Latka: A Language for Random Text Generation

Getty D. Ritter

POPL OBT — Jan 25, 2014

Background: Roguelikes

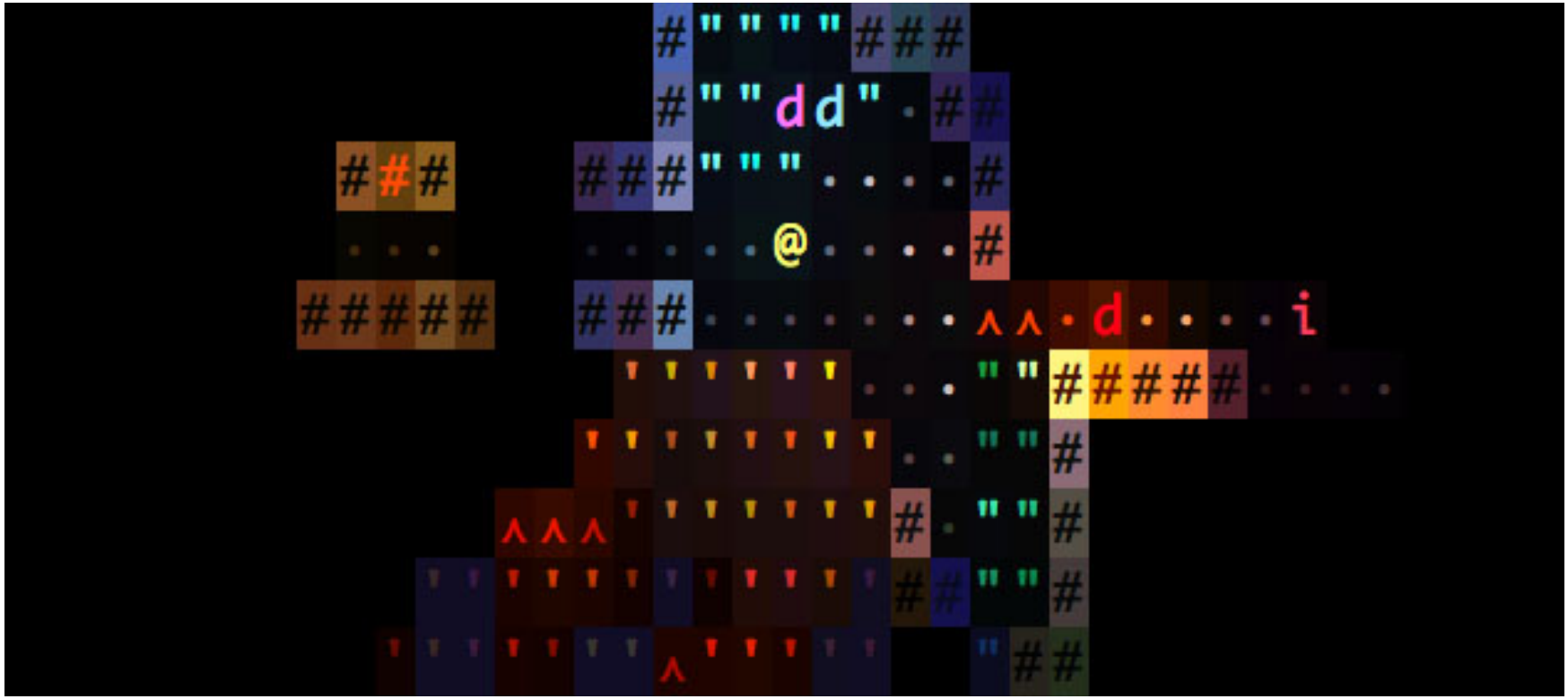


