Solidity

Solidity to Ethereum Bytecode in Spoofax

Taico Aerts

Overview

- Smart Contracts
- Project focus
- Solidity
- The Ethereum Virtual Machine
- EVM Bytecode (EBC)
- Limitations and technical issues
- Demonstration

Smart Contracts

- Program on the blockchain
- Transaction with money
- All execution costs money
- Remaining money at the end is returned

Smart Contracts: Example

- Event Ticket Sale
  - Transfer money to contract
  - Ticket is created for you and returned
  - Ticket validation is stored on the blockchain
  - Ticket money is sent to organizer
  - No infrastructure required

Smart Contracts: Example

- Kickstarter without 3rd party
  - People can invest money
  - If the goal is met
    - Money sent to creator
  - Otherwise
    - Money returned to backers
  - Contract holds the money
  - Blockchain guarantees
    - Immutable
    - Output validated

Project Focus

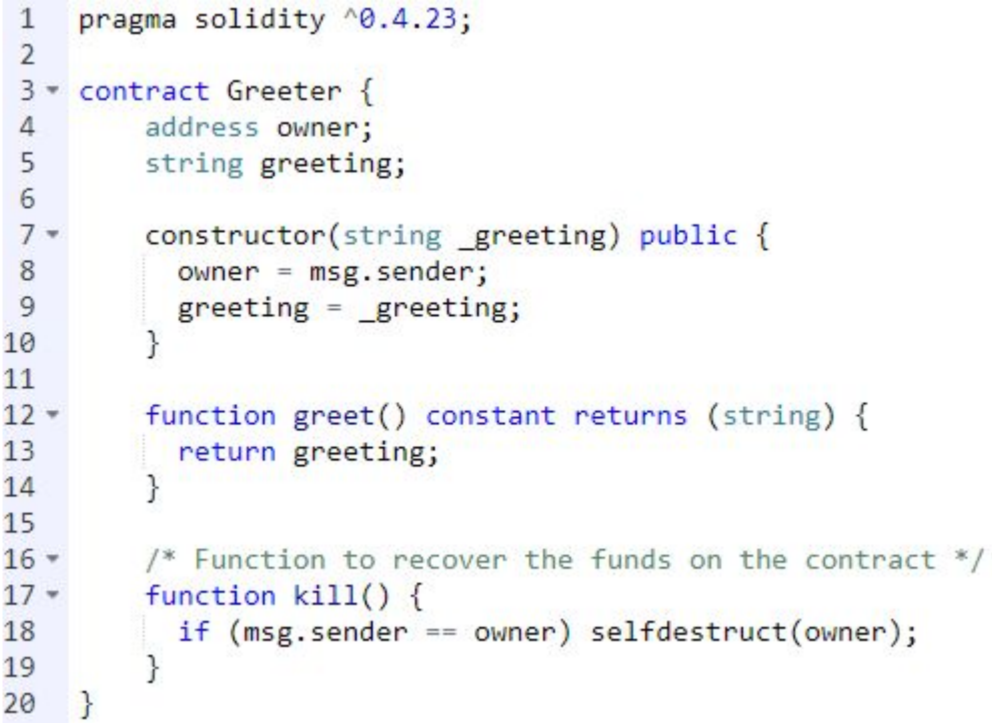
- Initially
  - Compiler and optimization
- Gradually
  - Type checking and compiler
    - Good type checking required for correct compilation
    - Good type checking required for optimizations

Solidity

Solidity

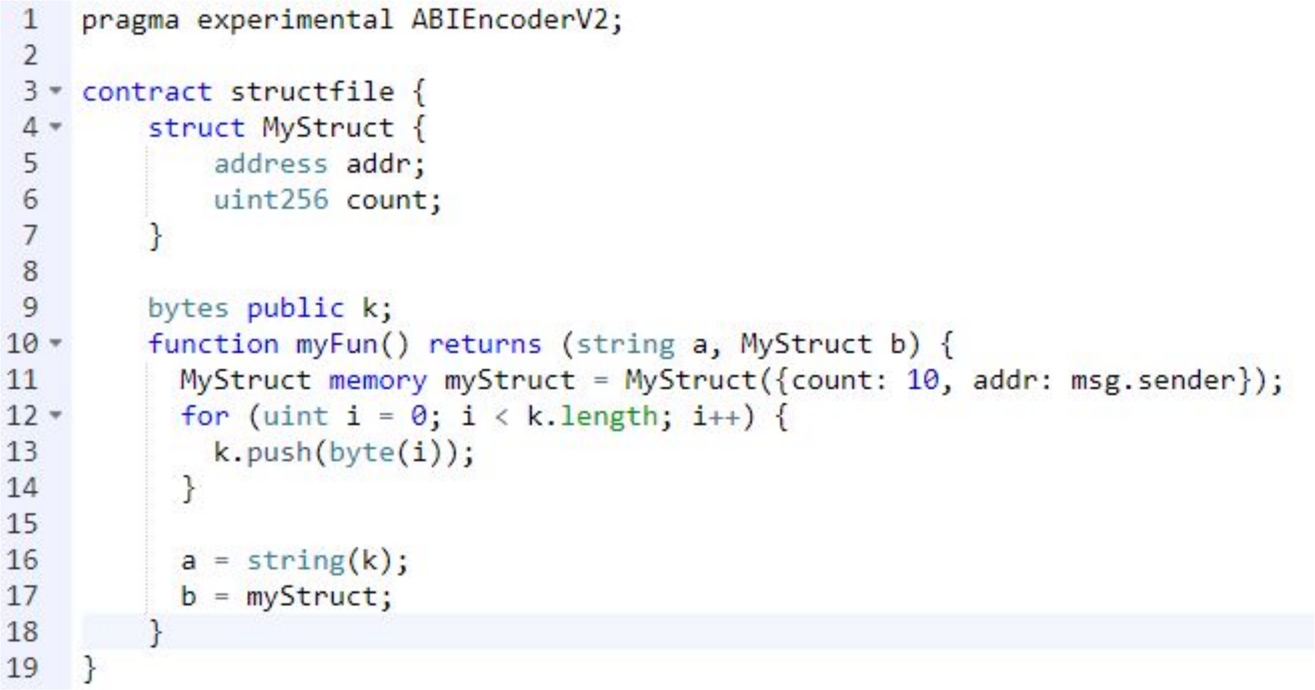
- A bit like JavaScript / Python / C++
- Contract = Class
- Very strict type system
- Complex user-defined types

