Little Languages in Rogue

Abe Pralle
August 1, 2018

Rogue Language Designers

Abe Pralle

- Created Rogue language in 2015
- Ad tech developer at AppOnboard
- Indie game developer (Runegate, Plasmaworks)

Programming Interests

- Games, languages, APIs

Contact

- abe.pralle@gmail.com

Install Rogue

- Repo with installation instructions: github.com/AbePralle/Rogue
- Windows, macOS, Linux

Rogue Language Designers

Murphy McCauley

- Frequent collaborator & consultant
- Joined Rogue project in 2016
- Ph.D. student at Berkeley (Computer Science, 2018)
- Programmed "SENSE" packet-level network simulator in Rogue and Python

Major Contributions to Rogue

Automatic garbage collection, multithreading, two Python extension generators, tuples, method template type inference by parameter types, core Windows compatibility, the *Rose* prototype language, and the book-in-progress "Hacking with Rogue"

ParseKit

Motivations

- "It's better to write your own tokenizer and parser than to use Lex and Yacc or Flex and Bison. What they do is fairly straightforward and it's easier and more flexible to just roll your own." --Abe Pralle, 2017

- "I'm tired of rewriting the same damn tokenizer and parser a dozen times for small language projects." --Abe Pralle, 2018

- I often want a little language as part of a larger project. Besides Flex and Bison not generating Rogue code, I want a parser that I can use as an accessory rather than as a standalone program.

ParseKit

Overview

- ParseKit is a Rogue Library for the rapid development of fast, efficient runtime parsers
- Not a compiler-compiler
- ParseKit is bundled with Rogue
- Parsers are defined with module template classes
- Any number of ParseKit parsers can be used concurrently
- Parsers can be add-ons to other projects or used to create standalone compilers
- ParseKit facilitates the creation of Scanner, Token, Tokenizer, Parser, Visitor, Error and Cmd (AST node) classes
- Calls to Parser.parse_x() return the root *Cmd* node of a fully structured AST
- Further analysis, interpretation, and/or output is up to you

ParseKit by Example: Simple

About

- SimpleParser will parse simple expressions
- Supports integers, identifiers, + - * / ( )

Step 1: *include* ParseKit, declare *uses*

- Edit "Simple.rogue" and start with this:

```
$include "ParseKit"
uses ParseKit<<Simple>>
```

- Creates these classes (among others) in the SimpleParseKit module/namespace:
  - ParseError
  - TokenType, Token, Tokenizer
  - Parser, Cmd
  - Visitor

Simple Error, TokenType

Step 2: Extend ParseError

- class SimpleError : ParseError
  endClass
- Add or override select methods later on as desired

Step 3: Augment TokenType enum with language tokens

```
augment TokenType
  EOI( "eoi", "[end of input]" )
  EOL( "\n",  "[end of line]" )
  IDENTIFIER( "identifier" )
  INTEGER( "integer" )
  PLUS( "+", &is_symbol )
  MINUS ( "-", &is_symbol )
  ASTERISK( "*", &is_symbol )
  SLASH( "/", &is_symbol )
  OPEN_PAREN(  "(", &is_symbol )
  CLOSE_PAREN( ")", &is_symbol, &is_structural )
endAugment
```

Simple Tokenizer

Step 4: Extend Tokenizer

- Base tokenizer automatically turns characters into tokens for symbols & keywords based on TokenType definitions
- Override *on_comment(String)*, *on_identifier(String)*, *on_integer(Int64)*, *on_real(Real64)*, and/or *on_string(String)* and in each case have any number of **tokens.add(Token(TokenType.X,value))** statements
- Can override other methods to change default tokenization of strings & comments etc.
- class SimpleTokenizer : Tokenizer

```
  METHODS
    method on_identifier( name:String )
      tokens.add( Token(TokenType.IDENTIFIER,name) )

    method on_integer( value:Int64 )
      tokens.add( Token(TokenType.INTEGER,value) )
endClass
```

Simple Cmd Nodes

Step 5: Extend Cmd (AST)

```
class CmdAdd : CmdBinary
  METHODS
    method op->String
      return "+"
endClass

class CmdSubtract [...]
class CmdMultiply [...]
class CmdDivide [...]

class CmdNegate : CmdUnary
  METHODS
    method op->String
      return "-"
endClass
```

Step 5 Code (continued)

```
class CmdAccess( t, name:String )
    : Cmd
  METHODS
    method init( t )
      name = t->String

    method to->String
      return name
endClass

class CmdLiteralInt32(t,value:Int32)
    : Cmd
  METHODS
    method init( t )
      value = t->Int32

    method to->String
      return ""+value
endClass
```

