CS 304 Compiler Design

Text books

1. Compilers – Principles, Techniques & Tools , Aho, Ravi Sethi, D. Ullman

- **Introduction to compilers and lexical analysis:- Analysis of the source program - Analysis and synthesis phases, Phases of a compiler.**
- **Compiler writing tools. Bootstrapping.**
- **Lexical Analysis - Role of Lexical Analyser, Input Buffering, Specification of Tokens, Recognition of Tokens**

Compiler Writing Tools

- To make it convenient to write a compiler
- Helps to implement various phases of a compiler
- Compiler-compilers, compiler-generators or translator-wiring systems

- Lexical analyzers for all languages are essentially the same, except for the particular keywords and signs recognized
- Compiler-compilers produce fixed lexical analysis routines for use in the generated compiler
- These routines differ only in the list of keywords recognized and this list is all that needs to be supplied by the user

Some useful compiler-construction tools

- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines

Scanner generators

- Automatically generate lexical analyzers, normally from a specification based on regular expressions
- The basic organization is a finite automata

Parser generators

- Produces syntax analyzers, normally from input that is based on a CFG
- In early compilers, syntax analysis consumed not only a large fraction of the running time of the compiler, but also a large fraction of the intellectual effort of writing a compiler
- This phase is now considered one of the easiest to implement

Syntax-directed translation engines

- These produce collections of routines that walk through the parse tree, generating the intermediate code
- One or more translations are associated with each node of the parse tree and
- Each translation is defined in terms of the translation at its neighbor nodes in the tree

Automatic code generators

- This takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine
- The rules must include sufficient detail that we can handle the different possible access methods for data
- The basic technique is “template matching”
- The intermediate code statements are replaced by “templates” that represent sequence of machine instructions, in such a way that the assumptions about storage variables match from template to template
- Since there are many options regarding where variables are to be placed, there are many possible ways to “tile” intermediate code with a given set of templates and it is necessary to select a good tiling without a combinatorial explosion in running time of the compiler

Data flow engines

- Good code optimization involves “data-flow analysis”, the gathering of information about how values are transmitted from one part of a program to each other part
- Different tasks of this nature can be performed by the same routine, with the user supplying details of relationship between intermediate code statements and the information being gathered

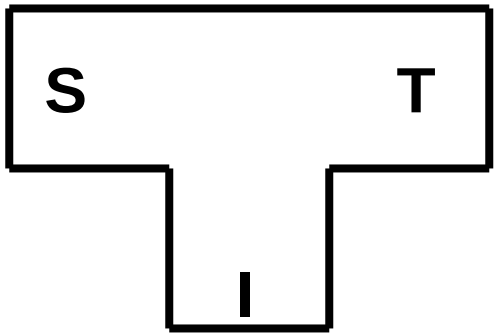
Approaches to compiler development

- There are several general approaches that a compiler writer can adopt to implement a compiler
- Simplest technique : Retarget or rehost and existing compiler
- If there are no suitable existing compiler, the compiler writer might adopt the organization of a known compiler for a similar language and implement the corresponding components, using component-generation tools or implementing them by hand
- It is relatively rare that a completely new compiler organization is required

- Prefer to write the compiler in a friendlier language than the assembly language
- In the UNIX programming environment, compiles are usually written in C
- Even the C compilers are written in C
- Using the facilities offered by a language to compile itself is the essence of boot strapping
  - Boot strapping can be used to create compilers and to move them from one machine to another by modifying the backend

Boot Strapping

- We assume that a compiler is characterized by three languages :
  - The source language S that it compiles
  - The target language T that it generates code for
  - The implementation language I that it is written in



- A T-diagram, because of its shape

- Within text written as $S_I T$
- The three languages may all be quite different
- A compiler may run on one machine and produce the target code for another machine
    - Cross Compiler

- Example 1
- Write a cross-compiler for a new language L, in implementation language S, to generate code for the machine N

