

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

*- functions,
loops*

2

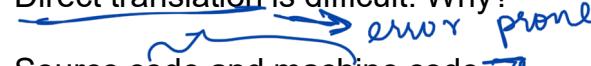
Goals of translation

- Good compile time performance
- Good performance for the generated code
- Correctness
 - A very important issue.
 - Can compilers be proven to be correct?
 - Tedious even for toy compilers!
 - Undecidable in general.
 - However, the correctness has an implication on the development cost

*How can we monitor the compiler's job?
⇒ Best code can't be generated.
⇒ Optimize basic blocks*

1

How to translate?

- Direct translation is difficult. Why?

- Source code and machine code mismatch in level of abstraction
 - Variables vs Memory locations/registers
 - Functions vs jump/return
 - Parameter passing
 - structs
- Some languages are farther from machine code than others
 - For example, languages supporting Object Oriented Paradigm

5

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

6

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)

7

The first few steps

- The next step to understand the sentence is recognizing words
 - How to recognize English words?
 - Words found in standard dictionaries
 - Dictionaries are updated regularly



ABOUT · OXFORD GLOBAL LANGUAGES · THE OED · PRESS AND NEWS

December 2016 ·

Around 500 new words, phrases, and senses have entered the Oxford English Dictionary this quarter, including *glam-ma*, *Youtuber*, and *upstander*.

We have a selection of release notes this December, each of which takes a closer look at some of our additions. The last few years have seen the emergence of the word *Brexit*, and you can read more about the huge increase in the use of the word, and how we go about defining it, in [this article](#) by Craig Leyland.

8

The first few steps

- How to recognize words in a programming language?
 - a dictionary (of keywords etc.)
 - rules for constructing words (identifiers, numbers etc.)
- This is called lexical analysis
- Recognizing words is not completely trivial. For example:
w hat ist his se nte nce?

9

Lexical Analysis: Challenges

- 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.

10

Lexical Analysis: Challenges

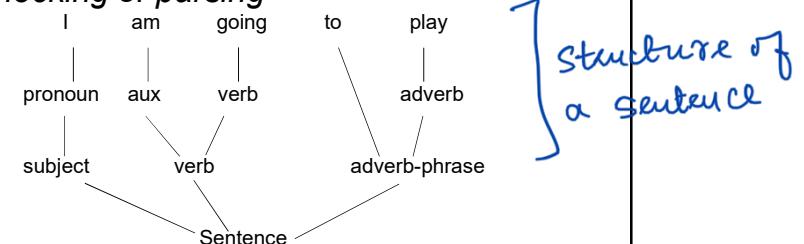
- In programming languages a character from a different class may also be treated as word separator. **x = y**
- The lexical analyzer breaks a sentence into a sequence of words or tokens:
 - If a == b then a = 1; else a = 2;
 - Sequence of words (total 14 words)
if a == b then a = 1 ; else a = 2 ;
~~~~~ Breaking words.

11

Now we know, it is a sentence.
Check if it is a correct sentence

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*

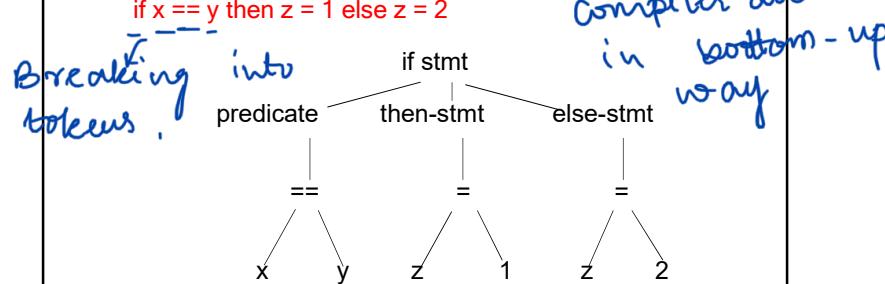


12

Parsing

- Parsing a program is exactly the same process as shown in previous slide.
- Consider an expression

`if x == y then z = 1 else z = 2`



13

Understanding the meaning

- Once the sentence structure is understood we try to understand the meaning of the sentence (semantic analysis) \Rightarrow meaning
- A challenging task
- Example:
`Prateek said Nitin left his assignment at home`
- What does `his` refer to? `Prateek` or `Nitin`?

14

Understanding the meaning

- Worse case
`Amit said Amit left his assignment at home`
- Even worse
`Amit said Amit left Amit's assignment at home`
- How many `Amits` are there? Which one left the assignment? Whose assignment got left?

15

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;
  }
}
```

