flow

Programming Language Implementation / Cadence Implementation

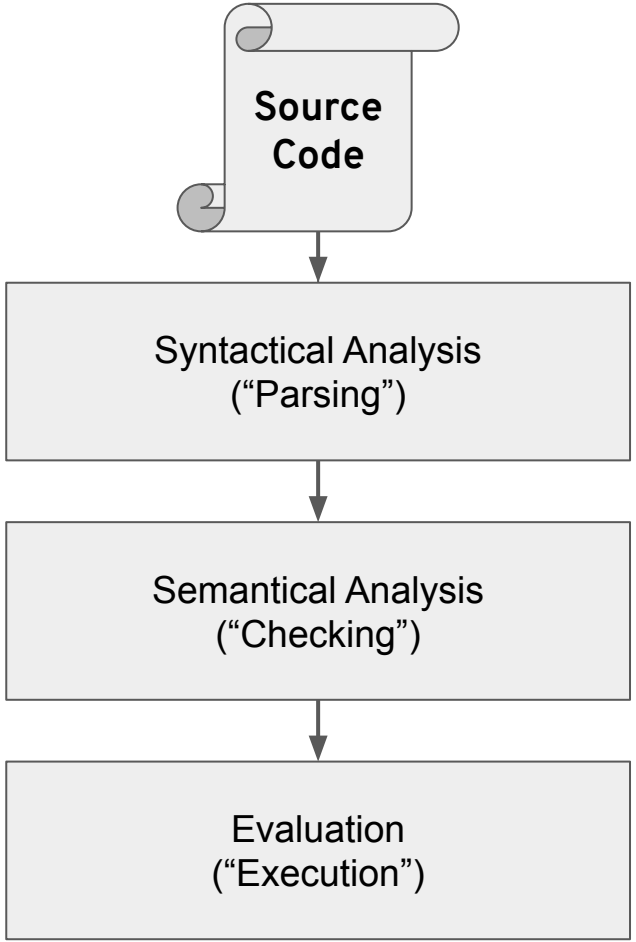
flow

Programming Language Implementation

flow

Phases

- Input is a program in text form
- On a high level there are 3 phases
- Each phase transforms and/or generates more information
- These phases are often sequential (as separate passes), but can be performed together (in one pass)
- Trade-off: complexity/features vs performance

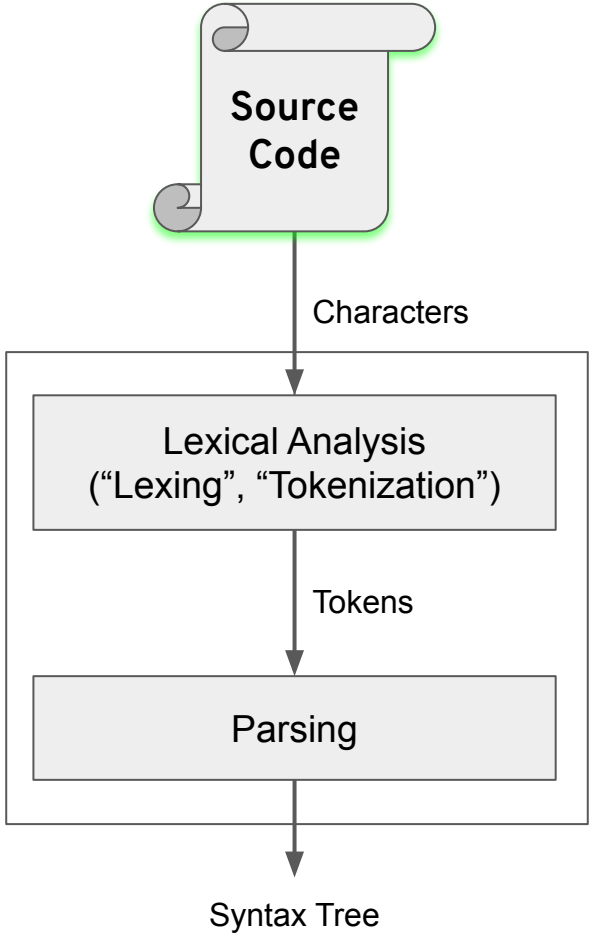


Syntactical Analysis

flow

Syntactical Analysis

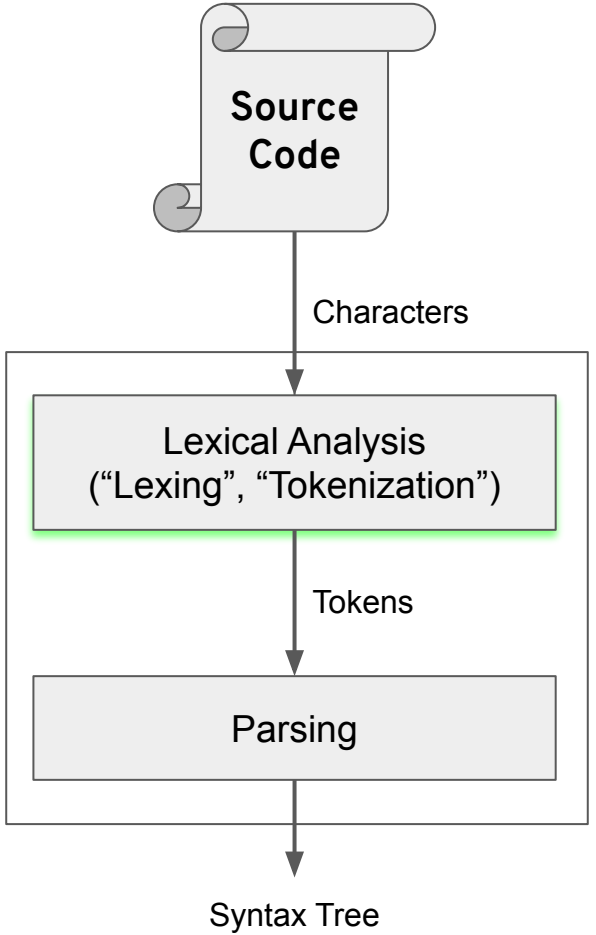
- Input is a program in text form (characters)



