# Core

July 7, 2025

## Contents



## 1 Introduction

This is the 4th credit hour project for CS 421 students. Undergrads and 3 credit hour students are welcome to attempt this as well, but it will not affect your grade.

One common technique of modern compilers is to compile a high level language first to an intermediate representation (IR), which later can be compiled to LLVM or a specific CPU architecture. Not only does this simplify the compilation process, but it reduces the "space complexity" of the compilers from $\mathcal{O}(nm)$ to $\mathcal{O}(m+n)$, where $m$ is the number of languages and $n$ is the number of CPUs.

**The goal of this project is to write an interpreter for an intermediate representation for functional languages called *Core*.**

It will be somewhat similar to the MPs you have seen so far, but with much less starter code provided. You will get to make several design choices as you go that were constrained for you in

the MPs. You will also do a simple compilation stage: the data type we use to represent Core is simpler than the syntax we use to write it, so you will convert the syntax to Core as you parse.

The syntax and operation of Core is taken from *Implementing Functional Languages, a Tutorial* by Simon L Peyton Jones and David R Lester. We have copied many examples from it, and will refer to it as `IFLT` in our text. If you enjoy this project and want to go a *lot* further, we highly recommend this book. Note that it is written using Miranda; it was written before monads, Hash Array Mapped Tries (which made hash maps practical in functional languages), and monadic parser combinators were developed.

# 2 The Syntax

The formal specification of the syntax is given in figure 2. Here is a more informal discussion.

## 2.1 Declarations

There are three things you can declare in this language: variables, algebraic types, and functions.

### 2.1.1 Variables

A token that is not a keyword and begins with a lowercase letter can be considered a variable. We will consider the uppercase and lowercase letters, the digits, the single quote character, and the underscore as valid characters for a variable.

Examples:

- `x`
- `y123`
- `the_names`
- `x'`

Declaration just uses the equal sign as you might expect.

```
x = 10 ;
answer = 42
```

Note the use of semicolons as a declaration separator. (NOT terminator!) We want you to write a parser for this, but we didn't want you to have to figure out how to make an indentation sensitive parser. We might do that to next semester's cohort.

### 2.1.2 Algebraic Types

Algebraic type names have the same format as variables. Constructors have names like variables, but they will begin with an upper case letter. These are similar to Haskell. Type variables will be given by one or more `*` characters.

Here are some examples[1]:

[1] Many of these examples are taken from *Implementing Functional Languages, a Tutorial* by Simon L Peyton Jones and David R Lester.

```
colour ::= Red | Green | Blue ;
complex ::= Rect num num | Polar num num ;
numPair ::= MkNumPair num num ;
tree * ::= Leaf * | Branch (tree *) (tree *)
```

**We will not handle nested parameters.** So constructors like `Branch (tree (tree num))` are not valid.

All programs you receive as test cases will be validly typed, **and you are not responsible to type-check the program**. (If this project turns out to be too easy, next semester's cohort *will* be responsible!)

1. Pack

   Once a program has been type-checked, we will not need to know the names of the constructors. We will therefore compile them to a notation that indicates the number of the constructor and the number of arguments expected. We will use the keyword `Pack` when we refer to it in text, but in fact your interpreter most likely will not use that keyword. `Pack` has two arguments, the constructor number and its arity. So `Pack{4,3}` represents the 4th constructor with arity 3.

   You can see what the above code would look like in figure 1.

```
Red        = Pack{1,0}
Green      = Pack{2,0}
Blue       = Pack{3,0}

Rect       = Pack{4,2}
Polar      = Pack{5,2}

MkNumPair  = Pack{6,2}

Leaf       = Pack{7,1}
Branch     = Pack{8,1}
```

Figure 1: Constructor Representation using Pack

   As you can see, the constructor number starts from one and increments for each new constructor declared by the user. The second argument to `Pack` indicates the arity.

   Hint: There are several ways you can deal with converting the constructor names to integers. Here are two examples:

   - Pre-parse the code to find out a list of all the constructor names and then use that mapping in the actual parse.
   - Embed a state monad or pass around a counter and map to do this during the parse.

### 2.1.3 Functions

Function declarations may only appear at the top level. Local function definitions (e.g., within a `let`) are not allowed. (You can still define local functions using lambdas.) Pattern matching is not supported in function declarations — we will use `case` expressions for that. In the parlance of Core, top-level function declarations are called *supercombinators*.

```
twice f x = f (f x) ;
plus2 x = x + x
```

#### 2.1.4 Mutual Recursion

All top-level declarations are mutually recursive. See the discussion in `letrec` definition below to learn how to approach this. The first few test programs we give you will not take advantage of mutual recursion, so you can get started on this and get some initial work done and add the mutual recursion later if you want.

### 2.2 Let and Letrec

Like in Haskell, we can use `let` and `letrec`. Unlike Haskell, `let` is not recursive by default.

This version of `let` will compute the first `a` and then use that `a` to compute the second `a`. Each variable is created after its definition is evaluated. If you used `letrec` it would cause an infinite loop in the second definition of `a`. But the compiler would not allow this code because there are two definitions of `a`.

```
hypotsq x y =
  let a = x * x ;
      a = a + y * y
   in xx + yy + a
```

With `letrec`, all variables are created immediately, and all definitions have access to them. Thus you can get access to a variable defined later in the `letrec`.

```
fuzz a =
  letrec b = c + a ;
         d = b + a ;
         c = a + a
      in a + b + c + d
```

Calling this function as `fuzz 1` would yield $1 + 2 + 3 + 4 = 10$ where $c = 2, b = 3$, and $d = 4$.

**HINT: Take advantage of Haskell's laziness to make this work.** Suppose you are creating a new environment `env'` for use in the body of the `letrec`. The basic technique is to create a list of pairs using a list comprehension where the first element is the variable name and the second element is the result of calling `eval` *using* ***`env'`*** *as the environment argument.*

Once you have that, use `fromList` and `union` from the `Map` module to create `env'`.

Another note: Unlike Haskell, `let` and `letrec` do not allow for function declaration. If you need a function, use the lambda form.

### 2.3 Case

As mentioned above, we do function deconstruction using `case` expressions. An underscore indicates that we are ignoring the parameter to the case. Here is an example:

```
isRed c = case c of
    <1> -> True ;
    <2> -> False ;
    <3> -> False
```

```
;
depth t = case t of
    <1> _ -> 0 ;
    <2> t1 t2 -> 1 + max (depth t1) (depth t2)
```

### 2.4 Lambda

The syntax for lambda is almost identical to Haskell's. In Core, a lambda is a backslash followed by one or more parameter names, a dot, and then the body of the function.

```
plus = \a b . a + b
```

### 2.5 Binary Operations

Finally, we have binary operations. We will support plus, minus, times, (integer) division, the comparators, and, and or.

The list of binary operations along with their precedences is in figure 1. These are taken from `IFTL`. A 'none' associativity indicates that these cannot be chained. So for example `10 - 4 - 2` is not valid in Core.

Table 1: Binary Operations in Core

| Precedence | Associativity | Operators |
|---|---|---|
| 6 | Left | Application |
| 5 | Right | `*` |
| | None | `/` |
| 4 | Right | `+` |
| | None | `-` |
| 3 | None | `==`, `~=`, `>`, `>=`, `<`, `<=` |
| 2 | Right | `&` |
| 1 | Right | `\|` |

## 3 The Expression

Here are the types you will use. You may *not* modify the `Expr` type constructors that we have given you, but you may add constructors if you want. (You may want to do this, e.g., to represent constructor declarations in Core before converting them to `EPack`.) Your parse may not return any `Expr` that has added constructors.

We declare types for names and (they are strings now, but maybe one day you will want to support arrays or objects), and `IsRec` is a boolean to indicate if a `let` is recursive. This is a standard pattern used to avoid "boolean blindness".

Declarations for supercombinators hold the name, parameter list, and body.

Finally, a program is a list of supercombinator declarations.

```
import qualified Data.HashMap.Lazy as M

type Env = M.HashMap Name Int  -- You will probably need to change this
```

