Compiler Construction

Chapter 11

A New Compiler

- Perhaps a new source language
- Perhaps a new target for an existing compiler
- Perhaps both

Source Language

- Larger, more complex languages generally require larger, more complex compilers

- Is the source language expected to evolve?
  - E.g., Java 1.0 $\longrightarrow$ Java 1.1 $\longrightarrow$ . . .
  - A brand new language may undergo considerable change early on
  - A small working prototype may be in order
  - Compiler writers must anticipate some amount of change and their design must therefore be flexible
  - Lexer and parser generators (like Lex and Yacc) are therefore better than hand-coding the lexer and parser when change is inevitable

Target Language

- The nature of the target language and run-time environment influence compiler construction considerably
- A new processor and/or its assembler may be buggy

  Buggy targets make it difficult to debug compilers for that target!

- A successful source language will persist over several target generations
  - E.g., 386 $\longrightarrow$ 486 $\longrightarrow$ Pentium $\longrightarrow \ldots$
  - Thus the design of the IR is important
  - Modularization of machine-specific details is also important

Compiler Performance Issues

- Compiler speed
- Generated code quality
- Error diagnostics
- Portability
- Maintainability

Compiler Speed

- Reduce the number of modules
- Reduce the number of passes

  Perhaps generate machine code on the first pass

- Disadvantages:
  - Target code may not be high quality
  - The compiler may be difficult to maintain

Compiler Portability

- Retargetability

  Easily modified to generate code for a different target language

- Rehostability

  Easily modified to run on a different machine

- A portable compiler may not be as effi cient as a compiler designed for a specifi c machine

  A specific machine compiler can be tuned to the specific target language

Bootstrapping

- Illustrated by remarks such as “A C compiler for a new platform written in C”

- The process is:

1. Devise a minimal subset of language.
2. Write a compiler for that minimal subset *in the language of that minimal subset*
3. Hand-translate this minimal subset source code into assembly language and assemble it—this produces a working compiler for a subset of the target language
4. Write a new compiler *in the language of the working compiler* to accept more source features that are missing from the language of the working compiler
5. Use the working compiler to compile this new compiler—this produces a new working compiler which accepts a superset of the language under which it was compiled
6. Repeat the last two steps until the working compiler realizes all features in the source language

Bootstrapping History

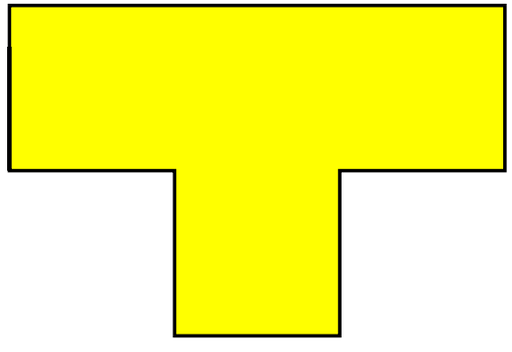
- The concept was developed in the mid-1950s
- The first LISP interpreter was built using bootstrapping

T-Diagrams

Useful for describing the bootstrapping process

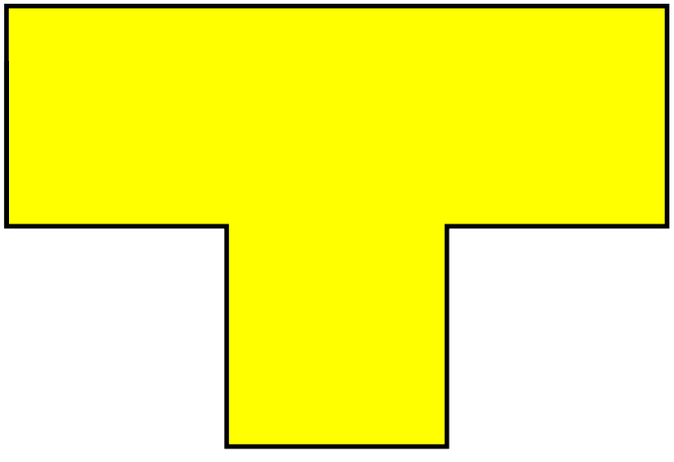
A compiler can be characterized by three languages:

