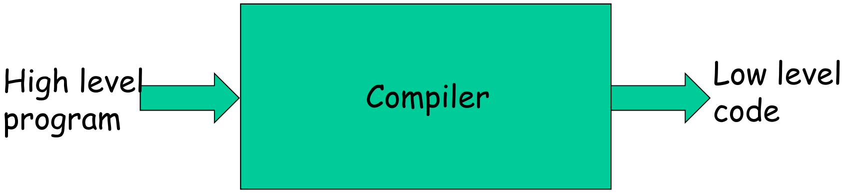
# What are Compilers?

- Translates from one representation of the program to another
- Typically from high level source code to low level machine code or object code
- Source code is normally optimized for human readability
  - Expressive: matches our notion of languages (and application?!)
  - Redundant to help avoid programming errors
- Machine code is optimized for hardware
  - Redundancy is reduced
  - Information about the intent is lost

# How to translate?

- Source code and machine code mismatch in level of abstraction
- Some languages are farther from machine code than others
- Goals of translation
  - Good performance for the generated code
  - Good compile time performance
  - Maintainable code
  - High level of abstraction
- Correctness is a very important issue. Can compilers be proven to be correct? Very tedious!
- However, the correctness has an implication on the development cost



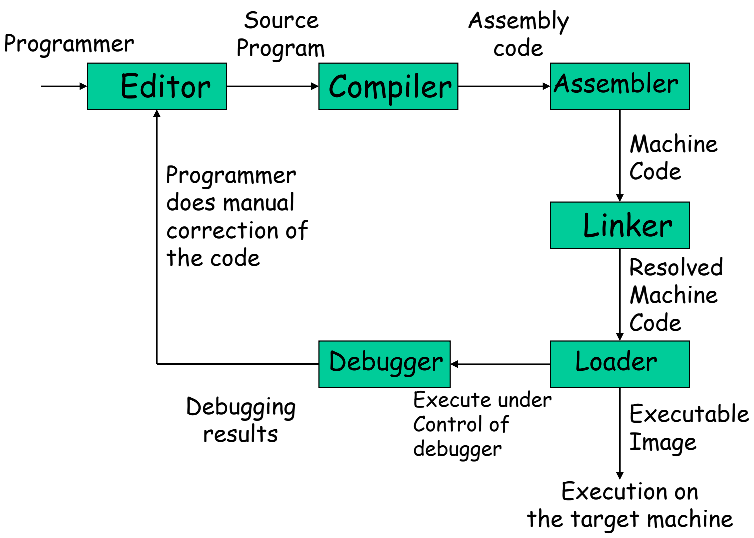
High level program

Compiler

Low level code

# The big picture

- Compiler is part of program development environment
- The other typical components of this environment are editor, assembler, linker, loader, debugger, profiler etc.
- The compiler (and all other tools) must support each other for easy program development



Programmer

Source Program

Assembly code

Editor

Compiler

Assembler

Machine Code

Linker

Resolved Machine Code

Loader

Debugger

Programmer does manual correction of the code

Debugging results

Execute under Control of debugger

Executable Image

Execution on the target machine

Normally end up with error

# How to translate easily?

- Translate in steps. Each step handles a reasonably simple, logical, and well defined task
- Design a series of program representations
- Intermediate representations should be amenable to program manipulation of various kinds (type checking, optimization, code generation etc.)
- Representations become more machine specific and less language specific as the translation proceeds

# The first few steps

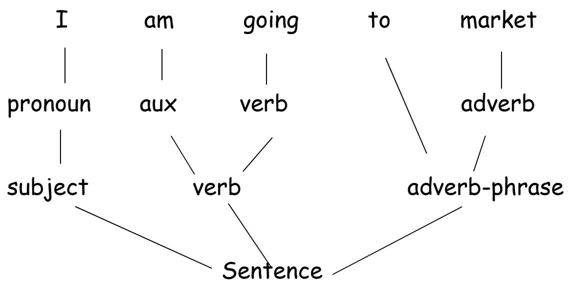
- The first few steps can be understood by analogies to how humans comprehend a natural language
- The first step is recognizing/knowing alphabets of a language. For example
  - English text consists of lower and upper case alphabets, digits, punctuations and white spaces
  - Written programs consist of characters from the ASCII characters set (normally 9-13, 32-126)
- The next step to understand the sentence is recognizing words (lexical analysis)
  - English language words can be found in dictionaries
  - Programming languages have a dictionary (keywords etc.) and rules for constructing words (identifiers, numbers etc.)