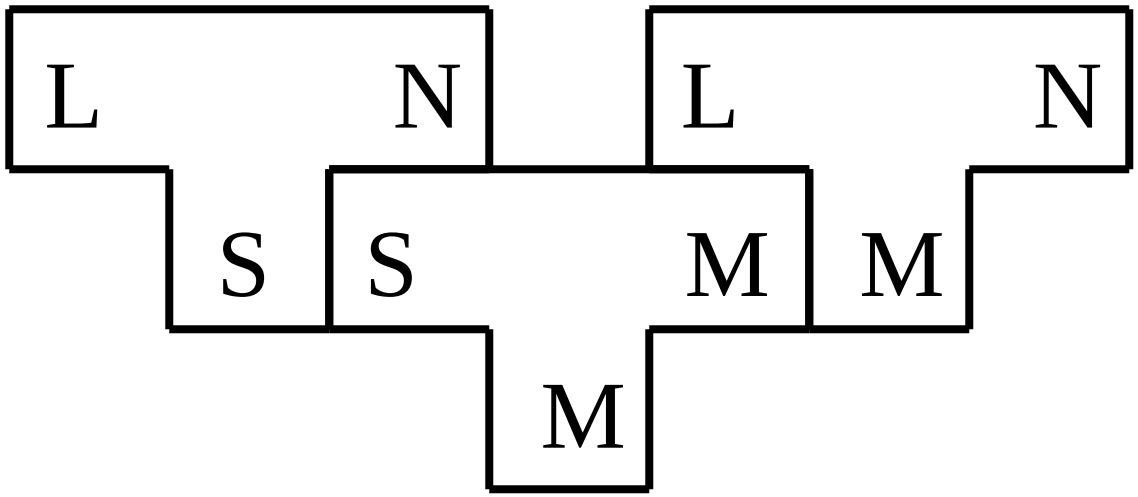
- Example 2 :
- An existing compiler for S runs on machine M and generates the code for M

Source Language – S

Implementation Language - M

Target Language – M

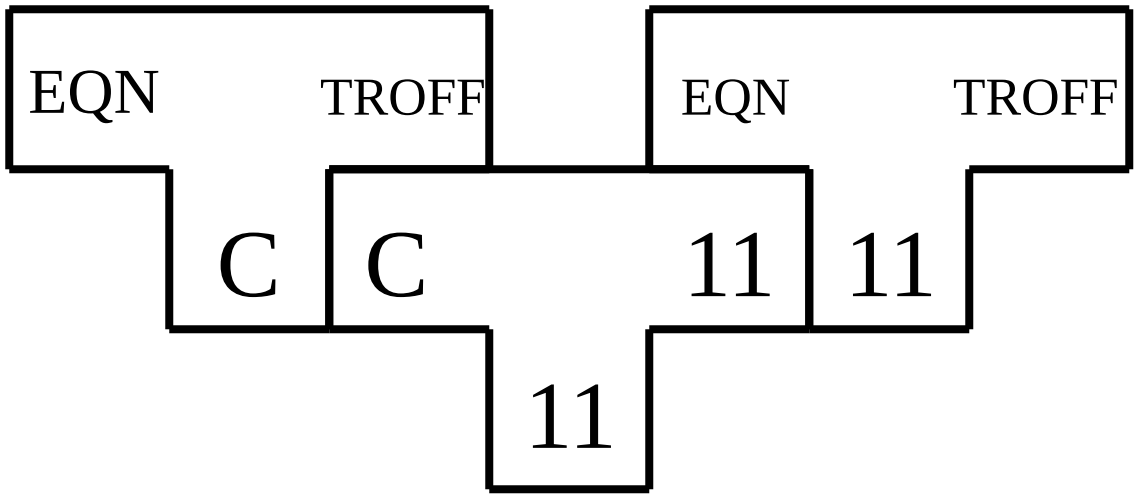
- Example -3
- What happens if $L_S N$ is run through $S_M M$



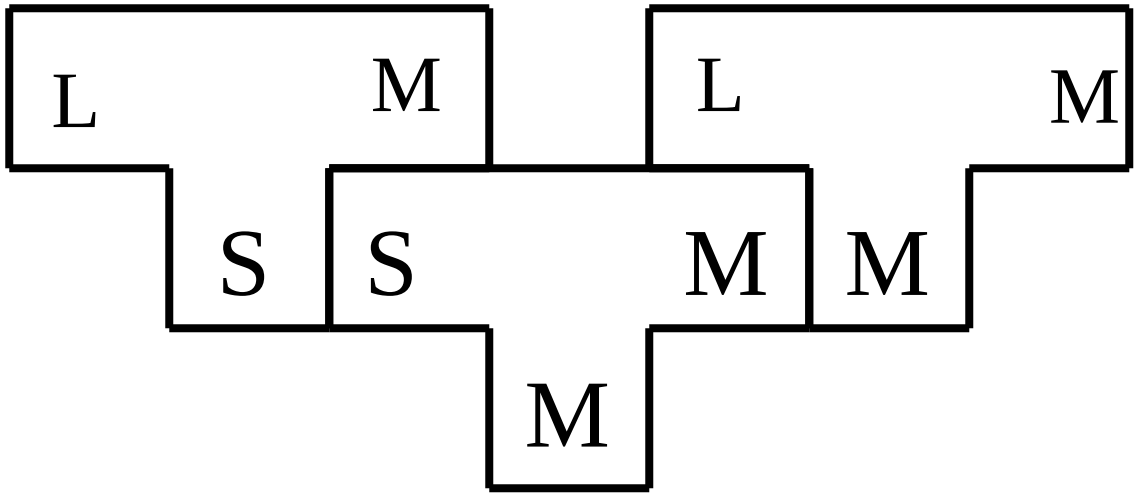
- This is what we call “Compiling a compiler”
- When T-diagrams are put together, note that
  - The implementation language S of the compiler L_SN must be the same as that source language of the existing compiler S_MM
  - The target language M of the existing compiler S_MM must be same as that of the implementation language of the translated form (new language) L_MN
- A trio of T-diagram can be thought of as an equation

