Compiler and Language Processing Tools

Summer Term 2011

Introduction

Prof. Dr. Arnd Poetzsch-Heffter

Software Technology Group
TU Kaiserslautern

Outline

Language processing tools

- Processing of source texts in (source) languages
- Analysis of (source) texts
- Translation to target languages

Language processing tools (2)

Typical source languages

- Programming languages: C, C++, C#, Java, Scala, Haskell, ML, Smalltalk, Prolog
- Script languages: JavaScript, bash
- Languages for configuration management: make, ant
- Application and tool-specific languages: Excel, JFlex, CUPS
- Specification languages: Z, CASL, Isabelle/HOL
- Formatting and data description languages: LaTeX, HTML, XML
- Design and architecture description languages: UML, SDL, VHDL, Verilog

Language processing tools (3)

Typical target languages

- Assembly, machine, and bytecode languages
- Programming language
- Data and layout description languages
- Languages for printer control
- ...

Language processing tools (4)

Language implementation tasks

- Tool support for language processing
- Integration into existing systems
- Connection to other systems

Application domains

- Programming environments
  - Context-sensitive editors, class browers
  - Graphical programming tools
  - Pre-processors
  - Compilers
  - Interpreters
  - Debuggers
  - Run-time environments (loading, linking, execution, memory management)

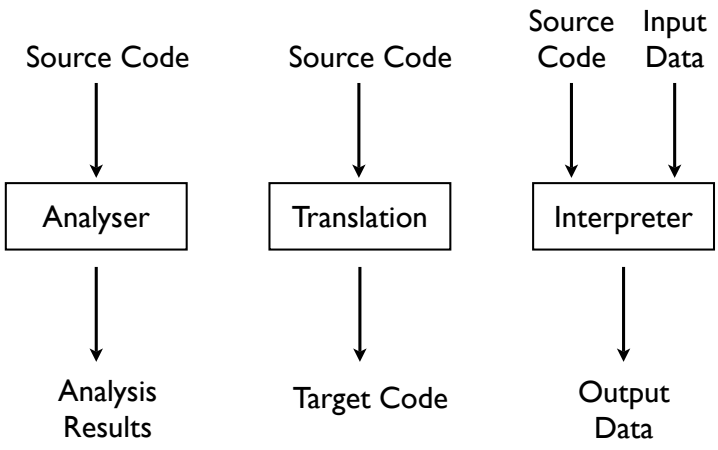
Application domains (2)

- Generation of programs from design documents (UML)
- Program comprehension, re-engineering
- Design and implementation of domain-specific languages
  - Robot control
  - Simulation tools
  - Spread sheets, active documents
- Web technology
  - Analysis of Web sites
  - Active Web sites (with integrated functionality)
  - Abstract platforms, e.g. JVM, .NET
  - Optimization of caching

Related fields

- Formal languages, language specification and design
- Programming and specification languages
- Programming, software engineering, software generation, software architecture
- System software, computer architecture

Tasks of Language-Processing Tools



Analysis, translation and interpretation are often combined.

Tasks of Language-Processing Tools (2)

1. **Translation**
   - Compiler implements analysis and translation
   - OS and real machine implement interpretation

   Pros:
   - Most efficient solution
   - One interpreter for different programming languages
   - Prerequisite for other solutions