Solidity



```
pragma solidity ^0.4.23;

contract Greeter {
    address owner;
    string greeting;

    constructor(string _greeting) public {
      owner = msg.sender;
      greeting = _greeting;
    }

    function greet() constant returns (string) {
      return greeting;
    }

    /* Function to recover the funds on the contract */
    function kill() {
      if (msg.sender == owner) selfdestruct(owner);
    }
}
```

Solidity



```
pragma experimental ABIEncoderV2;

contract structfile {
    struct MyStruct {
        address addr;
        uint256 count;
    }

    bytes public k;
    function myFun() returns (string a, MyStruct b) {
      MyStruct memory myStruct = MyStruct({count: 10, addr: msg.sender});
      for (uint i = 0; i < k.length; i++) {
        k.push(byte(i));
      }

      a = string(k);
      b = myStruct;
    }
}
```

Types

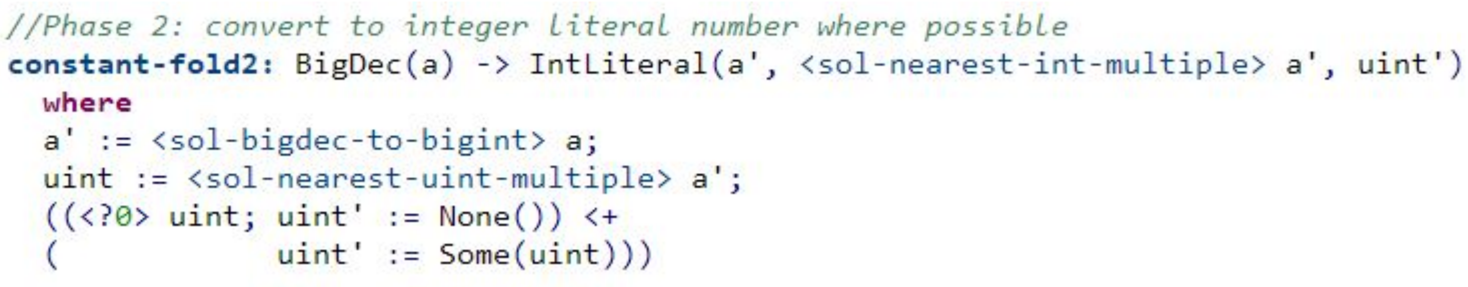
- int8, int16, int24, …, int 256
- uint8, …
- bytes1, bytes2, …, bytes32
- Does it matter?
  - Yes
  - Type checking is VERY strict
  - Compiler needs to know exact type

Types

- Picky example:
    - -1 ** 2 → allowed, 1
    - int8 y = -1; y ** 2 → not allowed
    - uint8 y = 1; y ** 2 → allowed
- Difference between literal int and int variable

Types

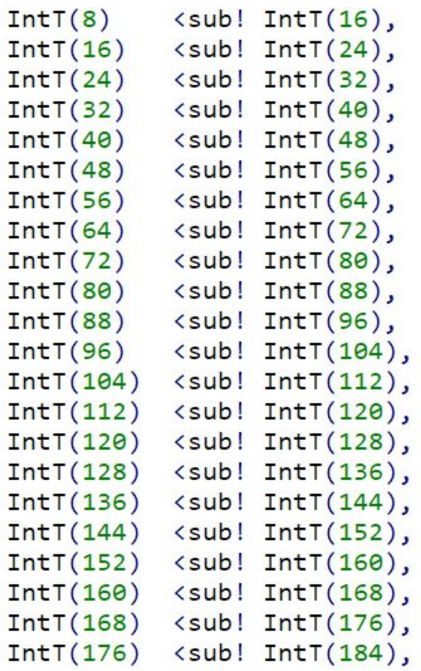
- Solution part 1:
  - Compute amount of bits required for numbers



