

BIRL++

É hora do show!



Introdução

A nossa linguagem é inspirada na *Bambam's "It's show time" Recursive Language* [1], que por sua vez é baseada na linguagem C. Dessa forma, denominamos nossa linguagem de: **BIRL++**.

A origem do **BIRL** se deu em meados de maio de 2016, através de um vídeo no YouTube, protagonizado por Kleber “Bambam” de Paula, vencedor do primeiro *Big Brother Brasil*. O vídeo, devido ao conteúdo cômico da personalidade do Kleber, rapidamente viralizou e se tornou um *meme* [2].

Descrição da linguagem

As terminações de linha são feitas através do caractere ponto-e-vírgula: ;

As terminações de bloco são feitas através da palavra reservada: BIRL

Comentários

Comentários de uma linha são feitos através dos símbolos: // comentário

Um comentário multi-linha pode ser declarado com: /* comentario multi-linha */

Declaração de variáveis

A linguagem possui os tipos tradicionais do C, com outro nome:

| BIRL++ | C |
|----------------------|--------|
| FRANGO | char |
| MONSTRINHO | short |
| MONSTRO | int |
| MONSTRAO | long |
| TRAPEZIO | float |
| TRAPEZIO DESCENDENTE | double |

Os 4 primeiros tipos podem ser transformados em *unsigned* através do modificador "BICEPS":

Exemplo: **BICEPS MONSTRO** x = 13;

Condicional

A estrutura usual de if/else if/else no BIRL++ é a seguinte, tendo apenas o primeiro bloco como obrigatório:

```
ELE QUE A GENTE QUER? (X > 2)
    // X > 2
QUE NAO VAI DAR O QUE? (X < 2)
    // X < 2
NAO VAI DAR NAO
    // X = 2
BIRL
```

As constantes booleanas true e false são, respectivamente: **FIBRA** e **AGUA**

A negação utiliza a palavra: **NEGATIVA**

Iteração

A iteração em BIRL++ se restringe à construção *while*, por simplicidade. A sua sintaxe é da forma:

```
MONSTRO X = 5;
NEGATIVA BAMBAM (X > 2)
    X--;
BIRL
```

As palavras reservadas para uso dentro de uma iteração, *continue* e *break*, são:

| BIRL++ | C |
|--------------|----------|
| VAMO MONSTRO | continue |
| SAI FDP | break |

Declaração de função

A declaração de função segue a seguinte sintaxe, onde *return* é substituído por **BORA CUMPADE**.

```
OH O HOME AI PO (MONSTRO soma(MONSTRO x, MONSTRO y))  
    BORA CUMPADE x + y;  
BIRL
```

Execução de função

A execução de função é realizada através da expressão **AJUDA O MALUCO TA DOENTE**:

```
MONSTRO A = 5;  
MONSTRO B = 8;  
MONSTRO C = AJUDA O MALUCO TA DOENTE soma(A, B);
```

Função principal (main)

A *main* é uma função especial, declarada com a seguinte sintaxe:

```
HORA DO SHOW  
    // código aqui  
BIRL
```

