

Processamento de Linguagens e Compiladores  
**Gerador de Processadores de Linguagens Yacc**  
Relatório de Desenvolvimento  
Grupo 14

André Sá (76361)

João Bastos (47419)

Pedro Sá (78164)

6 de Janeiro de 2019

## **Resumo**

Este relatório debruça-se sobre o desenvolvimento de uma linguagem imperativa que com o auxílio do Yacc<sup>1</sup> gera código Assembly, sendo esse código executável em uma Máquina Virtual(VM)<sup>2</sup> fornecida pelo professor desta Unidade Curricular.

---

<sup>1</sup>Yacc - Gerador de Processadores de Linguagens

<sup>2</sup>Maquina Virtual que mostra o estado das várias stack's ao longo da execução do código gerado pelo Yacc em Assembly

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
1.1	Enquadramento . . . . .	2
1.2	Estrutura do Relatório . . . . .	2
<b>2</b>	<b>Análise e Especificação</b>	<b>3</b>
2.1	Descrição informal do problema . . . . .	3
2.2	Especificação dos Requisitos . . . . .	3
<b>3</b>	<b>Linguagem Imperativa</b>	<b>4</b>
3.1	Variáveis . . . . .	4
3.2	Tipos . . . . .	4
3.3	Operadores . . . . .	4
3.4	Declaração de Variáveis . . . . .	5
3.5	Atribuições . . . . .	5
3.6	Condição If . . . . .	5
3.7	Ciclos . . . . .	6
3.8	Leitura . . . . .	6
3.9	Escrita . . . . .	6
3.10	Impressão . . . . .	6
<b>4</b>	<b>Gramática</b>	<b>7</b>
<b>5</b>	<b>Concepção/desenho da Resolução</b>	<b>9</b>
<b>6</b>	<b>Testes</b>	<b>10</b>
6.1	Testes realizados e Resultados . . . . .	10
6.1.1	Declarações e Atribuições . . . . .	10
6.1.2	Operadores Aritméticos . . . . .	11
6.1.3	Leitura . . . . .	12
6.1.4	Condição IF . . . . .	12
6.1.5	Ciclo Repetir..Até . . . . .	12
<b>7</b>	<b>Conclusão</b>	<b>13</b>
<b>A</b>	<b>Código do Programa</b>	<b>14</b>

# Capítulo 1

## Introdução

Este relatório debruça-se sobre o desenvolvimento de uma linguagem imperativa que com o auxílio do Yacc<sup>1</sup> gera código Assembly, sendo esse código executável em uma Máquina Virtual(VM)<sup>2</sup> fornecida pela professor desta Unidade Curricular.

### 1.1 Enquadramento

No âmbito da Unidade Curricular Processamento de Linguagens e Compiladores, o projecto proposto consiste na criação de uma Linguagem de Programação Imperativa que permita declarar variáveis atómicas dos tipos Inteiro, Real e Booleano e fazer as operações de atribuição de expressões a variáveis declaradas, leitura (de inteiros ou reais), escrita (de inteiros, reais, booleanos ou strings), condições e ciclos repetir....até e gere código Assembly para a VM fornecida pela professor.

### 1.2 Estrutura do Relatório

Neste relatório começamos com a Introdução (Capítulo 1) onde, de uma forma breve tentamos descrever o enquadramento do documento e a estrutura do mesmo.

No capítulo 2 analisamos o texto a processar e o problema que nos foi proposto de forma mais detalhada bem como uma introdução da nossa abordagem ao mesmo.

No capítulo 3 indicamos a estrutura de dados utilizada e explicamos o código por nós criado.

No capítulo 4 apresentamos alguns testes realizados e o respectivo output.

O capítulo 5 é composto pela conclusão, onde fazemos uma análise ao projecto executado.

Por último o documento tem o apêndice onde foi introduzido o código fonte do ficheiro Yacc.

---

<sup>1</sup>Yacc - Gerador de Processadores de Linguagens

<sup>2</sup>Maquina Virtual que mostra o estado das várias stack's ao longo da execução do código gerado pelo Yacc em Assembly

## Capítulo 2

# Análise e Especificação

### 2.1 Descrição informal do problema

Este projecto tem como objectivo o aprofundamento de conhecimento Yacc (um gerador de processadores de linguagens). Para tal, foi-nos proposto construir uma linguagem imperativa e gerar um compilador que transforma a sintaxe da linguagem para "Assembly" reconhecido pela máquina virtual fornecida pelo professor da Unidade Curricular.

