Writing a Modern Compiler

Abe Pralle

May 3, 2017

About Me

Abe Pralle

- Indie game developer (Runegate, Plasmaworks)
- Rogue language designer
- github.com/AbePralle/Rogue

Programming Interests

- Games, languages, API's

Contact

- abe.pralle@gmail.com

Overview

Modern Compiler

- Embraces OO design patterns and infrastructure
- Expects modern computing power - RAM, stack size, processor speed, storage space
- Breaks free from legacy dependance on Lex & Yacc (Flex & Bison)

Paradigm

- Implementation details geared towards imperative OO general-purpose language (Rogue/C#/Java)
- Examples given in Rogue

Target Audience

- Experienced software developers

Disclaimer

- Terminology varies
- This is just one way to do things

Overview

Topics

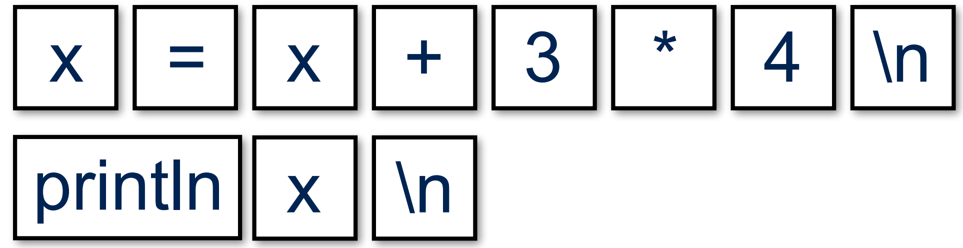
- Compiler Pipeline
- Syntax Specification
- Tokenization
- Basic Parsing
- Basic Semantic Analysis
- Code Generation
- Adding Fixed Types
- Adding Scope
- Adding Global Functions
- Adding Custom Classes
- Virtual Machines
- Saving the Code Tree
- Interpreters
- Miscellaneous
- Tombstone Diagrams
- Bootstrapping

Compiler Pipeline

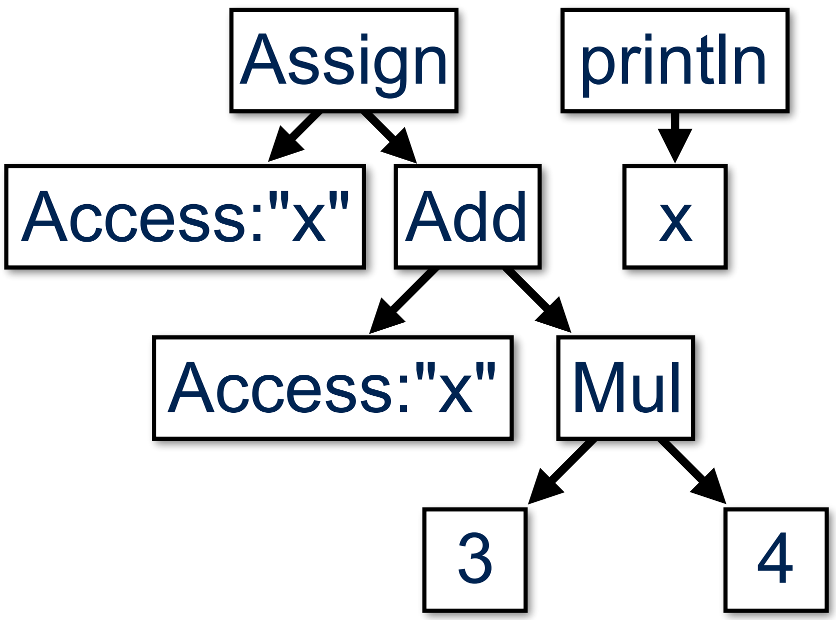
Source Code

```
x = x + 3 * 4  # do stuff
println x
```

