```
module target where

-- Operator precedence and associativity
infix 4 _≤s_
infixl 6 _∸s_ _−s_

open import lib

-- Stack descriptor: (frames, displacement)
record SD : Set where
  constructor ⟨_,_⟩
  field
    f : ℕ
    d : ℕ

-- Stack descriptor operations
_+s_ : SD → ℕ → SD
⟨ f , d ⟩ +s n = ⟨ f , d + n ⟩

_∸s_ : SD → ℕ → SD
⟨ f , d ⟩ ∸s n = ⟨ f , d ∸ n ⟩

-- _-s_ : (sd : SD) → (n : ℕ) → n ≤ SD.d sd → SD
-- (⟨ S_f , S_d ⟩ -s n) p = ⟨ S_f , (S_d - n) p ⟩

_−s_ : (sd : SD) → Fin (suc (SD.d sd)) → SD
⟨ f , d ⟩ −s n = ⟨ f , d − n ⟩

−s≡ : ∀ {f d d' n} → (d' − n ≡ d) → ⟨ f , d ⟩ ≡ ⟨ f , d' ⟩ −s n
−s≡ p rewrite p = refl

-- Stack descriptor lexicographic ordering
data _≤s_ : SD → SD → Set where
  <-f : ∀ {f f' d d'} → f < f' → ⟨ f , d ⟩ ≤s ⟨ f' , d' ⟩
  ≤-d : ∀ {f d d'} → d ≤ d' → ⟨ f , d ⟩ ≤s ⟨ f , d' ⟩

≤s-refl : ∀{sd : SD} → sd ≤s sd
≤s-refl {⟨ f , d ⟩} = ≤-d ≤-refl

≤s-trans : ∀{sd sd' sd" : SD} → sd ≤s sd' → sd' ≤s sd" → sd ≤s sd"
≤s-trans (<-f f<f') (≤-d _) = <-f f<f'
≤s-trans (<-f f<f') (<-f f'<f") = <-f (<-trans f<f' f'<f")
≤s-trans (≤-d _) (<-f f'<f") = <-f f'<f"
≤s-trans (≤-d d≤d') (≤-d d'≤d") = ≤-d (≤-trans d≤d' d'≤d")

+s→≤s : ∀{sd : SD} → ∀{n : ℕ} → sd ≤s sd +s n
+s→≤s = ≤-d +→≤

-- Operator
data UnaryOp : Set where
  UNeg : UnaryOp

data BinaryOp : Set where
  BPlus : BinaryOp
  BMinus : BinaryOp
  BTimes : BinaryOp

data RelOp : Set where
  RLeq : RelOp
  RLt : RelOp

-- Nonterminals
-- Lefthand sides
data L (sd : SD) : Set where
  l-var : (sd' : SD) → sd' ≤s sd → L sd
  l-sbrs : L sd

-- Simple righthand sides
data S (sd : SD) : Set where
  s-l : L sd → S sd
  s-lit : ℤ → S sd

-- Righthand sides
data R (sd : SD) : Set where
  r-s : S sd → R sd
  r-unary : UnaryOp → S sd → R sd
  r-binary : S sd → BinaryOp → S sd → R sd

-- Instruction sequences
data I (sd : SD) : Set where
  stop : I sd
  assign-inc : (δ : ℕ) → L (sd +s δ) → R sd → I (sd +s δ) → I sd
  assign-dec : (δ : Fin (suc (SD.d sd))) → L (sd −s δ) → R sd
                 → I (sd −s δ) → I sd
  if-then-else-inc : (δ : ℕ) → S sd → RelOp → S sd
                       → I (sd +s δ) → I (sd +s δ) → I sd
  if-then-else-dec : (δ : Fin (suc (SD.d sd))) → S sd → RelOp → S sd
                       → I (sd −s δ) → I (sd −s δ) → I sd
  adjustdisp-inc : (δ : ℕ) → I (sd +s δ) → I sd
  adjustdisp-dec : (δ : Fin (suc (SD.d sd))) → I (sd −s δ) → I sd
  popto : (sd' : SD) → sd' ≤s sd → I sd' → I sd
```