Especificações da linguagem Łukasiewicz estendida

Laércio Lima Pilla
criador da linguagem

Gustavo Zambonin
gerente de projeto, projetista de linguagem

Douglas Martins
arquiteto de sistema

Marcello Klingelfus
testador

Construção de Compiladores (UFSC – INE5426)*

Sumário

*laercio.pilla@ufsc.br, {marcelino.douglas,gustavo.zambonin,marcello.klingelfus}@grad.ufsc.br src/

1 Linguagem introdutória

A versão introdutória da linguagem tem as seguintes características:

i. Um tipo inteiro `int` com valores representados por números naturais compostos por dígitos decimais.
ii. Quatro operadores binários: soma `+`, subtração `-`, multiplicação `*` e divisão `/`.
iii. Um operador menos unário `-`.
iv. Parênteses `()`.
v. Variáveis com nomes começados por uma letra maiúscula ou minúscula, podendo ser seguidos por outros caracteres, dígitos decimais e '_'.
vi. Atribuições `=`.

As diferentes instruções da linguagem são separadas apenas pela marcação de fim de linha. Múltiplas variáveis de mesmo tipo podem ser declaradas em uma mesma linha, necessitando-se apenas da separação entre seus nomes com uma vírgula `,`. Cada variável pode receber uma atribuição de valor inicial de valores constantes. A precedência entre os operadores é a mesma da matemática padrão. Não é possível fazer múltiplas atribuições em uma mesma linha.

1.1 Exemplo de código

```
int a_
int    BB, c
int d=0,        e1=1
a_ = d+2*3
BB = (-a_)/  12-1
c = e1*e1/a_
```

1.2 Saída do compilador

```
int var: a_
int var: BB, c
int var: d = 0, e1 = 1
= a_ + d * 2 3
= BB - / -u a_ 12 1
= c * e1 / e1 a_
```

1.3 Saída do *transpiler*

```
d = 0
e1 = 1
a_ = (d + (2 * 3))
BB = (-a_ / 12) - 1)
c = (e1 * e1) / a_)
```

1.4 Tratamento de erros

i. No caso de erros léxicos (símbolos e palavras desconhecidas), o compilador deve apenas emitir uma mensagem de erro informando sobre o caractere ou palavra desconhecida, não passando nenhum token para a análise sintática.

```
int a&
[Line 1] lexical error: unknown symbol &
int #####a
[Line 2] lexical error: unknown symbol #####
```

ii. No caso de um erro sintático, o compilador deve ignorar todo o bloco de código comprometido (declaração de variáveis, atribuições etc.). A mensagem deve seguir similar a original. O compilador poderá emitir erros sintáticos verbosos, apresentando maneiras para programar o bloco de código corretamente.

```
int 10b
[Line 1] syntax error, unexpected INT, expected FUN
```

iii. No caso de uso de uma variável ainda não declarada, uma mensagem de erro semântico deve ser impressa.

```
A = 2
[Line 1] semantic error: undeclared variable A
```

iv. No caso de uma variável ser declarada novamente, a declaração original deve ser mantida e a nova declaração deve ser ignorada. Uma mensagem de erro deve ser emitida.

```
int a, a
[Line 1] semantic error: re-declaration of variable a
```

1.5 Saída padrão

```
nameT: [a-zA-Z][a-zA-Z0-9_]*
intT: [0-9]+
typeT: int
typeT var: nameT {= intT}?{, nameT {= intT}?}*
```

1.6 Saída de erros padrão

Note que <> pode representar qualquer *token* interno da ferramenta `bison`.

```
[Line N] lexical error: unknown symbol X
[Line N] syntax error, unexpected <>, expecting <>{ or <>}*
[Line N] semantic error: undeclared variable X
[Line N] semantic error: re-declaration of variable X
```

2 Ponto flutuante e valores booleanos

Esta versão adiciona as seguintes características à linguagem:

i. Tipo ponto flutuante `float` com valores representados por números reais compostos por dígitos decimais e uma vírgula.
ii. Tipo booleano `bool` com valores `true` e `false`.
iii. Operações unárias e binárias sobre valores de ponto flutuante.
iv. Operadores relacionais para comparação entre inteiros ou valores de ponto flutuante igual `==`, diferente `!=`, maior `>`, menor `<`, maior ou igual `>=` e menor ou igual `<=`.
v. Operadores binários booleanos e `&` e ou `|`.
vi. Operador unário booleano de negação `!`.

