Arithmetic: Our First Interpreter

CS 350

Dr. Joseph Eremondi

Last updated: July 9, 2024

Interpreters: Overview

The Church Turing Thesis

Alonzo Church

(lambda calculus)

Kurt Gödel

(general recursive functions)

Alan Turing

(Turing machines)

Turing Completeness

Turing Completeness

- The following can all simulate each other:

The Church Turing Thesis

Turing Completeness

- The following can all simulate each other:
  - Turing Machines

The Church Turing Thesis

Turing Completeness

- The following can all simulate each other:
  - Turing Machines
  - General-recursive functions

The Church Turing Thesis

Turing Completeness

- The following can all simulate each other:
    - Turing Machines
    - General-recursive functions
    - Lambda calculus (we'll see later)

Turing Completeness

- The following can all simulate each other:
  - Turing Machines
  - General-recursive functions
  - Lambda calculus (we'll see later)
- We call a programming language that can simulate a Turing machine *Turing Complete*

The Church Turing Thesis

Turing Completeness

- The following can all simulate each other:
  - Turing Machines
  - General-recursive functions
  - Lambda calculus (we'll see later)
- We call a programming language that can simulate a Turing machine *Turing Complete*
  - Any language with `while` loops or recursion is Turing Complete

Turing Completeness

- The following can all simulate each other:
  - Turing Machines
  - General-recursive functions
  - Lambda calculus (we'll see later)
- We call a programming language that can simulate a Turing machine *Turing Complete*
  - Any language with `while` loops or recursion is Turing Complete

All Turing Complete Languages can simulate each other

Turing Completeness and Interpreters

- You can write an interpreter for any language in any Turing-complete language

Turing Completeness and Interpreters

- You can write an interpreter for any language in any Turing-complete language
- The features of a language you're interpreting are *completely unrelated* to the features of the language the interpreter is written in

Turing Completeness and Interpreters

- You can write an interpreter for any language in any Turing-complete language
- The features of a language you're interpreting are *completely unrelated* to the features of the language the interpreter is written in
    - Sometimes you can piggyback on the implementation language features, but that's a matter what's *convenient*, not what's *possible*

Keeping it all straight

- The implementation language is NOT the language you're interpreting

Keeping it all straight

- The implementation language is NOT the language you're interpreting
- In this class, the implementation language is Racket/plait

Keeping it all straight

- The implementation language is NOT the language you're interpreting
- In this class, the implementation language is Racket/plait
- We'll write interpreters for a bunch of small languages

Keeping it all straight

- The implementation language is NOT the language you're interpreting
- In this class, the implementation language is Racket/plait
- We'll write interpreters for a bunch of small languages
    - We'll call them "Curly" because we write them with curly brackets

Keeping it all straight

- The implementation language is NOT the language you're interpreting
- In this class, the implementation language is Racket/plait
- We'll write interpreters for a bunch of small languages
  - We'll call them "Curly" because we write them with curly brackets
  - Write Curly programs in Racket files using quotation

General Form of an Interpreter

General Form of an Interpreter

```
(define (interp [e : Expr]
                [x : SomeContext]
                ...
                [y : OtherContext])
        : Value
  ....)
```

- `Expr` is the is the AST datatype for whatever language we're interpreting

General Form of an Interpreter

```
(define (interp [e : Expr]
                [x : SomeContext]
                ...
                [y : OtherContext])
        : Value
  ....)
```

- `Expr` is the is the AST datatype for whatever language we're interpreting
- What the context arguments and `Value` datatype are depend on the language

General Form of an Interpreter

```
(define (interp [e : Expr]
                [x : SomeContext]
                ...
                [y : OtherContext])
        : Value
  ....)
```

- `Expr` is the is the AST datatype for whatever language we're interpreting
- What the context arguments and `Value` datatype are depend on the language
  - Initially we have no context arguments, and `Value` is very simple

General Form of an Interpreter

```
(define (interp [e : Expr]
                [x : SomeContext]
                ...
                [y : OtherContext])
        : Value
  ....)
```

