

CS:3820

Programming Language Concepts

Spring 2019

Introduction and Overview

Parts of these notes were originally developed by Cesare Tinelli, Allen Tucker, Robert Noonan and Peter Sestoft and are used with permission. They are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

1 Principles

Distinguishing programming languages properties:

- Syntax
- Names
- Types
- Abstractions
- Semantics

For any language:

- Its designers must define these properties
- Its programmers must master these properties

Syntax

The *syntax* of a programming language is a precise description of all its grammatically correct programs

When studying syntax, we ask questions like:

- What is the grammar for the language?
- What is the basic vocabulary?
- How are syntax errors detected?

Names

Various kinds of entities in a program have names:

variables, types, functions, parameters, classes, objects, ...

Named entities are bound in a running program to:

- Scope
- Visibility
- Type
- Lifetime

Types

A *type* is a collection of values and of operations on those values

- Simple types
 - numbers, characters, Booleans, ...
- Structured types
 - Strings, lists, trees, hash tables, ...
- A language's *type system* can help:
 - determine legal operations
 - detect type errors
 - optimize certain operations

Abstractions

Mechanisms for generalizing computations or data:

- Procedures/functions
- Modules
- Abstract data types
- Classes
- Memory models

Semantics

The meaning of a program is called its *semantics*

In studying semantics, we ask questions like:

- When a program is running, what happens to the values of the variables?
- What does each construct do?
- What underlying model governs run-time behavior, such as function call?
- How are variables and objects allocated to memory at run-time?

2 Paradigms

A programming *paradigm* is a pattern of problem-solving thought that underlies a particular genre of programs and languages

There are several main programming paradigms:

- Imperative
- Object-oriented
- Functional
- Logic
- Dataflow



focus of this course

Imperative Paradigm

Follows the classic von Neumann-Eckert model:

- Program and data are indistinguishable in memory
- Program = sequence of commands modifying current *state*
- State = values of all variables when program runs
- Large programs use procedural abstraction

Example imperative languages:

- Cobol, Fortran, C, Ada, Perl, ...

Object-oriented (OO) Paradigm

An OO Program is a collection of objects that interact by passing messages that transform local state

Major features:

- Encapsulated State
- Message passing
- Inheritance
- Subtype Polymorphism

Example OO languages:

Smalltalk, Java, C++, C#, Python, ...

Functional Paradigm

Functional programming models a computation as a collection of mathematical functions

- Input = domain
- Output = range

Major features

- Functional composition
- Recursion
- Referential transparency

Example functional languages:

- Lisp, Scheme, ML, Haskell, F#,...

Functional Paradigm

Functional programming models a computation as a collection of mathematical functions

- Input = domain
- Output = range

Notable features of modern functional languages:

- Functions as values
- Symbolic data types
- Pattern matching
- Sophisticated type systems and module systems

Logic Paradigm

Logic programming declares what outcome of the program should be, rather than how it should be achieved

Major features:

- Programs as sets of constraints on a problem
- Computation of all possible solutions
- Nondeterministic computation

Example logic programming languages:

- Prolog, Datalog

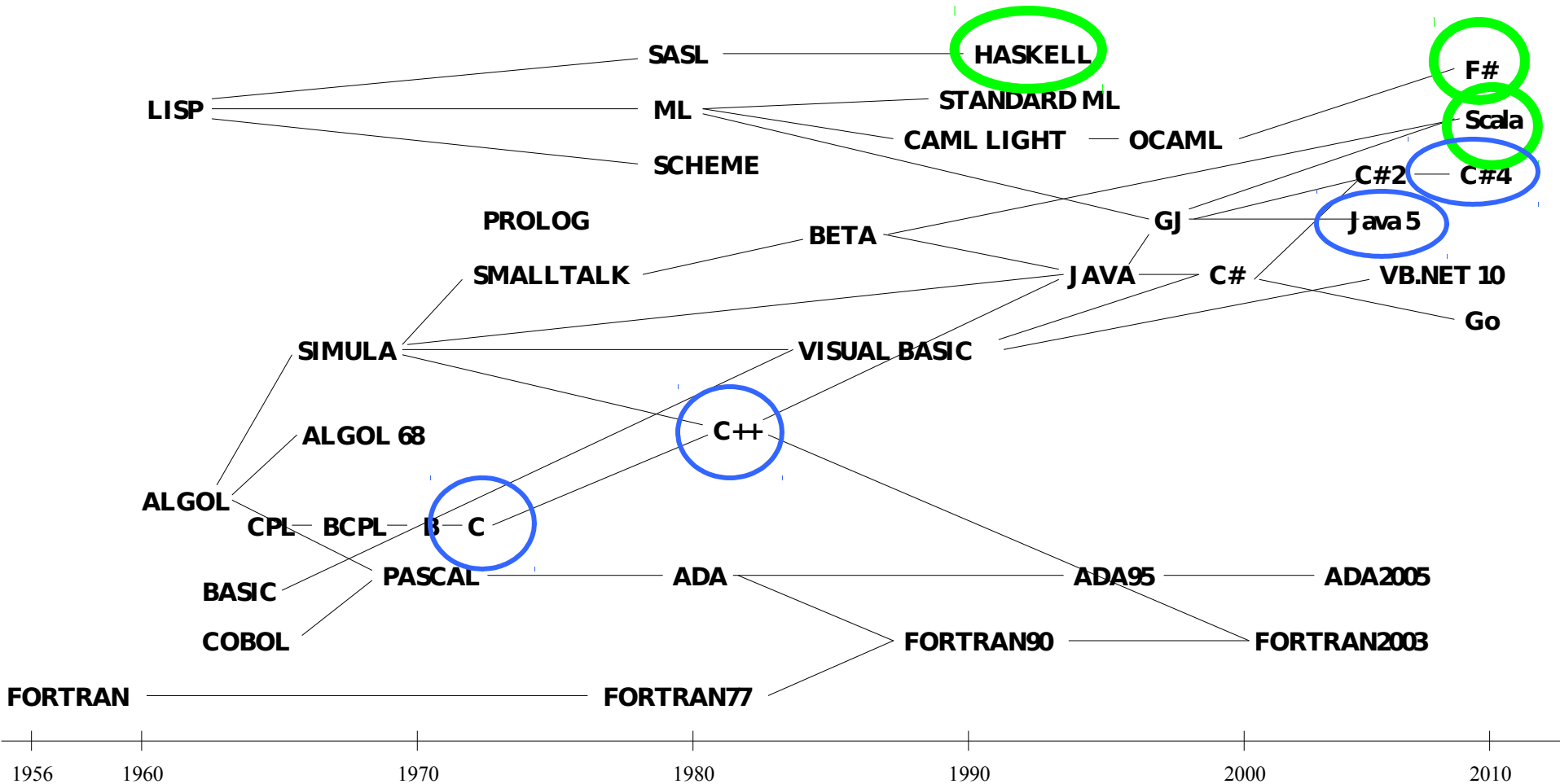
3 A Brief History

How and when did programming languages evolve?

What communities have developed and used them?

- Artificial Intelligence
- Computer Science Education
- Science and Engineering
- Information Systems
- Systems and Networks
- World Wide Web
- ...

