Davi Melo Morales

Projeto de compilador para linguagem C-

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Compiladores.

Professora: Ana Carolina Lorena

São José dos Campos - SP
2017

Conteúdo

1 Introdução

1.1 Contextualização e Motivação

Este trabalho dá continuidade ao projeto de um sistema computacional, iniciado através da elaboração de um processador. A fim de que se projetem programas a serem executados nele, torna-se necessário que se desenvolva um compilador para uma linguagem de alto nível.

Tem-se como compilador um software que converte um conjunto de trechos de uma linguagem de alto nível para uma representação em nível mais baixo [4].

Essa função é fundamental para que se possa obter agilidade e para que se elaborem programas mais complexos através das linguagens de nível mais alto. Dentre as principais razões para que se estudem compiladores, enumeram-se[8]:

1. Parsers e interpretadores são extremamente populares: programadores competentes precisam compreender parsers e interpretadores porque, eventualmente, é necessário que pequenas interações destes sejam escritas: cada vez em que se desenvolve um programa extensível ou se lida com um novo tipo de arquivo de entrada, isso é realizado.

2. Melhor competência em escrever e corrigir códigos: supõe-se que um compilador traduza corretamente qualquer programa em sua linguagem de entrada. A fim de se atingir esta meta, desenvolvedores de compiladores precisam entender completamente a linguagem de entrada, inclusive casos excepcionais nunca vistos pela maioria dos programadores. Este entendimento é um passo importante na compreensão de uma linguagem como ela realmente é, e não somente como ela é normalmente escrita.

3. Aprende-se a programar mais rápido: ao se entender um compilador, ha uma ideia mais clara sobre as possíveis otimizações que estão dentro do escopo de um compilador.

Portanto, há motivações tanto acadêmicas quanto profissionais para a realização do presente projeto.

1.2 Objetivos

Projetar e desenvolver uma solução computacional em compiladores que permita a conversão da linguagem didática à linguagem de desenvolvimento do processador objeto elaborado durante a disciplina de Laboratório de Arquitetura e Organização de Computadores.

1.3 Ferramentas Utilizadas

A linguagem base utilizada para o desenvolvimento da etapa de análise do compilador foi C. Juntamente a ela, utilizaram-se ferramentas acessórias, sobre as quais se descorre nas seções a seguir. Para a análise léxica, utilizou-se o Flex 1.3.1, enquanto para a análise sintática, se fez uso do Bison 1.3.2

1.3.1 Flex

Flex consiste em uma ferramenta para se gerar programas de varredura, os quais reconhecem padrões léxicos. Ele lê determinados arquivos de entrada que descrevem

o programa de varredura a ser gerado. A descrição se encontra na forma de pares de expressões regulares e código C, denominados regras. Flex gera, então, um arquivo fonte em C como saída, o *lex.yy.c*, o qual defina uma rotina *yylex()*. Tal arquivo é compilado e e ligado à biblioteca *-lfl* a fim de produzir um executável. Quando o dado executável é executado, ele analisa sua entrada em busca de expressões regulares. Ao encontrar uma, ele executa o código C correspondente [1].

1.3.2 Bison

Bison é um gerador de parser de propósito geral que converte uma gramática livre de contexto para um parser LR determinístico ou LR geral empregando tabelas parser LALR(1). Bison pode ser utilizado para desenvolver uma ampla gama de parsers para linguagens, desde os utilizados em simples calculadoras até linguagens de programação complexas.

Ele tem compatibilidade total com Yacc [2], de maneira que quaisquer gramáticas escritas em Yacc devem funcionar sobre o Bison sem necessidade de mudanças [3].

1.4 Compilador: Fase de Análise

O primeiro estágio de um compilador é a Fase de Análise. Essa fase é chamada de fase de varredura, onde o compilador varre o código fonte, caracter por caracter, e os agrupa em *tokens*. Cada *token* representa uma sequência coerente de caracteres, como variáveis. Essa fase se divide em 3 estágios [9]:

1.4.1 Análise Léxica

Sendo a primeira das três fases, a análise léxica recebe código fonte modificado por pré-processadores de linguagem, que o organiza em forma de sentenças. O analisador léxico quebra os lexemas em *tokens*, removendo comentários e espaços em branco. Caso haja um *token* inválido, um erro é gerado[7].

1.4.2 Análise Sintática

A análise sintática visa a validação da gramática do programa. Nela, verifica-se se as regras sintáticas da linguagem são obedecidas [6].

Varrem-se os *tokens* em sequência recebidos do analisador léxico e se produz uma árvore sintática para representar a hierarquia do programa fonte. Um erro deve ser gerado caso seja reconhecida uma construção indevida.

1.4.3 Análise Semântica

Para que um programa seja semanticamente válido, suas variáveis, funções e demais itens precisam ser definidos apropriadamente; expressões e variáveis devem ser utilizadas de modo a respeitarem o sistema de tipos, controle de acesso, e assim por diante.

Boa parte da análise semântica consiste no rastreamento de declarações de variáveis, funções ou tipos, e checagem de tipos. Em diversas linguagens, por exemplo, identificadores devem ser declarados antes de seu uso. Ao encontrar uma nova declaração, um compilador deve guardar o tipo da informação atribuído ao dado identificador. Então, examinando o restante do programa, ele verifica se o tipo do identificador está sendo respeitado quanto às operações que se realizam [10].

