Na apostila anterior (*Analisador Sintático através de Calculadora*) foi apresentada a utilização das ferramentas de compilação flex & bison através da construção de uma calculadora. Agora, essa calculadora é estendida com a adição de comandos e funções pré-definidas, bem como aquelas definidas pelo usuário. O restante do documento está organizado da seguinte maneira. Na Seção 1 discute-se os conflitos que podem ocorrer nas reduções *shift-reduce* e a precedência de operadores. A Seção 2 apresenta uma descrição geral da calculadora que será construída nesse documento. Na Seção 3 tem-se a definição de todos elementos do analisador sintático (*parser*) da calculadora baseada em comandos e funções. Após, a Seção 4 detalha a construção e interpretação das AST's (*Abstract Syntax Tree*) utilizadas na implementação da calculadora.

A referência principal para elaboração desse documento é:

LEVINE, John. **Flex & Bison**. Editora O'Reilly. 2009. 271 p.

1 Conflitos e Precedência de Operadores

A definição de precedência de operadores (conforme visto na construção de algumas gramáticas) pode ser estabelecida através das diferentes regras para `exp`, `factor` e `term`.

Caso tenhamos uma gramática com mais regras e operadores, e consequentemente mais níveis de precedência. O resultado é uma gramática difícil de ler e manter.

Assim, o Bison provê uma maneira para descrever a precedência de operadores separada da organização das regras sintáticas, o que torna a gramática menor e mais fácil de manter e inclusive estender.

Incialmente, todas expressões são escritas utilizando símbolos `exp` (ver Exemplo 1):

```
1  %type <a> exp

3  %%
     ...
5  exp: exp '+' exp { $$ = newast('+', $1, $3); }
     | exp '-' exp { $$ = newast('-', $1, $3); }
7    | exp '*' exp { $$ = newast('*', $1, $3); }
     | exp '/' exp { $$ = newast('/', $1, $3); }
9    | '(' exp ')' { $$ = $2; }
     | NUMBER { $$ = newnum($1); }
11   ;
   %%
```

Listing 1: Exemplo Precedência de Operadores – 1

Note que essa gramática do Exemplo 1 tem um problema, ela é extramamente ambígua. Por exemplo, a entrada $2 + 3 * 4$ pode significar $(2 + 3) * 4$ ou $2 + (3 * 4)$, conforme ilustra a Fig. 1.

Se essa gramática for compilada no Bison, o compilador irá informar que existem 24 conflitos *shift-reduce*. O problema ocorre pois não foi dito ao Bison a precedência e associatividade de operadores. Em qualquer gramática de expressões, os operadores são agrupados em níveis de precedência do mais baixo para o mais alto. O número total de níveis depende da linguagem de programação, a linguagem C tem muitos níveis de precedência, um total de 15 níveis.

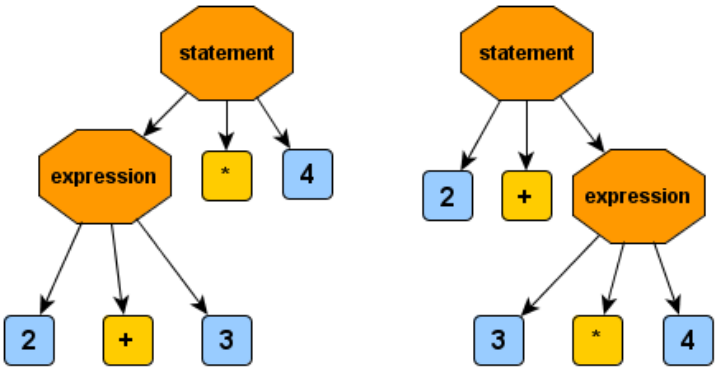

Figura 1: Exemplo de Precedência de Operadores

O operador de associatividade controla o agrupamento de operadores no mesmo nível de precedência. Operadores podem ser associados à esquerda ou à direita.

Há duas maneiras para especificar precedência e associatividade numa gramática, implícita e explicitamente. A maneira implícita é aquela vista anteriormente, ou seja através da organização das regras da gramática. Aqui será apresentada também a forma explícita, conforme visto no Exemplo 2. É possível adicionar essas linhas (de código Bison) na seção de declaração da especificação feita no Bison, para então resolver as conflitos.

```
%left '+' '-'
%left '*' '/'

%type <a> exp

%%
   ...
exp: 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); }
   ;
%%
```

Listing 2: Exemplo Precedência de Operadores – 2

Cada uma dessas declarações define um nível de precedência, conforme a ordenação das declarações $\%left$, $\%right$ e $\%noassoc$, as quais definem a ordem de precedência da menor para a maior. Essas declarações dizem ao Bison que os operadores $+$ e $-$ são associativos à esquerda e que possuem a menor precedência; $*$ e $/$ são associativos à esquerda e têm maior precedência.

Quando o Bison encontra um conflito *shift-reduce*, a tabela de precedência é consultada para resolver o conflito. Na gramática em questão, todos os conflitos ocorrem nas regras da forma `exp OP exp`, então definindo precedências para os quatro operadores aritméticos é suficiente para resolver os conflitos.