flow

Syntactical Analysis

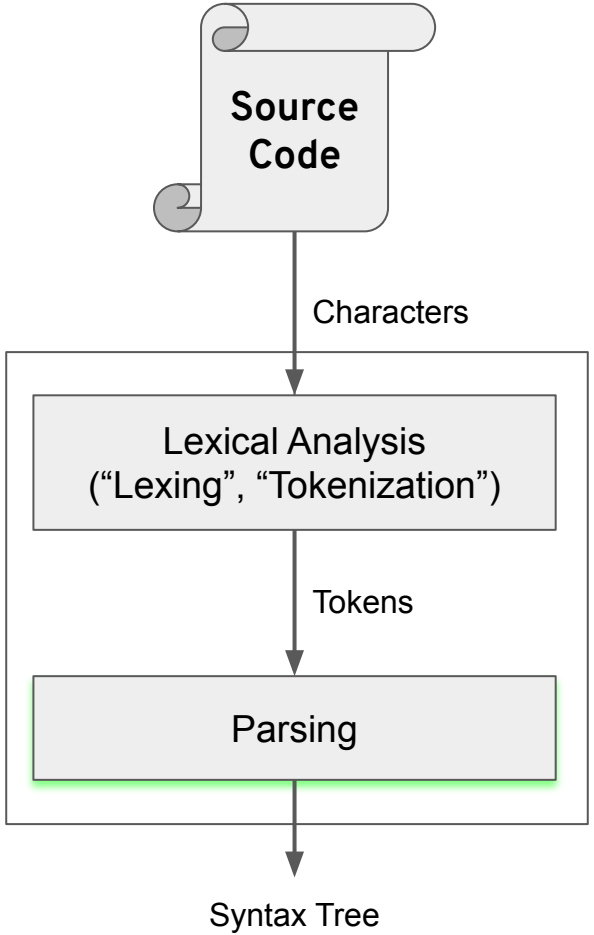
- First, the characters are split into tokens, e.g. "`fun foo() { return }`":
  - Keyword "`fun`"
  - Identifier "`foo`"
  - Open paren
  - Close paren
  - Open brace
  - Keyword "`return`"
  - Close brace



flow

Syntactical Analysis

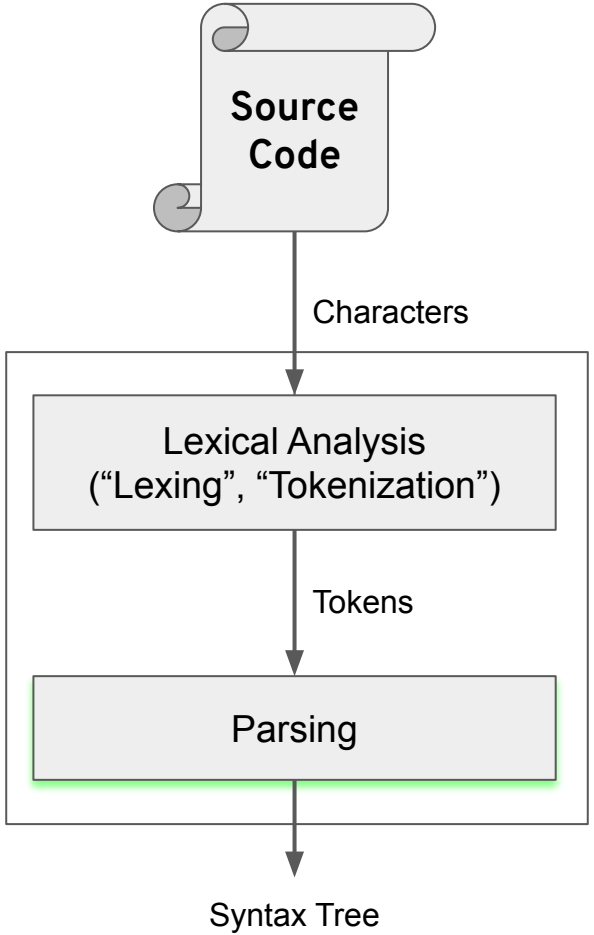
- The parser validates that the source code has a valid form, which is defined in a grammar (rules)
- For example, a function declaration:
  - Must start with the "`fun`" keyword
  - Must follow with an identifier, the name
  - Must follow with an open paren
  - Must follow with parameters (nested rule)
  - ...



flow

Syntactical Analysis

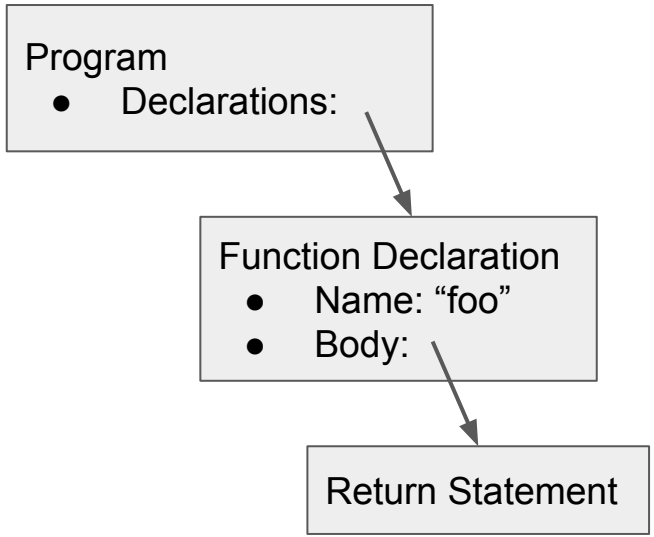
- If the program is invalid,
  i.e. the tokens don't follow the grammar,
  then errors are reported
- The parser ideally recovers from
  problems and parses the remainder

