruction: d = t2 + t1

We call getreg to determine the location L of the result d. Register R0 already contains t2, and we do not need t2 again later, so getreg can re-use the register R0 for d. Since t2 is already in R0 there is no need to output any MOV instruction for t2.

We look at the AD of t1 and then we know that t1 is already in register R1, so there is no need to output any MOV instruction for t1.

We do instruction selection using a table: the operator is + and the types of t2 and t1 are integer, so we select the instruction ADDI, we output the instruction ADDI R1, R0 (since we know from the ADs that t1 is in R1 and t2 is in R0), we then remove R0 from the AD of t2 (since t2 has been overwritten in R0 by d) which means that t2 has now disappeared, we remove d from all RDs (it wasn't in any RD anyway), add d to the RD for R0, and update the AD for d to indicate that d is now in R0:

```
ADDI R1, R0      R0 contains d       a in memory
                 R1 contains t1      b in memory
                 R2 is empty         c in memory
                                     d in R0
```

t1 in R1

Since t1 is not used again later and is in register R1, we clear the RD for R1 (which becomes empty) and update the AD for t1, which means that t1 has now disappeared:

```
ADDI R1, R0      R0 contains d       a in memory
                 R1 is empty        b in memory
                 R2 is empty        c in memory
                                    d in R0
```

We are then done generating the assembly code for all the 3AC. Since we have reached the end of the basic block, we now have to clear all the registers so that they are all empty when we start generating the code for the next basic block. So we look at the RDs and determine that d is currently in R0, so we might have to save d into memory. Since we assumed that we need the value of d later in the program, we then know that in fact we do have to save d in memory (if d were not re-used later, then we could just forget about d). We then look at the AD of d to check whether d is already in memory or not. If d is already in memory then there is no need to save R0. In the present case the AD for d tells us that d in only in register R0, so we do indeed have to save d. To do that we output a MOV instruction to save R0 somewhere in memory (at the address which we computed for d when we computed d’s size and offset in the static area of the program, during the intermediate code generation phase of the compiler), and we then update the AD of d to reflect the copy in memory:

```
MOV R0, d        R0 contains d       a in memory
                 R1 is empty        b in memory
                 R2 is empty        c in memory
                                    d in memory and R0
```

Finally we can clear all the RDs to ensure all registers are empty, and remove R0 from the variable d which we just saved in memory:

```
                 R0 is empty        a in memory
                 R1 is empty        b in memory
                 R2 is empty        c in memory
                                    d in memory
```

And we are done with generating the assembly code for the whole basic block. Here is the complete code:

```
MOV a, R0
MOV b, R1
SUBI R1, R0
MOV a, R1
MOV c, R2
SUBI R2, R1
ADDI R1, R0
ADDI R1, R0
MOV R0, d
```

We can then start generating the assembly code for other basic blocks, and at the end we will just glue all the different pieces of assembly code back together using microprocessor instructions for gotos.

There are two more things to note here:

- the code we generated is not optimal. For example, we first move the value of a from memory into R0, then overwrite R0 with t0, then move the value of a again from memory into R1. It would be faster to keep the value of a around in a register rather than having to copy it from memory twice.

- we assumed that our microprocessor has three registers, but in fact we can generate code that uses only two registers: every time we compute an intermediate result like t0 or t1 we just immediately save that intermediate result back into memory. This means then that we always have two registers available to store the operands of

every operation we need to do. Then when we need t0 or t1, we just move them back into a register from memory. The drawback of course is that we have to spend a lot more time moving values to and from memory. So, while it's possible to generate code for a microprocessor that has only two registers, in practice the more registers we have, the better.

Compiler Bootstrapping (Optional Section)

To finish, we have to look at one last interesting problem: how to compile the compiler itself. For example, if we write a Java compiler in Java, then we cannot execute directly this compiler, since the compiler is written in Java and the microprocessor can only execute microprocessor instructions. So before we can use this compiler, we first have to compile it into an executable program (an executable compiler) which we can then use to compile other program. This is not as easy as it seems though...

To describe the problem, we are going to represent a compiler using what is called a T-diagram:

```
X ----> Y
    |
    Z
```

Here X is the source language that the compiler accepts as input, Y is the target language that the compiler produces as output, and Z is the language which is used to implement the compiler itself. For example, this T-diagram:

```
Java ----> i386
       |
       C
```

describes a compiler that takes Java as input language (i.e. the compiler compiles Java source programs), that produces as output microprocessor instructions for the Intel 386 family of microprocessors (80386, 80486, Pentium, etc), and is implemented using the C language (i.e. the compiler itself is written in C).