Esta versão não possibilita a conversão entre tipos. Operações relacionais geram valores booleanos. A precedência dos operadores para valores inteiros e de ponto flutuante é a padrão. A precedência para expressões booleanas é de operador unário > operadores relacionais > operadores binários.

2.1 Exemplo de código

```
float f=1.0, g=0., h= .10, i
bool b = true
i = -f*g-h/2.1
b = ! (i > 0.0) | (i < -2.3)
```

2.2 Saída do compilador

```
float var: f = 1.0, g = 0., h = .10, i
bool var: b = true
= i - * -u f g / h 2.1
= b | ! > i 0.0 < i -u 2.3
```

2.3 Saída do *transpiler*

```
f = 1.0
g = 0.
h = .10
b = True
i = (-f * g) - (h / 2.1))
b = (not (i > 0.0)) | (i < -2.3))
```

2.4 Tratamento de erros

i. No caso de tipos incompatíveis usados para uma operação, uma mensagem de erro semântico deve ser emitida. No caso de operações aritméticas ou atribuições, o tipo esperado é o tipo do operando à esquerda. Para reduzir o encadeamento de erros, o tipo retornado por essas operações deve ser o tipo do operando à esquerda. Operações relacionais e booleanas retornam sempre booleanos. Em caso de erro, o menos unário retorna o tipo inteiro.

```
int a = 1.0
[Line 1] semantic error: attribution operation expected integer but received float
a = a + true
[Line 2] semantic error: addition operation expected integer but received boolean
```

2.5 Saída padrão

```
typeT: int | float | bool
boolT: true | false
```

2.6 Saída de erros padrão

```
op: addition | subtraction | multiplication | division | attribution |
    unary minus | equal | different | greater than | less then |
    greater or equal than | less or equal than | and | or | negation
type: integer | float | boolean
[Line N] semantic error: op operation expected type but received type
```

3 Conversão de tipos

Esta versão adiciona as seguintes características à linguagem:

i. Coerção de valores `int` para valores `float` em operações binárias com outros valores `float`.
ii. *Casting* entre quaisquer tipos básicos através das operações unárias `[int]`, `[float]` e `[bool]`.

As operações de *casting* têm a menor precedência possível.

3.1 Exemplo de código

```
int i = 0, j
float f = 1.1
bool b = true
j = [int] [int] i + f
i = [int] j
b = b & [bool] f
f = ([float] b) + 0.0
```

3.2 Saída do compilador

```
int var: i = 0, j
float var: f = 1.1
bool var: b = true
= j [int] [int] + [float] i f
= i [int] j
= b & b [bool] f
= f + [float] b 0.0
```

3.3 Saída do *transpiler*

```
i = 0
f = 1.1
b = True
j = int(int(float(i) + f))
i = int(j)
b = (b & bool(f))
f = (float(b) + 0.0)
```

3.4 Saída padrão

```
[typeT]
```

4 Expressões condicionais

Esta versão inclui estruturas de execução condicional `if-then` e `if-then-else`. O teste da expressão condicional recebe um valor ou variável booleana. Chaves `{}` são usadas para marcar o início da região `then`. No caso da presença da cláusula `else`, ela deve vir na mesma linha que o fechamento de chaves do `then`. Nada mais deve constar nessas linhas. Na saída padrão, o corpo de cada cláusula altera a tabulação em dois espaços em branco cumulativos. Os corpos das expressões condicionais podem ser vazios e podem incluir a declaração de novas variáveis.

