

Compilation 2011

The What and Why of Compilers

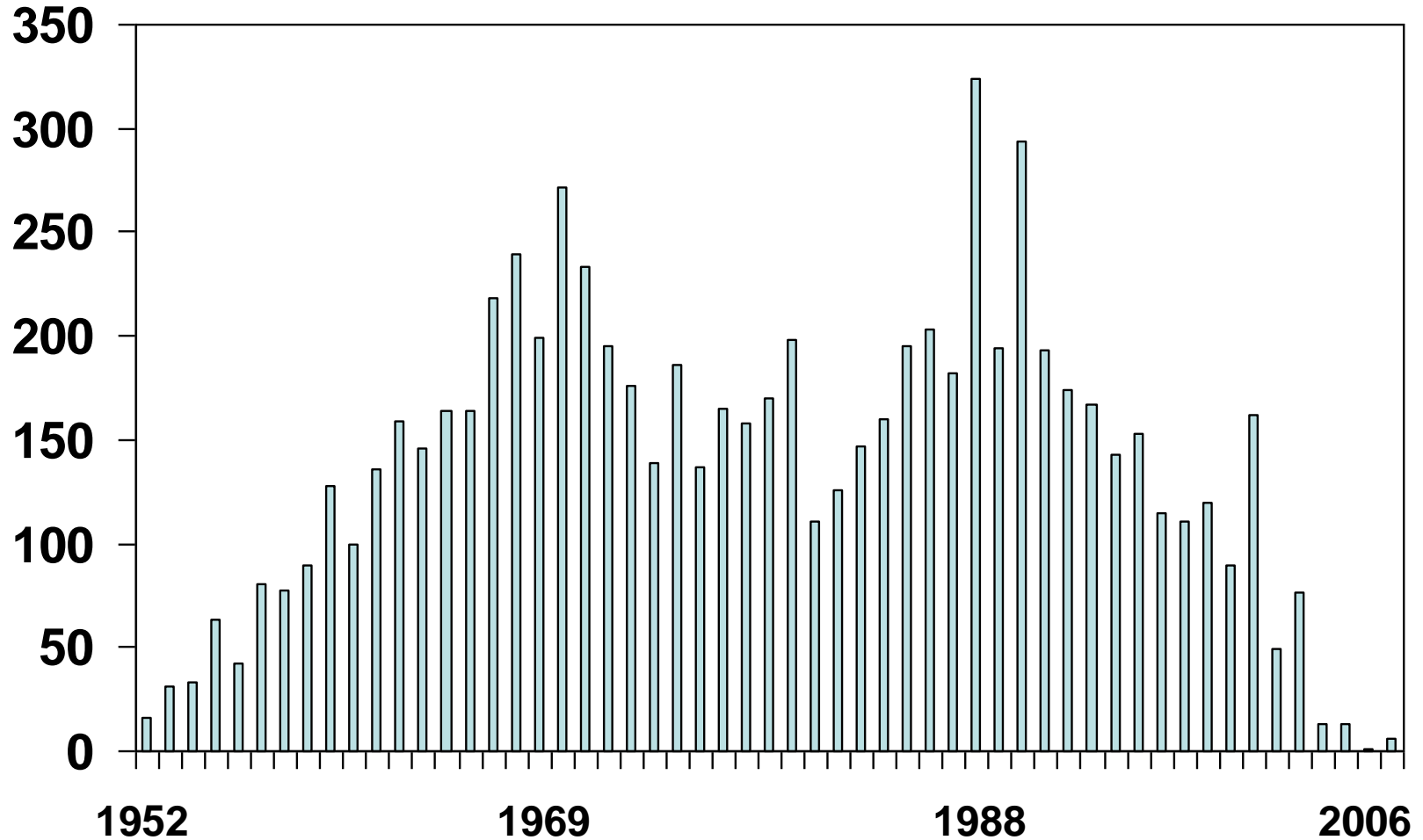
Jan Midtgaard
Michael I. Schwartzbach
Aarhus University

What is a Compiler?

