Interpretador PL/0 em Clojure

Teoria da Computação/Linguagens Formais e Autômatos

Prof. Dr. Jefferson O. Andrade

2023/2

1 Introdução

Este arquivo descreve a implementação de um interpretador para a Linguagem PL/0. O interpretador descrito aqui considera a versão revisada da linguagem que inclui primitivas para entrada e saída de dados, representadas pelos símbolos **"?"** e **"!"**, respectivamente.

Além disso, é importante notar que este arquivo está sendo gerado pelo sistemas Org Mode do editor de textos GNU Emacs – ou simplesmente *Emacs* – e o código fonte Clojure será gerado extraindo-se os blocos de código deste arquivo[1]. Ou seja, para facilitar o entendimento do código estamos utilizando uma técnica chamada programação literada (*"literate programming"* em inglês). Por esse motivo, começamos declarando o *namespace* e as bibliotecas que serão utilizadas.

```
(ns ifes.pl0.core
  (:require [instaparse.core :as insta]
            [clojure.core.match :refer [match]]))
```

2 Gramática da Linguagem PL/0

Para processar os caracteres (texto) de entrada na linguagem PL/0 será usada a biblioteca Instaparse. Esta biblioteca lê a definição da gramática em formato EBNF e gera uma função que recebe um texto, processa esse texto de acordo com as definições da gramática e gera o *Abstract Syntax Tree* (AST) correspondente.

Para a linguagem PL/0 foi gerado o arquivo de gramática `$PRJ/resources/pl0.bnf`[2], mostrado abaixo:

```
<program> = block <"."> .

block = [ const_decl ]
        [ var_decl ]
        { proc_decl }
        statement .

const_decl = <"const"> ident "=" number {<","> ident "=" number} <";"> .

var_decl = <"var"> ident {<","> ident} <";"> .

proc_decl = <"procedure"> ident <";"> block <";"> .

statement = [ ident ":=" expression
            | "call" ident
            | "?" ident
            | "!" expression
            | "begin" statement { <";"> statement } <"end">
```

[1] Depois que for feito o processo de `tangle`, isso não será muito importante. Existirá um arquivo chamado `pl0.clj` com todo o código Clojure nele e este será o arquivo usado pelo complilador para construir o programa.

[2] A variável `$PRJ` será usada para indicar o diretório raiz do projeto.

```
            | "if" condition <"then"> statement
            | "while" condition <"do"> statement
            ].

condition = "odd" expression
          | expression ("="|"<>"|"#"|"<"|"<="|">"|">=") expression .

expression = term { ("+"|"-") term }.

term = factor {("*"|"/") factor}.

factor = ["+"|"-"] factor | ident | number | <"("> expression <")"> .

ident = #"[A-Za-z_][0-9A-Za-z_]*".

number = #"[0-9]+".
```

Para poder lidar com espaços em branco e comentários nos arquivos de entrada PL/0, é necessário definir uma gramática à parte – e mais tarde um parser separado. A gramática para espaços em branco e comentários foi definida no arquivo `$PRJ/resources/pl0-ws.bnf` e é dada abaixo.

```
ws-or-comment = (#'\s+' | comment)+ .
<comment> = line-comment | block-comment .
line-comment = <'//'> #'[^\n\r]*' .
block-comment = <'(*'> inside-comment* <'*)'> .
<inside-comment> = !( '*)' | '(*' ) #'.|[\n\r]' | block-comment .
```

Consulte a documentação do Instaparse para uma descrição detalhada de como definir gramáticas.

3 Definição do Parser da Linguagem PL/0

Uma vez que a gramática esteja definida, podemos usar a função `parser` do pacote Instaparse para construir uma função que fará a análise sintática (*parsing*) dos códigos em PL/0. Entretanto, primeiro precisamos construir o parser para lidar com espaços em branco e comentários e só depois podemos construir o parser principal.

```
(def whitespace-or-comment
  "Parser que se encarrega de identificar os espaços em branco e comentários do
  código fonte."
  (insta/parser (clojure.java.io/resource "pl0-ws.bnf")))

(def parser
  "Parser que reconhece o código fonte da linguagem PL/0. Note que o parser
  `whitespace-or-comment` é passado para a opção `:auto-whitespace`."
  (insta/parser (clojure.java.io/resource "pl0.bnf")
                :auto-whitespace whitespace-or-comment))
```

Note que na criação do analisador sintático principal, `parser`, foi passada a opção `:auto-whitespace whitespace-or-comment`, que instrui o criador do analisador sintático a usar o analisador sintático secundário `whitespace-or-comments` para lidar com espaços em branco.