### 2.2 Especificação dos Requisitos

A linguagem a criar deverá permitir declarar Variáveis atómicas dos tipos Inteiro, Real e Booleano e fazer Operações de Atribuição de Expressões a Variáveis declaradas, Leitura (de Inteiros ou Reais), Escrita (de Inteiros, Reais, Booleanos ou Strings), condições e ciclos repetir .. até. Após definir a linguagem deveremos a partir de um código fonte com essa linguagem transformar em código Assembly da Máquina Virtual fornecida para o efeito.

## Capítulo 3

# Linguagem Imperativa

Neste capítulo explicaremos a linguagem imperativa definida e a sua sintaxe.

### 3.1 Variáveis

As variáveis terão de começar obrigatoriamente por uma letra ou underscore, podendo ser seguida de números, letras ou underscores. A título de exemplo, as variáveis `_aluno`, `Aluno22`, `aluno` e `aluNo` são válidas.

### 3.2 Tipos

Apresentamos os tipos aceites na linguagem criada.

Tipo	Descrição	Exemplos
<b>Int</b>	Inteiro	1 -22 35
<b>Float</b>	Reais	1.333 3.0E10
<b>Bool</b>	Booleano	True False

### 3.3 Operadores

Os operadores dividem-se em 3 tipos: aritméticos, relacionais e lógicos. De seguida apresentamos os operadores utilizados e a correspondente descrição.

ARITMÉTICOS	Operador	Descrição
	+	Soma
	-	Subtração
	*	Multiplicação
	/	Divisão
	%	Resto da divisão inteira

RELACIONAIS	Operador	Descrição
	<	Menor
	>	Maior
	=	Igual
	<=	Menor ou igual
	>=	Maior ou igual
	!=	Diferente

LÓGICOS	Operador	Descrição
	&	E lógico
		OU lógico
	~	Negação

### 3.4 Declaração de Variáveis

A declaração de variáveis tem a seguinte sintaxe:

```
(: tipo var)
```

Ao declarar, as variáveis são iniciadas com o valor pré-definido do tipo em questão:

Tipo	Default
<b>Int</b>	0
<b>Float</b>	0.0
<b>Bool</b>	False

No entanto também é possível definir no momento um *default* diferente iniciando-a com o valor, do tipo da variável, definido. Nestes casos utilizamos a seguinte sintaxe:

```
(: tipo var default)
```

Só é permitido declarar variáveis (*var*) do *tipo* Int, Float e Bool.

### 3.5 Atribuições

Na atribuição a variáveis (*var*) temos a seguinte sintaxe:

```
(= var exp )
```

Em que *exp* pode ser um valor Int, Float, Bool, uma variável ou uma operação aritmética.

Podemos também incrementar numa unidade o valor da variável (*var*), que apenas é válido para variáveis do tipo Int e Float:

```
(++ var)
```

### 3.6 Condição If

As condições *if then else* terão a seguinte sintaxe:

```
(if exp (code_block) [ (else_code_block) ] )
```

Em que *exp* terá expressões com valor lógico que corresponderá à condição necessária para executar a porção de código *code\_block*, caso contrário, se definido, será executada a porção de código *else\_code\_block*.

### 3.7 Ciclos

Os ciclos executam uma determinada porção de código *code\_block* até a condição *exp* ser verdadeira. Nesse sentido, *exp* será uma expressão com valor lógico. Cada ciclo terá então a seguinte sintaxe:

**(until** exp (code\_block) **)**

### 3.8 Leitura

A leitura será feita do stdin e irá guardar na variável *var* que será do tipo Int ou Float. A sintaxe será:

**(read** var)

### 3.9 Escrita

A escrita será feita após receber um Int, Float, Bool ou String seguindo a seguinte sintaxe:

**(write** exp)

### 3.10 Impressão

A impressão utiliza a seguinte sintaxe imprimindo o argumento que lhe é passado:

**(print** texto)



## Capítulo 4

# Gramática

Seguindo a definição da linguagem descrita no Capítulo 3 criamos a seguinte gramática que será a base do nosso projecto. A partir dela, posteriormente, aplicaremos as respectivas ações semânticas como veremos a seguir.

```
programa          : code_block
                  ;
code_block        : statements
                  | statements code_block
                  ;
statements        : '(' statement ')'
                  ;
statement         : ':' TYPE VAR DEFAULT
                  | INC VAR
                  | '=' VAR expression
                  | WRITE writable
                  | PRINT writable
                  | READ VAR
                  | IF expression2 '(' code_block ')' else_clause
                  | UNTIL expression2 '(' code_block ')'
                  ;
writable          : expression
                  | STR
                  ;
DEFAULT          :
                  | expression
                  ;
else_clause       :
                  | '(' code_block ')'
                  ;
expression        : VALUE
                  | expression2
                  | '(' expression_list ')'
                  ;
VALUE            : INT_VALUE
                  | FLOAT_VALUE
                  ;
arith_op          : '+'
                  | '*'
                  | '-'
                  | '/'
                  | '%'
                  ;
```

```

expression_list : arith_op expression expression
                ;
expression2     : VAR
                | BOOLVALUE
                | '(' expression2_list ')'
                ;
num_op          : '<'
                | '='
                | '>'
                | GEQ
                | LEQ
                | NEQ
                ;
log_op          : '&'
                | '|'
                ;
expression2_list : '~' expression2
                | num_op expression expression
                | log_op expression2 expression2
                ;

```

## Capítulo 5

# Concepção/desenho da Resolução

Neste capítulo iremos descrever a concepção/desenho do analisador sintático iniciando cada ação semântica com um pequeno resumo seguido do código referido.

# Capítulo 6

## Testes

### 6.1 Testes realizados e Resultados

Para efeitos de teste preparamos vários ficheiros que cada um contém desde declarações, a atribuições, condições, ciclos, etc...

Estes testes serão organizados em tabelas abaixo apresentadas com a primeira coluna a representar o código-fonte de acordo com a nossa linguagem imperativa criada e a segunda coluna o código em "assembly" em conformidade com a máquina virtual fornecida.

#### 6.1.1 Declarações e Atribuições

Input	Output
	PUSHI 0
	PUSHI 0
	PUSHI 1
	PUSHI 1
	PUSHF 0.0
	PUSHF 0.0
	START
(: Int x)	PUSHI 2
(: Int y 2)	STOREG 1
(= x (+ x y))	PUSHGP
(: Bool a)	LOAD 0
(: Bool b True)	PUSHGP
(: Float z 10.0)	LOAD 1
(: Float t)	ADD
	STOREG 0
	PUSHI 1
	STOREG 3
	PUSHF 10.0
	STOREG 4
	STOP

### 6.1.2 Operadores Aritméticos

Input	Output
	START
	PUSHI 1
	PUSHI 2
	ADD
	WRITEI
	PUSHS "\n"
	WRITES
	PUSHI 1
	PUSHI 2
	MUL
	WRITEI
	PUSHS "\n"
	WRITES
	PUSHI 1
	PUSHI 2
	DIV
	WRITEI
	PUSHS "\n"
	WRITES
	PUSHI 1
	PUSHI 2
	SUB
	WRITEI
(print (+ 1 2))	PUSHS "\n"
(print (* 1 2))	WRITES
(print (/ 1 2))	PUSHI 1
(print (- 1 2))	PUSHI 2
(print (% 1 2))	MOD
	WRITEI
(print (+ 1.0 2.0))	PUSHS "\n"
(print (* 1.0 2.0))	WRITES
(print (/ 1.0 2.0))	PUSHF 1.0
(print (- 1.0 2.0))	PUSHF 2.0
	FADD
	WRITEF
	PUSHS "\n"
	WRITES
	PUSHF 1.0
	PUSHF 2.0
	FMUL
	WRITEF
	PUSHS "\n"
	WRITES
	PUSHF 1.0
	PUSHF 2.0
	FDIV
	WRITEF
	PUSHS "\n"
	WRITES
	PUSHF 1.0
	PUSHF 2.0
	FSUB
	WRITEF
	PUSHS "\n"
	WRITES
	STOP

### 6.1.3 Leitura

Input	Output
(: Float x) (read x) (print x)	PUSHF 0.0 START READ ATOF STOREG 0 PUSHGP LOAD 0 WRITEF PUSHS "\n" WRITES STOP