2 Processador

A arquitetura projetada é RISC e se baseia em uma variação monocíclica da arquitetura MIPS, apresentando modificações inspiradas em conceitos presentes no livro *Logic and Computer Design Fundamentals*[5].

2.1 Conjunto de Instruções

Projetou-se um conjunto de instruções específico, o qual serve como fundamento para o desenvolvimento da presente arquitetura. Cada instrução se divide em seções que especificam os recursos a serem utilizados na operação. Tais recursos são explicados a seguir:

- DR: endereço de 5 bits que se refere ao registrador de destino da operação dentre os 32 presentes no banco de registradores;
- SA, SB: endereços de 5 bits que especificam os registradores de origem;
- IM: refere-se a um imediato de tamanho variável que é passado de maneira direta;
- Alavancas: refere-se às chaves de entrada presentes no kit FPGA;
- *Displays*: se refere aos oito *displays* de sete segmentos presentes no kit FPGA;
- PC: refere-se ao *Program Counter*;
- Shamt: quantidade de *bits* a serem deslocados;
- FZ e FN: são dois *flipflops* que guardam sinais checados para a realização de certas operações.

O *opcode* informa à unidade de controle a operação a ser realizada. Para isso, existe a necessidade do mapeamento das instruções em código binário. A Tabela 1 lista o conjunto de instruções executáveis pela arquitetura proposta e apresenta, para cada uma delas, seu mapeamento, mnemônico e a operação que realiza:

Instrução	Opcode	Mnemônico	Operação
Adição	000000	ADD	$R[DR] \leftarrow R[SA] + R[SB]$
Adição Imediato	000001	ADDI	$R[DR] \leftarrow R[SA] + IM$
Subtração	000010	SUB	$R[DR] \leftarrow R[SA] + \overline{R[SB]} + 1$
Subtração Imediato	000011	SUBI	$R[DR] \leftarrow R[SA] + \overline{IM} + 1$
Multiplicação	000100	MUL	$R[DR] \leftarrow R[SA] * R[SB]$
Divisão	000101	DIV	$R[DR] \leftarrow R[SA]/R[SB]$
Incrementa	000110	INC	$R[DR] \leftarrow R[SA] + 1$
Decrementa	000111	DEC	$R[DR] \leftarrow R[SA] - 1$
And	001000	AND	$R[DR] \leftarrow R[SA] \wedge R[SB]$
Or	001001	OR	$R[DR] \leftarrow R[SA] \vee R[SB]$
Resto	001010	MOD	$R[DR] \leftarrow R[SA]\%R[SB]$
Xor	001100	XOR	$R[DR] \leftarrow R[SA] \bigoplus R[SB]$
Not	001101	NOT	$R[DR] \leftarrow \overline{R[SA]}$
Desloca Esquerda	010000	SHL	$R[DR] \leftarrow sl(shamt)R[SA]$
Desloca Direita	010001	SHR	$R[DR] \leftarrow sr(shamt)R[SA]$
Pré-branch	011111	PBC	se $R[SA] = 0, FZ = 1$; $seR[SA] < 0, FN = 1$;
Branch em Zero	010011	BOZ	se $FZ = 1$, então $PC \leftarrow PC + 1 + IM$ se $FZ = 0$, então $PC \leftarrow PC + 1$
Branch em Negativo	010100	BON	se $FN = 1$, então $PC \leftarrow PC + IM$ se $FN = 0$, então $PC \leftarrow PC + 1$
Jump	010101	JMP	$PC \leftarrow IM$
Set on Less Than	010111	SLT	$seR[SA] < R[SB]$, então $R[DR] = 1$
Load	011000	LD	$R[DR] \leftarrow M[IM]$
Store	011001	ST	$M[IM] \leftarrow R[SA]$
Load Imediato	011010	LDI	$R[DR] \leftarrow IM$
Nop	011011	NOP	Sem Operação
HLT	011100	HLT	Parar Operação
Entrada	011101	IN	$R[DR] \leftarrow alavancas$
Saída	011110	OUT	$Displays \leftarrow R[SA]$

Tabela 1: Mapeamento e especificações referentes ao conjunto de instruções

2.1.1 Formatos das Instruções

Projetaram-se quatro formatos distintos para as instruções com o objetivo de transmitir as informações necessárias de maneira simples e eficiente. Por se tratar de uma arquitetura RISC, todas as instruções apresentam o mesmo tamanho, 32 *bits*:

- Formato de três registradores: engloba a maioria das instruções lógicas e aritméticas - tais como ADD, SUB, AND, OR, XOR, SLT -, sendo que dois registradores fonte - SA e SB - fornecem os operandos e o resultado é gravado no registrador destino.

31 - 26	25 - 21	20 - 16	15 - 11	10 - 0
opcode	SA	SB	DR	—

- Formato de dois registradores: apresenta as instruções aritméticas ADDI e SUBI, de modo que os operandos estão contidos no registrador fonte e no imediato. As funções INC e DEC também são descritas por esse formato; nelas o último campo da instrução torna-se desnecessário. As instruções NOT, SL e SR também aparecem nesse formato, de modo que o registrador fonte é negado, deslocado à esquerda ou à direita, respectivamente; o campo final da instrução, no caso das duas últimas,

indica quantos *bits* o operando deve ser deslocado. Todas gravam informações no registrador destino (DR).