Sometimes we will also give a number to each compiler, to make it easy to recognize the various compilers that we are talking about:

```
X ----> Y
    |3
    Z
```

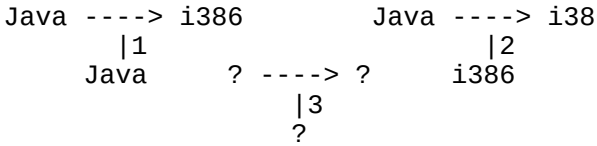
To see the problem with compiling a compiler, let's look at a compiler for the Java language that is written in Java and generates code for Intel microprocessors:

```
Java ----> i386
       |1
     Java
```

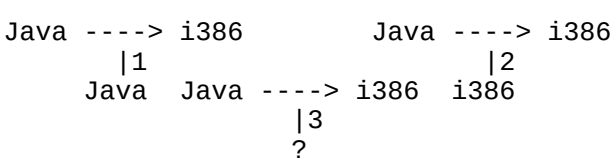
This compiler cannot be executed, since it is written in Java, and only programs written using microprocessor instructions can be executed by a microprocessor. What we really want is therefore an executable Java compiler that we can run on our Intel machine:

```
Java ----> i386
       |2
     i386
```

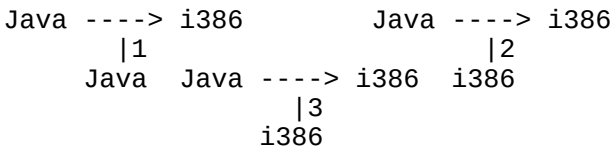
So how do we go from our Java compiler 1 written in Java to an executable Java compiler 2? Well, we have to use another compiler 3 to compile our Java compiler 1 written in Java into the executable Java compiler 2 that we want:



The compiler 3 needs to be able to read the Java source code of compiler 1, and it needs to produce i386 microprocessor instructions so that the compiler 2 generated by compiler 3 is indeed an executable compiler:



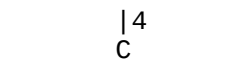
What implementation language do we use for compiler 3? Since we need to execute compiler 3 in order to compile compiler 1, compiler 3 needs to be itself written using i386 instructions:



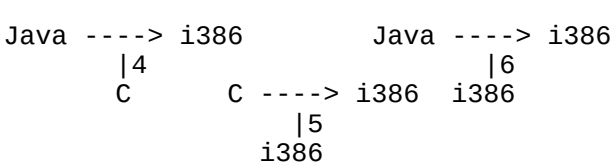
We now have a problem: to generate compiler 2 we need to have compiler 3, but if you compare compiler 2 and compiler 3 you can see that they are the same: same input language, same output language, same implementation language! This means that, to generate compiler 3 we already need to have a compiler which is exactly like compiler 3! In short, to create compiler 3 we already need to have compiler 3. It's like saying that, to build a house, the first thing you need is to have the same house already built! So we are stuck, and trying to generate compiler 3 using this method is never going to work.

Instead we have to use some other methods. The first one is to use two different compilers for Java, first one written in C to create the first Java compiler, then one in Java to create future Java compilers.

So we start with a Java compiler 4 written in C:

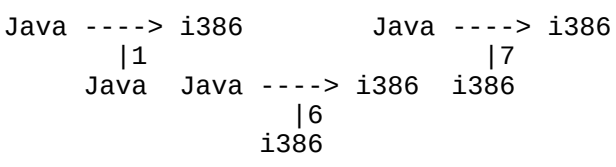


We then use an executable C compiler 5 to compile compiler 4, to get an executable Java compiler 6:



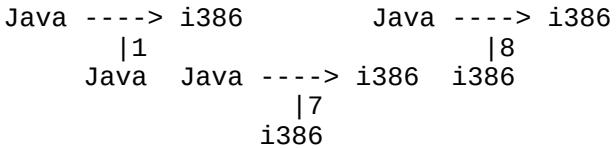
We know that the compiler 6 takes Java as input and produces i386 instructions as output because that's what compiler 4 does and compiler 5 preserve the meaning of compiler 4 when it translates compiler 4 into compiler 6.

Once we have done that we have an executable compiler for Java. We can then take the source code of our Java compiler 1 written in Java and use compiler 6 to generate a new executable Java compiler 7:



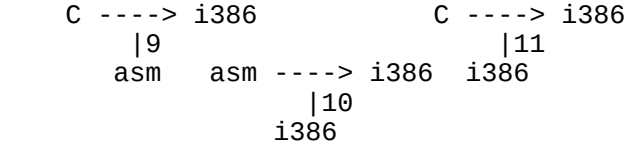
Note that compilers 6 and 7 are both executable Java compilers, but compiler 6 was created by the C compiler 5 starting from the C source code of compiler 4, while compiler 7 was created by compiler 6 starting from the Java source code of compiler 1. While it is not mandatory to create compiler 7 (since we could always use compiler 6 instead) in general it is nice to have the compiler for some programming language written in the programming language itself (like compiler 1, which is a Java compiler written in Java). It is usually a good test for the programming language (since a compiler like compiler 1 written in its own language is usually one of the first programs written when a new programming language is created) and it makes it easier for users of the compiler to modify the compiler itself (i.e. the Java users of compiler 7 understand Java (since they use compiler 7 to compile their own Java programs) so it easier for them to modify the Java source code of compiler 1 (and then re-recompile it into a new version of compiler 7) rather than modify the C source code of compiler 4).

