Escola de Ciências e Tecnologias

Disciplina: Linguagens de Programação

Professores: Teresa Gonçalves e João Leitão

Extensões Para o Interpretador de Cálculo Lambda

Segundo Trabalho Prático de Linguagens de Programação

Trabalho Elaborado Por:

Luís Polha Nº 20464

Marlene Oliveira Nº 25999

Ano Lectivo 2011 / 2012

Introdução

No âmbito da elaboração do segundo trabalho prático da disciplina de Linguagens de Programação foi-nos solicitada a elaboração de extensões para o interpretador de cálculo lambda elaborado para o primeiro trabalho prático desta mesma disciplina.

Uma vez que existiam ainda alguns problemas com o interpretador do trabalho anterior, começou-se por resolver os mesmos. Para que fosse possível corrigir os problemas encontrados, foi necessário remodelar as funções que realizavam a redução utilizando a estratégia de redução *call-by-name* e criar algumas funções que servem de auxiliares às que efectuam a redução. Neste momento o interpretador consegue reduzir correctamente todos os exemplos do primeiro trabalho prático bem como os exemplos extra criados, apesar de efectuar um passo de redução a mais no exemplo 9 fornecido no enunciado do primeiro trabalho prático.

Como foi referido anteriormente, para este trabalho prático pretende-se que sejam elaboradas extensões para o interpretador do primeiro trabalho. São estas extensões as seguintes:

- Construção seja;
- Expressões aritméticas;
- Construção se.

A construção *seja* será o equivalente à construção *let* do cálculo lambda. Esta construção tem prioridade inferior à da aplicação e da abstracção. No cálculo lambda a construção *let* não adiciona nada ao cálculo mas torna o mesmo mais "amigável", permitindo fazer declarações locais.

A extensão das expressões aritméticas permitirá ao interpretador resolver eventuais cálculos que resultem das reduções, ou seja, se o resultado final de uma redução for uma operação aritmética (no nosso caso, se for uma soma ou uma subtracção ou uma multiplicação ou uma divisão), o interpretador deverá resolver a mesma, devolvendo o valor numérico correspondente ao cálculo efectuado.

A construção *se* corresponde a uma condição. Assim, caso a expressão dada reduza para *verdade*, o termo lambda seleccionado será o que se encontrar antecedido por "então". Caso contrário, o termo escolhido será o que se encontrar antecedido por "senão". Os termos booleanos "verdade" e "falso" são também considerados parte desta construção, sendo que tal também acontece com o operador "=". Esta construção possui a mesma prioridade que a construção *seja.*

Implementação das Extensões do Interpretador

A primeira fase na implementação das extensões do nosso interpretador consistiu na alteração do analisador lexical do trabalho anterior. Tal alteração foi efectuada, como foi através do uso do analisador lexical Flex.

Os terminais acrescentados da linguagem do cálculo lambda e as alterações efectuadas são os seguintes:

- As variáveis passaram a poder apenas ser letras minúsculas. Tal deve-se a uma alteração na gramática que irá ser explicada posteriormente. Assim, a regra para as variáveis é a seguinte:

```
[a-z]                    {yylval = yytext[0]; return VARIAVEL;}
```

- Como existem operações aritméticas, os inteiros passam a ter de constar na lista de terminais. Assim foi adicionada a seguinte regra:

```
[0-9]+          {yylval = atoi(yytext); return INTEIRO;}
```

- De modo a que seja reconhecido o início de uma construção *se* foi adicionado o terminal "se". A regra adicionada foi a seguinte:

```
"se"            {return IF;}
```

- De modo a que pudessem ser reconhecidas pela gramática, as palavras "entao" e "senao" foram adicionados os seguintes terminais:

```
"entao"         {return THEN;}

"senao"         {return ELSE;}
```

- Para que fosse possível reconhecer os termos "verdade" e "falso", foram adicionados os seguintes terminais:

```
"verdade"       {return TRUE;}
"falso"         {return FALSE;}
```

- Como existem operações aritméticas, foram adicionados os terminais equivalentes aos operadores aritméticos. As regras adicionadas foram as seguintes:

```
"+"             {return '+';}
"-"             {return '-';}
"*"             {return '*';}
"/"             {return '/';}
```

- Para que as palavras “seja” e “em” pudessem ser reconhecidas pela gramática, foram adicionadas as seguintes regras:

  "em" {return IN;}
  "seja" {return LET;}

- Foi ainda adicionada a regra para o terminal correspondente ao operador “=”. A regra adicionada foi a seguinte:

  "=" {return '=';}

Tal como acontecia anteriormente, os terminais !, (,), . e espaço continuam presentes e os tabs (\t) continuam a ser ignorados.