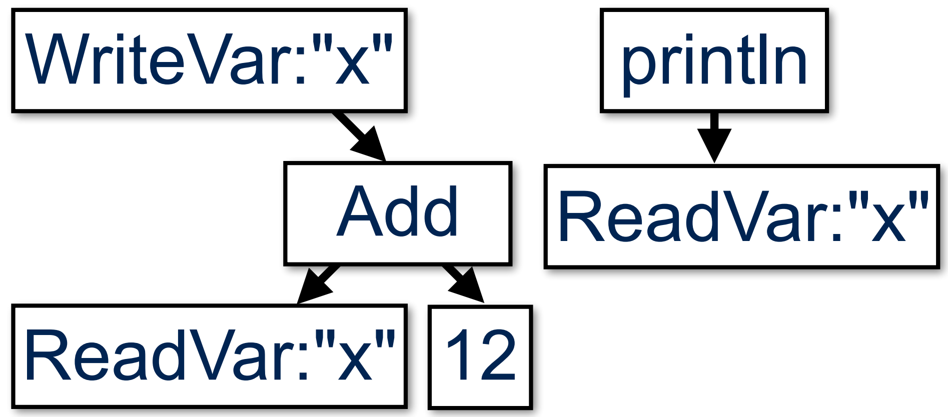
Tokenization



Parsing



Analysis



Code Generation

02 00 01 0C 05 02 00 04 00

Syntax Specification

EBNF (and similar)

```
program    = statements EOF | EOF
statements = statement statements
           | statement
statement  = 'println' expression
           | expression '=' expression
           | expression
expression = add_or_sub
add_or_sub = add_or_sub '+' mul_or_div
           | add_or_sub '|' mul_or_div
           | mul_or_div
mul_or_div = mul_or_div '*' term
           | mul_or_div '/' term
           | term
term       = '(' expression ')'
           | identifier | integer
```

Tokenization

Overview

- Better to roll your own than use Lex/Flex - easier, more flexible, more powerful, no dependency
- Tokenizer().tokenize("Filename.xyz")->Token[]

Components

- A **Scanner** reads characters from the source file
- A **Tokenizer** parses tokens using the scanner
- A **Token** contains a *type* (TokenType) reference, source position (filepath, line, and column), and *value* (when used to encapsulate literals such as Int32 and String).
- Global/singleton **TokenType** objects describe token types: name, attributes, etc.

Tokenization

Scanner

PROPERTIES

filepath : String

line, column : Int32

METHODS

has_another()->Logical

peek(lookahead=0:Int32)->Character

read()->Character

consume(ch:Character)->Logical

consume_whitespace()->Logical

consume_eols()->Logical

must_consume(ch:Character)

Scanner

- Load entire file as string
- Convert tabs to spaces etc.
- Rogue has a Scanner in the standard library

Tokenization

Token

GLOBAL PROPERTIES

next_filepath : String

next_line, next_column : Int32

PROPERTIES

type : TokenType

filepath : String

line, column : Int32

Token

METHODS

init(type)

error(mesg:String)->CompileError

is_structural()->Logical

name()->String

to->Int32

to->Real64

to->String

Int32Token : Token

PROPERTIES

value : Int32

Tokenization

TokenType

GLOBAL PROPERTIES

```
keywords = Table<<String,TokenType>>()
EOF=TokenType("end of file",&structural)
EOL=TokenType( "end of line" )
IDENTIFIER=TokenType( "identifier" )
PRINTLN=KeywordTokenType( "println" )
SYMBOL_PLUS=TokenType( "+" )
SYMBOL_OPEN_PAREN=TokenType( "(" )
SYMBOL_CLOSE_PAREN=TT( ")", &structural )
```

PROPERTIES

```
name : String
is_structural : Logical
```

KeywordToken : Token

```
method init( name, &structural )
  prior.init( name, structural )
  TokenType.keywords[ name ] = this
```

Tokenization

Tokenizer

PROPERTIES

scanner : Scanner

tokens : Token[]

METHODS

tokenize(filepath:String)->Token[]

consume(ch:Character)->Logical

tokenize_another->Logical

tokenize_identifier()->Logical

tokenize_integer()->Logical

tokenize_string()->Logical

scan_symbol_type()->TokenType

Tokenizer

```
method tokenize(...)->Token[]
  ... while (tokenize_another) noAction
  tokens.add( Token(TokenType.eof) )

method tokenize_another->Logical
  scanner.consume_whitespace
  if (not scanner.has_another) return false
  Token.next_line = scanner.line; ...
  if (tok_id or tok_int or tok_str) return true
  tokens.add( Token(scan_symbol_type) )

method tokenize_identifier()->Logical
  ... if (TokenType.keywords.contains(st))
        tokens.add( Token(TT.keywords[st]) )
        ...
method scan_symbol_type->TokenType
  if (consume('\n')) return TokenType.EOL
  elseIf (consume('+')) return TT.SYM_PLS
  else throw CompileError( "Syntax error" )
```

