

João Victor Alves de Meira e Thomas Yoshihiro Kofuji

Trabalho Prático 1
Construção de Analisadores Léxico e Sintático

PONTA GROSSA

2024

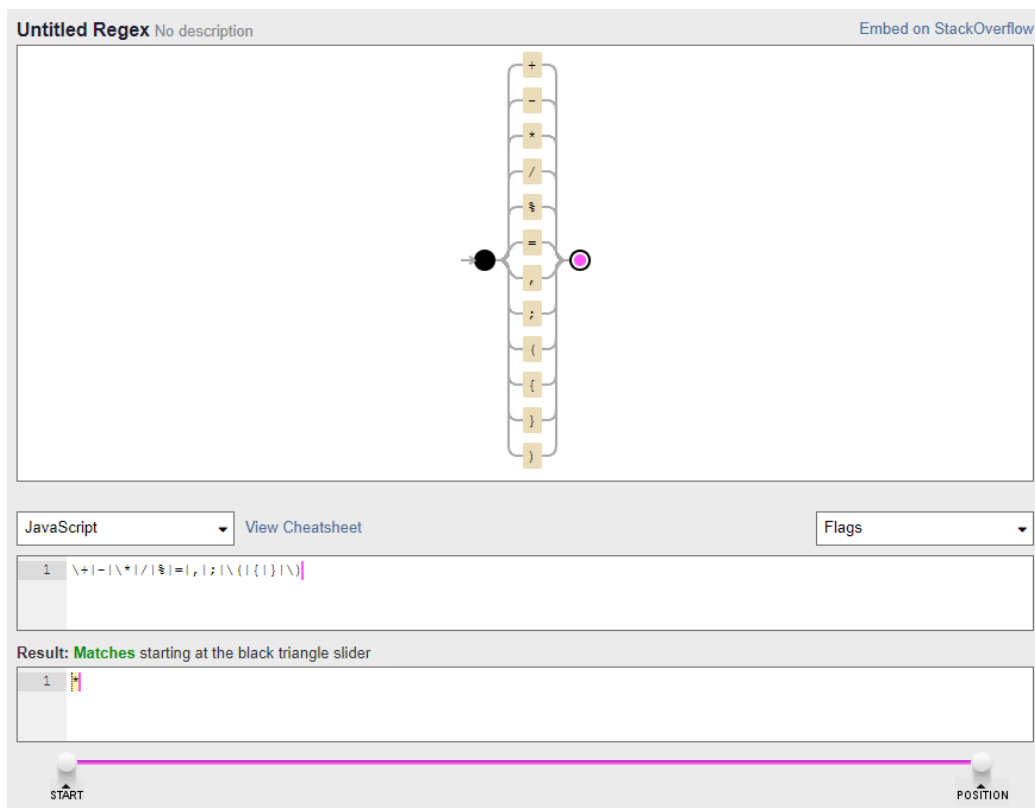
1. Linguagem de Programação Básica

A linguagem de programação apresentada neste trabalho é uma linguagem com o objetivo de realizar condições, loopings, expressões numéricas e criar funções. Para isso foram feitos os analisadores Léxico e Sintático, que definem a forma como devem ser estruturados os códigos que realizarão os objetivos dessa linguagem.

Os comandos que foram implementados nesta linguagem são o condicional “se, então”, “se, então, senão”, “enquanto, faça”, “para, faça”, também são aceitas expressões, comparações e por fim a criação de qualquer função utilizando esses comandos e expressões.

2. Análise Léxica - Diagrama

A análise léxica é a fase onde são definidas quais palavras serão aceitas na linguagem de programação, utilizando de regras léxicas, como expressões regulares, então temos:



Neste diagrama estão representadas as regras para os operadores que só possuem um caractere;

Untitled Regex No description [Embed on StackOverflow](#)

JavaScript [View Cheatsheet](#) Flags

```
1 >|<|<>|==|>=|<=
```

Result: Matches starting at the black triangle slider

```
1 >
```

POSITION

Neste diagrama estão as regras para os operadores de comparação;

Untitled Regex No description [Embed on StackOverflow](#)