$$L_SN + S_MM = L_MN$$

- Example : 11.1
- The first version of the EQN compiler had C as the implementation language and generated commands for the text formatter TROFF
- Assume that there exists a C language compiler written in PDP-11 which generates code for PDP-11 machine
- What is the cross-compiler obtained?

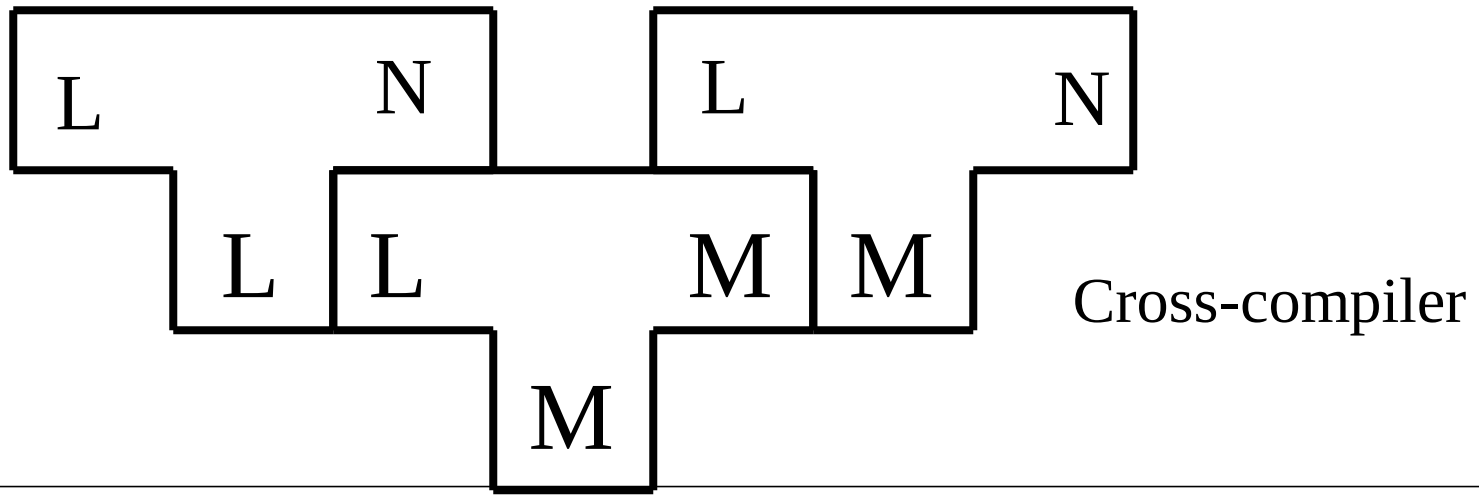


- Boot strapping builds a compiler for larger and larger subsets of a language
- Suppose we are developing a new language L which is to be run on the machine M. So, we want the compiler of L to be written in M
- But assume that L is a very large language
- So, consider a subset of L, namely S
- Write the compiler for S in M that run on M
- Now write the compiler for L in the language S which generates the target code M