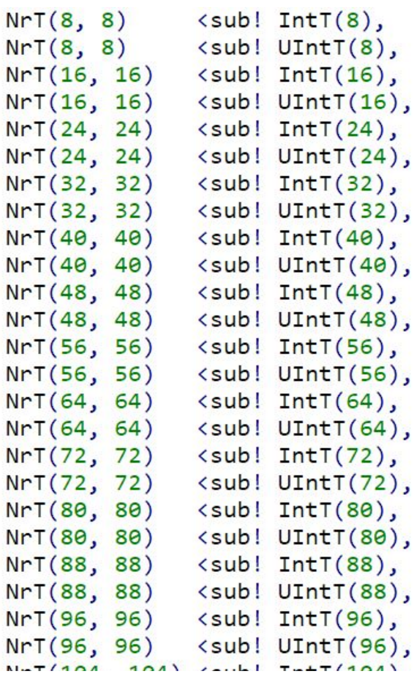
```
//Phase 2: convert to integer literal number where possible
constant-fold2: BigDec(a) -> IntLiteral(a', <sol-nearest-int-multiple> a', uint')
  where
  a' := <sol-bigdec-to-bigint> a;
  uint := <sol-nearest-uint-multiple> a';
  ((<?0> uint; uint' := None()) <+
  (            uint' := Some(uint)))
```

Types

- Solution part 2: massive type lattice



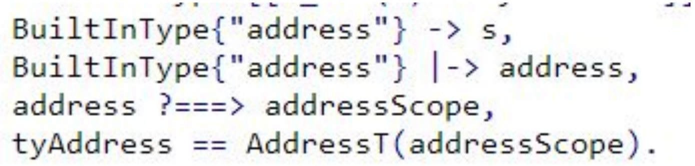
```
IntT(8)     <sub! IntT(16),
IntT(16)    <sub! IntT(24),
IntT(24)    <sub! IntT(32),
IntT(32)    <sub! IntT(40),
IntT(40)    <sub! IntT(48),
IntT(48)    <sub! IntT(56),
IntT(56)    <sub! IntT(64),
IntT(64)    <sub! IntT(72),
IntT(72)    <sub! IntT(80),
IntT(80)    <sub! IntT(88),
IntT(88)    <sub! IntT(96),
IntT(96)    <sub! IntT(104),
IntT(104)   <sub! IntT(112),
IntT(112)   <sub! IntT(120),
IntT(120)   <sub! IntT(128),
IntT(128)   <sub! IntT(136),
IntT(136)   <sub! IntT(144),
IntT(144)   <sub! IntT(152),
IntT(152)   <sub! IntT(160),
IntT(160)   <sub! IntT(168),
IntT(168)   <sub! IntT(176),
IntT(176)   <sub! IntT(184),
```



```
NrT(8, 8)       <sub! IntT(8),
NrT(8, 8)       <sub! UIntT(8),
NrT(16, 16)     <sub! IntT(16),
NrT(16, 16)     <sub! UIntT(16),
NrT(24, 24)     <sub! IntT(24),
NrT(24, 24)     <sub! UIntT(24),
NrT(32, 32)     <sub! IntT(32),
NrT(32, 32)     <sub! UIntT(32),
NrT(40, 40)     <sub! IntT(40),
NrT(40, 40)     <sub! UIntT(40),
NrT(48, 48)     <sub! IntT(48),
NrT(48, 48)     <sub! UIntT(48),
NrT(56, 56)     <sub! IntT(56),
NrT(56, 56)     <sub! UIntT(56),
NrT(64, 64)     <sub! IntT(64),
NrT(64, 64)     <sub! UIntT(64),
NrT(72, 72)     <sub! IntT(72),
NrT(72, 72)     <sub! UIntT(72),
NrT(80, 80)     <sub! IntT(80),
NrT(80, 80)     <sub! UIntT(80),
NrT(88, 88)     <sub! IntT(88),
NrT(88, 88)     <sub! UIntT(88),
NrT(96, 96)     <sub! IntT(96),
NrT(96, 96)     <sub! UIntT(96),
NrT(104, 104)   <sub! IntT(104),
```

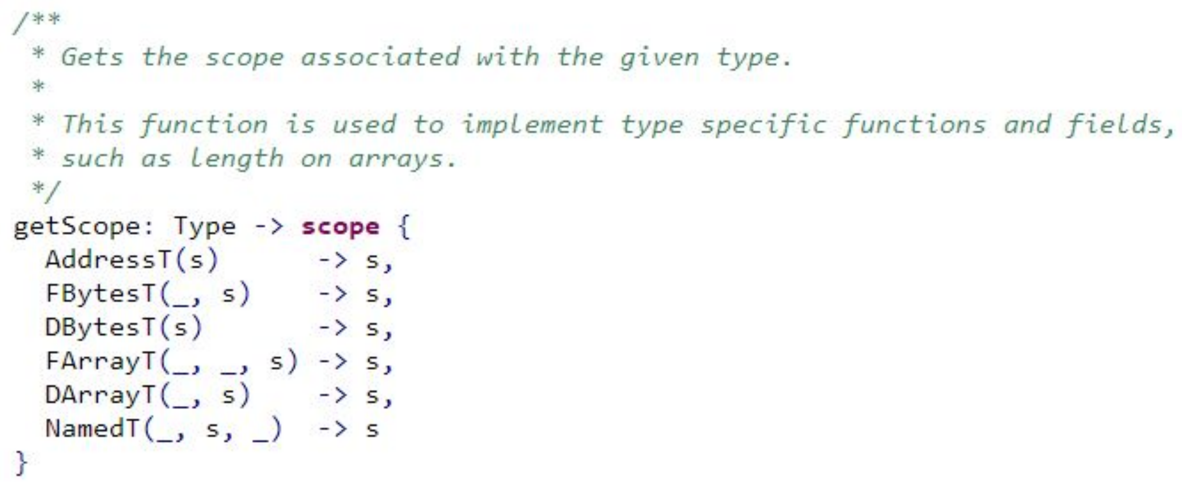
Calls on simple types

- Length on arrays
- Fields and methods on addresses
- Solution for flexible support
  - Scopes in basic types
  - getScope function
  - Works, but resolve required to get a type