16

Ambiguous semantics: No ordering
↳ left to the compiler in C

20-01-2021

Python: always left to right

More on Semantic Analysis

- Compilers perform many other checks besides variable bindings
- Type checking → Rules made for resolution
Amit left her work at home
- There is a type mismatch between **her** and **Amit**. Presumably **Amit** is a male. And they are not the same person.

17

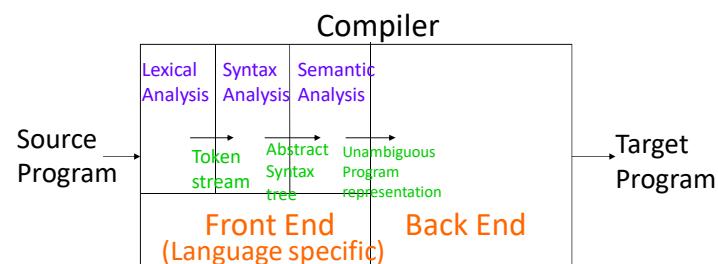
Example from Mahabharat

अश्वथामा हतः इति नरो वा कुञ्जरो वा

**“Ashwathama hathaha iti,
narova kunjarova”**

Ashwathama is dead. But, I am not certain whether it was a human or an elephant

Compiler structure once again



19

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 |) | {

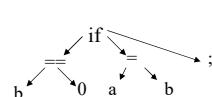
 - Error reporting
 - Model using regular expressions
 - Recognize using Finite State Automata

20

Syntax Analysis

- Check syntax and construct abstract syntax tree

`if (b == 0) a = b ;`



It finds
as many
errors as
possible

Trees and graphs
easier to
manipulate

- Error reporting and recovery
- Model using context free grammars → PDA
- Recognize using Push down automata/Table Driven Parsers → $O(N)$ time

PDA → slow.

21

We need to design our grammar
acc. to TDGs.



Semantic Analysis

- Check semantics

- Error reporting

- Disambiguate overloaded operators

- Type coercion → Implicit

- Static checking

- Type checking

- Control flow checking

- Unique ness checking

- Name checks

→ target to goto exists or
not

• Initialized or not

Name,
checkin
g [linking]

22

Code Optimization (optional Phase)

↙ for this course

- No strong counter part with natural language, but is like editing/précis writing (modifies program to make it run faster with less resources)

- Automatically modify programs so that they
 - Run faster
 - Use less resources (memory, registers, space, fewer fetches etc.)

24

Code Optimization

- Some common optimizations
 - Common sub-expression elimination
 - Copy propagation
 - { - Dead code elimination
 - Code motion
 - Strength reduction
 - Constant folding
- Example: $x = 15 * 3$ can be transformed to $x = 45$

Eliminate recomputations

Remove code that'll never be executed.

Replace key less time taking i wet
→ $\frac{1}{2} \times 2$

{ - Dead code elimination

Move code around
eg. something in loop
→ can be written outside.

