History of compiler development

1953 IBM develops the 701 EDPM (Electronic Data Processing Machine), the first general purpose computer, built as a “defense calculator” in the Korean War

History of compilers (cont'd)

No high-level languages were available, so all programming was done in machine and assembly language.

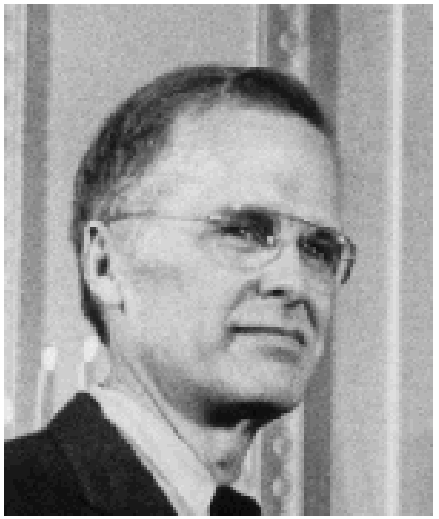
History of compilers (cont'd)

As expensive as these early computers were, most of the money companies spent was for software development, due to the complexities of assembly.

History of compilers (cont'd)

In 1953, John Backus came up with the idea of "speed coding", and developed the first interpreter. Unfortunately, this was 10-20 times slower than programs written in assembly.

He was sure he could do better.



John Backus

History of compilers (cont'd)

In 1954, Backus and his team released a research paper titled "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN."

The initial release of FORTRAN was in 1956, totaling 25,000 lines of assembly code. Compiled programs run almost as fast as handwritten assembly!

History of compilers (cont'd)

Projects that had taken two weeks to write now took only 2 hours. By 1958 more than half of all software was written in FORTRAN.

Why study compilers?

- You may never write a commercial compiler, but that's not why we study compilers. We study compiler construction for the following reasons:

Why study compilers? **(cont'd)**

1) Writing a compiler gives a student experience with large-scale applications development. Your compiler program may be the largest program you write as a student. Experience working with really big data structures and complex interactions between algorithms will help you out on your next big programming project

Why study compilers? **(cont'd)**

- Compiler writing is one of the shining triumphs of CS theory. It is very helpfull in the solutions of different problems.

Why study compilers? **(cont'd)**

3) Compiler writing is a basic element of programming language research. Many language researchers write compilers for the languages they design.

Why study compilers? **(cont'd)**

4) Many applications have similar properties to one or more phases of a compiler, and compiler expertise and tools can help an application programmer working on other projects besides compilers

Cousins Of The Compiler

1) Preprocessor.
2) Assembler
3) Loader and Link-editor.

PRE-PROCESSOR

Pre-processors produce input to compilers they might perform the following functions

I. Macro Processing

A pre-processor may allow a user define macros that are Short hands for a longer construction,pre-processor expand a Macros into source language statements.

PRE-PROCESSOR (cont'd)

II. File Inclusion

A pre-processor may include header files into the program text.

PRE-PROCESSOR (cont'd)

III. Rational pre-processors

These pre-processor augment older languages with more modern flow of control and data structuring facilities.

For example such a pre-processor might provide the user with built-in macros for constructs like while statements or if-statements ,where none exits in the program language itself.

PRE-PROCESSOR (cont'd)

IV. Language Extensions

These processors attempt add capabilities to the language.

For example, the language EQUEL is a database query language embedded in C.

2-ASSEMBLERS

Some compiler produce assembly code that is passed to an Assembler for further processing. Other compiler perform the Job of the assembler ,producing relocatable machine code That can be passed directly to the loader /link –editor

Assembly Code

Assembly code is a mnemonic version of machine code ,in Which names are used instead of binary code for operation And names are also given to memory addresses. A typical Sequence of assembly instruction might be :

Assembly Code (cont'd)

Move R1, a

Add R1, #2 

Move b, R1

Assembly Code (cont'd)

This code moves the contents of the address into register 1.Then adds the constant 2 to it and finally stores the result in the location named b.Thus it computes

$$b = a+2$$

Two –Pass Assembler

The simplest form of assembler makes two passes over the input, where a pass consists of reading on input file once.

In the first pass all the identifiers that denote storage locations are found and stored in a symbol table.

Identifiers are assigned storage locations as they are encountered for the first time. For example , the symbol table might contain the entries as follows.

Two –Pass Assembler (cont'd)

IDENTIFIER	ADDRESS
a	0
b	4

Two –Pass Assembler (cont'd)

Here we have assumed that a word ,consisting Of four bytes ,is set for each identifier,and that Addresses are assigned starting from byte 0.