JavaScript [View Cheatsheet](#) Flags

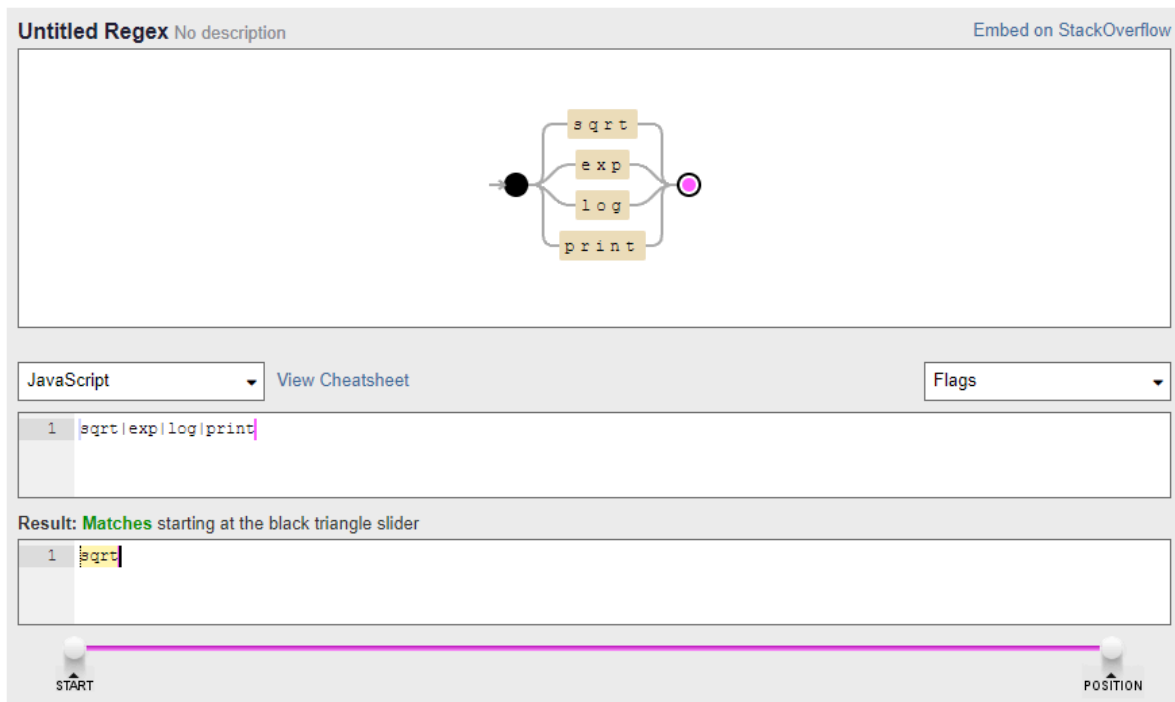
```
1 if|then|else|while|do|let|for|and|or
```

Result: Matches starting at the black triangle slider

```
1 while
```

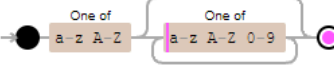
START POSITION

Neste diagrama agora estão representadas as regras para os condicionais e para os comandos de loop, como o “for” e o “while”, assim como a regra para aceitar o and, or e o “let” que serve para criação de funções dentro dessa linguagem;



Neste diagrama estão representadas as regras para as funções prontas da linguagem (“sqrt”, “exp”, “log”, “print”);

Untitled Regex No description [Embed on StackOverflow](#)



JavaScript [View Cheatsheet](#) Flags

```
1 [a-zA-Z][a-zA-Z0-9]*
```

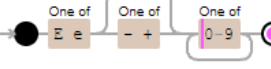
Result: **Matches** starting at the black triangle slider

```
1 printSquaresOfPrimesBetween
```

START POSITION

Neste diagrama está representada a regra para os identificadores desta linguagem, ou seja, para a nomeação de variáveis ou funções criadas pelo código, são aceitas palavras que começam com qualquer letra maiuscula ou minúscula, seguido de qualquer caractere alfanumérico;

Untitled Regex No description [Embed on StackOverflow](#)



JavaScript [View Cheatsheet](#) Flags

```
1 [Ee][+-]?[0-9]+
```

Result: **Matches** starting at the black triangle slider

```
1 7e+7
```

START POSITION

Untitled Regex No description [Embed on StackOverflow](#)

JavaScript [View Cheatsheet](#) [Flags](#)

```
1 [0-9]+\.[0-9]*([Ee][+-]?[0-9])?|(\.[0-9]+)([Ee][+-]?[0-9])?
```

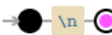
Result: **Matches** starting at the black triangle slider

```
1 2.3E+1
```

START POSITION

Estes diagramas representam a regra para aceitar números, podendo ser números inteiros, com ponto flutuante ou até mesmo com expoente. O primeiro diagrama mostra a regra para o expoente, e o segundo utiliza dele para formar qualquer número do formato explicado;

Untitled Regex No description [Embed on StackOverflow](#)



JavaScript [View Cheatsheet](#) Flags

1 \n

Result: **Matches** starting at the black triangle slider


1
2

START POSITION

Detailed description: This is a screenshot of a web-based regex testing tool. The title bar shows 'Untitled Regex' and a link to 'Embed on StackOverflow'. The main workspace contains a diagram of a regular expression engine state: a black circle (start), a yellow box containing '\n', and a pink circle (end). Below this, a dropdown menu is set to 'JavaScript' with a 'View Cheatsheet' link and a 'Flags' dropdown. The input field contains the text '\n'. The results section shows 'Result: Matches starting at the black triangle slider' and a table with two rows: row 1 with a yellow highlight and row 2 with a pink highlight. At the bottom, a horizontal slider bar with 'START' and 'POSITION' markers is shown.

