

Original Grammar for Algebra Programs

```
<prog>   ::=   <defn>* <expr>
<defn>   ::=   <id>(<id>) = <expr>
<expr>   ::=   (<expr> + <expr>)
              ::=   (<expr> - <expr>)
              ::=   <id>(<expr>)
              ::=   <id>   |   <num>
<id>     ::=   a variable name: f, x, y, z, ...
<num>    ::=   a number: 1, 42, 17, ...
```

- Change step-by-step to Scheme

Change #1: Move Paren and Add "define"

```
<prog> ::= <defn>* <expr>
<defn> ::= (define (<id> <id>) <expr>)
<expr> ::= (+ <expr> <expr>)
           ::= (- <expr> <expr>)
           ::= (<id> <expr>)
           ::= <id> | <num>
<id>   ::= a variable name: f, x, y, z, ...
<num>  ::= a number: 1, 42, 17, ...
```

- + and - moved to initial position

Change #1: Move Paren and Add "define"

```
<prog>   ::=  <defn>* <expr>
<defn>   ::=  (define (<id> <id>) <expr>)
<expr>   ::=  (+ <expr> <expr>)
              ::=  (- <expr> <expr>)
              ::=  (<id> <expr>)
              ::=  <id> | <num>
<id>     ::=  a variable name: f, x, y, z, ...
<num>    ::=  a number: 1, 42, 17, ...
```

- Open parenthesis for function call moved to before the function name

Change #1: Move Paren and Add "define"

```
<prog>   ::=  <defn>* <expr>
<defn>   ::=  (define (<id> <id>) <expr>)
<expr>   ::=  (+ <expr> <expr>)
              ::=  (- <expr> <expr>)
              ::=  (<id> <expr>)
              ::=  <id> | <num>
<id>     ::=  a variable name: f, x, y, z, ...
<num>    ::=  a number: 1, 42, 17, ...
```

- Definition use the **define** keyword, and parenthesis moved

Change #1: Move Parentheses and Add "define"

```
<prog> ::= <defn>* <expr>
<defn> ::= (define (<id> <id>) <expr>)
<expr> ::= (+ <expr> <expr>)
           ::= (- <expr> <expr>)
           ::= (<id> <expr>)
           ::= <id> | <num>
<id>    ::= a variable name: f, x, y, z, ...
<num>   ::= a number: 1, 42, 17, ...
```

f(x) = (x + 1)

f((2 + 3))



(define (f x) (+ x 1))

(f (+ 2 3))

Change #2: Generalize + and -, Add Primitives

```
<expr> ::= ...
          ::= (+ <expr>*)
          ::= (- <expr>+)
          ::= (* <expr>*)
          ::= (/ <expr>+)
          ::= (modulo <expr> <expr>)
          ::= (expt <expr> <expr>)
          ::= ...
```

(+ 1 2 3 7) → 13

Change #2: Generalize + and -, Add Primitives

```
<expr> ::= ...
          ::= (+ <expr>*)
          ::= (- <expr>+)
          ::= (* <expr>*)
          ::= (/ <expr>+)
          ::= (modulo <expr> <expr>)
          ::= (expt <expr> <expr>)
          ::= ...
```

(+) → 0

Change #2: Generalize + and -, Add Primitives

```
<expr> ::= ...
          ::= (+ <expr>*)
          ::= (- <expr>+)
          ::= (* <expr>*)
          ::= (/ <expr>+)
          ::= (modulo <expr> <expr>)
          ::= (expt <expr> <expr>)
          ::= ...
```

$$(- 1) \rightarrow -1$$

Change #2: Generalize + and -, Add Primitives

```
<expr> ::= ...
          ::= (+ <expr>*)
          ::= (- <expr>+)
          ::= (* <expr>*)
          ::= (/ <expr>+)
          ::= (modulo <expr> <expr>)
          ::= (expt <expr> <expr>)
          ::= ...  
  
(*) → 1
```

Change #2: Generalize + and -, Add Primitives

```
<expr> ::= ...
          ::= (+ <expr>*)
          ::= (- <expr>+)
          ::= (* <expr>*)
          ::= (/ <expr>+)
          ::= (modulo <expr> <expr>)
          ::= (expt <expr> <expr>)
          ::= ...
```

