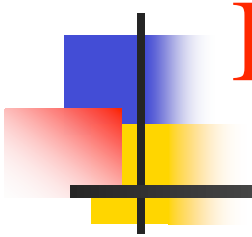
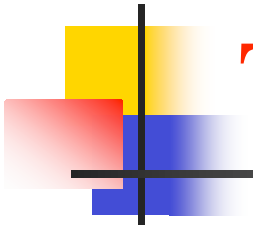
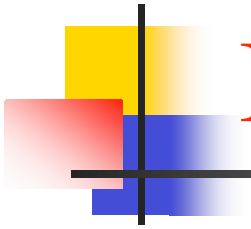


Syntax and Semantics of Beginner Scheme



Today's Goals

- Tools
  - A mathematical notion of formal languages
  - Backus-Naur Form (BNF)
- A Grammar for Scheme
- Review of semantics
- Semantic equivalences
- Runtime errors



Backus-Naur Form (BNF)

- Backus invented FORTRAN (50's)
- Nauer worked on Algol 60 (60's). Scheme is 70's
- We model formal languages mathematically as:
  - A set of sentences, each sentence a sequence of words
  - Possibly infinite set, but still with interesting grammar
- Examples
  - Language$_1$ = {"`Ringo`", "`Bingo`"}. Finite # sentences
  - BNF is <exp-1> ::= `Ringo` | `Bingo`
  - Language$_2$ = {"`A`", "`A B`", "`A B B`", …}. Infinite.
  - BNF is <exp-1> ::= `A` | <exp-1> `B`

Beginner Scheme Grammar

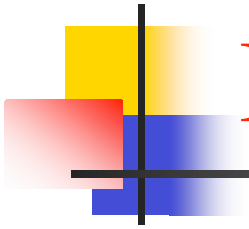
<var> ::= `x` | `y` | `my-posn` | …

// Using "…" in a BNF is a bit informal

<con> ::= `true` | `false` | `1` | `-1` | `.04` | …

| `'a` | `'rabbit` | … // other examples?

<prm> ::= `+` | `-` | `*` | …



Beginner Scheme Grammar

<exp> ::= <var> | <con>

| (<prm> <exp>$^{+}$) // one or more

| (<var> <exp>*) // zero or more

| (`cond` [<exp> <exp>]$^{+}$) // not (<var> …)

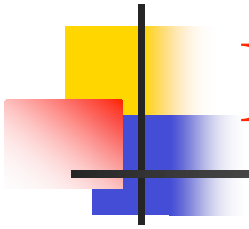
| (`cond` [<exp> <exp>]* [`else` exp])

| (`and` <exp>$^{+}$)

| (`or` <exp>$^{+}$)

// We have zero argument constructors, not functions.

// We say “pi” is a constant, but, technically, it is a variable



Beginner Scheme Grammar

<val> ::= <con>

<def> ::=

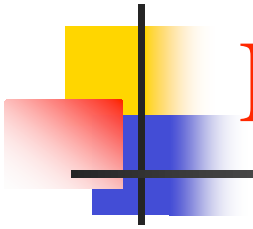
(`define` <var> <exp>)

| (`define` (<var> <var>$^{+}$) <exp>)

| (`define-struct` <var> (<var>*))

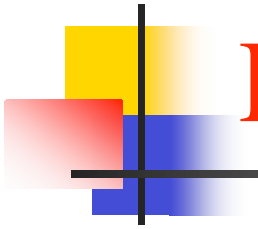
// Syntax (grammar) drives semantics, just like date types

// drives programs in our recipe.



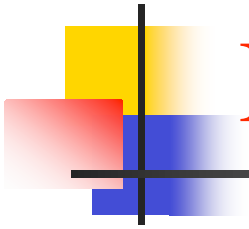
Reductions

- Define the meaning (or semantics) of programs
- The basic rule (applies everywhere!)
  - Start with the outermost left most expressiong that can be evaluated
- We have already seen reductions for
  - Arithmetic operations
  - `(+ 1 (- 3 2)) = (+ 1 1) = 2`



Reductions

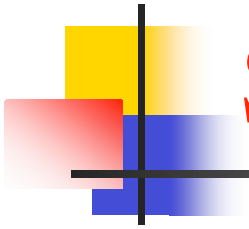
- We have already seen reductions for (cont'd)
  - Boolean operations
    - `(and true false)` = `false`
    - `(and false (/ 1 0))` = `false`
    - The `or` operator has similar, sequential semantics
  - Testing equality of symbols
    - `(symbol=? 'Rabbit 'Wabbit)` = `false`
  - Conditional statements
    - Possibly the most involved. But very useful statement
  - Errors
    - `(error 'Label "Message")` = *`Label : Message`*



Reductions

More subtle aspects of reduction

- Some definitions should be reduced first
  - Variable definitions, but not function definitions
  - `(define simple-value (/ 1 0))`
    - Raises and error
  - `(define (my-function x) (/ 1 0))`
    - By itself, does not raise an error
- Some definitions introduce other definitions
  - `(define-struct pnt (x y))`
    - Introduces: `make-pnt, pnt-x,pnt-y, pnt?`

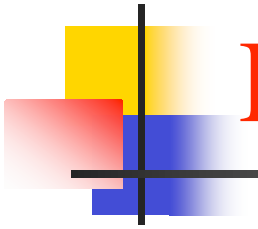


Semantic Equivalences

- One use: Some programs have
  - Different syntax
  - But same semantics

- Example
  - (`and` <exp1> <exp2>)
  - (`cond` [<exp1> <exp2>] [`else false`])

  - (`or` <exp1> <exp2>)
  - (`cond` [<exp1> `true`] [`else` <exp2>])



For Next Class

- Homework posted online
- Reading:
  - Chapter 9 and companion notes
  - Lists: Our first recursively defined type
- Quiz:
  - Chapter 9