

# COMPILER DESIGN

---

Course description

Project description

Introduction to compilation

---

## Course description

## Course description

- **Instructor's name:** Joey Paquet
- **Position:** Associate Professor, Department of Computer Science and Software Engineering
- **Teaching topics:** Programming languages, Compiler design, Software engineering and programming methodology
- **Research topics:** Design and implementation of programming languages, Parallel and/or distributed computing, Demand-driven computation models, Context-driven computation models
- **Contact information:**
  - Web: [www.cse.concordia.ca/~paquet](http://www.cse.concordia.ca/~paquet)
  - E-mail: [paquet@cse.concordia.ca](mailto:paquet@cse.concordia.ca)
  - Office: EV 3-221

## Calendar description

- COMP 442/6421 - Compiler Design (4 credits)
- Prerequisites (COMP442): COMP 228 or SOEN 228 or COEN 311; COMP 335; COMP 352 or COEN 352  
Prerequisites (COMP6421): COMP 5201, 5361, 5511.
- Compiler organization and implementation. Programming language constructs, their syntax and semantics. Syntax directed translation, code optimization. Run-time organization of programming languages. Project. Lectures: three hours per week. Laboratory: two hours per week.

## Course description

- **Topic**
  - Compiler organization and implementation.
  - Lexical, syntactic and semantic analysis, code generation.
- **Outline**
  - Design and implementation of a simple compiler.
  - Lectures related to the project.
- **Grading**
  - Assignments (4) : 40%
  - Final Examination : 30%
  - Final Project : 30%
- **Letter grading scheme**
  - Only one grading scheme for undergraduate and graduate.
  - Letter grading scheme is based on a normal curve based on the class average.

Undergraduate	Graduate
A+	A+
A	A
A-	A-
B+	B+
B	B
B-	B-
C+	C
C	C
C-	C
D+	C
D	C
D-	C
F	F

## Project description

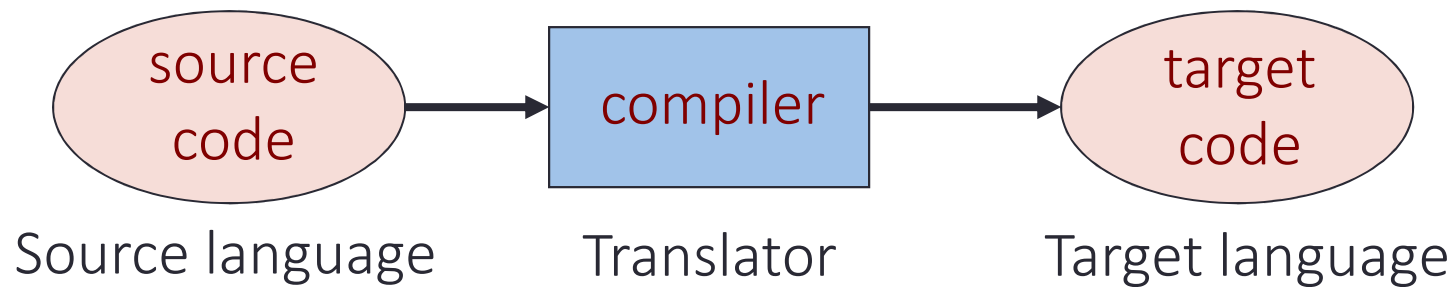
- Design and coding of a simple compiler
  - Individual work
  - Divided in four assignments
  - Final project is graded at the end of the semester, during a final demonstration
  - Testing is VERY important
- A complete compiler is a fairly complex and large program: from 10,000 to 1,000,000 lines of code.
- Programming one will force you to go over your limits.
- It uses many elements of the theoretical foundations of Computer Science.
- It will probably be the most complex program you have ever written.

---

# Introduction to compilation

## Introduction to compilation

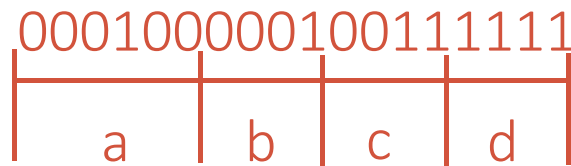
- A compiler is a translation system.
- It translates programs written in a high level language into a lower level language, generally machine (binary) language.
- The initial source code is essentially a stream of characters.
- The task of the compiler is to figure out its meaning and translate this meaning into a program that is understandable by the computer, preserving the same meaning.





## Introduction to compilation

- The only language that the processor understands is *binary*.
- A binary program is segmented into words.
- Every binary instruction word is segmented in portions that each have a different meaning.



a: Operation: Register addition (from a symbol table or op-code table)

b: First operand: register R1

c: Second operand: register R3

d: Destination: register R15

## Introduction to compilation

- Assembly language was the first higher level programming language.