A segunda fase na implementação das extensões do trabalho consistiu na alteração da gramática do primeiro trabalho prático de modo a que esta aceite as expressões que serão tratadas pelas extensões a implementar. Todas as alterações efectuadas à gramática do cálculo lambda foram elaboradas utilizando o Bison.

Foram adicionados os seguintes tokens à gramática:

- %token IF
- %token THEN
- %token ELSE
- %token TRUE
- %token FALSE
- %token LET
- %token IN
- %token INTEIRO

O espaço também passa a associar à esquerda tal como é indicado por:

- %left ' '

Seguidamente procedemos às alterações à gramática, acrescentando as seguintes regras:

const_let: LET ' ' VARIAVEL ' ' '=' ' ' termo IN ' ' termo ;

int: INTEIRO;

aritm: '+’ | '-' | '*' | '/' ;

bool: TRUE | FALSE ;

const_se: IF ' ' termo ' ' THEN ' ' termo ELSE ' ' termo ;

A regra “termo” da gramática também foi alterada de modo a que pudessem ser utilizadas as novas regras, que aceitarão as expressões que serão tratadas pelas extensões implementadas. Foi também adicionada uma regra para o operador ‘=’, que avalia se dois inteiros são iguais. Assim, a regra resultante é a seguinte:

termo: VARIAVEL | int | '=' | bool | aritm | const_let| const_se | abstraccao | aplicacao | termo_par ;

A regra *const_let* permite que sejam aceites expressões semelhantes a seja a = b em x, ou seja, permite aceitar expressões que utilizem a construção *let* do cáculo lambda.

A regra *int*, tal como o nome indica, permite aceitar expressões constituídas por inteiros.

A regra *aritm* permite que sejam aceites os operadores aritméticos necessários para realizar os eventuais cálculos necessários. Optámos por separar os inteiros dos operadores aritméticos de modo a simplificar a gramática. Assim, os inteiros possuem uma regra própria, que já foi mencionada anteriormente.

A regra *bool* permite que sejam aceites pela gramática os valores lógicos “verdade” e “falso”, que são constantes.

A regra *const_se* foi elaborada de modo que pudessem ser aceites as expressões que irão ser avaliadas pela extensão “se”, ou seja, se esta regra não existir, não será possível que expressões como “se verdade então x senão y” sejam aceites pela gramática.

Tal como acontecia no trabalho anterior, a informação aceite pela gramática será armazenada numa String global, que será posteriormente passada como argumento das funções que irão realizar a redução da mesma. As palavras “seja”, “em”, “se”, “então” e “senão” serão copiadas para a String global em letra maiúscula. Foi devido a esta alteração na gramática que as letras maiúsculas deixaram de poder ser usadas como nomes de variáveis. Isto permite que, durante as diversas verificações que ocorrem dentro das funções que implementam as extensões do trabalho, estas procurem pela inicial da palavra pretendida, evitando que ocorram conflitos com nomes de variáveis.

NOTA: Aquando da elaboração da gramática foram detectados problemas causados pela inclusão de um espaço entre um termo e um terminal da gramática. Este espaço fazia com que a gramática considerasse toda a informação a seguir ao espaço como parte de uma aplicação (iniciada no termo imediatamente antes do referido espaço). O espaço que causava os referidos problemas foi então removido das regras em cujas causava problemas. Assim, quando se inserem expressões como “seja a = !b.b em i a”, o espaço entre “b” e “em” não deve ser inserido. Note-se que isto também ocorre nas expressões que utilizam a construção “se”. Expressões semelhantes a “se verdade então !a.a senão !b.b” deverão também ser inseridas sem que sejam incluídos o espaço entre “!a.a” e “senão”. Assume-se que não podem ser inseridas aplicações ou abstracções a seguir à palavra “se”, pelo que a inserção de informação ocorre normalmente. Apesar de todos estes inconvenientes, a expressão inserida será armazenada correctamente na String global. Uma solução alternativa passaria pela

remoção do caractere espaço da gramática. Porém, após se ter tentado essa opção, verificou-se que levantava mais problemas que a solução escolhida.