- A program that:
 - translates from one programming language to another
 - preserves the semantics (in some sense)
 - a compiler also implicitly **defines** a semantics but it is hard to reason about programs in terms of a compiler-based semantics
- Each language must have at least one compiler (or interpreter)
- It is often the case that:
 - the *source* language is high-level
 - the *target* language is low-level

New Programming Languages 1952-2006

<http://hopl.murdoch.edu.au/>



Famous Programming Languages

1952 Autocoder	1970 SQL	1987 Perl
1956 IPL	1971 Forth	1988 Oberon
1954 Fortran	1972 Prolog	1990 Haskell
1958 Lisp	1972 Smalltalk	1991 Python
1960 Algol	1973 C	1994 Dylan
1960 Cobol	1973 ML	1994 Java
1964 APL	1975 Scheme	1996 OCaml
1964 PL/1	1979 Modula-2	1997 JavaScript
1965 BCPL	1980 Ada	1997 Ruby
1967 Simula 67	1981 C++	1998 Erlang
1967 Snobol 4	1984 SML	2000 C#
1968 Algol 68	1985 Eiffel	2001 AspectJ
1970 Pascal	1987 Caml	2006 C ω

Famous Programming Languages

1952 Autocoder	1970 SQL	1987 Perl
1956 IPL	1971 Forth	1988 Oberon
1954 Fortran	1972 Prolog	1990 Haskell
1958 Lisp	1972 Smalltalk	1991 Python
1960 Algol	1973 C	1994 Dylan
1960 Cobol	1973 ML	1994 Java
1964 APL	1975 Scheme	1996 OCaml
1964 PL/1	1979 Modula-2	1997 JavaScript
1965 BCPL	1980 Ada	1997 Ruby
1967 Simula 67	1981 C++	1998 Erlang
1967 Snobol 4	1984 SML	2000 C#
1968 Algol 68	1985 Eiffel	2001 AspectJ
1970 Pascal	1987 Caml	2006 C ω

Domain-Specific Languages

- Extend software design
- Are concrete artifacts that permit better:
 - representation
 - optimization
 - analysis
- Replace low-level programs and libraries
- Exist in tens of thousands
- Require full-scale compiler technology

Some Interesting DSL Compilers

- LaTeX
 - document descriptions → PostScript
- Esterel
 - reactive systems → hardware designs
- WS-BPEL
 - flow algebras → Web services
- Nyquist
 - musical compositions → sounds
- SQL
 - queries → evaluation plans

The FORTRAN Compiler

- Implemented in 1957
- The world's first compiler
- Motivated by the economics of programming
- Had to overcome deep skepticism
- Focused on efficiency of the generated code
- Pioneered many concepts and techniques
- Revolutionized computer programming

How Good Are Compilers?

```
/* naive */  
for (i = 0; i < N; i++) {  
    a[i] = a[i] * 2000;  
    a[i] = a[i] / 10000;  
}
```

```
/* expert */  
b = a;  
for (i = 0; i < N; i++) {  
    *b = *b * 2000;  
    *b = *b / 10000;  
    b++;  
}
```

Better Than We Are!

```
/* naive */  
for (i = 0; i < N; i++) {  
    a[i] = a[i] * 2000;  
    a[i] = a[i] / 10000;  
}
```

```
/* expert */  
b = a;  
for (i = 0; i < N; i++) {  
    *b = *b * 2000;  
    *b = *b / 10000;  
    b++;  
}
```

loop	level	sparc	mips	alpha
naive		20.5	21.6	7.9
naive	-O1	8.8	12.3	3.3
naive	-O2	7.9	11.2	3.0
expert		19.5	17.6	7.6
expert	-O1	12.4	15.4	4.1
expert	-O2	10.7	12.9	3.9