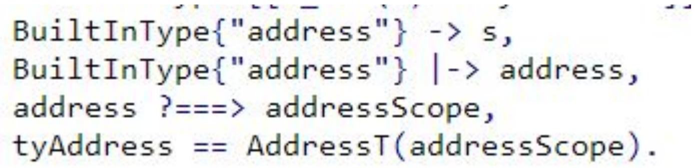


```
BuiltInType{"address"} -> s,
BuiltInType{"address"} |-> address,
address ?===> addressScope,
tyAddress == AddressT(addressScope).
```

Calls on simple types



```
/**
 * Gets the scope associated with the given type.
 *
 * This function is used to implement type specific functions and fields,
 * such as length on arrays.
 */
getScope: Type -> scope {
  AddressT(s)       -> s,
  FBytesT(_, s)     -> s,
  DBytesT(s)        -> s,
  FArrayT(_, _, s) -> s,
  DArrayT(_, s)     -> s,
  NamedT(_, s, _)   -> s
}
```



```
BuiltInType{"address"} -> s,
BuiltInType{"address"} |-> address,
address ?===> addressScope,
tyAddress == AddressT(addressScope).
```

Big Numbers

- 256-bits integers
    - Stratego: 32-bits integers
    - BigDecimal and BigInteger
- Flexible constant syntax
    - Fractions with infinite precision
    - Constant Folding
        - Compute constants with compiler

Big Numbers

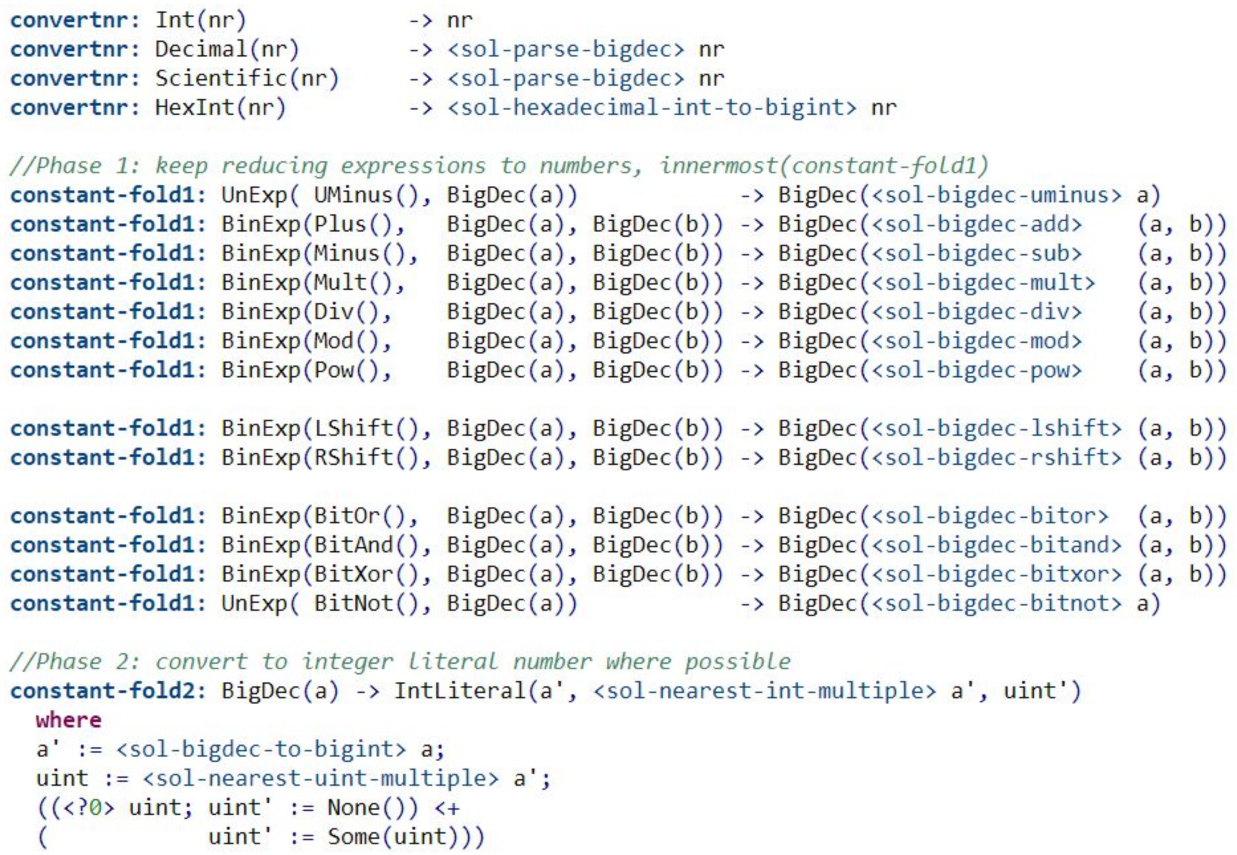
int x = 1.01 * 100; → int x = 101;

int x = 1.01 * 10; → error, 10.1 is not an integer

int8 x = 2 ** 2000 - 2 ** 2000 → int8 x = 0;

As long as the end result fits, everything is fine.