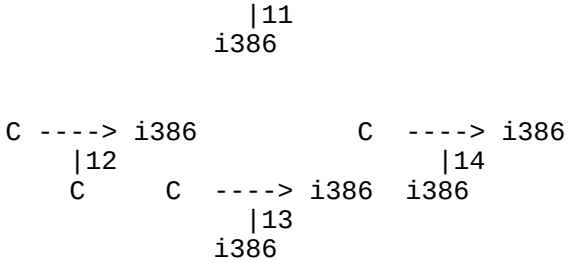
Once we have done this, it is often a good idea to recompile compiler 1 using the executable version of itself (which is the compiler 7 created in the previous step when we compiled compiler 1 into compiler 7):



Again this is not strictly necessary, but it is a good idea in practice: since both executable Java compilers 7 and 8 are generated from the same Java source code of compiler 1, we expect the two compilers 7 and 8 to be exactly the same (bit by bit). If compilers 7 and 8 are the same, then it is a good indication (but not a guarantee) that executable compiler 7 is correct, therefore that executable compiler 6 is correct too (since it was used to create compiler 7) and therefore that compiler 1 (the Java compiler written in Java) and compiler 4 (the Java compiler written in C) produce the same code (since compiler 7 was generated from compiler 1 and compiler 6 was generated from compiler 4). If compiler 7 and compiler 8 are not the same, then the same Java source code of compiler 1 was compiled by compilers 6 and 7 into two different executable compilers 7 and 8, which probably means that either compiler 4 (the Java compiler written in C) or compiler 1 (the Java compiler written in Java), or both of them, contains bugs. So recompiling compiler 1 using compiler 7 (which was generated from compiler 1) to get compiler 8 is a good way to detect bugs in compilers 1 or 4.

The next question of course is: where does the executable C compiler 5 come from? If that compiler was originally written in C, then again we have a problem since, to compile a C compiler written in C and generate the corresponding executable C compiler, you already need to have an executable C compiler. No problem: we can just use the same trick again: first write a C compiler 9 in assembly language, compile that C compiler 9 into an executable C compiler 11 using an assembly code compiler 10 (i.e. an assembler program 10), then rewrite the C compiler 9 in C to get the compiler 12, compile compiler 12 using executable compiler 11 to get a new executable compiler 13, then, just to double check for bugs, recompile compiler 12 using compiler 13 to get a new executable C compiler 14, and check whether executable compilers 13 and 14 are the same:





We can then use either compiler 11, compiler 13, or compiler 14 to play the role of compiler 5 above.

The next question of course is: where does the executable assembly compiler 10 come from? If that compiler was originally written in assembly language then again we have a problem since, to compile an assembly language compiler written in assembly language and generate the corresponding executable assembly language compiler, you already need to have an executable assembly language compiler.

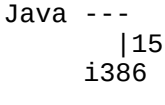
This time we cannot use another compiler for assembly language written in some other programming language though, because there is no compilable programming language lower than assembly language. The solution is then to create compiler 10 by manually writing the binary i386 microprocessor instructions for compiler 10 directly into memory, bit by bit. Indeed this is how the first assembler program was written (well, it was not written directly in memory, it was written on punch cards and the bits from the punch cards were then loaded into the memory of the computer).

Once we have compiler 10, then we can rewrite the assembler program in assembly language and re-compile it, then use the result to re-compile itself (just like what we did above with C and Java, for extra checking), then use the resulting compiler as compiler 10.

In fact you can still do the whole process today on your own computer: write an executable assembly language compiler bit by bit, then rewrite the assembly language compiler in assembly language, compile it and then re-compile it using itself, write a C compiler in assembly language, use the executable assembly language compiler to get an executable C compiler, then rewrite the C compiler in C, compile it using the executable C compiler created from the C compiler written in assembly language and then re-compile it using itself, write a Java compiler in C, use the executable C compiler to get an executable Java compiler, then rewrite the Java compiler in Java, compile it using the executable Java compiler created from the Java compiler written in C then re-compile it using itself, etc...