4.1 Exemplo de código

```
int a = 0, b = 1, c, d
  bool teste_falso = false
if a > b
then {
if (a > 0)
then {
c = 10
}

}

if teste_falso
then {
d = 0

} else {
d = 20
}
```

4.2 Saída do compilador

```
int var: a = 0, b = 1, c, d
bool var: teste_falso = false
if: > a b
then:
  if: > a 0
  then:
    = c 10
if: teste_falso
then:
  = d 0
else:
  = d 20
```

4.3 Saída do *transpiler*

```
a = 0
b = 1
teste_falso = False
if (a > b):
  if (a > 0):
    c = 10
if teste_falso:
  d = 0
else:
  d = 20
```

4.4 Tratamento de erros

i. Testes devem ser valores booleanos. No caso de um tipo diferente no resultado do teste, um erro semântico deve ser gerado.

```
int a = 0
if a
[Line 2] semantic error: test operation expected boolean but received integer
```

4.5 Saída padrão

```
if:
then:
  ...
{else:
  ...
}?
```

4.6 Saída de erros padrão

```
op: op | test
```

5 Laços

Esta versão inclui laços for em um estilo similar à linguagem C. Os laços seguem a estrutura for inicialização, teste, iteração. O trecho de teste é obrigatório, enquanto os outros trechos são opcionais. A inicialização e a iteração podem conter apenas uma atribuição. O teste pode conter qualquer expressão, desde que booleana. O corpo do laço é definido por chaves, pode ser vazio e pode conter a declaração de novas variáveis. Na saída padrão, o corpo de cada laço altera a tabulação em dois espaços em branco cumulativos.

5.1 Exemplo de código

```
int i, j = 0
for , j < 10 , j = j + 2 {
}
for i = 0, i < 10   , i = i + 1 {
int temp
temp = j + i
j = temp
}
j = j + 0
```

5.2 Saída do compilador

```
int var: i, j = 0
for: , < j 10, = j + j 2
do:
for: = i 0, < i 10, = i + i 1
do:
  int var: temp
  = temp + j i
  = j temp
= j + j 0
```

5.3 Saída do *transpiler*

```
j = 0
while (j < 10):
  j = (j + 2)
i = 0
while (i < 10):
  temp = (j + i)
  j = temp
  i = (i + 1)
j = (j + 0)
```

5.4 Tratamento de erros

Os testes dos laços devem respeitar às mesmas regras que os testes de expressões condicionais e devem gerar os mesmos erros.

5.5 Saída padrão

```
for: , ,
do:
    ...
```

6 Múltiplos escopos

Esta versão adiciona a diferenciação entre escopos. A linguagem trabalha com um escopo global e um novo escopo interno para cada corpo de `then`, `else` e `for`. Dentro de novos escopos, novas variáveis com o mesmo nome de variáveis de escopos externos podem ser declaradas. O tempo de vida das variáveis fica também atrelado à duração do escopo onde elas são declaradas.

6.1 Exemplo de código

```
int i
if true {
  float i = 0.0
}
for i = 0, i < 2, i = i + 2 {
  int a
}
bool a = true
```

6.2 Saída do compilador

```
int var: i
if: true
then:
  float var: i = 0.0
for: = i 0, < i 2, = i + i 2
do:
  int var: a
bool var: a = true
```

6.3 Saída do *transpiler*

```
exec(open('../../../src/scope_manager.py', 'r').read())
s_context()
if True:
  i = 0.0
r_context()
s_context()
i = 0
while (i < 2):
  i = (i + 2)
r_context()
a = True
```

6.4 Tratamento de erros

i. Variáveis somente são visíveis dentro de seus escopos e seus escopos filhos, levando a erros semânticos como listados no item 1.3.3 no caso de tentativas indevidas de uso de variáveis.

