CariocaScript

- **Course:** INF1022
- **Semester:** 2019.2
- **Supervisor:** Edward Hermann Haeusler
- **Group members:**
  - *Geraldo Luiz de Carvalho Pereira Junior*

Seções

Aquecendo os tambores - Instalando os pré-requisitos

Para rodar o projeto é necessário ter os pacotes *Flex* e *Bison* instalados, o que pode ser feito através da linha de comando no Ubuntu:

```
$ sudo apt install flex
$ sudo apt install bison
```

E no Fedora:

```
$ sudo yum install flex.x86_64
$ sudo yum install bison.x86_64
```

Pisando fundo - Rodando seu primeiro programa

Para rodar o programa é muito simples, é só rodar o programa "Coe" passando um arquivo com extensão ".cara" por argumento, como por exemplo:

```
$ ./Coe tests/t3-nested_operations.cara
```

Vemos abaixo o programa `"t3-nested_operations.cara"`:

```
----------------------------
CHEGAMAIS X,Y,Z NAMORAL

SEPA X TA_LGD
  MARCA Y RAPIDAO
    SEPA X TA_LGD
      X--
    VALEU
```

```
    Z++
  VALEU
SENAO
    X++
VALEU


FALATU(X)
FALATU(Y)
FALATU(Z)

VALEU
------------------------------
```

A execução do programa com o input de X,Y e Z declarado como 1,2 e 3, devolve o seguinte output:

```
CariocaScript 1.0 Copyright (C) 2019 PUC-Rio
Falai meu consagrado, que que você manda?
Meu Brother: 1 2 3

Meu Parcerasso:0
Meu Parcerasso:2
Meu Parcerasso:5
```

Verificamos o output do programa com a string "Meu Parcerasso". Resultante do comando `FALATU(id)`(equivalente do print), que imprime respectivamente, o valor de X,Y e Z que passamos como entrada.

Queimando a mufa - Modificando a gramática

Concertando conflitos Shift-Reduce

Aqui eu relato as mudanças que fiz sobre a Provol-One, linguagem proposta no cabeçalho do trabalho:

```
program -> ENTRADA varlist SAIDA varlist cmds FIM
varlist -> id varlist | id
cmds -> cmd cmds | cmd
cmd -> FACA id VEZES cmds FIM
cmd -> ENQUANTO id FACA cmds FIM
cmd -> SE id ENTAO cmds SENAO cmds | SE id ENTAO cmds
cmd -> id = id | INC(id) | ZERA(id)
```

Tendo uma gramática ambígua, duas mudanças formais precisaram ser feitas na ProvolOne para evitar conflitos de Shift/Reduce:

1. Inverter a ordem da chamada de si mesmo em comandos recursivos como `cmds` e `var_list` da seguinte forma:

```
cmds: cmd cmds (Antes)
cmds: cmds cmd (Depois)

var_list: var_dev var_list (Antes)
var_list: var_list var_def (Depois)
```

Isso é necessário pois no caso da recursão a direita, todos os elementos precisam ser colocados na pilha antes da regra poder ser aplicada e os elementos poderem ser reduzidos.(1)

2. Distinguir o comando `SE var_ref ENTAO cmds` de `SE var_ref ENTAO cmds SENAO cmds`, para acabar com a ambiguidade entre eles. O que pode ser feito terminando cada expressão com o token `FIM`.

Após as alterações iniciais que tornaram a linguagem funcional, a gramática passou por outra transformação para se tornar a CariocaScript:

```
program -> CHEGAMAIS input NAMORAL cmds VALEU
input -> var_list
var_list -> var_list var_def
var_def -> id
var_ref -> id
cmds -> cmds cmd | cmd
cmd -> MARCA var_ref RAPIDAO cmds VALEU
cmd -> ENQUANTO var_ref FACA cmds VALEU
cmd -> SEPA var_ref TA_LGD cmds SENAO cmds | SE var_ref VALEU cmds
cmd -> var_ref = var_ref | var_ref++ | id-- | RELAXOU(var_ref) |
FALATU(var_ref) | var_ref += var_ref
| var_ref -= var_ref
```