4 Estruturas de Dados do Interpretador PL/0

4.1 Abstract Syntax Tree

O *Abstract Syntax Tree* da linguagem PL/0 foi implicitamente definido ao se definir a gramática da linguagem[3]. Se temos, por exemplo, uma regra de produção que diz:

```
assign = variable "<-" number ";" .
```

[3] Assim como de qualquer outra definida usando-se o Instaparse.

E o parser criado à partir dessa gramática recebe a cadeia de entrada "`x <- 10`", então a estrutura de dados que será retornada é:

```
[:assign [:variable "x"] "<-" [:number "10"] ";"]
```

Como relação a isso, o máximo de controle que se tem é dizer que alguns símbolos podem ser descartados. Por exemplo, os símbolos "`<-`" e "`;`" na regra de produção acima tem função apenas sintática, sem nenhuma importância semântica. Podem ser descartados sem nenhuma perda de informação. A sinalização de que um símbolo pode ser descartado é feita através de parênteses angulares.

```
assign = variable <"<-"> number <";"> .
```

Nesta nova gramática a estrutura de dados retornada é:

```
[:assign [:variable "x"] [:number "10"]]
```

4.2 Ambiente de Execução

Para o interpretador funcionar adequadamente não é suficiente ter o AST representando o código fonte em PL/0 que foi passado. É necessário ter algum meio de manter registro do comportamênto dinâmico do programa. Ou seja, de como o estado do programa evolui com o tempo.

Toda variável, toda constante, toda função ou procedimento que são definidos no programa PL/0 precisam estar "guardados" em alguma estutura de dados para que o interpretador tenha acesso a essas definições quando for necessário para interpretar algum código. Mais ainda, é necessário que exista uma forma de isolar algumas definições de outras.

Por exemplo, se temos um procedimento `p1` e outro procedimento `p2`, não podemos deixar que `p1` acesse as var iáveis declaradas em `p2` e vice-versa. Mas tanto `p1` quanto `p2` precisam ter acesso às variáveis declaradas no programa principal. De modo mais concreto, considere o código PL/0 abaixo:

```
1  var ret, arg, acc;
2  procedure p1;
3  var x;
4  begin
5     x := arg;
6     ret := x / 3
7  end;
8  procedureprocedure p2;
9  var y;
10 begin
11    y := arg;
12    ret := y * 2;
13 end;
14 begin
15    ?arg;
16    acc := 0;
17    p1;
18    acc := acc + ret;
19    p2;
20    acc := acc + ret;
21    !acc
22 end.
```

É necessário que tanto `p1` quanto `p2` tenham acesso a `arg` e a `ret`, mas `p1` não pode ter acesso a `y` e `p2` não pode ter acesso a `x`. E no program a principal não deve ser possível acessar nem `x` e nem `y`.

A forma mais direta de resolver essa questão é criar um ambiente de execução que tenha níveis que espelhem os níveis de execução do código. Toda vez que um procedimento for executado um novo nível no ambiente de execução é criado e o procedimento será executado neste novo nível que foi criado. Evidentemente, esse novo nível apontará para o nível que existia no momento em que ele foi criado. Se for pedido um valor associado a um nome que não existe no nível atual do ambiente de execução, esse valor será buscado nos níveis superiores, até que se atinja o nível mais alto.

Para implementar o ambiente de execução dos programas do nosso interpretador, vamos definir dois registro em Clojure: `Binding` e `Env`.