Este diagrama representa a regra para aceitar a quebra de linha (\n);

Untitled Regex No description [Embed on StackOverflow](#)



JavaScript [View Cheatsheet](#) Flags

1 .

Result: **Matches** starting at the black triangle slider

1

START POSITION

Detailed description: This is a screenshot of the same web-based regex testing tool. The title bar is identical. The main workspace shows a diagram with a black circle (start), a yellow box containing '.', and a pink circle (end). The dropdown menu is still 'JavaScript' with the 'View Cheatsheet' link and 'Flags' dropdown. The input field now contains the text '.'. The results section shows 'Result: Matches starting at the black triangle slider' and a table with one row: row 1 with a yellow highlight. The bottom slider bar with 'START' and 'POSITION' markers remains the same.

Por fim, este diagrama mostra a regra que recebe todas as palavras que não se encaixam em nenhuma das outras regras acima, considerando essas palavras como um erro de sintaxe.

3. Análise Léxica - FLEX

A seguir são mostradas as regras léxicas utilizadas nesta linguagem de programação através do FLEX:

```
"+" | /* Operadores de caractere único */
"-" |
"*" |
"/" |
"%" |
"=" |
"," |
";" |
"(" |
"{" |
"}" |
")" { return yytext[0]; }
```

Estas regras apenas retornam o próprio símbolo que foi reconhecido, com isso o analisador aceita qualquer um desses símbolos de operadores nesta linguagem;

```
">" { yylval.fn = 1; return CMP; } /* operadores de comparacao, todos sao token CMP */
"<" { yylval.fn = 2; return CMP; }
"<>" { yylval.fn = 3; return CMP; }
"==" { yylval.fn = 4; return CMP; }
">=" { yylval.fn = 5; return CMP; }
"<=" { yylval.fn = 6; return CMP; }
```

Estes são os operadores de comparação, eles estão sendo retornados como “CMP” junto de um valor atribuído ao símbolo, para ser possível diferenciar na hora de decidir o operador escolhido;

```

"if"    { return IF; }      /* Palavras chave */
"then"  { return THEN; }
"else"  { return ELSE; }
"while" { return WHILE; }
"do"    { return DO; }
"let"   { return LET; }
"for"   { return FOR; }
"and"   { return AND; }
"or"    { return OR; }

```

Estas são as palavras chaves, são palavras que são reservadas para identificar um comando específico da linguagem, são retornadas as próprias palavras, pois elas só correspondem a elas mesmas;

```

"sqrt" { yylval.fn = B_sqrt; return FUNC; } /* Funções pré-definidas */
"exp"  { yylval.fn = B_exp; return FUNC; }
"log"  { yylval.fn = B_log; return FUNC; }
"print" { yylval.fn = B_print; return FUNC; }

```

Estas são funções pré-definidas, ou seja, funções já prontas, onde são retornadas como "FUNC" junto de um valor atribuído ao seu nome, para que seja possível diferenciar cada função;

```

[a-zA-Z][a-zA-Z0-9]* { yylval.s = lookup(yytext); return NAME; } /* Nomes */

```

Esta é a regra léxica para os nomes de variáveis ou funções da linguagem, onde são aceitas palavras que começam com uma letra (maiúscula ou minúscula) seguida de zero ou mais caracteres que podem ser tanto letras ou algarismos;

```

/* Expoente float */
EXP ([Ee][-+]?[0-9]+)

```

```

[0-9]+ "." [0-9]* {EXP}? |
"."? [0-9]+ {EXP}? { yylval.d = atof(yytext); return NUMBER; }

```

Esta é a regra léxica para os valores numéricos da linguagem, onde são aceitos números inteiros, números com ponto flutuante e também números exponenciais;

```
"/".*  
[ \t] /* Igora comentários */
```

Esta regra permite a utilização de comentários no código desta linguagem, ou seja, linhas de código que serão ignoradas pelo compilador;

```
\\n { printf("c> "); } /* Ignora continuação de linha */
```

Esta regra apenas ignora a continuação das linhas do código desta linguagem;

```
\n { return EOL; }
```

Esta regra apenas ignora a quebra de linha do código desta linguagem;

```
. { yyerror("Caractere desconhecido %c\n", *yytext); }
```

Por fim, temos esta última regra que receberá qualquer palavra que não esteja especificada em uma das regras apresentadas acima, causando um erro de sintaxe na linguagem.

4. Tabela de Símbolos

A tabela de símbolos é usada para armazenar variáveis e funções. A ideia principal de uma tabela de símbolos é mapear nomes de variáveis e funções para seus respectivos valores e metadados.

```

/* Tabela de símbolos */
struct symbol{
    char *name;
    double value;
    struct ast *func; /* Stmt para função */
    struct symlist *syms; /* Lista de argumentos */
};

/* Tabela de símbolos de tamanho fixo */
#define NHASH 9997
extern struct symbol symtab[NHASH];

```

Campo name:

Nome da variável ou função (uma string, como "x", "y", ou "soma").
Representa a "chave" que será usada para buscar o símbolo na tabela.

