Ocampiler

Autores : Felipe Assad , Jorge Chagas, Thiago

Data : 07/06/2019

Universidade Federal Fluminense

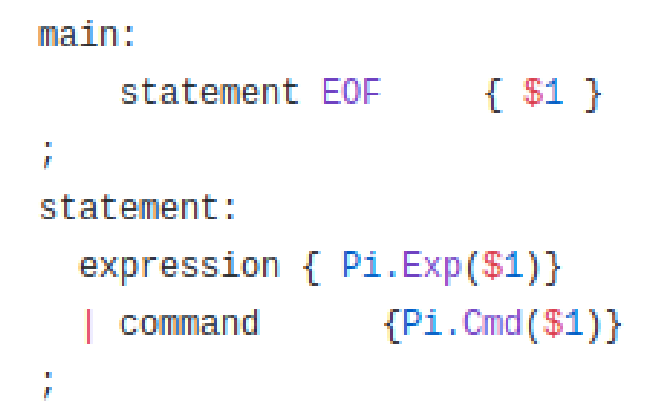
Objetivos desta apresentação

- Apresentar modificações no autômato para receber as declarations

O que foi feito

- Corrigimos o parser
- Implementamos o Autômato

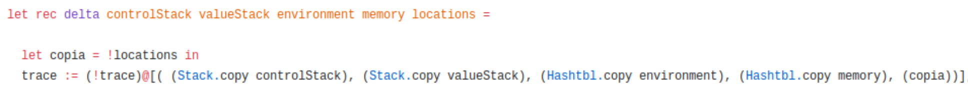
Correção do Parser



```
main:
    statement EOF     { $1 }
;
statement:
  expression { Pi.Exp($1)}
  | command      {Pi.Cmd($1)}
;
```

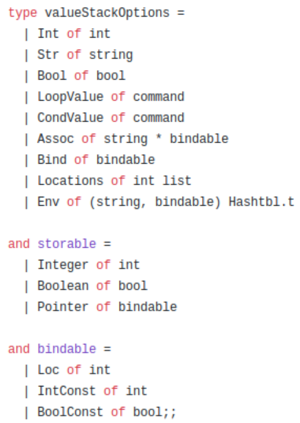
Implementação do Automaton

Definição de trace em Automaton.ml



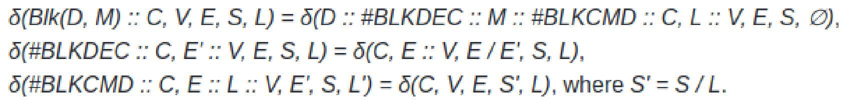
```
let rec delta controlStack valueStack environment memory locations =

  let copia = !locations in
  trace := (!trace)@[( (Stack.copy controlStack), (Stack.copy valueStack), (Hashtbl.copy environment), (Hashtbl.copy memory), (copia))]
```

Definição dos tipos aceitáveis na pilha de valores

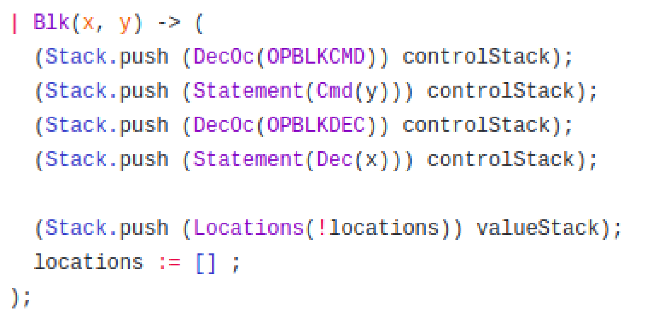


```
type valueStackOptions =
  | Int of int
  | Str of string
  | Bool of bool
  | LoopValue of command
  | CondValue of command
  | Assoc of string * bindable
  | Bind of bindable
  | Locations of int list
  | Env of (string, bindable) Hashtbl.t

and storable =
  | Integer of int
  | Boolean of bool
  | Pointer of bindable

and bindable =
  | Loc of int
  | IntConst of int
  | BoolConst of bool;;
```

