Implementation Strategies

Torben Mogensen

Programming Language Design

Learning Goals

- A birds-eye view of Compilation and interpretation and implementation methods that combine both
- Diagrams for compiling, interpreting, and executing programs.
- Methods for "bootstrapping" compilers

Motivation

- Computers are built to execute *machine language*, which is ill suited for people to read, write and maintain.
- High-level languages are designed for human use, but should run on actual machines.
- This requires someone to write *implementations* of high-level languages.

This can essentially be done in two ways: By compilation or by interpretation.

Compilation

- Compilation is *translation* of code from one language to another.
- Typically, from a high-level language to a low-level language (such as machine code).
- Similar to translating a song text from English to Danish before singing it.

Advantages:

- Compile once, run many times.
- Optimisations can be applied to make code run at almost "native" speed.

Disadvantages:

- Compilation can be slow.
- Complexity.

Interpretation

- The program syntax is represented (possibly in simplified form) as data in memory, and a program (called an *interpreter*) inspects this data to decide what to do in each step of computation.
- Similar to translating a song text from English to Danish as you sing it.

Advantages:

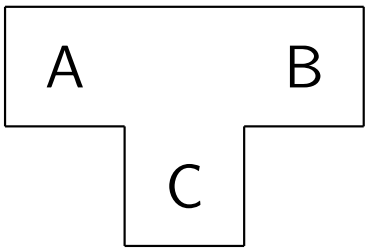
- Fast edit-run cycle, as little or no delay for compilation.
- Can execute run-time generated source code without delay for compilation.

Disadvantage:

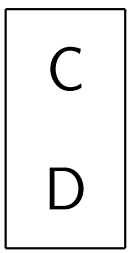
- Slow execution speed.

Notation: T-diagrams (Bratman)

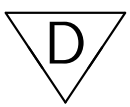
A *compiler* from A to B written in C:



A *interpreter* for C written in D:

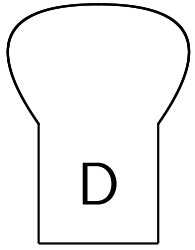


A *machine* for D:

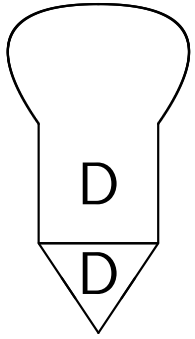


Running a Program

Arbitrary program (compiler, interpreter, or application) in D:

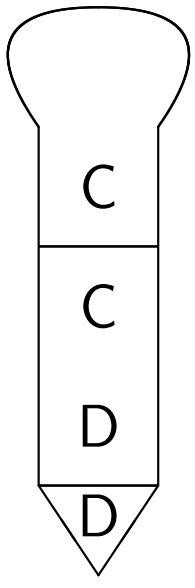


Arbitrary program in D, running on a D-machine:

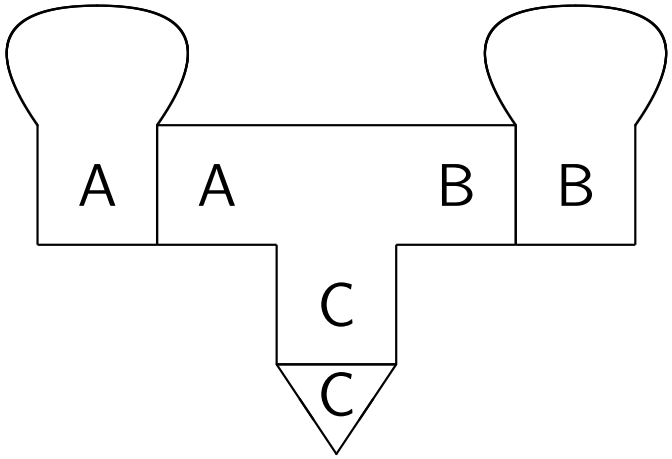


Note that the languages match up.

Interpretation



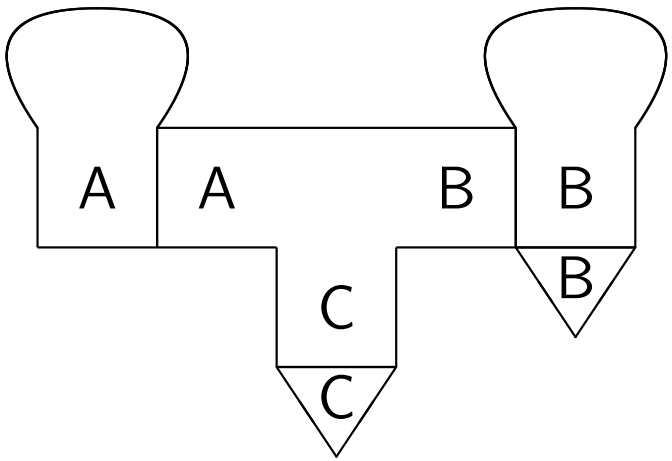
Compilation



Note that *source* and *target* programs (in languages A and B) are not executed – they are just data.

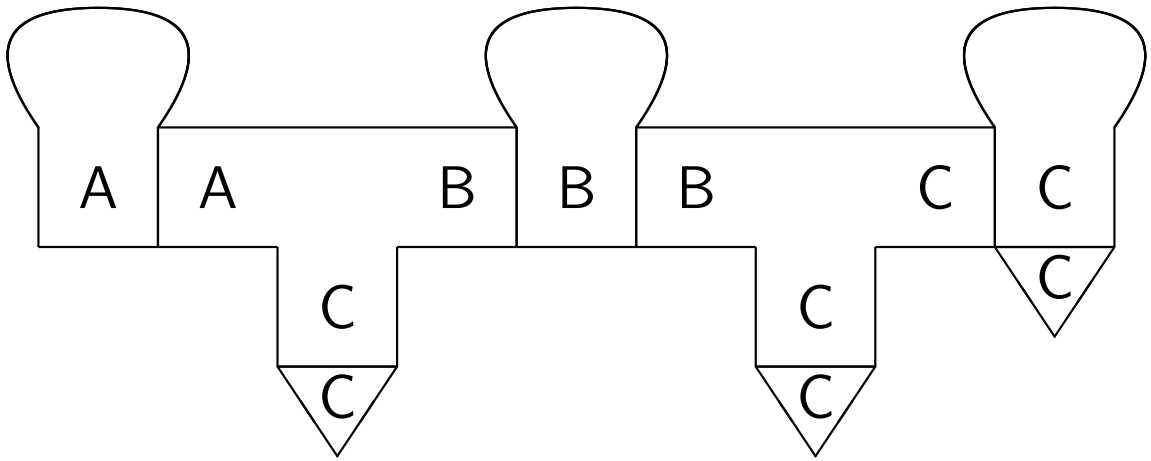
Combining Diagrams

Compilation followed by execution of target program:



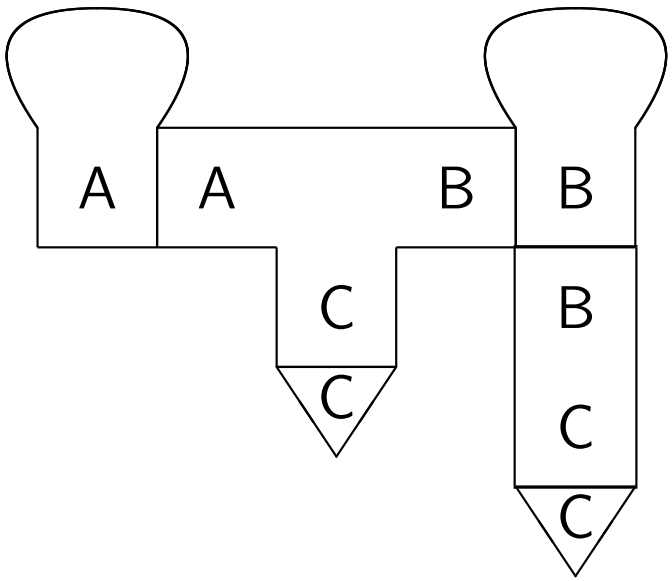
Note that the target program has two rôles: Output from the compiler and program running on B-machine, so this diagram describes two executions.

Compiling Through Intermediate Language



- Same intermediate language can be used for several source and target languages – reduces M×N compilers to M+N compilers.
- Optimisations and simplifications done on B can be shared.

Compiling to Interpreted Intermediate Language



B is often called a *virtual machine*.

Hybrid Methods

- Intermediate language is executed by a combination of interpretation and compilation.
- Common strategy: Interpret the first N times a function is called, then compile.
- Can add third stage: Optimise compiled code after another M calls.
- General assumption: Number of future calls likely to be similar to number of past calls, so amortise compilation and optimisation cost over *expected* future calls.

IDEs and REPLs

To avoid overhead of loading programs from files when they are edited, some interpreters have built-in editors, so you can edit programs in memory and run them without storing in a file. Such *integrated development environments* (IDEs) often offer additional features such as keyword completion, syntax colouring, and "folded" views.

Some interpreters also allow a user to type in expressions and statements that are executed in the context of the current program. Such *read-eval-print loops* (REPLs) were introduced with LISP and are found in many languages such as BASIC, ML, F#, and Python.

While originally associated with interpreters, IDEs and REPLs are now found also for compilers. Compilers are for that reason often made *incremental*: They (re)compile only the parts of the program that are affected by the modifications made since the last compilation.

Cross Compilers and Reverse Compilers

A *cross compiler* compiles from one high-level language to another. Motivation:

- Exploit available compiler for target language.
- Migration of software to another platform.

A *reverse compiler* compiles from a low-level language to a high-level language. Motivation:

- Maintaining or migrating code that is available only in compiled form.
- Reverse-engineering programs from compiled form.

Obfuscation and Water Marking

To prevent code theft, you might want to make a program hard to understand and modify – by other people. To do so, you can make a compiler from a language to the same language that preserves behaviour but limits readability, i.e., an *obfuscator*.

You might additionally want to prove ownership of code, so the obfuscator might add data that encodes ownership in such a way that it is difficult for a software pirate to detect and remove this information. This is called *software water marking*.

Exercise

Before moving on, consider the following:

1. Which is easiest to write, a compiler or an interpreter?
2. Can this depend on the language we want to implement?
3. Can it depend on the language used for implementation?

For all of the above, explain why.

Bootstrapping

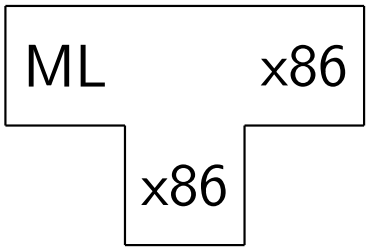
We might want to write a compiler for a language L in L itself. But how do we then get this compiler compiled?

Bootstrapping

We might want to write a compiler for a language L in L itself. But how do we then get this compiler compiled?
We would like the compiler to compile itself, but that sounds like pulling yourself up by your bootstraps or (as Baron von Münchausen) by your hair.

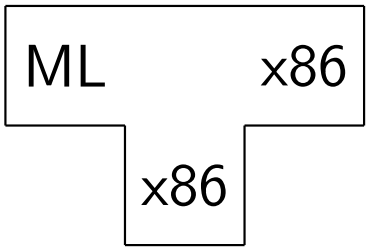
Half bootstrap: when we already have a compiler, but written in the wrong language

Suppose we want an ML-to-x86 compiler written in x86:

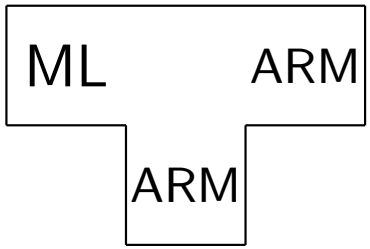


Half bootstrap: when we already have a compiler, but written in the wrong language

Suppose we want an ML-to-x86 compiler written in x86:

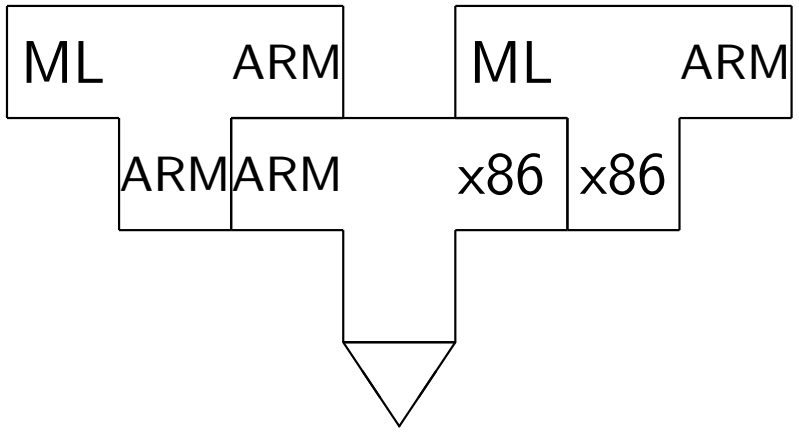


We have the following tools:



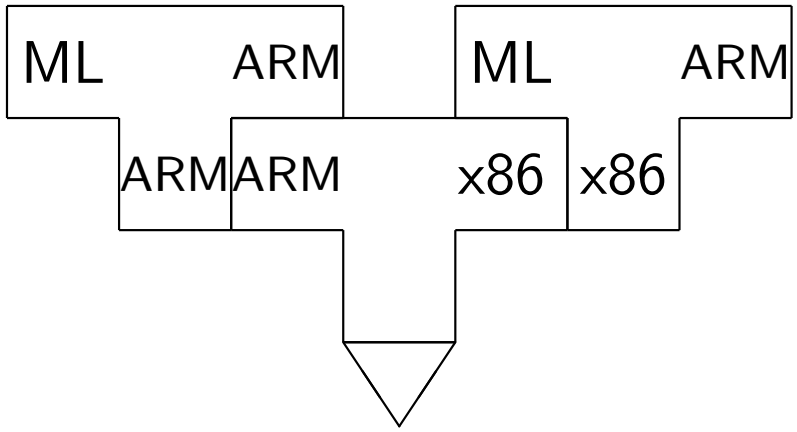
First Try: Binary Translation

Write ARM-to-x86 compiler to compile our compiler:

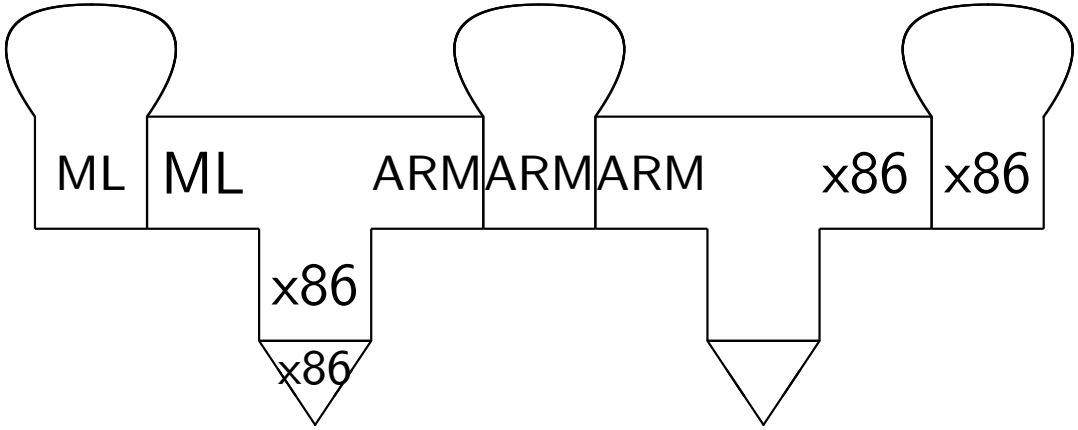


First Try: Binary Translation

Write ARM-to-x86 compiler to compile our compiler:

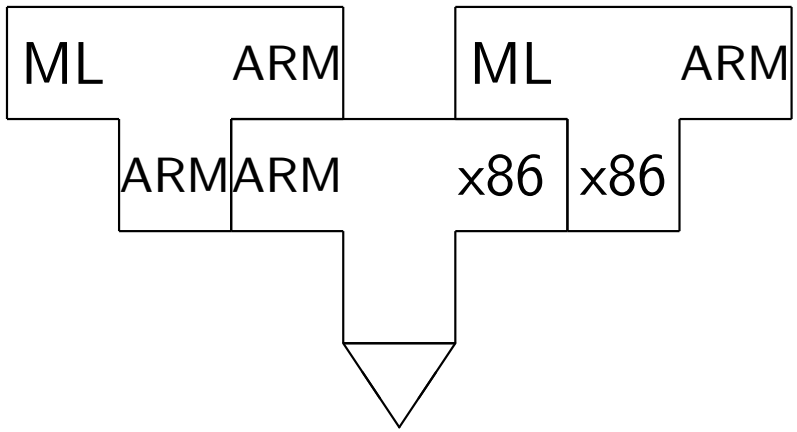


And compile the output of this from ARM to x86:

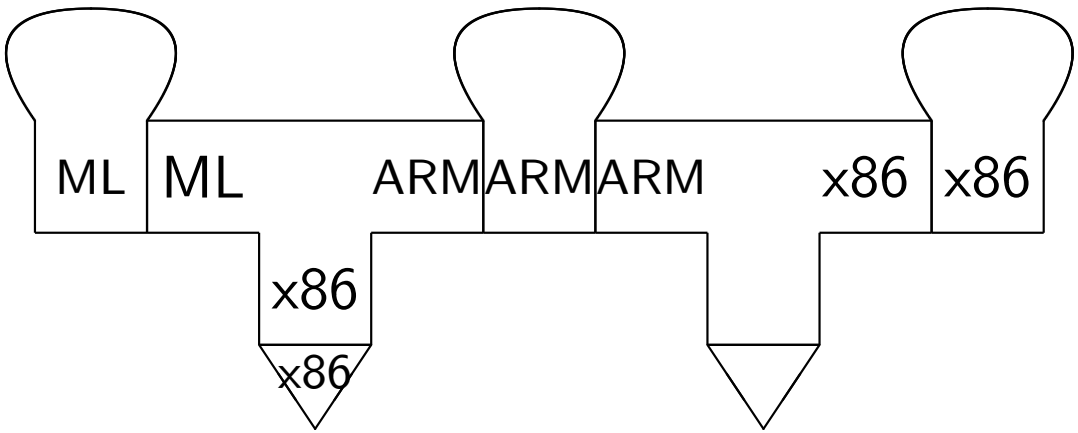


First Try: Binary Translation

Write ARM-to-x86 compiler to compile our compiler:



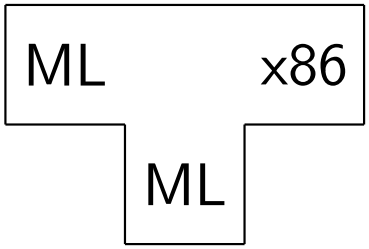
And compile the output of this from ARM to x86:



- ML-to-x86 is a 2-pass compilation: inefficient.
- In which language do we write ARM-to-x86?

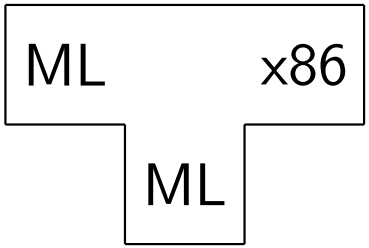
Second Try: ML Compiler in ML

Write ML-to-x86 compiler in ML.

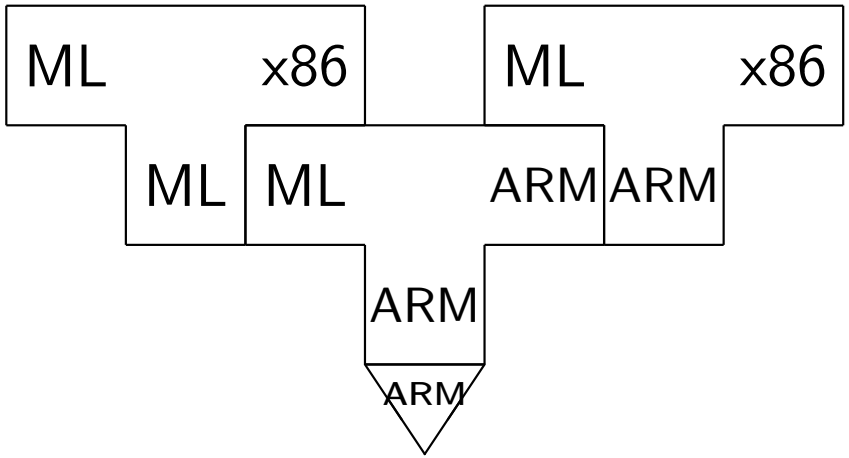


Second Try: ML Compiler in ML

Write ML-to-x86 compiler in ML.

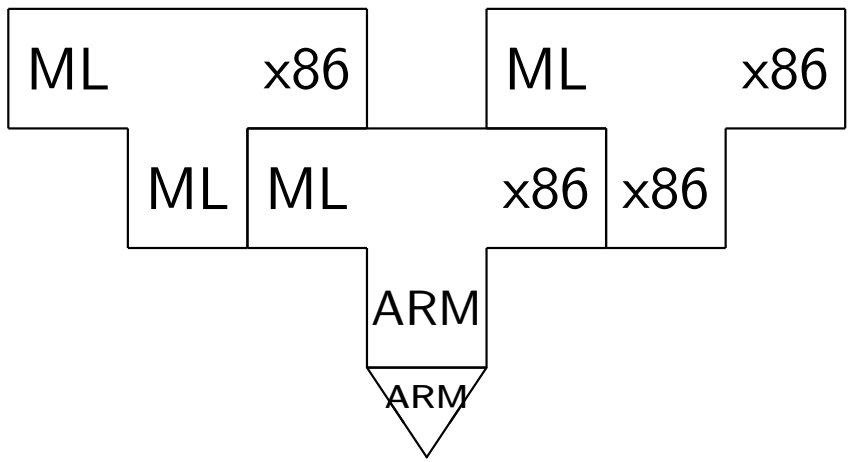


Compile with ML-to-ARM.

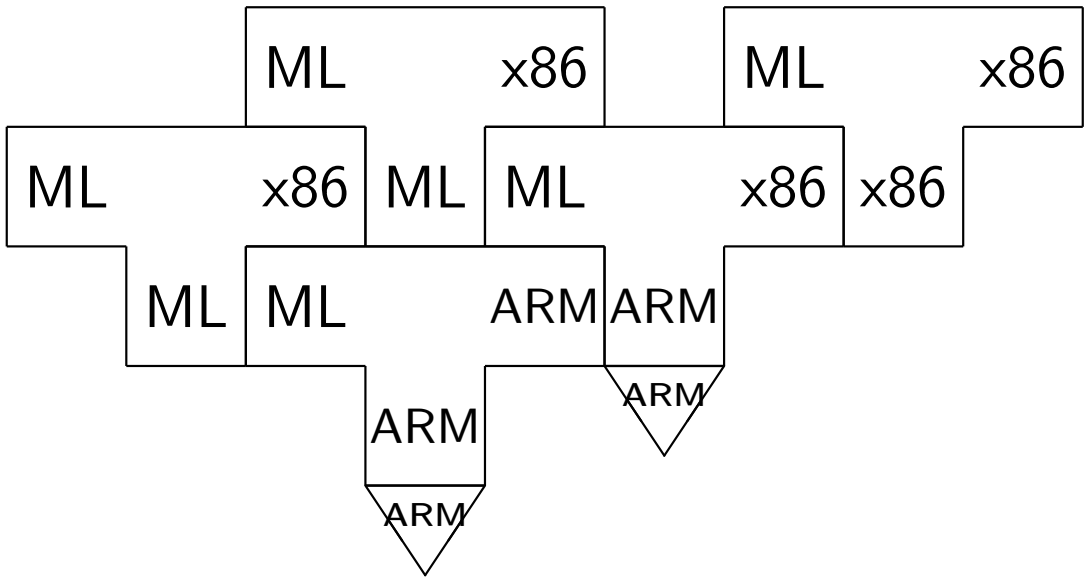


Step 2: Bootstrap

Compile *again* using the compiler produced in first step.



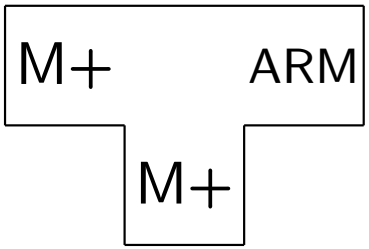
Half Bootstrap in one Diagram



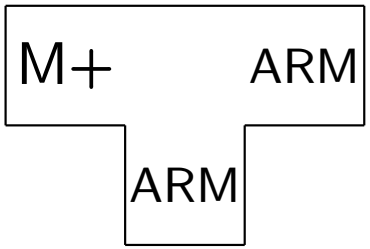
(Note the x86/ML ambiguity—not a mismatch.)

Full bootstrap: when we have no running compiler for the source language

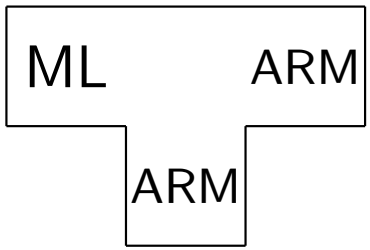
We have a fancy new language M+, and want a flashy M+-to-ARM compiler (written in M+) to run on ARM.



but want

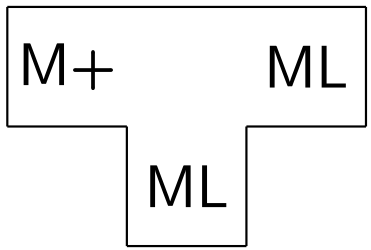


Again, with the following tools:



Full bootstrap with QAD compiler

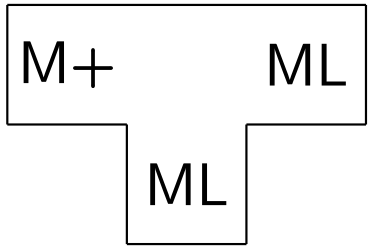
Quick-and-dirty: Write M+-to-ML compiler in ML



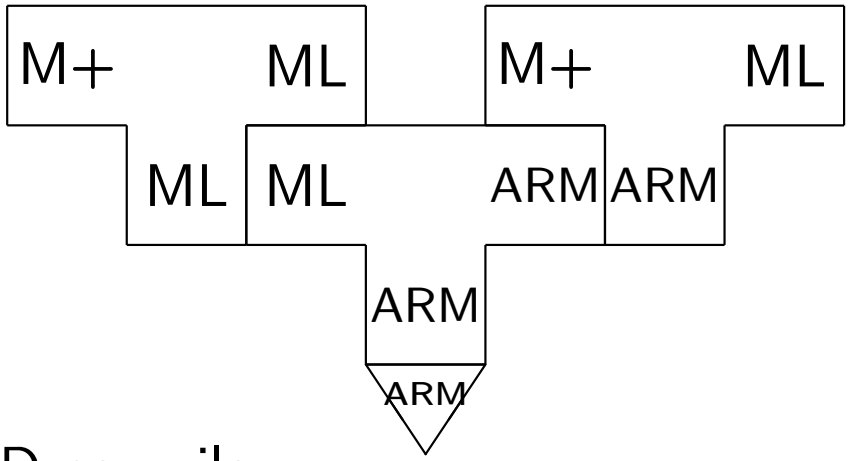
Key point: only used to make M+ programs executable. Other qualities (speed, target optimisation, etc.) *do not matter*.

Full bootstrap with QAD compiler

Quick-and-dirty: Write M+-to-ML compiler in ML

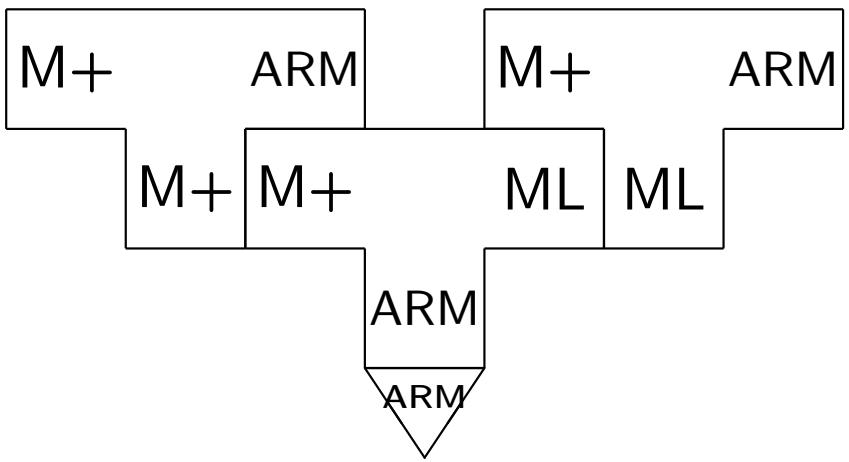


Key point: only used to make M+ programs executable. Other qualities (speed, target optimisation, etc.) *do not matter*.



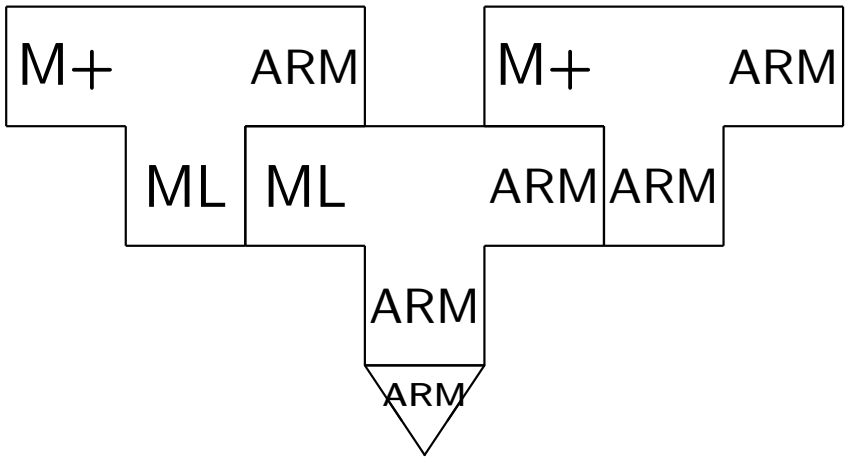
1. Compile QAD compiler

Full bootstrap with QAD compiler



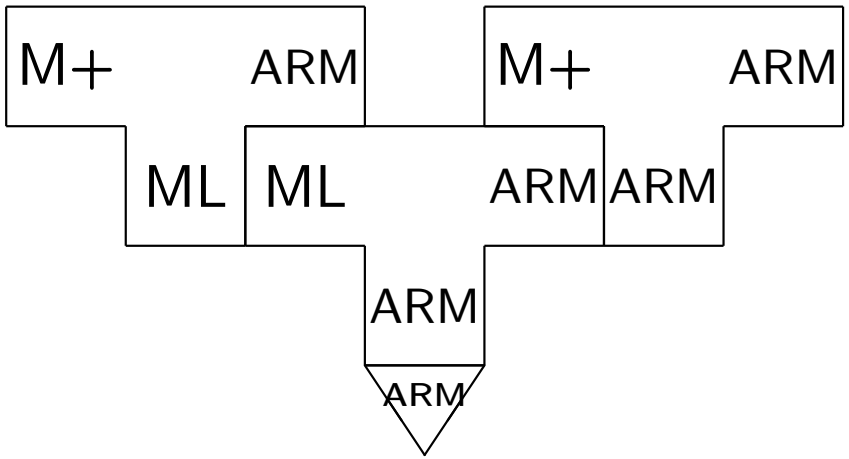
2. Compile ‘real’ compiler using QAD compiler

Full bootstrap with QAD compiler



3. Compile from ML to ARM using the existing ML compiler

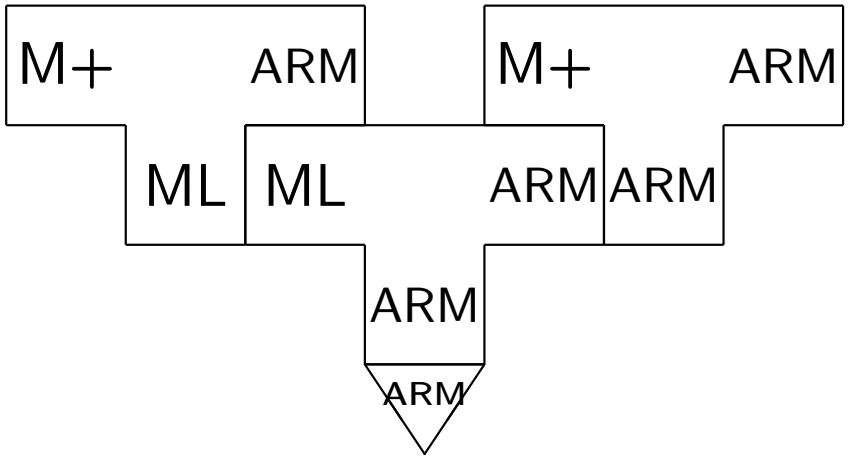
Full bootstrap with QAD compiler



3. Compile from ML to ARM using the existing ML compiler

Are we done?