4.2.1 Binding

Um `Binding` (amarração) é a estrutura de dados que está associada a um nome no ambiente em que o código PL/0 estará sendo executado. Um `Binding` possui dois campos:

- `:kind` – o tipo de binding que está sendo criado (:var, :const, :proc)
- `:value` – o valor que está associado ao binding

```
23 (defrecord Binding [kind value])
```

Ao criar o registro `Binding` também são implicitamente definidas as funções `->Binding` e `map->Binding` para criação de objetos desse tipo.

4.2.2 Env

Um 'Env' (environment/ambiente) é a estrutura de dados que irá armazenar todas as informações necessárias para a execução do código dos programas PL/0. Um 'Env' possui dois campos:

- `:bindings` – guarda todos os nomes definidos no ambiente. Cada nome está associado a um 'Binding'.
- `:parent` – guarda o ambiente pai do ambiente atual. Se este valor for `nil`, significa que este ambiente é o ambiente raiz.

```
24 (defrecord Env [bindings parent])
25
26 (defn new-env
27   "Cria um novo ambiente. Se não for especificado `parent`, é criado um ambiente
28   raiz, caso contrário é criado um ambiente filho do ambiene indicado."
29   ([] (->Env {} nil))
30   ([parent] (->Env {} parent)))
```

Além das funções implicitamente definidas padrão, `->Env` e `map->Env`, também foi criada explicitamente uma função `new-env` para construção de um novo ambiente sem nenhum *binding*.

Abaixo são definidas várias funções auxiliares para lidar com os ambientes de execução.

```
31 (defn is-defined?
32   "Retorna `true` se o `name` estiver definido em `env`, e `false` caso contrário.
33   Testa o ambiente no nível atuale toda a sequência de `parent` até o nível
34   raiz."
35   [env name]
36   (or (contains? (:bindings env) name)
37       (and (some? (:parent env)) (is-defined? (:parent env) name))))
38
39
40 (defn def-name
41   "Acrescenta uma definição do tipo (`kind`) indicado para `name` com o `value`
42   indicado. Não valida se o valor é apropriado para o tipo informado."
43   [env name kind value]
44   (assoc-in env [:bindings name] (->Binding kind value)))
45
46 (defn def-var [env name] (def-name env name :var nil))
47 (defn def-const [env name value] (def-name env name :const value))
48 (defn def-proc [env name body] (def-name env name :proc body))
49
50
51 (defn set-var
52   "Retorna um novo ambiente à partir de `env` com o valor `value` associado à
```

```
  variável `name`. Gera um erro se `name` não estiver definido ou se não for uma
  variável."
  [env name value]
  ;; (printf "(set-var %s %s %s)\n" env name value)
  (cond
    (some? (get-in env [:bindings name]))
    (let [v (get-in env [:bindings name])]
      (match [(:kind v)]
        [:var] (assoc-in env [:bindings name :value] value)
        [:const] (throw (ex-info (str "Name is a constante: " name) v))
        [:proc] (throw (ex-info (str "Name is a procedure: " name) v))))
    (some? (:parent env)) (assoc env :parent (set-var (:parent env) name value))
    :else (throw (ex-info (str "Undefined variable: " name) {}))))


(defn get-value
  "Retorna o valor da variável ou constante `name` no ambiente `env`. Gera um
  erro se `name` não for definido, ou se não for uma variável ou uma constante."
  [env name]
  ;; (printf "(get-value %s %s)\n" env name)
  (cond
    (some? (get-in env [:bindings name]))
    (let [v (get-in env [:bindings name])]
      (match [(:kind v)]
        [:var] (:value v)
        [:const] (:value v)
        [:proc] (throw (ex-info (str "Name is a procedure: " name) v))))
    (some? (:parent env)) (recur (:parent env) name)
    :else (throw (ex-info (str "Undefined variable or constant: " name) {}))))


(defn get-proc
  "Retorna a definição do procedimento `name` no ambiente `env`. Gera um erro se
  `name` não for definido, ou se não for um procedimento."
  [env name]
  (cond
    (some? (get-in env [:bindings name]))
    (let [v (get-in env [:bindings name])]
      (match [(:kind v)]
        [:var] (throw (ex-info (str "Name is a variable: " name) v))
        [:const] (throw (ex-info (str "Name is a constante: " name) v))
        [:proc] (:value v)))
    (some? (:parent env)) (get-proc (:parent env) name)
    :else (throw (ex-info (str "Procedure not defined: " name) {}))))
```

5 Funções do Interpretador PL/0

Serão definidos dois tipos de função:

1. as funções `exec` são para execução de declarações e comandos e retornam ambientes;
2. as funções `eval` são para avaliação de condições e expresões e retornam o valor da condição ou expressão.

