

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) no pattern matching \rightarrow 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{ 1,2}
```

pattern matching 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)

the core language
grammar

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

Figure 1.1: BNF syntax for the Core language

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
```

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

don't treat the first two productions for Expr (for the time being)

the productions for 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)

parseExpr :: Parser (Expr Name)

parseAExpr :: Parser (Expr Name)

parseDef :: Parser (Def Name)

parseAlt :: Parser (Alter Name)

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 <- readloop inh
```

```
    hClose 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)
```

```
readloop inh = do ineof <- hIsEOF inh
                  if ineof
                    then return []
                    else do
                        x <- hGetLine inh
                        xs <- readloop inh
                        return (x ++ xs)
```

test input :

$f = 3;$

$g\ x\ y = \text{let } z = x \text{ in } z;$

$h\ x = \text{case } (\text{let } y = x \text{ in } y) \text{ of}$

$\langle 1 \rangle \rightarrow 2$

$\langle 2 \rangle \rightarrow 5$