Basic Parsing

Overview

- Better to roll your own than use Yacc/Bison - easier, more flexible, more powerful, no dependency
- Parser("Filename.xyz").parse_expression()->Cmd
- Syntax vs semantics - Parser deals in syntax and handles syntax errors, semantic analysis will come later
  - "x = 3 a" is a syntax error
  - "x = 3 * a" is syntactically valid but would be a semantic error if either *x* or *a* is undefined

Components

- A **Cmd** is an object that models a program statement, operation, directive, etc.
- A **Parser** parses input tokens and creates trees of Cmd nodes - these trees are called **code trees, parse trees,** or **Abstract Syntax Trees (AST)**.

Basic Parsing

Cmd

- Cmd nodes are

```
class Cmd( t:Token );
class CmdStatements( t, list=Cmd[]:Cmd[] ) : Cmd;

class CmdPrintln( t, expression:Cmd ) : Cmd;
class CmdAccess( t, name:String ) : Cmd;
class CmdInt32( t, value:Int32 );

class CmdAssign( t, operand:Cmd, new_value:Cmd ) : Cmd;

class CmdBinaryOp( t, lhs:Cmd, rhs:Cmd ) : Cmd;
class CmdAdd : CmdBinaryOp;
class CmdSubtract : CmdBinaryOp;
class CmdMultiply : CmdBinaryOp;
class CmdDivide : CmdBinaryOp;
```

- For parsing purposes Cmd objects are simple nodes that are linked by the parser into a code tree
- Each Cmd references its corresponding Token

Basic Parsing

Parser

PROPERTIES

tokens : Token[]

position : Int32

METHODS

init(filepath:String)

consume(type:TokenType)->Logical

consume_eols->Logical

must_consume(type:TokenType)

peek/read()->TokenType

parse_statements(list:CmdStatements)

parse_statement(list:CmdStmts)->Logical

parse_expression()->Cmd

parse_add_or_sub(lhs:Cmd)->Cmd

...

Statement Parsing

```
method parse_stmts(list:CmdStatements)
  # statements = statement statements | stmt
  while (parse_statement(list)) noAction

method parse_stmt( list:CmdStmts )->Logical
  # statement  = 'println' expression
  #                      | expression '=' expression
  #                      | expression
  consume_eols
  local t = peek; if (t.is_structural) return false
  if (consume(TokenType.PRINTLN))
    list.add( CmdPrintln(t, parse_expression) )
  else
    local expr = parse_expression
    if (next_is(TokenType.SYMBOL_EQUALS))
      list.add(CmdAssign(read,expr,parse_expr))
    else
      list.add( expr )
    endIf
  endIf
  return true
```

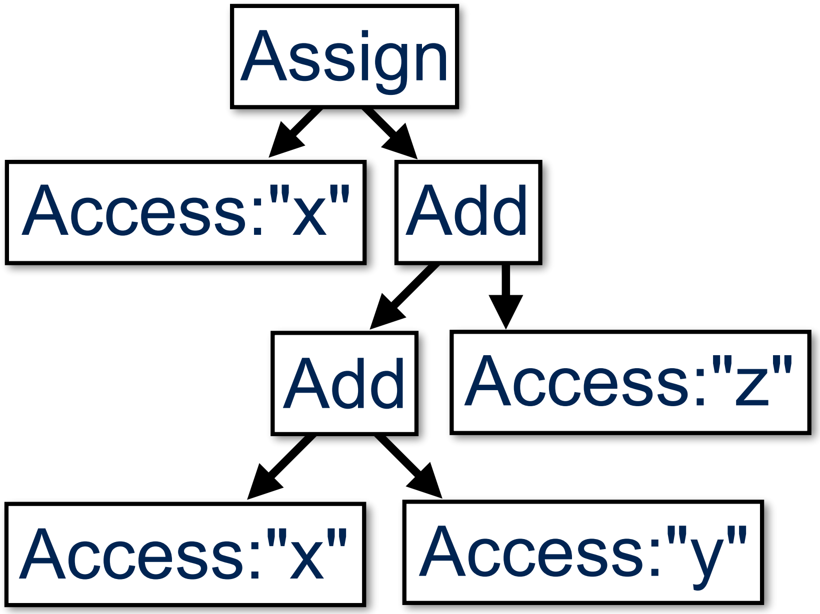
Basic Parsing

Expression Parsing

- Need to solve left recursion in EBNF

```
addsub = addsub '+' muldiv
       | addsub '|' muldiv
       | muldiv
```