* Pipeline stages
are never stalled
CS220

25

Example of Optimizations

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

PI = 3.14159
.Area = 4 * PI * R^2
.Volume = (4/3) * PI * R^3

3A+4M+1D+2E

X = 3.14159 * R * R
Area = 4 * X
Volume = 1.33 * X * R

3A+5M

Area = 4 * 3.14159 * R * R
Volume = (Area / 3) * R

2A+4M+1D

Area = 12.56636 * R * R
Volume = (Area / 3) * R

2A+3M+1D

X = R * R
Area = 12.56636 * X
Volume = 4.18879 * X * R

3A+4M

26

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

Target independent optimization

27

Code Generation

- Most compilers perform translation between successive intermediate representations
- Intermediate languages are generally ordered in decreasing level of abstraction from highest (source) to lowest (machine)

28

Code Generation

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

29

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
 - int, float
 - strings in python

30

Code Generation

- Convert conditionals and iterations to a test/jump or compare instructions
- Layout parameter passing protocols: locations for parameters, return values, layout of activations frame etc.
- Interface calls to library, runtime system, operating systems

31

LTO: Link Time Optimization
⇒ LLVM

Post translation Optimizations

- Algebraic transformations and reordering

– Remove/simplify operations like

- Multiplication by 1
- Multiplication by 0
- Addition with 0

$$\begin{array}{r} 999 - 5 + 5 \\ \hline 993 + 5 \end{array} \Rightarrow 998$$

– Reorder instructions based on

- Commutative properties of operators
- For example, $x+y$ is same as $y+x$ (always?)

limit = 1000

$$\begin{array}{r} 999 + 5 - 6 \\ \hline \downarrow \text{overflow} \end{array}$$

x and y can be expressions.

* Cases about overflow

$x+1$ can be smaller than x
⇒ Overflowing

32

Post translation Optimizations

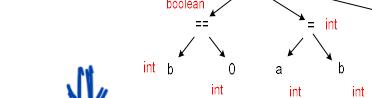
Instruction selection

- Addressing mode selection
- Opcode selection
- Peephole optimization

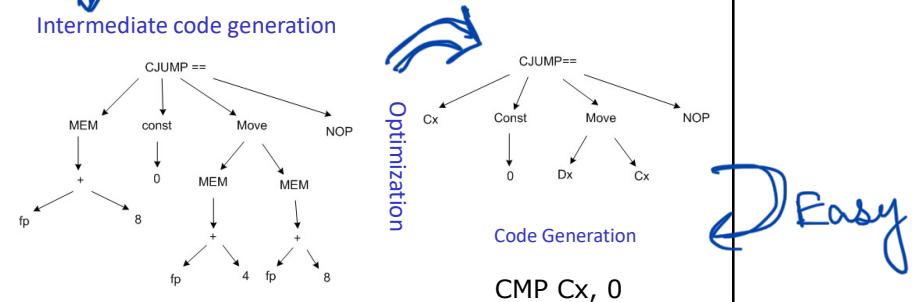
$$c[i,j] = \sum A[i,k] * B[k,j]$$

MAC
Multiply and accumulate.