Implementação do Automaton

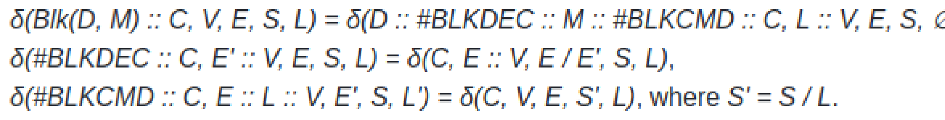


$\delta(Blk(D, M) :: C, V, E, S, L) = \delta(D :: \#BLKDEC :: M :: \#BLKCMD :: C, L :: V, E, S, \emptyset)$,
$\delta(\#BLKDEC :: C, E' :: V, E, S, L) = \delta(C, E :: V, E / E', S, L)$,
$\delta(\#BLKCMD :: C, E :: L :: V, E', S, L') = \delta(C, V, E, S', L)$, where $S' = S / L$.



```
| Blk(x, y) -> (
  (Stack.push (DecOc(OPBLKCMD)) controlStack);
  (Stack.push (Statement(Cmd(y))) controlStack);
  (Stack.push (DecOc(OPBLKDEC)) controlStack);
  (Stack.push (Statement(Dec(x))) controlStack);

  (Stack.push (Locations(!locations)) valueStack);
  locations := [] ;
);
```

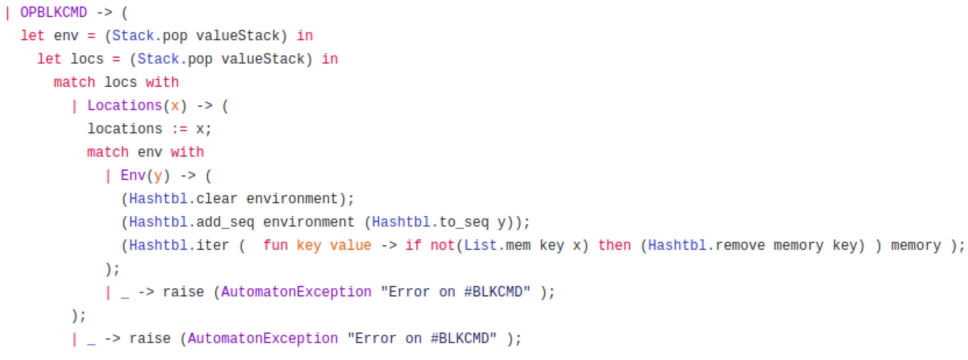
Implementação do Automaton



$\delta(Blk(D, M) :: C, V, E, S, L) = \delta(D :: \#BLKDEC :: M :: \#BLKCMD :: C, L :: V, E, S, \varnothing$

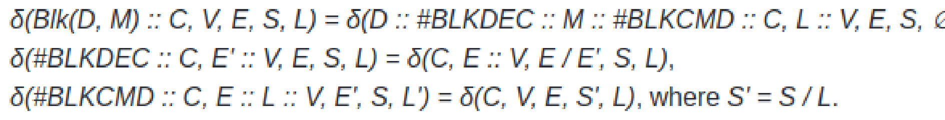
$\delta(\#BLKDEC :: C, E' :: V, E, S, L) = \delta(C, E :: V, E / E', S, L),$

$\delta(\#BLKCMD :: C, E :: L :: V, E', S, L') = \delta(C, V, E, S', L)$, where $S' = S / L$.



```
| OPBLKCMD -> (
  let env = (Stack.pop valueStack) in
    let locs = (Stack.pop valueStack) in
      match locs with
        | Locations(x) -> (
          locations := x;
          match env with
            | Env(y) -> (
              (Hashtbl.clear environment);
              (Hashtbl.add_seq environment (Hashtbl.to_seq y));
              (Hashtbl.iter (  fun key value -> if not(List.mem key x) then (Hashtbl.remove memory key) ) memory );
            );
            | _ -> raise (AutomatonException "Error on #BLKCMD" );
        );
        | _ -> raise (AutomatonException "Error on #BLKCMD" );
```

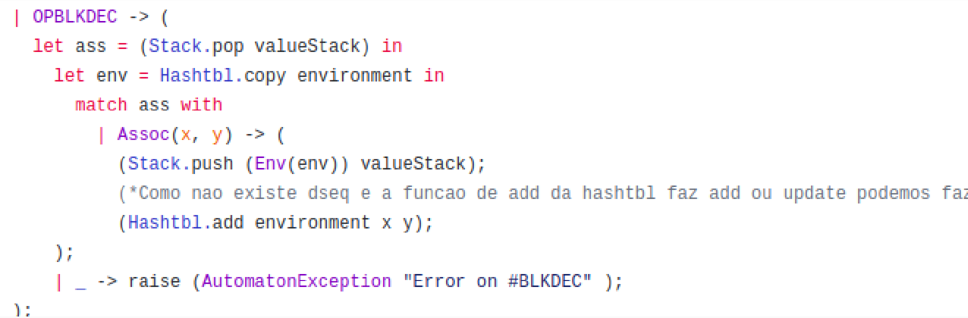
Implementação do Automaton



$\delta(Blk(D, M) :: C, V, E, S, L) = \delta(D :: \#BLKDEC :: M :: \#BLKCMD :: C, L :: V, E, S, \emptyset$