### 6.1.4 Condição IF

Input	Output
(if (~ (= 1 2)) (write "then")) (write "else"))  (if (!= 1 1) (write "then"))	START PUSHI 1 PUSHI 2 EQUAL NOT JZ ELSE0 PUSHS "then" WRITES JUMP ENDIF0 ELSE0: PUSHS "else" WRITES ENDIF0: PUSHI 1 PUSHI 1 SUB JZ ELSE1 PUSHS "then" WRITES JUMP ENDIF1 ELSE1: ENDIF1: STOP

### 6.1.5 Ciclo Repetir..Até

Input	Output
(until (= 1 2) (write "wut"))	START UNTIL0: PUSHS "wut" WRITES PUSHI 1 PUSHI 2 EQUAL JZ UNTIL0 STOP

## Capítulo 7

# Conclusão

Este projecto mostrou-nos que tendo já desenvolvido a gramática de uma linguagem imperativa, podemos criar um compilador através do Yacc que usando a estrutura 'Produção-Ação', isto é, para cada instrução do código fonte que satisfaça alguma produção da linguagem imperativa tem como ação a criação do código "assembly", o que consequentemente, deu-nos uma visão mais detalhada sobre como um compilador funciona. Este projeto está finalizado para o que nos foi proposto.

# Apêndice A

## Código do Programa

Lista-se a seguir o código Yacc do programa que foi desenvolvido.

```
%{
#include <stdbool.h>
#include <stdio.h>

#include "env.h"
#include "gen.h"
#include "str.h"
#include "rope.h"

#include "lex.yy.h"

int yyerror (const char *s);

/**
 * Verifica se uma condicao e verdadeira e, caso nao seja,
 * imprime uma mensagem de erro e aborta o programa
 */
#define assert(cond, ...) \
    if (!(cond)) do { \
        fprintf(stderr, "ERROR: " __VA_ARGS__); \
        return 1; \
    } while (0)

/** Se activado, mostra as producoes por onde passa */
#ifdef TRACE
#define trace(...) ((void) fprintf(stderr, "TRACE: %s:%d\n", __FILE__, __LINE__))
#else
#define trace(...) ((void) 0)
#endif /* TRACE */

#define type_valid(t) ((t) > TYPE_ERROR && (t) < TYPE_DEFAULT)
#define type_compat(t1, t2) ((t1) == (t2))

/**
 * Concatena dois blocos de codigo, @a self e @a other, e imprime
 * uma mensagem em caso de erro
 */
```



```

#define cbapp(self, other) do { \
    assert(rope_append(&(self), &(other)), \
        "appending code blocks: %s:%d\n", \
        __FILE__, __LINE__); \
    (other) = rope_free(other); \
} while (0)

/**
 * Gera wrappers para funcoes geradoras de codigo que imprimem
 * uma mensagem em caso de erro
 */
#define gen_(f, ...) \
    assert((gen_##f)(__VA_ARGS__), \
        #f "(): %s:%d\n", \
        __FILE__, __LINE__)

#define gen_jump( c, l, n)    gen_(jump, (c), (l), (n))
#define gen_jz( c, l, n)    gen_(jz, (c), (l), (n))
#define gen_nlbl( c, l, n)    gen_(nlbl, (c), (l), (n))
#define gen_load( c, i)    gen_(load, (c), (i))
#define gen_op( c, o, t)    gen_(op, (c), (o), (t))
#define gen_push( c, t, a, b) gen_(push, (c), (t), (a), (b))
#define gen_pushgp(c)    gen_(pushgp, (c))
#define gen_storeg(c, i)    gen_(storeg, (c), (i))

static struct rope _var_decs = {0};
static struct rope * const var_decs = &_var_decs;

%}

%union {
    struct expr {
        enum type type;
        struct rope code;
    } valExpr;

    bool valBool;
    char * valString;
    enum type valType;
    float valFloat;
    int valInt;
    struct rope valCode;
}

%token INC
%token GEQ
%token IF
%token LEQ
%token NEQ
%token PRINT
%token READ
%token UNTIL
%token WRITE