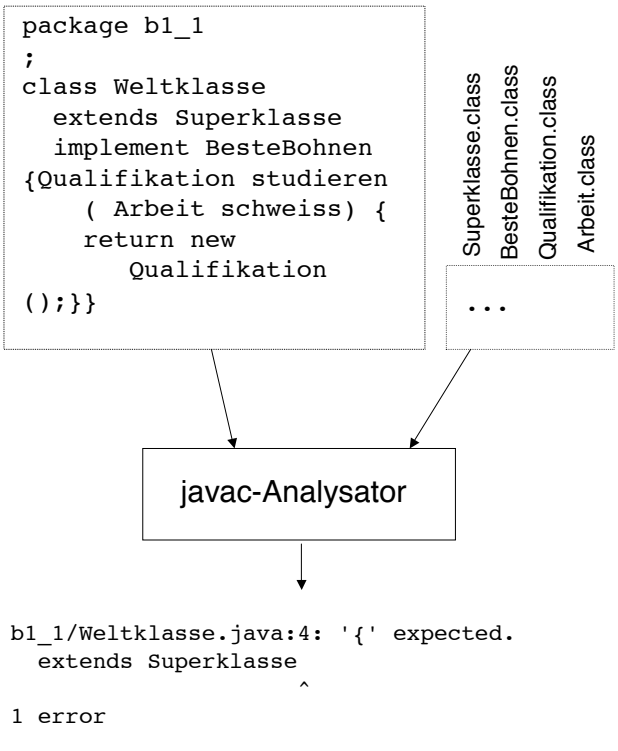
Tasks of Language-Processing Tools (3)

2. **Direct interpretation**
   - Interpreter implements all tasks.
   - Examples: JavaScript, command line languages (bash)
   - Pros: No translation necessary (but analysis at run-time)

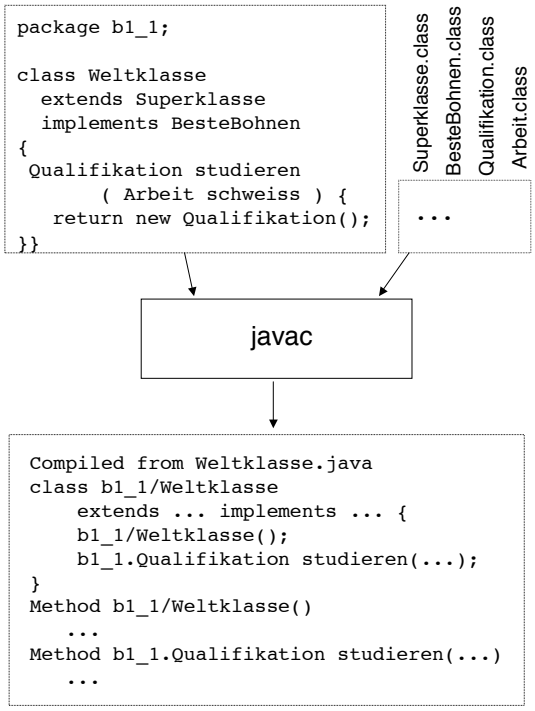
Tasks of Language-Processing Tools (4)

3. **Abstract and virtual machines**
   - Compiler implements analysis and translation to abstract machine code
   - Abstract machine works as interpreter
   - Examples: Java/JVM, C#, .NET
   - Pros:
     - Platform independent (portability, mobile code)
     - Self-modifing programs possible
4. **Other combinations**

Example: Analysis



Example: Translation



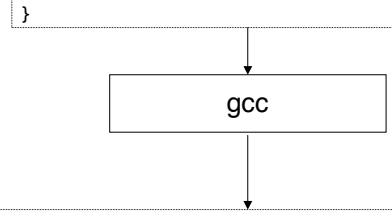
Example: Translation (2)

Result of translation

```
Compiled from Weltklasse.java
class b1_1/Weltklasse
      extends b1_1.Superklasse
      implements b1_1.BesteBohnen {
    b1_1/Weltklasse();
    b1_1.Qualifikation studieren(b1_1.Arbeit);
}

Method b1_1/Weltklasse()
   0 aload_0
   1 invokespecial #6 <Method b1_1.Superklasse()>
   4 return

Method b1_1.Qualifikation studieren(b1_1.Arbeit)
   0 new #2 <Class b1_1.Qualifikation>
   3 dup
   4 invokespecial #5 <Method b1_1.Qualifikation()>
   7 areturn
```

Example 2: Translation

```
int main() {
  printf("Willkommen zur Vorlesung!");
  return 0;
}
```



