Review

CS 350

Dr. Joseph Eremondi

Last updated: July 11, 2024

:EXPORT_FILE_NAME: pdf/slides008-review.pdf

:header-args:racket: :results code :lang plait

Overview

The Story So Far

- Functional Programming

The Story So Far

- Functional Programming
  - Immutable variables

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion
  - Data-types

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion
  - Data-types
  - Type-case

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion
  - Data-types
  - Type-case
- Interpreters

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion
  - Data-types
  - Type-case
- Interpreters
  - BNF

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion
  - Data-types
  - Type-case
- Interpreters
  - BNF
  - Abstract Syntax

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion
  - Data-types
  - Type-case
- Interpreters
  - BNF
  - Abstract Syntax
  - Parsing

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion
  - Data-types
  - Type-case
- Interpreters
  - BNF
  - Abstract Syntax
  - Parsing
  - Interpretation

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion
  - Data-types
  - Type-case
- Interpreters
  - BNF
  - Abstract Syntax
  - Parsing
  - Interpretation
  - Desugaring

The Story So Far

- Functional Programming
  - Immutable variables
  - Recursion
  - Data-types
  - Type-case
- Interpreters
  - BNF
  - Abstract Syntax
  - Parsing
  - Interpretation
  - Desugaring
  - Substitution

Functional Programming

How to evaluate functional programs

- Repeat until we have a value:

How to evaluate functional programs

- Repeat until we have a value:
  - Take all the functions defined with `define`

How to evaluate functional programs

- Repeat until we have a value:
  - Take all the functions defined with `define`
  - Replace them with their definitions, with arguments replacing parameters

How to evaluate functional programs

- Repeat until we have a value:
  - Take all the functions defined with `define`
  - Replace them with their definitions, with arguments replacing parameters
  - Simplify any `if`, `cond`, `type-case` etc.

The Languages We've Built

Curly

- Many different languages

Curly

- Many different languages
  - Gradually adding features

Curly

- Many different languages
  - Gradually adding features
  - I’ll give them names so we can tell them apart

Curly

- Many different languages
  - Gradually adding features
  - I'll give them names so we can tell them apart
- Each time we add a new feature, is a new language

Curly

- Many different languages
  - Gradually adding features
  - I'll give them names so we can tell them apart
- Each time we add a new feature, is a new language
- Called "curly because" we write with Curly-brackets

Curly

- Many different languages
  - Gradually adding features
  - I'll give them names so we can tell them apart
- Each time we add a new feature, is a new language
- Called "curly because" we write with Curly-brackets
- Write as S-expression strings

Curly

- Many different languages
  - Gradually adding features
  - I'll give them names so we can tell them apart
- Each time we add a new feature, is a new language
- Called "curly because" we write with Curly-brackets
- Write as S-expression strings
  - Racket backtick ‘ turns strings into S-expressions

Curly

- Many different languages
  - Gradually adding features
  - I'll give them names so we can tell them apart
- Each time we add a new feature, is a new language
- Called "curly because" we write with Curly-brackets
- Write as S-expression strings
  - Racket backtick ‘ turns strings into S-expressions
  - Separates different names/symbols and nests brackets

Curly

- Many different languages
  - Gradually adding features
  - I'll give them names so we can tell them apart
- Each time we add a new feature, is a new language
- Called "curly because" we write with Curly-brackets
- Write as S-expression strings
  - Racket backtick ‘ turns strings into S-expressions
  - Separates different names/symbols and nests brackets
  - Parse turns S-expressions into AST

Curly-Arith

- Just has addition, multiplication, and numbers

Curly-Arith

- Just has addition, multiplication, and numbers
- AST type `Expr`

Curly-Arith

- Just has addition, multiplication, and numbers
- AST type `Expr`
- Value type `Number`

Curly-Arith

- Just has addition, multiplication, and numbers
- AST type `Expr`
- Value type `Number`
- Pipeline:

Curly-Arith

- Just has addition, multiplication, and numbers
- AST type `Expr`
- Value type `Number`
- Pipeline:
  - String $\rightarrow_{\text{backtick}}$ S-Exp $\rightarrow_{\text{parse}}$ Expr $\rightarrow_{\text{interp}}$ Number

Curly-Cond

- Adds `ifo`

Curly-Cond

- Adds `if0`
  - Conditional expressions, branching depending on whether a value is `0`

