PFL_TP2_T12_G11

Tópico

O objetivo deste projeto é desenvolver um programa em Haskell capaz de compilar e executar uma pequena linguagem de programação imperativa, com expressões aritméticas e booleanas, e *statements* que consistem em *assign*, *if-then-else*, e *while loops*.

Group T12-G11

Nome	Número UP	Contribuição(%)
Eduardo Machado Teixeira de Sousa	up202103342	50
André Gonçalves Pinto	up202108856	50

Instalação e Execução

Para utilizar esta ferramenta, é necessária a instalação do Glasgow Haskell Compilation System. Além disso, é necessário o ficheiro main.hs, que deve ser carregado no ghci.

Descrição do Problema e da Implementação

Parte 1

A primeira parte do problema consiste em utilizar uma *stack* para executar as seguintes instruções low-level: push-n, add, mult, sub, true, false, eq, le, and, neg, fetch-x, store-x, noop, branch(c1, c2) and loop(c1, c2).

Para implementar estas regras, começamos por definir a Stack e fazer as regras mais simples da função `run`: push-n, add, mult, sub, true e false com a Stack e estas operações mais simples conseguimos perceber mais facilmente como iríamos proceder com o State. Definidos um datatype chamado StackTypes, e a Stack como uma lista de StackTypes. Definimos também o State como uma lista de tuplos (String, StackTypes).

```
data StackTypes =
  Int Integer | FF | TT
  deriving (Show, Eq, Ord)
type Stack = [StackTypes]
type State = [(String, StackTypes)]
```

De seguida definimos as regras de `run` para eq, le, and, neg, fetch-x, store-x e noop, tendo cuidado com erros como fetch-x de uma variável inexistente ou por exemplo a tentativa de uma igualdade entre um inteiro e um booleano, usando para tal a função error “Run-time error”.

Por último, nesta fase fizemos as regras para o branch(c1, c2) e loop (c1, c2). Depois de ter feito todas as funções mais simples, estas ficaram mais fáceis. Sendo que o branch verifica o topo da Stack, se este for true

executa c1, se for false executa c2. Para o loop(c1,c2) usamos a dica do enunciado de transformar o loop(c1,c2) em c1++[branch([c2,loop(c1,c2)],[noop])].

Alguns exemplos de testes do `run`:

```
ghci> testAssembler [Push 10,Store "i",Push 1,Store "fact",Loop [Push 1,Fetch
"i",Equ,Neg] [Fetch"i",Fetch "fact",Mult,Store "fact",Push 1,Fetch "i",Sub,Store
"i"]]
("","fact=3628800,i=1")

ghci> [Fals,Push 3,Tru,Store "var",Store "a", Store "someVar"]
[Fals,Push 3,Tru,Store "var",Store "a",Store "someVar"]
```

Parte 2

A segunda parte do problema consiste em ler um input numa pequena linguagem de programação imperativa e transformar esse input em instruções *low-level* implementadas na Parte 1.

Para tal, começámos por implementar as estruturas *Aexp*, *Bexp*, *Stm* e *Program*, correspondentes a expressões aritméticas, booleanas, *statements* e uma sequência de *statements*:

```
data Aexp =
  Val Integer | Var String | AddAexp Aexp Aexp | SubAexp Aexp Aexp | MultAexp Aexp
Aexp
  deriving Show

data Bexp =
  EquAexp Aexp Aexp | LeAexp Aexp Aexp | AndBexp Bexp Bexp | EquBexp Bexp Bexp |
NegBexp Bexp | TruB | FalsB
  deriving Show

data Stm =
  BranchS Bexp [Stm] [Stm] | LoopS Bexp [Stm] | AssignVar String Aexp
  deriving Show

type Program = [Stm]
```