```

```

%token <valBool>  BOOL_VALUE
%token <valFloat> FLOAT_VALUE
%token <valInt>   INT_VALUE
%token <valString>STR
%token <valString>VAR
%token <valType>  TYPE

%type <valCode>code_block
%type <valCode>else_clause
%type <valCode>statement
%type <valCode>statements

%type <valExpr>DEFAULT
%type <valExpr>expression
%type <valExpr>expression2
%type <valExpr>expression2_list
%type <valExpr>expression_list
%type <valExpr>writable

%type <valInt>arith_op
%type <valInt>log_op
%type <valInt>num_op

%type <valType>VALUE

%%

programa : code_block { trace();
                      rope_fprint(var_decs, yyout);
                      fputs("START\n", yyout);
                      rope_fprint(&$1, yyout);
                      fputs("STOP\n", yyout);
                      $1 = rope_free($1);
                      }
          ;

code_block : statements          { trace(); $$ = $1; }
          | statements code_block { trace();
                      $$ = $1;
                      cbapp($$, $2);
                      }
          ;

statements : '(' statement ')' { trace(); $$ = $2; }
          ;

statement : ':' TYPE VAR DEFAULT { trace();
                      assert($4.type != TYPE_ERROR
                      && (!type_valid($4.type) || type_compat($2, $4.type)),
                      "%s:%s but default value is of type %s\n",
                      $3, type2str($2), type2str($4.type));

                      struct var var = { .id = $3, .type = $2, };
                      assert(env_new_var(env, var), "creating variable '%s'\n", $3);

```

```

    $$ = $4.code;
    if (!rope_is_empty(&$$))
        gen_storeg(&$$, env_var_gp_idx(env, $3));

    gen_push(var_decs, $2,
        (($2 == TYPE_FLOAT) ? "0.0" : "0"),
        yylval.valBool);
}
| INC VAR { trace();
    $$ = (struct rope) {0};
    struct var * v = env_var(env, $2);
    enum type t = env_typeof(env, $2);
    unsigned gidx = env_var_gp_idx(env, $2);

    assert(v != NULL, "Variable not found: '%s'\n", $2);

    assert(t == TYPE_INT || t == TYPE_FLOAT,
        "type: Expected Int or Float but got %s\n",
        type2str(t));

    const char * arg = (t == TYPE_INT) ? "1" : "1.0";

    gen_pushgp(&$$);
    gen_load(&$$, gidx);
    gen_push(&$$, t, arg, false);
    gen_op(&$$, '+', t);
    gen_storeg(&$$, gidx);
}
| '=' VAR expression { trace();
    struct var * v = env_var(env, $2);
    assert(v != NULL, "Variable not found: '%s'\n", $2);
    $$ = $3.code;
    gen_storeg(&$$, env_var_gp_idx(env, $2));
}
| WRITE writable { trace();
    $$ = $2.code;
    gen_op(&$$, WRITE, $2.type);
}
| PRINT writable { trace();
    $$ = $2.code;
    gen_op(&$$, WRITE, $2.type);
    gen_push(&$$, TYPE_STRING, "\\n\\n", false);
    gen_op(&$$, WRITE, TYPE_STRING);
}
| READ VAR { trace();
    $$ = (struct rope) {0};
    struct var * v = env_var(env, $2);
    assert(v != NULL, "Variable not found: '%s'\n", $2);

    assert(v->type == TYPE_INT || v->type == TYPE_FLOAT,
        "'read': Expected Int or Float but got %s\n",
        type2str(v->type));
}

```

```

    gen_read(&$$);
    gen_aton(&$$, v->type);
    gen_storeg(&$$, env_var_gp_idx(env, $2));
}
| IF expression2 '(' code_block ')' else_clause { trace();
    unsigned num = gen_ifno();

    $$ = $2.code;
    gen_jz(&$$, "ELSE", num); /* jump to else label? */
    cbapp($$, $4); /* then block */
    gen_jump(&$$, "ENDIF", num); /* jump to endif label */
    gen_nlbl(&$$, "ELSE", num); /* else label */
    cbapp($$, $6); /* else block (possibly empty) */
    gen_nlbl(&$$, "ENDIF", num); /* endif label */
}
| UNTIL expression2 '(' code_block ')' { trace();
    $$ = (struct rope) {0};
    unsigned num = gen_untilno();

    gen_nlbl(&$$, "UNTIL", num); /* until label */
    cbapp($$, $4); /* loop body block */
    cbapp($$, $2.code); /* condition */
    gen_jz(&$$, "UNTIL", num); /* jump to until label? */
}
;

writable : expression { trace(); $$ = $1; }
| STR { trace();
    $$ = (struct expr) {0};
    $$>.type = TYPE_STRING;
    gen_push(&$$>.code, TYPE_STRING, yytext, false);
}
;

DEFAULT : { trace();
    $$ = (struct expr) {0};
    $$>.type = TYPE_DEFAULT;
}
| expression { trace(); $$ = $1; }
;

else_clause : { trace(); $$ = (struct rope) {0}; }
| '(' code_block ')' { trace(); $$ = $2; }
;

expression : VALUE { trace();
    $$ = (struct expr) {0};
    $$>.type = $1;
    gen_push(&$$>.code, $1, yytext, false);
}
| expression2 { trace(); $$ = $1; }
| '(' expression_list ')' { trace(); $$ = $2; }
;