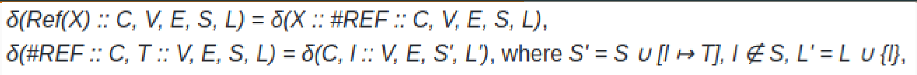
$\delta(\#BLKDEC :: C, E' :: V, E, S, L) = \delta(C, E :: V, E / E', S, L),$

$\delta(\#BLKCMD :: C, E :: L :: V, E', S, L') = \delta(C, V, E, S', L)$, where $S' = S / L$.



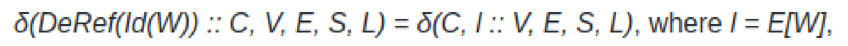
```
| OPBLKDEC -> (
  let ass = (Stack.pop valueStack) in
    let env = Hashtbl.copy environment in
      match ass with
        | Assoc(x, y) -> (
          (Stack.push (Env(env)) valueStack);
          (*Como nao existe dseq e a funcao de add da hashtbl faz add ou update podemos fa
          (Hashtbl.add environment x y);
    );
    | _ -> raise (AutomatonException "Error on #BLKDEC" );
);
```

Implementação do Automaton

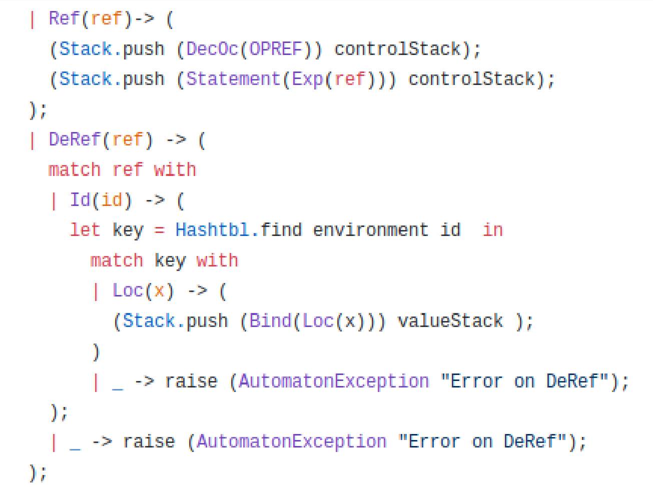


$\delta(Ref(X) :: C, V, E, S, L) = \delta(X :: \#REF :: C, V, E, S, L)$,

$\delta(\#REF :: C, T :: V, E, S, L) = \delta(C, l :: V, E, S', L')$, where $S' = S \cup [l \mapsto T], l \notin S, L' = L \cup \{l\}$,

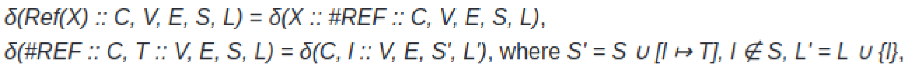


$\delta(DeRef(Id(W)) :: C, V, E, S, L) = \delta(C, l :: V, E, S, L)$, where $l = E[W]$,