000100000100111111  $\Leftrightarrow$  Add R1,R3,R15

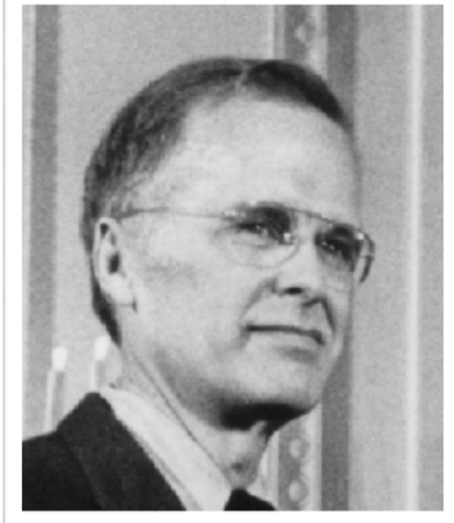
- There is a one-to-one correspondence between lines of assembly code and the machine code lines.
- A *op-code table* is sufficient to translate assembly language into machine code.
- Compared to binary, it greatly improved the **productivity** of programmers. Why?
- Though a great improvement, assembly is not ideal:
  - Not easy to write
  - Even less easy to read and understand
  - Extremely architecture-dependent

## Introduction to compilation

- A compiler translates a given high-level language into assembler or machine code.

<code>X=Y+Z;</code>	assign to X the value of the result of adding Z to Y
<code>L 3,Y</code> <code>A 3,Z</code> <code>ST 3,X</code>	Load working register with Y Add Z to working register Store the result in X
<code>00001001001011</code> <code>00010010010101</code> <code>00100100101001</code>	Load working register with Y Add Z to working register Store the result in X

## First compilers: Fortran

- The problems with assembly led to the development of the first compilers for higher-level programming languages. Notable among these is the compiler for the Fortran language.
  - IBM Mathematical **Formula Translating System**, later popularly know as Fortran.
  - Originally developed by a team lead by **John Backus** at **IBM** in the 1950s for scientific and engineering applications on the IBM704, introduced in 1954.
- 
- A black and white portrait of John Backus, a man with glasses, wearing a suit and tie, looking slightly to the right.
- John Backus
- General-purpose, **procedural, imperative** programming language that is especially suited to numeric computation and scientific computing.
  - Originally designed to improve on the **economics of programming**, as programming using low level languages had become to be more costly than the time it actually saved.
  - This was an incredible feat, as the theory of compilation was not available at the time.

## First compilers: Fortran

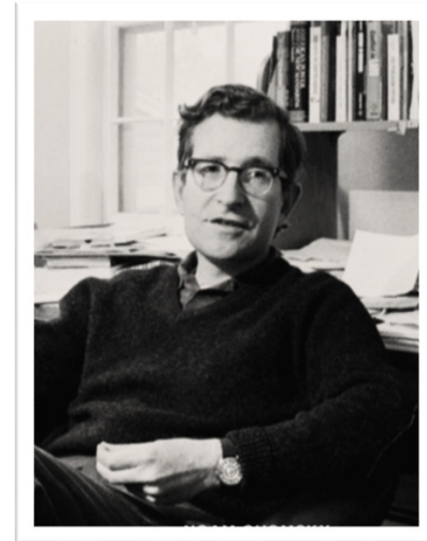
- The solution proposed by Fortran was to design a **higher level** programming language enabling scientists to write programs using a **mathematical notation/language**.
- Great emphasis was put on the **efficiency** of the translated machine code, which is still the case today, explaining why Fortran programs are still considered as a **benchmark** for execution speed.
- Emphasis was put on **number computations**. Only much later was it possible to have Fortran programs manipulate characters or strings, and eventually objects.

## First compilers: Fortran

- **Features** introduced by earlier versions of Fortran:
  - Comments
  - Assignment statement using complex expressions
  - Control structures (conditional, loop)
  - Subroutines and functions used similarly to the mathematical notion of function
  - Formatting of input/output
  - Machine-independent code
  - Procedural programming
  - Arrays
  - Early development of compilers
  - Showed the importance/possibility/relevance of higher-level programming languages
  - Compiler development tools/techniques
  - Optimizing compiler

## Paving down the road

- In parallel to that, Noam Chomsky, a **linguist**, was investigating on the structure of natural languages.
- His studies led the way to the classification of languages according to their complexity (aka the *Chomsky hierarchy*) using the notion of *generative grammar*.
- This was used by various theoreticians in the 1960s and early 1970s to design a fairly complete set of solutions to the parsing problem.
- These solutions have been used ever since.
- As parsing solutions became well understood and formally defined, efforts were devoted to the development of *parser generators*.
- One of the earliest and still most commonly used parser generators is **YACC** (*Yet Another Compiler Compiler*).
- Developed by Stephen C. Johnson in 1975 at Bell Labs.
- Nowadays, many tools exist that can automate the generation of compiler parts based on the theoretical foundations elucidated in the late 1960s and early 1970s.