Two –Pass Assembler (cont'd)

Second pass

In the second pass ,the assembler scans the input Again.This time it translates each operation code into the sequence of bits representing a location into the address given for the identifier in the symbol table. The output of the 2nd pass is usually relocatable m/c code.

3-Loaders and Link Editor

Using a program called a loader performs the Two functions of loading and link editing. The process of **loading consists of taking relocatable M/c** code, **Altering** the **relocatable addresses** and **Placing** the **altered** instructions and data in memory at **Proper locations**.

The link editor allows us to make a **Single program** from **several files of relocatable** m/c Code.

Loaders and Link Editor(cont'd)

These files may have been the result of several different compilations and one or more may be library files of routines provided by the system And available to any program that need them.

Modern Compilers

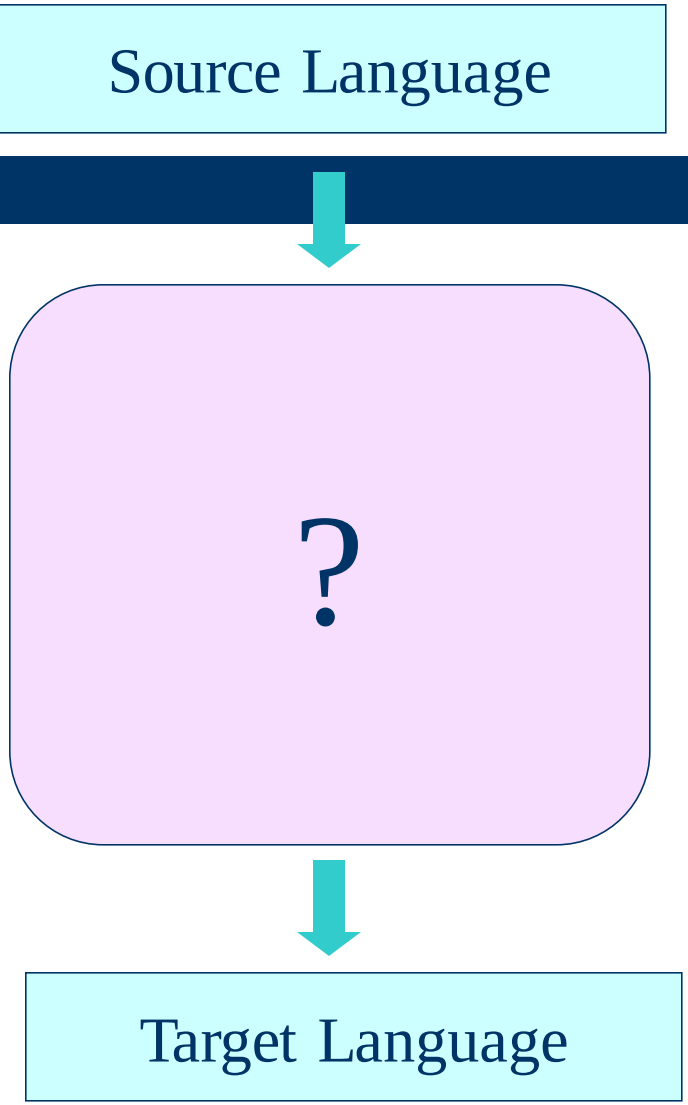
Compilers have not changed a great deal since the days of Backus. They still consist of two main components:

The **FRONT-END** reads in the program in the source languages, makes sense of it, and stores it in an internal representation…

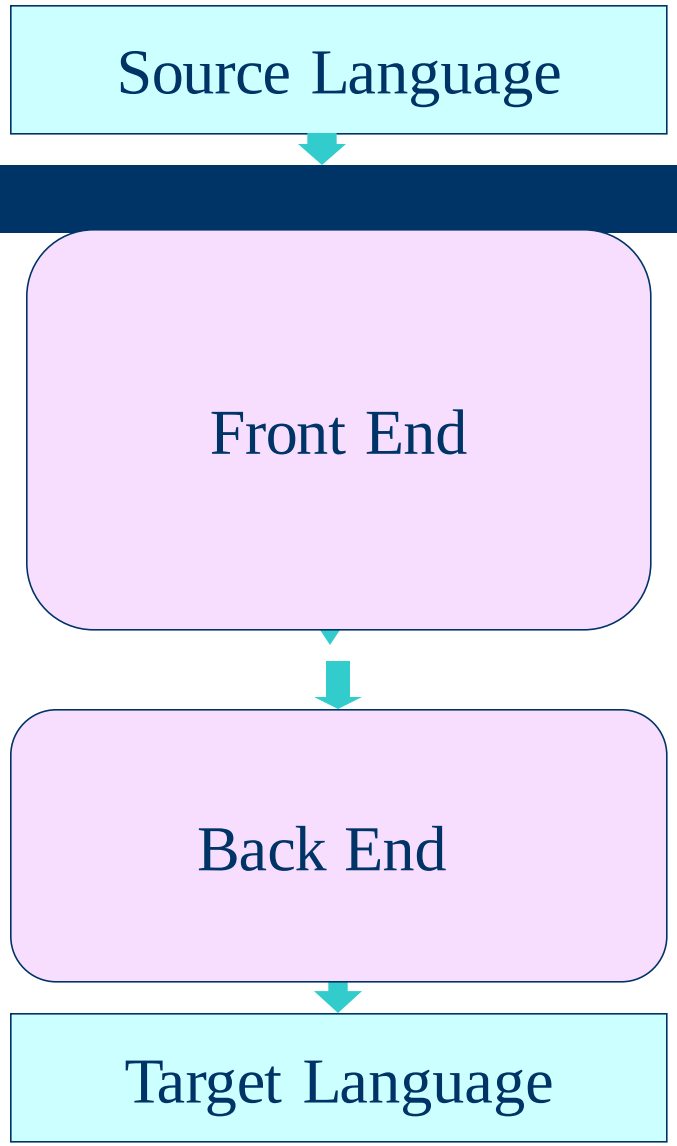
Modern Compilers(cont'd)

The **BACK-END**, which converts the internal representation into the target language, perhaps with optimizations. The target language used is typically an assembly language.