Figure : Brogue ©2012 Brian Walker

Background: Roguelikes

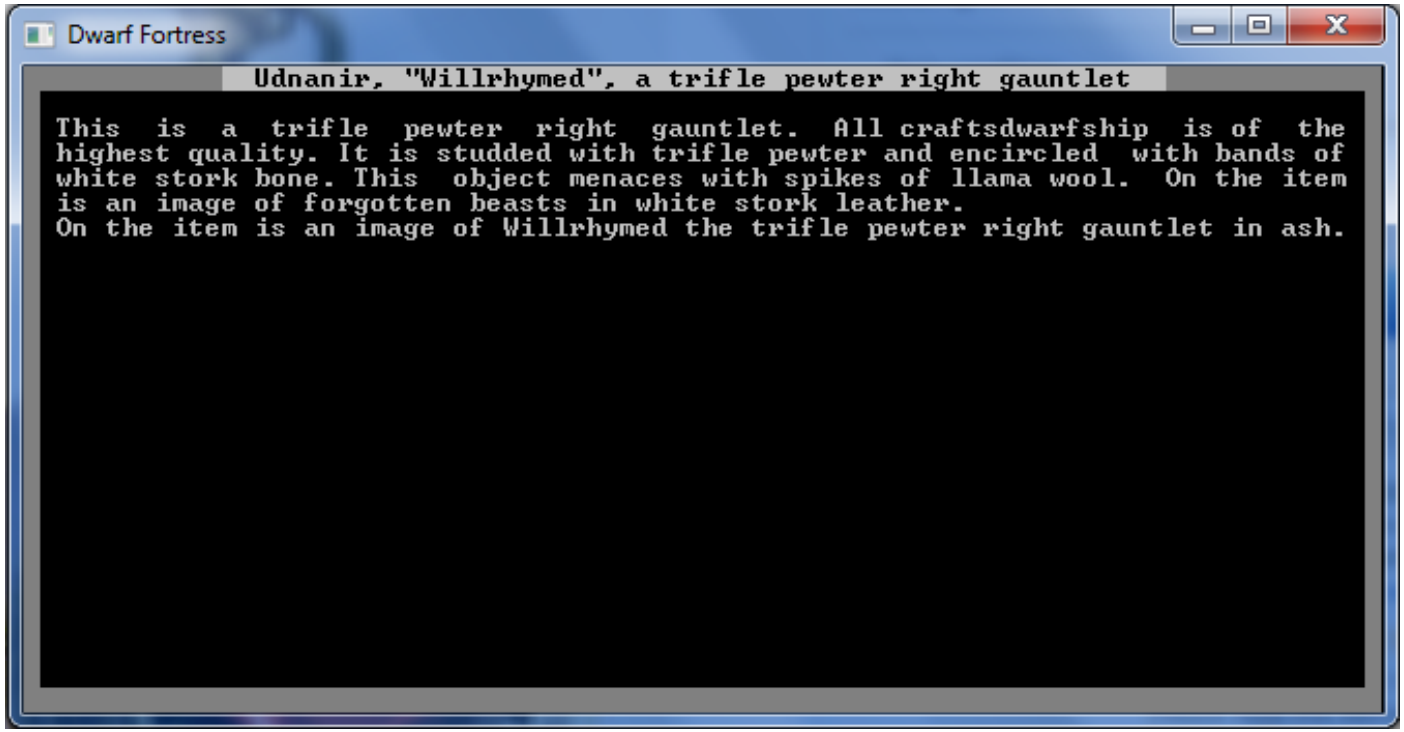
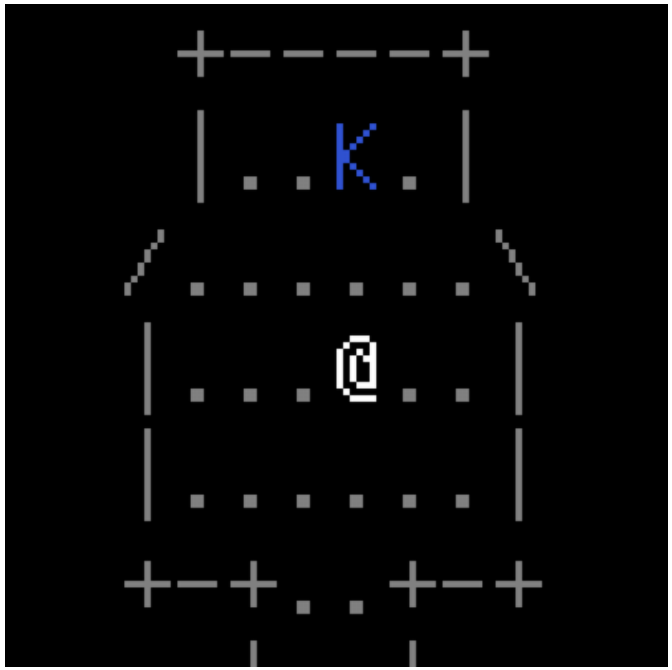