31 - 26	25 - 21	20 - 16	15 - 0
opcode	SA	DR	imediato/endereço/shamt

- Formato de um registrador fonte: contém as instruções PBC, LDI, IN e OUT. No caso de PBC, o valor contido no registrador fonte é comparado a uma referência e, a depender desse resultado, FZ ou FN são *setados*; essa instrução apresenta a exceção de que o endereço do registrador é informado das posições [20..16] da instrução. Em LDI, o valor contido no imediato é carregado para registrador destino. Para a instrução IN, os valores informados através dos dispositivos de entrada (alavancas) tomarão o espaço do campo final da instrução e esse valor será carregado para o registrador destino. Para a função de saída, o valor contido no registrador é encaminhado para a saída de dados.

31 - 26	25 - 21	20 - 0
opcode	SA/DR	imediato/entrada/saída

- Formato sem registradores: utilizado pelas instruções JMP, BOZ, BON, NOP e HLT. A instrução JMP confere o valor do imediato ao *PC*. BOZ e BON somam o valor informado no imediato ao PC seguinte. NOP e HLT não necessitam desse valor, de modo que informam apenas a não realização ou parada de operação.

31 - 26	26 - 0
opcode	imediato/–

2.1.2 Modos de Endereçamento

Como pode-se observar nas estruturas das instruções, os seguintes modos de endereçamento foram adotados:

- Imediato: rápido, porém de alcance limitado, esse modo de endereçamento pode ser visto em instruções como SUBI e ADDI, nas quais o valor informado no imediato torna-se um operando.
- Direto: com equilíbrio entre eficiência e alcance, o modo direto está presente, também, nessa arquitetura. As instruções LD e ST apresentam esse modo, uma vez que buscam valores na memória utilizando endereços guardados em instruções.
- Por registrador: utilizado para acessar operandos guardados em registradores, esse modo é utilizado por muitas instruções, tais como: ADD, SUB, OR, XOR entre outras.

Os modos imediato e direto foram adotados por permitirem operações que utilizam os recursos de maneira eficiente em situações onde seja possível um alcance limitado.

Escolheu-se o modo por registrador por ele ser ideal para determinados tipos de operações, como as operações lógicas e aritméticas, que precisam ser realizadas relativamente rápido mas necessitam, também, de operandos de tamanho razoável.

2.2 Esboço da Arquitetura do Processador

À semelhança da arquitetura MIPS, a presente arquitetura contém memória - neste caso subdividida entre memória de dados e memória de instruções -, um banco de registradores, uma unidade lógica e aritmética (ULA) e o PC.

A memória de instruções tem por função receber a instrução do PC e distribuí-la para seus diversos propósitos. O banco de registradores funciona como uma memória de curto prazo e é dotado de 32 registradores de 32 *bits* cada. A ULA tem por finalidade realizar as operações lógicas e aritméticas, enquanto a memória de dados armazena e carrega dados.

Para o caso de haverem operações entre um dado proveniente de um registrador e outro proveniente de um imediato - seja ele de 16 ou 21 *bits* -, há um extensor que emparelha a quantidade de *bits* do imediato em relação ao registrador.

O projeto da arquitetura foi realizado com o objetivo de se adequar às instruções definidas. Portanto, assim como no caso do conjunto de instruções, a arquitetura se baseou no MIPS.

Os dispositivos de entrada e saída também foram levados em consideração na elaboração dessa arquitetura, de maneira que dados tanto deverão poder ser inseridos no sistema através dos periféricos de entrada, quanto haverá retorno para o usuário através do dispositivo de saída.

Estes foram adaptados ao projeto dessa arquitetura, assim como a presença de imediatos de tamanhos distintos e modificações realizadas em virtude de funções como BOZ e BON, onde seu resultado funciona como seletor, de maneira que o valor do imediato é somado ou não ao endereço seguinte para o PC. A Figura 1 o esboço inicial simplificado para a arquitetura elaborada.