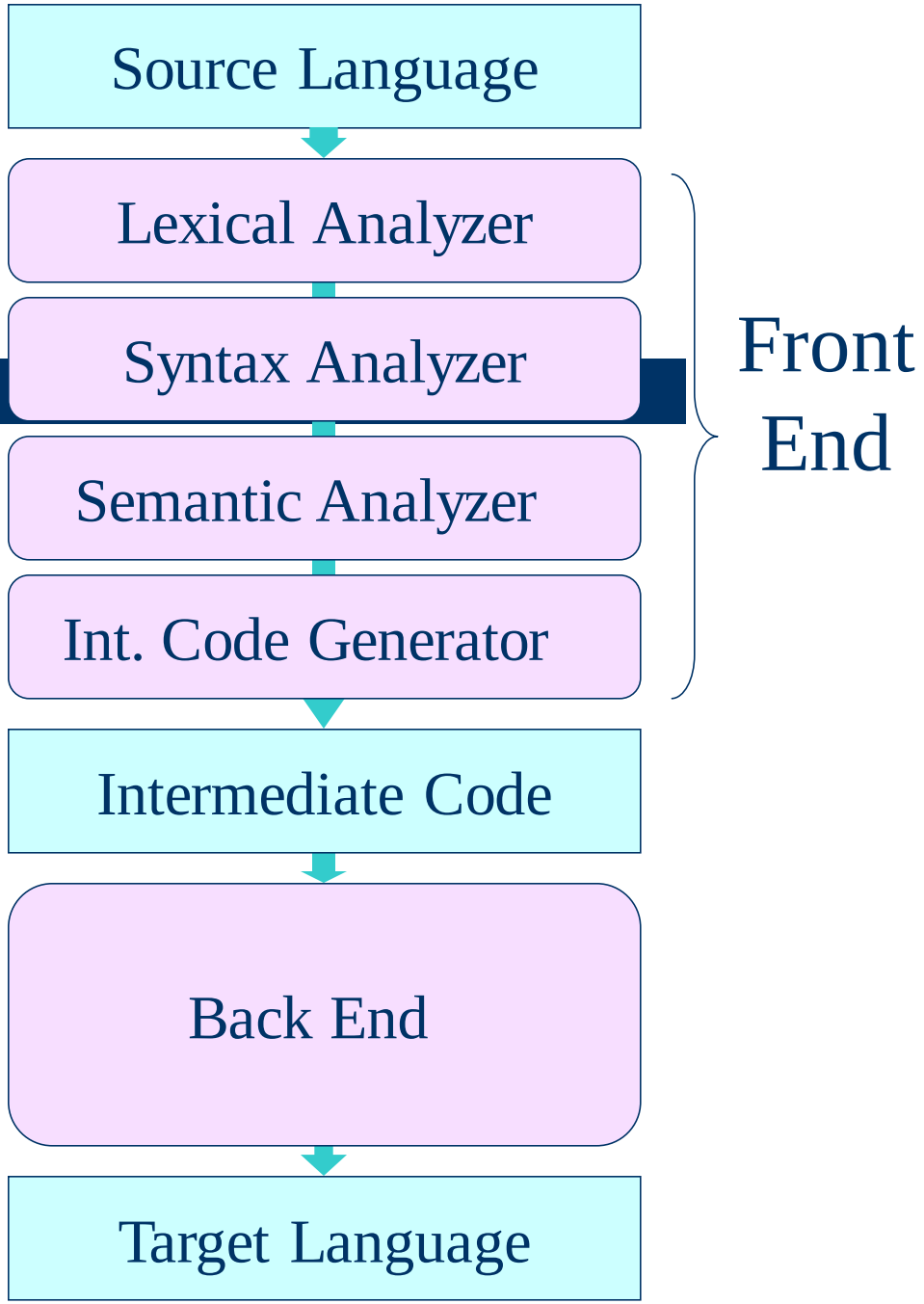
Structure of a Compiler



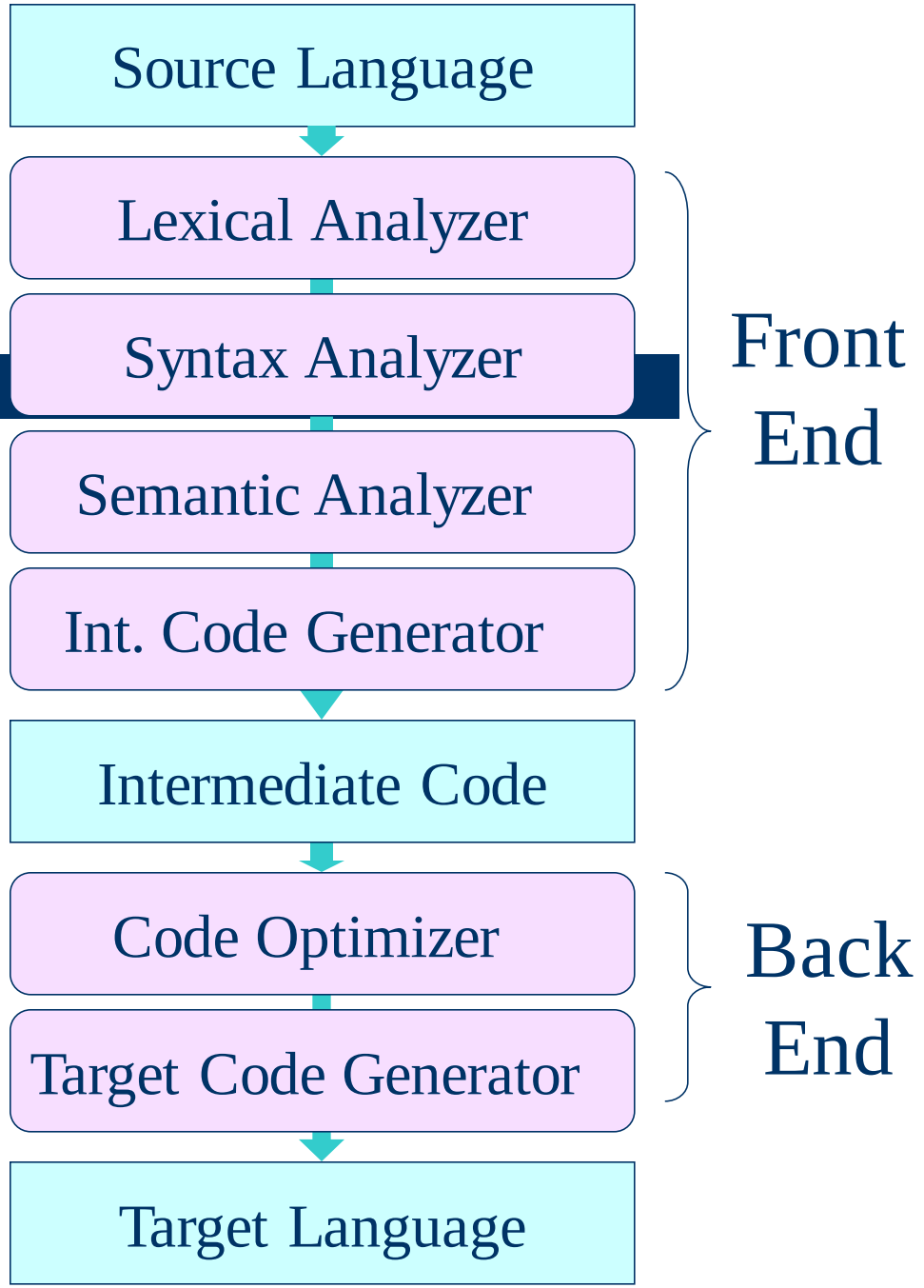
Structure of a Compiler



Structure of a Compiler



Structure of a Compiler



THANKS