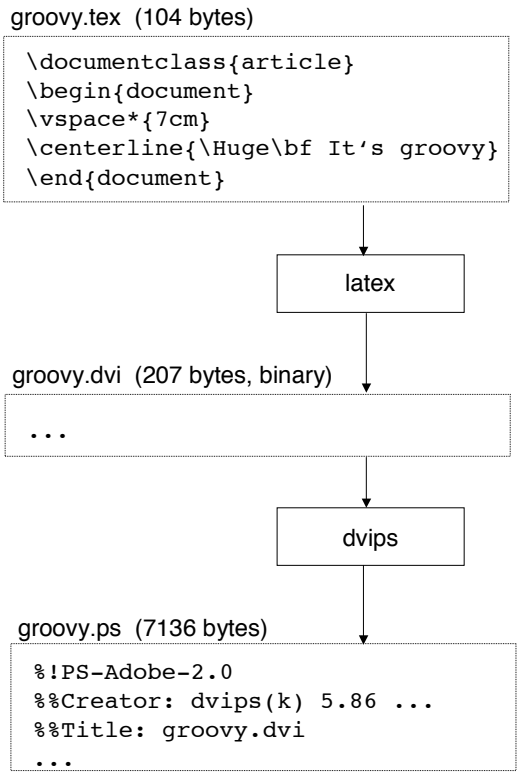

```
   .file       "hello_world.c"
   .version    "01.01"
gcc2_compiled.:
.section      .rodata
.LC0:
   .string     "Willkommen zur Vorlesung!"
.text
   .align 16
.globl main
   .type        main,@function
main:
   pushl %ebp
   movl %esp,%ebp
   subl $8,%esp

...
```


Example 2: Translation (2)

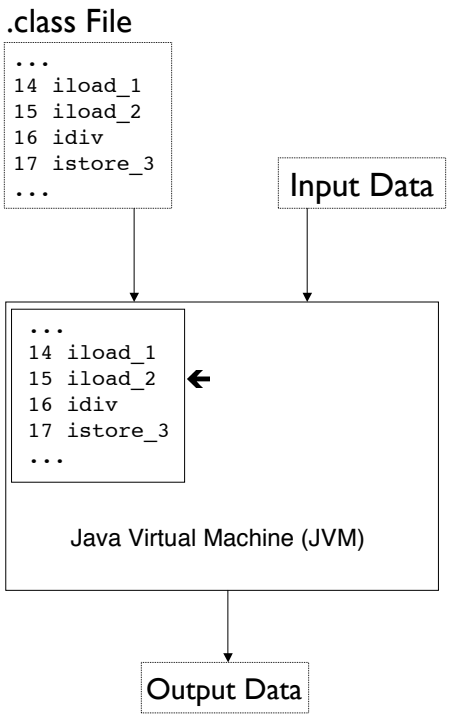
Result of translation

```
   .file       "hello_world.c"
   .version    "01.01"
gcc2_compiled.:
.section      .rodata
.LC0:
   .string     "Willkommen zur Vorlesung!"
.text
   .align 16
.globl main
   .type        main,@function
main:
   pushl %ebp
   movl %esp,%ebp
   subl $8,%esp
   addl $-12,%esp
   pushl $.LC0
   call printf
   addl $16,%esp
   xorl %eax,%eax
   jmp .L2
   .p2align 4,,7
.L2:
   movl %ebp,%esp
   popl %ebp
   ret
.Lfe1:
   .size  main,.Lfe1-main
   .ident "GCC: (GNU) 2.95.2 19991024 (release)"
```