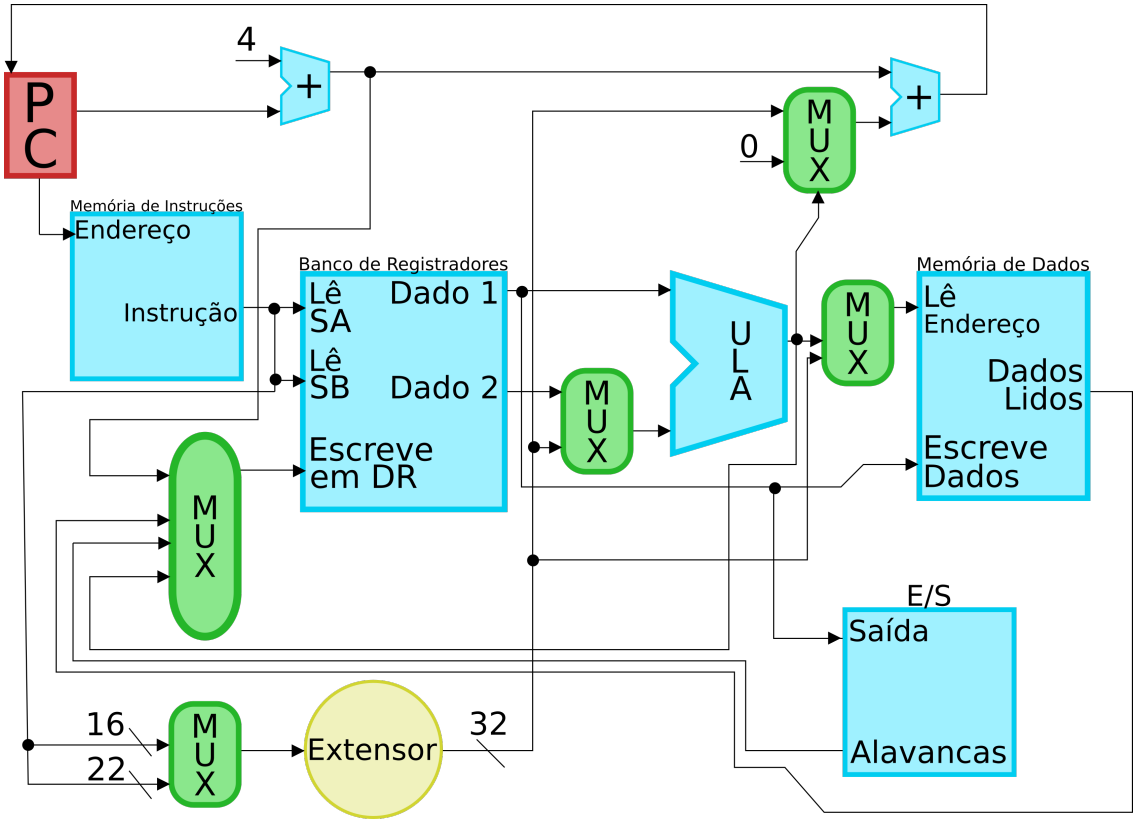

Figura 1: Esquemático do sistema computacional projetado.

3 O compilador: fase de análise

Até a presente etapa do projeto, apenas a fase de análise do compilador foi implementada. O compilador desenvolvido analisa códigos em C menos, gera uma tabela de literais na análise léxica, uma tabela de símbolos e uma árvore sintática na análise sintática. Efetua, também, a checagem de erros de natureza léxica, sintática ou semântica no código.

3.1 Análise Léxica

A análise léxica foi implementada com o uso da ferramenta *flex* 1.3.1. Nela, procura-se gerar uma tabela de literais a partir do código analisado. Os possíveis literais definidos para esta implementação se apresentam na Tabela 2.

Entrada	**Token**
int	INT
float	FLOAT
if	IF
else	ELSE
return	RETURN
void	VOID
while	WHILE
+	PLUS
−	MINUS
*	TIMES
<	LT
/	OVER
<=	LET
>	HT
>=	HET
==	EQ
! =	NEQ
=	ASSIGN
;	SEMI
,	COMMA
(	LPAREN
)	RPAREN
[	LBRACK
]	RBRACK
{	LCAPSULE
}	RCAPSULE

Tabela 2: Mapeamento e especificações referentes ao conjunto de instruções

Nessa etapa, os espaços em branco e comentários são removidos, os quais começam com '/*' e terminam com '*/'. A linguagem regular adotada se apresenta no código abaixo.

```
digit       [0-9]
numberi     {digit}+
numberf     {digit}+?\.{digit}+|-?{digit}+\.{digit}+?
letter      [a-zA-Z]
```

```
identifier   {letter}+{letter}*
newline      \n
whitespace   [ \t\r]+
other    [^0-9a-zA-Z;/=\-"+""*""("")""\n"\[\]\,\{\}\<\>|!=\==\<=\>=]
```

Nela, consideram-se digitos de 1 a 9. Números são considerados, tanto inteiros quanto números no formato *float*. As letras são consideradas, assim como identificadores. Identificam-se novas linhas e espaços, além de um conjunto de símbolos diversos.

Nessa varredura, os identificadores foram guardados em uma tabela hash de listas dinâmicas. A estrutura guarda o escopo, linhas em que o identificador aparece, nome, tipo e se é função ou não.

O código gerado pelo Flex referente a esta etapa se encontra na Seção 1 do apêndice.

3.2 Análise Sintática

O parser elaborado nesta fase foi realizado através da ferramenta *Bison* 1.3.2. Geram-se nela a Tabela de símbolos (Figura 2) e a árvore de análise sintática (Figura 3.

Name(ID)	Type(ID)	Type(Data)	Scope	Appears in lines
output	func	void	none	36
input	func	int	none	34,35
i	var	int	main	30,44
main	func	void	none	22
u	var	int	gdc	4,7,8
u	var	int	main	28,39,43,44
v	var	int	gdc	4,7,8
v	vet	int	main	29,44
x	var	int	output	17
x	var	int	main	26,34,36,40,41,45
y	var	int	main	27,35,36
gdc	func	int	none	4,8,36

Figura 2: Exemplo de tabela de símbolos.

Conforme a linguagem C menos, as seguintes construções são permitidas para este trabalho:

1. Variáveis e funções devem ser declaradas antes de serem usadas;
2. Tipos possíveis de variáveis: int e float;
3. Tipos possíveis de funções: void, int e float;
4. Variáveis podem ser declaradas como vetores da forma nome[tamanho];
5. O laço de repetição permitido é o *while*;
6. As estruturas condicionais permitidas são *if* e *else*;
7. Exceto na declaração de funções, laços de repetição ou estruturas condicionais, todas as linhas devem terminar com ';';
8. Laços de repetição, condicionais e funções devem apresentar chaves no começo e em seu fim da seguinte forma: est(){ ... }.