Simple Parser

Step 6: Extend Parser

- Add **ParseRule** objects to parser during initialization
- **ParseRule**, **BinaryParseRule**, and **UnaryParseRule** handle various parsing needs
- Multiple token handlers (productions) can be assigned to each parse rule
- Use **ParseRule** for arbitrary parsing
- **BinaryParseRule** and **UnaryParseRule** accept AST Cmd types and will create and link those nodes appropriately
- You can *add()* or *add_nested()* each ParseRule to the Parser. *add()* creates a new top-level rule and *add_nested()* hooks in the rule as a recursive descent target of the previously added rule
- Each nested parse rule parses at a higher precedence level

Simple Parser

Step 6 Code

```
class SimpleParser : Parser
  PROPERTIES
    parse_expression : ParseRule

  METHODS
    method init
      # expression
      local rule = add( ParseRule("expression") )

      # add_subtract
      rule = add_nested( BinaryParseRule("add_subtract") )
      rule.on( "+", <<CmdAdd>> )
      rule.on( "-", <<CmdSubtract>> )

      # multiply_divide
      rule = add_nested( BinaryParseRule("multiply_divide") )
      rule.on( "*", <<CmdMultiply>> )
      rule.on( "/", <<CmdDivide>> )
```

Simple Parser

Step 6 Code

```
    # negate
    rule = add_nested( UnaryParseRule("negate") )
    rule.on( "-", <<CmdNegate>> )

    # term
    rule = add_nested( ParseRule("term") )
    rule.on( "(",
      function (parser:SimpleParser)->Cmd
        parser.must_consume( TokenType.OPEN_PAREN )
        local result = parser.parse_expression()
        parser.must_consume( TokenType.CLOSE_PAREN )
        return result
      endFunction
    )
    rule.on( "identifier", <<CmdAccess>> )
    rule.on( "integer", <<CmdLiteralInt32>> )
endClass
```

Simple Test

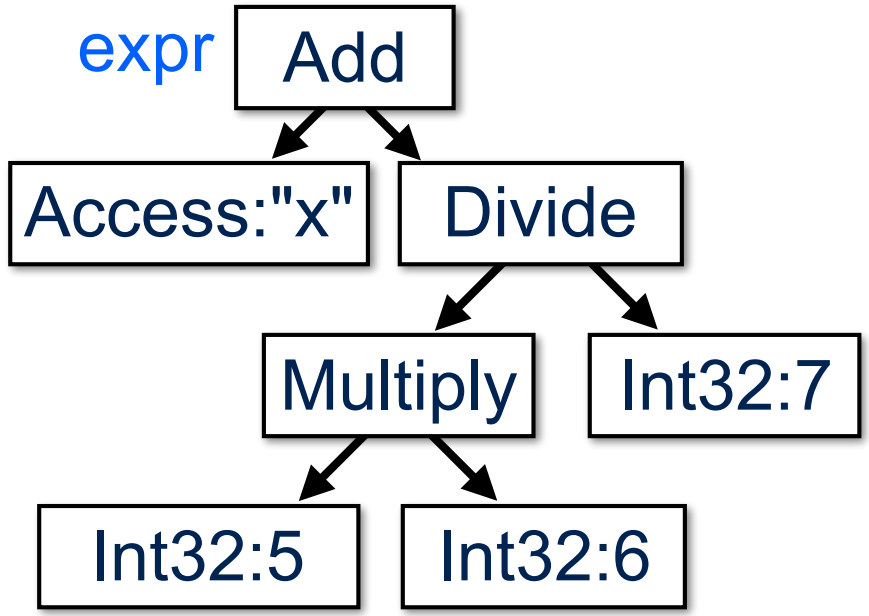
Test Driver

```
class SimpleTest
  METHODS
    method init
      local parser = SimpleParser()
      local args =
   System.command_line_arguments
      parser.set_source( "[Source]",
          args.join(" ") )
      parser.consume_eols
      while (parser.has_another)
        local expr =
            parser.parse_expression()
        <fold_constants>  # no-op label
        println expr
        parser.consume_eols
      endWhile
endClass
```

Output Log

```
> ./simple x+5
  (x + 5)
> ./simple x+5*6
  (x + (5 * 6))
> ./simple x+5*6/7
  (x + ((5 * 6) / 7))
```

AST



Constant Fold Visitor

Changes to Cmd Classes

```
augment Cmd
  METHODS
    method is_constant->Logical
      return false

    method to->Int32
      return 0
endAugment

augment CmdLiteralInt32
  METHODS
    method is_constant->Logical
      return true

    method to->Int32
      return value
endAugment
```