A construção do *parser* através de precedência explícita é ligeiramente mais rápida e menor (considerando a quantidade de regras) que o *parser* através da precedência implícita.

Observação: a utilização de regras de precedência é recomendada para resolver conflitos apenas para gramática de expressões e de comandos como if/then/else. Não recomenda-se utilizar precedência para resolver outros conflitos *shift-reduce* que podem sugir em outras estruturas da gramática. Possivelmente, outros conflitos indicam que a gramática tem ambiguidades

e assim é adequado que a própria gramática seja revista e corrigida.

2 Uma Calculadora Avançada

Nesta apostila, a calculadora vista nas apostilas anteriores é estendida para uma versão mais próxima de uma aplicação real de técnicas de compilação. Assim são adicionados: (**i.**) nomes de variáveis e atribuições; (**ii.**) expressões de comparação (>, <, ...); (**iii.**) controle de fluxo com if/then/else e while/do; (**iv.**) funções pré-definidas; (**v.**) funções definidas pelo usuário e (**vi.**) técnicas de recuperação de erro.

A versão anterior da calculadora não apresentava muitas vantagens em utilizar AST para representar expressões. Porém, nessa versão a AST é o aspecto chave para implementar estruturas de controle de fluxo e funções definidas pelo usuário.

Aqui apresenta-se um exemplo de uma função definida pelo usuário, no caso como chamá-la e ainda como usar um função pré-definida como argumento:

```
> let avg(a, b) = (a+b)/2;
Defined avg
> avg(3,sqrt(25))
= 4
```

No Código 3 apresentam-se as declarações do arquivo de cabeçalho para a construção da calculadora.

```
1  /*
   * Declaracoes para uma calculadora avancada
3  */

5  /* interface com o lexer */
   extern int yylineno;
7  void yyerror(char *s, ...);

9  /* tab. de simbolos */
   struct symbol {       /* um nome de variavel */
11   char *name;
     double value;
13   struct ast *func; /* stmt para funcao */
     struct symlist *syms; /* lista de argumentos */
15 };

17 /* tab. de simbolos de tamaho fixo */
   #define NHASH 9997
19 struct symbol symtab[NHASH];

21 struct symbol *lookup(char*);

23 /* lista de simbolos, para uma lista de argumentos */
   struct symlist {
25   struct symbol *sym;
     struct symlist *next;
27 };

29 struct symlist *newsymlist(struct symbol *sym, struct symlist *next);
   void symlistfree(struct symlist *sl);
31
   /* tipos de nos
```

```
33 4*  + - * /
   *  0-7 operadores de comparacao, 04 igual, 02 menor que, 01 maior que
35 *  L expressao ou lista de comandos
   *  I comando IF
37 *  W comando WHILE
   *  N symbol de referencia
39 *  = atribuicao
   *  S lista de simbolos
41 *  F chamada de funcao pre-definida
   *  C chamada de funcao def. p/ usuario
43 */

45 enum bifs {   /* funcoes pre-definidas */
     B_sqrt = 1,
47   B_exp,
     B_log,
49   B_print
   };
51
   /* nos na AST */
53 /* todos tem o "nodetype" inicial em comum */

55 struct ast {
     int nodetype;
57   struct ast *l;
     struct ast *r;
59 };

61 struct fncall {   /* funcoes pre-definida */
     int nodetype; /* tipo F */
63   struct ast *l;
     enum bifs functype;
65 };

67 struct ufncall {     /* funcoes usuario */
     int nodetype;    /* tipo C */
69   struct ast *l;     /* lista de argumentos */
     struct symbol *s;
71 };

73 struct flow {
     int nodetype;    /* tipo I ou W */
75   struct ast *cond;  /* condicao */
     struct ast *tl;    /* ramo "then" ou lista "do" */
77   struct ast *el;    /* ramo opcional "else" */
   };
79
   struct numval {
81   int nodetype;    /* tipo K */
     double number;
83 };

85 struct symref {
     int nodetype;    /* tipo N */
87   struct symbol *s;
   };
89
   struct symasgn {
91   int nodetype;    /* tipo = */
     struct symbol *s;
```

```
93    struct ast *v;      /* valor a ser atribuido */
};
95
/* construcao de uma AST */
97
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
99 struct ast *newcmp(int cmptype, struct ast *l, struct ast *r);
struct ast *newfunc(int functype, struct ast *l);
101 struct ast *newcall(struct symbol *s, struct ast *l);
struct ast *newref(struct symbol *s);
103 struct ast *newasgn(struct symbol *s, struct ast *v);
struct ast *newnum(double d);
105 struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *
    tr);

107 /* definicao de uma funcao */
void dodef(struct symbol *name, struct symlist *syms, struct ast *stmts);
109
/* avaliacao de uma AST */
111 double eval(struct ast *);

113 /* deletar e liberar uma AST */
void treefree(struct ast *);
```

Listing 3: Calculadora – Declarações

A tabela de símbolos é adaptada do exemplo visto nos materias anteriores (**Apostila 01 - Flex, Arquivos e TS**). Na Calculadora, cada símbolo pode ser tanto uma variável, como uma função definida pelo usuário. O campo `value` sinaliza o valor do símbolo como uma variável, o campo `func` aponta para a AST com a definição dado pelo usuário para a função, e `syms` aponta para uma lista encadeada de argumentos. No exemplo anterior, `avg` é a função, a e b são os argumentos.

