DeceptiCompiler

Mais do que os olhos podem ver.

P1 de Compiladores (2018.1), por Raffael Paranhos, Wallace Baleroni e Júlia Falcão

25 de Abril de 2018

Universidade Federal Fluminense

Parser

Classe `OptimusParser < Parslet::Parser`, contendo regras do tipo:

```
rule(:exp) { mathexp | boolexp | integer | ident }
```

Uma "expressão" pode ser uma exp. matemática, booleana, um inteiro ou um identificador (uma variável).

```
rule(:ini) { ident >> ini_op >> exp }
```

Uma inicialização é composta por um identificador, o operador e uma "expressão".

A regra na qual o parser inicia o processo é declarada por:

```
root(:ex_proc)
```

Parser

- **Expressão aritmética**:

```
rule(:mathexp) { (ident | integer) >> arithop
>> (mathexp | ident | integer) }
```

- **Expressão booleana**:

```
rule(:boolexp) { neg_op.maybe >> (ident | integer)
>> boolop >> exp }
```

- **Comando**:

```
rule(:cmd) { (cmd_unt >> cho_op >> cmd |
cmd_unt >> seq_op >> cmd | cmd_unt) }

rule(:cmd_unt) { ex_if | ex_while | ex_print |
ex_exit | call | ident >> ass_op >> exp }
```

Parser

Executando o parse de um módulo inteiro:

```
OptimusParser.new.rollOut("
    module Fact
    var y, x
    init y = 1

    proc fact(x) {
        if (~ x == 0) {
            y := x * y ; y := x + 1
        }
    }
    end
");
```

Transform

A classe `Bumblebee < Parslet::Transform` recebe a saída do parser e determina como e onde cada parte do código será empilhada, inicializando a estrutura SMC.

Para fazer isso, são usados *aliases* no código do `OptimusParser` que identificam as partes de cada expressão/comando para que o transform as reconheça.

```
rule(:mathexp) { (ident | integer).as(:left) >> arithop
>> (mathexp | ident | integer).as(:right) }

rule(:sum_op) { str("+").as(:op) >> blank? }
```

Transform

Nessa classe `Bumblebee`, são implementadas estruturas para cada tipo de expressão/comando, que avaliam os componentes e constroem o `SMC` correspondente.

```
Addition = Struct.new(:left, :right) do
  def eval
    $smc.empilhaControle('add')
    $smc.empilhaControle(left.eval)
    $smc.empilhaControle(right.eval)
  end
end
```

No caso de uma adição, por exemplo, empilhamos na pilha de controle a string "add", o lado esquerdo da expressão, e por fim o lado direito.

SMC

A classe SMC é usada para representar uma instância da estrutura <S, M, C>, que tem como atributos as pilhas de controle, de valor e a lista de memória. Nessa lista, um item é o nome do identificador e o item seguinte é seu valor atual guardado em memória.

Exemplo:

```
@memoria = ["x", 5, "y", 1]
```

A classe também define as operações para manipular a estrutura, ou seja, empilhar e desempilhar itens nas pilhas, acessar a memória e escrever nela.

BPLC

A classe `BPLC` possui o método `vamosRodar` que recebe uma instância de `SMC` e vai desempilhando o que está nas pilhas e executando as operações devidas.

```
val = smc.topoControle()
if (val == 'assign') smc.ce()
```

smc.rb

```
def ce()
  self.desempilhaControle()
  val = self.desempilhaValor()
  ident = self.desempilhaValor()
  self.escreveMemoria(ident, val)
end
```

Execução

Exemplo:

```
code = "x := 2"  # código em IMP

$smc = SMC.new  # cria instância de SMC

bplc = BPLC.new  # cria instância de BPLC

Bumblebee.new.apply(OptimusParser.new.rollOut(code)).eval
# roda o parser no código em IMP e passa a saída para o método
que aplica a transformação e constrói o SMC

bplc.vamosRodar($smc)
# chama o método de BPLC que vai executar o programa a partir
do SMC
```

Saída

- **Saída do parser**

```
{:cmd => {:ident => {:id => "x" @0},
:ass_op => ":= " @2, :val => {:int => "2" @5}}}
```

- **Situação do SMC a cada passo**

```
Controle   ["x", "ident", 2, "assign"]
Valor      [1, 1, "x"]
Memoria    ["x", 1, "y", 720]
```

- **Regras**

```
E op E
C := E
```

- **Comandos de print**

```
Print
720
```

Makefile

O comando `make` instala as dependências necessárias para o compilador funcionar e roda o teste do fatorial de $x = 6$.

Testes adicionais podem ser executados digitando `make [test]`, e os disponíveis são `add`, `sub`, `mul`, `div`, `if`, `while`.

`testes/add.rb`, por exemplo, contém um procedimento simples em IMP que executa algumas adições e imprime o resultado. `make add` executa esse teste e imprime o retorno.