flow

Cadence

Implementation: Past, Present, and Future

flow

Implementation Language

- Implemented in Go
- Also considered Rust
- Trade-off: Implementation/iteration speed / time to MVP > performance

Implementation in Go

- Pros:
  - Lots of expertise at Dapper Labs
  - Commonly used in blockchain space
  - Fast iteration / development speed
  - Easily integratable into node software
  - Can leverage garbage collection in Go for garbage collection in Cadence
  - Can easily cross-compile (Linux, Windows, WebAssembly)
  - Performance is good enough

Implementation in Go

- Cons:
  - Go is very minimalistic (no generics, structural typing, no enums, etc.) – Suboptimal for safely implementing a programming language
  - Performance could be better

flow

Implementation Language

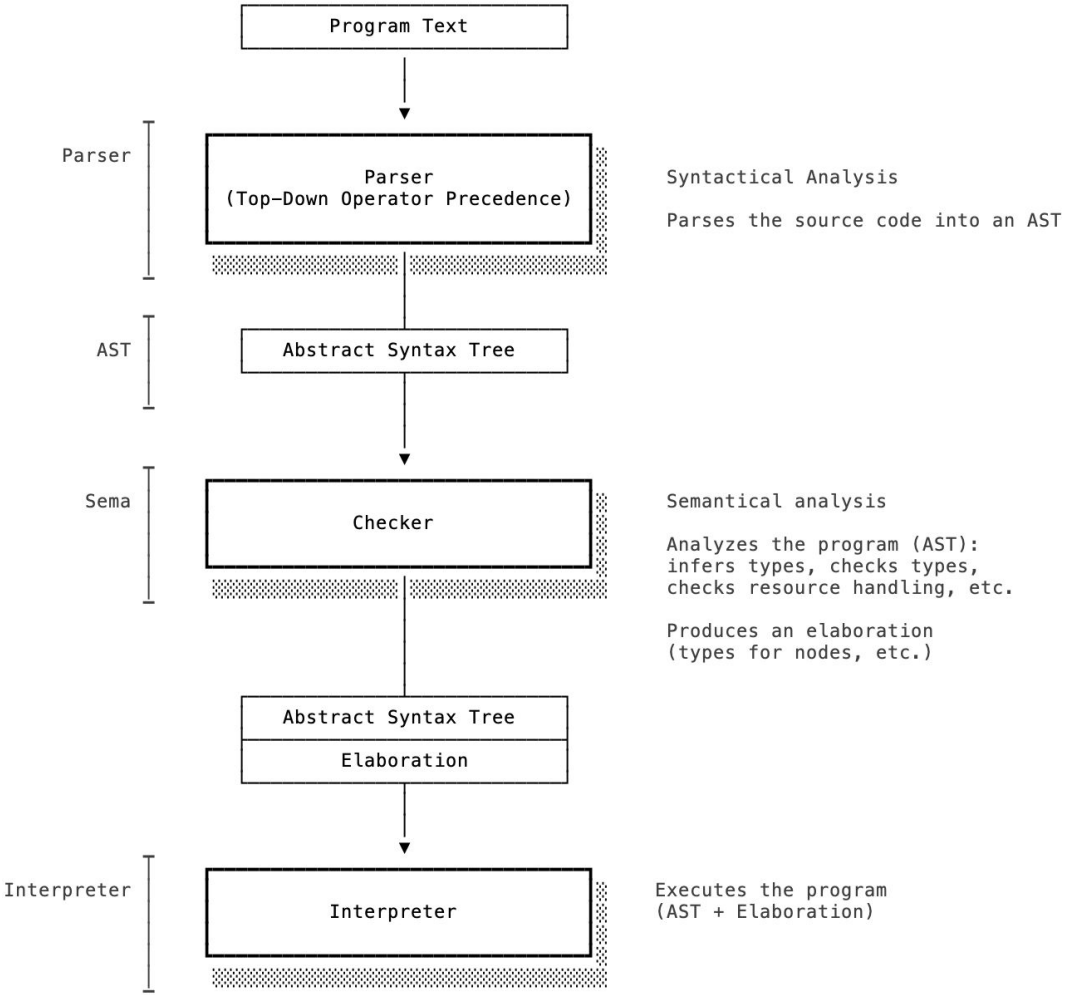
- Might implement new components or re-implement components in another language

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


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

- **Input:** Source
- Lexer recognizes tokens from source (e.g. identifier, string literal, etc.)
- Parser:
  - Hand-written Top-Down Operator Precedence (Pratt) parser
    - Can declaratively specify prefix, infix, and postfix operators and their precedence
  - Supports backtracking (buffering + replay) for ambiguous cases
    - Less than operator vs. type argument: `a<b>()`
    - Greater than operator vs. bitwise right shift operator: `a >> b` (again due to type arguments: `a<b<c>>()`)
- **Output:** AST
  - JSON output for other tools

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
    - Concrete Syntax Tree (CST) is additional step with associated overhead

Checking

- **Input:** AST

- Semantic analysis of program
  - Type checks (e.g. subtyping, restrictions, etc.)
  - Resource tracking (construction, moves, destroys)
  - Interface conformance
  - Type requirement conformance
  - Definite initialization
  - Literal range checks
  - Type storability
- Based on visitor over AST, breadth-first traversal

- **Output:** Elaboration: Type information for AST nodes

Execution

- **Input:** AST + Elaboration

- AST-walking interpreter, uses visitor
- Based on trampolining:
  - Prevent stack overflow during execution
  - Can halt + resume execution (useful for debugger)
  - Limited by heap space instead of stack space (though Go's goroutines automatically grow!)

flow

Storage

- Cadence has a high-level storage API, path-value based
- Mapped to low-level storage API, key-value based (bytes)
- Data is encoded as CBOR
  - Standard: RFC
  - Efficient: size, encoding/decoding performance
  - Extensible: allows encoding values with rich types, self-describing
- Storage optimization: “Deferral” (transparent persistence)
  - Save values of resource dictionaries in separate storage keys
  - Lazy-loading of values when accessed by program

Tests

- Extensive source based tests (i.e. input is program)
  for parsing, checking, interpretation,
- Generation of tests to cover all cases (e.g. test all number types)
- Fuzzing

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

flow

Integration

- “Runtime” interface
  - Import handling
  - 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.)

Integration

- Currently a “pull”-based architecture:
  Cadence contains type and value declarations (e.g. block information, crypto)
  and requires host environment to provide functionality

- Future: “push”-based architecture:
  Invert, refactor non-core/Flow-specific types and values
  from Cadence to node software

- Cadence recently gained support for allowing the host environment
  to inject values when executing programs (transactions and scripts)

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 → derive 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

Future: Features

- Expand standard library
  - Account API (e.g. key management)
  - Storage API (e.g. querying, iteration)
- Extensibility
  (add data and functionality to existing types without changing original code)

- Roadmap

Future: Tools

- Testing framework

- Pretty printer
- Documentation generator
- Debugger

flow

Future: Performance and Efficiency

- 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

flow

Future: Performance and Efficiency

- 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

Future: Performance and Efficiency

- 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

Future: Performance and Efficiency

- 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

Future: Performance and Efficiency

- 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

Future: Performance and Efficiency

- 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

Future: Performance and Efficiency

- Extend transparent persistence

flow

Questions?