As funções escritas em C, `newsymlist` e `symlistfree`, respectivamente servem para criar e liberar tais símbolos. Essa versão da calculadora tem diversos tipos de nós na AST. Assim como antes, cada tipo de nó inicia com um `node type`, assim o código implementado para percorrer a árvore pode usar essa informação para determinar qual é o tipo de nó que está sendo avaliado.

O nó básico `ast` é usado em comparações com cada tipo de operador ($>$, $<$, ...) e para a lista de expressões. Funções pré-definidas têm um nó `fncall` com a AST dos argumento e um `enum` que determina qual a função utilizada. Existem três funções padrão: **sqrt**, **exp** e **log**, além da função **print**, a qual é usada para imprimir os argumentos e retornar os argumentos como seus valores.

Chamadas para funções do usuário têm um nó `ufncall` com um ponteiro para função, o qual é uma entrada na tabela de símbolos, e uma AST que é uma lista de argumentos.

Expressões de fluxo de controle if/then/else e while/do utilizam um nó `flow` com a expressão de controle, o ramo `then` ou a lista `do`, e o ramo opcional `else`. Constantes são `numval` assim como na versão anterior da calculadora. Referências para símbolos são `symref` com um ponteiro para o símbolo na tabela de símbolos. Atribuições são `symasg` com um ponteiro para o símbolo a ser atribuído e a AST do valor a ser atribuído.

Toda AST tem uma **valor**. O valor de um comando if/then/else é o valor da ramificação escolhida; o valor do while/do é o último valor da lista referente ao **do**; e o valor da lista de expressões é a última expressão. Por fim, tem-se os procedimentos em C para criar cada nó da AST e um procedimento para criar uma função definida pelo usuário.

3 Parser para a Calculadora

O Código 4 mostra o *parser* para a calculadora com AST.

```
/*
 * Parser para uma calculadora avancada
 */

%{
# include <stdio.h>
# include <stdlib.h>
# include "bison-calc.h"
%}

%union {
  struct ast *a;
  double d;
  struct symbol *s;     /* qual simbolo? */
  struct symlist *sl;
  int fn;       /* qual funcao? */
}

/* declaracao de tokens */
%token <d> NUMBER
%token <s> NAME
%token <fn> FUNC
%token EOL

%token IF THEN ELSE WHILE DO LET

%nonassoc <fn> CMP
%right '='
%left '+' '-'
%left '*' '/'

%type <a> exp stmt list explist
%type <sl> symlist

%start calclist
%%

stmt: IF exp THEN list       { $$ = newflow('I', $2, $4, NULL); }
   | IF exp THEN list ELSE list  { $$ = newflow('I', $2, $4, $6); }
   | WHILE exp DO list    { $$ = newflow('W', $2, $4, NULL); }
   | exp
;

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

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 ')'       { $$ = $2; }
    | NUMBER        { $$ = newnum($1); }
    | NAME        { $$ = newref($1); }
    | NAME '=' exp      { $$ = newasgn($1, $3); }
    | FUNC '(' explist ')'    { $$ = newfunc($1, $3); }
    | NAME '(' explist ')'    { $$ = newcall($1, $3); }
;

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

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

calclist:   /* vazio! */
  | calclist stmt EOL {
    printf("= %4.4g\n> ", eval($2));
    treefree($2);
    }
  | calclist LET NAME '(' symlist ')' '=' list EOL {
          dodef($3, $5, $8);
          printf("Defined %s\n>", $3->name); }
  | calclist error EOL  { yyerrok; printf("> "); }
;
```

Listing 4: Calculadora com AST - *parser*

O comando % `union` define vários tipos de valores de símbolos. Assim como, um ponteiro para uma AST e um valor numérico. Existe um *token* novo `FUNC` para as funções definidas, com o valor indicando qual função e seis palavras reservadas, de `IF` até `LET`. O *token* `CMP` é qualquer um dos seis operadores de comparação, com o valor indicando qual o operador específico é analisado. **Observação**: essa estratégia de usar um *token* para vários operadores sintaticamente similares, ajuda a não aumentar o tamanho da gramática.

A lista de declarações de precedências começa com os novos operadores `CMP` e =. Uma declaração com % `start` identifica a regra de alto nível da gramática, de forma que seja necessário colocá-la no início do *parser*.

3.1 Calculadora: sintaxe dos comandos

A gramática diferencia entre comandos (`stmt`) e expressões (`exp`). Um comando é um controle de fluxo (**if** ou **while**) ou uma expressão. Os comandos **if** e **while** aceitam listas de comandos, onde cada comando na lista é seguido por um ponto-e-vírgula. Cada regra que casa com um comando chama uma rotina para construir um nó AST apropriado.

A abordagem para determinar a sintaxe permite uma certa variação para experimentar diferentes estilos de sintaxe em uma gramática. No caso dessa gramática, se a definição de `list` tivesse ponto-e-vírgula entre os comandos, ao invés de ser no final de cada comando, a gramática seria ambígua, a não ser que fossem adicionados *tokens* específicos, `FI` e `ENDDO` para explicitar o fechamento dos respectivos comandos `if` e `while`.