Campo value:

O valor associado à variável, caso seja uma variável simples. Por exemplo, se a variável for $x = 5.0$, o valor armazenado seria 5.0. Para funções, esse campo pode ser ignorado.

Campo astfunc:

Para funções, aponta para uma árvore sintática abstrata (AST) com nodetype igual a 'L' que representa o corpo da função. Isso permite a execução de vários comandos.

Campo syms:

Lista de argumentos da função, caso o símbolo represente uma função. Essa lista define os nomes das variáveis locais usadas como parâmetros da função.

#define NHASH 9997

extern struct symbol symtab[NHASH];

Declara a tabela de símbolos como um vetor de structs do tipo symbol. O extern indica que ela será definida em outro arquivo, mas pode ser usada em qualquer lugar onde seja declarada.

```

/* Função Hashing */
static unsigned symhash(char *sym){

    unsigned int hash = 0;
    unsigned c;

    while(c = *sym++)
        hash = hash * 9 ^ c;

    return hash;
}

struct symbol *lookup(char *sym){

    struct symbol *sp = &syntab[symhash(sym) % NHASH];
    int scout = NHASH;

    while(--scout >= 0){

        if(sp->name && !strcasecmp(sp->name, sym))
            return sp;

        if(!sp->name){ /* Nova entrada na TS */

            sp->name = strdup(sym);
            sp->value = 0;
            sp->func = NULL;
            sp->syms = NULL;
            return sp;

        }

        if(++sp >= syntab + NHASH)
            sp = syntab; /* Tenta a próxima entrada */

    }

    yyerror("Overflow na tabela de símbolos\n");
    abort(); /* Tabela está cheia */

}

```

Função symhash:

Gera um valor hash único para um símbolo (nome da variável ou função). Cada caractere do nome influencia o valor do hash.

Função lookup:

Procura o símbolo na tabela usando o hash gerado por symhash, se o símbolo já existe na tabela (comparação com strcmp) é retornado o ponteiro para a entrada correspondente. Se o símbolo não existe é criada uma nova entrada na tabela, inicializado o nome, valor, outros campos (value = 0, func = NULL, syms = NULL) e retorna o ponteiro para essa nova entrada. Se a tabela estiver cheia um erro é gerado com yyerror.

Tratamento de colisões:

Caso duas entradas resultem no mesmo índice de hash, a função avança para a próxima posição disponível, e retorna ao início da tabela, se necessário.

Função dodef:

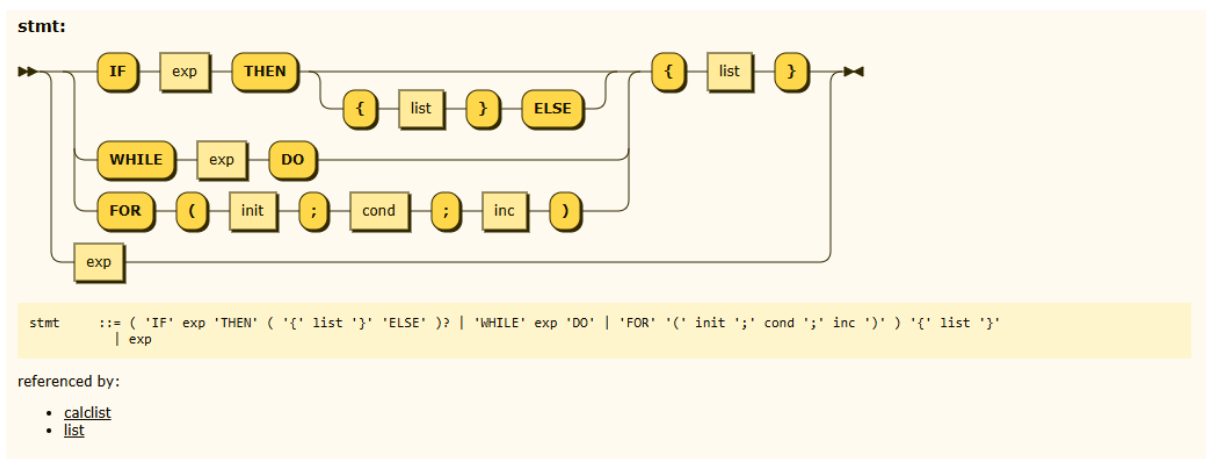
Preenche os campos syms (lista de variáveis/argumentos) e func (comandos/AST) de uma entrada na tabela. Essa função é chamada quando o usuário define uma função usando o comando let.