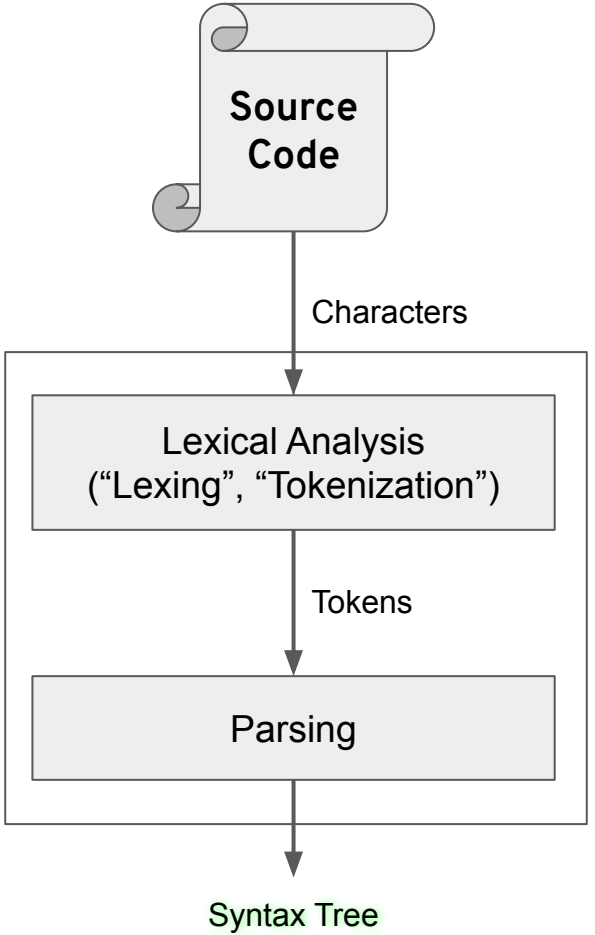


flow

Syntactical Analysis

- Then the tokens are transformed into a syntax tree, e.g.





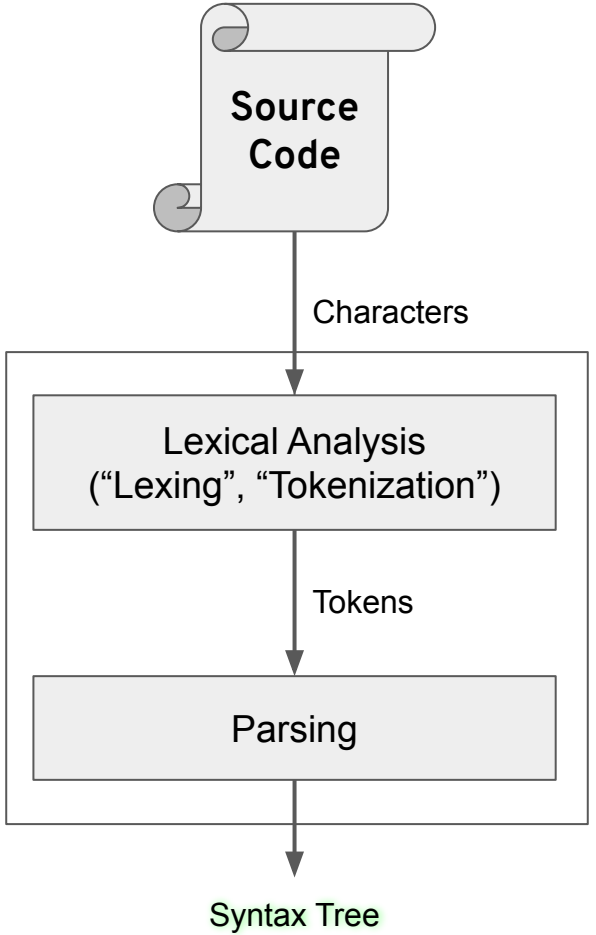
flow

Syntactical Analysis

- The syntax tree might be **concrete** or **abstract**

- A **concrete** syntax tree is shaped in the form of the grammar rules
- An **abstract** syntax tree is shaped for the semantical analysis and execution

