System F Language Specification

Syntax

Expressions

e	:=	l	literals
		v	expression identifier
		`(`e`)`	parenthesized
		e`[`τ`]`	type concretization
		$e_1\ e_2$	application
		$e_1\ op\ e_2$	binary operation
		`(`e_1`, ..., `e_n`)`	n-tuples, $n \geq 2$
		`λ (`v_1`:`t_1`) ... (`v_n`:`t_n`):` τ_{ret}`.` e	lambda abstraction
		`Λ` $t_1 \ldots t_n$`.` e	type abstraction
		`let` p `=` e_1 `in` e_2	let binding
		`if` e_1 `then` e_2 `else` e_3	if expression
l	:=	`null`	unit literal: `Unit`
		`true` | `false`	boolean literals: `Bool`
		... | `~2` | `~1` | `0` | `1` | `2` | ...	64-bit signed ints: `Int`
p	:=	`_ :` τ	discarded pattern
		v `:` τ	single argument
		`(`p_1`, ..., `p_n`)`	n-tuple destructor, $n \geq 2$

Types

τ	:=	t	type identifier
		`(`τ`)`	parenthesized
		τ_1 `->` τ_2	arrow types
		`∀` t`.` τ	universal types
		τ_1 `*` ... `*` τ_n	tuple types, $n \geq 2$
		`Int` | `Bool` | `Unit`	built-in types

Declarations

δ	:=	`let` p `=` e	declaration

Alternative syntax

We can write \ or **`lambda`** instead of **λ**.
We can write **`any`** in place of **Λ**.
We can write **`forall`** in place of ∀.

Semantics:

Call-by-value big step semantics.
When a bound variable is bound again, the new binding takes over.
There is no one-type tuples
Lexical scope.