# Lexical Analysis

- Recognizing words is not completely trivial. For example:
  ist his ase nte nce?
- Therefore, we must know what the word separators are
- The language must define rules for breaking a sentence into a sequence of words.
- Normally white spaces and punctuations are word separators in languages.
- In programming languages a character from a different class may also be treated as word separator.
- The lexical analyzer breaks a sentence into a sequence of words or tokens:
  - If a == b then a = 1 ; else a = 2 ;
  - Sequence of words (total 14 words)
    if  a  ==  b  then  a  =  1  ;  else  a  =  2  ;

# The next step

- Once the words are understood, the next step is to understand the structure of the sentence
- The process is known as syntax checking or parsing



I am going to market

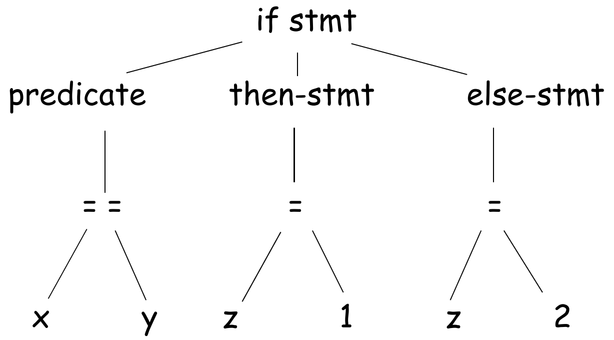
pronoun aux verb adverb

subject verb adverb-phrase

Sentence

# Parsing

- Parsing a program is exactly the same as shown in previous slide.
- Consider an expression
  if x == y then z = 1 else z = 2



if stmt

predicate then-stmt else-stmt

= = = =

x y z 1 z 2

# Understanding the meaning

- Once the sentence structure is understood we try to understand the meaning of the sentence (semantic analysis)
- Example:
  Prateek said Nitin left his assignment at home
- What does his refer to? Prateek or Nitin?
- Even worse case
  Amit said Amit left his assignment at home
- How many Amits are there? Which one left the assignment?

# Semantic Analysis

- Too hard for compilers. They do not have capabilities similar to human understanding
- However, compilers do perform analysis to understand the meaning and catch inconsistencies
- Programming languages define strict rules to avoid such ambiguities

```
{ int Amit = 3;
     { int Amit = 4;
        cout << Amit;
     }
}
```

# More on Semantic Analysis

- Compilers perform many other checks besides variable bindings

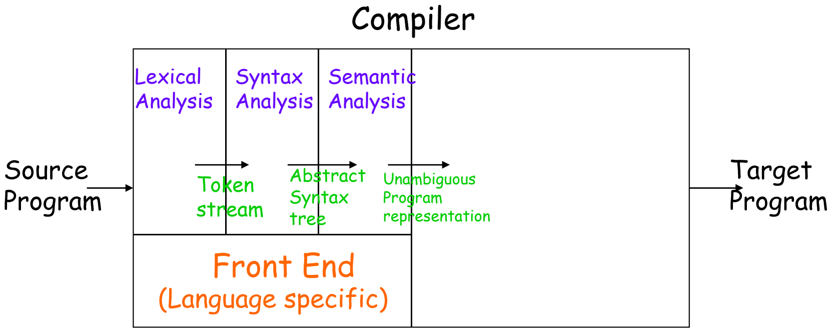
- Type checking

  Amit left her work at home

  Torid/Valdis/Torbjorg left her bag in the car

- There is a type mismatch between her and Amit. Presumably Amit is a male. And they are not the same person.