Curly-Cond

- Adds `ifo`
  - Conditional expressions, branching depending on whether a value is `0`
  - Adds a constructor to `Expr`

Curly-Cond

- Adds `ifo`
  - Conditional expressions, branching depending on whether a value is `0`
  - Adds a constructor to `Expr`
  - Adds case to parser and interp

Curly-Sub

- Adds `{- x y}`

Curly-Sub

- Adds `{- x y}`
- `interp` unchanged from Curly-Cond

Curly-Sub

- Adds `{- x y}`
- `interp` unchanged from Curly-Cond
- Introduces an intermediate AST type `SurfExpr`

Curly-Sub

- Adds `{- x y}`
- `interp` unchanged from Curly-Cond
- Introduces an intermediate AST type `SurfExpr`
    - Expressions with syntactic sugar

Curly-Sub

- Adds `{- x y}`
- `interp` unchanged from Curly-Cond
- Introduces an intermediate AST type `SurfExpr`
  - Expressions with syntactic sugar
- *Desugaring* converts `SurfExpr` to `Expr`

Curly-Sub

- Adds `{- x y}`
- `interp` unchanged from Curly-Cond
- Introduces an intermediate AST type `SurfExpr`
  - Expressions with syntactic sugar
- *Desugaring* converts `SurfExpr` to `Expr`
  - Translate away certain features

Curly-Sub

- Adds `{- x y}`
- `interp` unchanged from Curly-Cond
- Introduces an intermediate AST type `SurfExpr`
  - Expressions with syntactic sugar
- *Desugaring* converts `SurfExpr` to `Expr`
  - Translate away certain features
  - Also called *elaboration*

Curly-Sub

- Adds `{- x y}`
- `interp` unchanged from Curly-Cond
- Introduces an intermediate AST type `SurfExpr`
  - Expressions with syntactic sugar
- *Desugaring* converts `SurfExpr` to `Expr`
  - Translate away certain features
  - Also called *elaboration*
- `{- x y}` is the same as `{+ x {* -1 y}}`

Curly-Sub

- Adds `{- x y}`
- `interp` unchanged from Curly-Cond
- Introduces an intermediate AST type `SurfExpr`
  - Expressions with syntactic sugar
- *Desugaring* converts `SurfExpr` to `Expr`
  - Translate away certain features
  - Also called *elaboration*
- `{- x y}` is the same as `{+ x {* -1 y}}`
- String $\rightarrow_{\text{backtick}}$ S-Exp $\rightarrow_{\text{parse}}$ SurfaceExpr $\rightarrow_{\textit{elab}}$ Expr $\rightarrow_{\text{interp}}$ Number

Curly-Fun

- Adds function definitions and function calls

Curly-Fun

- Adds function definitions and function calls
  - Single parameter functions, number in, number out

Curly-Fun

- Adds function definitions and function calls
  - Single parameter functions, number in, number out
- Functions have *parameters*

Curly-Fun

- Adds function definitions and function calls
  - Single parameter functions, number in, number out
- Functions have *parameters*
  - So we add variables to `Expr` and `SurfExpr`

Curly-Fun

- Adds function definitions and function calls
  - Single parameter functions, number in, number out
- Functions have *parameters*
  - So we add variables to `Expr` and `SurfExpr`
- Interpreter now parameterized by list of function definitions

Curly-Fun

- Adds function definitions and function calls
  - Single parameter functions, number in, number out
- Functions have *parameters*
  - So we add variables to `Expr` and `SurfExpr`
- Interpreter now parameterized by list of function definitions
  - Parsed separately

Curly-Fun

- Adds function definitions and function calls
  - Single parameter functions, number in, number out
- Functions have *parameters*
  - So we add variables to `Expr` and `SurfExpr`
- Interpreter now parameterized by list of function definitions
  - Parsed separately
- Function calls interpreted by *substitution*

Curly-Fun

- Adds function definitions and function calls
  - Single parameter functions, number in, number out
- Functions have *parameters*
  - So we add variables to `Expr` and `SurfExpr`
- Interpreter now parameterized by list of function definitions
  - Parsed separately
- Function calls interpreted by *substitution*
  - Replace variable with value of concrete argument

Where we’re going

Functional Programming Features

- Functions that take functions as arguments

Functional Programming Features

- Functions that take functions as arguments
- Functions that produce functions