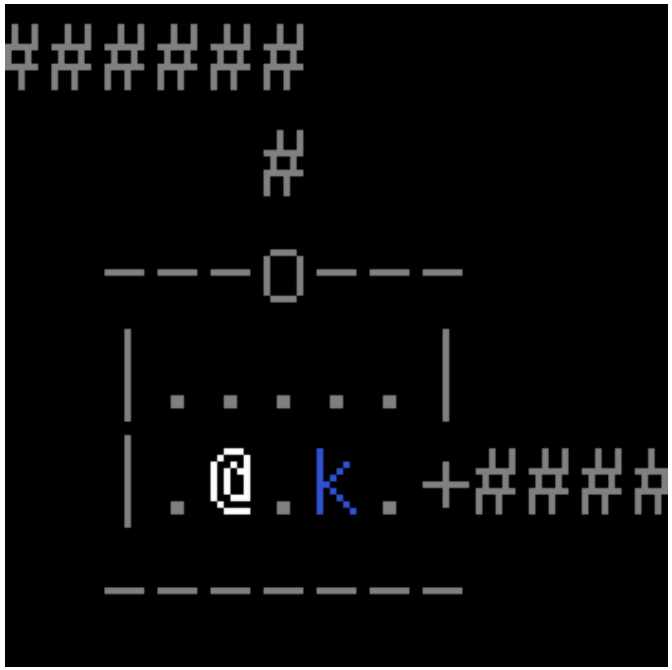
Figure : Dwarf Fortress ©2002-2012 Tarn Adams

Lord Tiahaki'ne



This is Lord Tiahaki'ne, an Elder of the Kute'aha. Its carapace is mottled and it wears fine jewelry of gold and pearls.

Hatchling Wipa'a



This is Wipa'a, a Hatchling of the Kute'aha. It has just been born to Peko, just one of 133 siblings.

Two Basic Problems

1. Concisely and reliably express a nondeterministic set of strings for human consumption.
2. Provide primitives appropriate to constructing text at a higher level than string concatenation.

Solution

Solution

Write an embedded DSL for an existing language. Don't reinvent the wheel.

```
description :: StringChoiceM ()
description = chooseRole >>= \case
                Lord -> do
                  name <- genName
                  appr <- genAppearance
                  ...
```

More Entertaining Solution

Because reinventing the wheel is fun: the **Latka** language. Embed probabalistic choice as an effect, and select primitive operations appropriate to the problem domain.

```
consonant = "p" | "t" | "k" | "w"
          | "h" | "m" | "n"
vowel = ("a" | "e" | "i" | "o" | "u") (4: "" | "'")
syllable = 4: consonant vowel | vowel
word = (4 | 5 | 6) @ syll

puts word

{- kinane, hupitu', wuteni, newai,
 - puwupo, kuikia, teipi', ituwoi,
 - kaekupu, ikukukiepa', henewu, topano
 -}
```

Latka Command Language

A single command, only available at the top-level:

```
puts "Hello, World!"
```

Latka Expression Language

Basic choice and string concatenation:

```
a = "this" "that" "whatever"
-- a evaluates to "thisthatwhatever"

b = "this" | "that" | "whatever"
-- b evaluates to "this" or "that" or "whatever"

c = 9: "this" | "that"
-- c evalutes to "this" nine times out of ten

d = (8 @ "na") " batman"
-- d evaluates to exactly what you'd expect
```