Implementámos depois as funções `compA`, `compB`, `compile` e `compStm`, que recebem um argumento dos tipos `Aexp`, `Bexp`, `Program` e `Stm` respetivamente e o convertem em instruções *low-level* implementadas na Parte 1 (sendo que o `compile` recebe a lista de *statements* e devolve uma lista destas instruções, e utiliza as outras três funções para o fazer).

```
ghci> compile [AssignVar "x" (AddAexp (Val 1) (Val 2)), BranchS (EquAexp (Var "x")
(Val 3)) [AssignVar "y" (MultAexp (Val 2) (Val 4))] [AssignVar "y" (Val 2)]]
[Push 2,Push 1,Add,Store "x",Push 3,Fetch "x",Equ,Branch [Push 4,Push 2,Mult,Store
"y"] [Push 2,Store "y"]]
```

Uma vez feito este passo, falta passar de um *string* de *input* para uma lista de *statements*. Para fazer isto, implementámos a função `parse`. Esta chama primeiro as funções `stringToken2Token`, `mergeVarToks`, `mergeIntToks`, `parse_tokens` e `parse_tokens_aux`, que realizam os seguintes passos:

- **parse_tokens_aux**: converte um *string* numa lista de *tokens* e *strings*, sendo que os *strings* que restam correspondem aos nomes de variáveis e valores numéricos
- **parse_tokens**: usa o *output* de `parse_tokens_aux` e converte cada caratere de um nome de variável num *token* do tipo VarTok e cada dígito dos valores num *token* IntTok
- **mergeIntToks**: utiliza o *output* de `parse_tokens` e junta os IntToks adjacentes (por exemplo se houver IntTok 1 e IntTok 2 adjacentes, junta os dois num IntTok 12)
- **mergeVarToks**: faz o mesmo para os nomes das variáveis
- **stringToken2Token**: converte uma lista de *StringTokens* numa lista de *tokens*, mais prática para uso nos passos seguintes.

Por último, `parse` chama `parse_aux`, que tem a tarefa de receber a lista de tokens e devolver um *program* que possa ser compilado e executado. Para isso, lê recursivamente cada *statement*, e utiliza `parse_aexp` e `parse_bexp` para o *parsing* das expressões aritméticas e booleanas.

Alguns exemplos do *output* de `parse_aux` e de `parse`:

```
ghci> parse_aux [IfTok,OpenTok,IntTok 1,EqualATok,IntTok 0,AddTok,IntTok
1,EqualBTok,IntTok 2,AddTok,IntTok 1,EqualATok,IntTok 3,CloseTok,ThenTok,VarTok
"x",AssignTok,IntTok 1,BreakTok,ElseTok,VarTok "x",AssignTok,IntTok 2,BreakTok] []
[BranchS (EquBexp (EquAexp (Val 1) (AddAexp (Val 0) (Val 1))) (EquAexp (AddAexp
(Val 2) (Val 1)) (Val 3))) [AssignVar "x" (Val 1)] [AssignVar "x" (Val 2)]]

ghci> parse "i := 10; fact := 1; while (not(i == 1)) do (fact := fact * i; i := i
- 1;);"
[AssignVar "i" (Val 10),AssignVar "fact" (Val 1),LoopS (NegBexp (EquAexp (Var "i")
(Val 1))) [AssignVar "fact" (MultAexp (Var "fact") (Var "i")),AssignVar "i"
(SubAexp (Var "i") (Val 1))]]

ghci> parse "x := 42; if x <= 43 then (x := 33; x := x+1;) else x := 1;"
[AssignVar "x" (Val 42),BranchS (LeAexp (Var "x") (Val 43)) [AssignVar "x" (Val
33),AssignVar "x" (AddAexp (Var "x") (Val 1))] [AssignVar "x" (Val 1)]]
```

Conclusão

No geral achamos que concluímos com sucesso o trabalho prático da cadeira de PFL referente a Haskell. Conseguimos superar as dificuldades encontradas com sucesso. O ponto do trabalho em que sentimos mais dificuldades foi no *parse*, porém abordamos o problema de forma a resolver pequenos problemas sistematicamente para ser mais fácil encarar o problema principal no seu conjunto.

De uma forma geral consideramos o trabalho interessante, a aprendermos bastante sobre a linguagem Haskell e algumas bases sobre compiladores.