Noam Chomsky



Stephen C. Johnson

---

## Compilation vs. interpretation

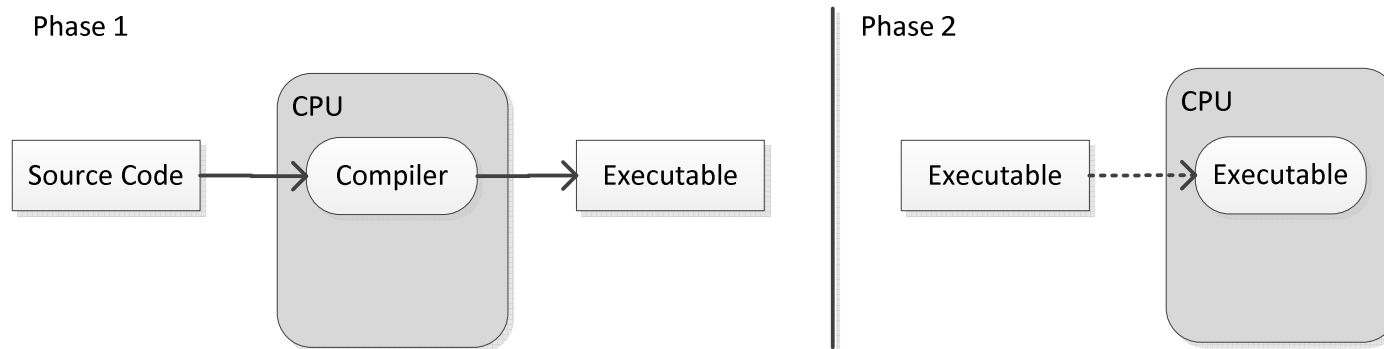


## Compilation vs. interpretation

- Programming languages can be implemented (i.e. translated/executed) by any of the following three general methods:
  - **Compilation:** programs are translated directly into machine language by a compiler, the output of which is later loaded and executed directly by a computer.
  - **Interpretation:** programs are interpreted (i.e. simultaneously translated and executed) by an interpreter.
  - **Compilation/Interpretation Hybrid:** programs are first translated into an intermediate representation by a compiler, the output of which is later executed by and interpreter.
- No matter what model is used, the same language translation steps are applied, except at different times during the translation or execution.
- Languages that delay much of the checking to run-time are called “dynamic languages”.

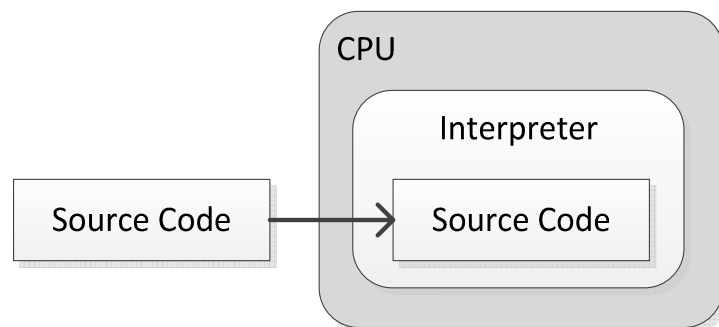
## Compilation

- A **compiler** is a computer program (or set of programs) that transforms source code written in a computer language (the source language) into another computer language (the target language, often having a binary form known as object code).
- The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code) **before execution time**.



## Interpretation

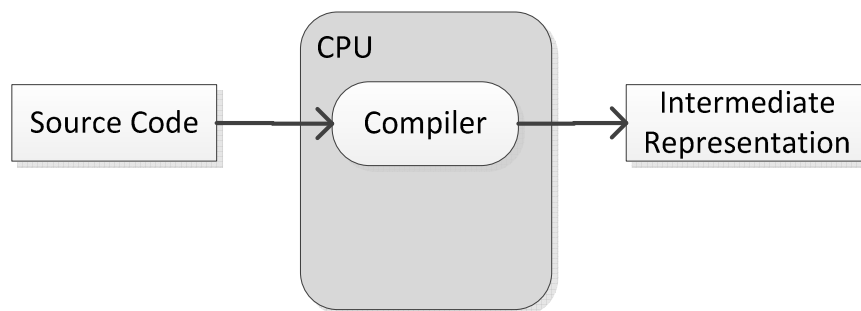
- An interpreted language is a programming language whose programs are not directly executed by the host CPU but rather executed (or said to be *interpreted*) by a software program known as an **interpreter**.
- Initially, interpreted languages were compiled line-by-line; that is, each line was compiled as it was about to be executed, and if a loop or subroutine caused certain lines to be executed multiple times, they would be retranslated every time. This has become much less common.
- In this case, all the program analysis phases must be happening at run-time, which can lead to unnecessary overhead.



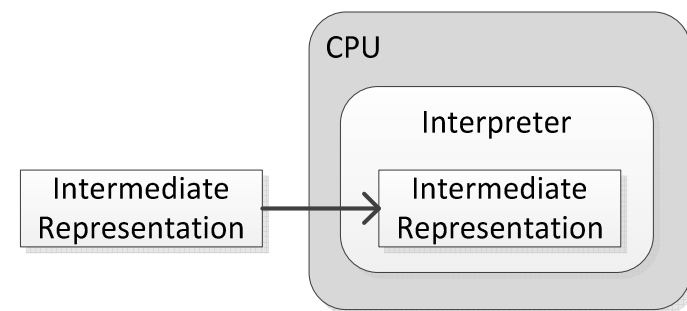
## Hybrid compilation/interpretation

- Nowadays, most interpreted languages use an **intermediate representation**, as a bridge between compilation and interpretation.
- In this case, a compiler may output some form of bytecode or threaded code, which is then executed by an interpreter. Examples include Python, Java, Perl and Ruby.
- The intermediate representation can be compiled once and for all (as in Java), each time before execution (as in Perl or Ruby), or each time a change in the source is detected before execution (as in Python).

Phase 1



Phase 2



## Hybrid compilation/interpretation

- The source code of the program is often translated to a form that is more **convenient** to interpret, which may be some form of machine language for a virtual machine (such as Java's bytecode).
- An important and **time-consuming** part of the source code analysis is done **before** the interpretation starts.
- Some program **errors** are detected **before** execution starts, thus making the execution more **robust**.