A terceira fase de actualização do nosso interpretador de cálculo lambda passou pela implementação das extensões indicadas no enunciado deste trabalho prático (construção seja, construção se e expressões aritméticas).

A função que implementa a extensão da construção seja denomina-se *let* e recebe como argumentos a String a ser tratada e a String de destino. Assume-se que a expressão que é fornecida como argumento desta função já tem a palavra "seja" no início da String. Assim, esta função começa por avançar na String de modo a "saltar" a palavra "seja" e o espaço que se encontra imediatamente a seguir à referida palavra. A variável a ser substituída é então copiada para um char temporário. Seguidamente, a função avança na String de modo a que seja ignorada a variável, o espaço a seguir à variável, o sinal de igual e o outro espaço que se encontra imediatamente a seguir ao sinal de igual. De seguida, a função copia o termo substituto para uma String temporária. A função volta então a avançar na string de modo a que possa ser ignorado o termo substituto, o espaço entre o termo e a palavra "em" e o espaço que se encontra imediatamente a seguir à palavra "em". Note-se que o termo onde pretendemos efectuar a substituição da variável é a única coisa que resta na String. Antes de efectuar a substituição, a função irá verificar se o termo onde pretendemos efectuar a substituição é também um let, ou seja, se começa com a palavra "seja". Caso isto se verifique, a função é executada recursivamente com os argumentos: o termo resultante e uma String global que irá armazenar o resultado. Caso contrário, é efectuada a substituição e retornada a String final. Quando todos os lets se encontrarem resolvidos, é retornada a String final.

A função que implementa a extensão da construção se denomina-se se e recebe como argumentos uma String correspondente à expressão a avaliar e uma String correspondente à String para onde pretendemos retornar o resultado da avaliação da expressão original. Esta função utiliza uma outra função auxiliar que avalia a expressão dada como argumento ao se.

A função auxiliar denomina-se avalia_se e recebe como argumento a String a avaliar. A função de avaliação começa por verificar se o primeiro símbolo da expressão fornecida como argumento corresponde ao sinal de igual. Caso isto se verifique, a função avança na String de modo a que seja possível ignorar o '=' e o espaço que se encontra imediatamente a seguir ao mesmo. Seguidamente, a função copia o primeiro argumento da expressão para uma String temporária. Caso este argumento corresponda a "verdade" ou "falso", a função atribui o valor 'v' ou 'f' a um char temporário. Caso este argumento corresponda a um valor numérico, a função converte o argumento para um inteiro. A função volta então a avançar na String de modo a que seja ignorado o primeiro argumento da expressão e o espaço que se encontra imediatamente a seguir. O segundo argumento da expressão é então copiado para uma String temporária e segue-se uma avaliação da mesma semelhante à que ocorreu para o primeiro argumento. Depois de terem sido avaliados os argumentos, estes são comparados. No caso de os argumentos corresponderem a uma das palavras "verdade" ou "falso", comparam-se os chars temporários aos quais foram atribuídos valores 'v' ou 'f' de acordo com a avaliação efectuada. Caso os argumentos sejam valores numéricos, estes são comparados. Existe

também uma condição que verifica se a String inserida como argumento corresponde à palavra “verdade”.