Better Than We Are!

```
/* naive */  
for (i = 0; i < N; i++) {  
    a[i] = a[i] * 2000;  
    a[i] = a[i] / 10000;  
}
```

```
/* expert */  
b = a;  
for (i = 0; i < N; i++) {  
    *b = *b * 2000;  
    *b = *b / 10000;  
    b++;  
}
```

loop	level	sparc	mips	alpha
naive		20.5	21.6	7.9
naive	-O1	8.8	12.3	3.3
naive	-O2	7.9	11.2	3.0
expert		19.5	17.6	7.6
expert	-O1	12.4	15.4	4.1
expert	-O2	10.7	12.9	3.9

Phases of Modern Compilers

- preprocessing
- scanning
- parsing
- desugaring
- weeding
- environments
- linking
- type checking
- static analysis
- resource allocation
- code generation
- code optimization
- code emission
- assembly

Compiler Architecture

- Compiler phases suggest a modular design:
one phase = one module
- Many phases can be generated automatically
- Tools range from industrial to experimental
- Compilers divide into *frontends* and *backends*
- Both may be retargeted

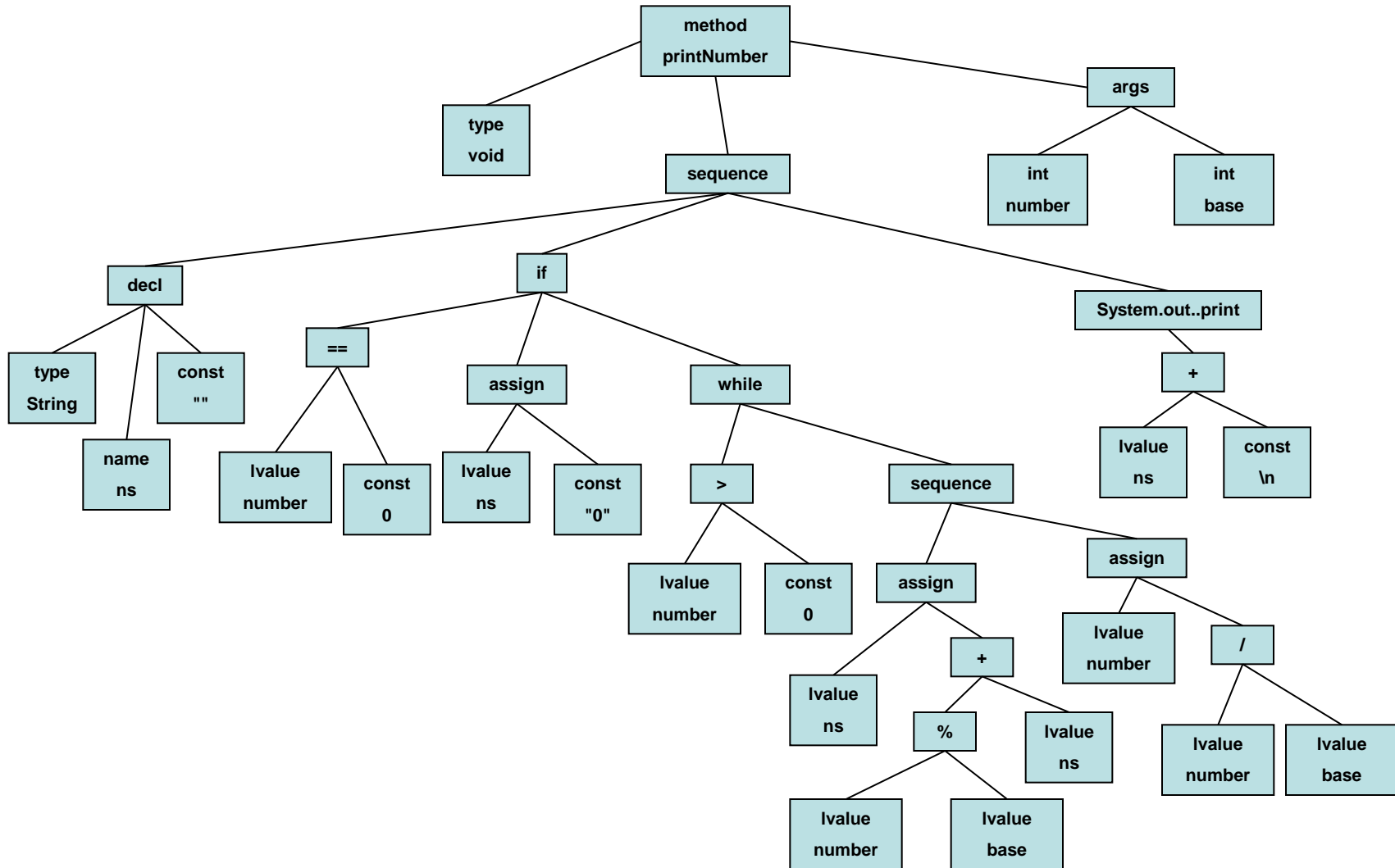
A Tiny Java Method

```
public void printNumber(int number, int base)
    throws Exception {
    String ns = "";
    if (number == 0) {
        ns = "0";
    } else {
        while (number > 0) {
            ns = (number % base) + ns;
            number = number / base;
        }
    }
    System.out.print(ns+"\n");
    return;
}
```