- Some syntactical analysis phases may have both, some just one

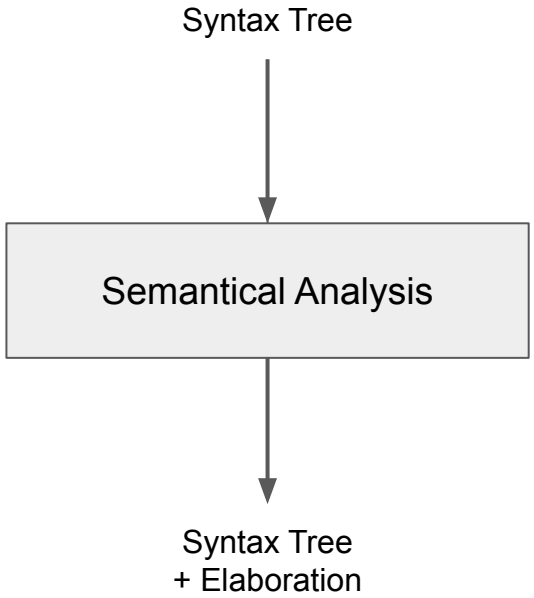


Semantical Analysis

flow

Semantical Analysis

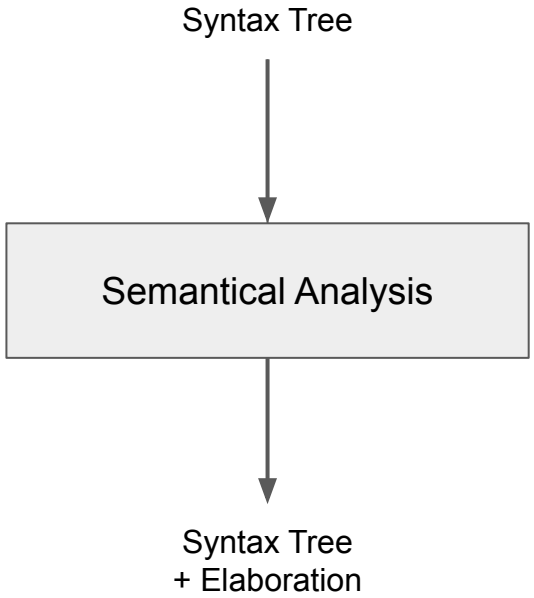
- Input is a syntax tree
- Produces an **elaboration**: Information about the program
- For example, resulting type of an expression like “a + 1”



flow

Semantical Analysis

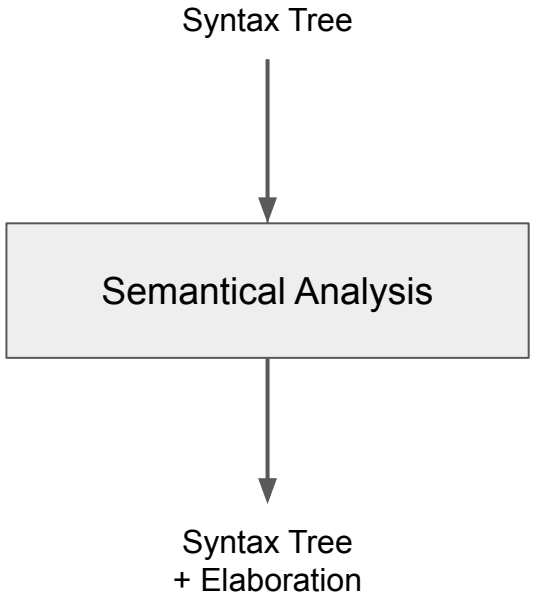
- Scoping: Regions where declarations are valid
- Determines (infers) types and validates typing rules (subtyping)



flow

Semantical Analysis

- Checks that the program is valid,
  which is defined in the semantics (rules)
- For example, a rule might be
  “+ is only defined for numbers”
- Given a variable “a” of type “String”,
  a program “a + 1” is invalid



flow

Evaluation

Evaluation

- Programs can be executed in different ways:
  - Interpretation
  - Compilation
  - A mix of both, e.g.
    - Compiling while interpreting: “Just-in-time compilation”
    - Compiling to a non-native instruction set
      (e.g. JVM, WebAssembly, etc.),
      then interpreting the binary

Evaluation

- How a program is executed is **not** defined by the language!
  - For example, C is not a “compiled language”:
    there are many C compilers, but also interpreters!
- There might be many implementations of a language:
  - Python: CPython (reference), PyPy, Jython, IronPython, etc.
  - Ruby: Ruby MRI (reference), Mruby, JRuby, IronRuby, etc.
  - C: GCC, Clang, CINT, etc.
  - JavaScript: V8, SpiderMonkey, JavaScriptCore, etc.

Evaluation

- **Interpretation:**
  - Input is AST and elaboration, or program of “flat” instructions
  - Program that executes the input program
  - For example:

    If an AST element or an instruction in the program **represents addition**, then the interpreter **performs the addition**

Evaluation

- **Compilation:**
  - Input is AST and elaboration
  - Output is binary of “flat” instructions,
    which can usually be run by a CPU
  - Might perform **optimizations**:
    - For example, the code “1 + 2” might be directly evaluated
      at compile time, once, so the addition does not has to be performed
      each time the program is executed
    - Dead code might be removed

Evaluation

- Trade-off: Time to execution vs execution time
  - For example, if a program is run many times,
    it makes sense to spend more time upfront, once,
    to reduce the execution time
  - If a program is only run once,
    compilation time might be longer than total execution time

flow

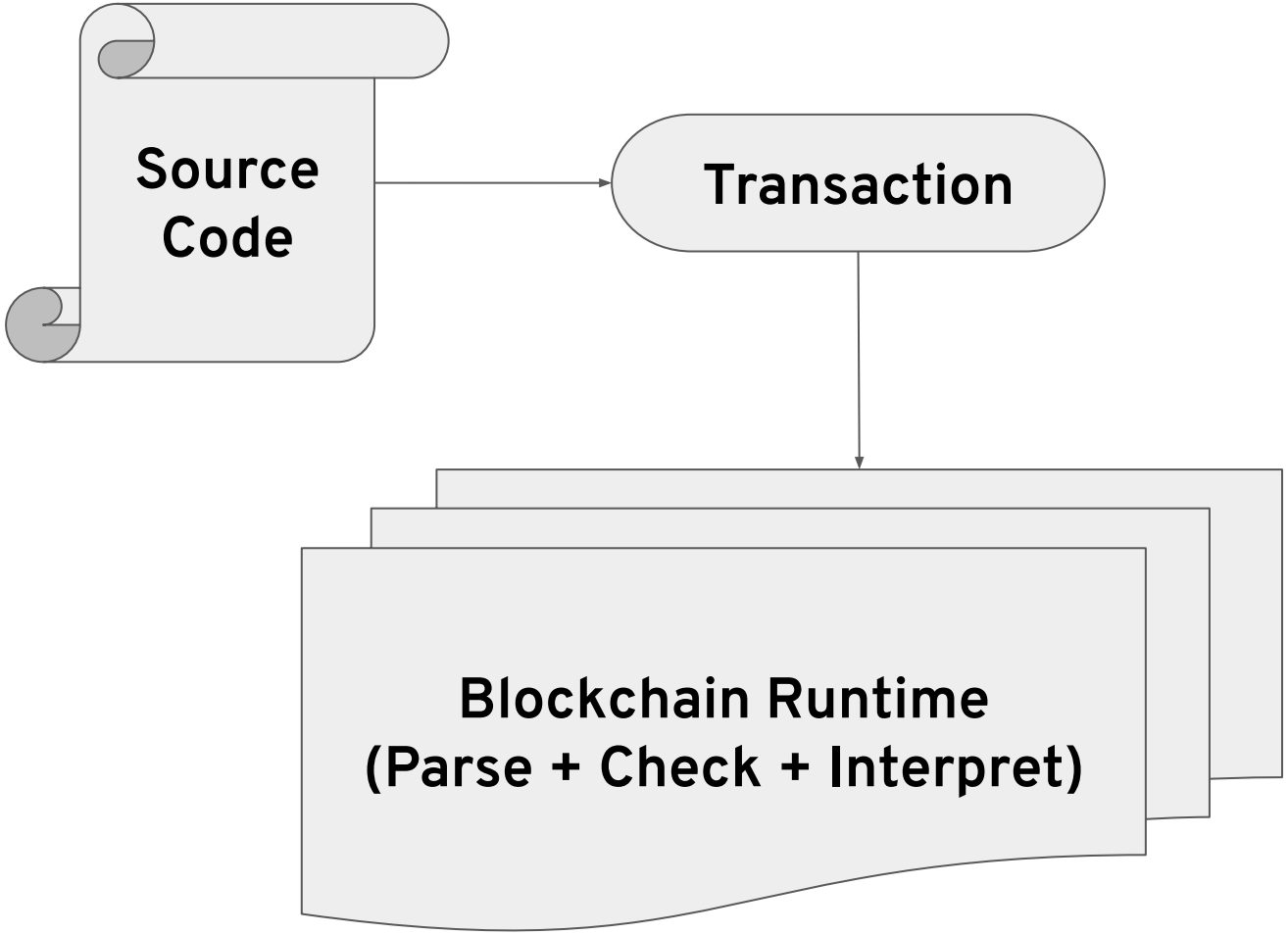
Cadence

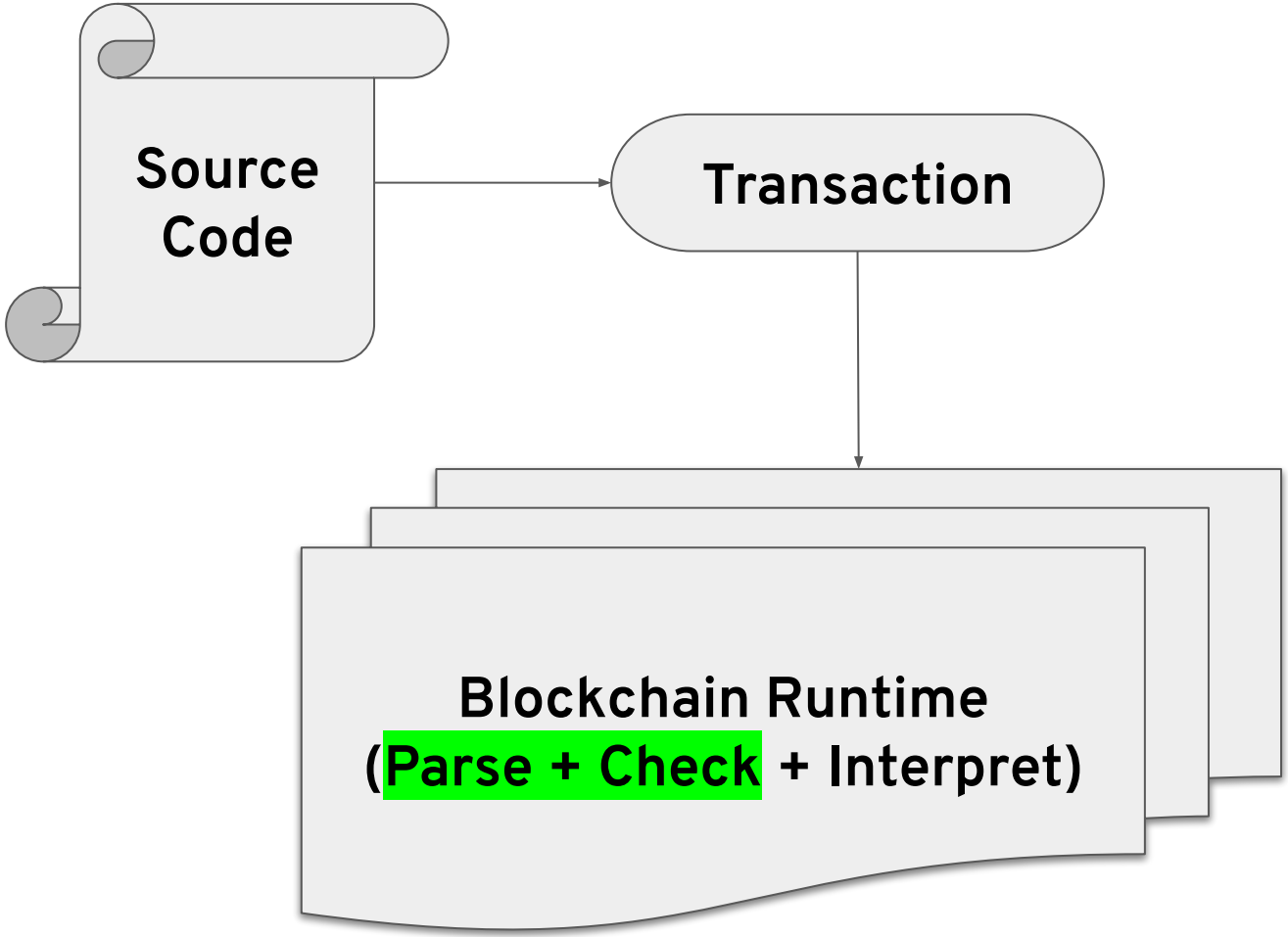
Phases

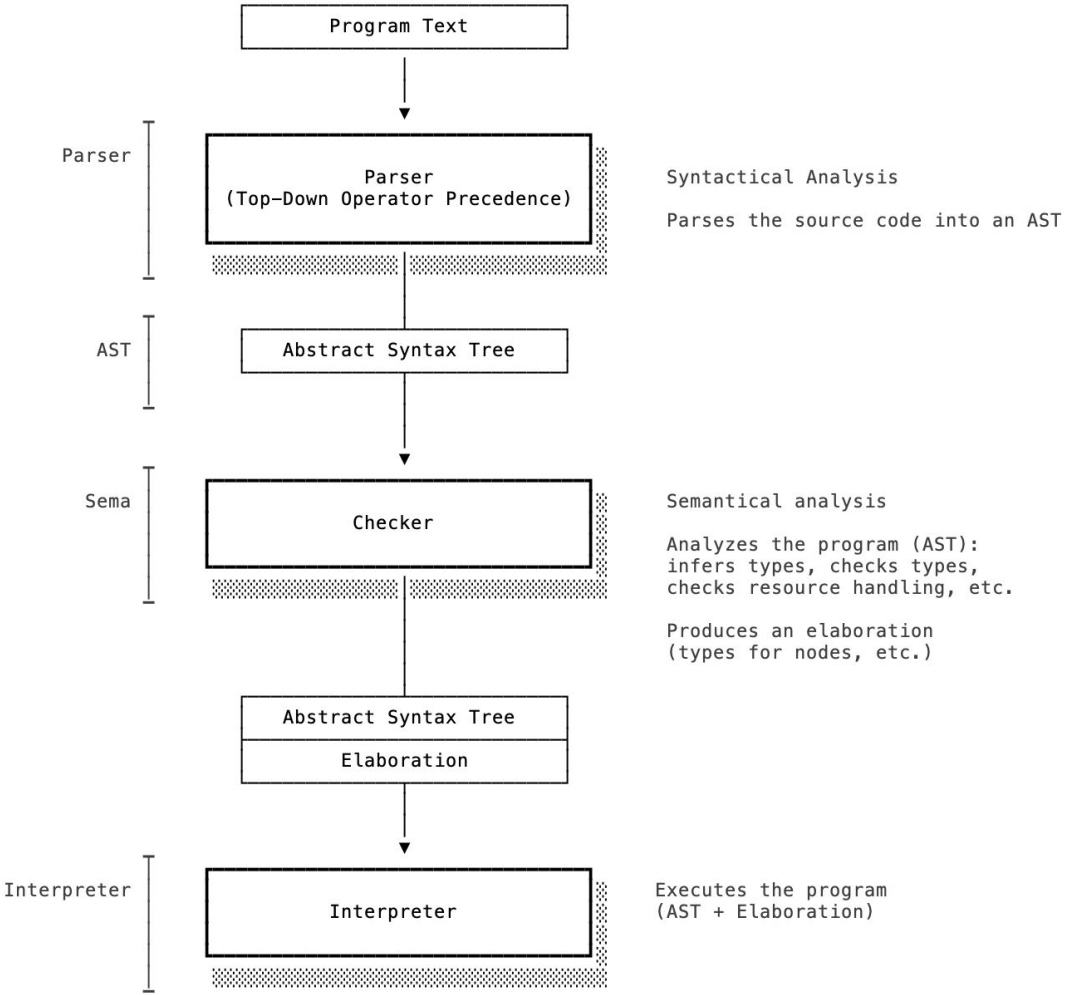
- Programs are uploaded to the chain as source code (deployed contracts, transactions, scripts)

- Execution:
  - Parse (syntactical analysis)
  - Check (semantic analysis)
  - Interpret

- Optional caching of parsed programs
- Checking result is elaboration (types of AST nodes)
- AST and elaboration are not stored on-chain


Source
Code
Transaction
Blockchain Runtime
(Parse + Check + Interpret)


Source Code
Transaction
Blockchain Runtime
(Parse + Check + Interpret)


Program Text
Parser
Parser
(Top-Down Operator Precedence)
Syntactical Analysis
Parses the source code into an AST
AST
Abstract Syntax Tree
Sema
Checker
Semantical analysis
Analyzes the program (AST):
infers types, checks types,
checks resource handling, etc.
Produces an elaboration
(types for nodes, etc.)
Abstract Syntax Tree
Elaboration
Interpreter
Interpreter
Executes the program
(AST + Elaboration)

Parsing

- Concurrent tokenization (goroutine)
- Produces AST directly, no CST

flow

Parsing

- Parser was initially generated using ANTLR
  - Pros:
    - Nice declarative grammar
    - Development speed
  - Cons:
    - Go backend is very slow
    - Hard to handle parse errors
    - Hard to handle ambiguities
    - Easy to introduce exponential blow-up
    - Construction of CST is additional step with associated overhead
- Replaced with hand-written Top-Down Operator Precedence (Pratt) parser

Checking

- Checks:
  - Type checks (e.g. subtyping, restrictions, etc.)
  - Resource tracking (construction, moves, destroys)
  - Interface conformance
  - Type requirement conformance
  - Definite initialization / use-before initialization
  - Literal range checks
  - Type storability
- Based on visitor over AST
  - Breadth-first traversal: use before declaration