Esta função auxiliar retorna 1 caso a expressão do argumento seja avaliada para verdade (caso os argumentos sejam iguais ou a String inserida corresponda a “verdade”) e retorna 0 caso a expressão do argumento seja avaliada para falso (caso a expressão inserida corresponda a “falso” ou os argumentos sejam diferentes). O retorno desta função será utilizado na função *se* de modo a que esta possa determinar qual é a expressão a ser retornada (a expressão que se encontra a seguir à palavra “então” caso o resultado seja 1 ou a expressão que se encontra a seguir à palavra “senão” caso o resultado seja 0). A função *se* recebe como argumentos a expressão a avaliar e a String para onde será retornada a expressão resultante da avaliação. Esta função começa por verificar se a String começa pela palavra “se”. Caso isto não se verifique, é retornada a expressão original. Caso a expressão fornecida como argumento possua “se” no início, esta função começa por avançar na String de modo a que seja ignorada a palavra “se” e o espaço imediatamente a seguir. De seguida, a função copia para um temporário a expressão que se encontra imediatamente a seguir à palavra “se”. Esta expressão será avaliada pela função auxiliar referida anteriormente. A expressão retornada dependerá, tal como foi dito anteriormente, da avaliação efectuada pela função auxiliar.

A extensão das expressões aritméticas foi implementada com recurso às seguintes funções:

- *Check_opr*: função auxiliar que verifica se existem operadores aritméticos na expressão, ou seja, verifica se podem ser efectuados cálculos. Recebe como argumento a expressão a avaliar e retorna 1 caso existam operadores aritméticos na expressão e 0 caso estes não se encontrem na mesma.
- *Is_operation*: função auxiliar que verifica se o primeiro símbolo de uma expressão corresponde a um operador aritmético. Esta função recebe como argumento a expressão a avaliar e retorna 1 caso o primeiro símbolo da mesma seja um operador aritmético e 0 caso este não o seja.
- *Find_op*: função auxiliar que retorna a posição da operação aritmética mais interior. Esta função começa por usar a função *check_opr* para verificar se existem operadores aritméticos na expressão. Caso isto se verifique e exista apenas um operador, a função percorre a expressão fornecida como argumento e retorna a posição do operador aritmético. Caso o número de operadores seja superior a 1, a função irá procurar o operador que se encontra mais interior na expressão e retorna a posição do mesmo. Caso não existam operadores, a função retorna o valor inicial da posição - zero.
- *Faz_conta*: função auxiliar que efectua uma operação aritmética e retorna o resultado do mesmo. Esta função começa por verificar qual é o operador aritmético e, de seguida, converte os argumentos da expressão aritmética em inteiros e efectua o cálculo. Caso a operação aritmética seja uma divisão, o cálculo só será efectuado se o segundo argumento for diferente de zero. Retorna o resultado do cálculo.
- *Aritm*: função que implementa a extensão com recurso às funções auxiliares. Esta função trata a String fornecida como argumento e efectua eventuais

operações aritméticas resultantes e retorna a expressão resultante da avaliação efectuada. A função aritm começa por verificar se a expressão é uma operação aritmética (com recurso à função auxiliar *is_operation*). Caso isto não se verifique, é retornada a expressão original. Caso contrário a função irá verificar se a expressão é constituída por um operador aritmético e apenas um inteiro. Caso isto se verifique, é retornada a expressão original. Esta verificação é também efectuada para o caso em que existem parêntesis. Seguidamente, a função verifica se a expressão constitui uma operação simples, ou seja, se tem apenas um operador aritmético e dois inteiros. Caso isto se verifique, é efectuado o cálculo e retornado o resultado. Caso existam vários operadores, a função efectua o tratamento da expressão e efectua o cálculo recursivamente. No final é retornado o resultado do cálculo.

Para estas funções funcionarem com a função para a redução feita no trabalho anterior de nome redux, foram feitas algumas alterações nesta função e criadas novas funções auxiliares de maneira a que o programa funcione correctamente. As funções auxiliares da função redux são as seguintes:

- char *tira_1esp(char *c, char *c): função auxiliar que remove algum espaço(' ') que esteja no inicio da String fornecida e devolve uma String igual mas sem o espaço no inicio se o tiver.
- int ultm_esp(char *c): função auxiliar que indica o local na String fornecida onde começa o primeiro termo a aplicar na abstracção lambda. Esta função usa a função fim_abst(c).
- int fim_termo(char *c): função auxiliar que indica a localização, em forma de um inteiro, onde termina o termo a aplicar na lambda abstracção. Esta função usa a função ultm_esp para descobrir o começo do termo e depois vai descobrir onde acaba, isto é onde começa um espaço em branco. Também resolve no caso de o termo começar com um parêntesis, encontrando o parêntesis equivalente.
- char *variaveis(char *c. char *dest): função auxiliar que armazena as variáveis lambda que se encontram na String fornecida, sendo estas guardadas, pela ordem que aparecem, na String dest.
- int strcountc(char *s, char ch) - Função auxiliar que indica o número de vezes que o carácter ch aparece na string s.
- int check_var(char *c, char *v): função auxiliar que verifica se a String c contêm as variáveis armazenadas na String v. Se tiver pelo menos uma das variáveis de v devolve 0. Caso contrário devolve 1.
- int fim_abst(char *c): função auxiliar que indica a localização na String c onde acaba a abstracção lambda.
- int abstr(char *c): função auxiliar que indica a localização na String c onde começa a lambda abstracção com maior prioridade.

- char *abstracao(char *c, char *dest): função auxiliar que usando as funções abstr e fim_abst devolve a substring dentro de c, que é equivalente à lambda abstracção com maior prioridade, para a String dest.
- char *ult_termo(char *c, char *dest): função auxiliar que usando as funções ultm_esp e fim_termo devolve a substring de c, para a String dest, equivalente ao primeiro termo a seguir às abstracções lambda.
- auxiliar int c_redux(char *c) : função auxiliar que indica se é possível reduzir mais uma expressão. Retorna 0 se não for possível reduzir mais a expressão e 1 se é possível fazer mais uma redução.
- char *aplica(char *abst, char *subs, char *dest): função auxiliar que dado uma String com uma abstracção(abst), e uma String equivalente ao termo(subs) para fazer a substituição na abstracção, devolve o resultado dessa substituição para a String dest.

A função redux também sofreu alterações de modo a permitir interagir com as funções das extensões e agir de modo recursivo. As modificações mais importantes na função redux são a detecção, logo ao início, se a String fornecida é diferente do formato normal de uma lambda abstracção. Se for diferente faz uma pesquisa para descobrir qual a extensão correcta a utilizar antes de fazer nova redução caso exista alguma para fazer. Se o inicio da String for uma expressão lambda a função c_redux verifica se é possível reduzir mais a expressão, caso não seja possível acaba aí a função e retorna a String fornecida sem alteração. Caso seja possível reduzir, a função redux vai efectuar todos os passos necessários para reduzir a expressão tal efectuado no primeiro trabalho.

Para além disto foram elaborados os seguintes exemplos:

Exemplo	***Retorno do Programa***
* 2 (+ 2 3)	10
+ 2 (+ 1 (+ 2 (+ 2 5)))	12
+ 3 (* 2 (/ 100 (* 10 5)))	7
seja z = a em (!x.x z) y	y a
seja x = y em seja b = a em b x	a y
seja z = a em seja b = (!x.x z) y em a b	a y a
se verdade entao x senao y	x
se verdade entao y senao !x.x z	y
se = 20 100 entao !x.x senao !y.y	!y.y
se falso entao !x.x senao se = 0 0 entao x senao y	x

Dos exemplos fornecidos no enunciado, o nosso interpretador apenas não resolve um deles – o último exemplo da construção “se” (dá erro e pára a meio). Uma vez que as funções que implementam o let e o se estão a funcionar correctamente nos outros casos, a única conclusão a que chegámos foi que o problema estava relacionado com alguma das funções intermédias que efectuam a redução do termo lambda.

Bibliografia

Livros:

- Mitchell, John C. 2003. *Concepts in Programming Languages*. Cambridge University Press.

Fontes da Internet:

- *A Tutorial Introduction to the Lambda Calculus* (Online). (Acedido em 17-05-2012). Disponível em: http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf
- *Lambda Calculus* (Online). (Acedido em 19-05-2012). Disponível em: http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Ch/
- *Lambda Calculus* (Online). (Acedido em 21-05-2012). Disponível em: http://www.mactech.com/articles/mactech/Vol.07/07.05/LambdaCalculus/index.html
- *Untyped Lambda Calculus* (Online). (Acedido em 11-06-2012). Disponível em: http://drona.csa.iisc.ernet.in/~deepakd/pav/lecture-notes.pdf
- *Lambda Calculus* (Online). (Acedido em: 07-06-2012). Disponível em: http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Ch/01.Calc.html

Outros:

- Slides das aulas teóricas