- Note left recursion becomes left child of expression tree: x = x + y + z



Expression Parsing

```
method parse_expression->Cmd
  # expression = add_or_sub
  return parse_addsub

method parse_addsub->Cmd
  return parse_addsub( parse_muldiv )

method parse_addsub( lhs:Cmd )->Cmd
  local t = peek
  if (consume(TokenType.SYM_PLUS))
    return parse_addsub(
        CmdAdd( t, lhs, parse_muldiv ) )
  elseIf (consume(TokenType.SYM_MNS))
    return parse_addsub(
        CmdAdd( t, lhs, parse_muldiv ) )
  else
    return lhs
  endIf

...
```

Basic Parsing

Expression Parsing

```
method parse_term
  # term = '(' expression ')'
  #         | identifier | integer
  local t = peek
  if (consume(TokenType.SYMBOL_OPEN_PAREN))
    local result = parse_expression
    must_consume( TokenType.SYMBOL_CLOSE_PAREN )
    return result

  elseIf (consume(TokenType.IDENTIFIER))
    return CmdIdentifier( t, t->String )

  elseIf (consume(TokenType.INT32))
    return CmdInt32( t, t->Int32 )

  else
    throw t.error(
      "Syntax error: unexpected $." (t) )

  endIf
```

Basic Semantic Analysis

Requirements

- (Statement-level)
- Validate each command
- Inject typecasts
- Convert generic Cmd nodes to more specialized types
- Ensure the CmdAccess on a left-hand side of an assignment becomes a *write* instead of a *read*
- Perform *constant folding* to collapse literal expressions
- Infer types
- Expand macros
- Remove dead code ("if false")

Implementation

- Each Cmd gets resolved()->Cmd and some get resolved_assignment(new_value:Cmd)->Cmd methods
- A command resolves itself *as normal* or *as an assignment target* and returns either itself or a different resolved Cmd node
- The code tree rebuilds itself as it resolves
- A **Program** singleton manages program elements, including global vars

Basic Semantic Analysis

Program [singleton]

PROPERTIES

statements : CmdStatements

globals = Table<<String,Int32>>()

METHODS

compile(filepath:String)->Logical

global_index(name:String)->Int32

resolve()

write(writer:CPPWriter)

Source Code

```
method compile( filepath:String )->Logical
  try
    statements = Parser(filepath).parse_stmts
    resolve
    write( CPPWriter("Program.cpp") )
    return true
  catch (err:CompileError)
    println err
    return false
  endTry

method global_index( name:String )->Int32
  if (not g.contains(name)) g[name] = g.count
  return globals[ name ]

method resolve
  statements.resolve

method write( writer:CPPWriter )
  # TODO
```

Basic Semantic Analysis

Resolved() Methods

```
# Cmd
method resolved->Cmd
  throw t.error( "[undefined resolved]" )
method resolved_assignment(
    new_value:Cmd )->Cmd
  throw t.error("Invalid assignment target.")

# CmdInt32
method resolved->Cmd: return this

# CmdAdd
method resolved->Cmd
  lhs = lhs.resolved
  rhs = rhs.resolved
  if (lhs instanceOf CmdInt32 and
      rhs instanceOf CmdInt32)
    return CmdInt32( t,
        lhs->Int32 + rhs->Int32).resolved
  else
    return this
  endIf
```

Resolved() Methods

```
# CmdAccess
method resolved->Cmd
  # TODO: verify variable declared etc.
  # if required by language design
  return CmdReadVar( t, name ).resolved

method resolve_assignment(
    new_value:Cmd )->Cmd
  return CmdWriteVar( t, name,
      new_value ).resolved

# CmdPrintln
method resolved->Cmd
  # Currently assuming every expression
  # produces a (non-nil/non-void) value
  expression = expression.resolved
  return this
```

Basic Semantic Analysis

Resolved() Methods

```
class CmdReadVar : Cmd
  PROPERTIES
    name : String
    index : Int32

  METHODS
    method init( t, name )
      index = Program.global_index( name )

    method resolved->Cmd
      return this
endClass
```

Resolved() Methods

```
# CmdAssign
method resolved->Cmd
  return operand.resolve_assignment(
    new_value )
```

Top of the Mountain

Consider This

- At this stage of the compile process we are on top of the mountain
- We have a syntactically valid, semantically meaningful program in its purest, most abstract, most versatile form
- It's all downhill from here!