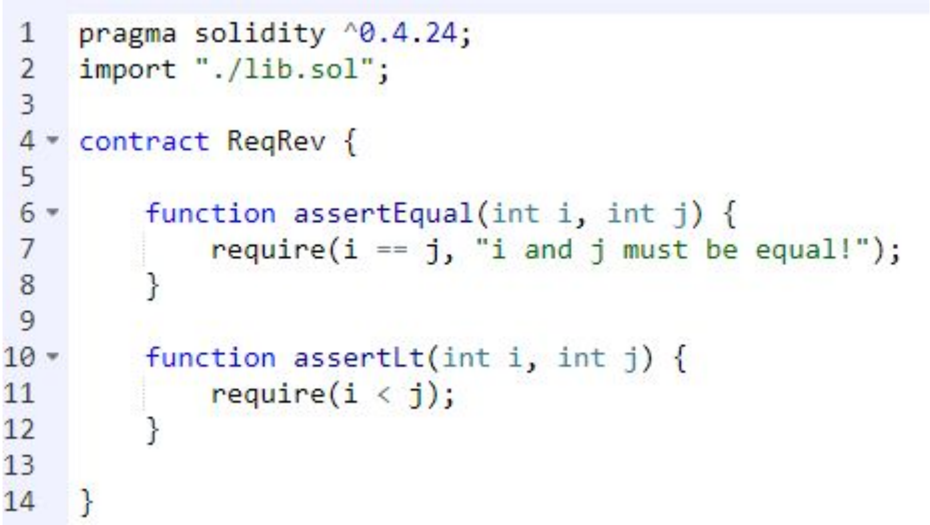
Big Numbers



```
convertnr: Int(nr)            -> nr
convertnr: Decimal(nr)        -> <sol-parse-bigdec> nr
convertnr: Scientific(nr)     -> <sol-parse-bigdec> nr
convertnr: HexInt(nr)         -> <sol-hexadecimal-int-to-bigint> nr

//Phase 1: keep reducing expressions to numbers, innermost(constant-fold1)
constant-fold1: UnExp( UMinus(), BigDec(a))            -> BigDec(<sol-bigdec-uminus> a)
constant-fold1: BinExp(Plus(),   BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-add>    (a, b))
constant-fold1: BinExp(Minus(),  BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-sub>    (a, b))
constant-fold1: BinExp(Mult(),   BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-mult>   (a, b))
constant-fold1: BinExp(Div(),    BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-div>    (a, b))
constant-fold1: BinExp(Mod(),    BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-mod>    (a, b))
constant-fold1: BinExp(Pow(),    BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-pow>    (a, b))

constant-fold1: BinExp(LShift(), BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-lshift> (a, b))
constant-fold1: BinExp(RShift(), BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-rshift> (a, b))

constant-fold1: BinExp(BitOr(),  BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-bitor>  (a, b))
constant-fold1: BinExp(BitAnd(), BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-bitand> (a, b))
constant-fold1: BinExp(BitXor(), BigDec(a), BigDec(b)) -> BigDec(<sol-bigdec-bitxor> (a, b))
constant-fold1: UnExp( BitNot(), BigDec(a))            -> BigDec(<sol-bigdec-bitnot> a)

//Phase 2: convert to integer literal number where possible
constant-fold2: BigDec(a) -> IntLiteral(a', <sol-nearest-int-multiple> a', uint')
  where
  a' := <sol-bigdec-to-bigint> a;
  uint := <sol-nearest-uint-multiple> a';
  ((<?0> uint; uint' := None()) <+
  (            uint' := Some(uint)))
```

Builtin Functions

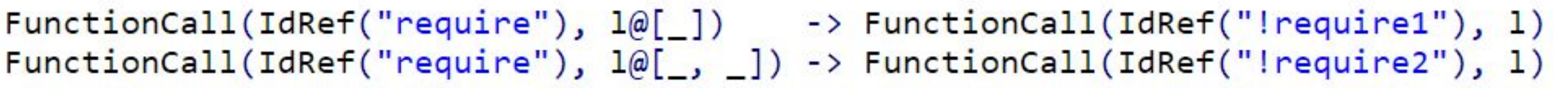
- Require and revert
- Used often
- Overloaded
    - Unsupported



```
pragma solidity ^0.4.24;
import "./lib.sol";

contract ReqRev {

    function assertEqual(int i, int j) {
        require(i == j, "i and j must be equal!");
    }

    function assertLt(int i, int j) {
        require(i < j);
    }

}
```

Builtin Functions

- Require and revert
- Used often
- Overloaded
  - Unsupported



```
FunctionCall(IdRef("require"), l@[_])    -> FunctionCall(IdRef("!require1"), l)
FunctionCall(IdRef("require"), l@[_, _]) -> FunctionCall(IdRef("!require2"), l)
```