# Compiler structure once again



# Front End Phases

- Lexical Analysis
  - Recognize tokens and ignore white spaces, comments

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i | f | | ( | x | 1 | | * | x | 2 | < | 1 | . | 0 | ) | { |

Generates token stream

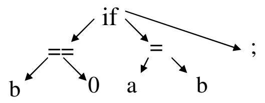
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| if | ( | x1 | * | x2 | < | 1.0 | ) | { |

  - Error reporting
  - Model using regular expressions
  - Recognize using Finite State Automata

# Syntax Analysis

- Check syntax and construct abstract syntax tree

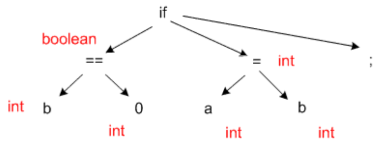
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| if | ( | b | == | 0 | ) | a | = | b | ; |



- Error reporting and recovery
- Model using context free grammars
- Recognize using Push down automata/Table Driven Parsers

# Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate overloaded operators
- Type coercion
- Static checking
  - Type checking
  - Control flow checking
  - Unique ness checking
  - Name checks



# Code Optimization

- No strong counter part with English, but is similar to editing/précis writing
- Automatically modify programs so that they
  - Run faster
  - Use less resources (memory, registers, space, fewer fetches etc.)
- Some common optimizations
  - Common sub-expression elimination
  - Copy propagation
  - Dead code elimination
  - Code motion
  - Strength reduction
  - Constant folding
- Example: x = 15 * 3 is transformed to x = 45

# Example of Optimizations

```
    PI = 3.14159                        3A+4M+1D+2E
    Area = 4 * PI * R^2
    Volume = (4/3) * PI * R^3
--------------------------------
    X = 3.14159 * R * R                 3A+5M
    Area = 4 * X
    Volume = 1.33 * X * R
--------------------------------
    Area = 4 * 3.14159 * R * R          2A+4M+1D
    Volume = ( Area / 3 ) * R
--------------------------------
    Area = 12.56636 * R * R             2A+3M+1D
    Volume = ( Area /3 ) * R
--------------------------------
    X = R * R                           3A+4M
    Area = 12.56636 * X
    Volume = 4.18879 * X * R
```

| | |
|---|---|
| A : assignment | M : multiplication |
| D : division | E : exponent |

# Code Generation

- Usually a two step process
  - Generate intermediate code from the semantic representation of the program
  - Generate machine code from the intermediate code
- The advantage is that each phase is simple
- Requires design of intermediate language
- Most compilers perform translation between successive intermediate representations
- Intermediate languages are generally ordered in decreasing level of abstraction from highest (source) to lowest (machine)
- However, typically the one after the intermediate code generation is the most important

# Intermediate Code Generation

- Abstraction at the source level
  identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)

- Abstraction at the target level
  memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems

- Code generation is mapping from source level abstractions to target machine abstractions

# Intermediate Code Generation …

- Map identifiers to locations (memory/storage allocation)

- Explicate variable accesses (change identifier reference to relocatable/absolute address

- Map source operators to opcodes or a sequence of opcodes

- Convert conditionals and iterations to a test/jump or compare instructions

# Intermediate Code Generation …

- Layout parameter passing protocols: locations for parameters, return values, layout of activations frame etc.

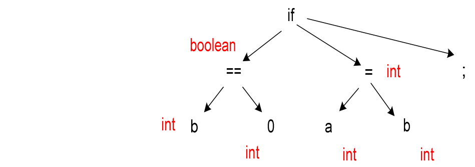
- Interface calls to library, runtime system, operating systems

# Post translation Optimizations

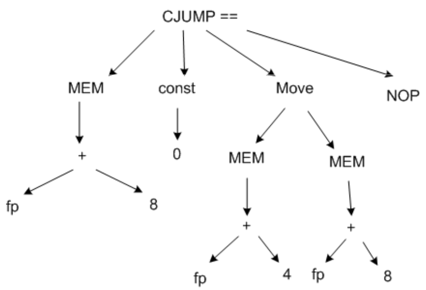
- Algebraic transformations and re-ordering
  - Remove/simplify operations like
    - Multiplication by 1
    - Multiplication by 0
    - Addition with 0
  - Reorder instructions based on
    - Commutative properties of operators
    - For example x+y is same as y+x (always?)