- `Expr` is the is the AST datatype for whatever language we're interpreting
- What the context arguments and `Value` datatype are depend on the language
    - Initially we have no context arguments, and `Value` is very simple
    - Will get more complicated as we go through the course

Our First Interpreter

Our First Interpreter

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
```

- Recursive function on structure of syntax

Our First Interpreter

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
```

- Recursive function on structure of syntax
    - Base cases are literals, translate directly into values

Our First Interpreter

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
```

- Recursive function on structure of syntax
  - Base cases are literals, translate directly into values
  - Recursive cases are operations

Our First Interpreter

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
```

- Recursive function on structure of syntax
  - Base cases are literals, translate directly into values
  - Recursive cases are operations
    - Interpret sub-expressions recursively

Our First Interpreter

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
```

- Recursive function on structure of syntax
  - Base cases are literals, translate directly into values
  - Recursive cases are operations
    - Interpret sub-expressions recursively
    - Combine according to value version of the operation

Our First Interpreter

- Interpreting arithmetic, so values are just plait `Number`

Our First Interpreter

- Interpreting arithmetic, so values are just plait `Number`

Our First Interpreter

- Interpreting arithmetic, so values are just plait `Number`

```
(define (interp [e : Expr] ) : Number
  (type-case Expr e
             [(NumLit n) n]
             [(Plus l r)
                (+ (interp l) (interp r))]
             [(Times l r)
                (* (interp l) (interp r))]))
(define (eval s-exp) (interp (parse s-exp)))
(eval `3)
(eval `{+ 2 5})
(eval `{+ {* 1 {+ 2 1}} {+ {* 3 4} {* 0 1000000}}})
```

Our First Interpreter

- Interpreting arithmetic, so values are just plait `Number`

```
(define (interp [e : Expr] ) : Number
  (type-case Expr e
             [(NumLit n) n]
             [(Plus l r)
                (+ (interp l) (interp r))]
             [(Times l r)
                (* (interp l) (interp r))]))
(define (eval s-exp) (interp (parse s-exp)))
(eval `3)
(eval `{+ 2 5})
(eval `{+ {* 1 {+ 2 1}} {+ {* 3 4} {* 0 1000000}}})
```

```
3
7
15
```

Adding features

- Need to update

Adding features

- Need to update
  - AST definition

Adding features

- Need to update
  - AST definition
  - Parser

Adding features

- Need to update
  - AST definition
  - Parser
  - Interpreter

Adding features

- Need to update
  - AST definition
  - Parser
  - Interpreter
- Example: `{if0 cond x y}`

Adding features

- Need to update
  - AST definition
  - Parser
  - Interpreter
- Example: `{if0 cond x y}`
  - Evaluates to `x` if `cond` evaluates to `0`

Adding features

- Need to update
  - AST definition
  - Parser
  - Interpreter
- Example: `{if0 cond x y}`
  - Evaluates to `x` if `cond` evaluates to `0`
  - Evaluates to `y` if `cond` evaluates to anything else

Updating the dataype

Updating the dataype

```
(define-type Expr
  (NumLit [n : Number])
  (Plus [left : Expr]
        [right : Expr])
  (Times [left : Expr]
         [right : Expr]))
  ;;NEW
  (If0 [test : Expr]
       [thenBranch : Expr]
       [elseBranch : Expr])
```

Updating the parser

Updating the parser

```
[(s-exp-match? `{if0 ANY ANY ANY} s)
 (If0 (parse (second (s-exp->list s)))
        (parse (third (s-exp->list s))
        (parse (fourth (s-exp->list s))]
```

Updating the interpreter

Updating the interpreter

```
(define (interp [e : Expr] ) : Number
  (type-case Expr e
             [(NumLit n) n]
             [(Plus l r)
                (+ (interp l) (interp r))]
             [(Times l r)
                (* (interp l) (interp r))])

             [(If0 test thenBranch elseBranch)
               (if (= 0 (interp test))
                 (interp thenBranch)
                 (interp elseBranch)
               )])
```