Example 3: Translation

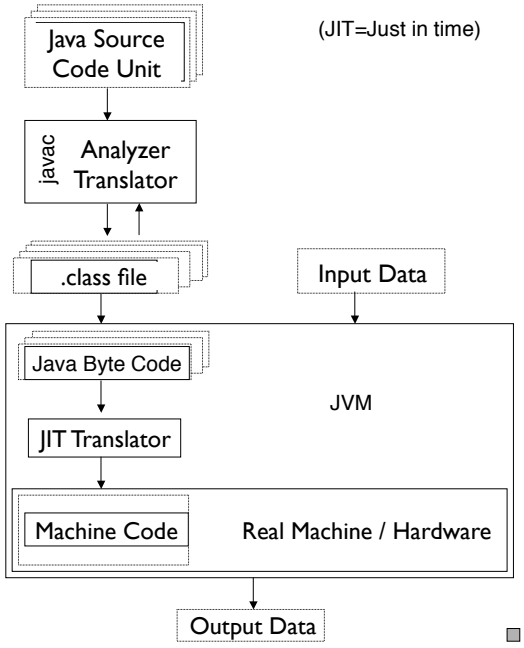


Example: Interpretation

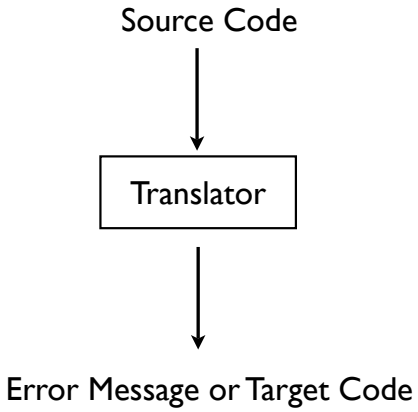


Example: Combined technique

Java implementation with just-in-time (JIT) compiler



Language processing: The task of translation



- **Translator** (in a broader sense): Analysis, optimization and translation
- **Source code**: Input (string) for translator in syntax of source language (SL)
- **Target Code**: Output (string) of translator in syntax of target language (TL)

Phases of language processing

- Analysis of input:
  - Program text
  - Specification
  - Diagrams
- Dependant on target of implementation
  - Transformation (XSLT, refactoring)
  - Pretty printing, formatting
  - Semantic analysis (program comprehension)
  - Optimization
  - (Actual) translation

Compile time vs. run-time

- **Compile time**: during run-time of compiler/translator
  Static: All information/aspects known at compile time, e.g.:
  - Type checks
  - Evaluation of constant expressions
  - Relative addresses
- **Run-time**: during run-time of compiled program
  Dynamic: All information that are not statically known, e.g.:
  - Allocation of dynamic arrays
  - Bounds check of arrays
  - Dynamic binding of methods
  - Memory management of recursive procedures

For *dynamic aspects* that cannot be handled at *compile time*, the compiler generates code that handles these aspects at *run-time*.

What is a good compiler?

Requirements for translators

- Error handling (static/dynamic)
- Efficient target code
- Choice: Fast translation with slow code
  vs. slow translation with fast code
- Semantically correct translation

Semantically correct translation

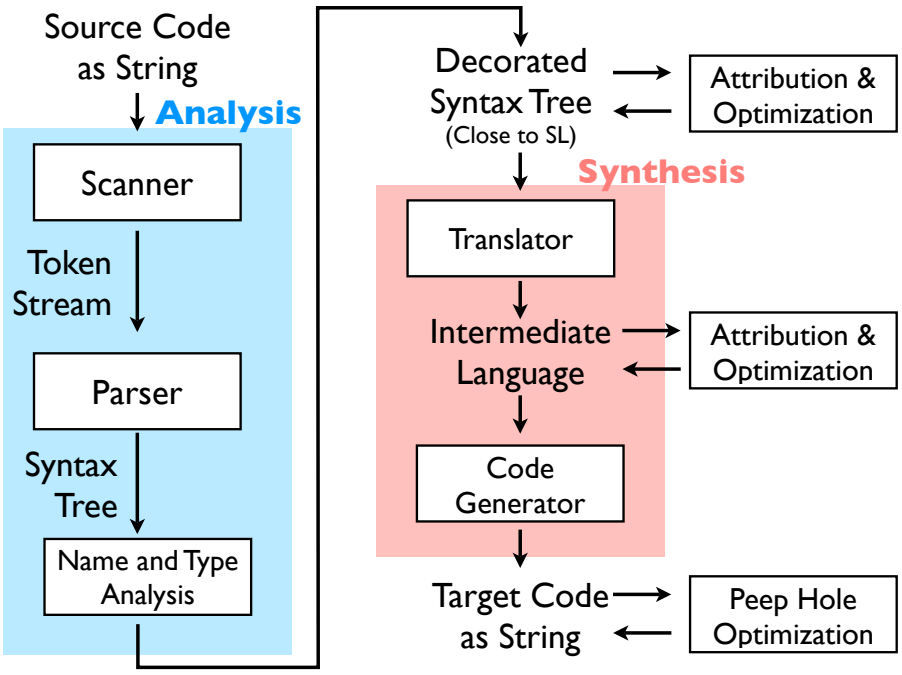
Intuitive definition: Compiled program behaves according to language definition of source language.