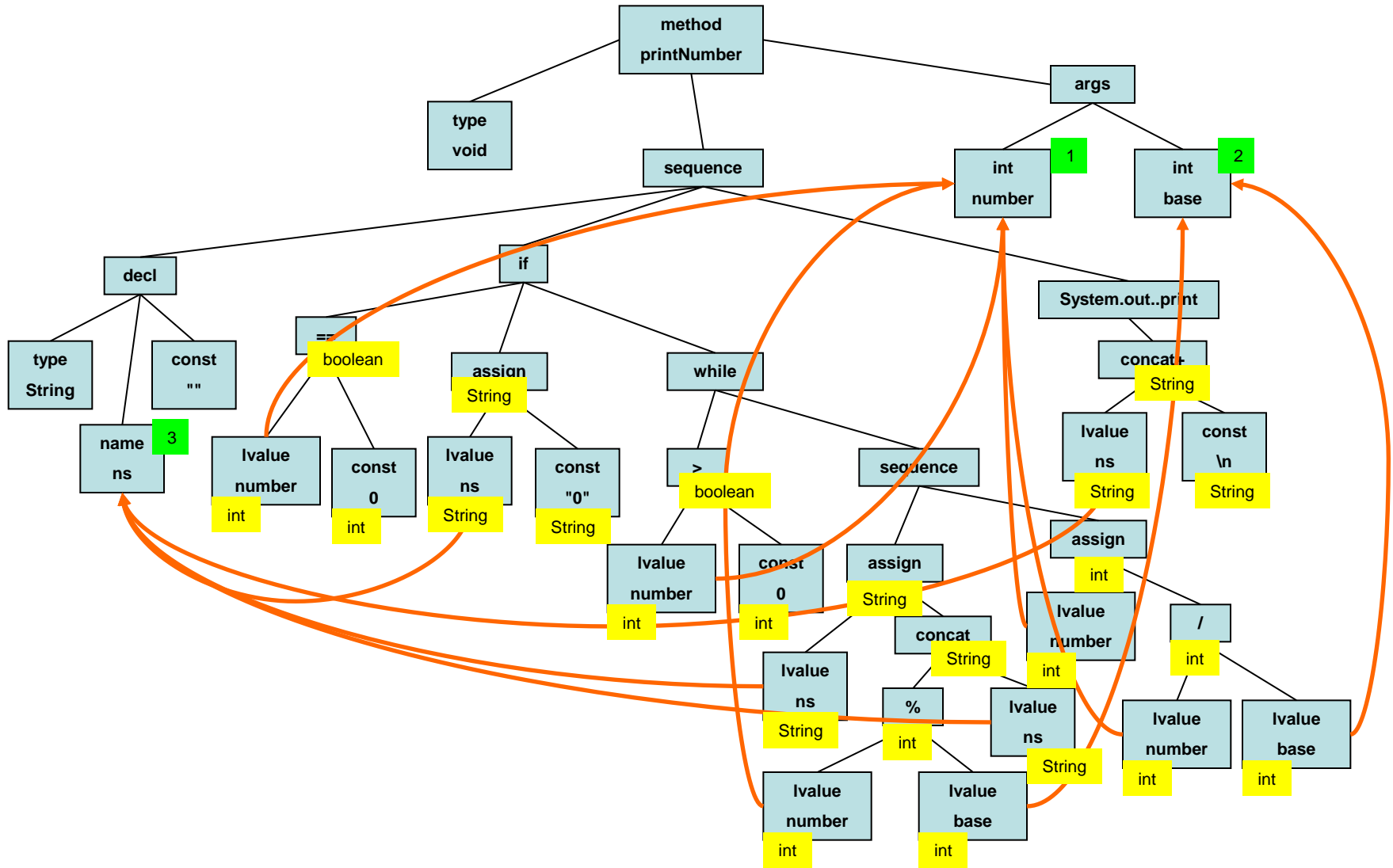
Stream of Tokens

```
keyword: public          symbol: (              symbol: {
keyword: void            identifier: number     identifier: ns
identifier: printNumber  symbol: ==            symbol: =
symbol: (                constant: 0            symbol: (
keyword: int             symbol: )            identifier: number
identifier: number       symbol: {            symbol: %
symbol: ,                identifier: ns        identifier: base
keyword: int             symbol: =            symbol: )
identifier: base         constant: "0"        symbol: +
symbol: )                symbol: ;            identifier: ns
keyword: throws          symbol: }            symbol: ;
identifier: Exception    keyword: else        identifier: number
symbol: {                symbol: {            symbol: =
identifier: String       keyword: while       identifier: number
identifier: ns           symbol: (            symbol: /
symbol: =                identifier: number    identifier: base
constant: ""            symbol: >            symbol: ;
symbol: ;                constant: 0          symbol: }
keyword: if              symbol: )            ...
```

