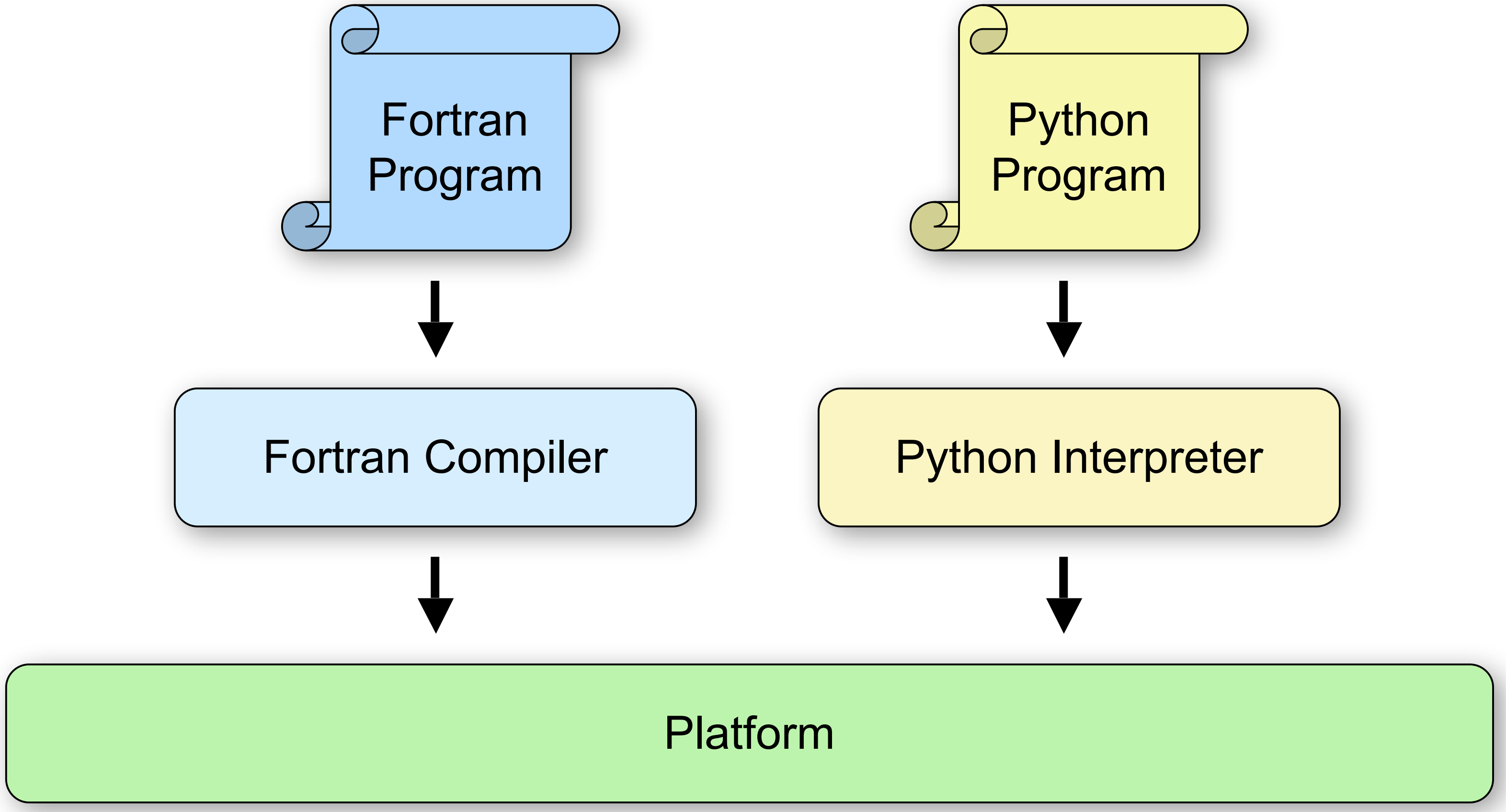
Katahdin

A Programming Language Where the Syntax and Semantics Are Mutable at Runtime

Chris Seaton

Traditional Development Tools

- A different runtime for each language that you use
- Using more than one language in a program is hard
- Languages are fixed by the original developer
- Developing new languages is hard



Fortran Program

Python Program

Fortran Compiler

Python Interpreter

Platform

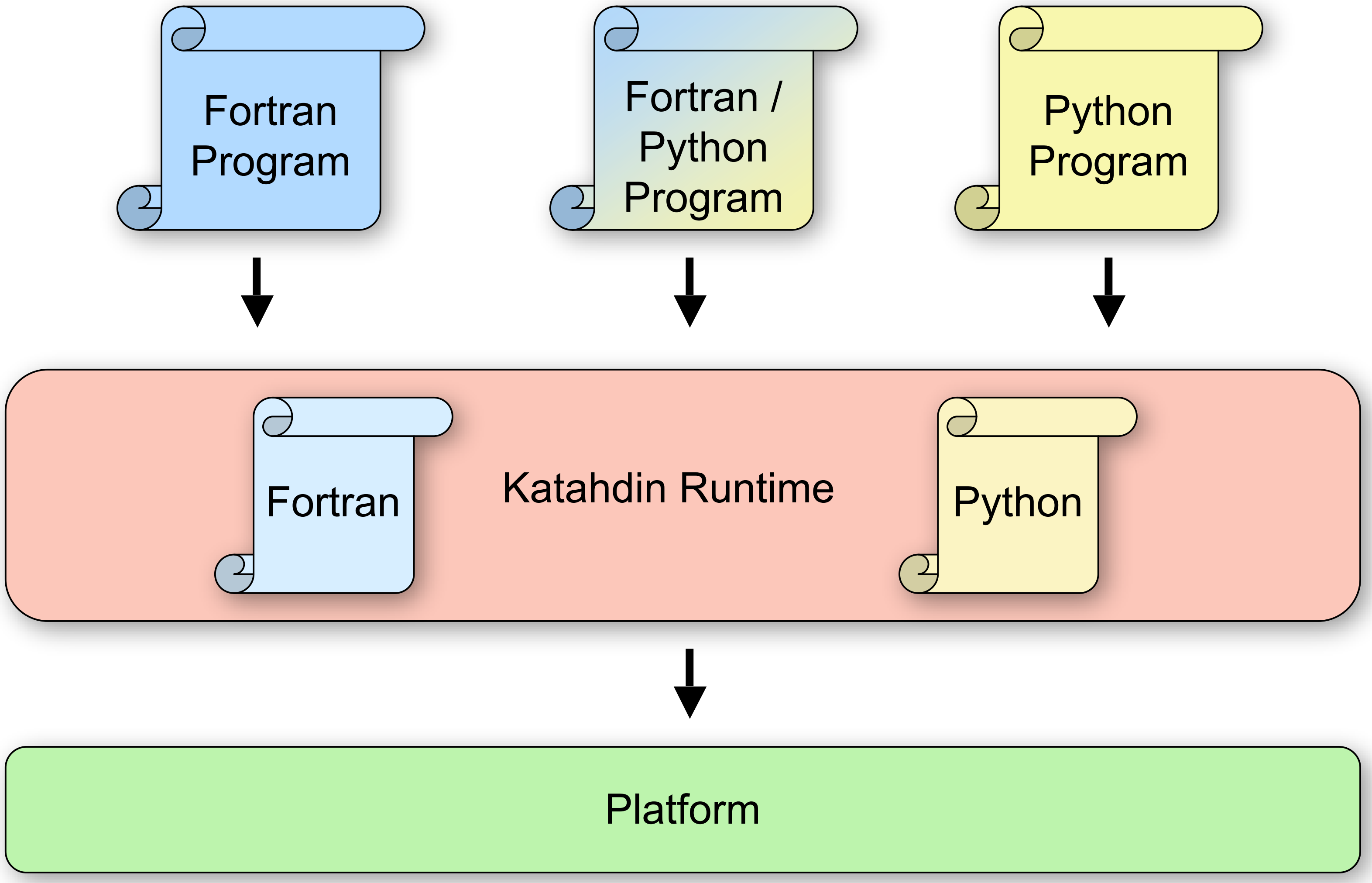
What Do We Want?

- We want a **single runtime** for multiple programming languages
- We want to **use more than one language** in the same program, the same file, even the same line
- We want to be able to **extend languages** as easily as we can define new functions
- We want to easily **define new languages**

Katahdin

A programming language and an interpreter

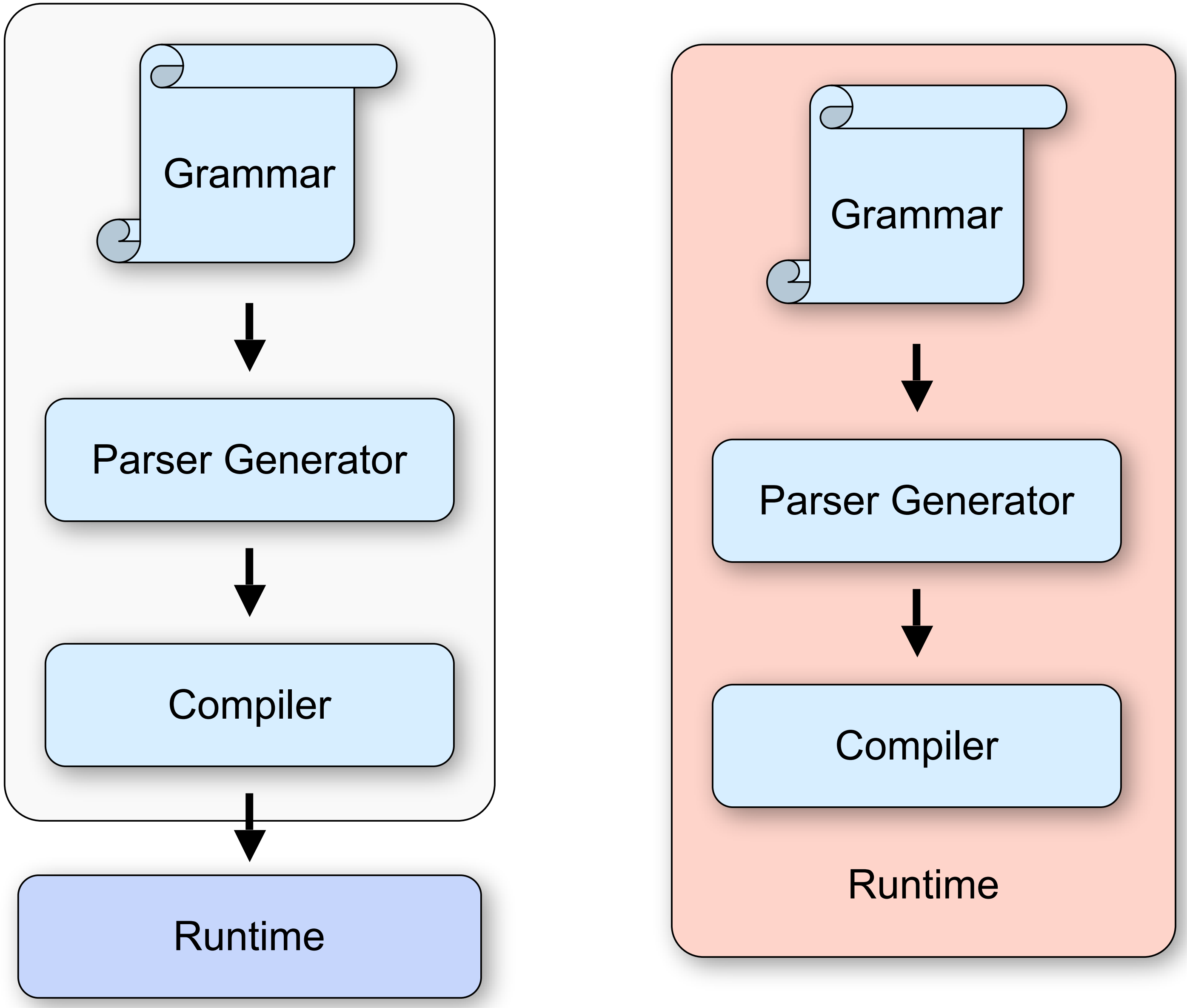
- The **syntax and semantics can be modified** by the running program
- Can **add new constructs** to the language, or define entire new languages
- **Composing the definitions** for two languages allows you to use those two languages at the same time


Fortran Program
Fortran / Python Program
Python Program
Fortran
Katahdin Runtime
Python
Platform

How Does Katahdin Work?

Take traditional development techniques and make them dynamic, runtime operations

- How is the **grammar** expressed?
- How is the grammar **parsed**?
- How are **semantics** expressed?
- How do we create **language definition modules**?



Grammar

Parser Generator

Compiler

Runtime

Grammar

Parser Generator

Compiler

Runtime

Expressing Grammar in Katahdin

- Based on **Parsing Expression Grammars** (PEGs)
- Described by Bryan Ford of MIT (2004) and related to the work of Alexander Birman (1970)
- Looks and feels very much like a regular expression or context-free grammar
- Expressed using Backus-Naur Form (BNF)
- My own extensions to better support modular grammars

Example: Modulo Operator

```
class ModExpression : Expression {
    pattern {
        option leftRecursive;
        a:Expression "%" b:Expression
    }
}
```

Katahdin’s Parsing Algorithm

- Based on **packrat parsing**
- Described by Bryan Ford (2002)
- Basically a top-down, recursive-descent parser that backtracks
- Sacrifices memory for speed – a linear time operation
- Other projects successfully applying packrat parsers to PEGs, but not from a mutable grammar, as Katahdin is

Expressing Semantics in Katahdin

- Semantics are the meaning of each construct in the language
- Express semantics as code that is directly executed
- Allows you to express one language in terms of another, or in terms of itself
- Code is written in methods in the construct’s `class` statement

Example: Modulo Operator

```
class ModExpression : Expression {
    pattern {
        option leftRecursive;
        a:Expression "%" b:Expression
    }

    method Get() {
        a = this.a.Get...();
        b = this.b.Get...();
        return a - (b * (a / b));
    }
}
```

Language Definition Modules

- If you define all the constructs in a language like this, you have a complete definition of the syntax and semantics of the language
- Store constructs in a module to be conveniently loaded
- Katahdin can automatically load a module based on file extension
- Users can explicitly load a module to merge with the current grammar

Results

- I have **achieved the goal** of making a programming language where the syntax and semantics are mutable at runtime
- My implementation of Katahdin is **mature and stable**
- Implemented **proof-of-concept language definitions** for SQL, Python, Fortran
- A paper describing the theory and implementation has been **submitted for publication** at GPCE 2007
- **Error handling and performance** need to be addressed – there is research that this work can be based on

Demonstration, Questions