```

```

VALUE : INT_VALUE  { trace(); $$ = TYPE_INT;  }
      | FLOAT_VALUE { trace(); $$ = TYPE_FLOAT; }
      ;

arith_op : '+' { trace(); $$ = '+'; }
        | '*' { trace(); $$ = '*'; }
        | '-' { trace(); $$ = '-'; }
        | '/' { trace(); $$ = '/'; }
        | '%' { trace(); $$ = '%'; }
        ;

expression_list : arith_op expression expression { trace();
    $$ = (struct expr) {0};
    enum type t1 = $2.type;
    enum type t2 = $3.type;
    assert(type_valid(t1) && type_valid(t2) && type_compat(t1, t2),
        "'c': Types don't match: op1:%s and op2:%s\n",
        $1, type2str(t1), type2str(t2));
    $$ .type = t1;

    assert($$.type != TYPE_FLOAT || $1 != '%',
        "'%': Expected Int, got %s\n",
        type2str($$.type));

    $$ .code = $2.code;
    cbapp($$.code, $3.code);
    gen_op(&$$ .code, $1, t1);
    }
    ;

expression2 : VAR { trace();
    $$ = (struct expr) {0};
    $$ .type = env_typeof(env, $1);
    assert(type_valid($$.type), "Variable not found: '%s'\n", $1);

    gen_pushgp(&$$ .code);
    gen_load(&$$ .code, env_var_gp_idx(env, $1));
    }
    | BOOL_VALUE { trace();
    $$ = (struct expr) {0};
    $$ .type = TYPE_BOOL;
    gen_push(&$$ .code, TYPE_BOOL, NULL, yylval.valBool);
    }
    | '(' expression2_list ')' { trace(); $$ = $2; }
    ;

num_op : '<' { trace(); $$ = '<'; }
       | '=' { trace(); $$ = '='; }
       | '>' { trace(); $$ = '>'; }
       | GEQ { trace(); $$ = GEQ; }
       | LEQ { trace(); $$ = LEQ; }
       | NEQ { trace(); $$ = NEQ; }
       ;

```

```

log_op : '&' { trace(); $$ = '&'; }
      | '|' { trace(); $$ = '|'; }
      ;

expression2_list : '~' expression2 { trace();
    $$ = (struct expr) {0};
    enum type t = $2.type;
    assert(t == TYPE_BOOL, "'~': Expected Bool, got %s\n", type2str(t));
    $$ .type = t;

    $$ .code = $2.code;

    gen_op(&$$ .code, '~', $2.type);
}
| num_op expression expression { trace();
    $$ = (struct expr) {0};
    enum type t1 = $2.type;
    enum type t2 = $3.type;
    assert(type_valid(t1) && type_valid(t2) && type_compat(t1, t2),
        "'%c': Types don't match: op1:%s and op2:%s\n",
        $1, type2str(t1), type2str(t2));
    $$ .type = TYPE_BOOL;

    $$ .code = $2.code;
    cbapp($$ .code, $3.code);
    gen_op(&$$ .code, $1, t1);
}
| log_op expression2 expression2 { trace();
    enum type t1 = $2.type;
    enum type t2 = $3.type;
    assert(t1 == TYPE_BOOL && t2 == TYPE_BOOL,
        "'%c': Expected Bool, got op1:%s and op2:%s\n",
        $1, type2str(t1), type2str(t2));

    $$ = $2;

    cbapp($$ .code, $3.code);
    gen_op(&$$ .code, $1, t1);
}
;

%%

int yyerror (const char *s)
{
    return fprintf(stderr, "ERRO: '%s'\n", s);
}

```