Definição de sintaxe

A sintaxe foi definida utilizando a seguinte descrição em EBNF (Wirth):

```
PROGRAMA = {FUNCAO} MAIN {FUNCAO} .

DECLARACAO_TIPO = TIPO IDENTIFICADOR ";" .

TIPO = "FRANGO" | "MONSTRINHO" | "MONSTRO" | "MONSTRAO" | "TRAPEZIO" |
"TRAPEZIO" "DESCENDENTE" | "BICEPS" "FRANGO" | "BICEPS" "MONSTRINHO" | "BICEPS"
"MONSTRO" | "BICEPS" "MONSTRAO" .

IDENTIFICADOR = LETRA {LETRA|DIGITO} .
LETRA = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" |
"m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" |
"z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" |
"M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" |
"Z" .
DIGITO = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

FUNCAO = "OH" "O" "HOME" "AI" "PO" "(" TIPO IDENTIFICADOR "(" LISTA_PARAMETROS
")" ")" BLOCO_CODIGO "BIRL" .
LISTA_PARAMETROS = TIPO IDENTIFICADOR { "," TIPO IDENTIFICADOR } .

MAIN = "HORA" "DO" "SHOW" BLOCO_CODIGO "BIRL" .

BLOCO_CODIGO = COMANDO {COMANDO} .

COMANDO = COMANDO_ATRIBUICAO ";" | CONDICIONAL | ITERATIVO | PULO ";" .
COMANDO_ATRIBUICAO = [ TIPO ] ATRIBUICAO { "," ATRIBUICAO } .
ATRIBUICAO = IDENTIFICADOR "=" EXPRESSAO | IDENTIFICADOR .

EXPRESSAO = EXPRESSAO_L { OPERADOR EXPRESSAO_L } .
EXPRESSAO_L = IDENTIFICADOR | NUMERO | CHAMADA_FUNCAO | "FIBRA" | "AGUA" |
"NEGATIVA" EXPRESSAO | "(" EXPRESSAO ")" .

OPERADOR = OPERADOR_BOOL | OPERADOR_ARITM | OPERADOR_COMP .
OPERADOR_BOOL = "&&" | "||" .
OPERADOR_ARITM = "+" | "-" | "/" | "*" | "%" .
OPERADOR_COMP = ">" [ "=" ] | "<" [ "=" ] | "==" | "!=" .

NUMERO = DIGITO{DIGITO} | DIGITO{DIGITO} "." {DIGITO} | "." DIGITO {DIGITO} .

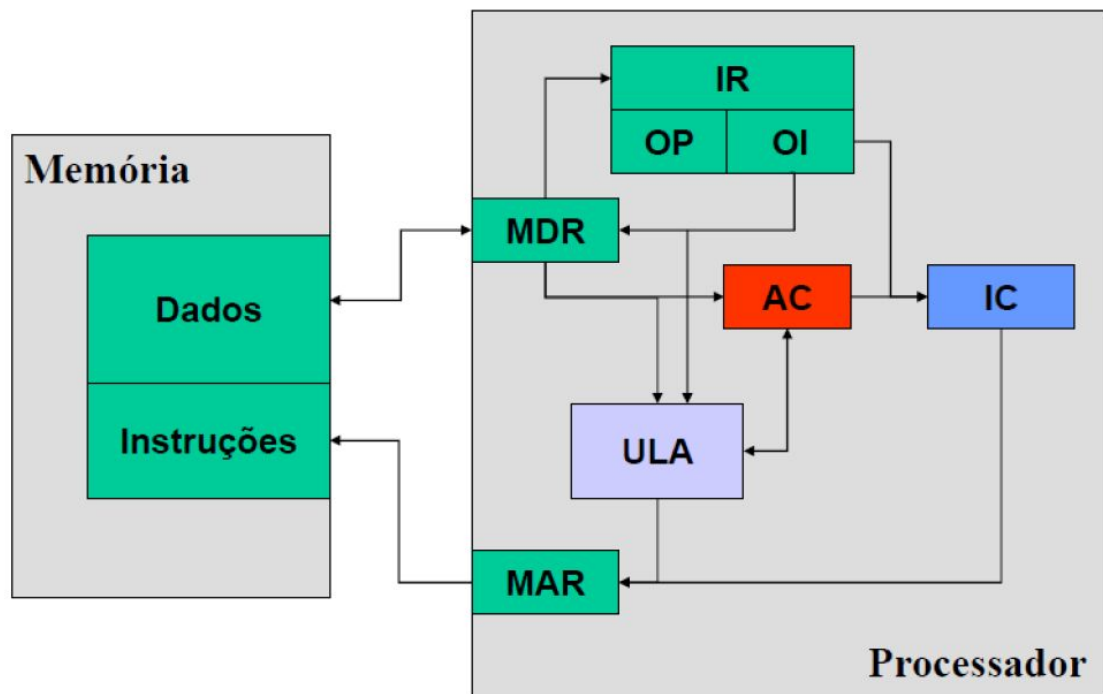
CHAMADA_FUNCAO = "AJUDA" "O" "MALUCO" "TA" "DOENTE" IDENTIFICADOR "(" [EXPRESSAO
{ "," EXPRESSAO } ] ")" .

CONDICIONAL = "ELE" "QUE" "A" "GENTE" "QUER" "?" "(" EXPRESSAO ")" BLOCO_CODIGO
[ELSE_IF] "BIRL" .
ELSE_IF = "QUE" "NAO" "VAI" "DAR" "O" "QUE" "?" "(" EXPRESSAO ")" BLOCO_CODIGO
[ELSE_IF] | "NAO" "VAI" "DAR" "NAO" BLOCO_CODIGO .

ITERATIVO = "NEGATIVA" "BAMBAM" "(" EXPRESSAO ")" BLOCO_CODIGO "BIRL" .
PULO = "BORA" "CUMPADE" EXPRESSAO | "SAI" "FDP" | "VAMO" "MONSTRO" .
```

Ambiente de execução

O ambiente de execução almejado para a compilação é a MVN (Máquina de von Neumann) [3].



