UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO

LINGUAGENS DE PROGRAMAÇÃO I – CK0115 2023.2 T02

PROFESSOR: PABLO MAYCKON SILVA FARIAS

TRABALHO: IMPLEMENTAÇÃO DE TIPAGEM (VERSÃO 1: 22/09/2023)

Sumário

1 Resumo

Você deverá escrever um programa que, seguindo as especificações abaixo, leia um termo do Cálculo λ Simplesmente Tipado e então imprima o tipo desse termo, como no seguinte exemplo:

```
$ ocamlopt main.ml
$ ./a.out
lambda cond : Bool . if cond then suc else pred endif end
( Bool -> ( Nat -> Nat ) )
```

A sessão acima ilustra aspectos importantes do trabalho, como o fato de que o programa não deve emitir mensagens de comunicação com o usuário, e também o de que a sintaxe esperada pelo programa tem pequenas diferenças em relação àquela utilizada nas aulas da disciplina. Esses aspectos serão detalhados mais à frente.

Este trabalho é **individual**: não deve haver compartilhamento de soluções entre estudantes, nem mesmo de soluções em estágio preliminar de desenvolvimento. Em caso de dificuldades, converse com o professor.

2 Requisitos sobre a Linguagem de Programação

Este trabalho possui alguns propósitos:

1. Contribuir para a fixação do conteúdo das aulas sobre a modelagem teórica de linguagens de programação, demandando esse conteúdo em instrumento avaliativo da disciplina;

2. Relacionar esse conteúdo teórico com seus aspectos práticos, demandando o conteúdo através de um trabalho de implementação;
3. Contribuir para o aprendizado de novas linguagens de programação, permitindo que o trabalho seja implementado em diferentes linguagens.

Assim sendo, conforme o último item acima, será permitido que cada estudante escolha a linguagem de programação na qual implementará o trabalho, dentro das estipulações a seguir:

1. As linguagens escolhidas pelos(as) estudantes da turma serão registradas pelo professor em uma planilha compartilhada, facilitando o acompanhamento das escolhas por toda a turma. Apenas as escolhas registradas nesse documento estarão confirmadas.
2. Cada estudante deverá manifestar a escolha da linguagem pretendida através de **e-mail enviado ao professor**. A ordem de chegada dos e-mails decidirá eventuais desempates, quando houver restrição sobre a linguagem. A aula da sexta-feira 22/09/2023 também poderá ser utilizada para manifestação da escolha da linguagem de programação.
3. Os(as) estudantes poderão combinar, entre si, trocas entre as linguagens já escolhidas (em caso de necessidade), mas tais trocas também precisam aderir às exigências acima, ou seja, precisam ser informadas ao professor por e-mail, com cópia para todos os envolvidos, e só estarão confirmadas após registradas na planilha compartilhada.
4. Qualquer estudante poderá utilizar a linguagem OCaml. Esta regra tem como objetivo reforçar o aprendizado da principal linguagem de programação apresentada na disciplina.
5. Quem não tiver se manifestado a respeito da linguagem de programação escolhida **até a quinta-feira 19/10/2023** terá que fazer o trabalho em OCaml.
6. Quem preferir realizar o trabalho noutra linguagem terá chance de fazê-lo. O objetivo com isso é dar, a quem porventura já tenha interesse em alguma linguagem específica, uma oportunidade de aprender um pouco dessa linguagem, utilizando-a na pequena implementação deste trabalho. Entretanto, deverão ser seguidas as seguintes regras:
   (a) Cada uma destas 3 linguagens poderá ser escolhida por no máximo uma pessoa: Python, C e Java. O motivo desta regra é que estas são linguagens tipicamente ensinadas em disciplinas de formação básica em programação, e que portanto provavelmente já são de conhecimento dos(as) estudantes da turma. Como o objetivo em questão do trabalho é contribuir para o aprendizado de novas linguagens, esta regra desestimula a utilização de linguagens já conhecidas.
   (b) Linguagens diferentes de Python, C e Java poderão ser escolhidas por múltiplos estudantes (lembrando que o objetivo é proporcionar oportunidade de aprendizado de novas linguagens).
   (c) A linguagem escolhida pelo(a) estudante somente será aceita pelo professor se ele conseguir acesso ao compilador ou interpretador sem necessidade de um esforço desproporcional. Assim, por exemplo, linguagens limitadas a um sistema operacional específico poderão ser rejeitadas (o professor necessariamente utilizará o Linux para realizar os testes), assim como linguagens que dependam de frameworks disponíveis mediante pagamento, cadastro, etc.

3 Submissão das Soluções

Este trabalho será cadastrado como tarefa na turma virtual do SIGAA e as soluções deverão ser por lá submetidas.

Somente serão consideradas válidas as submissões realizadas após a linguagem escolhida pelo(a) estudante ser registrada pelo professor na planilha compartilhada.