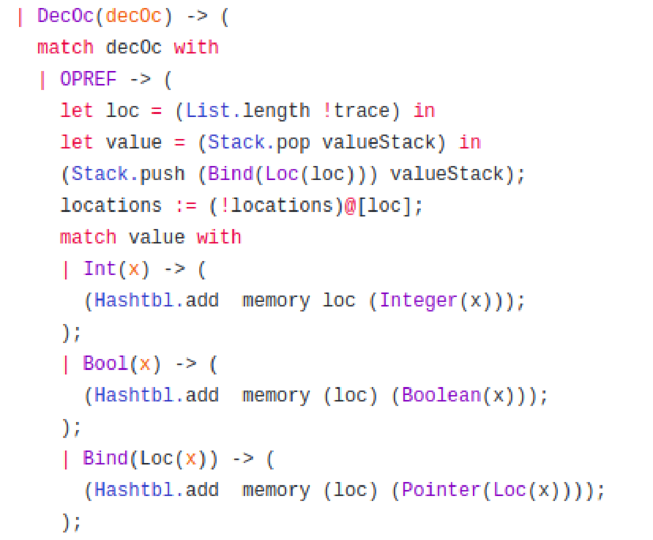
```
| Ref(ref)-> (
  (Stack.push (DecOc(OPREF)) controlStack);
  (Stack.push (Statement(Exp(ref))) controlStack);
);
| DeRef(ref) -> (
  match ref with
  | Id(id) -> (
    let key = Hashtbl.find environment id  in
      match key with
      | Loc(x) -> (
        (Stack.push (Bind(Loc(x))) valueStack );
      )
      | _ -> raise (AutomatonException "Error on DeRef");
  );
  | _ -> raise (AutomatonException "Error on DeRef");
);
```

Implementação do Automaton



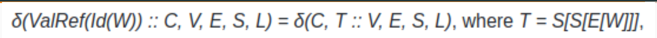
$\delta(Ref(X) :: C, V, E, S, L) = \delta(X :: \#REF :: C, V, E, S, L),$

$\delta(\#REF :: C, T :: V, E, S, L) = \delta(C, l :: V, E, S', L')$, where $S' = S \cup [l \mapsto T], l \notin S, L' = L \cup \{l\},$

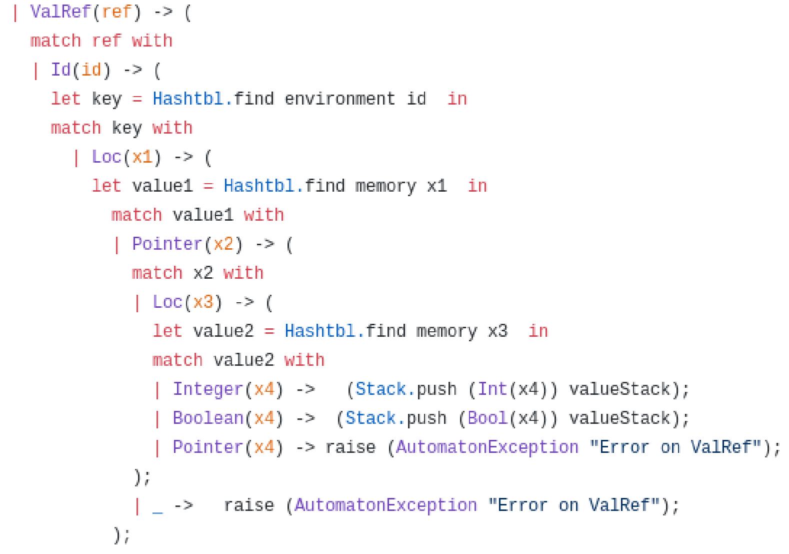


```
| DecOc(decOc) -> (
  match decOc with
  | OPREF -> (
    let loc = (List.length !trace) in
    let value = (Stack.pop valueStack) in
    (Stack.push (Bind(Loc(loc))) valueStack);
    locations := (!locations)@[loc];
    match value with
    | Int(x) -> (
      (Hashtbl.add  memory loc (Integer(x)));
    );
    | Bool(x) -> (
      (Hashtbl.add  memory (loc) (Boolean(x)));
    );
    | Bind(Loc(x)) -> (
      (Hashtbl.add  memory (loc) (Pointer(Loc(x))));
    );
```

Implementação do Automaton

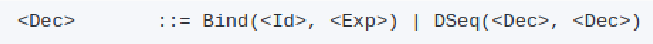


δ(ValRef(Id(W)) :: C, V, E, S, L) = δ(C, T :: V, E, S, L), where T = S[S[E[W]]],

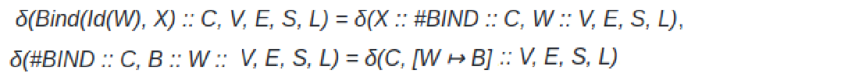


```
| ValRef(ref) -> (
  match ref with
  | Id(id) -> (
    let key = Hashtbl.find environment id  in
    match key with
      | Loc(x1) -> (
        let value1 = Hashtbl.find memory x1  in
          match value1 with
          | Pointer(x2) -> (
            match x2 with
            | Loc(x3) -> (
              let value2 = Hashtbl.find memory x3  in
              match value2 with
              | Integer(x4) ->   (Stack.push (Int(x4)) valueStack);
              | Boolean(x4) ->  (Stack.push (Bool(x4)) valueStack);
              | Pointer(x4) -> raise (AutomatonException "Error on ValRef");
            );
            | _ ->   raise (AutomatonException "Error on ValRef");
          );
```