Função newasgn:

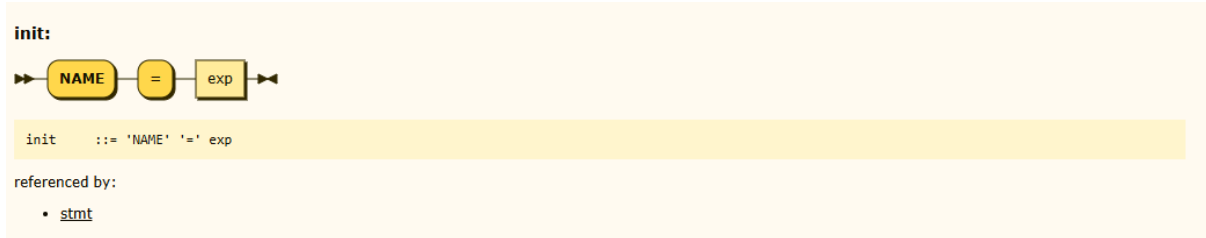
Atribui um valor ao campo value de uma entrada na tabela, representando a atribuição de uma variável.

5. Análise Sintática - Diagrama

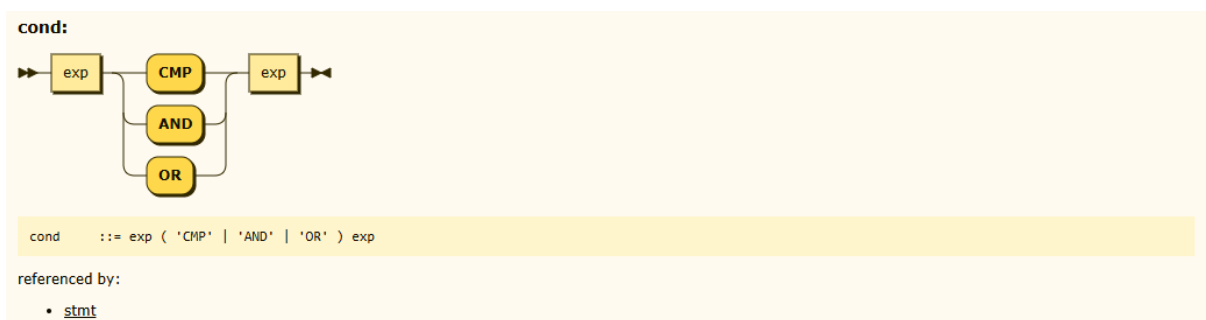
A análise sintática é a fase onde é definido como devem ser escritos os comandos da linguagem, ou seja, a ordem que devem aparecer as palavras chaves e expressões para conseguir realizar uma ação definida nesta linguagem de programação. Então temos os seguintes diagramas para as regras da análise sintática, montada como uma gramática livre de contexto:



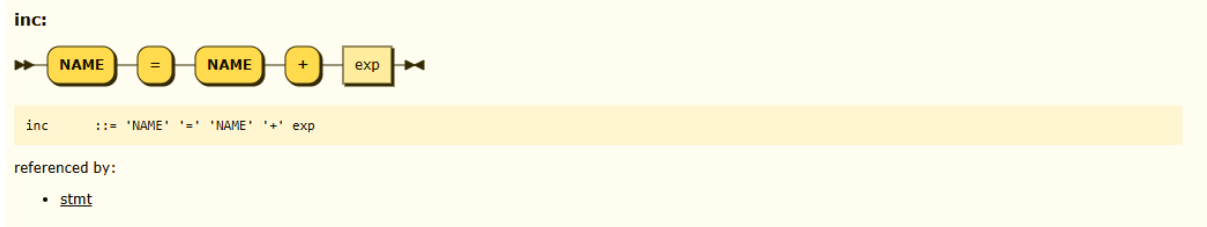
Nesta regra temos a cabeça “stmt” que pode seguir para os caminhos dos comandos “IF”, “IF e ELSE”, “WHILE”, “FOR” e “exp”, onde cada um tem um formato e ordem que deve ser obedecido, tendo suas variáveis representadas pelas palavras com letras maiúsculas e os terminais representados pelos símbolos entre aspas e as palavras com letras minúsculas;



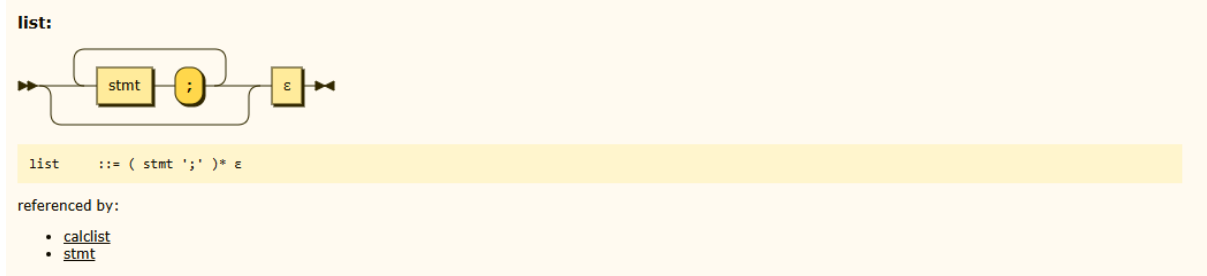
Esta regra define o “init” antes usado no comando “FOR” das regras anteriores, podendo gerar uma declaração de uma variável;