## Which one is better?

- In the early days of computing, language design was heavily influenced by the decision to use compilation or interpretation as a mode of execution.
- For example, some compiled languages require that programs must **explicitly state the data type** of a variable at the time it is declared or first used while some interpreted languages take advantage of the **dynamic** aspects of interpretation to make such declarations **unnecessary**.
- Such language issues delay the **type binding mechanism to run-time**. This does not mean that other program analysis phases must also be delayed to run-time.
- Theoretically, any language may be compiled or interpreted, so this designation is applied purely because of common implementation practice and not some underlying property of a language.
- Many languages have been implemented using both compilers and interpreters, including Lisp, Pascal, C, Basic, and Python.

## Which one is better?

- Interpreting a language gives implementations some additional **flexibility** over compiled implementations. Features that are often easier to implement in interpreters than in compilers include (but are not limited to):
  - platform independence (Java's byte code, for example)
  - reflection and reflective usage of the evaluator
  - dynamic typing and polymorphism
- The main disadvantage of interpreting is a **slower speed** of program execution compared to direct machine code execution on the host CPU. A technique used to improve performance is **just-in-time** compilation which converts frequently executed sequences of interpreted instruction to host machine code.
- Another disadvantage is that, as some part of the analysis/translation is done at runtime, it results in programs that are more likely to fail at runtime, i.e. the resulting programs are less **robust**. This can be somehow alleviated by proper testing prior to development.