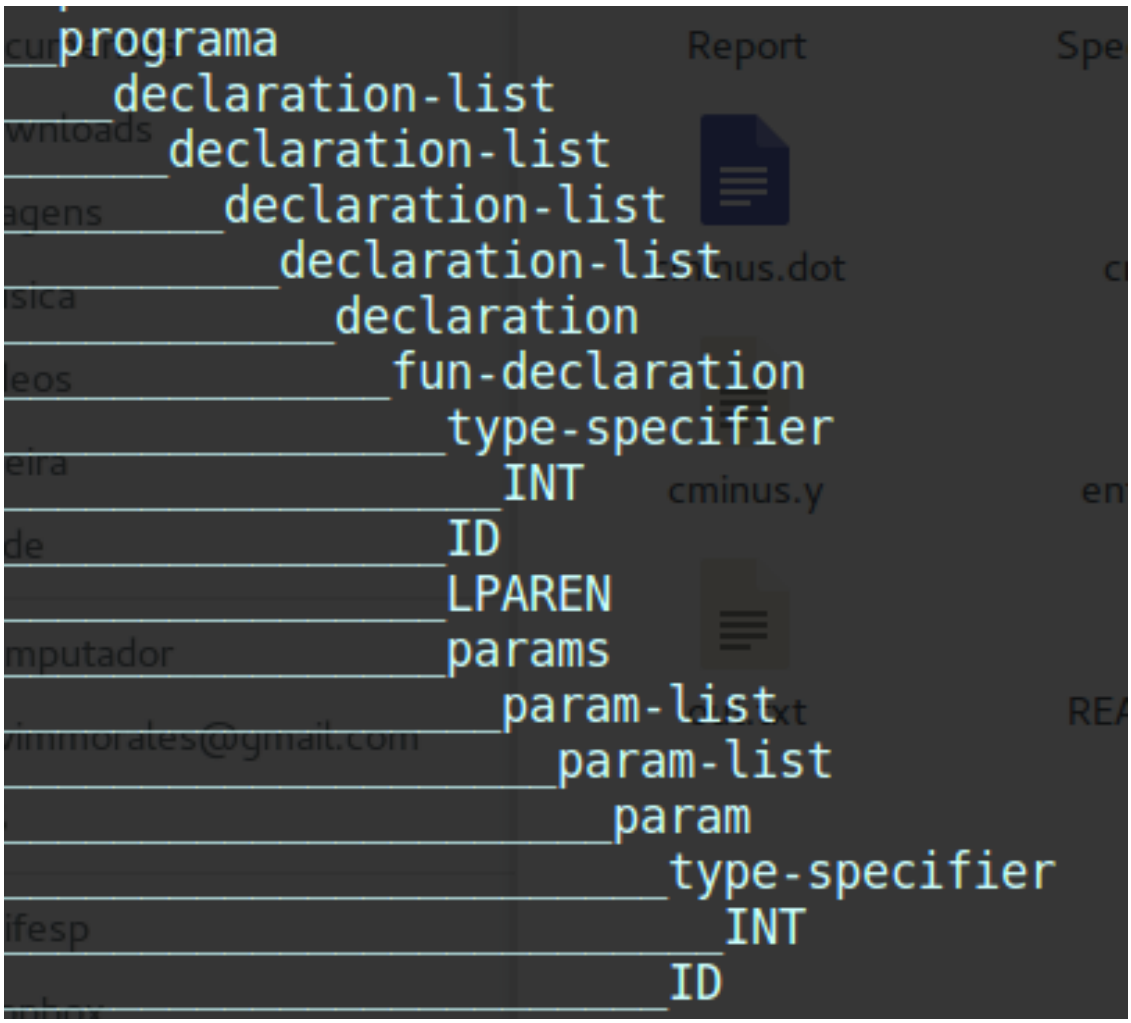

Figura 3: Exemplo de saída de árvore sintática - níveis representado através da identação.

A gramática BNF considerada se apresenta no Código 4.

A estrutura adotada nessa etapa foi a Árvore Genérica, onde se guarda a linha de aparição e item da gramática, assim como um nó filho e um nó irmão.

Hierarquicamente, portanto, as informações são guardadas dentro dessa árvore sintática.

O código referente aos desenvolvimentos dessa fase se encontram na Seção 2 do Apêndice.

3.3 Análise Semântica

A análise semântica elaborada verifica as seguintes condições:

1. Se as variáveis foram declaradas antes do uso;
2. Se há variáveis recebendo retorno de funções void;
3. Se há variáveis declaradas como void;
4. Se há declarações duplas num mesmo escopo;
5. Se as funções foram definidas antes do uso;
6. Se existe a função 'main';
7. Possíveis duplicações entre nomes de funções e variáveis.

programa → declaração-lista
declaração-lista → declaração-lista declaração | declaração
declaração → var-declaração | fun-declaração
var-declaração → tipo-especificador ID ; | tipo-especificador ID [NUM] ;
tipo-especificador → int | void
fun-declaração → tipo-especificador ID (params) composto-decl
params → param-lista | void
param-lista → param-lista, param | param
param → tipo-especificador ID | tipo-especificador ID [ ]
composto-decl → { local-declarações statement-lista } | {local-declarações} | {statement-lista} | { }
local-declarações → local-declarações var-declaração | var-declaração
statement-lista → statement-lista statement | statement
statement → expressão-decl | composto-decl | seleção-decl | iteração-decl | retorno-decl
expressão-decl → expressão ; | ;
seleção-decl → if (expressão) statement | if (expressão) statement else statement
iteração-decl → while (expressão) statement
retorno-decl → return ; | return expressão;
expressão → var = expressão | simples-expressão
var → ID | ID [expressão]
simples-expressão → soma-expressão relacional soma-expressão | soma-expressão
relacional → <= | < | > | >= | == | !=
soma-expressão → soma-expressão soma termo | termo
soma → + | -
termo → termo mult fator | fator
mult → * | /
fator → (expressão) | var | ativação | NUM
ativação → ID (arg-lista) | ID ( )
arg-lista → arg-lista , expressão | expressão

Figura 4: Gramática BNF para a linguagem C-

Essas verificações são feitas fazendo-se uso da tabela de símbolos, ao longo da própria varredura de análise léxica e, também, em uma pós varredura. O código se encontra na Seção 1 do Apêndice.