Code Generation

Overview

- Same general approach for writing out assembly, machine language, bytecode, C++, etc.
- Using C++ (CPP) as example
- Create CPPWriter class tailored to generating C++ source
- Similar to *resolved()*, add a *write(writer:CPPWriter)* method to the Program and every Cmd

Implementation

```
# Program
method write( writer:CPPWriter )
  writer.println ...
  @|#include <cstdio>
      |using namespace std;
      |int main()
      |{
  forEach (name in globals.keys)
    writer.println "  int $ = 0;"(name)
  endForEach
  statements.write( writer )
  writer.println "return 0;"
  writer.println "}"
  writer.close
```

Code Generation

Implementation

```
# CmdStatements
method write( writer:CPPWriter )
  forEach (cmd in list)
    writer.print( "  " )
    cmd.write( writer )
    writer.println( ";" )
  endForEach

# CmdAdd
method write( writer:CPPWriter )
  lhs.write( writer )
  writer.print( " + " )
  rhs.write( writer )
```

Implementation

```
# CmdWriteVar
method write( writer:CPPWriter )
  writer.print( name )
  writer.print( " = " )
  new_value.print( writer )

# CmdPrintln
method write( writer:CPPWriter )
  writer.print( 'printf( "%d\n", ' )
  expression.write( writer )
  writer.print( " )" )

# CmdInt32
method write( writer:CPPWriter )
  writer.print( ""+value )
```

Code Generation

Sample Source

```
x = 5
m = 4
y = m * x + 2 + 1
println x
println y
```

Sample Output

```
#include <cstdio>
using namespace std;

int main()
{
  int x = 0;
  int m = 0;
  int y = 0;
  x = 5;
  m = 3;
  y = m * x + 3;
  printf( "%d\n", x );
  printf( "%d\n", y );
  return 0;
}
```

Adding Fixed Types

Type

PROPERTIES

name : String

Program [singleton]

PROPERTIES

type_Int32 = Type("Int32")

type_Real64 = Type("Real64")

Cmd

METHODS

method cast_to(to_type:Type)->Cmd

method common_type(c:Cmd)->Type

method require_type->Type

method require_value->this

method type->Type: return null

Source Code

```
# Cmd
method cast_to( to_type:Type )->Cmd
  if (require_type is to_type) return this
  return CmdCast( t, this, to_type )

method common_type( other:Cmd )->Type
  local t1 = require_type
  local t2 = other.require_type
  if (t1 is Program.type_Real64 or t2 is R64)
    return Program.type_Real64
  elseIf (...)
  else
    throw t.error( "Types are incompatible." )
  endIf

method require_value->this
  if (type is null) throw t.error( "Value expec." )
  return this
```

Adding Fixed Types

Source Code

```
class CmdCast( t, operand:Cmd,
    to_type:Cmd )
  METHODS
    method write( writer:CPPWriter )
      writer.print( "(($)" (to_type.name) )
      operand.write( writer )
      writer.print( ")" )

# CmdAdd
method resolve->Cmd
  lhs = lhs.resolved
  rhs = rhs.resolved
  local ct = lhs.common_type( rhs )
  lhs = lhs.cast_to( ct )
  rhs = rhs.cast_to( ct )
  return this
```

Adding Scope

Overview

- Create Scope class which maintains list of active local variables

- A scope object is now passed to Cmd.resolved():

  method resolved(scope:Scope)->Cmd

- CmdStatements marks the local variable "stack" position at the beginning of *resolved()* and resets to that position later

- Local variable declarations add themselves to the scope's list of locals

Adding Global Functions

Overview

- Program maintains list of functions that have been parsed; each function has parameter types and return type
- CmdAccess gets a CmdArgs property; if parens are given then CmdArgs gets the list of comma-separated expressions inside the parens
- Scope gets a resolve_call(t, access, args) and is responsible for call resolution (finding best candidate, inserting default args, etc)

Adding Custom Classes

Overview

- *Program* manages custom type objects
- Parser parses types which parse properties and methods
- When parsing, can ask Program for a Type reference and it always returns one - empty shell if undefined
- When a type is parsed its "shell" is fetched and filled in
- Program *resolve()* calls *resolve()* on each Type, type throws an error if never defined