Extended Visitor

```
class ConstantFoldVisitor : Visitor
  METHODS
    method handle( cmd:CmdAdd )
      ->Cmd
      cmd.left  = visit( cmd.left )
      cmd.right = visit( cmd.right )
      if (cmd.left.is_constant ...
          and cmd.right.is_constant)
        return CmdLiteralInt32( cmd.t,
            cmd.left->Int32...
            + cmd.right->Int32 )
      else
        return cmd
      endIf

    method handle(cmd:CmdSubtract)
      ->Cmd ...
```

Constant Fold Visitor

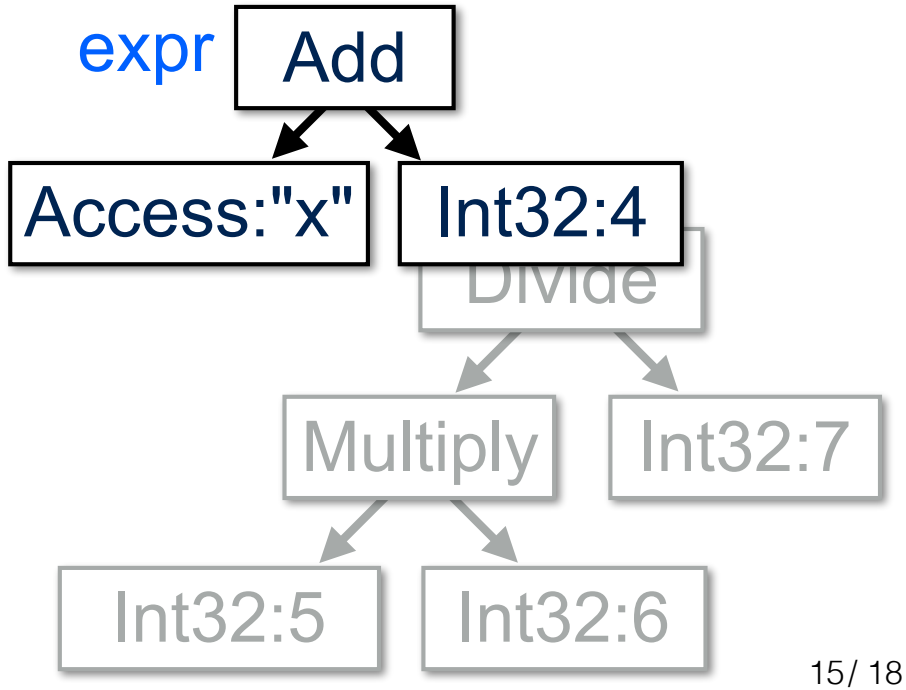
Augment SimpleTest

```
augment SimpleTest
  METHODS
    method init
      <insert_fold>
      local v = ConstantFoldVisitor()
      expr = v.visit( expr )
endAugment
```

Output Log

```
> ./simple x+5
  (x + 5)
> ./simple x+5*6
  (x + 30)
> ./simple x+5*6/7
  (x + 4)
```

AST



Visitor Details

About

- Base Visitor class visits all Cmd nodes in an AST
- Uses introspection to assist
- Calling visit(cmd) calls handle(cmd) with introspection assist
- Default handle(cmd) calls:
  - on_enter(cmd)
  - dispatch(cmd)
  - on_leave(cmd)
- Define any or all of those methods with specialized Cmd types as desired

LiteralCollectionVisitor

```
# Collect all literals Int32 values in
# a Simple AST
class LiteralCollectionVisitor: Visitor
  PROPERTIES
    int32s = Table<<Int32,Int32>>()

  METHODS
    method on_enter(
        cmd:CmdLiteralInt32 )
      int32s[ cmd.value ] = cmd.value
endClass


...
local v = LiteralCollectionVisitor()
println v.[ visit(expr) ].int32s
```

Misc: Built-In Cmd Nodes

Built-In Cmd Nodes

- Cmd(t:Token)
- CmdUnary(t, operand:Cmd) : Cmd
- CmdBinary(t, left:Cmd, right:Cmd) : Cmd
- CmdStatements() : Cmd[]
- CmdArgs() : Cmd[]

Notes

- CmdStatements and CmdArgs are properly handled in Visitor
- Must define method op->String for Unary & Binary
- Unary & Binary have default to->String

The End

Notes

- See Rogue/Demos/ParseKit/BitCalc for a more advanced example