3.2 Calculadora: sintaxe das expressões

Uma nova regra para `CMP` lida com seis operadores de comparação, usando o valor de `CMP` para determinar qual é o operador analisado e uma regra para atribuições cria um nó de atribuição.

Existem regras separadas para funções pré-definidas identificas por um nome reservado (`FUNC`) e funções definidas pelo usuário identificas pelo símbolo (`NAME`).

Adicionamente, destaca-se uma regra para `explist`, uma lista de expressões, constrói uma AST para expressões utilizadas para argumentos de uma chamada de função. E uma regra específica para `symlist`, uma lista de símbolos, constrói uma lista encadeada de símbolos para argumentos na definição de uma função.

3.3 Calculadora: nível superior da gramática

A última parte da gramática é o nível superior, o qual é responsável por reconhecer uma lista de comandos e declarações de funções. Em linhas gerais, assim como na versão anterior da calculadora, a gramática avalia a AST para um dado comando, imprime o resultado e então libera a AST.

3.4 Recuperação de erro básica

A última regra no *parser* provê uma recuperação de erro simples. Da maneira como o Bison funciona, não é adequado tentar corrigir erros. Mas é ao menos possível recuperar o analisador para um estado onde o *parser* possa continuar com a análise. Ou seja, aplicar o modo de recuperação de erro conhecido também como *recuperação em modo pânico*.

O *token* especial `error` indica um ponto de recuperação de erro. Quando um *parser* Bison encontra um erro, o *parser* começa descartando símbolos da pilha do *parser* até que atinge um ponto onde um *token* `error` seria válido; então o *parser* descarta *tokens* da entrada até que encontre um que possa ser transferido (*shift*) para o seu estado corrente, e então a análise possa continuar desse ponto.

Se o *parser* falhar novamente, ele descarta mais símbolos da pilha e *tokens* da entrada até que seja possível retomar a análise sintática ou que a pilha esteja vazia e a análise tenha falhado.

A macro `yyerrork` utilizada na ação correspondente avisa o *parser* que a recuperação de erro está feita, tal que mensagens de erro subsequentes sejam produzidas.

Apesar ser possível adicionar diversas regras de erro para tentar diversas recuperações de erro, na prática é raro ter mais que uma ou duas regras de erro. O *token* `error` é quase sempre usado para resincronizar a análise numa regra recursiva no nível superior das regras da gramática, como é feito no exemplo da calculadora descrito aqui.

3.5 Calculadora: o analisador léxico

O analisador léxico visto aqui (Código 5) adiciona algumas novas regras em relação ao exemplo visto previamente. Existem alguns novos operadores de caracter único. Todos seis operadores de comparação retornam um novo *token* `CMP` com um valor léxico para distinguir entre os operadores.

```
1 /*
  * Lexer para uma calculadora avancada
3 */

5 /* reconhecimento de tokens para a calculadora */
```

```
%option noyywrap nodefault yylineno
%{
# include "bison-calc.h"
# include "bison-calc.tab.h"
%}

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

%%
"+" |   /* operadores de caracter unico */
"-" |
"*" |
"/" |
"=" |
"," |
";" |
"(" |
")"    { return yytext[0]; }

">" { 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; }

"if"  { return IF; }        /* palavras-chave */
"then"  { return THEN; }
"else"  { return ELSE; }
"while" { return WHILE; }
"do"  { return DO; }
"let" { return LET; }

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

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

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

"//".*
[ \t]   /* ignora espaco em branco */

\\\n    { printf("c> "); }  /* ignora continuacao de linha */

\n    { return EOL; }

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

Listing 5: Calculadora com AST - *lexer*

As seis palavras-chave e quatro funções pré-definidas são reconhecidas para padrões literais. Note que essas regras devem preceder o padrão geral para casar um nome, tal que essas regras tenham preferência em relação ao padrão geral. O padrão para nomes busca o nome na tabela

de símbolos (através da função *lookup*) e retorna um ponteiro para o símbolo.

Uma nova linha (`EOL`) marca o final da entrada de uma string/cadeia. Já que uma função ou expressão pode ser muita longa para ser digitada em uma linha somente, permite-se a continuação de linhas. Um nova regra do analisador léxico casa com um *backslash* e uma nova linha e não retorna nenhuma ação para o *parser*, fazendo assim a continuação transparente para o *parser*. Mas, para o usuário é impresso um *prompt*.

3.6 Palavras reservadas

Nessa gramática, as palavras **if**, **then**, **else**, **while**, **do**, **let**, **sqrt**, **exp**, **log** e **print** são reservadas não podem ser usadas como símbolos criados pelo usuário.

A questão de permitir que usuários utilizem o mesmo nome para dois elementos no mesmo programa é algo a ser debatido. Por um lado, se a gramática permitir o uso do mesmo, teremos programas não tão legíveis. Por outro lado, o uso de palavras reservadas força o usuário a criar nomes que não conflitem com aquelas palavras reservadas.

4 Construíndo e Interpretando AST

Aqui é descrito o código auxiliar escrito em C. Parte desse código é similar ao exemplo visto na **Apostila 03**, as rotinas `main` e `yyerror` são as mesmas.

A finalidade principal desse código (Código 6) é construir e avaliar as AST's. Inicialmente é descrito o gerenciamento da tabela de símbolos, o qual é familiar ao exemplo visto na **Apostila 01 - Flex, Arquivos e TS**.