Abstract Syntax Tree



Resource Allocation



Bytecode Generation

```
.method public printNumber(II)V
.throw java/lang/Exception
.limit stack 4
.limit locals 4
ldc ""
dup
astore_3
pop
iload_1
iconst_0
if_icmpeq true_2
...
stop_1:
aload_3
invokestatic java/lang/String/valueOf(Ljava/lang/Object;)Ljava/lang/String;
ldc "\012"
invokevirtual java/lang/String/concat(Ljava/lang/String;)Ljava/lang/String;
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/print(Ljava/lang/Object;)V
return
.end method
```

64 instructions

Bytecode Optimization

```
.method public printNumber(II)V
.throw java/lang/Exception
.limit stack 4
.limit locals 4
ldc ""
astore_3
iload_1
iconst_0
if_icmpne start_4
ldc "0"
astore_3
...
stop_1:
aload_3
invokestatic java/lang/String/valueOf(Ljava/lang/Object;)Ljava/lang/String;
ldc "\012"
invokevirtual java/lang/String/concat(Ljava/lang/String;)Ljava/lang/String;
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/print(Ljava/lang/Object;)V
return
.end method
```

40 instructions

Code Assembly

printNumber.class:

```
cafe babe 0003 002d 0023 0100 116a 6176 612f 6c61 6e67 2f49 6e74 6567 6572 0100
106a 6176 612f 6c61 6e67 2f4f 626a 6563 7401 0005 2849 4929 5601 0006 3c69 6e69
743e 0700 020c 0004 000b 0700 1101 0026 284c 6a61 7661 2f6c 616e 672f 5374 7269
6e67 3b29 4c6a 6176 612f 6c61 6e67 2f53 7472 696e 673b 0c00 1900 1e0a 001f 0018
0100 0328 2956 0100 0663 6f6e 6361 7401 0004 436f 6465 0100 0a53 6f75 7263 6546
696c 6501 0004 2849 2956 0c00 0c00 0801 0010 6a61 7661 2f6c 616e 672f 5374 7269
6e67 0100 046e 756c 6c08 0012 0a00 0500 0601 0011 496e 7465 6765 7254 6f53 7472
696e 672e 6a01 000b 7072 696e 744e 756d 6265 7201 000f 496e 7465 6765 7254 6f53
7472 696e 670c 0004 000f 0100 0874 6f53 7472 696e 670a 0007 0010 0800 2007 0017
0a00 1f00 0901 0014 2829 4c6a 6176 612f 6c61 6e67 2f53 7472 696e 673b 0700 0101
0001 3008 0022 0100 0000 2100 1c00 0500 0000 0000 0200 0100 0400 0b00 0100 0d00
0000 1100 0100 0100 0000 052a b700 14b1 0000 0000 0001 0016 0003 0001 000d 0000
005c 0004 0004 0000 0050 1221 594e 571b 039f 0007 03a7 0004 0499 000b 121b 594e
57a7 0037 1b03 a300 0703 a700 0404 9900 2abb 001f 591b 1c70 b700 0ab6 001d 2d59
c600 06a7 0006 5712 13b6 001a 594e 571b 1c6c 593c 57a7 ffcf b1b1 0000 0000 0001
000e 0000 0002 0015
```

Virtual Machines and Compilers

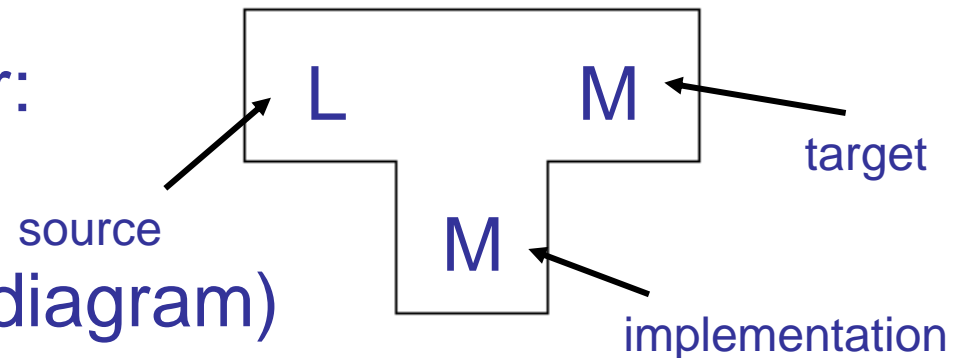
- Bytecode is just an intermediate representation of the program
- The VM is part of the compilation process
- But it is useful to separate the compilation process into (front-end) compiler and VM
 - N languages and M platforms require only N (front-end) compilers and M virtual machines, rather than $N \cdot M$ monolithic compilers
 - Enables adaptive optimization techniques
- VMs are quite complex pieces of software, too
 - Have their own sophisticated architecture
 - Register allocation, memory management, adaptive optimization, ...
- But in this course we will not say much about how VMs work

Bootstrapping Compilers (1/4)

- We are given:
 - a machine language M
 - a programming language L

- We need a compiler:

(this is called a T-diagram)