Esta regra define o “cond” também antes usado no comando “FOR”, porém esta regra gera uma comparação entre expressões, utilizando dos comparadores vistos na análise léxica “CMP” e os comparadores lógicos “AND” e “OR”;

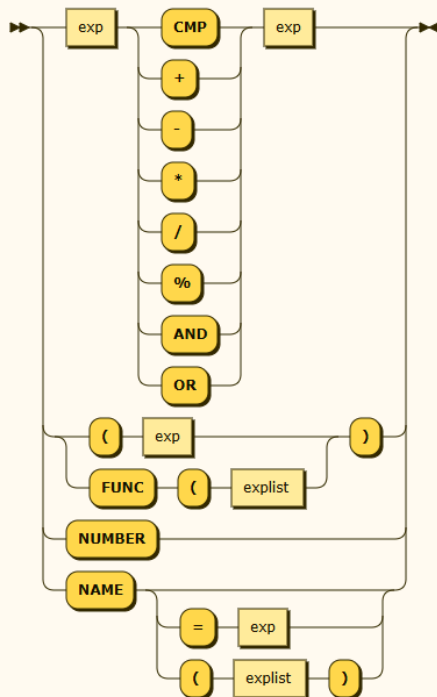


Esta regra define “inc” que também faz parte do comando “FOR”, ela gera um incremento em uma variável;



A regra “list” é uma regra recursiva, que pode gerar nenhum ou mais símbolos de “stmt” que é a primeira regra, seguindo de um delimitador que é o ponto e vírgula (;);

exp:

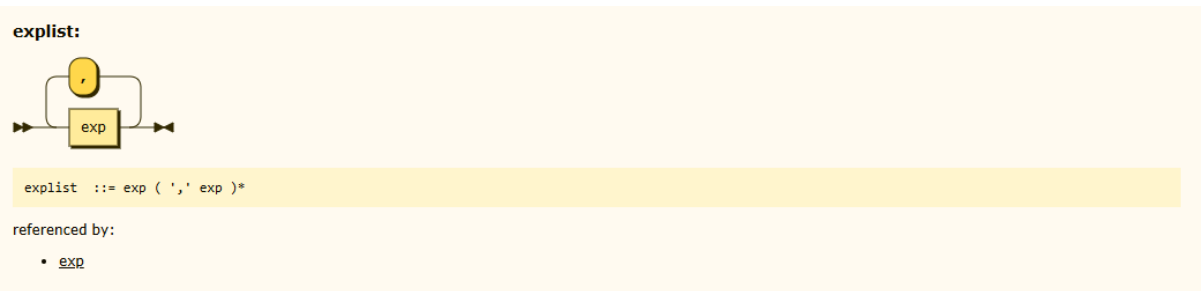


```
exp ::= exp ( 'CMP' | '+' | '-' | '*' | '/' | '%' | 'AND' | 'OR' ) exp
      | '(' exp | 'FUNC' '(' explist ')'
      | 'NUMBER'
      | 'NAME' ( '=' exp | '(' explist ')' )?
```

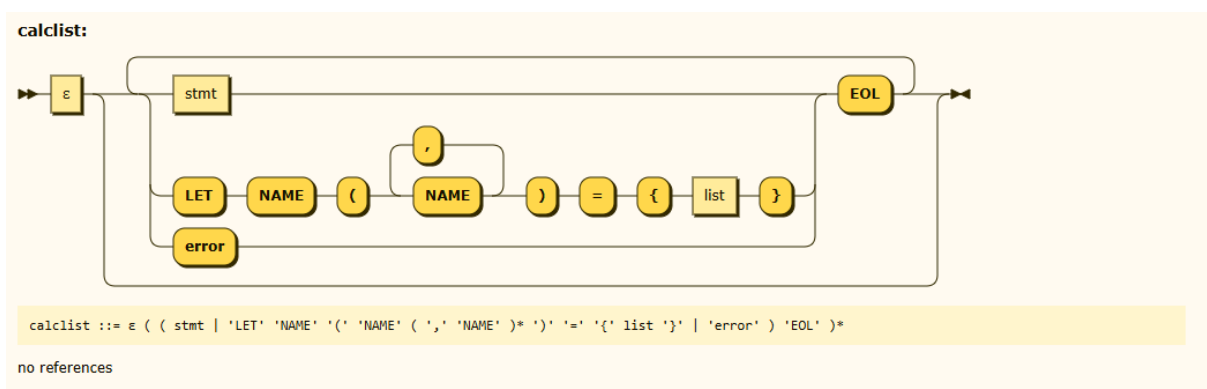
referenced by:

- [cond](#)
- [exp](#)
- [explist](#)
- [inc](#)
- [init](#)
- [stmt](#)

A regra “exp” gera qualquer comparação entre expressões utilizando o “CMP” antes visto na análise léxica, qualquer soma, subtração, multiplicação e divisão entre expressões, o resto da divisão entre duas expressões (mod), uma comparação utilizando os comparadores lógicos “AND” e “OR”, a delimitação de uma outra expressão através de parênteses, um terminal “NUMBER”, um terminal “NAME”, uma atribuição de uma expressão em um “NAME”, e por fim uma função, sendo ela uma função criada (“NAME”) ou uma já existente (“FUNC”);



A regra “explist” é uma regra recursiva que gera um ou mais “exp” seguidos por um ponto e vírgula (;);



Por fim temos a regra “calclist”, que é uma regra recursiva que gera nenhuma ou mais regras “stmt” com um fim de linha (“EOL”), ou criações de funções usando o “LET” seguidos de um fim de linha, ou um erro (“error”), caso algum comando tenha falhado, seguido de um fim de linha.

A regra “symlist” é uma regra recursiva que gera um ou mais terminais “NAME” seguidos por um ponto e vírgula (;), porém ela foi reduzida na regra “calclist”;

6. Análise Sintática - BISON

A seguir são mostradas as regras sintáticas utilizadas nesta linguagem de programação através do BISON:

```

stmt: IF exp THEN '{' list '}'           { $$ = newflow('I', $2, $5, NULL); }
    | IF exp THEN '{' list '}' ELSE '{' list '}' { $$ = newflow('I', $2, $5, $9); }
    | WHILE exp DO '{' list '}'           { $$ = newflow('W', $2, $5, NULL); }
    | FOR '(' explist ';' exp ';' explist ')' '{' list '}' { $$ = newfor('O', $3, $5, $7, $10); }
    | exp
    ;

```

Estas são as regras de “stmt” para os condicionais “IF”, “IF e ELSE”, para os loopings “WHILE” e “FOR” e para o terminal “exp”. Nestas regras, tirando a que vai para “exp”, são criados nodos na árvore para cada um dos tipos;

```

list: { $$ = NULL; }
    | stmt ';' list { if( $3 == NULL)
                    | $$ = $1;
                    else
                    | $$ = newast('L', $1, $3);
                    }
    ;

```

Esta é a regra recursiva “list”, onde ocorre uma verificação para saber quando a recursão acaba, colocando sempre um novo nodo na árvore enquanto estão sendo adicionadas regras “stmt”;

```

exp: exp CMP exp           { $$ = newcmp($2, $1, $3); }
    | exp '+' exp          { $$ = newast('+', $1, $3); }
    | exp '-' exp          { $$ = newast('-', $1, $3); }
    | exp '*' exp          { $$ = newast('*', $1, $3); }
    | exp '/' exp          { $$ = newast('/', $1, $3); }
    | exp '%' exp          { $$ = newast('%', $1, $3); }
    | exp '&' exp           { $$ = newast('&', $1, $3); }
    | exp '|' exp          { $$ = newast('|', $1, $3); }
    | '(' exp ')'          { $$ = $2; }
    | NUMBER               { $$ = newnum($1); }
    | NAME                  { $$ = newref($1); }
    | NAME '=' exp         { $$ = newasgn($1, $3); }
    | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
    | NAME '(' explist ')' { $$ = newcall($1, $3); }
    ;

```

Estas são as regras de “exp”, onde é possível fazer qualquer operação matemática simples, a operação de resto (mod) e os operadores lógicos “AND” e “OR” entre duas “exp”, também é possível ter uma delimitação por parênteses entre uma “exp”, além de trocar por terminais como “NUMBER” e “NAME” ou uma atribuição de uma “exp” para um “NAME”, e por fim as funções, sendo ela uma função pronta ou uma função criada em código;

```

explist: exp
| exp ',' explist { $$ = newast('L', $1, $3); }
;

```

Esta é a regra “explist”, onde ocorre uma recursão adicionando termos “exp” separados por vírgula (,);

```

symlist: NAME { $$ = newsymlist($1, NULL); }
| NAME ',' symlist { $$ = newsymlist($1, $3); }
;

```

Está é a regra “symlist”, onde ocorre uma recursão adicionando termos “NAME” separados por vírgula (,);

```

calclist:
| calclist stmt EOL {
    printf("= %4.4g\n", eval($2));
    treefree($2);
}
| calclist LET NAME '(' symlist ')' '=' '{' list '}' EOL {
    dodef($3, $5, $9);
    printf("Defined %s\n", $3->name);
}
| calclist error EOL { yyerrok; printf("> "); }
;

```

E por fim, a regra “calclist”, que gera nenhum ou mais regras “stmt” seguido de um fim de linha (“EOL”), ou a declaração de uma função usando o comando “LET” seguido de um fim de linha ou um erro (“error”) seguido de um fim de linha;