```
/*
 * Funcoes Auxiliares para uma calculadora avancada
 */
# include <stdio.h>
# include <stdlib.h>
# include <stdarg.h>
# include <string.h>
# include <math.h>
# include "bison-calc.h"

/* funcoes em C para TS */
/* funcao 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 = &symtab[symhash(sym)%NHASH];
  int scount = NHASH;

  while(--scount >= 0) {
    if (sp->name && !strcasecmp(sp->name, sym))
      return sp;

```

```
    if (!sp->name) {  /* nova entrada na TS */
34    sp->name = strdup(sym);
      sp->value = 0;
36    sp->func = NULL;
      sp->syms = NULL;
38    return sp;
    }
40
    if (++sp >= symtab+NHASH)
42    sp = symtab;  /* tenta a prox. entrada */
  }
44 yyerror("overflow na tab. simbolos\n");
  abort(); /* tabela estah cheia */
46}

48
struct ast * newast(int nodetype, struct ast *l, struct ast *r)
50{
  struct ast *a = malloc(sizeof(struct ast));
52
  if (!a) {
54   yyerror("sem espaco");
     exit(0);
56 }
  a->nodetype = nodetype;
58 a->l = l;
  a->r = r;
60 return a;
}
62
struct ast * newnum(double d)
64{
  struct numval *a = malloc(sizeof(struct numval));
66
  if (!a) {
68   yyerror("sem espaco");
     exit(0);
70 }
  a->nodetype = 'K';
72 a->number = d;
  return (struct ast *)a;
74}

76struct ast * newcmp(int cmptype, struct ast *l, struct ast *r)
{
78 struct ast *a = malloc(sizeof(struct ast));

80 if (!a) {
     yyerror("sem espaco");
82   exit(0);
  }
84 a->nodetype = '0' + cmptype;
  a->l = l;
86 a->r = r;
  return a;
88}

90struct ast * newfunc(int functype, struct ast *l)
{
92 struct fncall *a = malloc(sizeof(struct fncall));
```

```
94    if (!a) {
        yyerror("sem espaco");
96      exit(0);
      }
98    a->nodetype = 'F';
      a->l = l;
100   a->functype = functype;
      return (struct ast *)a;
102 }

104 struct ast * newcall(struct symbol *s, struct ast *l)
    {
106   struct ufncall *a = malloc(sizeof(struct ufncall));

108   if (!a) {
        yyerror("sem espaco");
110     exit(0);
      }
112   a->nodetype = 'C';
      a->l = l;
114   a->s = s;
      return (struct ast *)a;
116 }

118 struct ast * newref(struct symbol *s)
    {
120   struct symref *a = malloc(sizeof(struct symref));

122   if (!a) {
        yyerror("sem espaco");
124     exit(0);
      }
126   a->nodetype = 'N';
      a->s = s;
128   return (struct ast *)a;
    }
130
    struct ast * newasgn(struct symbol *s, struct ast *v)
132 {
      struct symasgn *a = malloc(sizeof(struct symasgn));
134
      if (!a) {
136     yyerror("sem espaco");
        exit(0);
138   }
      a->nodetype = '=';
140   a->s = s;
      a->v = v;
142   return (struct ast *)a;
    }
144
    struct ast * newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast
        *el)
146 {
      struct flow *a = malloc(sizeof(struct flow));
148
      if (!a) {
150     yyerror("sem espaco");
        exit(0);
```

```
152   }
      a->nodetype = nodetype;
154   a->cond = cond;
      a->tl = tl;
156   a->el = el;
      return (struct ast *)a;
158 }

160
    /* libera uma arvore de AST */
162
    void treefree(struct ast *a)
164 {
      switch(a->nodetype) {
166
        /* duas subarvores */
168   case '+':
      case '-':
170   case '*':
      case '/':
172   case '1': case '2': case '3': case '4': case '5': case '6':
      case 'L':
174     treefree(a->r);

176     /* uma subarvore */
      case 'C': case 'F':
178     treefree(a->l);

180     /* sem subarvore */
      case 'K': case 'N':
182     break;

184   case '=':
        free( ((struct symasgn *)a)->v);
186     break;

188     /* acima de 3 subarvores */
      case 'I': case 'W':
190     free( ((struct flow *)a)->cond);
        if( ((struct flow *)a)->tl) treefree( ((struct flow *)a)->tl);
192     if( ((struct flow *)a)->el) treefree( ((struct flow *)a)->el);
        break;
194
      default: printf("erro interno: free bad node %c \n", a->nodetype);
196   }

198   free(a); /* sempre libera o proprio no */
    }
200
    struct symlist * newsymlist(struct symbol *sym, struct symlist *next)
202 {
      struct symlist *sl = malloc(sizeof(struct symlist));
204
      if (!sl) {
206     yyerror("sem espaco");
        exit(0);
208   }
      sl->sym = sym;
210   sl->next = next;
      return sl;
```