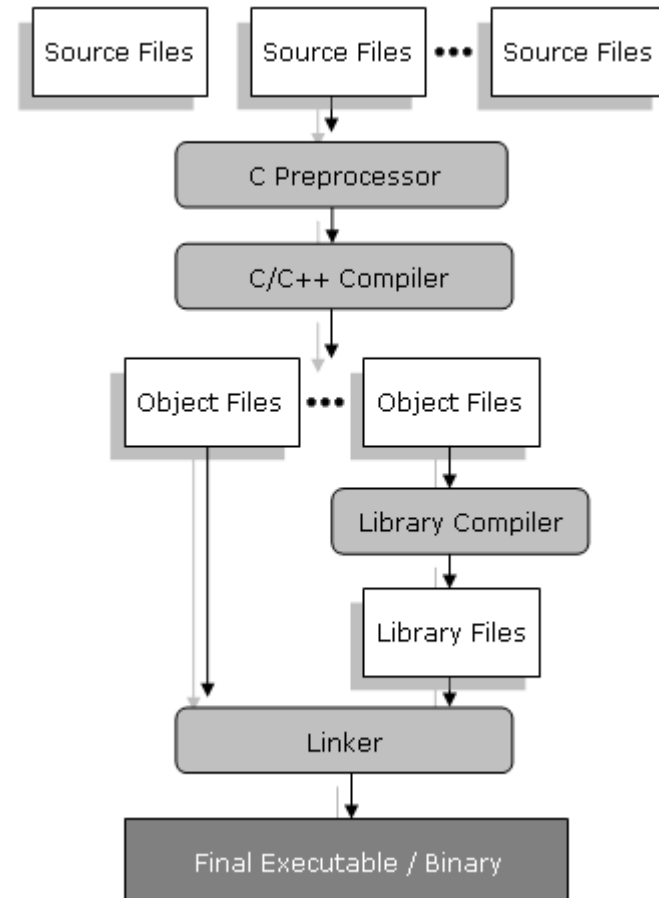
---

## Compilation process

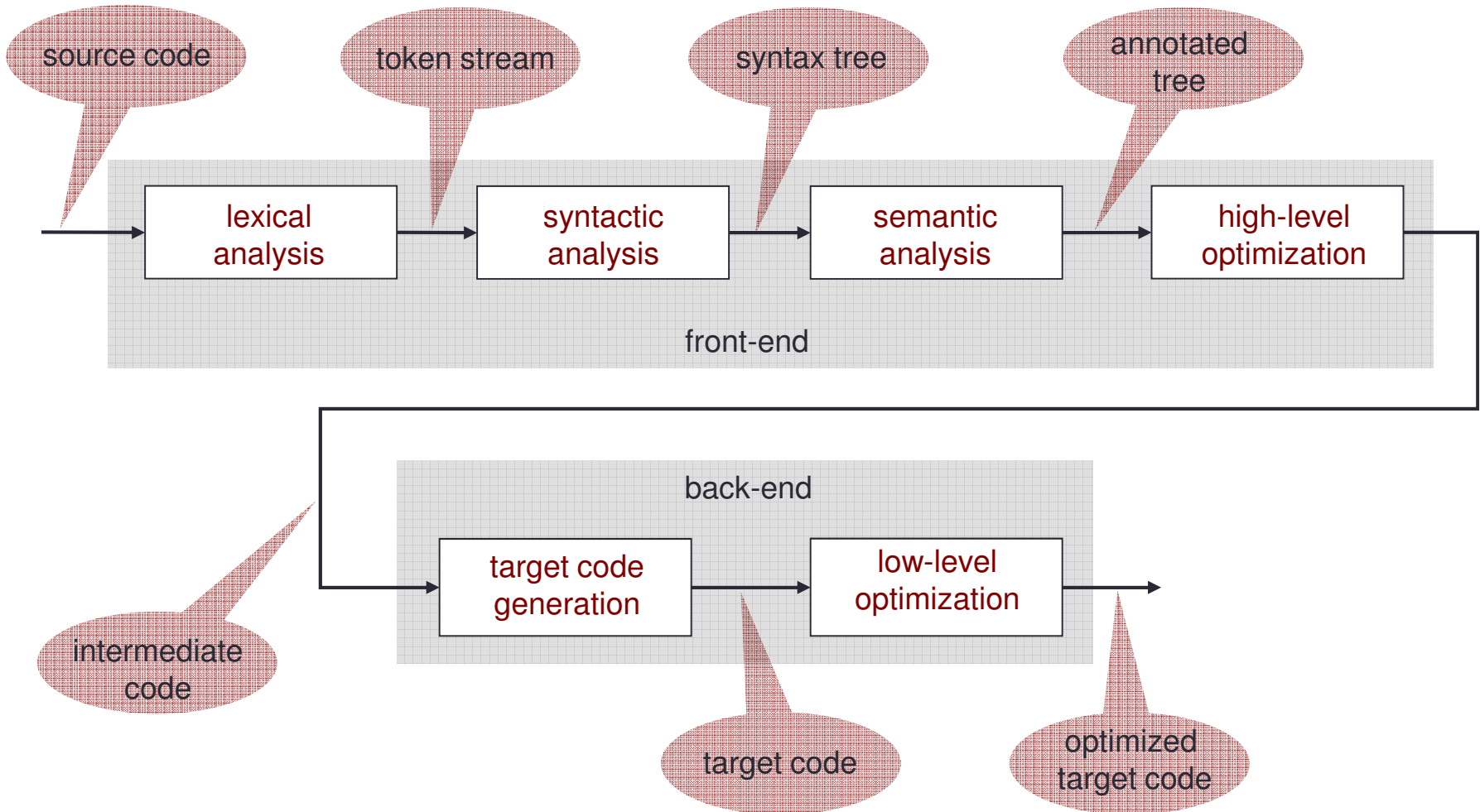


## Compiler's environment

- The compiler is most often one part of a translation/linkage system.
- For example in C++, the translation process involves different tools:
  - **Preprocessor:** translate directives that are generally about file inclusion or processing of macros.
  - **Template metaprocessor:** process and translate template declarations into C++ code which is then fed to the compiler.
  - **Library compiler:** process object files and generate dynamically linked libraries (dll).
  - **Linker:** take the resulting object files, resolve their cross-references and generate a unified executable.



# Phases of a compiler



---

Front end and Back end

## Front end

- The front end analyzes the source code to build an internal representation of the program, called the **intermediate representation**.
- It also manages the **symbol table**, a data structure mapping each symbol in the source code to associated information such as type and scope.
- The front-end is composed of: Lexical analysis, Syntactic analysis, Semantic analysis and High-level optimization.
- In most compilers, most of the front-end is **driven by the Syntactic analyzer**.
- It calls the Lexical analyzer for tokens and generates an *abstract syntax tree* when syntactic elements are recognized.
- The generated tree (or other intermediate representation) is then analyzed and optimized in a separate process.
- Heavily dependent on the source language compiled.
- Independent from the target machine on which the code is eventually executed.

## Back end

- The term back end is sometimes confused with code generator because of the included functionality of generating assembly/machine code. Some literature uses middle end to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generator.
- The back-end is composed of: Code generation and Low-level optimization.
- Uses the **intermediate representation** generated by the front-end to generate target machine code, which can then potentially be optimized.
- Heavily dependent on the target machine.
- Independent on the source programming language compiled.

---

## Compilation process phases

## Lexical analysis

- Lexical analysis is the process of converting a sequence of **characters** into a sequence of **tokens**.
- A program or function which performs lexical analysis is called a *lexical analyzer*, *lexer* or *scanner*.
- A scanner often exists as a single function which is called by the parser, whose functionality is to extract the next token from the source code.
- The lexical specification of a programming language is defined by a set of rules which defines the scanner, which are understood by a lexical analyzer generator such as *lex* or *flex*. These are most often expressed as **regular expressions**.
- The lexical analyzer (either generated automatically by a tool like *lex*, or hand-crafted) reads the source code as a stream of characters, identifies the lexemes in the stream, categorizes them into tokens, and outputs a token stream.
- This is called "tokenizing."
- If the scanner finds an invalid token, it will report a lexical error.

## Lexical analysis

tokens	lexeme examples
keyword	while, to, do, int, main
identifier	i, max, total, i1, i2
literal	123, 12.34, "Hello"
operator	=, +, *, and, >, <
punctuation	{, }, [, ], ;

```
Distance = rate * time;
```

```
[id, distance], [assignop, =], [id, rate], [multop, *], [id, time], [semi, ;]
```



## Syntactical analysis

- Syntax analysis involves **parsing** the token sequence to identify the syntactic structure of the program.
- The parser's output is some form of **intermediate representation** of the program's structure, typically a **parse tree**, which replaces the linear sequence of tokens with a tree structure built according to the rules of a **formal grammar** which is used to define the language's syntax.
- This is usually done using a **context-free grammar** which recursively defines components that can make up an valid program and the order in which they must appear.
- The resulting parse tree is then analyzed, augmented, and transformed by later phases in the compiler.
- Parsers are written by hand or generated by parser generators, such as *Yacc*, *Bison*, *ANTLR* or *JavaCC*, among other tools.