Sendo uma linguagem informal, cabe esclarecer algumas keywords:

1. `ENTRADA` é substituída pelo `CHEGAMAIS`.
2. `SAIDA` é substituída pelo `FALATU(id)`, que exerce a função de printar o valor da variável, de tal forma que o programador possa escolher as saidas do programa de maneira mais flexível.
3. `FACA id VEZES cmds FIM` é substituído pelo `MARCA id RAPIDAO cmds VALEU`.
4. `SE id ENTAO cmds VALEU` é substituído pelo `SEPA id TA_LGD cmds VALEU`, onde lê-se `TA_LGD` como "tá ligado?".
5. `ZERA(id)` é substituído pelo operador `RELAXOU(id)`.
6. E por `FIM`, há o operador `VALEU` que encerra todos os comandos não terminais para evitar ambiguidade.

Arrancando os cabelos - Geração de Código

Vemos abaixo o pseudo-código gerado pela Provol-One:

```
q0 COPIA(3,2) q1
q1 COPIA(4,1) q2
q2 ZERA(5) q3
q3 INC(3) q4
q4 INC(5) q5
q4 IF(5,4) q6,q3
q6 FIM
```

Sendo muito próximo ao assembly, a CariocaScript fez esse pulo e se tornou executável. Ao rodar o programa `Coe`:

1. Recompila-se o projeto através do arquivo `Makefile` e gera-se o executável.
2. Encaminha-se o arquivo "`.cara`" passado por argumento para o arquivo binário `bin/CariocaScript`, que por sua vez gera o arquivo assembly.
3. Compila-se o arquivo assembly junto de uma main no diretório `/program` que é encarregada de chamar a função CariocaScript, que consiste em nosso programa "`.cara`" e por isso não retorna nem recebe nenhum valor.

Contagem de Labels e controle de fluxo

A geração de código da CariocaScript se dá em puro assembly. A maior dificuldade encontrada em montar um bloco de código em assembly é na tradução das instruções de controle de fluxo de execução. O sistema é na bem simples na no caso da tradução do `SEPA` por exemplo:

```
cmpl $0,-8(%rbp)
je L1
addl $1,-8(%rbp) //Instrução caso verdadeiro
L1:
```

Em seguida incrementa-se em um o contador de labels. Se isso for feito a cada leitura de um `cmd`, e se na construção de um bloco de código, também for printado o valor da label vigente incrementando mais um, tudo parece funcionar. Me refiro ao caso do `SEPA id TA_LGD cmds SENAO cmds VALEU`, onde é necessário ao menos duas labels:

```
sprintf(s_if,  "         cmpl $0,-%d(%rbp) \n\t "
               "         je L%d\n",getRbpOffset($2),label);

-----------Caso primeira condição verdadeira----------------

sprintf(s_else,"      \t jmp L%d\n"
               "     L%d:\n",label+1,label);
```

```
----------------------------Else----------------------------

sprintf(s_exit,"     L%d:\n",label+1);
```

Administração de variáveis

Outra curiosidade é a administração de variáveis, que se dá através de alocação na pilha através de um cálculo simples. Todas as operações são feitas referenciando a pilha, usando um registrador callee-saved quando necessário, para contornar operações que são ilegais para dois endereços simultâneos, como `movl -8(%rbp), -16(%rbp)`(corresponderia a instrução `id = id`):

```
movl -8(%rbp), %r12d
movl %r12d, -16(%rbp)
```

Em suma, usa-se o %r12d para intermediar operações aritméticas com variáveis do usuário, e o %r13d para intermediar operações aritméticas com contadores criados pela linguagem, como no caso do comando `MARCA id RAPIDAO cmds VALEU`:

```
movl $0,%r13d              //i==0
L1:
   addl $1,%r13d           //i++
   addl $1, -16(%rbp)      //Y++
   cmpl %r13d,-8(%rbp)     //X==Y
   jne L1
```

Caso contrário, se utilizássemos o mesmo registrador para ambos o casos, uma atribuição dentro de um comando `MARCA id RAPIDAO`, iria sobreescrever o contador de iterações. Por outro lado, se o contador fosse armazenado na pilha, a comparação poderia ser realizada pelo %r12d. Por hora, não há necessidade de economizar registradores, e o código gerado fica mais enxuto.