Execution

- AST-walking interpreter, uses visitor

Tools

- Language Server + Visual Studio Code extension
  - Implements Language Server Protocol
  - Integrated into CLI
  - Integrated into Playground FE using WebAssembly
- REPL
  - Useful for development of Cadence
  - Would be nice to integrate it with emulator/network: re-use language server
- Documentation generator
- Debugger

Tools

- Command-line tools for parsing, checking, executing
  - Allow benchmarking
- Compatibility suite:
  - Checks out known repositories
  - Generates report about parsing/checking and performance regressions

Integration with Flow

- “Runtime” interface
  - Import handling (resolution, reading code)
  - Storage: read value, write value
  - Account management:
    - Account creation
    - Key management
    - Contract management
  - Event emission
  - Transaction information (signers)
  - Block information
  - Crypto (hashing, signature verification, etc.)
  - Random number generation

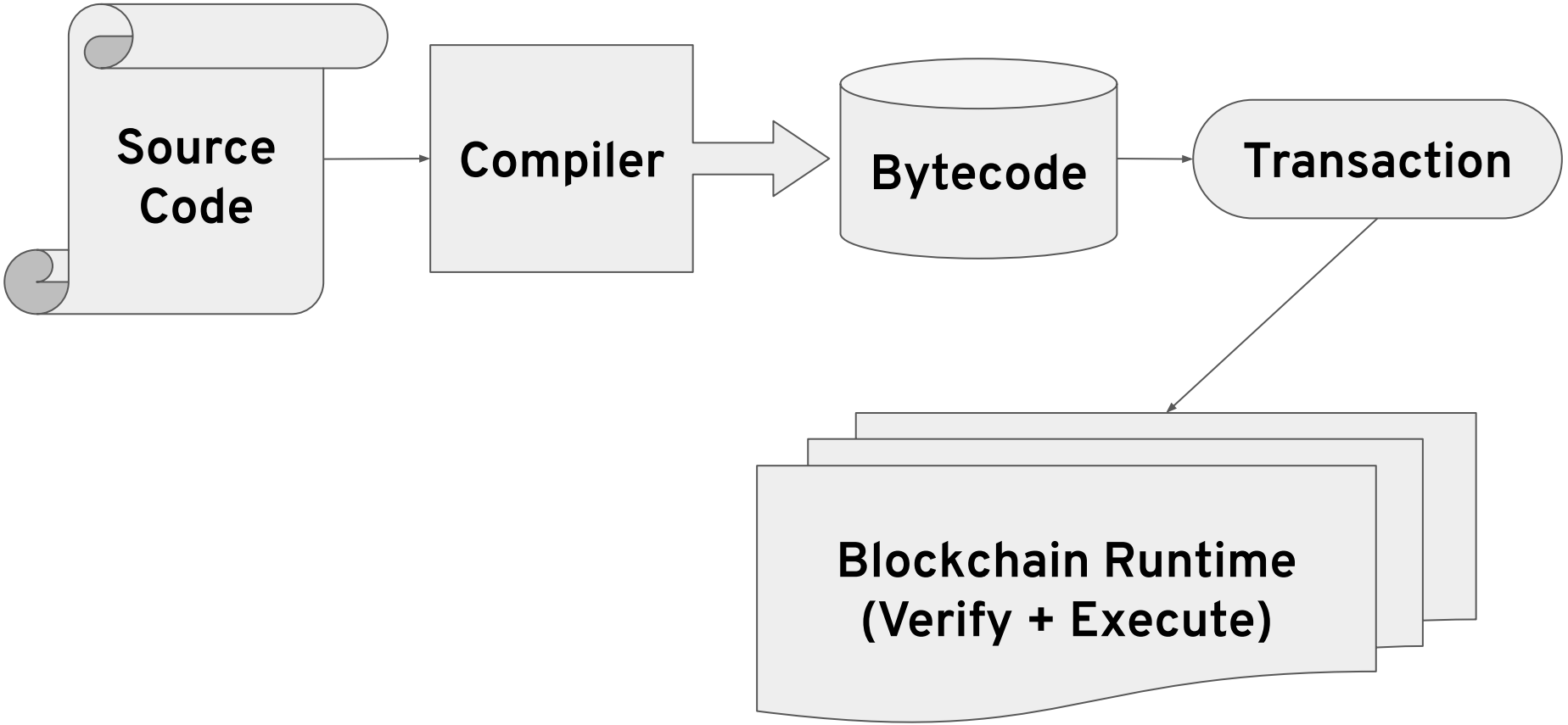
K Semantics

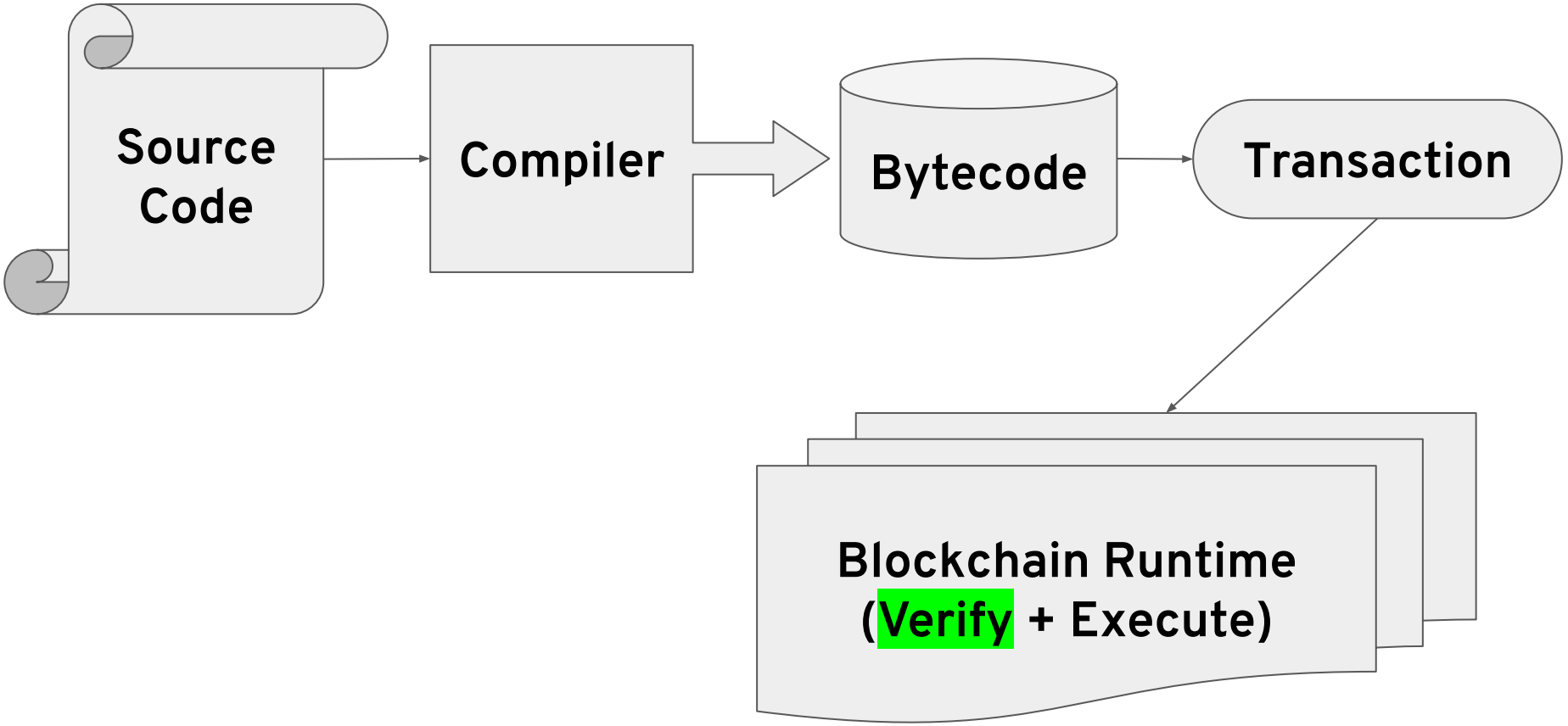
- We started with formalizing Cadence using the K Framework
- Subset of early version of Cadence (e.g., no resources)
- Declare syntax and semantics → **generate** interpreter, verifier, etc.
- “Correct by construction”
- Experience:
  - Requires lots of expert knowledge
  - Time consuming
  - Slow iteration
- Assumed we could use it as a tool for language design exploration
- Language better defined now, it would be nice to complete this eventually

flow

Performance and Efficiency: Execution

- Replace interpreter with compiler and virtual machine
- Compiler could initially be just used on-chain (to increase performance)
- By adding a bytecode verifier and using it on-chain,
  users could run compiler off-chain and submit compiled bytecode,
  the network wouldn't need to compile
- Bytecode verifier and Virtual Machine must enforce
  same security and safety semantics for bytecode
  as checker and interpreter do for Cadence source programs


Source Code
Compiler
Bytecode
Transaction
Blockchain Runtime
(Verify + Execute)


Source Code
Compiler
Bytecode
Transaction
Blockchain Runtime
(Verify + Execute)

Performance and Efficiency: Execution

- Requirements:
  - Deterministic (e.g. cannot have non-deterministic features like platform-dependent floating point semantics)
  - Portable
  - “Managed”: Instruction set must not allow direct access to private data
  - Size-efficient
  - Linear types / resource semantics
  - Similar to JVM/CL

Performance and Efficiency: Execution

- Requirements:
  - Deterministic (e.g. cannot have non-deterministic features like platform-dependent floating point semantics)
  - Portable
  - “Managed”: Instruction set must not allow direct access to private data
  - Size-efficient
  - Linear types / resource semantics
  - Similar to JVM/CL

flow

Performance and Efficiency: Execution

- Options
  - WebAssembly
    - Has external references/values
    - Standard
    - Many implementations and lots of tooling
    - Formal semantics
    - Is missing linear types
    - Need to extend instruction set
  - (Cont.)

flow

Performance and Efficiency: Execution

- Options (continued)
  - MoveVM
    - Has linear types
    - Is missing external references/values
    - Only one implementation
    - Need to extend instruction set and virtual machine
    - Opposition to extension, e.g. with dynamic features
  - (Cont.)

flow

Performance and Efficiency: Execution

- Options (continued)
  - LLVM IR:
    - Not portable: architecture specific
    - Generates very efficient code
    - Slow compilation speed
  - Custom?
    - Lots of work and re-inventing the wheel
    - Source of bugs

flow

Performance and Efficiency: Execution

- Currently we have
  - WebAssembly Binary (WASM) reader/writer
  - Start of IR
  - Start of compiler
- Goal is to have MVP
  - Compiler and verifier for subset (e.g resources, interfaces)
  - Demonstrate approach satisfies goals

flow

Performance and Efficiency: Storage

- Storage operations account for a large portion of execution
- Programs are read/write heavy, usually not compute intensive
- Storage format: CBOR
  - RFC standard
  - Efficient
- Streaming decoding and encoding (no intermediate objects)
- Lazy decoding (read data, only decode when/if needed)

Reliability

- Bugs: crashers, security issues (!)
- Known problems
  - Manually written tests
- Unknown problems
  - Automatic testing: Fuzzing

flow

Questions?