Características gerais

A MVN possui como características gerais:

- Memória principal: armazena programas e dados juntos
- Acumulador (AC): funciona como área de trabalho, para a execução de operações aritméticas e lógicas.
- Registradores auxiliares: empregados em diversas operações intermediárias no processamento dos programas. Os registradores auxiliares são:
 - Registrador de dados da memória (MDR): serve como ponte para os dados que trafegam entre a memória e os outros elementos da máquina.
 - Registrador de instrução (IR): contém a instrução em execução:
 - Código de operação (OP): parte do registrador de instrução que identifica a instrução que está sendo executada.

-
- Operando da instrução (OI): complementa a instrução indicando o dado ou o endereço sobre o qual ela deve agir.
 - Registrador de endereço de instrução (IC): indica em cada instante qual será a próxima instrução a ser executada pela máquina.
 - Registrador de endereço da memória (MAR): indica qual é a origem ou o destino, na memória principal, dos dados contidos no registrador de dados da memória.
 - Registrador de endereço de instrução (IC): indica em cada instante qual será a próxima instrução a ser executada pela máquina.
 - Registrador de instrução (IR): contém a instrução em execução:
 - Código de operação (OP): parte do registrador de instrução que identifica a instrução que está sendo executada.
 - Operando da instrução (OI): complementa a instrução indicando o dado ou o endereço sobre o qual ela deve agir.

Pseudoinstruções da linguagem de saída

Abaixo estão descritas as pseudoinstruções da linguagem montadora da MVN

| Pseudoinstrução | Função |
|-----------------|--|
| K | define constante |
| < | definição de endereço simbólico que referencia um entry-point externo |
| @ | define o endereço absoluto da primeira instrução do programa |
| & | define uma origem relocável |
| \$ | Reserva de área de dados, o tamanho da área a ser reservada (em bytes) |
| > | definição de endereço simbólico para exportar (entry point) |
| # | define o fim do programa |

Instruções da linguagem de saída

Abaixo estão detalhadas os mnemônicos, seus códigos hexadecimais correspondentes e o formato das instruções para a MVN.

| Mnemônico | Hexadecimal | Instrução | Operando |
|-----------|-------------|------------------------|--------------------------------|
| - | 5 | Subtract | endereço subtraendo |
| / | 7 | Divide | endereço divisor |
| JZ | 1 | Jump if Zero | endereço de desvio |
| LD | 8 | Load | endereço origem |
| LV | 3 | Load Value | constante |
| RS | B | Return from Subroutine | endereço do resultado |
| OS | F | Operating System | constante |
| GD | D | Get Data | dispositivo de entrada e saída |
| SC | A | Subroutine Call | endereço do subprograma |
| HM | C | Halt Machine | endereço do desvio |
| PD | E | Put Data | dispositivo de entrada e saída |
| JN | 2 | Jump if Negative | endereço de desvio |
| MM | 9 | Move to Memory | endereço destino |
| + | 4 | Add | endereço parcela |
| * | 6 | Multiply | endereço multiplicador |
| JP | 0 | Jump | endereço de desvio |

Implementação

O compilador foi implementado em C, com 3 módulos: análise léxica, análise sintática e análise semântica com geração de código para MVN. Todo o código se encontra em anexo.

Há um arquivo README.txt na raiz com instruções de execução.

Análise léxica

O analisador léxico é responsável por receber uma entrada textual e reconhecer símbolos da linguagem, de forma a preparar para a fase da análise sintática.

O resultado da análise léxica é uma lista de símbolos (tokens, átomos) e suas propriedades (posição, classe, etc).

| Token | Expressão Regular |
|---------------------|---|
| Identificador | [a-zA-Z][a-zA-Z0-9]* |
| Número | [0-9]+ [0-9]+\.[0-9]* \.[0-9]+ |
| String | ".*" '.*' |
| Palavras reservadas | BIRL FRANGO MONSTRINHO MONSTRO MONSTRAO BICEPS TRAPEZIO DESCENDENTE OH O HOME AI PO HORA DO SHOW FIBRA AGUA NEGATIVA AJUDA MALUCO TA DOENTE ELE QUE A GENTE QUER NAO VAI DAR BAMBAM MAIS QUERO BORA CUMPADE SAI FDP VAMO CE VER ESSA PORA |
| Símbolos Especiais | = == !=? >=? <=? \+ - * \/ & && , ; \\ \\\ \\\ \\\{\\} \\(\\) % |
| Comentário | \\/\\. * \\/*(. [\\n\\r])**\\/ |
| Espaçadores | [\\t\\n\\r]* |