Formal definition:

- semSL: SL_Program $\times$ SL_Data $\rightarrow$ SL_Data
- semTL: TL_Program $\times$ TL_Data $\rightarrow$ TL_Data
- compile: SL_Program $\rightarrow$ TL_Program
- code: SL_Data $\rightarrow$ TL_Data
- decode: TL_Data $\rightarrow$ SL_Data

Semantic correctness:
semSL(P,D) = decode(semTL(compile(P), code(D)))

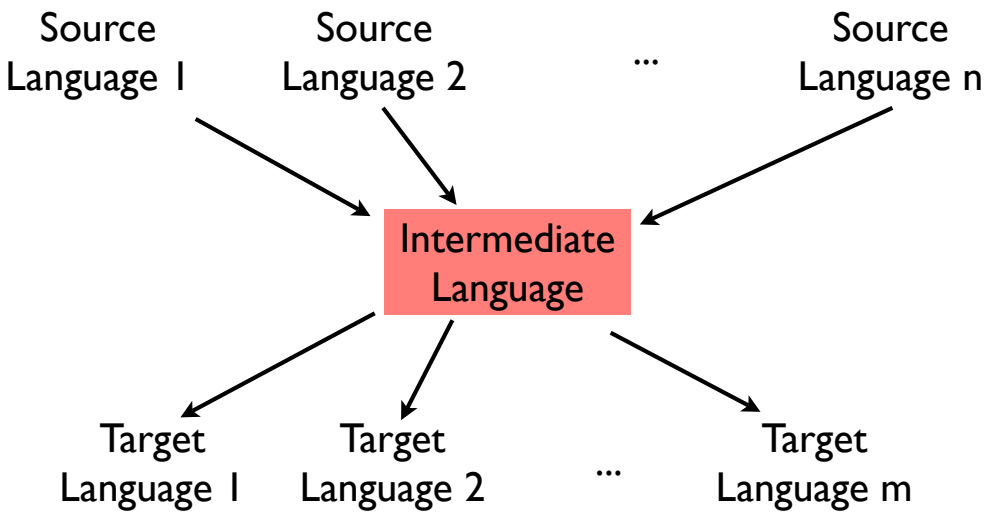
Compiler Architecture



Properties of compiler architectures

- Phases are conceptual units of translation
- Phases can be interleaved
- Design of phases depends on source language, target language and design decisions
- Phase vs. **pass** (phase can comprise more than one pass.)
- Separate translation of pogram parts
  (Interface information must be accessible.)
- Combination with other architecture decisions:
  Common intermediate language

Common intermediate language



Dimensions of compiler construction

- Programming languages
  - Sequential procedural, imperative, OO-languages
  - Functional, logical languages
  - Parallel languages/language constructs
- Target languages/machines
  - Code for abstract machines
  - Assembler
  - Machine languages (CISC, RISC, ...)
  - Multi-processor/multi-core architectures
  - Memory hierarchy
- Translation tasks: analysis, optimization, synthesis
- Construction techniques and tools: bootstrapping, generators
- Portability, specification, correctness

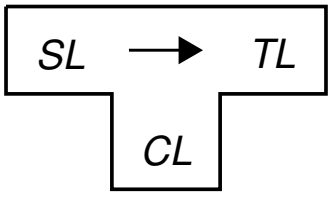
Compiler construction techniques

1. Stepwise construction
   - Construction with compiler for different language
   - Construction with compiler for different machine
   - Bootstrapping
2. Compiler-compiler: Tools for compiler generation
   - Scanner generators (regular expressions)
   - Parser generators (context-free grammars)
   - Attribute evaluation generators (attribute grammar)
   - Code generator generators (machine specification)
   - Interpreter generators (semantics of language)
   - Other phase-specific tools
3. Special programming techniques
   - General technique: syntax-driven
   - Special technique: recursive descend

Stepwise construction

Programming typically depends on an existing compiler for the implementation language. For compiler construction, this does not hold in general.

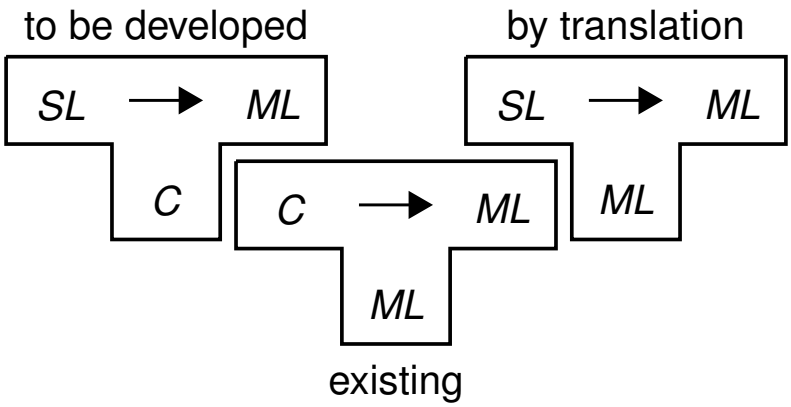
Source, target, and implementation languages of compilers can be denoted in T-diagrams.



T-diagram denotes compiler from source language *SL* to target language *TL* (*SL* $\rightarrow$ *TL* compiler) written in language *CL*.

Construction with compiler for different language

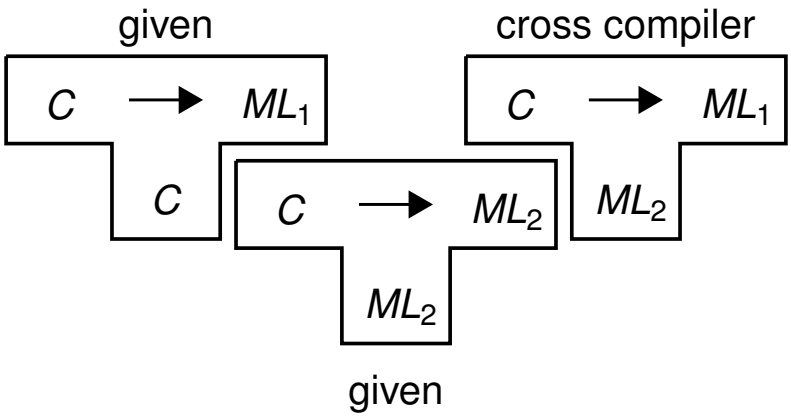
- Given: *C* $\rightarrow$ *ML* (machine language) compiler in *ML*
- Construct: *SL* $\rightarrow$ *ML* compiler in *ML*
- Solution: Develop *SL* $\rightarrow$ *ML* compiler in *C*, translate that compiler from *C* $\rightarrow$ *ML* by using the existing *C* $\rightarrow$ *ML* compiler



Construction with compiler for different machine

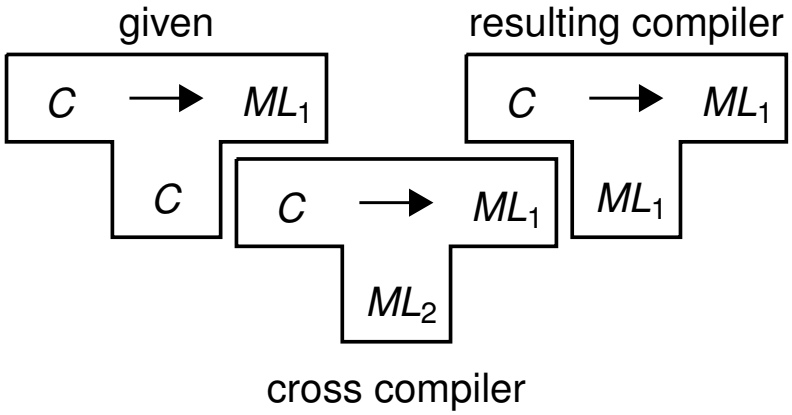
- Construct: $C \rightarrow ML_1$ compiler in ML_1
- Given
  1. $C \rightarrow ML_1$ compiler in C
  2. $C \rightarrow ML_2$ compiler in ML_2
- Method: construct cross compiler

First step



Construction with compiler for different machine (2)

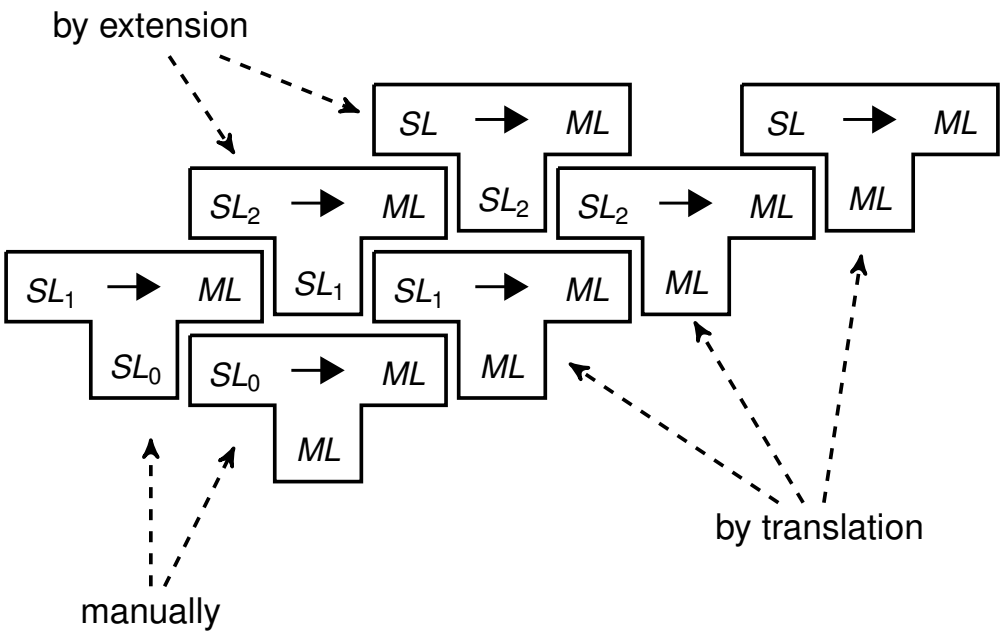
Second step



Bootstrapping

- Construct: $SL \rightarrow ML$ compiler in ML
- Suppose: yet no compiler exists
- Method:
  1. Construct partial language SL_i of SL such that $SL_0 \subset SL_1 \subset SL_2 \subset \ldots \subset SL$
  2. Implement SL_0 compiler for ML in ML
  3. Implement SL_{i+1} compiler for ML in SL_i
  4. Create SL_{i+1} compiler for ML in ML

Bootstrapping (2)



Recommended reading

Wilhelm, Maurer:

- Chap. 1, Introduction (pp. 1–5)
- Chap. 6, Structure of Compilers (pp. 225 – 238)

Appel

- Chap. 1, Introduction (pp. 3 – 14)