As seguintes verificações são feitas através da tabela de símbolos:

- Existência de variáveis void:

```
int checkVoid(TipoLista *list, int index){
TipoID *p = list[index].start;
//Laco que percorre todas as posicoes da tabela hash
while(p!=NULL){
    if(p->linhas[0] != 0) {
      //Aqui se comparam os valores. Caso haja uma variavel void, o
          erro e apresentado.
      if (!strcmp(p->tipoID, "var")&&!strcmp(p->tipoData,"void")) {
        return p->linhas[0];
      }
    }
      p = p->prox;
    }
    return 0;
}
```

- Existência da função main:

```
int checkMain(TipoLista *list, int index){
TipoID *p = list[index].start;
//Laco que percorre todas as posicoes da tabela hash
while(p!=NULL){
    if(p->linhas[0] != 0) {
      //Aqui se comparam os valores. Caso nao haja uma funcao void de
          nome main, retornara erro.
      if (!strcmp(p->tipoID, "func")&&
          !strcmp(p->tipoData,"void")&&
          !strcmp(p->nomeID, "main")&&
          p->linhas[1]==0)
        return 0;
    }
      p = p->prox;
    }
    return 1;
}
```

- Duas variáveis declaradas em um mesmo escopo:

```
int checkDecScope(TipoLista *list, int index){
  TipoID *p = list[index].start;
  TipoID *w;
//Laco que percorre todas as posicoes da tabela hash. Procuram-se
   apenas variaveis ou vetores
  while(p!=NULL&&(!strcmp(p->tipoID,"var")||!strcmp(p->tipoID,"vet"))
     ){
    if(p->linhas[0] != 0) {
      w = p->prox;
      //Segundo laco que percorre tabela para que haja comparacao.
      while (w!=NULL&&(!strcmp(p->tipoID,"var")||!strcmp(p->tipoID,"
          vet"))){
      //Aqui ocorrem as comparacoes entre as variaveis para que nao
          haja dupicadas.
        if(!strcmp(w->nomeID,p->nomeID)&&!strcmp(p->escopo,w->escopo)
           ){
          return w->linhas[0];
        }
        w = w->prox;
      }
    }
      p = p->prox;
    }
    return 0;
  }
```

De forma análoga às apresentadas anteriormente, se verifica também se há variáveis e funções com nomes iguais.

As verificações relacionadas a chamadas de funções ou variáveis não declaradas são feitas ao longo da primeira varredura e atribuição de *tokens*, onde quando uma variável ou função é chamada, checa-se se já houve declaração.

Por fim, a verificação quanto a atribuições de variáveis *void* a identificadores inteiros é feita em uma última varredura, onde se separam os dois lados da declaração e se verifica se a atribuição está sendo feita de forma válida.

4 Conclusões

O desenvolvimento desta ferramenta trouxe aprofundamento e aplicação aos conhecimentos adquiridos de linguagens, permitindo que se entendesse como funciona a parte de análise da construção de um compilador. Nota-se, portanto, a importância da aplicação prática para o desenvolvimento acadêmico e pessoal do aluno.

Ainda há detalhes de implementação a serem corrigidos a fim de que se facilitem os próximos passos de geração de código intermediário e de síntese, necessários para que esse projeto cumpra seu propósito de permitir a programação em alto nível para o processador desenvolvido.

Referências

[1] D. Compiletools. Flex. `http://dinosaur.compilertools.net/flex/manpage.html`. Acesso em 24-03-2017.

[2] D. Compiletools. Yacc. `http://dinosaur.compilertools.net/yacc/`. Acesso em 24-03-2017.

[3] GNU. Bison. `https://www.gnu.org/software/bison/`. Acesso em 24-03-2017.

[4] P. Magazine. Compiler definition. `http://www.pcmag.com/encyclopedia/term/40105/compiler`. Acesso em 24-03-2017.

[5] K. C. Mano Morris. *Arquitetura e organização de computadores: projeto para o desempenho*. PEARSON, 4 edition, 2008.

[6] J. D. Marangon. Compiladores para humanos - a estrutura de um compilador. `https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/structure-of-a-compiler.html`. Acesso em 24-03-2017.

[7] T. Point. Compiler design - lexical analysis. `https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm`. Acesso em 24-03-2017.

[8] J. Regehr. Why take a compilers course? `http://blog.regehr.org/archives/169`. Acesso em 24-03-2017.

[9] M. Tekwani. Phases of the compiler systems. `https://pt.slideshare.net/mukeshnt/phases-of-the-compilersystems`. Acesso em 24-03-2017.

[10] S. University. Semantic analysis. `https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/180%20Semantic%20Analysis.pdf`. Acesso em 24-03-2017.

5 Apêndice

A seguir estão os códigos fontes escritos para este projeto.