O professor possui testes automáticos para auxiliar na correção e procurará dar retorno sobre cada submissão **até o dia útil seguinte**, de forma que o(a) estudante possa ver o retorno, realizar correções e submeter nova solução, desde que dentro do prazo de submissão.

Atenção: naturalmente, para se beneficiar da oportunidade de receber retorno do professor e realizar submissões corrigidas, é importante que o(a) estudante não deixe para submeter a solução apenas no fim do prazo.

4 Entrada e Saída do Programa

Considere a sessão abaixo:

```
$ ocamlopt main.ml
$ ./a.out
lambda num : Nat . lambda cond : Bool . ehzero end end
( Nat -> ( Bool -> ( Nat -> Bool ) ) )
```

A primeira linha acima é naturalmente a compilação do código-fonte, a segunda é a chamada do programa, a terceira mostra a entrada digitada pelo usuário e a quarta a saída do programa. Observe que o programa não emitiu mensagens como `"Digite o termo a ser analisado:"` ou `"O tipo do termo digitado é:"`; esse comportamento é intencional: na avaliação realizada pelo professor, o seu programa será submetido a testes automáticos e, para facilitar esse processo, este trabalho determina que o programa deve ter a interface "seca" ilustrada acima.

Mais precisamente, a entrada consistirá em uma única linha, no formato `TERMO`⇓, ou seja, um termo imediatamente seguido de uma quebra de linha, aqui representada por "⇓". De forma semelhante, a saída consistirá numa única linha, no formato `TIPO`⇓, ou seja, um tipo imediatamente seguido de uma quebra de linha.

Há, entretanto, 2 situações excepcionais, nas quais o programa deve imprimir, ao invés de um tipo, um caractere indicador da circunstância excepcional:

1. Se a entrada não for exatamente um termo seguido do fim da linha, então deve ser impressa uma exclamação. Nesta execução, por exemplo,

   ```
   ( suc 1)
   !
   ```

   está faltando um espaço entre o `1` e o parêntese final (conforme detalhado mais à frente, a entrada deve possuir exatamente um espaço em branco entre cada sequência de não-brancos – o que também facilita a implementação em OCaml, que pode facilmente utilizar a função `String.split_on_char` para separar os elementos da entrada).

   Já a entrada abaixo começa com um termo válido, que é uma abstração λ, mas ele é seguido por um espaço em branco e `true`, e por isso a entrada é inválida (observe que, segundo a sintaxe deste trabalho, o termo abaixo não é uma aplicação, pois não possui parênteses ao redor dos dois termos em questão):

```
lambda teste : Bool . teste end true
!
```

2. Se a entrada consiste em exatamente um termo, mas ele não é bem-tipado no contexto vazio, então deve ser impresso um hífen ao invés de um tipo, como em

   ```
   ( lambda b : Bool . b end 1 )
   -
   ```

   e em

   ```
   ( suc varlivre )
   -
   ```

Importante: observe que, conforme explicado acima, o comportamento esperado para o programa, quando é digitada uma entrada que não consiste exatamente em um termo, ou quando o termo não possui um tipo no contexto vazio, é imprimir uma linha com apenas um caractere, ao passo que o mais intuitivo, do ponto de vista de um usuário humano, seria uma mensagem explicando o problema. O motivo para a especificação do trabalho ser esta acima é o mesmo de antes: a saída acima facilita os testes automáticos (imagine o problema de gerenciar de forma automatizada diferentes programas que emitissem diferentes mensagens de erro). Porém, cada estudante pode ter interesse em gerar mensagens de erro explicativas, especialmente durante a elaboração do código, para entender melhor como o programa está se comportando "internamente". Nesse caso, há duas alternativas simples:

1. Você pode escrever inicialmente um programa que imprima na tela mensagens de erro significativas, e depois remover essas mensagens antes de submeter a solução; é necessário, porém, avaliar se essa adaptação final vale a pena, tendo em vista a possibilidade de ressubmissões.
2. Alternativamente, o seu programa pode, inclusive na versão submetida como solução, imprimir as mensagens desejadas num arquivo como `registro.txt`, atendendo então à interface solicitada no trabalho, mas ainda assim abrindo um canal alternativo para o programa fornecer informações úteis a respeito do seu funcionamento ou da entrada.

5 Dicas

Uma maneira de fazer este trabalho é a seguinte:

1. O programa pode começar transformando a linha lida do usuário, que é uma *string*, numa sequência de símbolos (análise léxica). Isso pode ser simples de fazer, pois todos os símbolos estarão separados por um espaço em branco. Assim, por exemplo, uma linha como "`lambda b : Bool . b end`" pode ser transformada na seguinte sequência de símbolos: "`lambda`", "`b`", "`:`", "`Bool`", "`.`", "`b`" e "`end`".