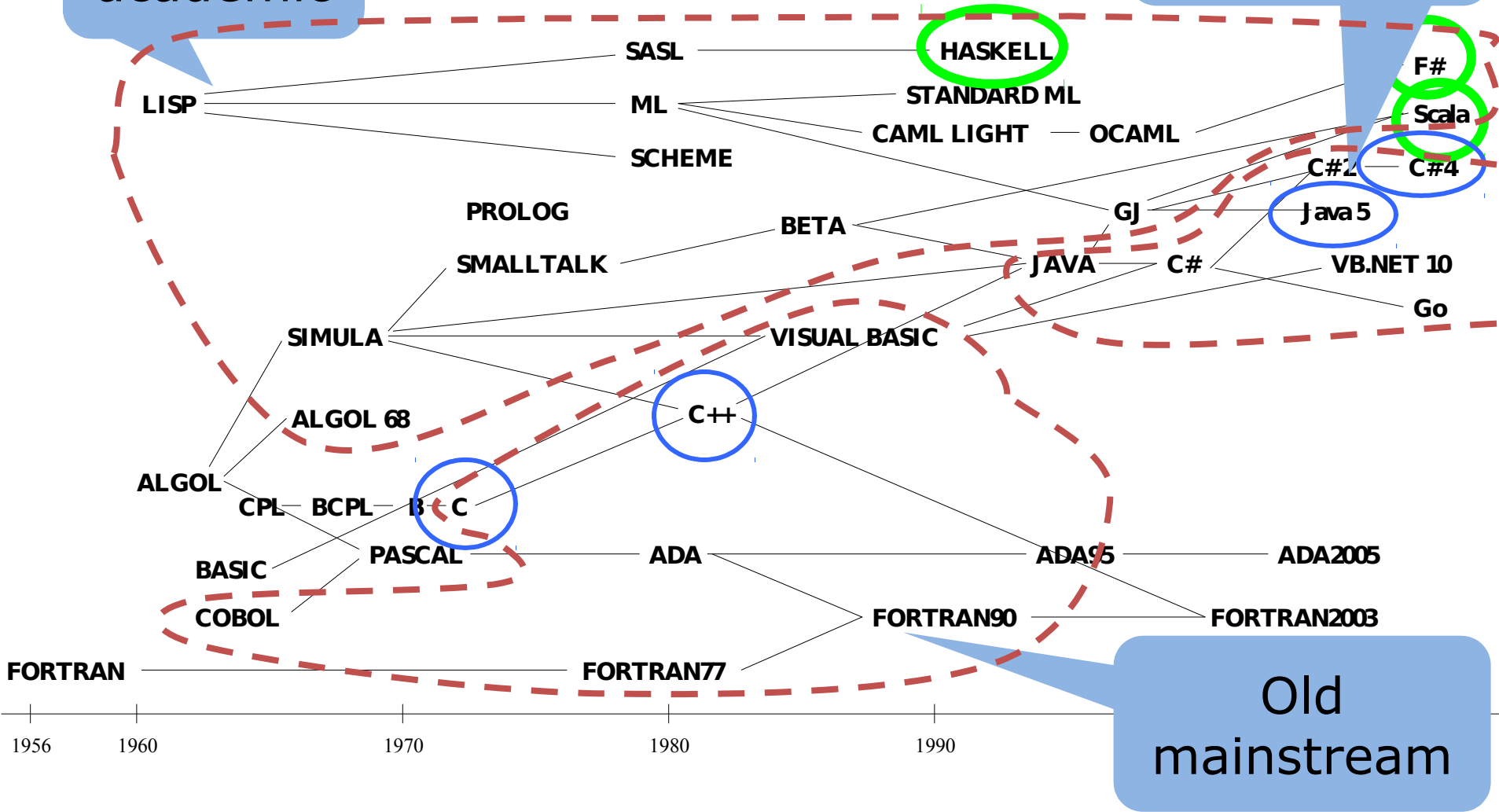
Language Genealogy



Language Genealogy

Mostly academic

Modern mainstream



4 On Language Design

Design Constraints

- Computer architecture
- Technical setting
- Standards
- Legacy systems

Design Outcomes and Goals

What makes a successful language?

Key characteristics:

- Simplicity and readability
- Reliability
- Support
- Abstraction
- Orthogonality
- Libraries
- Efficient implementation
- Community

Simplicity and Readability

- Small instruction set
 - E.g., Java vs Scheme
- Simple syntax
 - E.g., C/C++/Java vs Python
- Benefits:
 - Ease of learning
 - Ease of programming

Reliability

- Program behavior is the same on different platforms
 - E.g., early Fortran, C
- Type errors are detected
 - E.g., C vs Haskell
- Semantic errors are properly trapped
 - E.g., C vs C++
- Memory leaks are prevented
 - E.g., C vs Java

Language Support

- Accessible (public domain) compilers/interpreters
- Good texts and tutorials
- Wide community of users
- Integrated with development environments (IDEs)

Orthogonality

A language is *orthogonal* if its features are built upon a small, mutually independent set of primitive operations.

- Fewer exceptional rules = conceptual simplicity
 - E.g., restricting types of arguments to a function
- Tradeoffs with efficiency

Efficiency Issues

- Embedded systems
 - Real-time responsiveness (e.g., navigation)
 - Failures of early Ada implementations
- Web applications
 - Responsiveness to users (e.g., Google search)
- Corporate database applications
 - Efficient search and updating
- AI applications
 - Modeling human behaviors

5 Compilers and Interpreters

Compiler – produces machine code

Interpreter – executes instructions on a virtual machine

- Some compiled languages:
 - Fortran, C, C++, Rust, Swift
- Some interpreted languages:
 - Scheme, Python, Javascript
- Hybrid compilation/interpretation
 - Java Virtual Machine (JVM) languages (Java, Scala, Clojure)
 - .NET languages (C#, F#)

Compilation

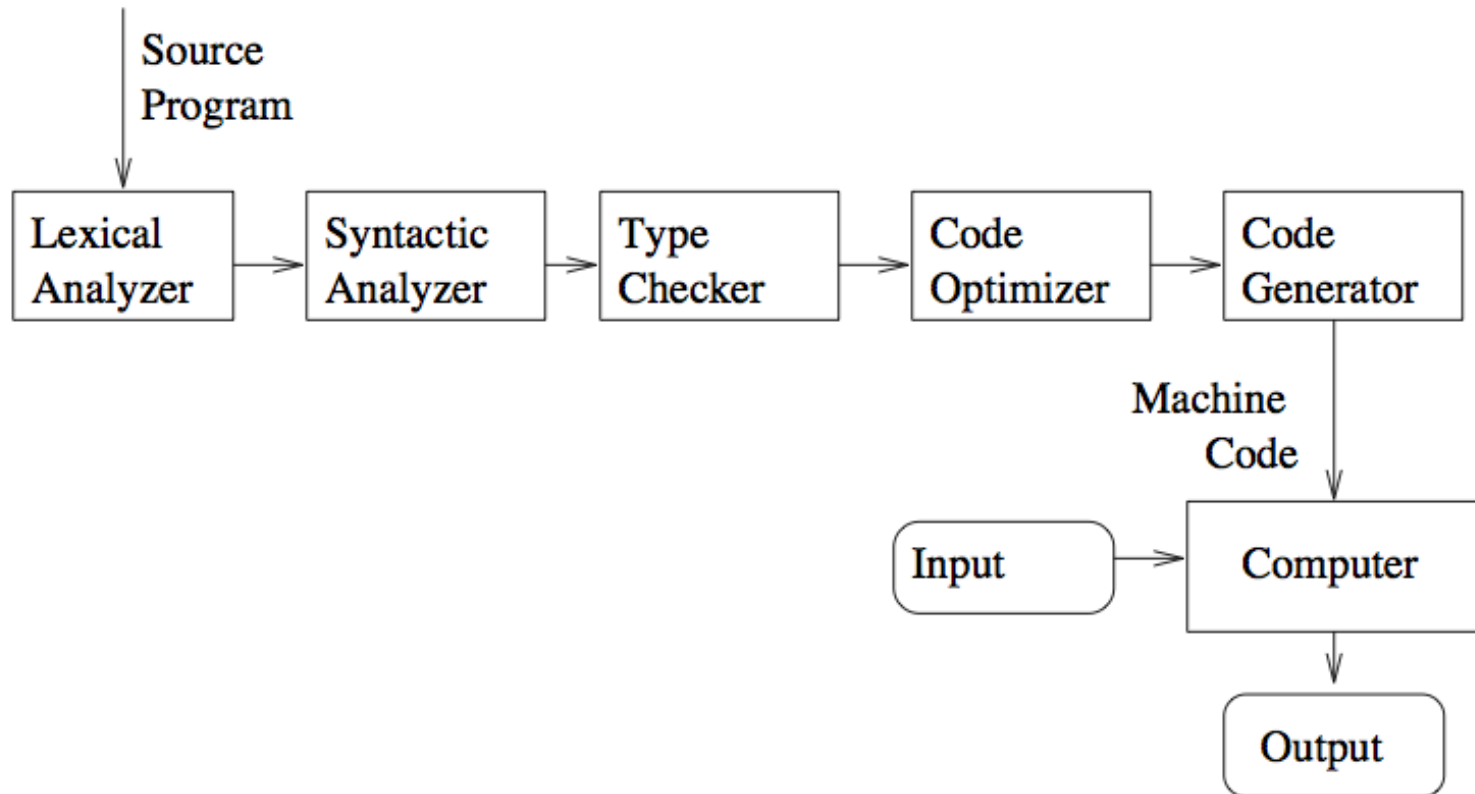


Figure 1.4: The Compile-and-Run Process

Interpretation

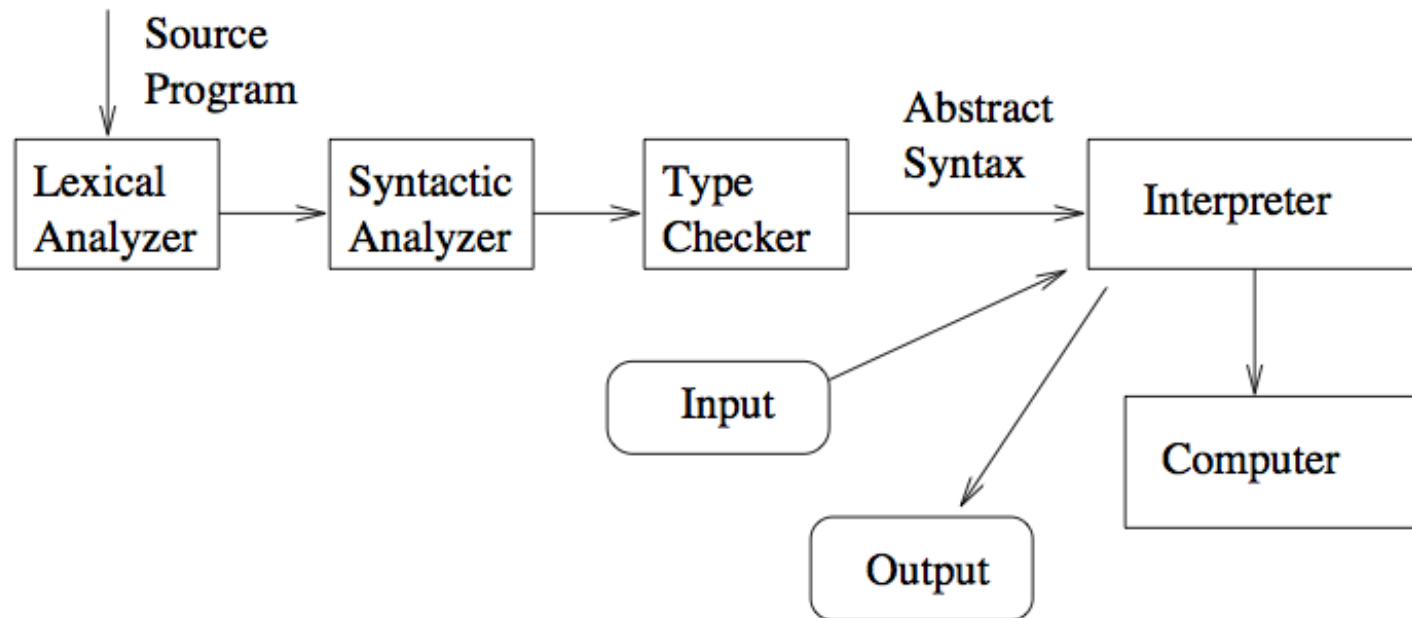


Figure 1.5: Virtual Machines and Interpreters

6 Course Contents

- A brief intro to functional programming with F#
- Lexical analysis, regular expressions, finite automata, lexer generators
- Syntax analysis, top-down versus bottom-up parsing, LL versus LR, parser generators
- Expression evaluation, stack machines, Postscript
- Compilation of a subset of C with *p, &x, pointer arithmetic, arrays
- Type checking, type inference, statically and dynamically typed languages
- The machine model of Java, C#, F#: stack, heap, garbage collection
- The intermediate bytecode languages of the Java Virtual Machine and .NET
- Garbage collection techniques, dynamic memory management
- Continuations, exceptions, a language with backtracking
- Selected advanced topics