Algoritmo 1: cminus.l - Expressões Regulares

```
/****************************************************/
/* File: cminus.l                                   */
/* Lex specification for C- and Table Generator     */
/* Trabalho Pratico de Compiladores                 */
/* Davi Morales and Mateus Franco                   */
/****************************************************/
/*there was, formerly, an optional minus on numberi and numberf*/
%{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "cminus.tab.h"
extern YYSTYPE yylval;


void abrirArq();

void printTree();

int lineno = 1;

%}

digit        [0-9]
numberi      {digit}+
numberf      {digit}+?\.{digit}+|-?{digit}+\.{digit}+?
letter       [a-zA-Z]
identifier   {letter}+{letter}*
newline      \n
whitespace   [ \t\r]+
other    [^0-9a-zA-Z;/=\-"+""*""(")""\n"\[\]\,\{\}\<\>|!=\==\<=\>=]
%option yylineno
%%

"int"                 {/*printf("INT ");*/ return INT;}
"float"             {/*printf("FLOAT ");*/ return FLOAT;}
"if"          {/*printf("IF ");*/ return IF;}
"else"              {/*printf("ELSE ");*/ return ELSE;}
"return"            {/*printf("RETURN ");*/ return RETURN;}
"void"              {/*printf("VOID ");*/return VOID;}
"while"             {/*printf("WHILE ");*/ return WHILE;}
"+"                           {/*printf("PLUS ");*/ return PLUS;}
"-"                           {/*printf("MINUS ");*/ return MINUS;}
"*"                           {/*printf("TIMES ");*/ return TIMES;}
"<"                           {/*printf("LT ");*/ return LT;}
"/"                           {/*printf("OVER ");*/ return OVER;}
"<="          {/*printf("LET ");*/ return LET;}
">"                           {/*printf("HT ");*/ return HT;}
">="          {/*printf("HET ");*/ return HET;}
"=="          {/*printf("EQ ");*/ return EQ;}
"!="          {/*printf("NEQ ");*/ return NEQ;}
"="                           {/*printf("ASSIGN ");*/ return ASSIGN;}
```

```
";"                             {/*printf("SEMI ");*/ return SEMI;}
","                             {/*printf("COMMA ");*/ return COMMA;}
")"           {/*printf("RPAREN ");*/ return RPAREN;}
"("                             {/*printf("LPAREN ");*/ return LPAREN;}
"]"                             {/*printf("RBRACK ");*/ return RBRACK;}
"["                             {/*printf("LBRACK ");*/ return LBRACK;}
"{"                             {/*printf("LCAPSULE ");*/ return
   LCAPSULE;}
"}"                             {/*printf("RCAPSULE ");*/ return
   RCAPSULE;}
"/*"            {       char c, d;
                        c = input();
                        do
                        {       if (c == EOF) break;
                                d = input();
                                if (c == '\n')
                                        lineno++;
                                if (c == '*' && d == '/') break;
                                c = d;
                    } while (1);
                  }
                {newline}       { lineno++; /*printf("\t%d\n", lineno);
                   */}
                {whitespace}    ;
                {numberi}       { /*printf("NUMI "); strcpy(id,yytext);
                   */ return NUMI;}
                {numberf}               {/*printf("NUMF "); strcpy(id,
                   yytext);*/ return NUMF;}
                {identifier}    {/*printf("ID "); strcpy(id,yytext);*/
                   return ID;}
<<EOF>>           return(0);
{other}           { printf("Lexical Error at line %d\n", lineno);}

%%

char L_side[20];
char R_side[20];
int verState = 0;
int voidFlag = 0;

typedef struct TipoID{
    char nomeID[20];
    char tipoID[20];
    char tipoData[10];
    char escopo[30];
    int linhas[50];
    int top;
    struct TipoID *prox;
}TipoID;

typedef struct{
    TipoID *start;
}TipoLista;

void inicializaLista(TipoLista *lista){
    lista->start = NULL;
}

int contaChar(const char *str)
{
```

```
    int i = 0;
    for(;str[i] != 0; ++i);
    return i;
}

int string2int(const char *num)
{
  int  i, len, a;
  int result=0;
  len = contaChar(num);
  for(i=0; i<len; i++){
    result = result * 10 + ( num[i] - '0' );
  }
  return result;
}

void insere(TipoLista *lista, char scope[], char nameID[], char
   typeID[], char typeData[], int nline, int index)
{


    // Alocacao do no que sera indexado
    TipoID *novoNo = malloc(sizeof(TipoID));
    // Inicializacao do vetor de linhas
    int i;
    for(i=0;i<50;i++) {
      novoNo->linhas[i] = 0;
    }
    // Atribuicao da linha na posicao inicial

    if(strcmp(nameID,"input") == 0 || strcmp(nameID,"output") == 0) {
      novoNo->linhas[0] = 0;
      novoNo->top = 0;
    } else {
      novoNo->linhas[0] = nline;
      novoNo->top = 1; // proxima posicao de inserir numero da linha
    }

    // Inicializacao dos demais campos do no com os parametros de
       entrada
    if(!strcmp(typeID,"func")) {
      strcpy(novoNo->escopo, "none");
      /*strcpy(novoNo->tipoData, "none");*/