- Source language: S
- Target language: T
- Implementation language: I



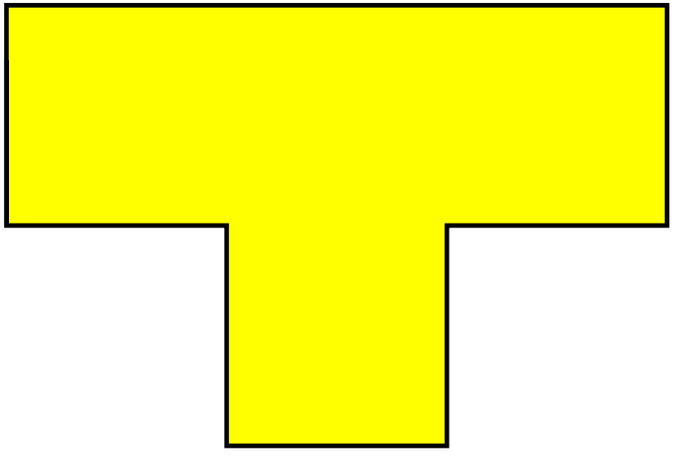
A T-diagram is also called a $S_I T$ diagram

Our Decaf T-Diagram



Early C++ Translator

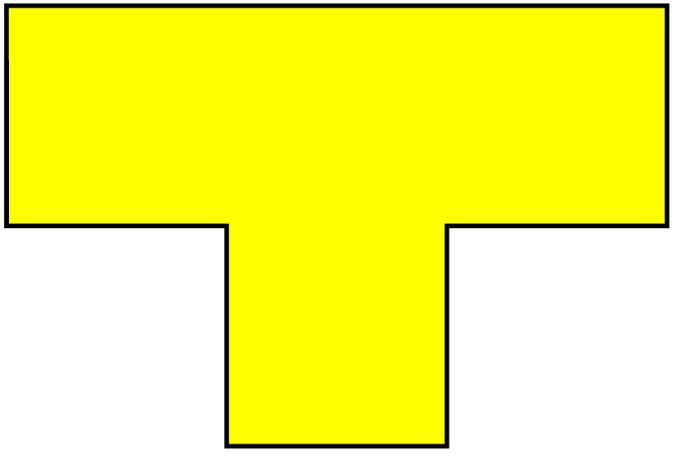
Cfront translated C++ into C to be compiled by a standard C compiler



Cross-compilation

A cross-compiler produces target code for a machine different from the one on which it is run

For example, running `gcc` on an Pentium Linux platform and generating code for a 68000 PalmOS platform