```
212 }

214 /* libera uma lista de simbolos */
    void symlistfree(struct symlist *sl)
216 {
      struct symlist *nsl;
218
      while(sl) {
220     nsl = sl->next;
        free(sl);
222     sl = nsl;
      }
224 }

226
    /* etapa principal >> avaliacao de expressoes, comandos, funcoes, ... */
228
    static double callbuiltin(struct fncall *);
230 static double calluser(struct ufncall *);

232 double eval(struct ast *a)
    {
234   double v;

236   if (!a) {
        yyerror("erro interno, null eval");
238     return 0.0;
      }
240
      switch(a->nodetype) {
242           /* constante */
        case 'K': v = ((struct numval *)a)->number; break;
244
        /* referencia de nome */
246     case 'N': v = ((struct symref *)a)->s->value; break;

248     /* atribuicao */
        case '=': v = ((struct symasgn *)a)->s->value = eval(((struct symasgn *)a)->
            v); break;
250
        /* expressoes */
252     case '+': v = eval(a->l) + eval(a->r); break;
        case '-': v = eval(a->l) - eval(a->r); break;
254     case '*': v = eval(a->l) * eval(a->r); break;
        case '/': v = eval(a->l) / eval(a->r); break;
256
        /* comparacoes */
258     case '1': v = (eval(a->l) > eval(a->r))? 1 : 0; break;
        case '2': v = (eval(a->l) < eval(a->r))? 1 : 0; break;
260     case '3': v = (eval(a->l) != eval(a->r))? 1 : 0; break;
        case '4': v = (eval(a->l) == eval(a->r))? 1 : 0; break;
262     case '5': v = (eval(a->l) >= eval(a->r))? 1 : 0; break;
        case '6': v = (eval(a->l) <= eval(a->r))? 1 : 0; break;
264
        /* controle de fluxo */
266     /* gramatica permite expressoes vazias, entao devem ser verificadas */

268     /* if/then/else */
        case 'I':
270       if( eval( ((struct flow *)a)->cond) != 0) {  /* verifica condicao */
```

```
        if ( ((struct flow *)a)->tl) {    /* ramo verdadeiro */
272       v = eval( ((struct flow *)a)->tl);
        } else
274       v = 0.0;  /* valor default */
      } else {
276     if ( ((struct flow *)a)->el) {    /* ramo falso */
          v = eval( ((struct flow *)a)->el);
278     } else
          v = 0.0;  /* valor default */
      }
      break;
282
    /* while/do */
284 case 'W':
      v = 0.0;     /* valor default */
286
      if ( ((struct flow *)a)->tl) {    /* testa se lista de comandos nao eh
          vazia */
288     while( eval(((struct flow *)a)->cond) != 0) /* avalia a condicao */
          v = eval(((struct flow *)a)->tl);     /* avalia comandos */
290   }
      break;            /* valor do ultimo comando eh valor do while/do */
292
    /* lista de comandos */
294 case 'L': eval(a->l); v = eval(a->r); break;

296 case 'F': v = callbuiltin((struct fncall *)a); break;

298 case 'C': v = calluser((struct ufncall *)a); break;

300 default: printf("erro interno: bad node %c\n", a->nodetype);
    }
302 return v;
}
304
static double callbuiltin(struct fncall *f)
306 {
  enum bifs functype = f->functype;
308 double v = eval(f->l);

310 switch(functype) {
  case B_sqrt:
312   return sqrt(v);
  case B_exp:
314   return exp(v);
  case B_log:
316   return log(v);
  case B_print:
318   printf(" =%4.4g\n", v);
    return v;
320 default:
    yyerror("Funcao pre-definda %d desconhecida\n", functype);
322 return 0.0;
  }
324

}
326

328 /* funcao definida por usuario*/
```

```
330 void dodef(struct symbol *name, struct symlist *syms, struct ast *func)
    {
332   if (name->syms) symlistfree(name->syms);
      if (name->func) treefree(name->func);
334   name->syms = syms;
      name->func = func;
336 }

338 static double calluser(struct ufncall *f)
    {
340   struct symbol *fn = f->s;    /* nome da funcao */
      struct symlist *sl;      /* argumentos (originais) da funcao */
342   struct ast *args = f->l;     /* argumentos (usados) na funcao */
      double *oldval, *newval;     /* salvar valores de argumentos */
344   double v;
      int nargs;
346   int i;

348   if (!fn->func) {
        yyerror("chamada para funcao %s indefinida", fn->name);
350     return 0;
      }
352
      /* contar argumentos */
354   sl = fn->syms;
      for(nargs = 0; sl; sl = sl->next)
356     nargs++;

358   /* prepara  o para salvar argumentos */
      oldval = (double *)malloc(nargs * sizeof(double));
360   newval = (double *)malloc(nargs * sizeof(double));
      if (!oldval || !newval) {
362     yyerror("Sem espa o em %s", fn->name);
        return 0.0;
364   }

366   /* avaliacao de argumentos */
      for (i = 0; i < nargs; i++) {
368     if (!args) {
          yyerror("poucos argumentos na chamada da funcao %s", fn->name);
370       free(oldval);
          free(newval);
372       return 0.0;
        }
374
        if (args->nodetype == 'L') {     /* se eh uma lista de nos */
376       newval[i] = eval(args->l);
          args = args->r;
378     } else {       /* se eh o final da lista */
          newval[i] = eval(args);
380       args = NULL;
        }
382   }

384   /* salvar valores (originais) dos argumentos, atribuir novos valores  */
      sl = fn->syms;
386   for (i = 0; i < nargs; i++) {
        struct symbol *s = sl->sym;
388
        oldval[i] = s->value;
```

```
390       s->value = newval[i];
          sl = sl->next;
392     }

394     free(newval);

396     /* avaliacao da funcao */
        v = eval(fn->func);
398
        /* recolocar os valores (originais) da funcao */
400     sl = fn->syms;
        for (i = 0; i < nargs; i++) {
402       struct symbol *s = sl->sym;

404       s->value = oldval[i];
          sl = sl->next;
406     }

408     free(oldval);
        return v;
410 }

412 void yyerror(char *s, ...)
    {
414     va_list ap;
        va_start(ap, s);
416
        fprintf(stderr, "%d: error: ", yylineno);
418     vfprintf(stderr, s, ap);
        fprintf(stderr, "\n");
420 }

422 int main()
    {
424     printf("> ");
        return yyparse();
426 }
```

Listing 6: Calculadora com AST - funções

Após são definidos os procedimentos para construir os nós AST e a lista de símbolos. Esses procedimentos alocam um nó e então preenchem os respectivos campos do tipo dos nós. Ademais, o procedimento `treefree` recursivamente percorre uma AST e libera todos nós na árvore.

A parte principal da calculadora é a função `eval`, a qual avalia uma AST construída no *parser*. Seguindo a abordagem da linguagem C, comparações retornam 1 ou 0 dependendo se a comparação tem sucesso ou não, e testes no if/then/else e while/do tratam qualquer valor diferente de zero como verdadeiro. Para expressões é realizado o procedimento de busca em profundidade em árvores para computar valores.

Com uma AST realiza-se o seguinte para implementar if/then/else: avalia a condição AST para decidir qual ramificação escolher, e então avaliar a AST conforme o caminho escolhido. Para avaliar laços while/do tem-se um laço na função `eval` para avaliar a condição AST, então o corpo é avaliado, conforme a condição AST permanecer verdadeira. Qualquer AST que tem variáveis de referência, e que sejam alteradas, terão um novo valor cada vez que for avaliada.

4.1 Avaliação de funções (pré-definidas)

A parte mais complicada na calculadora é lidar com funções. Funções pré-definidas são construídas a partir da especificação do código relativo a cada função.

4.2 Funções definidas pelo usuário

Uma função definida pelo usuário consiste no **nome** da função, uma lista de argumentos e uma AST que representa o **corpo** da função. Quando a função é definida a lista de argumentos e a AST são salvas na entrada da tabela de símbolos, sendo que versões prévias são substituídas. Note que isso é feito pela função `dodef` (ver linha 330 do Código 6).

Exemplo: considerando que seja definida uma função para calcular o máximo de dois argumentos.

```
> let max(x, y) = if x >= y then x; else y; ;
> max(4+5, 6+7)
```

A função `max` tem dois argumentos, x e y. Quando a função é chamada, o analisador faz os seguintes passos:

1. Avaliar os argumentos **reais**, $4 + 5$ e $6 + 7$.
2. Salvar os valores **atuais** dos argumentos (x, y) e atribuir os valores dos argumentos reais aos **atuais**.
3. Avaliar o corpo da função, o qual agora irá usar os argumentos **reais** quando os argumentos x e y forem referenciados.
4. Recolocar os antigos valores dos argumentos x e y.
5. Retornar o valor do corpo da expressão.

O código para função definida pelo usuário deve contar (*a quantidade*) os argumentos, alocar dois vetores temporários para valores antigos e novos argumentos, e então os cinco passos descritos acima são executados.

4.3 Utilização da Calculadora

Conforme foi feito na versão anterior da calculadora, aqui igualmente recomenda-se a criação de um *Makefile* para facilitar o processo de compilação dos arquivos.

Observação: caso seja utilizado o `gcc` para compilação do programa, lembre-se de utilizar o parâmetro $-lm$ para que as funções da biblioteca **math.h** funcionem corretamente.

A seguir são apresentados alguns exemplos de utilização da calculadora. Note a utilização dos comandos condicionais, repetição e funções, além da definição de funções pelo usuário.