Como Clojure exige que a função seja definida – ou ao menos declarada – antes de ser usada, faremos a declaração de todas as funções aqui e definiremos as funções numa abordagem *top-down*.

```
(declare exec-block exec-decl exec-const-decl exec-var-decl
         exec-sttmt exec-progn exec-if exec-while
         eval-expr eval-cond eval-rel-expr)
```

5.1 Funções que Executam Declarações e Comandos

A função `exec` é o ponto de entrada **principal** do interpretador. Ela recebe o código fonte do programa PL/0 e interpreta o programa. A função exec passa o código fonte PL/0 ao `parser`, exctrai o AST, e passa esse AST à função `exec-block`.

```
(defn exec
  "Avalia o programa `prog` escrito em PL/0."
  [prog & {:keys [return-env] :or {return-env false}}]
  (let [env (-> prog
                (parser)
                (first)
                (exec-block))]
    (if return-env env nil)))
```

a função `exec-block` processa a definição do programa principal, e também dos procedimentos. Em PL/0 tanto o programa principal, quanto os procedimentos são declarados como um bloco (`block`) na gramática. E um bloco é uma sequência de declarações, opcionais, de constantes, variáveis, procedimentos, e um comando (*statement*) obritatório ao final. As declarações de constantes, variáveis e procedimentos apenas preparam o ambiente para a execução do comando ao final do bloco.

```
(defn exec-block
  "Avalia um bloco de código PL/0. Um bloco de código é uma sequência de
  declarações. A avaliação/execução de cada declaração pode gerar mudanças no
  ambiente de execução do programa."
  ([ast] (exec-block ast (new-env)))
  ([ast env]
   (match [ast]
     [[]] env
     [[:block & decls]] (recur decls env)
     [[d1 & ds]] (->> env
                      (exec-decl d1)
                      (recur ds)))))


(defn exec-decl
  "Avalia/executa uma declaração de um bloco. Uma declaração pode ser: (i)
  declaração de constantes; (ii) declaração de variáveis; (iii) definição de
  procedimento; ou (iv) um `statement`."
  [decl env]
  (match [decl]
    ;; declaração de constantes
    [[:const_decl & const-inits]] (exec-const-decl const-inits env)
    ;; declaração de variáveis
    [[:var_decl & var-ids]] (exec-var-decl var-ids env)
    ;; declaração de procedimentos
    [[:proc_decl [:ident name] body]] (def-proc env name body)
    ;; statement
    [[:statement & _]] (exec-sttmt decl env)))


(defn exec-var-decl
  "Atualiza o ambiente `env` com as definições das variáveis dadas em `idents`.
  Cria um novo ambiente, à partir de `env` em que as variáveis comos nomes dados
  em `idents` estão definidas. Retorna o novo ambiente."
  [idents env]
  (match [idents]
    [[]] env
    [[[:ident name] & idents1]]
    (->> (def-var env name)
         (recur idents1))))


```

```clojure
(defn exec-const-decl
  "Atualiza o ambiente `env` com as definições das constantes dadas em `inits`.
  Cria um novo ambiente, à partir de `env` em que as constantes comos nomes e
  valores dados em `inits` estão definidas. Retorna o novo ambiente."
  [inits env]
  (match [inits]
    [[]] env
    [[[:ident name] "=" [:number num] & inits1]]
    (->> (Integer/parseInt num)
         (def-const env name)
         (recur inits1))))
```

A função `exec-sttmt`, que executa um comando, deve ser a última a ser chamada ao se processar um bloco. Ela também é chamada recursivamente ao se processar vários tipos de comandos, como o comando de sequência (`begin` ... `end`), o comando condicional (`if` ... `then` ...) e o comando de repetição (`while` ... `do` ...). Existem 7 tipos de comando em PL/0:

- **Atribuição:** associa à variável o valor indicado e retorna o novo ambiente resultante.
- **Chamada de procedimento:** cria um novo ambiente descendente do atual, executa o procedimento com o novo ambiente e retorna o ambiente no nível atual após a execução.
- **Entrada de dados:** Imprime o nome da variável como *prompt*. Lê um valor da entrada padrão. Atribui o valor à variável e retorna o ambiente resultante.
- **Saída de dados:** Calcula o valor da expressão indicada. Escreve o valor na saída padrão e retorna o ambiente atual.
- **Comando de sequência:** Executa os comandos indicados, em sequência. O primeiro comando é executado com o ambiente indicado. O ambiente retornado pelo primeiro comando é usado para executar o segundo comando. O ambiente retornado pelo segundo comando é usado para executar o terceiro comando e assim por diante. Retorna o ambiente retornado pelo último comando.
- **Comando condicional:** Avalia o valor da condição com o ambiente atual. Se o valor da condição for diferente de zero, executa o comando indicado com o ambiente atual e retorna o ambiente que o comando retornar. Caso contrário retorna o ambiente atual.
- **Comando de repetição:** Avalia o valor da condição com o ambiente atual. Se o valor da condição for zero, retorna o ambiente atual. Caso contrário, executa o comando indicado com o ambiente atual e usa o ambiente que o comando retornar para repetir o processo recursivamente.

```clojure
(defn exec-sttmt
  "Executa um comando (statement) PL/0 e retorn o ambiente resultante."
  [sttmt env]
  ;; (printf "(exec-sttmt %s %s)\n" sttmt env)
  (match [sttmt]
    ;; atribuição
    [[:statement [:ident var] ":=" expr]]
    (let [value (eval-expr expr env)]
      (set-var env var value))
    ;; chamada de procedimento
    [[:statement "call" [:ident name]]]
    (-> (get-proc env name)
        (exec-block (new-env env))
        (get :parent))
    ;; Lê um valor para a variável
    [[:statement "?" [:ident name]]]
    (->> (do (printf "%s? " name)
             (flush)
             (read-line))
```

```
        (Integer/parseInt)
        (set-var env name))
    ;; Imprime o valor da expressão
    [[:statement "!" expr]]
    (do (->> (eval-expr expr env)
             (println))
        env)
    ;; Comando de sequência
    [[:statement "begin" & sttmt-seq]]
    (exec-progn sttmt-seq env)
    ;; condicional
    [[:statement "if" cond1 sttmt1]]
    (exec-if cond1 sttmt1 env)
    ;; repetição
    [[:statement "while" cond1 sttmt1]]
    (exec-while cond1 sttmt1 env)
    ;;
    ))


(defn exec-progn
  "Executa os comandos em `sttmts` sequencialmente com o ambiente `env`. O
  primeiro comando é executado com `env`. O ambiente retornado pelo primeiro
  comando é usado para executar o segundo comando. O ambiente retornado pelo
  segundo comando é usado para executar o terceiro comando e assim por diante.
  Retorna o ambiente retornado pelo último comando."
  [sttmts env]
  ;; (printf "(exec-progn %s %s)\n" sttmts env)
  (match [sttmts]
    [[]] env
    [[stt1 & stts]]
    (->> (exec-sttmt stt1 env)
         (recur stts))))


(defn exec-if
  "Avalia o valor de `cnd` com o ambiente `env`; se o valor for diferente de zero,
  executa `sttmt` usando `env` e retorna o ambiente resultante. Caso contrário
  retorna `env`."
  [cnd sttmt env]
  (let [cnd-val (eval-cond cnd env)]
    (if (zero? cnd-val)
      env
      (exec-sttmt sttmt env))))


(defn exec-while
  "Avalia `cnd` em `env`; se o valor for zero, retorna `env`; caso contrário,
  executa `sttmt` em `env` e usa o ambiente resultante para repetir o processo
  recursivamente."
  [cnd sttmt env]
  (let [cnd-val (eval-cond cnd env)]
    (if (zero? cnd-val)
      env
      (->> (exec-sttmt sttmt env)
           (recur cnd sttmt)))))
```

5.2 Funções que Avaliam Condições e Expressões

Quando se trata de avaliar expressões, algumas linguagens de programação são tão "bem comportadas" assim. Em C, por exemplo, existem diversas formas de uma expressão mudar o ambiente de execução, e.g., `(x = 5) + y`, `(x++ + -y)`, etc. Para linguagens desse tipo são necessários mecanismos mais sofisticados de

interpretação. Em PL/0, felizmente, uma expressão não pode mudar o ambiente de execução. Isso facilita o processo de interpretação.