# Syntactical analysis

input program

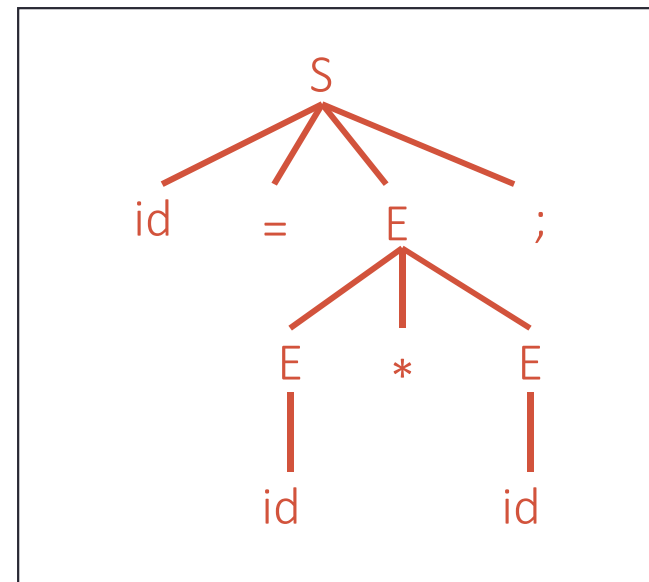
```
Distance = rate * time;
```

grammar

```
S ::= id = E;
```

```
E ::= E * E
```

```
E ::= id
```



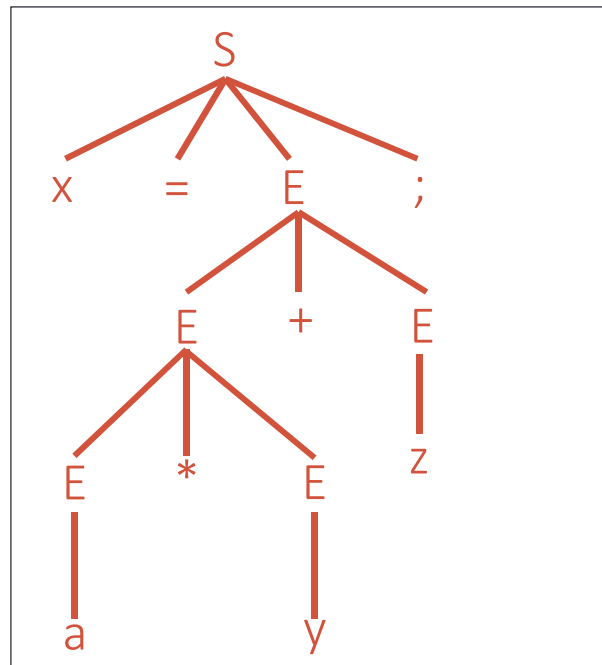
## Semantic analysis

- Semantic analysis is the phase in which the compiler adds **semantic information** to the parse tree and builds the **symbol table**.
- The symbol table is a dictionary containing information about all the identifiers defined in a program.
- This phase performs semantic checks such as **type checking** (checking for type errors), or **static binding** (associating variable and function references with their definitions/types), or **definite assignment** (requiring all local variables to be initialized before use), rejecting semantically incorrect programs or issuing warnings.
- This phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation, which is known as **syntax-directed translation**.
- Not all rules defining programming languages can be expressed by context-free grammars alone, for example semantic validity such as type validity and proper declaration of identifiers. These rules can be formally expressed with **attribute grammars** that implement **attribute migration** across syntax tree nodes when necessary.

## Semantic analysis

- Once semantic analysis is done, **semantic translation** may traverse the parse tree and generate an intermediate code representation.

`x = a*y+z;`



```
t1 = a;  
t2 = y;  
t3 = t1*t2;  
t4 = z;  
t5 = t3+t4;  
x = t5;
```

## High-level optimization

- High-level optimization involves an analysis phase for the gathering of program information from the intermediate representation derived by the front end.
- Typical analyzes are **data flow analysis** to build use-define chains, **dependence analysis**, **alias analysis**, **pointer analysis**, etc.
- Accurate analysis is the basis for any compiler optimization.
- The **call graph** and **control flow** graph are usually also built during the analysis phase.
- After analysis, the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms.
- Popular optimizations are **inline expansion**, **dead code elimination**, **constant propagation**, **loop transformation**, **register allocation** or even **automatic parallelization**.
- At this level, all activities are target machine independent.