Object-Oriented

- Parser maintains *this_type* and *this_method*
- References to *this/self* get *this_type*
- Scope also maintains *this_type, this_method*
- Access names are looked for in scope.this_method.properties
- Method return values are cast to/checked against scope.this_method.return_type

Virtual Machines

Overview

- Virtual machines are simple to make
- A VM runs on byte code or "int code" invented machine language (or actual machine language)
- Simple VM has instance properties for program code, main memory, stack, call_stack, fp, and ip
- Execution is big switch inside infinite loop
- Stack-based operation model is simple and elegant

Virtual Machines

Stack-Based Operation Model

- Alternative to register-based
- Machine performs computations by pushing values onto system stack, executing ops that implicitly modify the top 1 or 2 stack items, and popping result

Register-Based Assembly

```
# x = -y * 3 + z
mov eax, y
neg eax
mul eax, 3
add eax, z
mov x, eax
```

Stack-Based Assembly

```
# x = -y * 3 + z
push y
neg
push 3
mul
push z
add
pop x
```

Virtual Machines

Compiling For VM

```
# CmdInt32
method write( writer:VMWriter )
  writer.write( Opcode.INT32 )
  writer.write( value )

# CmdAdd
method write( writer:VMWriter )
  lhs.write( writer )
  rhs.write( writer )
  writer.write( Opcode.ADD )
```

VM Source Code

```
void VM::execute()
{
  for (;;)
  {
    switch (code[ip++])
    {
      case OP_INT32:
        stack.add( code[ip++] );
        continue;
      case OP_ADD:
      {
        int a = stack.remove_last().int32;
        stack.add( a + stack.rm_last().int32 );
        continue;
      }
      ...
    }
  }
}
```

Interpreters

Overview

- Interpreters can either compile to VM code and then execute the VM or execute the Cmd nodes directly (or compile to actual assembly and execute)
- A good VM is about 10 times slower than machine language
- Executing Cmd methods is faster than VM, but a bit less flexible

Interpreter Source Code

```
# CmdInt32
method execute_Int32->Int32
  return value

# CmdAddIntReal64
method execute_Real64->Real64
  return lhs.exec_R64+rhs.exec_R64

# CmdPrintlnInt32
method execute
  println operand.execute_Int32

# CmdWhile
method execute
  while (condition.execute_Logical)
    try
      statements.execute
    catch (e:EscapeWhileException)
      return
    endTry
  endWhile
```

Saving the Code Tree

Overview

- The code tree (AST) can be easily serialized using the following approach. For each Cmd node:
  - Write an integer ID identifying the class
  - Write properties - use indices into global look-up tables for complex data
  - Recursively write children

Example

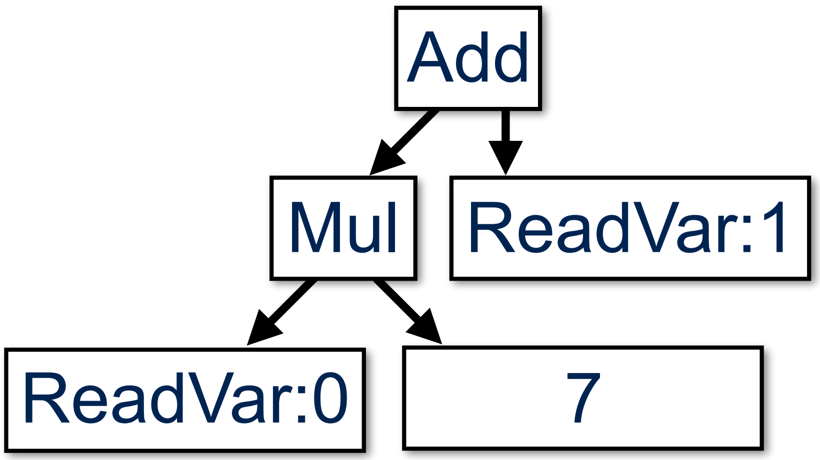
```
# CmdInt32
method write( writer:ASTWriter )
  writer.write( CmdCode.INT32 )
  writer.write( value )

# CmdAdd
method write( writer:ASTWriter )
  writer.write( CmdCode.ADD )
  lhs.write( writer )
  rhs.write( writer )

# CmdReadVar
method write( writer:ASTWriter )
  writer.write( CmdCode.READ_VAR )
  writer.write( index )
```

