Outline

- **Data Definitions and Templates**
- **Syntax and Semantics**
- **Defensive Programming**

Data Definitions

Question 1:
Are both of the following data definitions ok?

```
; A w-grade is either
;  - num
;  - posn
;  - empty
```
with
```
; A posn is
;  (make-posn num num)
```

```
; A z-grade is either
;  - num
;  - (make-posn num num)
;  - empty
```

Yes.

Data Definitions

Question 2:
Do **`w-grade`** and **`z-grade`** identify the same set of values?

```
; A w-grade is either
;  - num
;  - posn
;  - empty
```
with
```
; A posn is
;  (make-posn num num)
```

```
; A z-grade is either
;  - num
;  - (make-posn num num)
;  - empty
```

Yes, every **`w-grade`** is a **`w-grade`**,
and every **`z-grade`** is a **`w-grade`**

Data Definitions

Question 3:
Are **`w-grade`** and **`w-grade`** the same data definition?

```
; A w-grade is either
;  - num
;  - posn
;  - empty
```
with
```
; A posn is
;  (make-posn num num)
```

```
; A z-grade is either
;  - num
;  - (make-posn num num)
;  - empty
```

No, in the sense that they generate different templates

Data Definitions and Templates

The template depends on the *static*, *textual* content of a data definition, only

```
; A w-grade is either
;  - num
;  - posn
;  - empty
; A posn is
;  (make-posn num num)
```

```
(define (func-for-w-grade w)
  (cond
    [(number? w) ...]
    [(posn? w) ... (func-for-posn w) ...]
    [(empty? w) ...]))
(define (func-for-posn p)
  ... (posn-x p) ... (posn-y p) ...)
```

```
; A z-grade is either
;  - num
;  - (make-posn num num)
;  - empty
```

```
(define (func-for-z-grade z)
  (cond
    [(number? z) ...]
    [(posn? z) ... (posn-x z) ... (posn-y z) ...]
    [(empty? z) ...]))
```

Data Definitions and Templates

Why we treat the data definition statically to generate a template:

- Provides well-defined, simple rules for generating a template
  - "Dynamic" coverage is difficult in general
  - Recall 3520 anecdote: thinking in terms of dynamic coverage ⇒ broken programs
- Similar to the way that data choices affect modularity
  - Details of modularity are beyond the scope of this class, but we want to build the right instincts

Outline

- **Data Definitions and Templates**
- **Syntax and Semantics**
- **Defensive Programming**

Execution in DrScheme

Suppose that DrScheme's definition window contains

```
(define (f x)
  (/ x 2))
(f 10)
```

What's the result of clicking **Execute** ?

```
5
```

Execution in DrScheme

Suppose that DrScheme's definition window contains

```
(define (f x)
  (/ x 0))
(f 10)
```

What's the result of clicking **Execute** ?

/: divide by 0

Execution in DrScheme

Suppose that DrScheme's definition window contains

```
(define (f x)
  (/ x 0))
```

What's the result of clicking **Execute** ?

Nothing (although `f` would produce an error if it were used)

Execution in DrScheme

Suppose that DrScheme's definition window contains

```
(define (f x)
  (/ x (0)))
```

What's the result of clicking **Execute** ?

expected a name after an open parenthesis, found a number — even without using `f`

Execution in DrScheme

Suppose that DrScheme's definition window contains

```
(define (f x)
  (cond x))
```

What's the result of clicking **Execute** ?

cond: expected a question--answer clause — even without using `f`

Execution in DrScheme

Suppose that DrScheme's definition window contains

```
(define (f x)
  (cond
    [false x]))
```

What's the result of clicking **Execute** ?

Nothing

Execution in DrScheme

Suppose that DrScheme's definition window contains

```
(define (f x)
  (cond
    [false x]))
(f 10)
```

What's the result of clicking **Execute** ?

```
cond: all questions were false
```

Errors in DrScheme

DrScheme complains about a function body

- sometimes before the function is used
- sometimes only when the function is called

Why?

Because some errors are ***syntax errors*** and some errors are ***run-time errors***

Syntax Errors

A ***syntax error*** is like a question that isn't a well-formed sentence

- `f (x) = x + 0`
  - DrScheme doesn't understand this notation, just like...
- "Parlez vous Francais?"
  - English-only speaker doesn't understand this notation
- `(define (f x) (/ x (0)))`
  - Parens around a zero make no sense to DrScheme, just like...
- "Does rain dog cat?"
  - Not enough verbs for this to make sense in English

When DrScheme sees a syntax error, it refuses to evaluate

Run-Time Errors

A ***run-time error*** is like a well-formed question with no answer

- `(/ 12 0)`
  - A clear request to DrScheme, but no answer, just like...
- "Why are you wearing a green hat?"
  - There's no answer if I'm wearing a blue hat
- `(cond [false 10])`
  - There's no reasonable choice for DrScheme, just like...
- "If you can't understand me, what's your name?"
  - No one who understands the question should answer

DrScheme evaluates around run-time errors until forced to answer

The Difference between DrScheme and English

In a (good) programming language, all errors are well-defined, and the rules are relatively simple

- DrScheme has a simple, well-defined grammar, and deviations from the grammar are syntax errors
- The reduction rules for each construct and primitive operation are well-defined, producing either a value or an error

Beginner Scheme Grammar

A **<var>** is a name, a **<con>** is a constant, and a **<prm>** is an operator name

A **<defn>** is one of

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

A **<exp>** is one of

```
<var>
<con>
(<prm> <exp> ... <exp>)
(<var> <exp> ... <exp>)
(cond [<exp> <exp>] ... [<exp> <exp>])
(cond [<exp> <exp>] ... [else <exp>])
(and <exp> ... <exp>)
(or <exp> ... <exp>)
```

Evaluation Rules: and/or

```
(and true) → true
(and true question ... question)
                        → (and question ... question)
(and false question ... question) → false

(or false) → false
(or false question ... question)
                        → (or question ... question)
(or true question ... question) → true
```

Note that

```
(and 7 false)
```

fits the grammar, but has no matching evaluation rule, so it produces a run-time error

Outline

- **Data Definitions and Templates**
- **Syntax and Semantics**
- **Defensive Programming**

Execution in DrScheme

Suppose that DrScheme's definition window contains

```
; f : num -> num
(define (f x)
  (+ x 2))
(f 'apple)
```

What's the result of clicking **Execute** ?

```
+: expects a <number>, given 'apple
```

But this is really a contract violation at the call to **f**

The implementor of **f** might want to clarify that this error is someone else's fault, not a bug in **f**

Defensive Programming

```
; f : num -> num
(define (real-f x)
  (+ x 2))
(define (f x)
  (cond
    [(number? x) (real-f x)]
    [else (error 'f "not a number")]))
(f 'apple)
```

```
f: not a number
```

The **error** function triggers a run-time error

You don't have to program defensively in this course, but it sometimes helps to defend against your own mistakes!