# Trabalho Final Haskell

UDESC UNIVERSIDADE DO ESTADO DE SANTA CATARINA
Ciência da Computação

O objetivo desse trabalho é a implementação de um interpretador para uma versão simplificada de Prolog, demonstrando conhecimento do paradigma lógico através do paradigma funcional. A parte central do programa é um algoritmo de unificação. O trabalho pode ser feito em duplas.

Considere os seguintes tipos:

```
data Term = Atom String
          | Var Name
          | Predicate Predicate
          deriving (Show, Eq)

type Name = (Int, String)

type Predicate = (String, [Term])

type Rule = (Predicate, [Predicate])

type Substitution = [(Name, Term)]
```

O programa deve ler um arquivo em Prolog contendo um banco de dados (uma sequência de `Rule`), e aceitar *queries* sobre esse banco. Recomenda-se a implementação do *parser* com a biblioteca Parsec. A seguir são sugeridas algumas funções a serem declaradas para guiar a implementação.

- Para a estrutura principal do programa:

```
main :: IO ()
run :: FilePath → IO ()
repl :: [Rule] → IO ()
```

- Para o *parser* de Prolog:

```
whitespace :: Parser ()
atom :: Parser Term
variable :: Parser Term
arguments :: Parser [Term]
predicate :: Parser Predicate
term :: Parser Term
rule :: Parser Rule
rules :: Parser [Rule]
```

- Para compor duas substituições:

```
compose :: Substitution → Substitution → Substitution
```

- Para aplicar uma substituição a um termo ou a um predicado, respectivamente:

```
substTerm :: Substitution → Term → Term
substPred :: Substitution → Predicate → Predicate
```

- Para unificar termos, a parte principal do algoritmo:

```
unifyTerm :: Term → Term → Maybe Substitution
unifyPredicate :: Predicate → Predicate → Maybe Substitution
unifyBody :: [Term] → [Term] → Maybe Substitution
```

- Para verificar se uma variável ocorre dentro de um termo, necessário na unificação:

```
occursCheck :: Term → Name → Bool
```

- Para criar novos nomes para variáveis em uma regra, a fim de evitar colisões; essa função deve apenas percorrer a regra recursivamente, aumentando o índice de cada variável dentro dela em 1:

```
freshen :: Rule → Rule
```

- Para procurar uma solução; a expressão `resolve goal rules` deve procurar, dentro das regras especificadas **(com nomes novos)**, quais substituições podem torná-los iguais:

```
resolve :: Predicate → [Rule] → [Substitution]
```

- Para verificar as condições de uma regra, a expressão resolveBody rules mgu body deve verificar recursivamente a variável body, garantindo que seja possível resolver cada predicado da lista com as regras **(com nomes novos)**, acumulando as substituições.

```
resolveBody :: Substitution → [Rule] → [Predicate] → [Substitution]
```

Considere as regras de unificação fornecidas nos slides, e, para o *parser*, a gramática abaixo:

```
name ::= lowercase alphanum*
atom ::= name | digit+
variable ::= uppercase alphanum*
arguments ::= “(“ (term (“,” term)*)? “)”
predicate ::= name arguments
term ::= predicate | atom | variable
rule ::= predicate (“:-” predicate (“,” predicate)*)? “.”
rules ::= rule+ eof
```

Banco de dados de exemplo:

```
likes(alice, pear).
likes(alice, orange).
likes(bob, pear).
likes(bob, X) :- yellow(X).
likes(charlie, X) :- likes(alice, X), likes(bob, X).
```