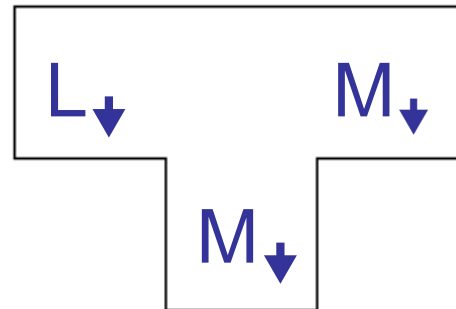


- The direct approach is hard and difficult

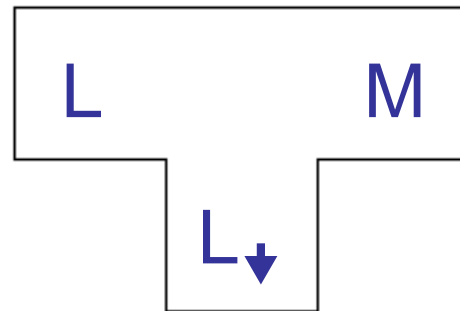
Bootstrapping Compilers (2/4)

- We define:
 - L_{\downarrow} is a simple subset of L
 - M_{\downarrow} is naive and inefficient M code

- It is easy to implement:

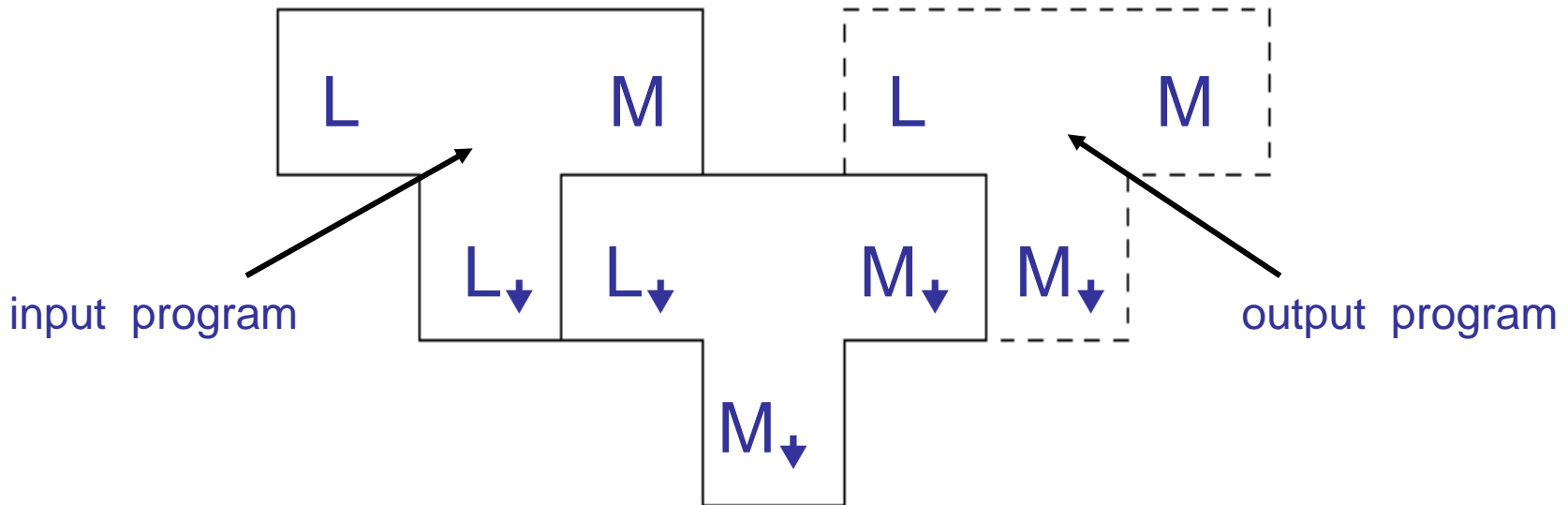


- And in parallel:

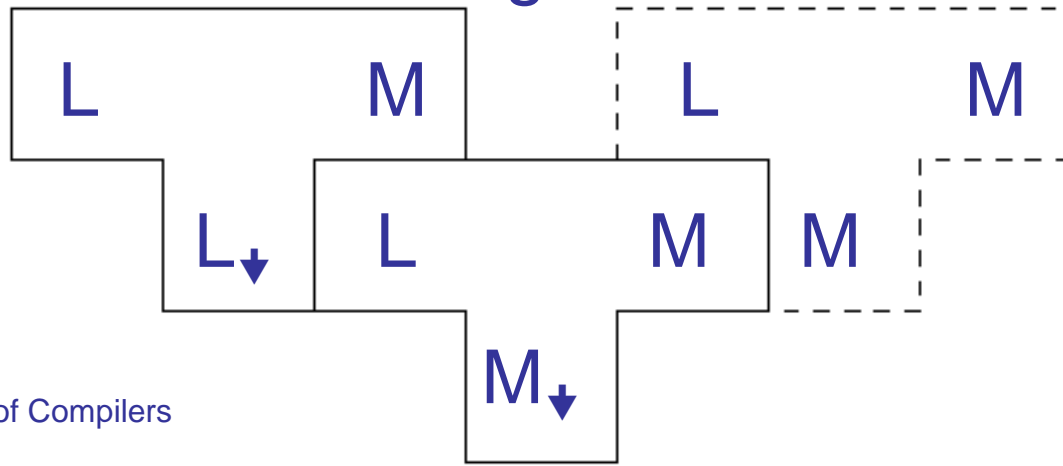


Bootstrapping Compilers (3/4)

- Combining the two compilers we get:

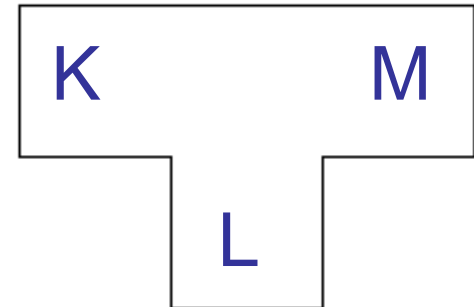


- A final combination gives us what we want:

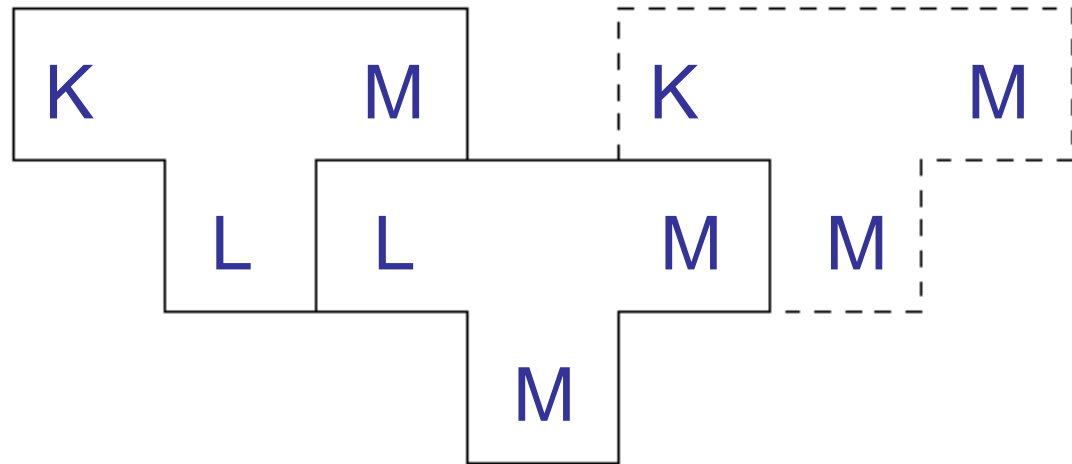


Bootstrapping Compilers (4/4)

- We now want a compiler for another language K
- We first program a compiler:



- The new compiler is then obtained as:



- And so on...