Advanced Student Language

A `<defn>` is one of

```
(define <var> <exp>)
(define (<var> <var> ... <var>) <exp>)
(define-struct <var> (<var> ... <var>))
```

An `<exp>` is one of

```
<var>
<con>
<prim>
(<exp> <exp> ... <exp>)
(cond [<exp> <exp>] ... [<exp> <exp>])
(cond [<exp> <exp>] ... [else <exp>])
(and <exp> ... <exp>)
(or <exp> ... <exp>)
(local [<defn> ...] <exp>)
(lambda (<var> ... <var>) <exp>)
(set! <var> <exp>)
(begin <exp> ... <exp>)
```
Mini Racket

A `<defn>` is one of

```
(define <var> <exp>)
(define <var> (lambda (<var>) <exp>))
```

An `<exp>` is one of

```
<var>
<num>
(+ <exp> <exp>)
(- <exp> <exp>)
(* <exp> <exp>)
(<var> <exp>)
```
Implementing Aquariums in Advanced Student
Implementing Aquariums in Advanced Student
Implementing Aquariums in Advanced Student

Represent fish, as opposed to stuffing real fish into DrRacket
Implementing Mini Racket in Advanced Student
Implementing Mini Racket in Advanced Student
Implementing Mini Racket in Advanced Student

Represent Mini Racket expressions, as opposed to typing real expressions into DrRacket
Representing Mini Racket Expressions

An `<exp>` is one of

- `<var>`
- `<num>`
- `( + <exp> <exp> )`
- `( - <exp> <exp> )`
- `( * <exp> <exp> )`
- `( <var> <exp> )`

We can’t simply write

```
(+ 1 2)
```

to represent a Mini Racket addition expression
Representing Mini Racket Expressions

An \(<\text{exp}>\) is one of
\(<\text{var}>\)
\(<\text{num}>\)
\((+ \ <\text{exp}> \ <\text{exp}>)\)
\((- \ <\text{exp}> \ <\text{exp}>)\)
\((* \ <\text{exp}> \ <\text{exp}>)\)
\((<\text{var}> <\text{exp}>)\)

We can write

\((\text{make-\text{plus} 1 2})\)
Representing Mini Racket Expressions

An <exp> is one of

<var>
<num>
(+ <exp> <exp>)
(- <exp> <exp>)
(* <exp> <exp>)
(<var> <exp>)

To represent the <var> x:

'x
Representing Mini Racket Expressions

An `<exp>` is one of

- `<var>`
- `<num>`
- `( + `<exp>` `<exp>` )`
- `( - `<exp>` `<exp>` )`
- `( * `<exp>` `<exp>` )`
- `( <var> `<exp>` )`

To represent the `<num>` 5:

5
Representing Mini Racket Expressions

An `<exp>` is one of
- `<var>`
- `<num>`
- `(+ `<exp>` `<exp>`)`
- `(- `<exp>` `<exp>`)`
- `(* `<exp>` `<exp>`)`
- `(<var> `<exp>`)`

To represent the application `(f (+ 1 2))`

`(make-app 'f (make-plus 1 2))`
Representing Mini Racket Expressions

Data definition:

; An expr is either
;   - sym
;   - num
;   - (make-plus expr expr)
;   - (make-minus expr expr)
;   - (make-times expr expr)
;   - (make-app sym expr)

Evaluation:

; evaluate : expr dictionary -> val
Definitions and Values

; A dictionary is
;   hash-table of sym to binding

; A binding is either
;   - val
;   - function

; A function is
;   (make-function sym expr)

; A val is a num
See miniracket1.rkt
Lambda Expressions

; An expr is either
;  - sym
;  - num
;  - (make-plus expr expr)
;  - (make-minus expr expr)
;  - (make-times expr expr)
;  - (make-app expr expr)
;  - (make-lambda sym expr)

; A val is either
;  - num
;  - (make-function sym expr)

; A dictionary is
;  hash-table of sym to val
See miniracket2.rkt