Saving the Code Tree

Example AST



CmdCode

ADD = 0

MUL = 1

READ_VAR = 2

INT32 = 3

Saved Tree Code

00 01 02 00 03 07 02 01

Loading Tree Code

```
# class ASTReader
method read_cmd->Cmd
  which (read_byte)
    case CmdCode.ADD
      return CmdAdd(this)

# CmdAdd
method init( reader:ASTR )
  lhs = reader.read_cmd
  rhs = reader.read_cmd
```

Miscellaneous

Strings

- Recommend always storing strings as UTF-8 per the UTF-8 Everywhere Manifesto: http://utf8everywhere.org/
- Each string can have internal cursor that makes sequential character access fast

Templates

- When "parsing" template, can store raw tokens and clone them later with a substitution table when template type is referenced
- Recommend storing all classes as templates

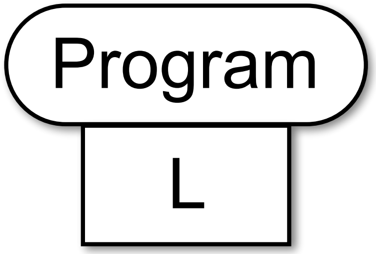
Tombstone Diagrams

About

- *Tombstone Diagrams* use Lego-style building blocks and are a way to visualize compiler bootstrapping as well as relationships between programs, machines, compilers, and VM's

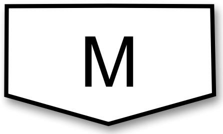
Source Code

- A program is expressed in some language L



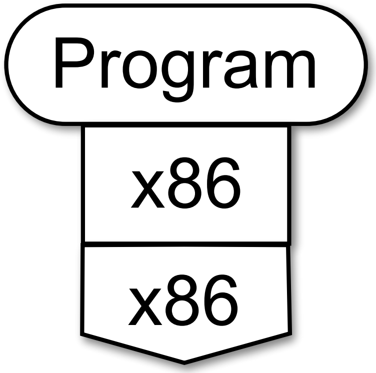
Machines

- Computers are machines that understand language M



Compiled Code

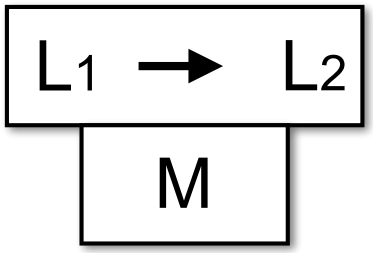
- A program expressed in *x86* machine language can run on an x86 machine



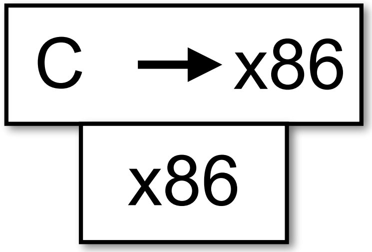
Tombstone Diagrams

Compilers

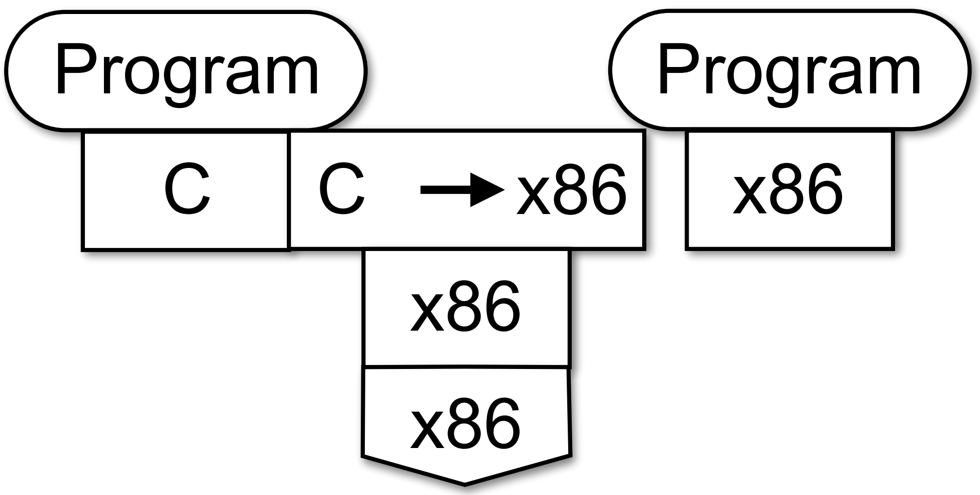
- Compilers are *language translators* and are themselves expressed in some language M