## Instruction selection

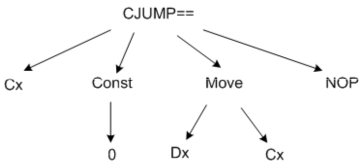
- Addressing mode selection
- Opcode selection
- Peephole optimization



## Intermediate code generation



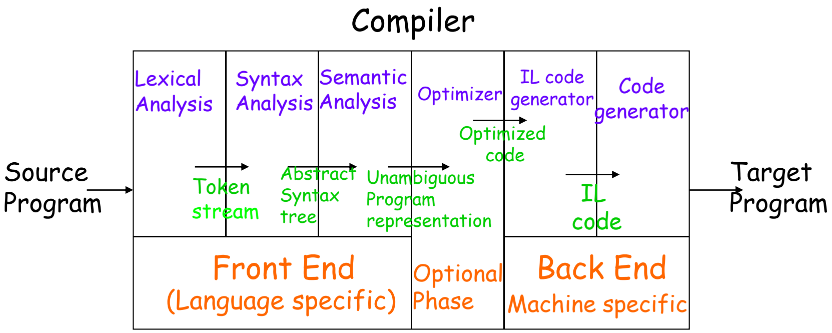
Optimization



### Code Generation

```
CMP Cx, 0
CMOVZ  Dx,Cx
```

# Compiler structure



Compiler

Source Program

Lexical Analysis

Token stream

Syntax Analysis

Abstract Syntax tree

Semantic Analysis

Unambiguous Program representation

Optimizer

Optimized code

IL code generator

IL code

Code generator

Target Program

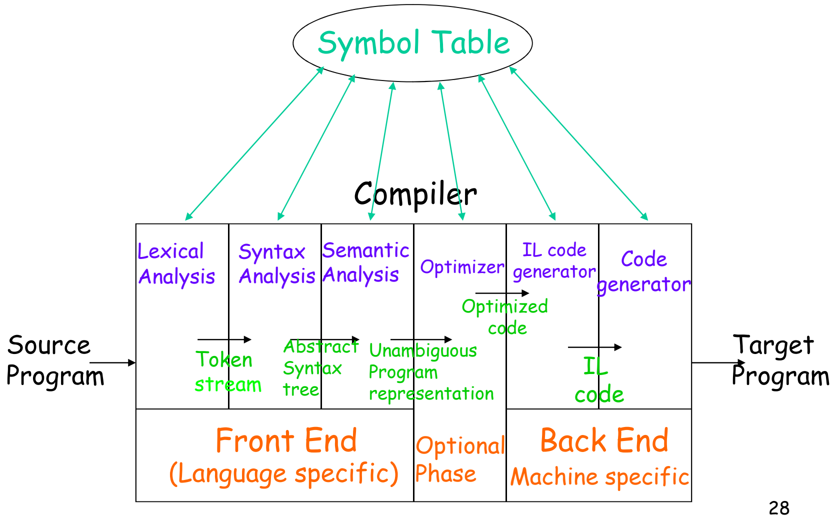
Front End (Language specific)

Optional Phase

Back End Machine specific

- Information required about the program variables during compilation
  - Class of variable: keyword, identifier etc.
  - Type of variable: integer, float, array, function etc.
  - Amount of storage required
  - Address in the memory
  - Scope information
- Location to store this information
  - Attributes with the variable (has obvious problems)
  - At a central repository and every phase refers to the repository whenever information is required
- Normally the second approach is preferred
  - Use a data structure called symbol table

# Final Compiler structure



Symbol Table

Compiler

Source Program

Lexical Analysis

Token stream

Syntax Analysis

Abstract Syntax tree

Semantic Analysis

Unambiguous Program representation

Optimizer

Optimized code

IL code generator

IL code

Code generator

Target Program

Front End (Language specific)

Optional Phase

Back End Machine specific

# Advantages of the model

- Also known as Analysis-Synthesis model of compilation
  - Front end phases are known as analysis phases
  - Back end phases are known as synthesis phases
- Each phase has a well defined work
- Each phase handles a logical activity in the process of compilation

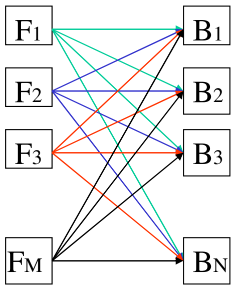
# Advantages of the model …

- Compiler is re-targetable
- Source and machine independent code optimization is possible.
- Optimization phase can be inserted after the front and back end phases have been developed and deployed

# Issues in Compiler Design

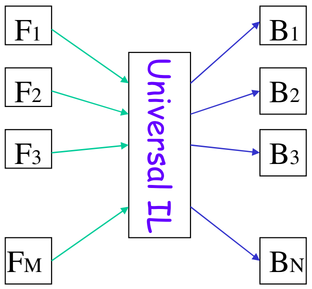
- Compilation appears to be very simple, but there are many pitfalls
- How are erroneous programs handled?
- Design of programming languages and architectures have a big impact on the complexity of the compiler
- M*N vs. M+N problem
  - Compilers are required for all the languages and all the machines
  - For M languages and N machines we need to develop M*N compilers
  - However, there is lot of repetition of work because of similar activities in the front ends and back ends
  - Can we design only M front ends and N back ends, and some how link them to get all M*N compilers?