Syntax coverage

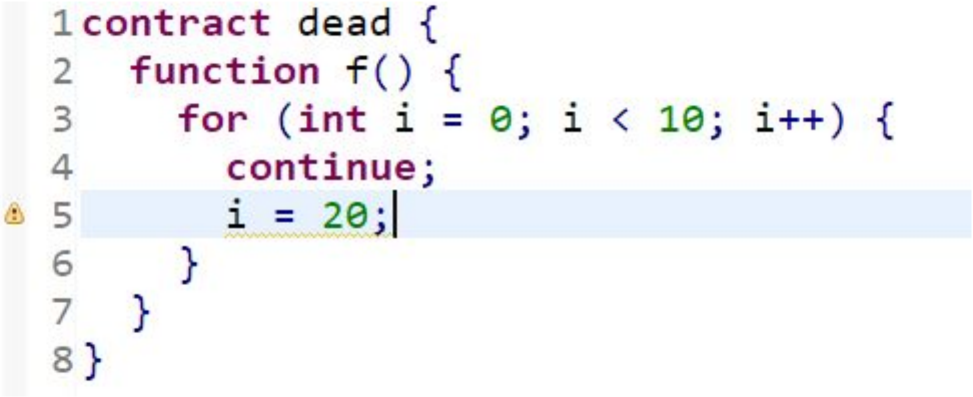
- All the syntax...
- Except Inline assembly
  - Parsed as set of strings

Type checking coverage

- Almost all of the type checking
- Except
  - Using … for … statements
  - Inline assembly
  - Visibility rules
  - Location rules
- And it is slightly less strict that the official compiler here and there

Optimization: Dead code

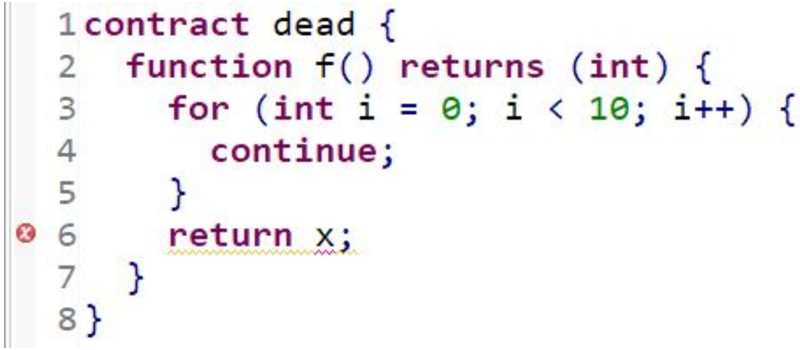
- Code after continue, break, return, etc.



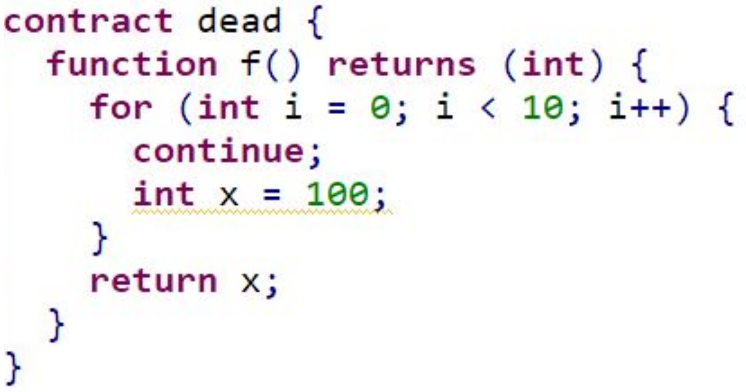
```
contract dead {
  function f() {
    for (int i = 0; i < 10; i++) {
      continue;
      i = 20;
    }
  }
}
```

Optimization: Dead code

- But variable declarations must be kept



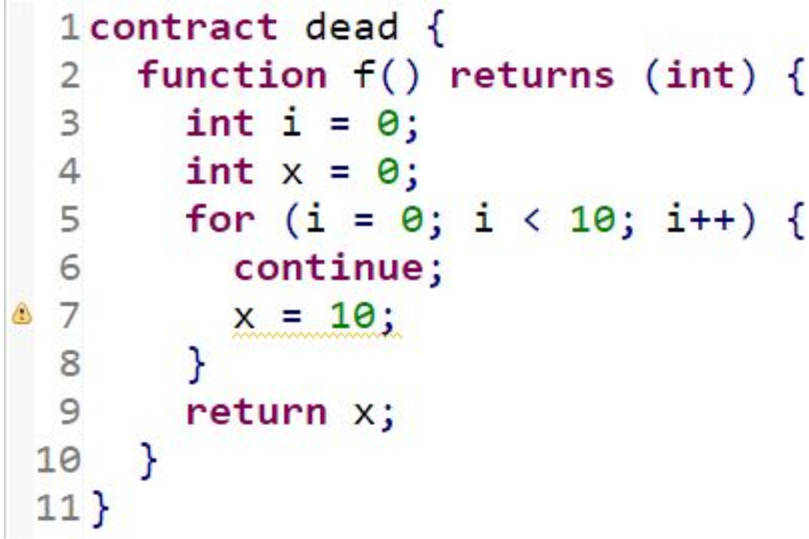
```
contract dead {
  function f() returns (int) {
    for (int i = 0; i < 10; i++) {
      continue;
    }
    return x;
  }
}
```



```
contract dead {
  function f() returns (int) {
    for (int i = 0; i < 10; i++) {
      continue;
      int x = 100;
    }
    return x;
  }
}
```

Optimization: Dead code

- “JavaScript Scoping”
- Use before declare is default value (0)



```
contract dead {
  function f() returns (int) {
    int i = 0;
    int x = 0;
    for (i = 0; i < 10; i++) {
      continue;
      x = 10;
    }
    return x;
  }
}
```