Exemplo de geração de código

Teste t3-nested_operations.cara

Para o programa mencionado na seção `Pisando Fundo`, temos o seguinte código assembly:

```
.globl  cariocaScript
 Si:  .string "Meu Brother: "

 Sii:  .string "%d"

 Nl:  .string "\n"
```

```
 Sf:  .string "Meu Parcerasso:%d\n"

 cariocaScript:
   pushq %rbp
   movq %rsp,%rbp
   subq $32, %rsp

   movq $Si, %rdi
   call printf
   movq $Sii, %rdi
   leaq -8(%rbp), %rsi
   call scanf

   movq $Sii, %rdi
   leaq -16(%rbp), %rsi
   call scanf

   bovq $Sii, %rdi
   leaq -24(%rbp), %rsi
   call scanf

  ddl $1, -8(%rbp)
L1:
   jmp L4
L3:
   addl $1, -16(%rbp)
L4:
     movl $0,%r13d
L5:
     addl $1,%r13d
   addl $1, -24(%rbp)
     cmpl %r13d,-8(%rbp)
   jne L5

L7:
   cmpl $0, -24(%rbp)
   je L8
   cmpl $0,-8(%rbp)
   je L6
   addl $1, -8(%rbp)
L6:
   subl $1, -24(%rbp)
   jmp L7
L8:
 movq $Nl, %rdi
   call printf
```

```
   cmpl $0,-8(%rbp)
   je L4
   movl $0,%r13d
L2:
   addl $1,%r13d
   cmpl $0,-8(%rbp)
   je L1
   subl $1, -8(%rbp)
L1:
   addl $1, -24(%rbp)
   cmpl %r13d,-16(%rbp)
   jne L2

   jmp L5
L4:
   addl $1, -8(%rbp)
L5:
   movq $Sf, %rdi
   movl -8(%rbp), %esi
   call printf

   movq $Sf, %rdi
   movl -16(%rbp), %esi
   call printf

   movq $Sf, %rdi
   movl -24(%rbp), %esi
   call printf

   movq %rbp, %rsp
   popq %rbp
   ret
```

Teste t2-all_operations.cara

A função desse teste é ser abrangente e usar todos os comandos.

```
CHEGAMAIS X, Y, Z, A, B NAMORAL
SEPA X TA_LGD
  SEPA Y TA_LGD
    X++
  VALEU
SENAO
  Y++
VALEU
```

```
MARCA X RAPIDAO
  Z++
VALEU


ENQUANTO Z FACA
  SEPA X TA_LGD
    X++
  VALEU
 Z--
VALEU

FALATU(X)
FALATU(Y)
FALATU(Z)
FALATU(A)
B++
FALATU(B)

VALEU
```

Aqui foi encurtado o trecho em assembly, retirando a parte dos prints, preparação e encerramento do bloco de código, que são iguais ao exemplo anterior. Foi preservado apenas a lógica do controle de fluxo:

```
   cmpl $0,-8(%rbp)
   je L3
   cmpl $0,-16(%rbp)
   je L1
   addl $1, -8(%rbp)
L1:
   jmp L4
L3:
   addl $1, -16(%rbp)
L4:
   movl $0,%r13d
L5:
   addl $1,%r13d
   addl $1, -24(%rbp)
   cmpl %r13d,-8(%rbp)
   jne L5

L7:
   cmpl $0, -24(%rbp)
   je L8
   cmpl $0,-8(%rbp)
```

```
   je L6
   addl $1, -8(%rbp)
L6:
   subl $1, -24(%rbp)
   jmp L7
L8:
```

Mais exemplos se encontram na pasta `tests/`, e seus respectivos códigos em assembly na sub-pasta `tests/asm`.

Bibliografia

1. [Recursive Rules] (https://www.gnu.org/software/bison/manual/html_node/Recursion.html)
2. Compiladores Princípios, Técnicas e Ferramentas - Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman