a parser for the Core Language

Project

Part 1

main restrictions:

1) no explicit type declarations
2) use a simple uniform representation for constructors
3) pattern matching ➔ is transformed into simple case expressions

```
data Colour = Red | Green | Blue
data Complex = Rect  Num Num | Polar Num Num
data Tree a = Leaf a | Node (Tree a) (Tree a)
data NumPair = MkNumPair Num Num
```

```
Pack {tag, arity}
data Colour = Red | Green | Blue
Red = Pack{1,0}
Green = Pack{2,0}
Blue =Pack{3,0}
data Complex = Rect  Num Num | Polar Num Num
Rect = Pack{4,2}
Polar = Pack{5,2}
data Tree a = Leaf a | Node (Tree a) (Tree a)
Leaf = Pack{6,1}
Node =Pack{7,2}
```

type checking guarantees that different types are not mixed

```
data Colour = Red | Green | Blue
Red = Pack{1,0}
Green = Pack{2,0}
Blue =Pack{3,0}
data Complex = Rect  Num Num | Polar Num Num
Rect = Pack{1,2}
Polar = Pack{2,2}
data Tree a = Leaf a | Node (Tree a) (Tree a)
Leaf = Pack{1,1}
Node =Pack{2,2}
```

pattern matching is transformed in case statements:

```
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)
```

Programs	*program*	→	sc_1 ; ... ; sc_n	$n \geq 1$
Supercombinators	*sc*	→	*var* $var_1 \ldots var_n$ = *expr*	$n \geq 0$
Expressions	*expr*	→	*expr aexpr*	Application
		|	$expr_1$ *binop* $expr_2$	Infix binary application
		|	`let` *defns* `in` *expr*	Local definitions
		|	`letrec` *defns* `in` *expr*	Local recursive definitions
		|	`case` *expr* `of` *alts*	Case expression
		|	`\` $var_1 \ldots var_n$ `.` *expr*	Lambda abstraction ($n \geq 1$)
		|	*aexpr*	Atomic expression
	aexpr	→	*var*	Variable
		|	*num*	Number
		|	`Pack{`*num*`,`*num*`}`	Constructor
		|	`(` *expr* `)`	Parenthesised expression
Definitions	*defns*	→	$defn_1$; ... ; $defn_n$	$n \geq 1$
	defn	→	*var* = *expr*	
Alternatives	*alts*	→	alt_1 ; ... ; alt_n	$n \geq 1$
	alt	→	`<`*num*`>` $var_1 \ldots var_n$ `->` *expr*	$n \geq 0$
Binary operators	*binop*	→	*arithop* | *relop* | *boolop*	
	arithop	→	`+` | `-` | `*` | `/`	Arithmetic
	relop	→	`<` | `<=` | `==` | `~=` | `>=` | `>`	Comparison
	boolop	→	`&` | `|`	Boolean
Variables	*var*	→	*alpha* $varch_1 \ldots varch_n$	$n \geq 0$
	alpha	→	*an alphabetic character*	
	varch	→	*alpha* | *digit* | `_`	
Numbers	*num*	→	$digit_1 \ldots digit_n$	$n \geq 1$

the core language grammar

type Name = String

```
> data Expr a
>   =  EVar Name                    -- Variables
>    | ENum Int                     -- Numbers
>    | EConstr Int Int              -- Constructor tag arity
>    | EAp (Expr a) (Expr a)        -- Applications
>    | ELet                         -- Let(rec) expressions
>        IsRec                      --   boolean with True = recursive,
>        [(a, Expr a)]   [Def a]    --   Definitions
>        (Expr a)                   --   Body of let(rec)
>    | ECase                        -- Case expression
>        (Expr a)                   --   Expression to scrutinise
>        [Alter a]                  --   Alternatives
>    | ELam [a] (Expr a)            -- Lambda abstractions
>     deriving (Text)
```

Show

Finally, a Core-language program is just a list of supercombinator definitions:

```
> type Program a = [ScDefn a]
> type CoreProgram = Program Name
```

supercombinator=function

A supercombinator definition contains the name of the supercombinator, its arguments and its body:

```
> type ScDefn a = (Name, [a], Expr a)
> type CoreScDefn = ScDefn Name
```

The argument list might be empty, in the case of a supercombinator with no arguments.

We conclude with a small example. Consider the following small program.

```
main = double 21 ;
double x = x+x
```

This program is represented by the following Miranda expression, of type `coreProgram`:

```
[("main",   [],    (EAp (EVar "double") (ENum 21))),
 ("double", ["x"], (EAp (EAp (EVar "+") (EVar "x")) (EVar "x")))
]
```

```
type Def a = (a, Expr a)  -- for let and letrec

type Alter a = (Int, [a], Expr a)  -- for case
```

in Expr using IsRec you use the constructor ELet for modelling both let and letrec

```
data IsRec = NonRecursive | Recursive
    deriving Show
```

first part of the project

write parseExpr for the cases:
let, letrec, case, lambda e aexpr
you need also parseDef and parseAlt that parseExpr calls for
Def (let and letrec) and Alter (case)
and also parseAExpr for parsing AExpr

for the time being don’t treat the first two productions for Expr

the functions parseProg and parseScDef are given as examples

```
parseProg :: Parser (Program Name)
parseProg = do p <- parseScDef
                    do character ';'
                       ps <- parseProg
                       return (p:ps)
                    <|> return [p]


parseScDef :: Parser (ScDef Name)
parseScDef = do v     <- parseVar
                pf    <- many parseVar
                character ‘=‘
                body  <- parseExpr  -- call to parseExpr
                return (v, pf, body)
```

these are the functions that you should write:

parseExpr :: Parser (Expr Name)

parseAExpr :: Parser (Expr Name)

parseDef :: Parser (Def Name)

parseAlt :: Parser (Alter Name)

useful for checking the parseProg on your PC

for opening and reading the input file:

```
import System.IO
import Parser
import ParseProg

readF :: IO String
readF = do  inh <- openFile "input.txt" ReadMode
            prog <- hGetContents inh
            return prog
```

```
main :: IO (Program Name)
main = do inp <- readF
                 return (comp (parse parseProg inp))   --here you call parseProg

comp :: [(Program Name, Name)] -> Program Name
comp []             =  error "no parse"
comp [(e,[])]       = e
comp [(_,a)]        = error ("doesn't  use all input"++ a)
```

the main that is in the moodle is as follows:

```
module Main where
import Parse
import ParseProg
import MyPrinter    --pretty printer, contains functions iDisplay and pprProgram
main :: IO ()
main =  do  inp <- getContents;
print (parse parseProg inp)
putStr (iDisplay (pprProgram (comp (parse parseProg inp))) ++ "\n")

comp :: [(Program Name, Name)] -> Program Name
comp []       =  error "no parse«
comp [(e,"")] = e
comp [(_,a)]  = error ("doesn't  use all input"++ a)
```

comp signals possible errors

the input+output tests that are in the moodle for the Project, are in file «tests» that you find in the same folder of this file. There are 8 tests. The last 2 tests should be work correctly even with the partial parser of the first part of the assignment. The first 6 tests need the full parser.

The output of each test consists of 2 parts that correspond to the main. The first part shoud be ovious. The second part uses the pretty-printer (embedded in the exercise) that is written according to the suggestions in section 1.5 of the book «implementing functional languages» included in the same folder.