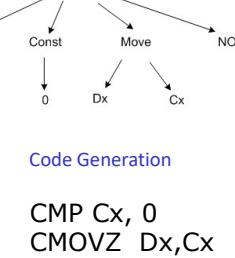
33



Intermediate code generation



Optimization

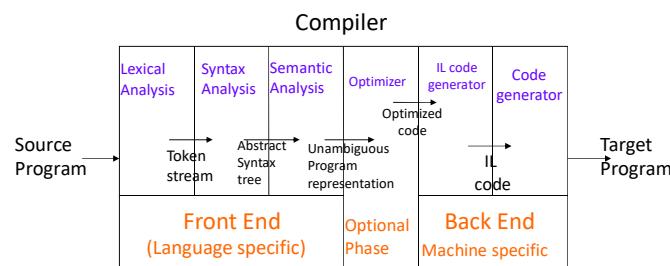


Easy

Storing in registers

34

Compiler structure



35

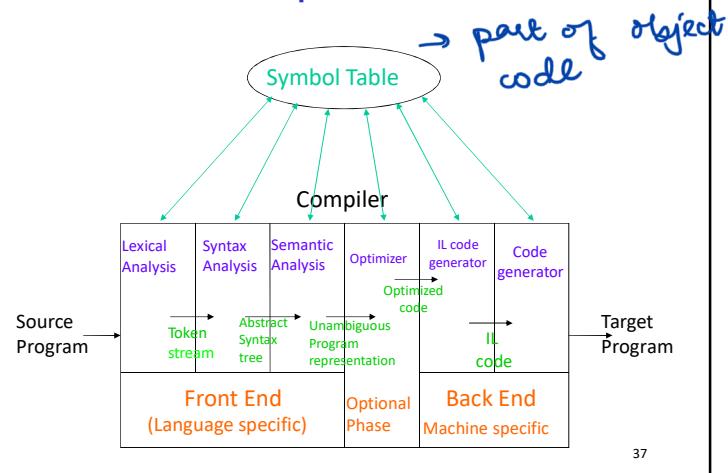
Something is missing

- 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

36

global repository

Final Compiler structure



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

38

Advantages of the model ...

- Compiler is re-targetable

I easily change your compiler to generate code for a diff' architecture
- 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

Change code generator & in bit of intermediate language for final target

Issues in Compiler Design

- Compilation appears to be very simple, but there are many pitfalls
- How are erroneous programs handled?
- Design of programming languages has 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 somehow link them to get all M^N compilers?

40

* Problem with Universal IL.

Each language handles diff. tasks efficiently
20-01-2021

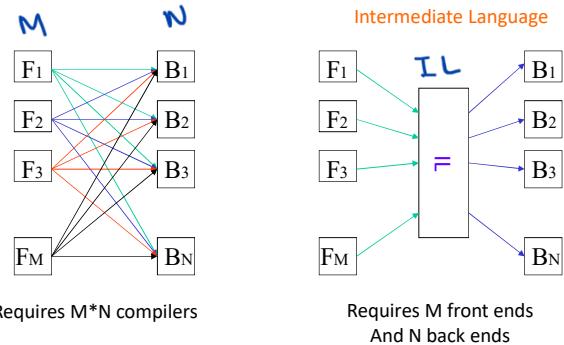
⇒ IL will have to be either machine code or union of all features in all languages.

→ Baffling

Universal Intermediate Language

To be able to handle all complexities.

M*N vs M+N Problem



41

- 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

42

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

43

• 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

source code
⇒ documented behaviour
⇒ Issues reported

• 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

44

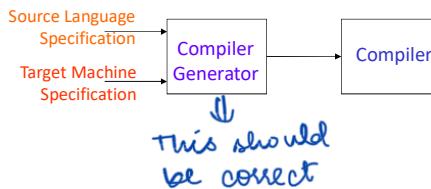
- Try to reuse existing things as much as possible.
- ↳ Try not to write new code. [Tested]

20-01-2021

* We generate phases of compilers

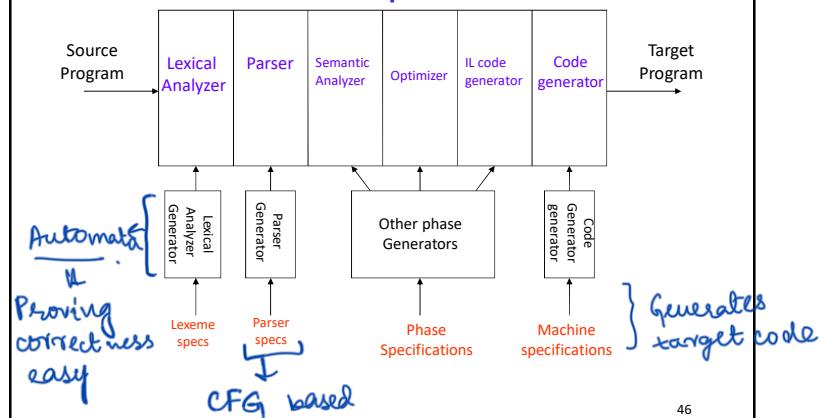
How to reduce development and testing effort?

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



45

Tool based Compiler Development



46

Q If we use HLL to compile our compiler, how was it originally written?