```
gleifer@Chomsky:~/Dropbox/UTFPR-PG/Disciplinas.2014-1/Compiladores/Aulas/Handouts/04-Bison-Parser-Full-Calc/code$ ./bison-calc
> (20 + 4) * 2
=    48
> 4 * 5 + 10
=    30
> 4 * (5 + 10)
=    60
>
```

Figura 2: Utilização da Calculadora – Exemplo 1

```
gleifer@Chomsky:~/Dropbox/UTFPR-PG/Disciplinas.2014-1/Compiladores/Aulas/Handouts/04-Bison-Parser-Ful
> if x >= 0 then x = x + 1
2: error: syntax error
> if x >= 0 then x = x + 1;
=    1
> x = 5
=    5
> if x >= 0 then x = x + 1;
=    6
> print(x)
 =    6
=    6
>
7: error: syntax error
> if x < 5 then x = x + 5; else x = z + x;
=    6
> z = 1
=    1
>
```

Figura 3: Utilização da Calculadora – Exemplo 2

```
gleifer@Chomsky:~/Dropbox/UTFPR-PG/Disciplinas.2014-1/Compiladores/Aulas/Handouts/04-Bison-Parser-Full-Calc/
> y = 10
=   10
> while y > 0 do y = y - 1; printf(y);
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
3: error: chamada para funcao printf indefinida
=    0
> y = 10
=   10
> while y > 0 do y = y - 1; print(y);
 =   9
 =   8
 =   7
 =   6
 =   5
 =   4
 =   3
 =   2
 =   1
 =   0
=    0
> while y <= 36 do y = print(exp(y));
 =   1
 =2.718
 =15.15
 =3.814e+06
= 3.814e+06
>
```

Figura 4: Utilização da Calculadora – Exemplo 3

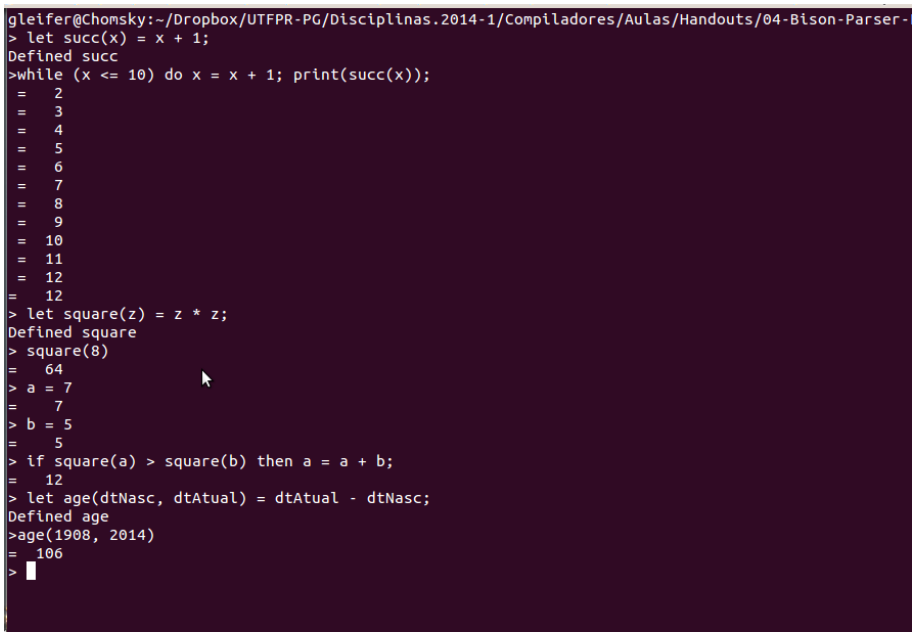

Figura 5: Utilização da Calculadora – Exemplo 4

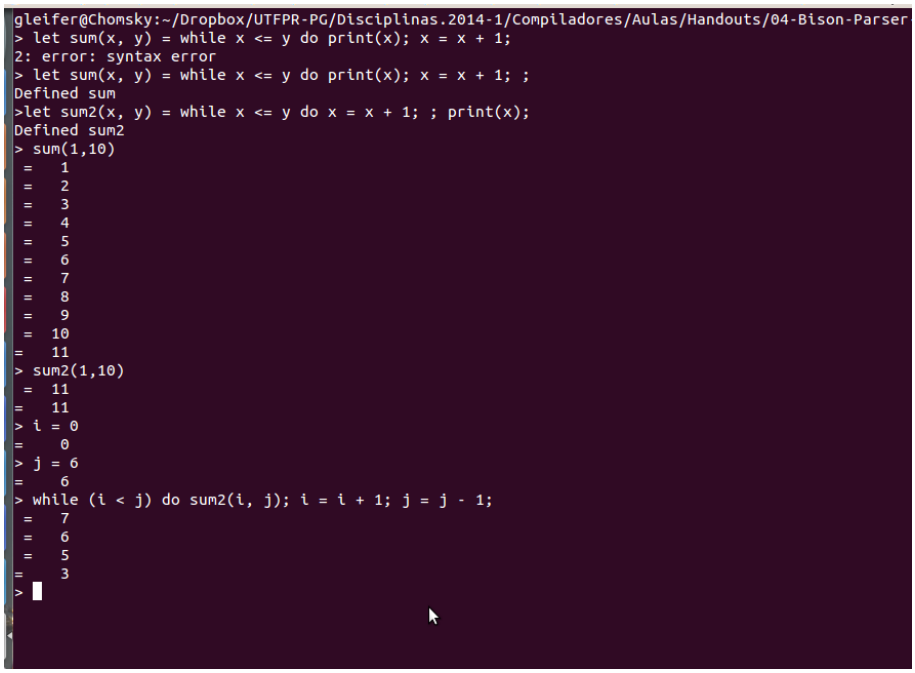

Figura 6: Utilização da Calculadora – Exemplo 5