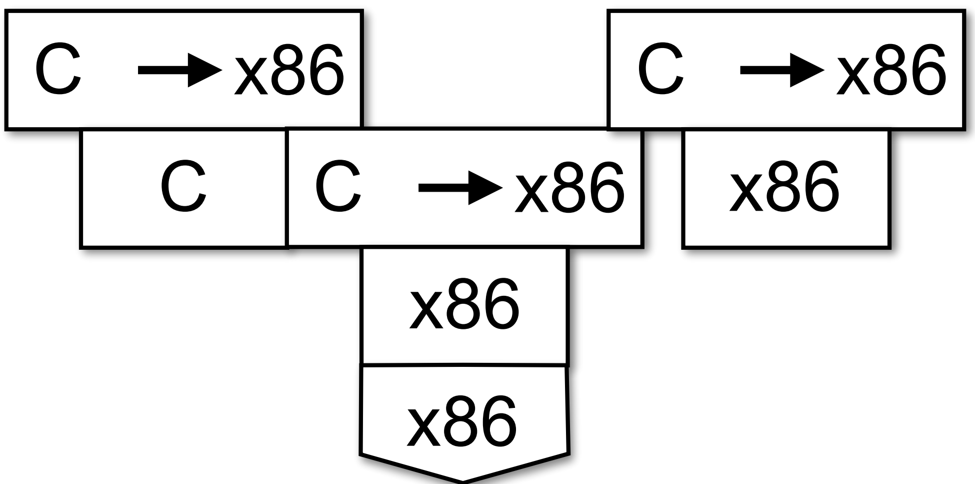
C to x86 Compiler



Compiling a Program



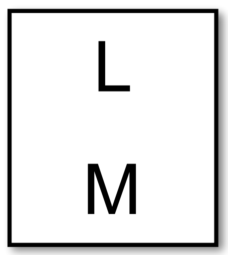
Compiling a Compiler



Tombstone Diagrams

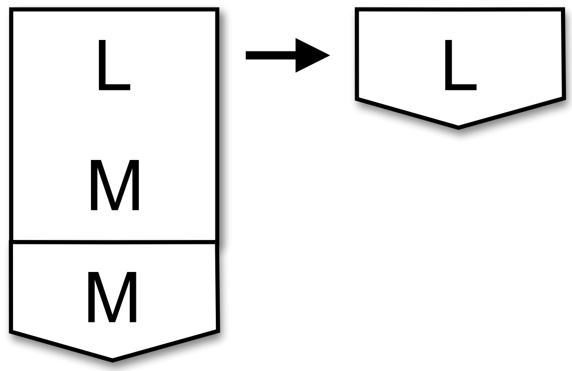
Virtual Machines

- VM's are expressed in one language M and host another language L

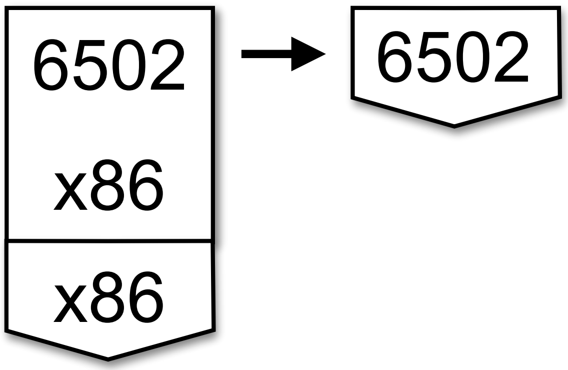


VM Abstraction

- A VM hosting language L is effectively an L machine

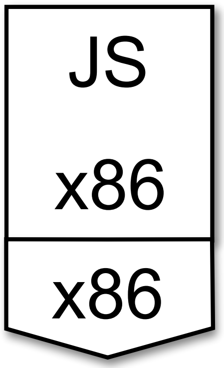


6502 VM



Interpreter

- An interpreter can be viewed as a VM that hosts source code rather than byte code



Bootstrapping a Language

Overview

- Write an minimal X compiler in existing language C (X1)
- Rewrite the minimal X1 compiler in X (X2)
- Write the full version of X using the X2 compiler (X3)