⇒ Kernel language set

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

47

Lisp → Functional

eval - written in LISP

e: also LISP argument

let the LISP expr be the compiler itself, a compiler for LISP

⇒ eval[e] ⇒ generates a compiler

compiler for LISP

Now this can be used to compile other LISP codes.

Bootstrap



→ means of solving issues using existing resources

Image By: No machine-readable author provided. Tarquin~commonswiki assumed (based on copyright claims). - No machine-readable source provided. Own work assumed (based on copyright claims)., CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=105468>

* Cross compiler?

Bootstrapping: Example

- Lets solve a simpler problem first
- Existing architecture and C compiler:
- gcc-x86 compiles C language to x86 Architecture
- New architecture:
- x335 → we want C compiler for this
- How to develop cc-x335?
- runs on x335, generates code for x335 (Assembly / object code)

49

Bootstrapping: Example

- How to develop cc-x335?
- Write a C compiler in C that emits x335 code → Target code has to be x335.
- Compile using gcc-x86 on x86 machine → we get a.out, if we pass a C program we get x335
- We have a C compiler that emits x335 code
 - But are we done?
 - It runs on x86, not x335

Our a.out is compiled using gcc-x86 ⇒ runs only on x86. not x335

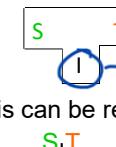
Bootstrapping: Example

- We have cc-x86-x335
 - Compiler runs on x86, generated code runs on x335.
- Compile the source code of C compiler with cc-x86-x335 → runs on x86, a.out runs on x335
- There it is
 - the output is a binary that runs on x335
 - this binary is the desired compiler : cc-x335

language of input program
I: language of compiler

Generated by compiler
S: language to be compiled
T: a.out language (machine)
I: language of compiler

- A compiler can be characterized by three languages: the source language (S), the target language (T), and the implementation language (I) [assembly for the target architecture where we are going to run our programs, executable]
- 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:



S

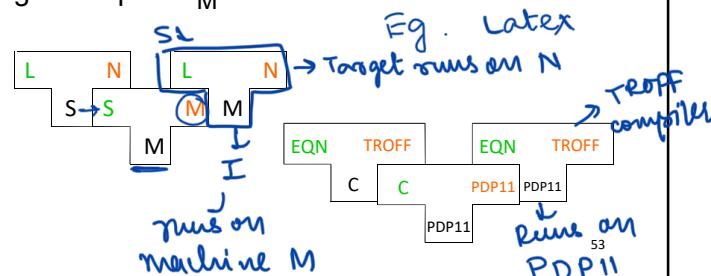
I

T

- In textual form this can be represented as

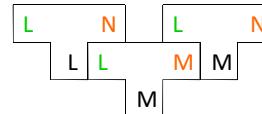
52

- 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** → **T**
- When Compiler **L_SN** is run through **S_MM** we get compiler **L_MN**

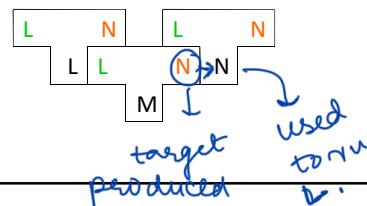


Bootstrapping a Compiler

- Suppose **L_NN** is to be developed on a machine **M** where **L_MM** is available

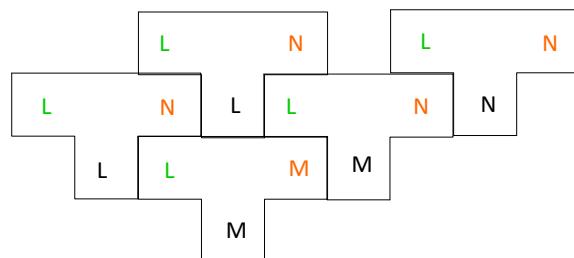


- Compile **L_NN** second time using the generated compiler



54

Bootstrapping a Compiler: the Complete picture



55

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

56