| | | |
|---|---|---|
| **Programs** | *program* $\rightarrow$ $\text{sc}_1$; ... ; $\text{sc}_n$ | ($n \geq 1$) |
| **Supercombinators** | *sc* $\rightarrow$ var $\text{var}_1$ ... $\text{var}_n$ = expr | ($n \geq 0$) |
| **Expressions** | *expr* $\rightarrow$ expr aexpr | (Application) |
| | \| $\text{expr}_1$ binop $\text{expr}_2$ | (Infix binary application) |
| | \| let defns in expr | (Local definitions) |
| | \| letrec defns in expr | (Local recursive definitions) |
| | \| case expr of alts | (Case expression) |
| | \| $\lambda$ $\text{var}_1$ ... $\text{var}_n$ . expr | (Lambda abstraction, $n \geq 1$) |
| **Atomic expressions** | *aexpr* $\rightarrow$ var | (Variable) |
| | \| num | (Number) |
| | \| Pack{num,num} | (Constructor) |
| | \| ( expr ) | (Parenthesised expression) |
| **Definitions** | *defns* $\rightarrow$ $\text{defn}_1$; ... ; $\text{defn}_n$ | ($n \geq 1$) |
| | *defn* $\rightarrow$ var = expr | |
| **Alternatives** | *alts* $\rightarrow$ $\text{alt}_1$; ... ; $\text{alt}_n$ | ($n \geq 1$) |
| | *alt* $\rightarrow$ <num> $\text{var}_1$ ... $\text{var}_n$ $\rightarrow$ expr | ($n \geq 0$) |
| **Binary operators** | *binop* $\rightarrow$ arithop \| relop \| boolop | |
| | *arithop* $\rightarrow$ + \| - \| * \| / | (Arithmetic) |
| | *relop* $\rightarrow$ < \| <= \| == \| ~= \| >= \| > | (Comparison) |
| | *boolop* $\rightarrow$ & \| \| | (Boolean) |
| **Variables** | *var* $\rightarrow$ alpha $\text{varch}_1$ ... $\text{varch}_n$ | ($n \geq 0$) |
| | *alpha* $\rightarrow$ an alphabetic character | |
| | *varch* $\rightarrow$ alpha \| digit \| _ | |
| **Numbers** | *num* $\rightarrow$ $\text{digit}_1$ ... $\text{digit}_n$ | ($n \geq 1$) |

Figure 2: BNF syntax for the Core language

```
type Name = String
type IsRec = Bool

data Expr = EVar Name                   -- Variables
          | ENum Int                    -- Numbers
          | EPack Int Int               -- Constructors
          | EAp Expr Expr               -- Applications
          | ELet
              IsRec                     -- is the let recursive?
              [(Name, Expr)]            -- Local Variable definitions
              Expr                      -- Body of the let
          | ECase                       -- Case Expressions
              Expr
              [(Int, [Name], Expr)]     -- Alternatives
          | ELam [Name] Expr            -- Functions (Lambdas)
   deriving (Eq, Show)

type Decl = (Name, [Name], Expr)        -- The name, parameter list, and body of a supercombinato

type Core = M.HashMap Name Decl         -- A core program is an environment of declarations
```

You will need a type to represent the values in your language. You can use `Expr` if you like (as in the Scheme MP) or create a new type to handle Values.

# 4 Your Work

## 4.1 Parsing

The first step is to be able to parse Core programs and generate correct `Expr` types. We recommend using parsec to write the parser, but you are free to write your own parser or use `happy` if you prefer LR parsing. Note that this grammar is not LL, so you will have to modify the grammar if you want to use recursive descent. **Note that we do no have negative numbers in the input.**

One batch of test cases will check for correct parsing.

## 4.2 Interpretation

Once you have parsed everything you can interpret the code. We don't have print statements! The test programs we give you will have a supercombinator `main` that takes no arguments. Run that, and the result should be a single integer, which you will return as a string.

## 4.3 Starting Files

Here is a github repository that has starting files. This will be announced separately.

The files you need to modify are `Parse.hs`, `Interp.hs`, and `Types.hs`. Be sure any modifications to `Type.hs` don't break anything in the other files. You may modify the other files, but you will only be allowed to hand in these three. (You will also hand in a `prompts.txt` if you use LLMs or have a partner.)

We will be putting sample programs in the `test` directory along with their expected outputs. We recommend you copy the `project` files to another directory or repository, or else set the distribution as an `upstream` reposotory so you can get updates.

### 4.4 Working Environment and LLMs

You are allowed to work with one other student who is currently taking the class if you like.

You are also allowed to use LLMs to assist you under these conditions:

- You can ask the LLM for help or advice about coding, and can use it to get help debugging your code. For example, if you are not sure how to get `letrec` to work, an LLM will be able to show you how to set up the recursion in Haskell. If you have written code and it's not working, you can use the LLM to debug it. While we are not going to forbid this, we recommend that you do not let the LLM write the "first pass" of your code.
- You may **not** simply paste the assignment or part of the assignment and have it write the code for you.
- **You must document the prompts you used.** Use the file `prompts.txt` for this. You will also use this file to specify who you worked with if you have a partner. Put their netid in the top, as in 'partner: mattox'.

### 4.5 Turning In

- We will set up a prairielearn testing suite in about a week.