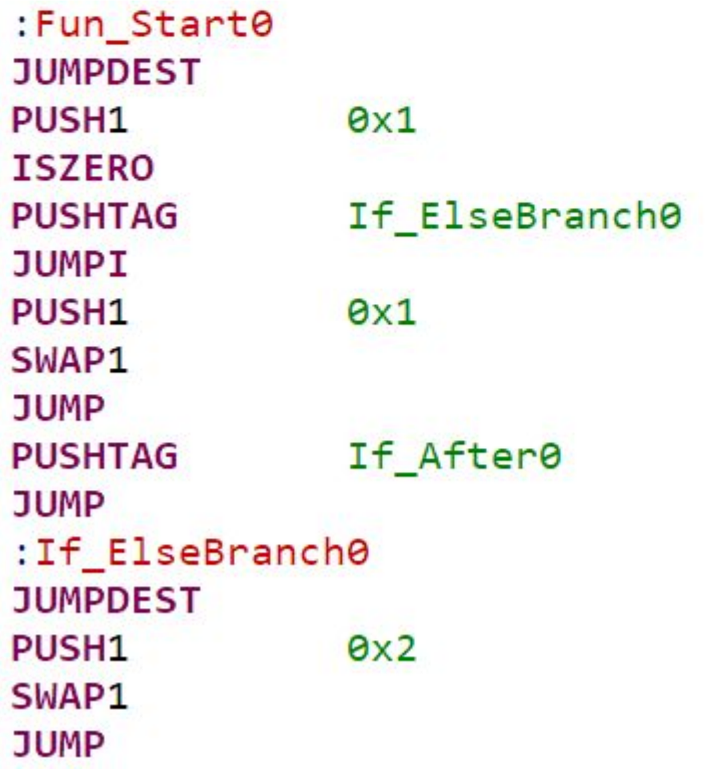
Ethereum

The Ethereum Virtual Machine

- Bytecode
- Pure stack machine
  - No local variables
- Expensive memory
- Expensive storage

The Ethereum Virtual Machine

- Second language: EBC
- Just Bytecode + Tags

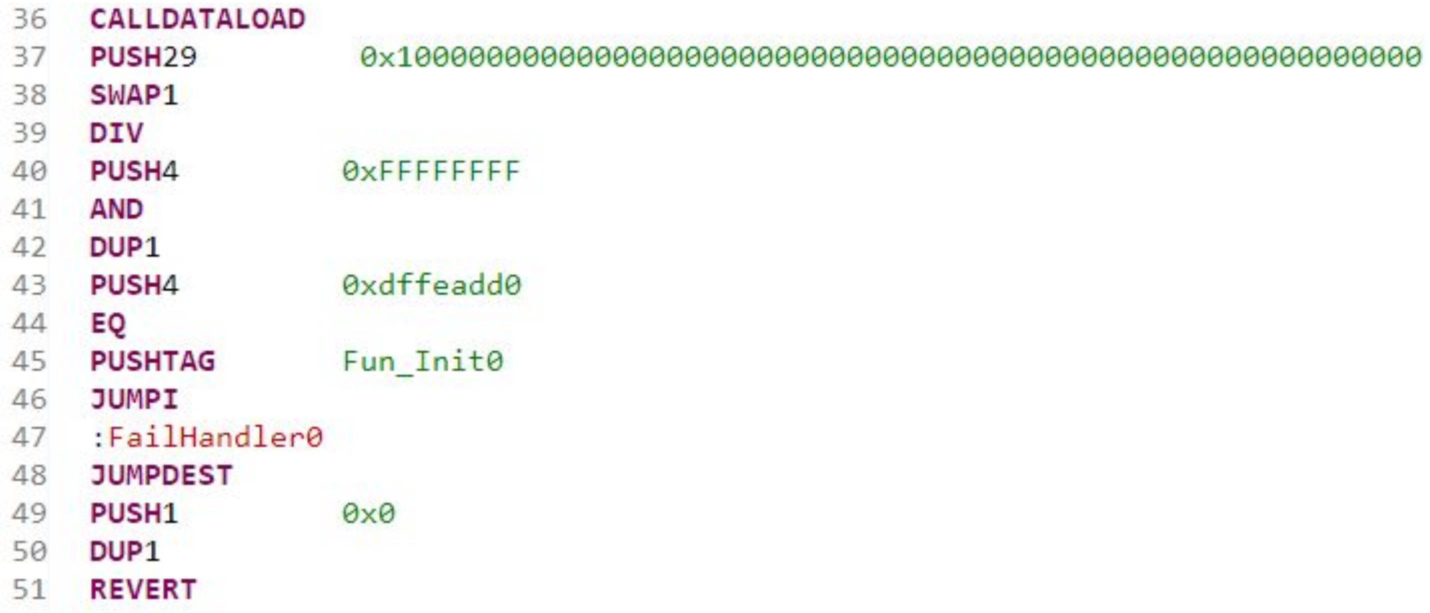


```
:Fun_Start0
JUMPDEST
PUSH1           0x1
ISZERO
PUSHTAG         If_ElseBranch0
JUMPI
PUSH1           0x1
SWAP1
JUMP
PUSHTAG         If_After0
JUMP
:If_ElseBranch0
JUMPDEST
PUSH1           0x2
SWAP1
JUMP
```

