#lang primal-form

Jacob Adley and Lalo Viduarri

Fundamental Theorem of Arithmetic

- Every integer n ≥ 2 can be factored into a product of primes in exactly one way (aside from rearranging the factors)
- $\mathbb{Z} \rightarrow$ Prime Factorization
- Default form of integers
- Still have access to base 10 integers
- Make working easy in this form
- User does not have to know implementation to do basic work

Why

- By representing number in their primal-form it becomes very simple to do some normally ‘complex” operations.
  - Multiplication -> Set union
  - Division -> Set subtraction
- Facilitate reasoning and working with numbers in their prime factorization
  - Euler’s Totient
  - Co-primes (≡ greatest common denominator)

Consequences

- Big numbers = slow
  - We have ideas
- Simple is hard?
  - add1
  - sub1
  - +
  - -
- Going between representations of integers

Data Type

- Normal stuff
  - Intuitive
  - Clean
  - Dank

Data Type: Syntax

- User input
  - Raw numbers
    - $\mathbb{Z}$
  - Primal numbers: **Literals**: ^ : ( ) **Variables**: x, w ∈ $\mathbb{Z}$; y, z ∈ $\mathbb{N}$
    - (x)
    - (x y)
    - (x : y)
    - (x ^ y)
    - (x ^ y w)
    - (x ^ y : w)
    - (x ^ y : w ^ z)
    - (x ^ y w ^ z)
  - Negative primal numbers
    - (neg …)
    - (¬ …)

Data Type: Syntax II

- User input
  - Normal integers, $n \in \mathbb{N}$
    - (int n) → n
  - The no kiddin operator !t
    - Trust on user
    - No checks performed on input values
    - (!t …) → (…)

Data Type: Implementation

- #%datum
- #%app
- parse-primal
- !-parse-primal
- base-normalize

Match

- Needed to provide ways to allow the programmer to use numbers without having to dig into our implementation of them. (lists)
- Adding match cases was a simple way to this.
  - Allows programmers to access and describe properties that they want numbers to have.
  - Since we are using match expander then programmers can use the match they are used to

Match: Syntax

Match on a Factor	Match on a power	Match on both
`(mach num [(fac 7 -> a b) bod)]`	`(mach num [(pow 4 -> a b) bod)]`	`(match num [(n-to-the 3 6 -> a b) bod]`
Looks to see if there is a factor of 7	Looks for anything raised to the 4th power	Looks for that specific factor raised to that power

Match: Implementation

```
(define-match-expander fac
  (λ (stx)
    (syntax-case stx (≫ >>)
      [(_ num ≫ n m) #`(app (factor num) n m)]
      [(_ num >> n m) #`(app (factor num) n m)])))
```

- Write a function to traverse a primal
- Use matches app syntax to call it and bind the variables.
- Function always succeeds and needs to be wrapped with guards since it always succeeds.

Dank Examples

- φ
- gcd
- lcm
- prime?

Going →: Changes

- ∫ Efficiency
  - ∬ Table with previous computed primes
- ∫ Printing
  - ∬ Current print shows implementation
  - ∬ Want to print similar to construction
    - ∭ Use structs
- ∫ Normalize output of implemented functions
  - ∬ Can have -1 ^ x s.t. x > 1
  - ∬ **Bug**

Going →: Future Implementation

- Variables in number construction
  - (define x 5)
  - (define y 2)
  - (x ^ y)
  - (!t x ^ y)
- Modular Arithmetic
  - Define at the top of the file what space to work in: $\mathbb{Z}$ (mod n)