Latka Expression Language

. . . with a dash of generic-strongly-typed-functional-language:

```
data Maybe t = Just t | Nothing

foo : Maybe (String * String) -> String
foo x = case x of
          Just.p  -> "(" fst.p "," snd.p ")"
          Nothing -> "()"
```

The `f.x` syntax for function application is borrowed from Dijkstra.

```
f.x.y == (f.x).y
```

Latka Expression Language

Expressions are typically **call-by-name**

```
cbn = let x = "a" | "b"
        in x x
-- { "aa", "ab", "ba", "bb" }
```

But you can force **call-by-value** by using the `fixed` keyword

```
cbv = let fixed x = "a" | "b"
        in x x
-- { "aa", "bb" }
```

Latka Expression Language

This is necessary for long text descriptions:

```
let gender = Female | Male | Other in
  "A " noun.gender " stands nearby. "
    "You approach " objpronoun.gender "."
```

A man stands nearby. You approach him.

A person stands nearby. You approach them.

Latka Expression Language

This is necessary for long text descriptions:

```
let gender = Female | Male | Other in
  "A " noun.gender " stands nearby. "
    "You approach " objpronoun.gender "."
```

A man stands nearby. You approach her.

A woman stands nearby. You approach him.

Latka Expression Language

This is necessary for long text descriptions:

```
let fixed gender = Female | Male | Other in
  "A " noun.gender " stands nearby. "
    "You approach " objpronoun.gender "."
```

A woman stands nearby. You approach her.

A person stands nearby. You approach them.

Static Guarantees

- Strongly typed with type inference and optional type annotations.

Static Guarantees

- Strongly typed with type inference and optional type annotations.
- Latka also only allows structural recursion.

Static Guarantees

- Strongly typed with type inference and optional type annotations.
- Latka also only allows structural recursion.
- Consequently, Latka programs by construction cannot go wrong.

Latka Type System

Unremarkable, save a single experimental feature: `Var a` is a type of arbitrary-length tuples of types convertible to `a`. The only valid operation on `Var a` is

```
fgt : Var a -> List a
```

`Var a` can only appear in negative position in an arrow type.

Latka Type System

A lot of work for relatively little benefit:

```
listAll : Var String -> String
listAll v =
  let go.[]   = "nothing."
      go.[x] = x "."
      go.[x,y] = x " and " y "."
      go (x::xs) = x ", " go xs
  in go.(fgt.v)
```

Latka Type System

But still a small convenience:

```
puts listAll.()
-- "nothing."
puts listAll.(2,True,"blue")
-- "2, True and blue."
puts listAll.(5,\x -> x)
-- compile-time error, can't convert (a -> a) to String
```

Latka Standard Library

Still a work-in-progress, but crucially must provide intelligent primitives for constructing human language, e.g. for creating sentences, words, proper nouns, &c

```
puts se.( "one two"
        , wd."three"
        , pn."four"
        )
-- "One two three Four."
puts pa.( lst.( "one", "two" ) )
-- "  One, two."
```

The Future of Latka

- Compile to a "reasonable" target

The Future of Latka

- Compile to a "reasonable" target
- Refine the human-language API

The Future of Latka

- Compile to a "reasonable" target
- Refine the human-language API
- Explore what constitutes a useful standard library (e.g. `Language.Natural.Latin`)

The Future of Latka

- Compile to a "reasonable" target
- Refine the human-language API
- Explore what constitutes a useful standard library (e.g. `Language.Natural.Latin`)
- Use Latka to produce data other than text

And So Forth

```
data Rank = Hatchling | Elder

carapaceColor : String
carapaceColor = "mottled" | "yellow" | "red" | "black"

descriptionFor : Rank -> String -> Para
descriptionFor.Hatchling.name =
  pa.( se.( "This is ", name
          , ", a Hatchling of the Kute'aha" )
     , ( se.( "It will ", "soon" | "eventually"
            , "graduate and become an adult" )
       | se.( "It has just been born to"
            , kuteahaName, "and has"
            , 1d200, "siblings" ) ) )
descriptionFor.Elder.name = ...
```