(/ 2) → 1/2

Change #2: Generalize + and -, Add Primitives

```
<expr> ::= ...
          ::= (+ <expr>*)
          ::= (- <expr>+)
          ::= (* <expr>*)
          ::= (/ <expr>+)
          ::= (modulo <expr> <expr>)
          ::= (expt <expr> <expr>)
          ::= ...
```

(**expt** 13 20) → 19004963774880799438801

Change #3: Generalize Defined Functions

```
<defn> ::= (define (<id> <id>*) <expr>)
<expr> ::= ...
          ::= (<id> <expr>*)
```

```
(define (f a b) (+ a b)) → (define (f a b) (+ a b))
(f 1 2)                      3
```

Change #3: Generalize Defined Functions

$\langle \text{defn} \rangle ::= (\mathbf{define} \ (\langle \text{id} \rangle \langle \text{id} \rangle^*) \ \langle \text{expr} \rangle)$

$\langle \text{expr} \rangle ::= \dots$

$\quad ::= (\langle \text{id} \rangle \langle \text{expr} \rangle^*)$

$\dots (\mathbf{define} \ (\langle \text{id} \rangle_0 \ \langle \text{id} \rangle_1 \dots \langle \text{id} \rangle_k) \ \langle \text{expr} \rangle_a) \ \dots$

$\dots (\langle \text{id} \rangle_0 \ \langle \text{expr} \rangle_1 \dots \langle \text{expr} \rangle_k) \ \dots$

\rightarrow

$\dots (\mathbf{define} \ (\langle \text{id} \rangle_0 \ \langle \text{id} \rangle_1 \dots \langle \text{id} \rangle_k) \ \langle \text{expr} \rangle_a) \ \dots$

$\dots \langle \text{expr} \rangle_b \ \dots$

where $\langle \text{expr} \rangle_b$ is $\langle \text{expr} \rangle_a$ with $\langle \text{id} \rangle_i$ replaced by $\langle \text{expr} \rangle_i$;

Change #4: Add Booleans

```
<expr> ::= ...
          ::= <bool>
          ::= (and <expr>*)
          ::= (or <expr>*)
          ::= (zero? <expr>)
          ::= ...
<bool> ::= #f
          ::= #t
```

(and #t #f) → #f

Change #4: Add Booleans

```
<expr> ::= ...
          ::= <bool>
          ::= (and <expr>*)
          ::= (or <expr>*)
          ::= (zero? <expr>)
```

```
          ...
<bool> ::= #f
          ::= #t
```

(zero? 1) → #f

Change #4: Add Booleans

```
<expr> ::= ...
          ::= <bool>
          ::= (and <expr>*)
          ::= (or <expr>*)
          ::= (zero? <expr>)
```

```
          ...
<bool> ::= #f
          ::= #t
```

(zero? 0) → #t

Change #5: Add Symbols

```
<expr>      ::=  ...
                ::=  <symbol>
                ::=  (eq? <expr> <expr>)
                ::=  ...
<symbol>  ::=  '<id>
```

(eq? 'a 'a) → #t

Change #5: Add Symbols

```
<expr>      ::=  ...
                ::=  <symbol>
                ::=  (eq? <expr> <expr>)
                ::=  ...
<symbol>  ::=  '<id>
```

(eq? 'a 'b) → #f

Change #6: Add Conditionals

```
<expr>      ::=  ...
               ::=  (cond <cond-line>*)
<cond-line>  ::=  [<expr> <expr>]
               ::=  [else <expr>]
```

(**cond** [#t 1]) → 1

Change #6: Add Conditionals

`<expr>` ::= ...
 ::= (**cond** `<cond-line>`*)
`<cond-line>` ::= [`<expr>` `<expr>`]
 ::= [**else** `<expr>`]

(**cond** [#f 1] [#t 2]) → (**cond** [#t 2])

(**cond** [#t 2]) → 2

Change #6: Add Conditionals

```
<expr>      ::=  ...
               ::=  (cond <cond-line>*)
<cond-line>  ::=  [<expr> <expr>]
               ::=  [else <expr>]
```

(**cond** [#t 1] [#t 2]) → 1

Change #6: Add Conditionals

`<expr>` ::= ...
 ::= (**cond** `<cond-line>`*)
`<cond-line>` ::= [`<expr>` `<expr>`]
 ::= [**else** `<expr>`]

(**cond** [#f 1] [**else** 5]) → ... → 5

Change #6: Add Conditionals

`<expr>` ::= ...
 ::= (**cond** `<cond-line>`*)
`<cond-line>` ::= [`<expr>` `<expr>`]
 ::= [**else** `<expr>`]

(**cond** [(**zero?** 5) 1] [**else** 2]) → (**cond** [#f 1] [**else** 2])

(**cond** [#f 1] [**else** 2]) → (**cond** [**else** 2]) → 2

Change #7: Values

While extending the set of expressions, we've also extended the set of **values**:

```
<val> ::= <num>
          ::= <bool>
          ::= <symbol>
```

- Evaluation stops when it reaches a member of **<val>**
- Function application requires a **<val>**

Change #7: Values

While extending the set of expressions, we've also extended the set of ***values***:

<val> ::= **<num>**
 ::= **<bool>**
 ::= **<symbol>**

(define (f x) (+ x 1)) → **(define (f x) (+ x 1))**
(f (+ 1 2)) **(f 3)**

not **(define (f x) (+ x 1))**
 (+ (+ 1 2) 1)

Change #7: Values

While extending the set of expressions, we've also extended the set of **values**:

```
<val> ::= <num>  
        ::= <bool>  
        ::= <symbol>
```

... (**define** (<id>₀ <id>₁...<id>_k) <expr>_a) ...

... (<id>₀ <val>₁...<val>_k) ...

→

... (**define** (<id>₀ <id>₁...<id>_k) <expr>_a) ...

... <expr>_b ...

where <expr>_b is <expr>_a with <id>_i replaced by <val>_i

Change #8: Add Pairs

```
<expr> ::= ...
          ::= (cons <expr> <expr>)
          ::= (car <expr>)
          ::= (cdr <expr>)

<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>)
```

(**cons** 1 2) \in ? <val>

yes

Change #8: Add Pairs

```
<expr> ::= ...
          ::= (cons <expr> <expr>)
          ::= (car <expr>)
          ::= (cdr <expr>)

<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>)
```

(**cons** 1 (**cons** 2 (**cons** 3 4))) \in <val>

Change #8: Add Pairs

```
<expr> ::= ...
          ::= (cons <expr> <expr>)
          ::= (car <expr>)
          ::= (cdr <expr>)

<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>)
```

(**cons** 1 (+ 1 1)) $\in ?$ <val>

no

(**cons** 1 (+ 1 1)) \rightarrow (**cons** 1 2)

Change #8: Add Pairs

```
<expr> ::= ...
          ::= (cons <expr> <expr>)
          ::= (car <expr>)
          ::= (cdr <expr>)

<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>)
```

(**car** (**cons** 1 2)) \in ? <val>

no

(**car** (**cons** 1 2)) \rightarrow 1

Change #8: Add Pairs

```
<expr> ::= ...
          ::= (cons <expr> <expr>)
          ::= (car <expr>)
          ::= (cdr <expr>)
<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>)
```

More generally:

$$(\mathbf{car} \ (\mathbf{cons} \ X \ Y)) \rightarrow X$$

$$(\mathbf{cdr} \ (\mathbf{cons} \ X \ Y)) \rightarrow Y$$

Change #8: Add Pairs

```
<expr> ::= ...
          ::= (cons <expr> <expr>)
          ::= (car <expr>)
          ::= (cdr <expr>)
<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>)
```

Also:

(**cadar** (**cons** (**cons** *X* (**cons** *Y* *Z*) *W*)) → *Y*

(or any combination of up to four "a"s and "d"s)

Change #9: Add the Empty List

```
<expr> ::= ...
          ::= '()

<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>) | '()

<lst>   ::= '()
          ::= (cons <val> <lst>)
```

'() \in ? <lst>

yes

Change #9: Add the Empty List

```
<expr> ::= ...
          ::= '()

<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>) | '()

<lst>   ::= '()
          ::= (cons <val> <lst>)
```

(cons 1 '()) $\in ?$ <lst>

yes

Change #9: Add the Empty List

```
<expr> ::= ...
          ::= '()

<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>) | '()

<lst>   ::= '()
          ::= (cons <val> <lst>)
```

(cons '() 1) ∈ ? <lst>

no

Change #9: Add the Empty List

```
<expr> ::= ...
          ::= '()

<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>) | '()

<lst>   ::= '()
          ::= (cons <val> <lst>)
```

(cons 1 (cons 2 (cons 3 '()))) $\in ? \langle \text{lst} \rangle$

yes

Change #9: Add the Empty List

```
<expr> ::= ...
          ::= '()

<val>  ::= <num> | <bool> | <symbol>
          ::= (cons <val> <val>) | '()

<lst>   ::= '()
          ::= (cons <val> <lst>)
```

Is every **<lst>** a **<val>**?

yes

List Shortcuts

- **(list 1 2 3) = (cons 1 (cons 2 (cons 3 '())))**
- **'(1 2 3) = (cons 1 (cons 2 (cons 3 '())))**
- **'(a b c) = (cons 'a (cons 'b (cons 'c '())))**
- **'(a (1 2) c)**
 = **(list 'a '(1 2) 'c)**
 = **(cons 'a (cons (cons 1 (cons 2 '()))) (cons 'c '())))**

List Predicates

- **(null? '()) = #t**
- **(null? (cons 1 2)) = #f**
- **(pair? '()) = #f**
- **(pair? (cons 1 2)) = #t**
- **(list? '()) = #t**
- **(list? (cons 1 2)) = #f**
- **(list? (cons 1 '())) = #t**

Change #10: Functions Are Values

<val>

::= ...

::= **<defined-id>**

<defined-id>

::= an **<id>** that is **defined**

```
(define (twice f x) (f (f x)))    →  (define (twice f x) (f (f x)))
(define (g y) (+ y 1))              →  (define (g y) (+ y 1))
(twice g 0)                         →  (g (g 0))
```

Change #10: Functions Are Values

<val>

::= ...

::= **<defined-id>**

<defined-id>

::= an **<id>** that is **defined**

```
(define (twice f x) (f (f x)))    →  (define (twice f x) (f (f x)))
(define (g y) (+ y 1))              →  (define (g y) (+ y 1))
(g (g 0))                          →  (g (+ 0 1))
```

Change #10: Functions Are Values

<val>

::= ...

::= **<defined-id>**

<defined-id>

::= an **<id>** that is **defined**

```
(define (twice f x) (f (f x)))    →  (define (twice f x) (f (f x)))
(define (g y) (+ y 1))              (define (g y) (+ y 1))
(g (+ 0 1))                        (g 1)
```

Change #10: Functions Are Values

<val>

::= ...

::= **<defined-id>**

<defined-id>

::= an **<id>** that is **defined**

```
(define (twice f x) (f (f x)))    →  (define (twice f x) (f (f x)))
(define (g y) (+ y 1))              (define (g y) (+ y 1))
(g 1)                                (+ 1 1)
```

Change #10: Functions Are Values

<val>

::= ...

::= **<defined-id>**

<defined-id>

::= an **<id>** that is **defined**

(**define** (**twice** f x) (f (f x))) → (**define** (**twice** f x) (f (f x)))
(**define** (g y) (+ y 1)) → (**define** (g y) (+ y 1))
(+ 1 1) 2

Alternate Notations (Used by the Book)

- **(define** <id>₀ **(lambda** (<id>₁...<id>_k) <expr>))
= **(define** (<id>₀ <id>₁...<id>_k) <expr>)
- **(if** <expr>₁ <expr>₂ <expr>₃)
= **(cond** [<expr>₁ <expr>₂] [**else** <expr>₃])

Stuff You Don't Need (Yet)

- local bindings
- **set!**
- **begin** (including implicit)
- I/O primitives
- vectors
- **do** (never need this one)