Proposta de extensão para a linguagem Łukasiewicz

Douglas Martins
arquiteto de sistema

Gustavo Zambonin
gerente de projeto, projetista de sistema

Marcello Klingelfus
testador

Construção de Compiladores (UFSC-INE5426)

1 Funções anônimas e de alta ordem

1.1 Definições

Adiciona-se à linguagem suporte a funções anônimas (não depende de um identificador para ser utilizada) e funções de alta ordem (toma uma ou mais funções como parâmetro) denominadas `map`, `fold` e `filter`.

Funções anônimas respeitam as seguintes restrições:

- `lambda` e a respectiva letra minúscula em grego, λ (`U+03BB`), tornam-se palavras reservadas da linguagem;
- devem conter no mínimo um (1) argumento;
- a função pode ser utilizada da seguinte maneira: $\lambda(a_1, a_2, \ldots, a_n)$, onde a_n é o n-ésimo argumento da função previamente declarada.
- uma função anônima pode ser descartada com a diretiva $\lambda()$, para que outra possa ser declarada;
- opera apenas sobre tipos básicos da linguagem (números inteiros e de ponto flutuante, booleanos e caracteres).

Funções de alta ordem respeitam as seguintes restrições:

- `map`, `fold` e `filter` tornam-se palavras reservadas da linguagem;
- devem ter obrigatoriamente dois argumentos, nesta ordem: uma função anônima e um `array`, chamados respectivamente de λ e `it` neste documento para facilitar a visualização das regras;
- será necessário estender a linguagem com a adição de um operador unário `len`, que retorna o tamanho do `array`, para que o código gerado nas funções de alta ordem seja semanticamente correto;
- o tipo de retorno de `map` deve ser igual ao tipo de `it`, que deve ter tipo primitivo igual ao tipo de retorno de λ;
- o tipo de retorno de `fold` deve ser igual ao de λ e ao tipo primitivo de `it`;
- o tipo de retorno de `filter` deve ser igual ao tipo de `it`, e λ deve ser sempre uma função booleana;
- para evitar erros de redefinição, as funções de alta ordem levam como prefixo o nome de seu segundo parâmetro seguido de um traço inferior;
- arrays, índices e variáveis temporários devem ser criados adicionando os sufixos `_ta` e `_ti` e `_tv` ao nome original do parâmetro, respectivamente.

1.2 Exemplo de código

```
int t[10], output[10]
output = map(lambda int x -> x + 2, t)
```

1.3 Saída esperada

```
int array: t (size: 10), output (size: 10)
int array fun: t_map (params: int array t)
  int fun: lambda (params: int x)
    ret + x 2
  int var: t_ti
  int 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
```

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

2 Suporte a caracteres e cadeias destes

2.1 Definições

Os seguintes recursos são adicionados à linguagem:

- suporte a algumas operações com caracteres ASCII entre 32_{10} (espaço) e 126_{10} (~), inclusive;
- variáveis com apenas um caractere, de tipo `char`, podem ser declaradas com aspas simples;
- variáveis com múltiplos caracteres, de tipo `str` podem ser declaradas com aspas duplas. Considera-se que um `str` é similar a um `char array`;
- o operador de adição causa coerção se utilizado entre `char` e `char`, ou `char` e `str` (independente da ordem);
- os operadores de comparação (maior, menor etc.) podem ser utilizados entre estes dois tipos;
- o operador de índice pode ser utilizado em um `str`; esta variável é tratada como um `char`;
- comentários podem ser feitos em linhas individuais começando com `#`. Os caracteres válidos são os mesmos citados acima. Serão ignorados pelo *parser* e não serão mostrados na representação intermediária de código.

2.2 Exemplo de código

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

2.3 Saída esperada

```
char var: let = 'a'
str var: wd = "luka", result
= result + wd [str] let
= let [index] result 1
```

3 Geração de código intermediário com LLVM

3.1 Roteiro

Propõe-se compilar código-fonte de Łukasiewicz para representação intermediária de LLVM (LLVM IR). A fragmentação desta tarefa será definida de forma similar ao Projeto 1:

i. linguagem introdutória
ii. ponto flutuante e valores booleanos
iii. conversão de tipos
iv. caracteres e cadeias
v. expressões condicionais
vi. laços
vii. múltiplos escopos
viii. funções (usuais, anônimas e de alta ordem)
ix. arranjos
x. ponteiros

4 *Transpiler* de Łukasiewicz para Python

4.1 Roteiro

Adicionalmente, propõe-se compilar código-fonte de Łukasiewicz para código Python, de modo a testar seu funcionamento e ver resultados reais sem a implementação de um interpretador completo. O roteiro é similar ao anterior. As características diferentes das linguagens fonte e destino devem ser levadas em conta, e assim, alguns comportamentos poderão ser simulados de modo diferente:

- tipagem dinâmica em Python faz com que as variáveis não precisem ser declaradas junto a seus tipos;
- as palavras reservadas `True` e `False`, em Python, podem ser operadas como inteiros de valor `0` e `1`;
- funcionamento de escopo léxico deve ser retrabalhado para que variáveis sejam terminadas ao fim de um bloco lógico (e.g. `if`);
- outros comportamentos serão documentados se descobertos.