Projeto de Compiladores – Etapa 6 (E6)

Rodrigo Kassick

2016-2

1 Descrição

Implementar um **interpretador** para a regra `funcbody` da linguagem μmML.
As seguintes construções devem ser corretamente implementadas:

- Declarações de Variáveis com `let`
- Fórmulas (`metaexpr`) que operam sobre tipos *inteiros*
- *Cast* de números inteiros para string
- *Cast* de *string* para *número inteiro*
- Concatenação de Strings

As seguintes construções **não são necessárias** nesta etapa:

- Operações com tipos `float` ou `char`
- Expressões *booleanas*
- Construção `if`

1.1 Importante

Uma variável declarada com `let` deve ter um *tipo* correto e seu *valor* deve estar armazenado em algum local.
Utilize uma **pilha** de Objetos (`Object` (Java) ou `object` (C#)) para armazenar os valores.

2 Dicas

- Use uma pilha de *Object*. Qualquer objeto (String, Integer) poderá ser armazenado nessa pilha.
- Não é necessário conferir os tipos dos objetos no topo da pilha antes de (no seu código do interpretador) fazer *cast* para um objeto `Integer` ou `String` – o controle de tipo feito na etapa anterior já garante que o *cast* irá funcionar.
- Um *símbolo* agora deve estar associado a um **tipo** e a um **valor**. Se na etapa anterior foi utilizada uma tabela de símbolos declara como

  ```
  NestedSymbolTable<Tipo> simbolosAtuais;
  ```

  você pode substituir por uma `NestedSymbolTable<EntradaSimbolo>`, onde `EntradaSimbolo` armazena um `Tipo` e um `Object`. Outra opção é gerenciar duas tabelas de símbolos:

  ```
  NestedSymbolTable<Tipo> simbolosAtuais;
  NestedSymbolTable<Object> simbolosValores;
  ```

  Ao declarar um símbolo, se armazena em `simbolosAtuais` o `Tipo` associado à variável e em `simbolosValores` o **valor** do símbolo (o `Object` no topo da pilha).

- Ao armazenar o valor de um símbolo na tabela, o `Object` do topo da pilha é **consumido**, i.e., seu código executará algo equivalente a:

```
// Primeiro: armazena em simbolosAtuais o *tipo* do simbolo
// ...

// Armazena o valor do simbolo
Object v = pilhaValores.pop();
simbolosValores.store($symbol.text, v);
```

- Quando uma `metaexpr` deriva para `symbol`, deve-se consultar a tabela de símbolos por aquele nome e **empilhar** o valor, i.e., seu código deve executar algo equivalente a:

```
SymbolEntry<Object> e = simbolosValores.lookup($symbol.text);
if (e != null) {
    pilhaValores.push(e.symbol); // uma SymbolEntry possui um
                                 // campo ''symbol'' que aponta
                                 // para o que foi armazenado
                                 // no store
} else {
    // ERRO!
    throw new Exception("Variavel desconhecida: " + $symbol.text);
}
```

- Todas as operações *binárias* (soma, subtração, divisão, multiplicação, concatenação) **consomem dois símbolos** da pilha e empilham um resultado, i.e. seu código fará algo equivalente a

```
// Soma: apenas inteiros. Verificacao de tipos ja garantiu que nao
// serao somados operandos strings
Integer v1 = (Integer)( pilhaValores.pop() );
Integer v2 = (Integer)( pilhaValores.pop() );
pilhaValores.push( new Integer(v1.intValue() + v2.intValue() ) );
```

- A expressão `let` deve **avaliar** a expressão que aparece no seu corpo (após o `in`). Ao fim da avaliação, o resultado estará na pilha e pode ser "devolvido" a quem chamou a regra `funcbody`:

```
letexpr
returns [Tipo tipo]
    : 'let'
    { /* Prepara a uma nova tabela de simbolos */
        // ...
        simbolosAtuais = new NestedSymbolTable<Entrada>(simbolosAtuais);
        simbolosValores = new NestedSymbolTable<Entrada>(simbolosValores);
        // ...
    }
    letlist                 // letlist atualiza currentSymbols
    'in'
    funcbody                // funcbody usa
                            // currentSymbols. Eventualmente, deixa
                            // algo no topo da pilha de dados. alguem
                            // pode usar esse valor, como se a letexpr
                            // fosse uma operacao que recebe uma expressao
                            // e devolve seu resultado
    {
        $tipo = $funcbody.tipo;
        /* Restaura a tabela de simbolos anterior */
        simbolosAtuais = simbolosAtuais.getParent();
        simbolosValores = simbolosValores.getParent();
    }

    #letexpression_rule
```

- Como ao fim de uma expressão `let` haverá um resultado no topo da pilha e ela terá um tipo associado, pode-se atribuir o *resultado* do `let` a uma variável:

```
let x = let y = 2
        in y * y + 2
in x + x
```

- Lembre-se que uma regra `let` será derivada a partir de funcbody. Cuide para repassar o *tipo* avaliado a partir de uma `metaexpr` para o `funcbody` do qual ela veio **e** de repassar o *tipo* do `funcbody` para o `letexpr` e do `letexpr` para o `funcbody` que o chamou.
  - `funcbody` $\Rightarrow$ `letexpr` $\Rightarrow$
  - *'let'* `letlist` *'in'* `funcbody` $\Rightarrow^{+}$
  - *'let'* *x* '=' *1 + 1* *'in'* `funcbody` $\Rightarrow$
  - *'let'* *x* '=' *1 + 1* *'in'* `metaexpr`

3 Exemplos de Entrada e Resultados Esperados

3.1 Exemplo 1

```
let x = 10,
    y = 20
in
    x + y * x
```

Resultado esperado no topo da pilha: 210

3.2 Exemplo 2

```
let x = "um",
    y = "dois"
in
    x :: y
```

Resultado esperado no topo da pilha: `"umdois"`

3.3 Exemplo 3

```
let x = let s1 = "inicio",
            s1 = "fim",
        in
            s1 :: (str 1000) :: s2
in
    "Resultado: " :: x
```

Resultado esperado no topo da pilha: `"Resultado:  inicio1000fim"`

3.4 Exemplo 4

```
let x = 1951
in
    let x3 = x / 1000,
        x3mod = x - ((x / 1000) * 1000),          // x mod 1000
        x2 = x3mod / 100,
        x2mod = x3mod - ((x3mod / 100) * 100),    // x mod 100
        x1 = x2mod / 10,
        x1mod = x2mod - ((x2mod / 10) * 10),      // x mod 10
        x0 = x1mod
    in
       (str x3) :: (str x2) :: (str x1) :: (str x0)
```

Resultado esperado no topo da pilha: `"1951"`

3.5 Exemplo 4

```
let x = int (let x2 = "9", x1 = "8", x0 = "7"
             in x2 :: x1 :: x0)
in
   x - 1
```

Resultado esperado no topo da pilha: 996