Análise sintática

Para a análise sintática, desenvolvemos uma solução baseada em um autômato de pilha estruturado. Em C, o autômato possui a seguinte interface:

```
typedef struct {
    const char *token;
    int estadoResultado;
    UT_hash_handle hh;
} Transicao;

typedef struct {
    const char *token;
    const char *submaquina;
    int estadoResultado;
    UT_hash_handle hh;
} TransicaoChamada;

typedef struct {
    int estado;
    Transicao *transicoes;
    TransicaoChamada *chamadas;
    UT_hash_handle hh;
} Estado;

typedef struct {
    const char *title;
    int estado;
    Estado *estadosTransicoes;
    UT_array *estadosFinais;
    UT_hash_handle hh;
} Automato;

typedef struct {
    Automato *automatos;
    Automato *automatoAtual;
    UT_array *pilha;
} APE;
```

Para popular o autômato de pilha estruturado com os estados e transições correspondentes ao Wirth, criamos um gerador em Python 3. Esse gerador envia um arquivo contendo o Wirth ao marcador de estados do Mc Barau [4]. O resultado é um arquivo contendo código C de inicialização desses *structs*.

Análise semântica

Para a análise semântica, são necessárias algumas rotinas semânticas, que serão acionadas nas transições do autômato de pilha. Foram definidas as seguintes rotinas:


| Rotina | Descrição |
|------------------------------------|---|
| initProgram() | Inicializa os endereços de memória relativos à área de dados e a área do código do programa adicionando as labels globais e o JUMP para o ponto inicial do código |
| pushToOperandStack() | Simplesmente empilha o operando na respectiva pilha |
| startExpression() | O compilador avalia uma expressão, no caso de uma expressão já estar sendo avaliada, ele retorna falha. Primeiramente ele aloca espaço de memória para calcular a expressão, depois ele empilha os operandos e o operador nas respectivas pilhas. |
| consumeLogicOperator() | Durante o cálculo da expressão essa rotina transforma as expressões lógicas em código objeto, manipulando a pilha de operandos e operadores |
| consumeArithmeticOperator() | Trabalho similar ao consumeLogicOperator() porém sendo usado para operações aritméticas, gerando código objeto para tal |
| finishAttribution() | Após ter avaliado a expressão, essa rotina salva o valor no endereço de memória da variável que recebe o resultado |
| printData() | Gera código para a impressão de dados a partir do comando de PD da MVN |
| pushToOperatorStack() | Empilha o novo operador lido levando em conta as regras de precedência |
| getData() | Gera o código para a leitura de dados no dispositivo de entrada |
| finishExpression() | Após empilhar os operandos e operadores em startExpression(), nessa etapa o compilador os desempilha e calcula o código relativo às operações da expressão e salva o resultado no ACC. |
| finishProgram() | Finaliza o programa (HALT MACHINE) |

Execução

O código de teste executado calcula o fatorial de um número:

```
HORA DO SHOW
  MONSTRO entrada = 6;
  MONSTRO fatorial = 1;
  NEGATIVA BAMBAM (entrada > 0)
    fatorial = fatorial * entrada;
    entrada = entrada - 1;
  BIRL
  BORA CUMPADE 0;
BIRL
```

Após a execução na MVN, a memória foi inspecionada para revelar o valor 02D0, 720 (6!) em hexadecimal.



```
> m 000c 000d
[ 0]:                                02 D0
Final do dump.
```

Conclusão

Acreditamos que a experiência de criar um compilador foi bastante desafiadora e interessante. Por meio do projeto foi possível entender realmente como as linguagens que utilizamos na vida acadêmica e profissional são pensadas e desenvolvidas. Além disso, dada toda a bagagem do conteúdo de computação que conquistamos durante o curso, passando por sistemas digitais e arquiteturas de sistemas computacionais, software e agora pela concepção deste compilador, vemos essa disciplina como um importante capítulo da construção de nosso conhecimento.

Referências

- [1] PADILHA, L. **Bambam's "It's show time" Recursive Language**. Disponível em: <<https://birl-language.github.io/>>. Acesso em: 12 dez. 2017.
- [2] KNOW YOUR MEME. **BIIIRL/ O MONSTRO TA SAINDO DA JAULA**. Disponível em: <<http://knowyourmeme.com/memes/biiirl-o-monstro-ta-saindo-da-jaula>>. Acesso em: 12 dez. 2017.
- [3] COSTA, A. H. R.; SICHMAN, J. S.; NETO, J. J.; SILVA, P. S. M.; ROCHA, R. L. A. **Máquina de von Neumann**. Disponível em: <https://edisciplinas.usp.br/pluginfile.php/4094732/mod_resource/content/1/MVN_-_2010.pdf>. Acesso em: 12 dez. 2017.
- [4] BARAU, M.; **State Marker for Wirth Notation**. Disponível em: <<http://mc-barau.herokuapp.com/>>. Acesso em: 12 dez. 2017.