   A maneira como os símbolos serão armazenados pode ser escolhida por você. Uma possibilidade é definir um tipo `simbolo` (em OCaml, pode ser usada uma variante com um caso para cada tipo de símbolo), outra é usar simplesmente uma *string*.

   Importante: consulte a gramática concreta da linguagem deste trabalho, definida mais à frente.

2. Uma vez que a entrada tenha sido convertida numa sequência de símbolos, essa sequência pode então ser consumida para gerar um `termo`, isto é, um valor de um tipo `termo` definido no seu programa (em OCaml, por exemplo, uma boa representação é usar uma variante,

como um construtor para cada forma possível de termo). Observe que, para definir um tipo `termo`, você provavelmente precisará definir um tipo `tipo` antes, representando os tipos do cálculo λ. Observe que as definições de termos e tipos podem e devem seguir de perto a forma das gramáticas abstratas para termos e tipos apresentadas em sala para o cálculo λ simplesmente tipado.

3. Uma vez que a entrada tenha sido convertida num `termo`, ele pode então ser fornecido a uma função que compute o seu tipo, a partir do contexto vazio. Novamente, o funcionamento dessa função pode e deve seguir de perto as regras apresentadas em sala para a relação de tipagem. Observe também que, apesar de o contexto inicial ser vazio, a análise de tipo dos subtermos pode precisar envolver contextos não-vazios, logo a sua função deve receber também um contexto como entrada.

4. Uma vez computado o tipo do termo, basta convertê-lo numa *string* e imprimi-lo. Observe também que o termo lido pode não ter um tipo no contexto vazio, ou pode ser que a própria entrada não consista em exatamente um termo; tais situações excepcionais também devem ser tratadas para que a saída adequada seja impressa na tela.

6 Gramática da Sintaxe Concreta dos Termos

A seguir é especificada uma gramática para a *sintaxe concreta* dos termos do Cálculo λ Simplesmente Tipado deste trabalho. Em síntese, as diferenças em relação à sintaxe escrita em sala são as seguintes (observe que as mudanças basicamente tornam a sintaxe mais facilmente legível e removem ambiguidades):

1. Os if's são terminados por "`endif`", como em "`if true then 1 else 0 endif`".
2. Ao invés de "`eh_zero`", nós escrevemos "`ehzero`".
3. Ao invés de "`T -> U`", nós escrevemos "`( T -> U )`".
4. Ao invés de "λ`x:T.t`", nós escrevemos "`lambda x : T . t end`".
5. Ao invés de "`t u`", nós escrevemos "`( t u )`".

Na gramática abaixo, nós utilizamos as seguintes convenções:

1. Para tornar a apresentação menos poluída, ao invés de "`<símbolo>`", nós escrevemos "`SÍMBOLO`".
2. Para maior clareza, cada espaço em branco que deve ocorrer na linguagem objeto (isto é, na linguagem do cálculo λ simplesmente tipado) está denotado pelo símbolo "␣". Isso também significa que espaços em branco presentes nas regras abaixo, como em

   `DÍGITO_NÃO_ZERO SEQ_DÍGITOS`

   existem apenas pela legibilidade da gramática, não representando espaços na linguagem objeto.

```
TERMO ::=   true
          | false
          | if␣TERMO␣then␣TERMO␣else␣TERMO␣endif
          | NÚMERO
```

```
          | suc
          | pred
          | ehzero
          | VARIÁVEL
          | (␣TERMO␣TERMO␣)
          | lambda␣VARIÁVEL␣:␣TIPO␣.␣TERMO␣end
```

```
NÚMERO ::= DÍGITO | DÍGITO_NÃO_ZERO SEQ_DÍGITOS

DÍGITO ::= 0 | DÍGITO_NÃO_ZERO

DÍGITO_NÃO_ZERO ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

SEQ_DÍGITOS ::= DÍGITO | DÍGITO SEQ_DÍGITOS
```

```
VARIÁVEL (*) ::= LETRA | LETRA SEQ_ALFA_NUM

LETRA ::=   a | b | c | d | e | f | g | h | i
          | j | k | l | m | n | o | p | q | r
          | s | t | u | v | w | x | y | z
          | A | B | C | D | E | F | G | H | I
          | J | K | L | M | N | O | P | Q | R
          | S | T | U | V | W | X | Y | Z

SEQ_ALFA_NUM ::= ALFA_NUM | ALFA_NUM SEQ_ALFA_NUM

ALFA_NUM ::= LETRA | DÍGITO
```

```
TIPO ::= Bool | Nat | (␣TIPO␣->␣TIPO␣)
```

---

* **Importante:** Variáveis têm que ser diferentes das seguintes palavras-chave: `true`, `false`, `if`, `then`, `else`, `endif`, `suc`, `pred`, `ehzero`, `lambda`, `Nat`, `Bool`, `end`.

– Que você tenha uma prática rica em aprendizados. Bom trabalho! –