Compiling a Compiler

1. Suppose we have cross-compiler for a new language L in implementation language S generating code for machine N.

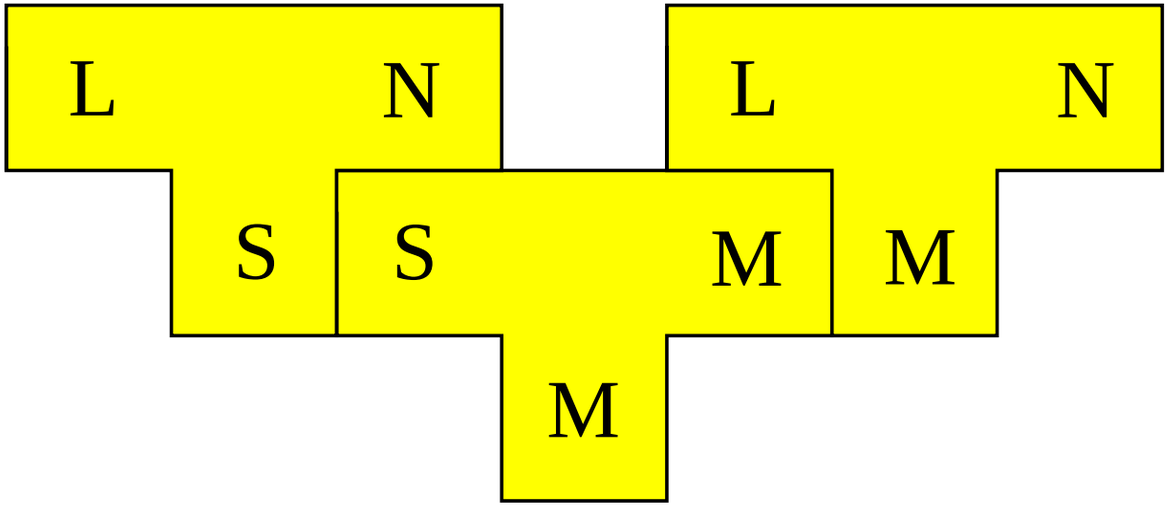
$$\mathrm{L_S N}$$

2. Suppose we also have an existing S compiler running on machine M implementing code for machine M:

$$\mathrm{S_M M}$$

3. Run $\mathrm{L_S N}$ through $\mathrm{S_M M}$ to produce $\mathrm{L_M N}$

Composing T-Diagrams



$$L_S N + S_M M = L_M N$$

Composition Example

1. Suppose we have cross-compiler for a new language, Decaf, implemented in C++:

$$\text{Decaf}_{\text{C++}}\text{MIPS}$$

2. Suppose we also have an existing C++ compiler for a PowerPC machine (e.g., Mac)

$$\text{C++}_{\text{PowerPC}}\text{PowerPC}$$

3. Run $\text{Decaf}_{\text{C++}}\text{MIPS}$ through $\text{C++}_{\text{PowerPC}}\text{PowerPC}$ to produce $\text{Decaf}_{\text{PowerPC}}\text{MIPS}$

Bootstrapping

Let's create a compiler for a new language L that runs on machine M

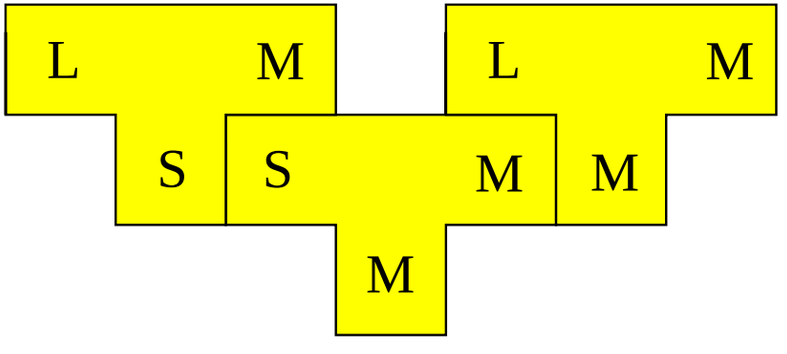
1. Write a compiler for S, a subset of L: $\mathrm{S}_{\mathrm{M}}\mathrm{M}$

   Here, M is assembly language

2. Write the compiler for L using language S: $\mathrm{L}_{\mathrm{S}}\mathrm{M}$

3. Compile $\mathrm{L}_{\mathrm{S}}\mathrm{M}$ under $\mathrm{S}_{\mathrm{M}}\mathrm{M}$ to make $\mathrm{L}_{\mathrm{M}}\mathrm{M}$

   $\mathrm{L}_{\mathrm{M}}\mathrm{M}$ is a compiler for language L that produces code for machine M



Retargeting *and* Rehosting the Compiler

Let's make an L compiler for a different machine, N

1. Write $\mathrm{L}_{\mathrm{L}}\mathrm{N}$
2. Compile $\mathrm{L}_{\mathrm{L}}\mathrm{N}$ with $\mathrm{L}_{\mathrm{M}}\mathrm{M}$ to produce $\mathrm{L}_{\mathrm{M}}\mathrm{B}$

   This make a cross-compiler for N that runs on machine M
3. Compile $\mathrm{L}_{\mathrm{L}}\mathrm{N}$ with the cross-compiler to produce $\mathrm{L}_{\mathrm{N}}\mathrm{N}$

   This makes a compiler for language L that runs on machine N