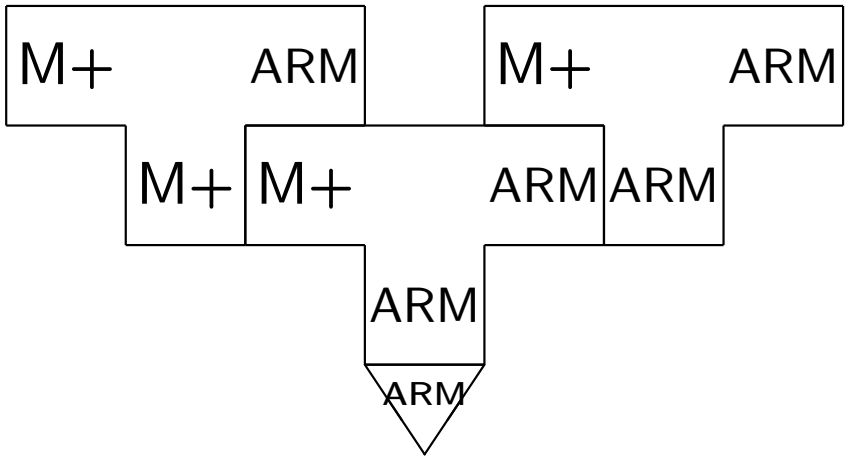
Full bootstrap with QAD compiler



3. Compile from ML to ARM using the existing ML compiler

Are we done? *No. Speed (but not functionality) of M+ to ARM compiler depends on QAD and ML-to-ARM compiler quality.*

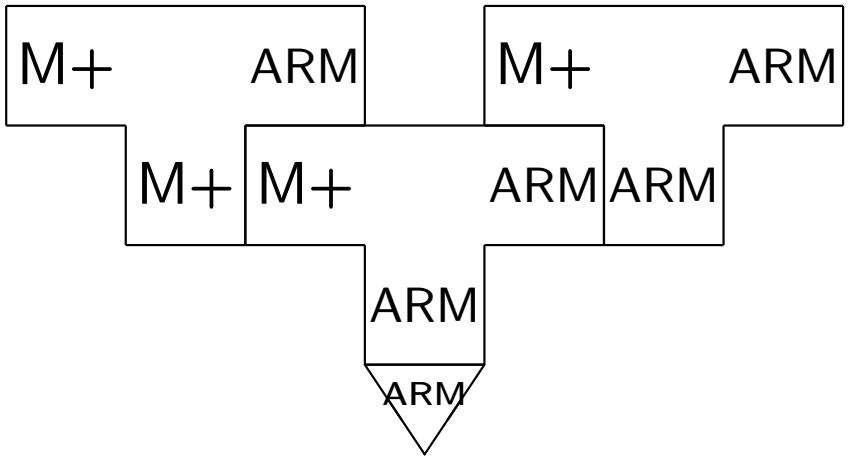
Full bootstrap with QAD compiler



4. Recompile M+ to ARM compiler using (translated version of) itself

(Because semantic functionality is not enough.)

Full bootstrap with QAD compiler

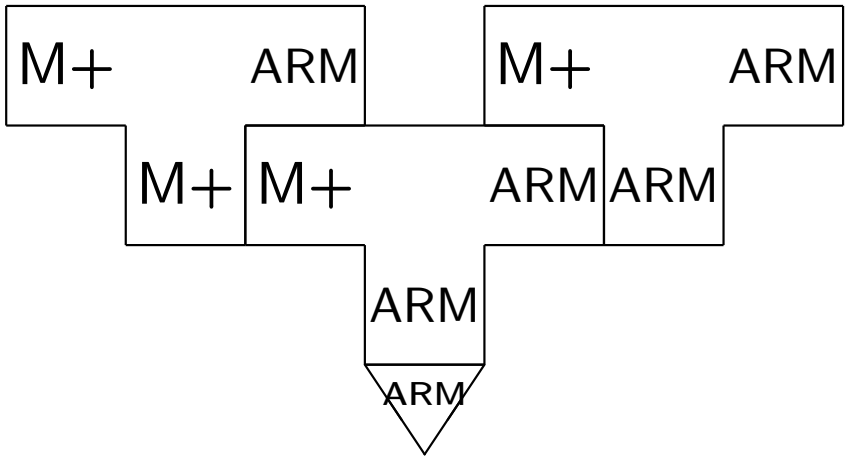


4. Recompile M+ to ARM compiler using (translated version of) itself

(Because semantic functionality is not enough.)

Q: What happens if we do another recompilation?

Full bootstrap with QAD compiler



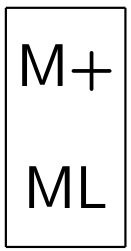
4. Recompile M+ to ARM compiler using (translated version of) itself

(Because semantic functionality is not enough.)

Q: What happens if we do another recompilation?
A: We get *exactly* the same compiler!

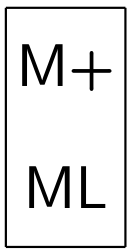
Full bootstrap with QAD interpreter

Above, the QAD M+-to-ML compiler is only used to make programs executable. Interpreters can do that. Write QAD M+-interpreter in ML:

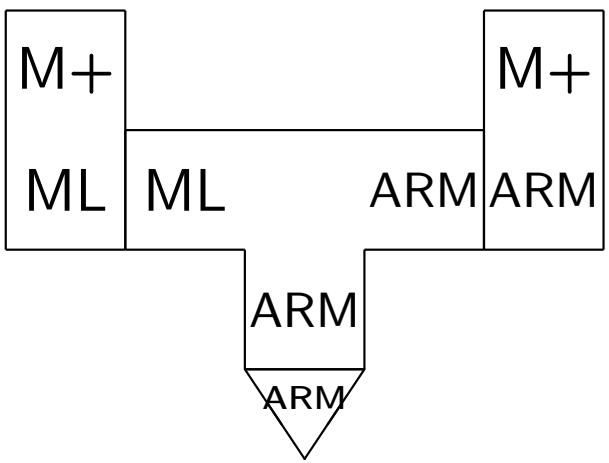


Full bootstrap with QAD interpreter

Above, the QAD M+-to-ML compiler is only used to make programs executable. Interpreters can do that. Write QAD M+-interpreter in ML:

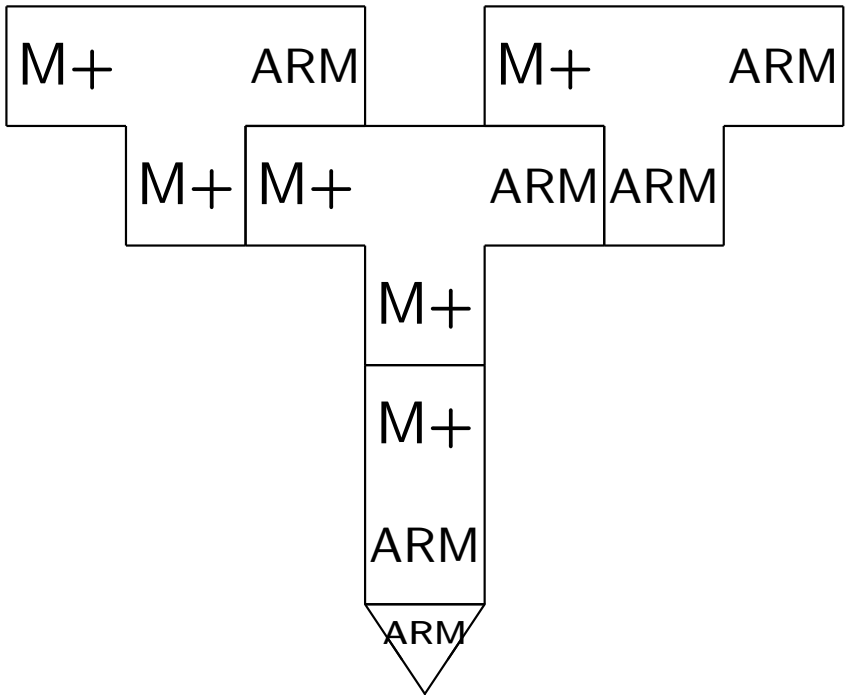


Translate to ARM:



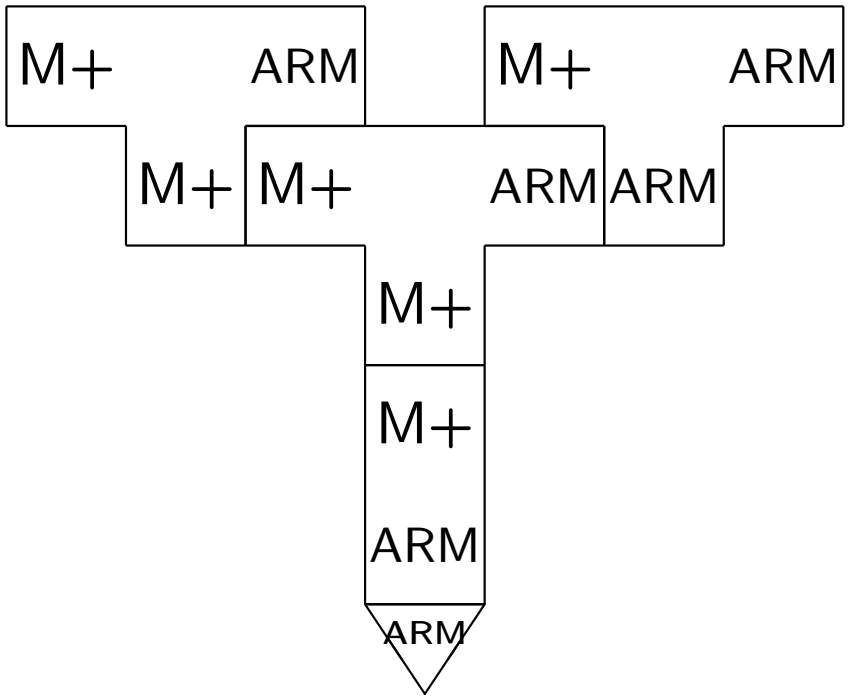
Full bootstrap with QAD interpreter

Bootstrap.



Full bootstrap with QAD interpreter

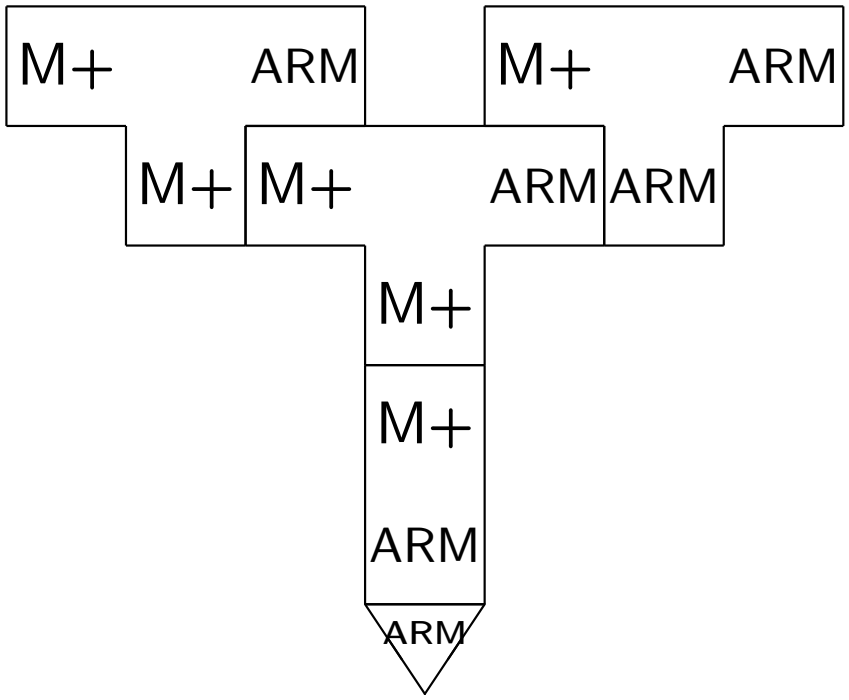
Bootstrap.



Q: Are we done?

Full bootstrap with QAD interpreter

Bootstrap.

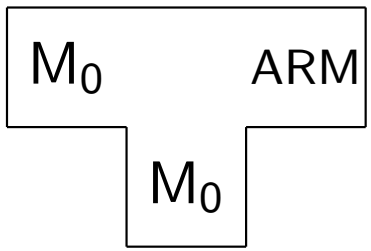


Q: Are we done? (Yes.)

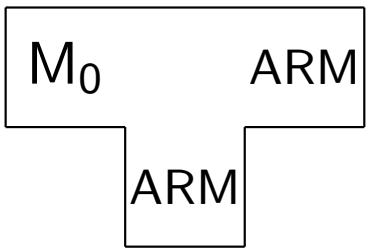
Incremental bootstrapping

You can go slowly when you develop a language and its compiler.

Bootstrap a compiler for a "base" language M_0, a subset of M+.

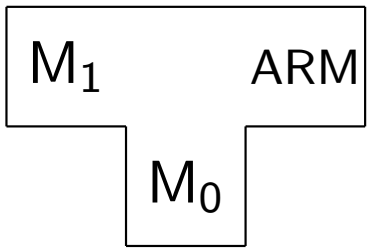


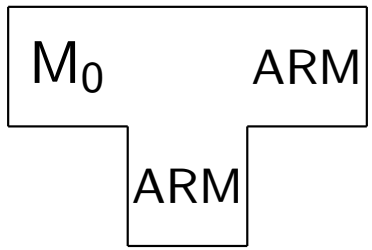
bootstrap to



Incremental bootstrapping

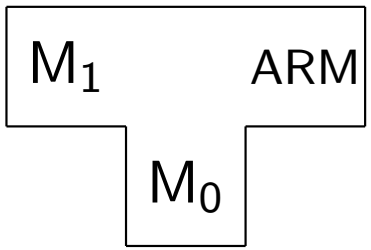
Key idea: Extend the compiler *source language* from M_0 to M_1, but keep the *platform* as M_0. We have:

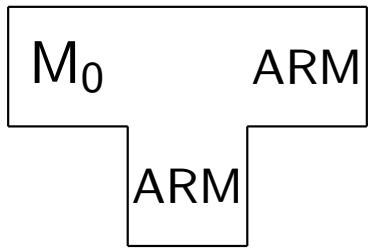




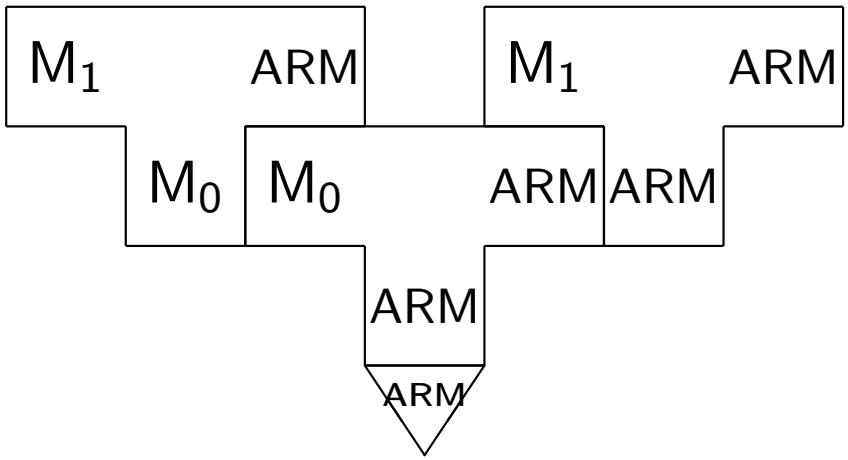
Incremental bootstrapping

Key idea: Extend the compiler *source language* from M_0 to M_1, but keep the *platform* as M_0. We have:



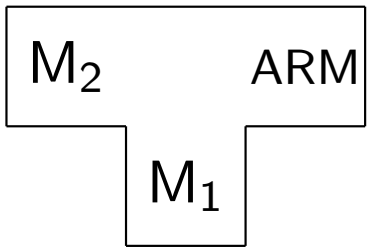


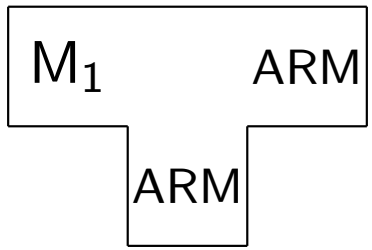
Compile with the M_0 compiler:



Incremental bootstrapping

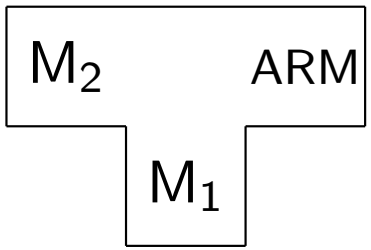
Extend the compiler source *again* from M_1 to M_2. Now, the platform can be M_1. We have:

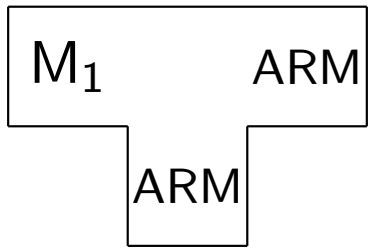




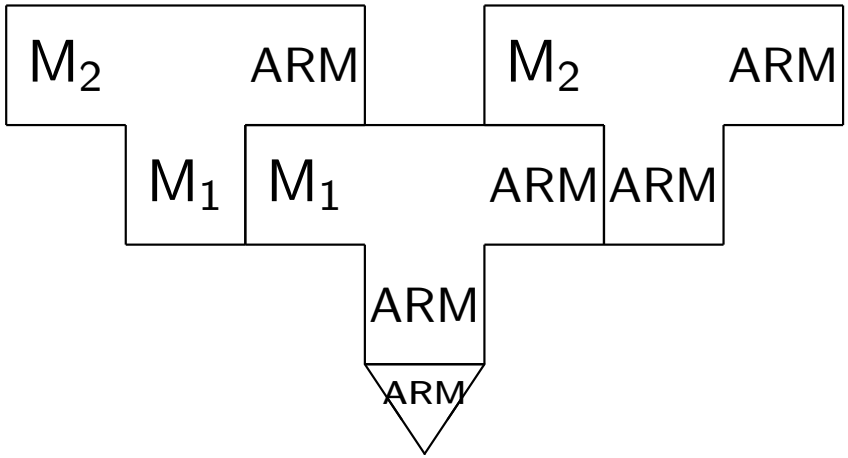
Incremental bootstrapping

Extend the compiler source *again* from M_1 to M_2. Now, the platform can be M_1. We have:





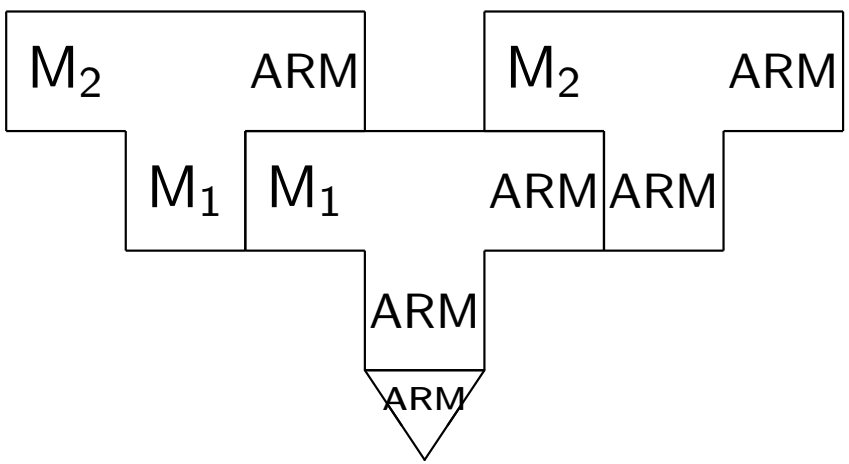
Compile with the M_1 compiler:



Incremental bootstrapping

Extend the compiler source *again* from M_1 to M_2. Now, the platform can be M_1. We have:

Compile with the M_1 compiler:



Repeat until $M_k = M+$. (And possibly further for optimisations.)

Is bootstrapping always a good idea?

To bootstrap a compiler, it has to be written in its own language.

That language may or may not be suitable for writing compilers.
Ideally (IMO), a language for writing compilers should have:

- Easy manipulation of trees, including garbage collection and deep pattern-matching.
- Type safety (useful for any major programming task).
- Modularisation (to separate the task into multiple modules with clear interfaces).
- Implementations available on multiple platforms.
- Turing completeness.

If your language does not have these features, it may be better to write your compiler in another language.
You can still use parts of the bootstrapping process.

Implementation Methods Affect Language Design

Design of early programming languages (FORTRAN, COBOL, . . .) were influenced by the then-known implementation methods. For example, static memory allocation implies no recursion and no unbounded tree-structured data.

Allegedly, the original LISP used dynamic scoping because the implementors (students of McCarthy) either did not know how to implement static scoping or did not know it would make a difference. Some scripting languages designed by people with little or no knowledge of compilers have strange scoping rules, probably for similar reasons. Later implementations of these languages stick to these rules for backwards compatibility.

Interpretation makes it easy to execute run-time generated source code, so this ability has been added to many languages that were originally interpreted (LISP, BASIC, . . .) – even when, later, they are compiled.

Exercise

Before you move on, solve exercise 3.3 from the notes.