No nosso interpretador, as funções que avaliam as condições e expressões recebem a condição ou expressão a avaliar e o ambiente no qual será veita a avaliação, e retornam o resultado da avaliação. Sempre considerando que o ambiente não muda durante o processo de avaliação.

A avaliação é realizada de forma recursiva em uma estrutura descendente que espelha a estrutura da gramática.

```
(declare eval-term eval-factor)

(defn eval-cond
  "Avalia a condição `cnd` de acrodo com as definições de `env` e retorna
  verdadeiro (1) ou falso (0). A condição pode ser `odd <expr>` ou `<expr> <op>
  <expr>`, onde `<op>` é um operador relacional."
  [cnd env]
  (match [cnd]
    [[:condition "odd" expr1]]
    (let [value (eval-expr expr1 env)]
      (mod value 2))
    [[:condition expr1 op expr2]]
    (eval-rel-expr expr1 op expr2 env)))


(defn eval-rel-expr
  "Avalia uma expressão relacional com duas subexpressões, `expr1` e `expr2`, e um operador `op`
  relacional. Retorna 1 se a expressão for verdadeira e 0 caso contrário."
  [expr1 op expr2 env]
  (let [val1 (eval-expr expr1 env)
        val2 (eval-expr expr2 env)]
    (match [op]
      ["="] (if (= val1 val2) 1 0)
      [">"] (if (> val1 val2) 1 0)
      ["<"] (if (< val1 val2) 1 0)
      [(:or "<>" "#")] (if (not= val1 val2) 1 0)
      [">="] (if (>= val1 val2) 1 0)
      ["<="] (if (<= val1 val2) 1 0)
      ;;
      )))


(defn eval-expr
  "Avalia a empressão `expr` no ambiente de execução `env`. Retorna o valor da
  expressão."
  ([expr env] (eval-expr expr nil env))
  ([expr acc env]
  (match [expr]
    [[]] acc
    [[:expression t1 & ts]] (let [v1 (eval-term t1 env)]
                              (eval-expr ts v1 env))
    [["+" t1 & ts]] (let [v1 (eval-term t1 env)]
                      (eval-expr ts (+ acc v1) env))
    [["-" t1 & ts]] (let [v1 (eval-term t1 env)]
                      (eval-expr ts (- acc v1) env)))))


(defn eval-term
  "Avalia o termo `term` no ambiente de execução `env`. Retorna o valor do termo."
  ([term env] (eval-term term nil env))
  ([term acc env]
   (match [term]
     [[]] acc
     [[:term f1 & fs]] (let [v1 (eval-factor f1 env)]
```

```
                          (eval-term fs v1 env))
     [["*" f1 & fs]] (let [v1 (eval-factor f1 env)]
                       (eval-term fs (* acc v1) env))
     [["/" f1 & fs]] (let [v1 (eval-factor f1 env)]
                       (eval-term fs (quot acc v1) env)))))


(defn eval-factor
  "Avalia o fator `factor` no ambiente de execução `env`. Retorna o valor do
  fator."
  [factor env]
  ;; (printf "(eval-factor %s %s)\n" factor env)
  (match [factor]
    [[:factor "+" f1]] (let [v1 (eval-factor f1 env)] v1)
    [[:factor "-" f1]] (let [v1 (eval-factor f1 env)] (- v1))
    [[:factor [:ident name]]] (get-value env name)
    [[:factor [:number n]]] (Integer/parseInt n)
    [[:factor [:expression & _]]] (eval-expr (nth factor 1) env)
    ;;
    ))
```