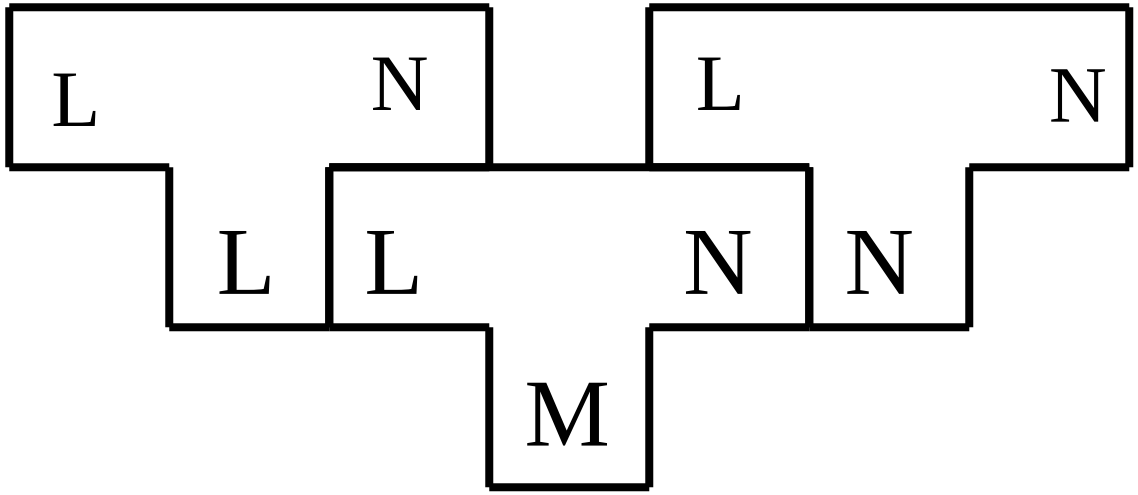

L
M
L
M
S
S
M
M
M

- Wirth notes that Pascal was first implemented by writing a compiler in Pascal itself
- The compiler was then translated “by hand” into an available low-level language without any attempt at optimization
- The compiler was for a subset “(>60 percent)” of Pascal; several boot strapping stages later a compiler for all of Pascal was obtained
- Lecarme and Peyrolle-Thomas[1978] summarize methods that have been used to boot strap Pascal compilers

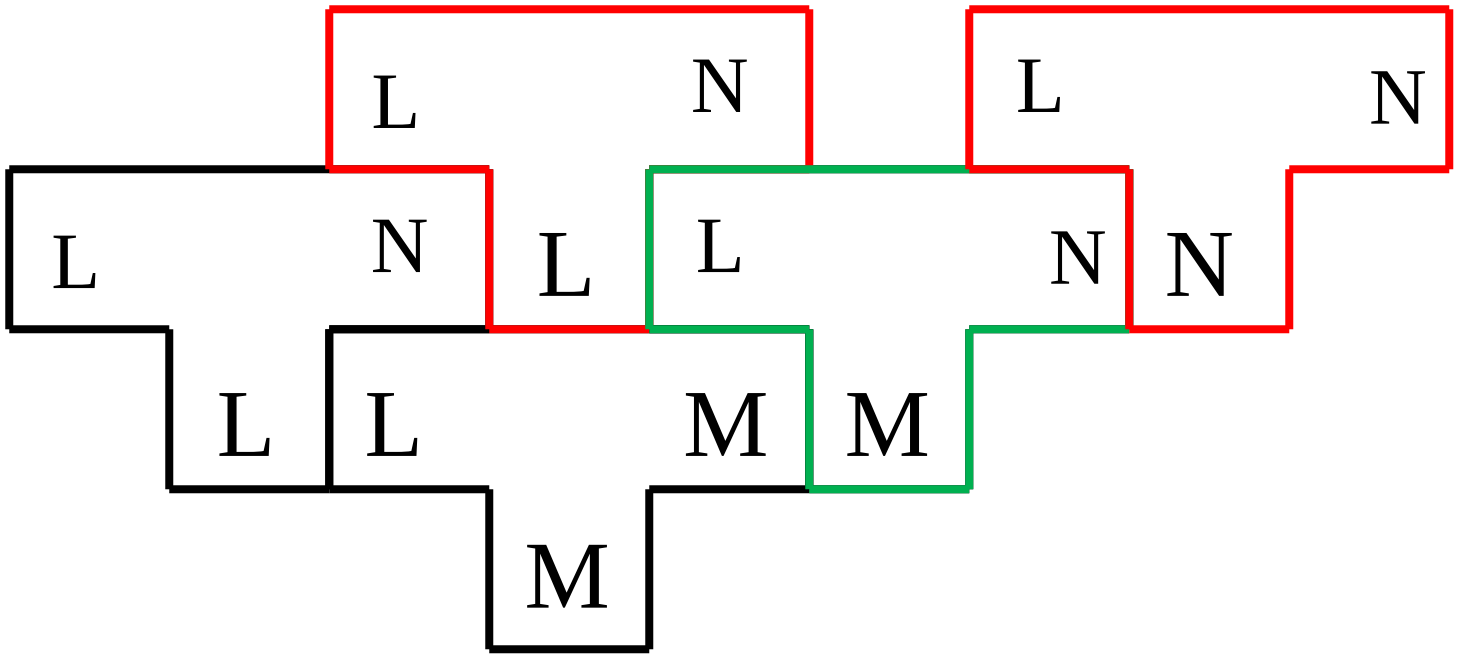
- For the advantages of boot strapping to be realized fully, a compiler has to be written in the language it compiles
- Suppose we have a compiler for a language L which generates the code for the machine N and is written in the language L itself (self-hosting)
- Also, assume that we have another compiler for L written in M which generates the code for M (boot strapping)
- What happens of these are run together?



Cross-compiler



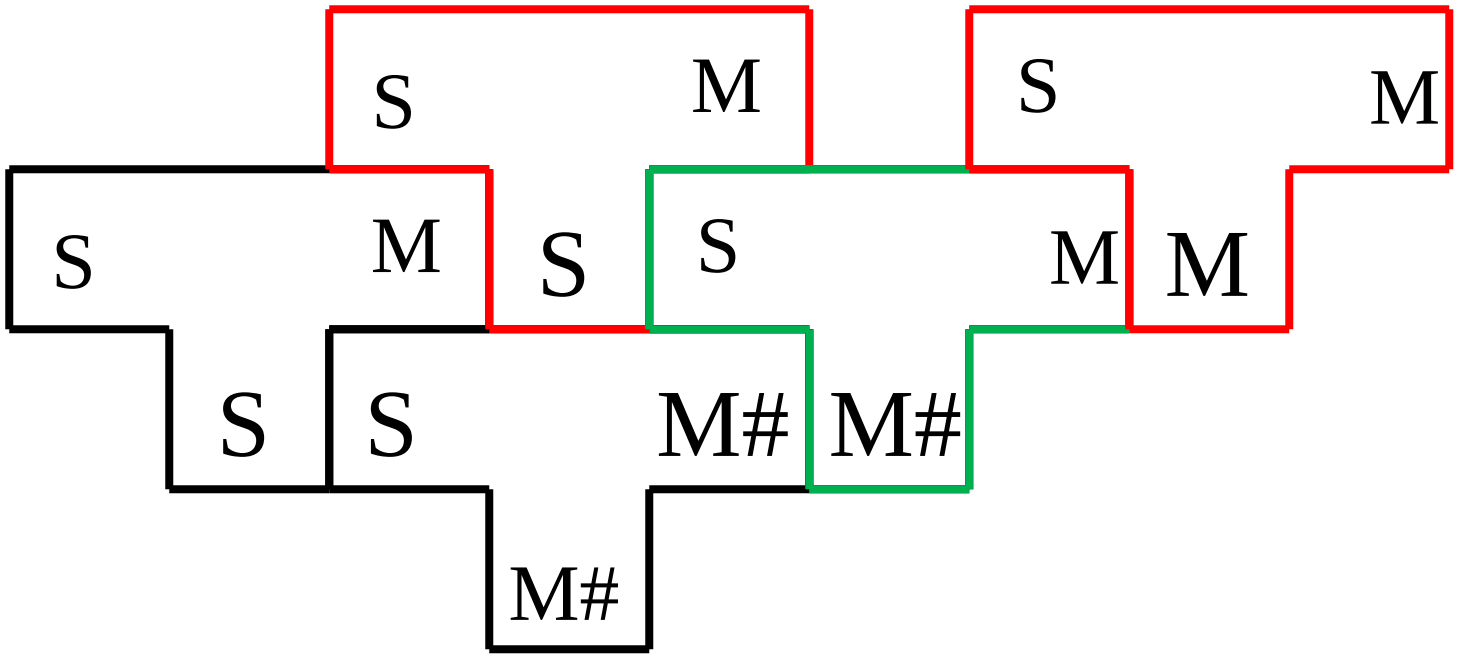
- This example is motivated by the development of the Fortran H compiler
- The compiler was itself written in Fortran and boostrapped three times
- The first time was to convert from running on the IBM 7004 to System/360 – an arduous procedure
- The second time was to optimize itself, which reduced the size of the compiler from about 550K to about 400K bytes



Boot strapping a compiler

- Using boot strapping techniques, an optimizing compiler can optimize itself

- Suppose all development is done on machine M
- We have $S_S M$, a good optimizing compiler for a language S written in S
- We want $S_M M$, a good optimizing compiler for S written in M
- For this, we can create $S_{M\#} M\#$, a quick-and-dirty compiler for S on M that not only generates a poor code, but also takes a long time to do so (M# indicates a poor implementation of M)
- However, we can use the indifferent compiler $S_{M\#} M\#$ to obtain a good compiler for S in two steps



- Ammann [1981] describes how a clean implementation of Pascal was obtained by a process similar to that of above
- Revisions to Pascal led to a fresh compiler being written in 1972 for the CDC 6000 series machines