## High-level optimization

- Simplistic example of high-level optimization applied on intermediate code

```
t1 = a;  
t2 = y;  
t3 = t1*t2;  
t4 = z;  
t5 = t3+t4;  
x  = t5;
```



```
t1 = a*y;  
x  = t1+z;
```

## Code generation

- The transformed intermediate language is translated into the output language, usually the native machine language of the system.
- This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes.

```
t1 = a*y;  
x  = t1+z;
```



```
LE 4,a      put a in register 4  
ME 4,y      multiply by y  
AE 4,z      add z  
STE 4,x     store register 4 in x
```

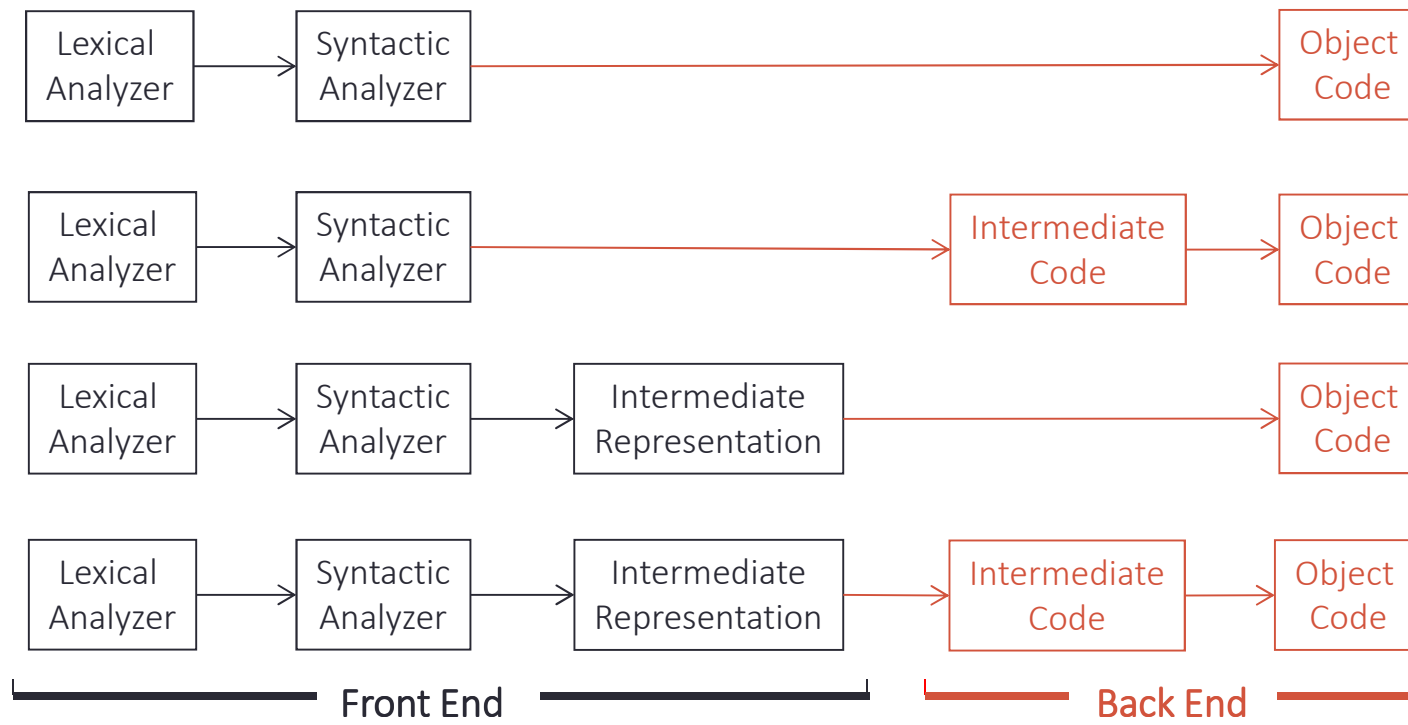
## Low-level optimization

- The generated target code is analyzed for inefficiencies in the generated code, such as dead code or code redundancy.
- Care is taken to exploit as much as possible the CPU's capabilities and the target machine's hardware architecture.
- This phase is heavily architecture-dependent.



## Intermediate representations

- There are different approaches when considering the intermediate representations to be used.
- An abstract intermediate representation (e.g. a tree) can be used in the front end to enable high-level representation/optimization.
- A more concrete intermediate code representation may be used to facilitate code generation and lower level optimizations.

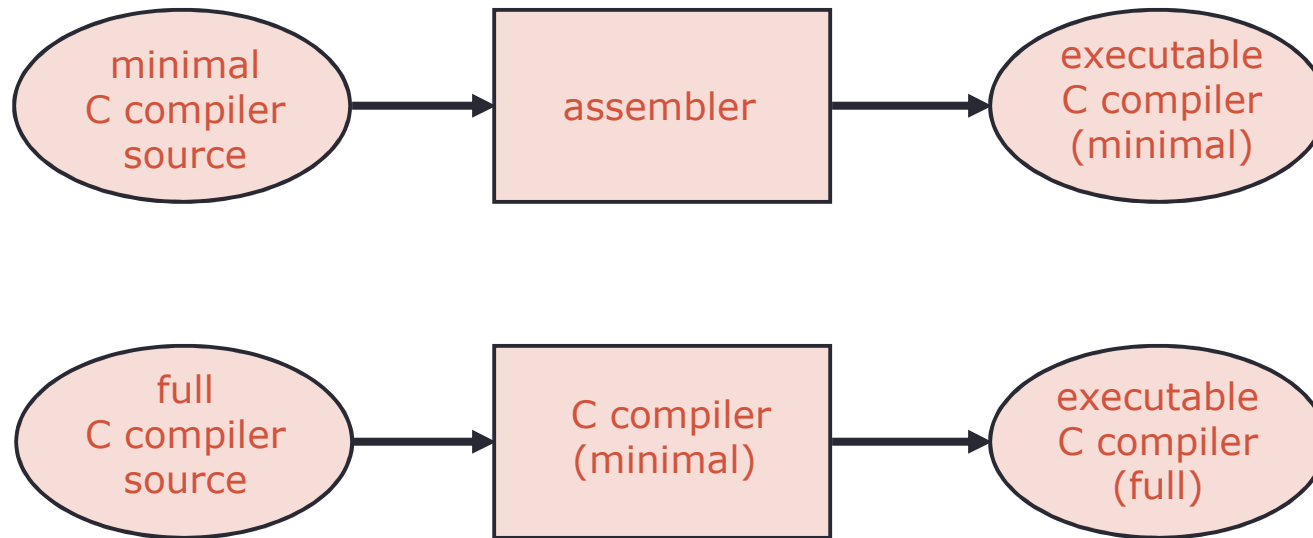


## Compilation/execution system support

- Symbol table
  - Central repository of identifiers (variable or function names) used in the compiled program.
  - Contains information such as the data type, function signature, or value in the case of constants.
  - Used to identify undeclared or multiply declared identifiers, as well as type mismatches or invalid function calls, or use of invalid members for data structures or objects.
  - Provides temporary variables for intermediate code generation.
- Run-time system
  - Some programming languages concepts raise the need for dynamic memory allocation. **What are they?**
  - The running program must be able to manage its own memory use.
  - Some will require a stack, others a heap. These are managed by the run-time system.
  - Other language features may require to dynamically associate certain identifiers with a specific implementation at run-time. **What are they?**
  - Such features require elaborated dynamic linking systems available at run-time.

## Writing of early compilers

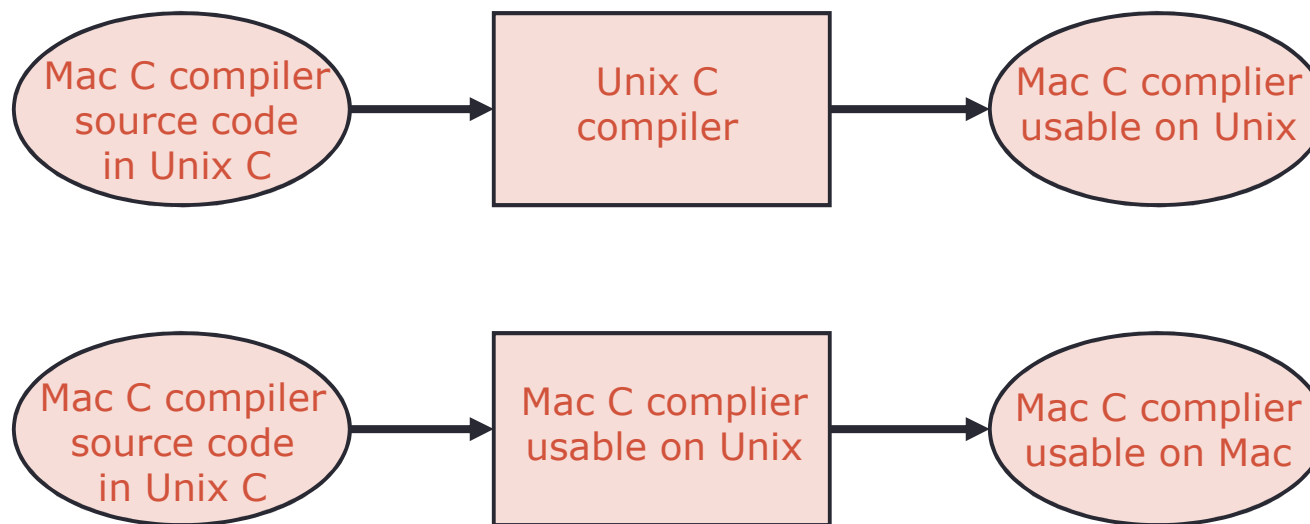
- The first C compiler



- It is much easier to write a compiler if a high-level language is available.
- Here if we do one more iteration, we can have a C compiler written in C.
- Or we may then use this compiler to build compilers for other languages.

## Cross-compilation

- Using the same concept, we can also create compilers for other platforms:



## Retargetable compilers

- Two methods:
  - Make a strict distinction between front-end and back-end, then use different back-ends.
  - Generate code for a virtual machine, then build a compiler or interpreter to translate virtual machine code to a specific machine code. That is what we do in the project.

## Summary

- The first compiler was the assembler, a one-to-one direct translator.
- Complex compilers were written incrementally, first using assemblers.
- All compilation techniques are well known since the 60's and early 70's.
- The compilation process is divided into phases.
- The input of a phase is the output of the previous phase.
- It can be seen as a pipeline, where the phases are filters that successively transform the input program into an executable.

## References

- Backus, J. W.; H. Stern, I. Ziller, R. A. Hughes, R. Nutt, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan (1957). *The FORTRAN Automatic Coding System*. Western joint computer conference: Techniques for reliability (Los Angeles, California: Institute of Radio Engineers, American Institute of Electrical Engineers, ACM): 188–198.  
doi:10.1145/1455567.1455599.
- Naomi Hamilton (Computerworld), [\*The A-Z of Programming Languages: YACC - The contribution YACC has made to the spread of Unix and C is a sense of pride for Stephen C. Johnson\*](#). July 2008.
- C.N. Fischer, R.K. Cytron, R.J. LeBlanc Jr., *Crafting a Compiler*, Addison-Wesley, 2009. Chapter 1, 2.