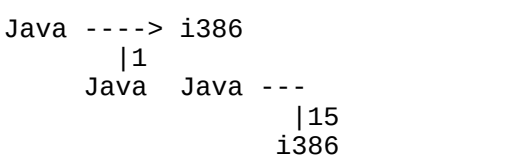
What we are doing then is climbing step by step up a ladder of more and more powerful programming languages: binary i386 microprocessor instructions, assembly language, C, then Java, etc. This is what is called compiler bootstrapping (named after the idea of moving yourself up in the air by pulling on the straps of your boots...)

The second way to solve our bootstrapping problem for our Java compiler 1 written in Java is to, instead of using a Java compiler 4 written in C, just use a Java interpreter. An interpreter does not have a back end (it directly executes the source program itself without generating a target program) so it is often easier to write than the equivalent compiler. Here we represent an interpreter using a "corner" diagram:

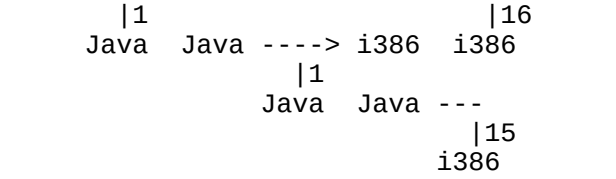


This represents an interpreter that takes Java code as source language and is implemented using i386 microprocessor instructions. The interpreter 15 then directly executes the Java (i.e. interpreter 15 is a Java Virtual Machine that executes Java code (not Java bytecode as

the usual JVM does)). We can the use interpreter 15 to run compiler 1 on it:



Once our Java compiler 1 (written in Java) is running on top of our Java interpreter 15, we can use the Java compiler 1 to compile itself to get an executable Java compiler 16:

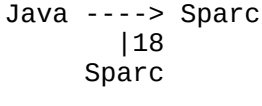


We can then use executable compiler 16 to re-compile compiler 1 to get another executable compiler 17, then compare compilers 16 and 17 bit by bit to try to find bugs in our compilation process...

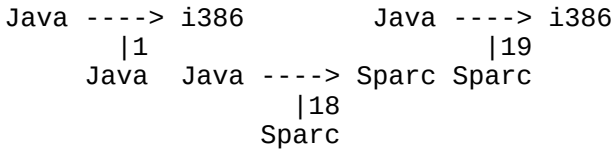
The question then is: where does executable Java interpreter 15 come from. Well, it has to be written in C, then compiled using a C compiler. Later, once compiler 16 has been generated, we can re-write the Java interpreter in Java and then use compiler 16 to compile it. So even when we use interpreters we still have to do some bootstrapping for the interpreters, but, as we said above, writing interpreters is often simpler than writing compilers.

The third (and last) way to solve our bootstrapping problem for our Java compiler 1 written in Java is to, instead of using a Java compiler written in another language, use a Java compiler on another machine.

Suppose we have for example an executable Java compiler 18 that runs on a Sparc machine (from Sun Microsystems, now Oracle) and that generates microprocessor instructions for such Sparc machine:

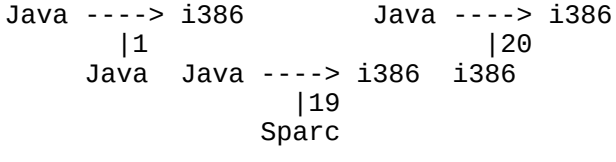


We can take our Java compiler 1 written in Java which is on our Intel machine, transfer the code to the Sparc machine, and compile it using the executable Java compiler 18 on the Sparc machine:



What we get is the strange compiler 19: an executable Java compiler that generates target code for the Intel microprocessor but which runs on a Sparc microprocessor. This is what is called a cross-compiler: a compiler that runs on one type of machine and generates target code that can run on another type of machine.

Once we have cross-compiler 19, we can use it to re-compiler compiler 1 on the Sparc machine:



What we then get is an executable Java compiler 20 than generates Intel microprocessor instructions and can run on an Intel machine. The only thing we have to do then is to copy compiler 20 from the Sparc machine back to the Intel machine, and then we can use compiler 20 on the Intel machine to compile Java code (include re-compiling compiler 1 with compiler 20 to get compiler 21, then check compilers 20 and 21 bit by bit to try to detect bugs...)

So when we have an executable compiler for language L for one type of machine M, it is often easy to get an executable compiler for the same language L on another type of machine N without having to do any bootstrapping. Of course, the executable compiler for language L on machine M has to be created using bootstrapping (or has to be created from another executable compiler for language L from another type of machine K). For example, before we can get the executable Java compiler 20 that can run on the Intel machine, we first need to have the executable Java compiler 18 that runs on the Sparc machine, which itself has to be created using bootstrapping or using another Java compiler from another type of machine (neither Intel nor Sparc...) Compilers are a lot of fun!