Implementação do Automaton

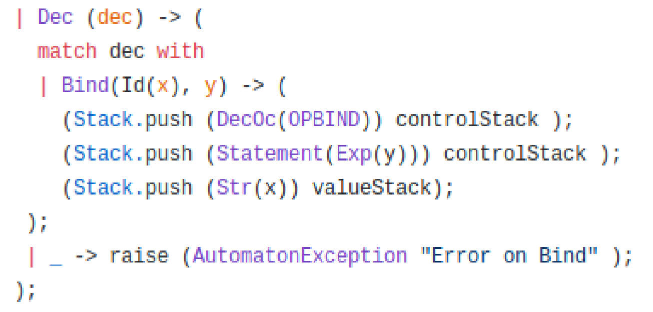


```
<Dec>       ::= Bind(<Id>, <Exp>) | DSeq(<Dec>, <Dec>)
```



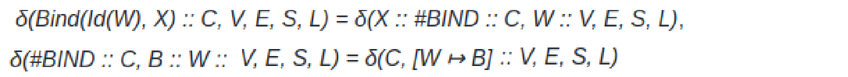
$\delta(Bind(Id(W), X) :: C, V, E, S, L) = \delta(X :: \#BIND :: C, W :: V, E, S, L),$

$\delta(\#BIND :: C, B :: W :: V, E, S, L) = \delta(C, [W \mapsto B] :: V, E, S, L)$



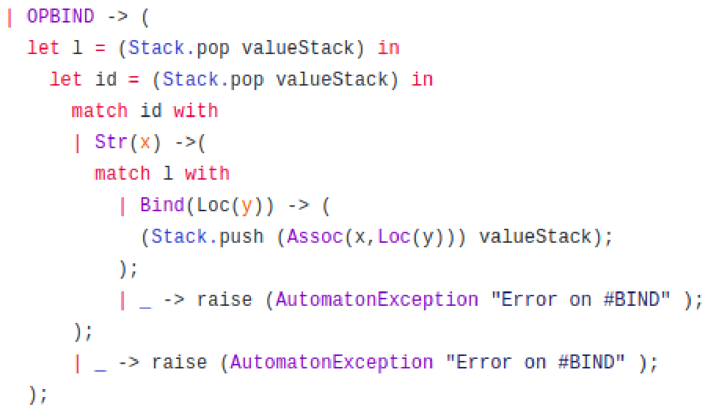
```
| Dec (dec) -> (
  match dec with
  | Bind(Id(x), y) -> (
    (Stack.push (DecOc(OPBIND)) controlStack );
    (Stack.push (Statement(Exp(y))) controlStack );
    (Stack.push (Str(x)) valueStack);
 );
 | _ -> raise (AutomatonException "Error on Bind" );
);
```

Implementação do Automaton



$\delta(Bind(Id(W), X) :: C, V, E, S, L) = \delta(X :: \#BIND :: C, W :: V, E, S, L),$

$\delta(\#BIND :: C, B :: W :: V, E, S, L) = \delta(C, [W \mapsto B] :: V, E, S, L)$



```
| OPBIND -> (
  let l = (Stack.pop valueStack) in
    let id = (Stack.pop valueStack) in
      match id with
      | Str(x) ->(
        match l with
          | Bind(Loc(y)) -> (
            (Stack.push (Assoc(x,Loc(y))) valueStack);
          );
          | _ -> raise (AutomatonException "Error on #BIND" );
      );
      | _ -> raise (AutomatonException "Error on #BIND" );
  );
```

O que não foi feito e porque

- Tratamento das constantes
- As combinações possíveis para as referências

  ex:

  x := *y + 2

  if (*x ¡ 4) then ... else ... done

Dúvidas

- Podemos gerar as locations aleatoriamente ou devem ser sequenciais?

Avaliação da evolução do trabalho

- Implementar um parser para a linguagem Imp-1 estendendo Imp-0 com declarações de variáveis e constantes. (OK)
- Implementar IR-mark1: (i) Interpreting Automata com ambientes (OK), (ii) declarações de variáveis e constantes. (1/2 OK)
- Implementar um compilador de Imp-1 para IR-mark1.