# M*N vs M+N Problem



Requires M*N compilers

Universal Intermediate Language



Requires M front ends
And N back ends

# Universal Intermediate Language

- Universal Computer/Compiler Oriented Language (UNCOL)
  - a vast demand for different compilers, as potentially one would require separate compilers for each combination of source language and target architecture. To counteract the anticipated combinatorial explosion, the idea of a linguistic switchbox materialized in 1958
  - UNCOL (UNiversal COmputer Language) is an intermediate language, which was proposed in 1958 to reduce the developmental effort of compiling many different languages to different architectures

# Universal Intermediate Language …

- The first intermediate language UNCOL (UNiversal Computer Oriented Language) was proposed in 1961 for use in compilers to reduce the development effort of compiling many different languages to many different architectures

- the IR semantics should ideally be independent of both the source and target language (i.e. the target processor) Accordingly, already in the 1950s many researchers tried to define a single universal IR language, traditionally referred to as UNCOL (UNiversal Computer Oriented Language)

  - it is next to impossible to design a single intermediate language to accommodate all programming languages
  - Mythical universal intermediate language sought since mid 1950s (Aho, Sethi, Ullman)
- However, common IRs for similar languages, and similar machines have been designed, and are used for compiler development

# How do we know compilers generate correct code?

- Prove that the compiler is correct.

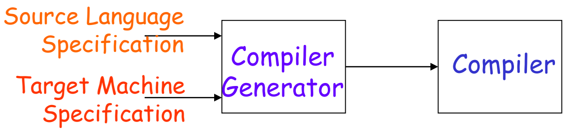
- However, program proving techniques do not exist at a level where large and complex programs like compilers can be proven to be correct

- In practice do a systematic testing to increase confidence level

- Regression testing
  - Maintain a suite of test programs
  - Expected behavior of each program is documented
  - All the test programs are compiled using the compiler and deviations are reported to the compiler writer
- Design of test suite
  - Test programs should exercise every statement of the compiler at least once
  - Usually requires great ingenuity to design such a test suite
  - Exhaustive test suites have been constructed for some languages

# How to reduce development and testing effort?

- DO NOT WRITE COMPILERS
- GENERATE compilers
- A compiler generator should be able to "generate" compiler from the source language and target machine specifications



Source Language Specification

Target Machine Specification

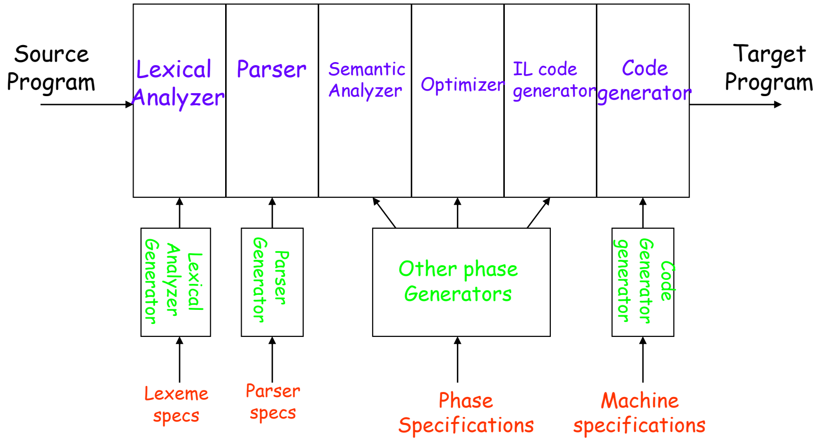
Compiler Generator

Compiler

# Specifications and Compiler Generator

- How to write specifications of the source language and the target machine?
  - Language is broken into sub components like lexemes, structure, semantics etc.
  - Each component can be specified separately. For example, an identifier may be specified as
    - A string of characters that has at least one alphabet
    - starts with an alphabet followed by alphanumeric
    - letter (letter|digit)*
  - Similarly syntax and semantics can be described
- Can target machine be described using specifications?