7 Funções

Esta versão adiciona as seguintes características relacionadas a funções:

i. Funções podem ser declaradas para uso posterior no código.

ii. Funções podem ser definidas, o que envolve a declaração da assinatura da função e de seu corpo.
iii. Funções podem ser usadas em testes e atribuições.
iv. Funções podem ser declaradas e definidas nos mesmos lugares onde variáveis podem.
v. Funções têm zero ou mais parâmetros.
vi. Ao fim do código, todas as funções declaradas precisam ter sido definidas.
vii. Todas as funções terminam com uma chamada de retorno com a palavra reservada `ret`.
viii. Os nomes de funções seguem as mesmas regras que os nomes de variáveis.
ix. Os corpos de funções possuem os próprios escopos.
x. Os parâmetros da função funcionam de forma similar a variáveis declaradas no corpo da função.
xi. Declarações de funções não geram mensagens de saída.
xii. Na saída padrão, o corpo de cada função altera a tabulação em dois espaços em branco cumulativos.

7.1 Exemplo de código

```
bool fun f ()
bool fun f () {
ret false
}
if f() {
  int a = 0
  int fun f2 ( int x ) {
          int a
         a = x + 1
         ret a
  }
  a = f2 ( a)
}
```

7.2 Saída do compilador

```
int fun: f (params: )
  ret false
if: f[0 params]
then:
  int var: a = 0
  int fun: f2 (params: int x)
    int var: a
    = a + x 1
    ret a
  = a f2[1 params] a
```

7.3 Saída do *transpiler*

```
exec(open('../../../src/scope_manager.py', 'r').read())
def f():
  return False

s_context()
if f():
  a = 0
  def f2(x):
    a = (x + 1)
    return a

  a = f2(a)
r_context()
```

7.4 Tratamento de erros

i. Uma função deve ter apenas um retorno, o qual deve ser feito na última linha de seu corpo. A ausência de um retorno ou a presença de outras linhas de código deve gerar um erro de sintaxe.
ii. Uma função com tipo de retorno diferente do tipo de sua declaração deve gerar um erro semântico.

```
int fun f() {
  ret false
}
[Line 3] semantic error: function f has incoherent return type
```

iii. Uma função declarada mas nunca definida até o fim de seu escopo deve gerar uma mensagem de erro semântico.

```
fool fun myfun()
^D
[Line 2] semantic error: function myfun is declared but never defined
```

iv. No caso de redeclaração ou redefinição de uma função em seu mesmo escopo original, um erro semântico deve ser gerado no mesmo estilo do item 1.3.4.

```
int fun f()
int fun f()
[Line 3] semantic error: re-definition of function f
```

v. No caso de um parâmetro de tipo incompatível, um erro semântico deve ser emitido.

```
int a
int fun f (int x, int y)
a = f (0.0, 0)
[Line 3] semantic error: parameter x expected integer but received float
```

vi. No caso de parâmetros faltantes ou excesso de parâmetros em uma chamada de função, erros semânticos devem ser gerados.

```
bool b
bool fun x (int a__)
b = x()
[Line 3] semantic error: function x expects 1 parameters but received 0
b = x(1, 2, 3)
[Line 3] semantic error: function x expects 1 parameters but received 3
```

vii. No caso de discordância sobre o número de parâmetros entre a declaração e a definição da função, um erro de redefinição deve ser gerado como no item 7.3.3. O mesmo deve ser feito para o caso de discordância entre os nomes do parâmetros.

7.5 Saída padrão

```
typeT fun: nameT (params: {typeT nameT{, typeT nameT}*}?)
ret
nameT[N params]
```

7.6 Saída de erros padrão

```
[Line N] semantic error: function nameT has incoherent return type
[Line N] semantic error: function nameT is declared but never defined
[Line N] semantic error: parameter nameT expected typeT but received typeT
[Line N] semantic error: function nameT expects N parameters but received N
```

8 Arranjos

Esta versão adiciona as seguintes características relacionadas a arranjos:

i. Arranjos de tipos básicos podem ser declarados com qualquer tamanho não-negativo.
ii. Índices de arranjos são sempre valores inteiros.
iii. Declarações de arranjos e acesso a seus elementos são feitos utilizando colchetes.
iv. O operador unário de tamanho `[len]` é suportado, com precedência igual aos operadores de coerção.
v. O operador de inserção `<-` é suportado, aumentando o tamanho do arranjo em 1 na sua utilização.

8.1 Exemplo de código

```
int a[10]
bool parity[10]
int i
for i = 0, i < 10, i = i + 1 {
a[i] = i
if a[i] / 2  == 0 {
    parity[i] = true
} else {
    parity[i] = false
}
}
```

8.2 Saída do compilador

```
int array: a (size: 10)
bool array: parity (size: 10)
for: = i 0, < i 10, = i + i 1
do:
  = [index] a i i
 if: == / [index] a i 2 0
  then:
    = [index] parity i true
  else:
    = [index] parity i false
```

8.3 Saída do *transpiler*

```
exec(open('../../../src/scope_manager.py', 'r').read())
a = [0] * 10
parity = [0] * 10
s_context()
i = 0
while (i < 10):
  a[i] = i
  s_context()
  if ((a[i] / 2) == 0):
    parity[i] = True
  else:
    parity[i] = False
  r_context()
  i = (i + 1)
r_context()
```

8.4 Tratamento de erros

i. No caso de um índice de um tipo outro que inteiro, um erro semântico deve ser gerado.

```
int a[10]
a[false] = 1
[Line 2] semantic error: index operation expects integer but received boolean
```

ii. O operador de inserção espera um tipo compatível com o tipo do arranjo. Caso contrário, um erro semântico deve ser gerado.

```
bool b[10]
b <- 1.2
[Line 3] semantic error: append operation expected boolean but received float
```

iii. No caso da declaração de um arranjo com um tamanho que não seja um valor inteiro, um erro de sintaxe deve ser gerado.

iv. Operações onde o tamanho do arranjo à direita é maior do que o tamanho do arranjo à esquerda devem gerar um erro semântico.

```
int a[9], b[10]
bool c
c = a != b
[Line 4] semantic error: operation between mismatched array sizes
```

v. Os operadores de índice e inserção esperam um arranjo no lado esquerdo da operação.

```
int a, b
b = a[1]
[Line 2] semantic error: left hand side of index operation is not an array
```

vi. O operador unário de tamanho do arranjo espera um arranjo.

```
int a, b
b = [len] a
[Line 4] semantic error: length operation expects an array
```

8.5 Saída padrão

```
typeT array: nameT (size: N)
[index]
[len]
[append]
```

8.6 Saída de erros padrão

```
op: op | index | length | append
[Line N] semantic error: index operation expects integer but received { boolean | float }
[Line N] semantic error: operation between mismatched array sizes
[Line N] semantic error: left hand size of { index | append } operation is not an array
[Line N] semantic error: length operation expects an array
```

9 Ponteiros

Esta versão inclui a declaração e uso de ponteiros sobre os tipos básicos, arranjos e ponteiros. Ponteiros são identificados pela palavra reservada `ref`. O `lvalue` de qualquer variável pode ser acessado pelo operador `addr`. Referências não são feitas para arranjos nem funções, sendo usadas apenas para variáveis e itens de arranjos. Ponteiros podem ser usados como parâmetros de funções e retorno de funções.

9.1 Exemplo de código

```
int a[10]
int i = 0
int ref mypointer
int ref pointers[2]
mypointer = addr i
i = ref mypointer + 1
pointers[0] = mypointer
pointers[1] = addr a[3]
int ref ref doublepointer
doublepointer = addr pointers[0]
```

9.2 Saída do compilador

```
int array: a (size: 10)
int var: i = 0
int ref var: mypointer
int ref array: pointers (size: 2)
= mypointer [addr] i
= i + [ref] mypointer 1
= [index] pointers 0 mypointer
= [index] pointers 1 [addr] [index] a 3
int ref ref var: doublepointer
= doublepointer [addr] [index] pointers 0
```

9.3 Saída do *transpiler*

```
exec(open('../../../src/scope_manager.py', 'r').read())
a = [0] * 10
i = 0
pointers = [0] * 2
mypointer = i
i = (mypointer + 1)
pointers[0] = mypointer
pointers[1] = a[3]
doublepointer = pointers[0]
```

9.4 Tratamento de erros

i. No caso de operações binárias com tipos diferentes, erros semânticos devem ser emitidos.

```
int i = 0
int ref point
point = i
[Line 3] semantic error: attribution operation expects integer pointer but received integer
```

ii. O operador unário `ref` espera um valor que seja ponteiro de algo. Caso o valor não seja um ponteiro, um erro semântico deve ser disparado.

```
int i
i = ref 0
[Line 2] semantic error: reference operation expects a pointer
i = ref i
[Line 3] semantic error: reference operation expects a pointer
```

iii. O operador unário `addr` espera uma variável ou item de arranjo. Caso contrário, um erro semântico deve ser gerado.

```
bool ref b
b = addr true
[Line 0] semantic error: address operation expects a variable or array item
```

9.5 Saída padrão

```
typeT {ref}* var: nameT
[addr]
[ref]
```

9.6 Saída de erros padrão

```
[Line N] semantic error: op operation expects typeT {pointer}* but received typeT {pointer}*
[Line N] semantic error: reference operation expects a pointer
[Line N] semantic error: address operation expects a variable or array item
```

10 Caracteres e cadeias destes

Esta versão adiciona as seguintes características à linguagem:

i. Tipo de caractere `char` com possíveis valores ASCII no intervalo 32_{10} (espaço) e 126_{10} (~), inclusive, exceto aspas simples (') e duplas (").
ii. Um `char` pode ser declarado utilizando aspas simples, e não pode ser vazio.
iii. Um arranjo de `char` pode ser declarado utilizando aspas duplas, e pode ser a palavra vazia.
iv. Acontecerá coerção para arranjos de `char` com o operador `[word]` na operação entre dois caracteres, ou um caractere e um arranjo de caracteres.
v. Palavras com valores além da capacidade de seus tipos (1 para `char`, n para `char array`) serão truncadas.
vi. Suporte a comentários de linha única, que iniciam com o símbolo `#`, em qualquer parte da linha.
vii. Não existem restrições quanto às operações entre caracteres e cadeias.

10.1 Exemplo de código

```
char let = 'a'
char wd[4], result[10]
wd = "luka"
# i can add letters together
result = wd + let
let = result[1]
```

10.2 Saída do compilador

```
char var: let = 'a'
char array: wd (size: 4), result (size: 10)
= wd "luka"
= result + wd [word] let
= let [index] result 1
```

10.3 Saída do *transpiler*

```
exec(open('../../../src/scope_manager.py', 'r').read())
let = 'a'
wd = [0] * 4
result = [0] * 10
wd = "luka"
result = (wd + str(let))
let = result[1]
```

10.4 Tratamento de erros

i. Caso um caractere seja declarado com mais de uma letra, seu valor deve ser truncado para a primeira letra, e uma advertência deve ser gerada.

```
char a = 'as'
[Line 2] warning: value truncated to 'a'
```

ii. Uma cadeia de caracteres inicializada com mais letras do que sua declaração original deve ser truncada, e uma advertência deve ser gerada.

```
char a[5]
a = "too big of a word"
[Line 3] warning: value truncated to "too b"
```

10.5 Saída padrão

Note que todos os comentários serão ignorados pela ferramenta `flex`.

```
charT: [\'][ !#-&(-~]+[\']
wordT: [\"][ !#-&(-~]*[\"]
typeT: typeT | char
type: type | character
[word]
```

10.6 Saída de erros padrão

```
[Line N] warning: value truncated to X
```

11 Funções anônimas e de alta ordem

Esta versão adiciona as seguintes características à linguagem:

i. `lambda` e a respectiva letra minúscula em grego, λ (`U+03BB`), tornam-se palavras reservadas da linguagem, bem como o símbolo `->`.
ii. Funções anônimas devem conter no mínimo um (1) argumento.
iii. Uma função anônima pode ser descartada com a diretiva λ`()`, para que outra possa ser declarada.

iv. `map`, `fold` e `filter` tornam-se palavras reservadas da linguagem.
v. Funções de alta ordem devem ter obrigatoriamente dois argumentos, nesta ordem: uma função anônima e um arranjo, chamados respectivamente de λ e `it` neste documento para facilitar a visualização das regras.
vi. Os tipos dos parâmetros devem ser coerentes com o tipo da função de alta ordem.
vii. `map` sempre retornará um arranjo de tipo e tamanho igual a `it`. Sua função anônima deve ter um (1) parâmetro.
viii. `fold` sempre retornará um elemento de tipo igual ao tipo primitivo de `it`. Sua função anônima deve ter dois (2) parâmetros.
ix. `filter` sempre retornará um arranjo de tipo igual a `it`, mas não necessariamente com o mesmo tamanho. Ademais, seu primeiro argumento deve ser sempre uma função anônima booleana com um (1) parâmetro.
x. Para evitar erros de redefinição, as funções de alta ordem levam como prefixo o nome de seu segundo parâmetro seguido de um traço inferior.
xi. Podem ser utilizadas apenas em atribuições para novas variáveis.
xii. Como o código que simula o funcionamento das funções de alta ordem é gerado automaticamente, erros em cascata são comuns caso as restrições não sejam obedecidas.

11.1 Exemplo de código

```
int ref t[10], output[10]
output = map(lambda int ref x -> x, t)
```

11.2 Saída do compilador

```
int ref array: t (size: 10), output (size: 10)
int ref array fun: t_map (params: int ref array t)
  int ref fun: lambda (params: int ref x)
    ret x
  int var: t_ti
  int ref array: t_ta (size: 10)
  for: = t_ti 0, < t_ti [len] t, = t_ti + t_ti 1
  do:
    = [index] t_ta t_ti lambda[1 params] [index] t t_ti
  ret t_ta
= output t_map[1 params] t
```

11.3 Saída do *transpiler*

```
exec(open('../../../src/scope_manager.py', 'r').read())
t = [0] * 10
output = [0] * 10
def t_map(t):
  def U+03BB(x):
    return x

  t_ta = [0] * 10
  s_context()
  t_ti = 0
  while (t_ti < len(t)):
    t_ta[t_ti] = U+03BB(t[t_ti])
    t_ti = (t_ti + 1)
  r_context()
  return t_ta

output = t_map(t)
```

11.4 Tratamento de erros

i. Funções anônimas e de alta ordem com número de parâmetros ilegal geram erros sintáticos.
ii. Funções de alta ordem cujos λ têm número de parâmetros ilegal geram erros semânticos.

```
int t[10], output[10]
output = map(lambda int x, y -> x + 2, t)
[Line 6] semantic error: function lambda expects 2 parameters but received 1
[Line 8] semantic error: map's lambda expects 1 parameters but received 2
```

```
...
```

iii. Caso `it` não seja um arranjo, um erro semântico deve ser gerado.

```
int t
int output[10]
output = map(lambda int x -> x + 2, t)
[Line 4] semantic error: high order function's second parameter must be of array type
...
```

11.5 Saída padrão

```
funcH: map | fold | filter
func: funcH | lambda
U+03BB
```

11.6 Saída de erros padrão

```
[Line N] semantic error: high order function's second parameter must be of array type
[Line N] semantic error: funcH's lambda expects N parameters but received N
```

12 *Transpiler* para Python

Este recurso é adicionado à linguagem, com as seguintes considerações:

i. Python não necessita de tipos junto à declaração da linguagem, então declarações sem inicializações não fazem sentido e podem ser descartadas.

ii. Todas as operações são cercadas de parênteses para que seus significados sejam o mais próximo possível da linguagem aqui descrita.

iii. Todos os códigos gerados iniciam com um cabeçalho que importa um par de funções utilizadas automaticamente para que os escopos sejam corretamente simulados.

iv. Funções declaradas mas não definidas não existem em Python; portanto, a palavra chave `pass` é utilizada para inicializar uma função sem qualquer utilidade.

v. Como a ideia de ponteiros não existe em Python, não existe uma possível conversão válida. Portanto, todos os operadores `addr` e `ref` são retirados.

vi. Python não contém qualquer estrutura de iteração similar a um `for` estilo C, então todos os `for` são traduzidos para estruturas `while`.