7. Conjunto de Testes

Para finalizar, temos um conjunto de testes utilizando todas as funcionalidades da linguagem de programação básica:

Teste 1

```
let printSquaresOfPrimesBetween(n, m) = {  
  if n > m then {  
    temp = n;  
    n = m;  
    m = temp;  
  };  
  
  for(i = n; i <= m; i = i + 1) {  
    if i >= 2 then {  
      isPrime = 1;  
  
      for(j = 2; j * j <= i; j = j + 1) {  
        temp = i;  
  
        while temp >= j do {  
          temp = temp - j;  
        };  
  
        if temp == 0 then {  
          isPrime = 0;  
        };  
      };  
  
      if isPrime == 1 then {  
        print(i * i);  
      };  
    };  
  };  
}  
printSquaresOfPrimesBetween(1, 10)
```

Neste teste foi criada uma função utilizando o comando “LET”, onde o objetivo da função é imprimir na tela o quadrado de todos os números primos dentro de um intervalo passado pelos parâmetros dessa função, e no final essa função é chamada com os parâmetros 1 e 10, ou seja, deve aparecer na tela os resultados 4, 9, 25 e 49;

```
##### Teste 2

let fibonacciSum(n) = {

    a = 1;
    b = 1;
    sum = 0;

    for(i = 0; i < n; i = i + 1) {
        sum = sum + a;
        temp = a;
        a = b;
        b = temp + b;
    };

    print(sum);
}
fibonacciSum(5)
```

Neste teste foi criada uma função utilizando o comando “LET”, onde o objetivo desta função é calcular a soma de Fibonacci de um número que é passado pelo parâmetro da função. No final essa função é chamada passando o número 5 como parâmetro, então o resultado deve ser 12;

```
≡ teste3.txt
1    3 + 5
2    10 - 4 * 2
3    (8 + 2) * 3
4    
```

Neste teste são realizadas algumas operações matemáticas onde os resultados devem ser respectivamente 8, 2 e 30;

```

≡ teste4.txt
1  x = 5
2  x + 3
3  y = x * 2
4  y / 2
5

```

Neste teste foi feita uma atribuição à variável x, depois é feita uma operação utilizando essa variável, o resultado deve ser 8, o mesmo é feito com y, porém com uma operação na atribuição, depois uma divisão, o resultado deve ser 10;

```

##### Teste 5

let fatorial(n) = {
  if(n < 0) then {
    (-1);
  } else {
    if((n == 0) or (n == 1)) then {
      1;
    } else {
      n * fatorial(n - 1);
    };
  };
};

fatorial(6)
fatorial(4)
fatorial((1 - 2))

```

Neste teste foi criada uma função utilizando o comando “LET”, onde o objetivo desta função é calcular o fatorial de um número que é passado pelo parâmetro da função. Depois essa função é chamada passando os valores 6, 4 e uma operação que resulta em -1, o resultado deve ser respectivamente, 720, 24 e -1;

```

joaovictoradm@DESKTOP-E0FIK82:~/projeto1-teste-arquivos$ ./calc teste1.txt teste2.txt teste3.txt teste4.txt teste5.txt
Lendo arquivo: teste1.txt

Defined printSquaresOfPrimesBetween
= 4
= 9
= 25
= 49
= 11

Arquivo teste1.txt processado.

Lendo arquivo: teste2.txt

Defined fibonacciSum
= 12
= 12

Arquivo teste2.txt processado.

Lendo arquivo: teste3.txt

= 8
= 2
= 30

Arquivo teste3.txt processado.

Lendo arquivo: teste4.txt

= 5
= 8
= 10
= 5

Arquivo teste4.txt processado.

Lendo arquivo: teste5.txt

Defined fatorial
= 720
= 24
= -1

Arquivo teste5.txt processado.

```

Este é o terminal contendo os resultados de todos os testes utilizando a linguagem de programação básica.

Agora temos um conjunto de testes que retornam um erro de sintaxe:

```

##### Teste 6

for(x = 0,; x < 10; x = x + 1){
    print(x);
}

Lendo arquivo: teste6.txt
1: error: syntax error
>
Arquivo teste6.txt processado.

```

Neste teste há um erro de sintaxe onde foi inserida uma vírgula após a condição de início do comando “FOR”;


```
let sum(n1, n2) = { n1 + n2; }  
sum(13; 22)  
|
```

```
Lendo arquivo: teste8.txt  
Defined sum  
2: error: syntax error  
>  
Arquivo teste8.txt processado.
```

Neste teste foi usado um ponto e vírgula ao invés de uma vírgula nos parâmetros da chamada da função sum;

```
x = 0  
if x == 0 then print(x);  
|
```

```
Lendo arquivo: teste7.txt  
=  
0  
2: error: syntax error  
>  
Arquivo teste7.txt processado.
```

Neste teste está faltando as chaves delimitadoras do condicional "IF";