# Tool based Compiler Development



Source Program

Lexical Analyzer

Parser

Semantic Analyzer

Optimizer

IL code generator

Code generator

Target Program

Lexical Analyzer Generator

Parser Generator

Other phase Generators

Code Generator generator

Lexeme specs

Parser specs

Phase Specifications

Machine specifications

# How to Retarget Compilers?

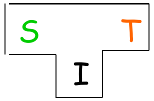
- Changing specifications of a phase can lead to a new compiler
  - If machine specifications are changed then compiler can generate code for a different machine without changing any other phase
  - If front end specifications are changed then we can get compiler for a new language
- Tool based compiler development cuts down development/maintenance time by almost 30-40%
- Tool development/testing is one time effort
- Compiler performance can be improved by improving a tool and/or specification for a particular phase

# Bootstrapping

- Compiler is a complex program and should not be written in assembly language
- How to write compiler for a language in the same language (first time!)?
- First time this experiment was done for Lisp
- Initially, Lisp was used as a notation for writing functions.
- Functions were then hand translated into assembly language and executed
- McCarthy wrote a function eval[e] in Lisp that took a Lisp expression e as an argument
- The function was later hand translated and it became an interpreter for Lisp

# Bootstrapping …

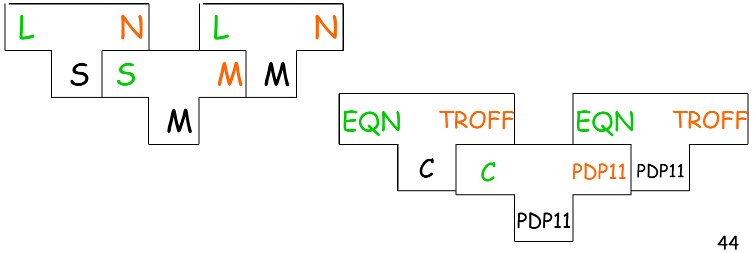
- A compiler can be characterized by three languages: the source language (S), the target language (T), and the implementation language (I)
- The three language S, I, and T can be quite different. Such a compiler is called cross-compiler
- This is represented by a T-diagram as:



- In textual form this can be represented as

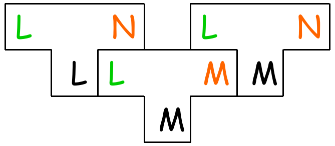
$S_IT$

- Write a cross compiler for a language L in implementation language S to generate code for machine N
- Existing compiler for S runs on a different machine M and generates code for M
- When Compiler $L_SN$ is run through $S_MM$ we get compiler $L_MN$

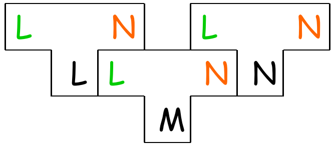


# Bootstrapping a Compiler

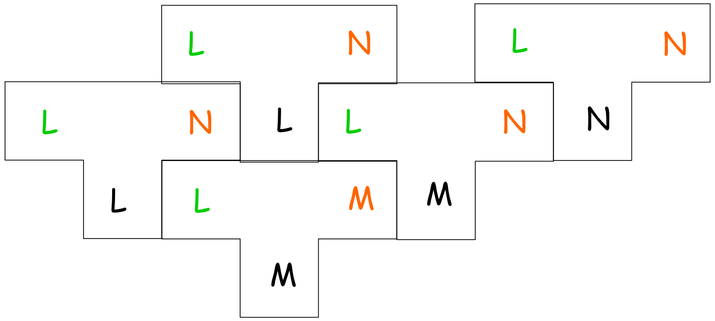
- Suppose $L_LN$ is to be developed on a machine M where $L_MM$ is available



- Compile $L_LN$ second time using the generated compiler



# Bootstrapping a Compiler: the Complete picture



# Compilers of the 21st Century

- Overall structure of almost all the compilers is similar to the structure we have discussed

- The proportions of the effort have changed since the early days of compilation

- Earlier front end phases were the most complex and expensive parts.

- Today back end phases and optimization dominate all other phases. Front end phases are typically a smaller fraction of the total time