The EVM: Function calls

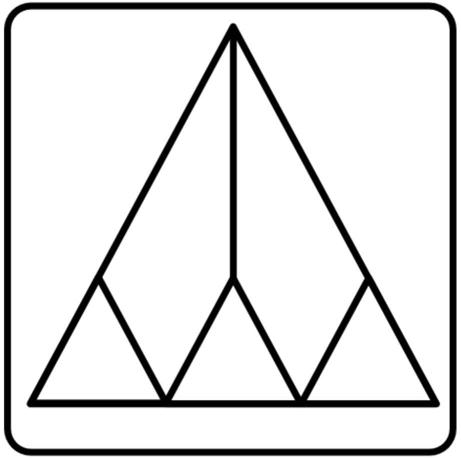
- No actual function calls
- ABI specification
  - JSON file with function signatures
- Hash of function signature is passed to contract
  - Contract looks up function signature
  - Jumps to starting location if it exists
  - Fail otherwise

The EVM: Function calls



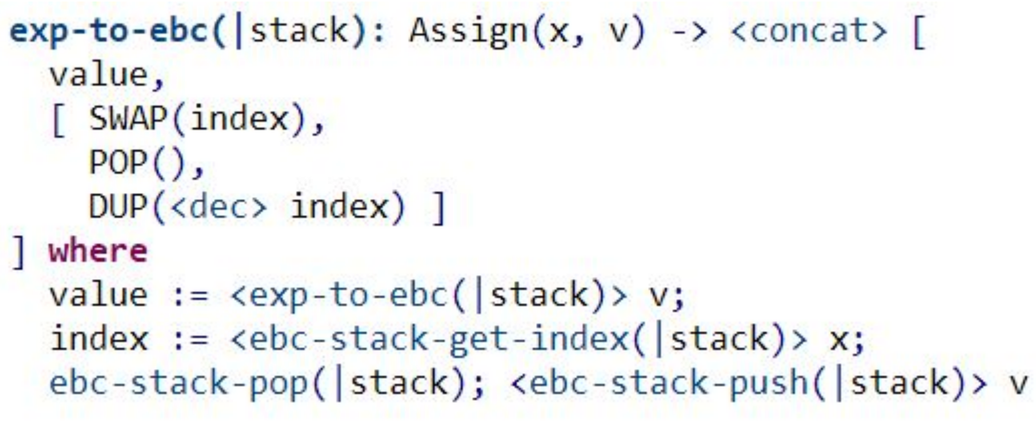
```
CALLDATALOAD
PUSH29          0x100000000000000000000000000000000000000000000000000000000
SWAP1
DIV
PUSH4          0xFFFFFFFF
AND
DUP1
PUSH4          0xdffeadd0
EQ
PUSHTAG        Fun_Init0
JUMPI
:FailHandler0
JUMPDEST
PUSH1          0x0
DUP1
REVERT
```

EBC



EBC: Local variables

- Assign each local variable a spot on the stack
- Keep track of the stack state
- Assign = SWAP(n)



```
exp-to-ebc(|stack): Assign(x, v) -> <concat> [
  value,
  [ SWAP(index),
    POP(),
    DUP(<dec> index) ]
] where
  value := <exp-to-ebc(|stack)> v;
  index := <ebc-stack-get-index(|stack)> x;
  ebc-stack-pop(|stack); <ebc-stack-push(|stack)> v
```

EBC: Local variables

- Assign each local variable a spot on the stack
- Keep track of the stack state
- Assign = SWAP(n)
- Reference = DUP(n)
- But
  - SWAP and DUP go from 1 to 16
  - n > 16?

EBC: Local variables

- Some valid programs are not compilable
  - Official compiler rejects them
  - My compiler generates invalid instructions
    - SWAP(20)
  - Optimization after might remove some items
  - Potential for moving items around

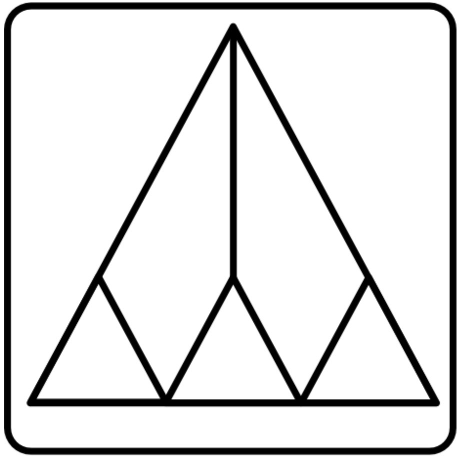
EBC: Return

- Stack must be emptied
- return a;
    - POP until a on top
    - SWAP a down the stack
    - Repeat until only a is on the stack

EBC: Continue and Break

- Continue = jump to before condition
- Break = jump after loop
- Compiler keeps track of continue and break jump locations.

{S} spoofax



Limitations

- Compilation really difficult
  - Not a lot of documentation
  - C++ code
  - Bytecode generated changes significantly for minor code changes
- So
  - Compiler itself only supports very basic programs

Limitations

- No fields, only local variables
- Only one function compiles correctly
- Parameters are not loaded
- Numbers are not sanitized correctly
  - The higher order bits must be cleared sometimes, as they could contain nonsense and mess up calculations.
- Many more

Technical Issues

- NaBL2 pretty printer
  - Crashes completely for the simplest of rules.
- NaBL2 max file length limit
  - Timeout for parsing due to number of subtyping relations
- Syntax
  - Priorities not always respected
  - Bracket rule conflicts with tuples
  - Keyword rejection with many keywords (int8)

Demo

TUDelft

Solidity

Solidity to Ethereum Bytecode in Spoofax

Taico Aerts