Part I

Language of Arithmetic

```
{+ 2 1}
```

$\twoheadrightarrow$ `3`

Language of Arithmetic

```
{* 2 1}
```

$\twoheadrightarrow$ `2`

Language of Arithmetic

```
{+ 2 {* 4 3}}
```

$\twoheadrightarrow$ `14`

Language of Arithmetic

2

$\twoheadrightarrow$ 2

Representing Expressions

- numbers
- addition expressions
  - first and second arguments are expressions
- multiplication expressions
  - first and second arguments are expressions

Representing Expressions

```
(define-type ArithC
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])
```

Part 2

Concrete vs. Abstract Syntax

```
{+ 2 1}
```

```
(plusC (numC 2) (numC 1))
```

Concrete Syntax as an S-expression

```
'{+ 2 1}
```

```
(test (parse '{+ 2 1})
      (plusC (numC 2) (numC 1)))
```

Concrete Syntax as an S-expression

```
; An arith-S-exp is either
;   - number
;   - (list '+ arith-S-expr arith-S-expr)
;   - (list '* arith-S-expr arith-S-expr)
```

Part 3

Subtraction

```
{- 5 2}
```

↠ 3

Subtraction

```
(define-type ArithC
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [bminusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])
```

NEW

?

Subtraction

```
{- 5 2}
= {+ 5 -2}
```

Subtraction

```
{- 5 2}
= {+ 5 {* -1 2}}
```

Desugaring

```
(define-type ArithC
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])

(define-type ArithS ; not ArithC
  [numS (n : number)]
  [plusS (l : ArithS) (r : ArithS)]
  [bminusS (l : ArithS) (r : ArithS)]
  [multS (l : ArithS) (r : ArithS)])

desugar : (ArithS -> ArithC)
```

Part 4

Functions

```
{define {double x}
  {+ x x}}
{define {quadruple x}
  {double {double x}}}

{quadruple 2}
```

→↠ 8

Functions

```
{+ {define {double x} {+ x x}}
   1}
```

?

No: a function ***definition*** is not an expression

Functions

```
{+ {double 4}
   1}
```

?

Yes: a function ***call*** is an expression

We'll use ***call*** and ***application*** interchangably

Function Definitions

```
{define {triple x}
  {+ x {+ x x}}}
```

A function has

- a name
- an argument name
- a *body*

```
(define-type BodyC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : BodyC) (r : BodyC)]
  [multC (l : BodyC) (r : BodyC)])
```

?

Function Definitions

```
{define {triple x}
  {+ x {+ x x}}}
```

A function has

- a name
- an argument name
- a *body*

Allow `x` to be an expression, and then

```
{+ x {+ x x}}
```

is also an expression

Functions and Function Calls

- numbers
- identifiers
- addition expressions
  - first and second arguments are expressions
- multiplication expressions
  - first and second arguments are expressions
- function-call expressions
  - a function name and an argument expression

- a function definition
  - a function name, argument name, and body expression

Functions and Function Calls

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC)
         (r : ExprC)]
  [multC (l : ExprC)
         (r : ExprC)]
  [appC (s : symbol)
        (arg : ExprC)])

(define-type FunDefC
  [fdC (name : symbol)
       (arg : symbol)
       (body : ExprC)])
```

Representing Programs

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC)
         (r : ExprC)]
  [multC (l : ExprC)
         (r : ExprC)]
  [appC (s : symbol)
        (arg : ExprC)])

(define-type FunDefC
  [fdC (name : symbol)
       (arg : symbol)
       (body : ExprC)])
```

```
{+ 1 2}
```

```
(plusC (numC 1)
       (numC 2))
```

Representing Programs

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC)
         (r : ExprC)]
  [multC (l : ExprC)
         (r : ExprC)]
  [appC (s : symbol)
        (arg : ExprC)])

(define-type FunDefC
  [fdC (name : symbol)
       (arg : symbol)
       (body : ExprC)])
```

```
{+ x 2}
```

```
(plusC (idC 'x)
       (numC 2))
```

Representing Programs

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC)
         (r : ExprC)]
  [multC (l : ExprC)
         (r : ExprC)]
  [appC (s : symbol)
        (arg : ExprC)])

(define-type FunDefC
  [fdC (name : symbol)
       (arg : symbol)
       (body : ExprC)])
```

```
{define {plus-two x}
  {+ x 2}}


(fdC 'plus-two
     'x
     (plusC (idC 'x)
            (numC 2)))
```

Representing Programs

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC)
         (r : ExprC)]
  [multC (l : ExprC)
         (r : ExprC)]
  [appC (s : symbol)
        (arg : ExprC)])

(define-type FunDefC
  [fdC (name : symbol)
       (arg : symbol)
       (body : ExprC)])
```

```
{plus-two 9}
```

```
(appC 'plus-two
      (numC 9))
```

Representing Programs

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC)
         (r : ExprC)]
  [multC (l : ExprC)
         (r : ExprC)]
  [appC (s : symbol)
        (arg : ExprC)])

(define-type FunDefC
  [fdC (name : symbol)
       (arg : symbol)
       (body : ExprC)])
```

```
{define {double x}
  {+ x x}}
{define {quadruple x}
  {double {double x}}}

{quadruple 2}
```

```
(list (fdC 'double 'x
           (plusC (idC 'x)
                  (idC 'x)))
      (fdC 'quadruple 'x
           (appC 'double
                 (appC 'double
                       (idC 'x)))))

(appC 'quadruple (numC 2))
```

Part 5

Evaluating Function Calls

```
{define {double x}
  {+ x x}}

{double 3}

↠ {+ 3 3}

↠ 6
```

```
interp : (ExprC (listof FunDefC) -> number)

get-fundef : (symbol (listof FunDefC) -> FunDefC)

subst : (ExprC symbol ExprC -> ExprC)
```