    } else {
      strcpy(novoNo->escopo, scope);
      /*strcpy(novoNo->tipoData, typeData);*/
    }
    strcpy(novoNo->tipoData, typeData);
    strcpy(novoNo->nomeID, nameID);
    strcpy(novoNo->tipoID, typeID);
    /*printf("%s\n", novoNo->escopo);*/
    TipoID *p = lista[index].start;

    if(p == NULL) { // Lista vazia
        lista[index].start = novoNo;
    } else {  // Lista nao vazia. Insere no final
        while(p->prox!=NULL){
            p = p->prox;
        }
        p->prox = novoNo;
```

```
    }
}

int checkExistance(TipoLista *Lista, char s[], int noline, int index,
    char scope[], int flag){
    int i;
    TipoID *c = Lista[index].start;
    if(flag)
      return 0;
    while(c!=NULL){
        if(!strcmp(s,c->nomeID)){
          if(!strcmp(scope,c->escopo)||!strcmp(c->tipoID, "func")||!
             strcmp(c->escopo, "global")){
          for(i=0;i<c->top;i++){
            if(c->linhas[i]==noline)
              return 1;
          }
          c->linhas[c->top] = noline;
          c->top++;
          return 1;
        }
      }
        c = c->prox;
    }
    return 0;
}

void printWTable(TipoLista *lista, int index) {
  int i;
  TipoID *p = lista[index].start;
  while(p!=NULL){
      i = 0;
      if(p->linhas[0] != 0) {
        printf("%6s    %6s    %6s    %6s        ", p->nomeID, p->
           tipoID, p->tipoData, p->escopo);
        while(p->linhas[i]!=0){
          printf("%d", p->linhas[i]);
          if(i<p->top-1)
            printf(",");
          i++;
        }
        printf("\n");
      }
        p = p->prox;
  }
}

void abrirArq()
{
  yyin = fopen("entrada.txt", "r");
}

/*Semantic Analysis functions*/

/*Checks existance of a given variable*/
int buscaVariavel(TipoLista *list, char nomeID[], char escopo[]) {
    int hash = string2int(nomeID)%211;
    TipoID *c = list[hash].start;
    while(c != NULL) {
```

```
      if((!strcmp(nomeID, c->nomeID)) && (!strcmp(escopo, c->escopo))
          || (!strcmp("global", c->escopo))) {
        return 1;
      }
      c = c->prox;
    }
    if(c == NULL) {
      /*printf("\nVariavel %s nao encontrada no escopo %s\n", nomeID,
          escopo);*/
      return 0;
    }
}

int functionType(TipoLista *list, char nomeID[]){
  int hash = string2int(nomeID)%211;
  TipoID *c = list[hash].start;
  while(c != NULL){
    if(!strcmp(nomeID, c->nomeID)&&!strcmp(c->tipoData, "void"))
      return 0;
    else
      return 1;
  }
}

/*Checks existance of a given function*/
int functionLookup(TipoLista *list, char nomeID[]) {
    int hash = string2int(nomeID)%211;
    TipoID *c = list[hash].start;
    char function[] = "func";
    while(c != NULL) {
      if((!strcmp(nomeID, c->nomeID)) && (!strcmp(function, c->tipoID
          ))) {
        return 1;
      }
      c = c->prox;
    }
    /*if(c == NULL) {*/
      return 0;
    /*}*/
}

/*Role: Checks Existance of void variables*/
int checkVoid(TipoLista *list, int index){
TipoID *p = list[index].start;
while(p!=NULL){
    if(p->linhas[0] != 0) {
      //p->nomeID, p->tipoID, p->tipoData, p->escopo);
      if (!strcmp(p->tipoID, "var")&&!strcmp(p->tipoData,"void")) {
        return p->linhas[0];
      }
    }
      p = p->prox;
    }
    return 0;
}


/*Role: Checks Existance of Main function*/
int checkMain(TipoLista *list, int index){
TipoID *p = list[index].start;
```

```
while(p!=NULL){
    if(p->linhas[0] != 0) {
      //p->nomeID, p->tipoID, p->tipoData, p->escopo);
      if (!strcmp(p->tipoID, "func")&&
          !strcmp(p->tipoData,"void")&&
          !strcmp(p->nomeID, "main")&&
          p->linhas[1]==0)
        return 0;
    }
      p = p->prox;
    }
    return 1;
}

/*Role: Checks Existance of equal declarations in a same scope*/
int checkDecScope(TipoLista *list, int index){
  TipoID *p = list[index].start;
  TipoID *w;

  while(p!=NULL&&(!strcmp(p->tipoID,"var")||!strcmp(p->tipoID,"vet"))
     ){
    if(p->linhas[0] != 0) {
      w = p->prox;
      while (w!=NULL&&(!strcmp(p->tipoID,"var")||!strcmp(p->tipoID,"
          vet"))){
        if(!strcmp(w->nomeID,p->nomeID)&&!strcmp(p->escopo,w->escopo)
           ){
          return w->linhas[0];
        }
        w = w->prox;
      }
    }
      p = p->prox;
    }
    return 0;
  }

/*Role: Checks Existance of variables and functions with similar
    names*/
int checkSameVarFunc(TipoLista *list, int index){
  TipoID *p = list[index].start;
  TipoID *w;
  while (p!=NULL) {
    if (p->linhas[0] != 0) {
      w = p->prox;
      while (w!=NULL) {
        if (!strcmp(w->nomeID,p->nomeID)&&(((!strcmp(w->tipoID,"var")
           ||!strcmp(w->tipoID,"vet"))
            &&!strcmp(p->tipoID,"func"))||((!strcmp(p->tipoID,"var")
               ||!strcmp(w->tipoID,"vet"))
            &&!strcmp(w->tipoID,"func"))
          )){
            if (w->linhas[0])
              w->linhas[0];
            else
              return p->linhas[0];
        }

            w = w->prox;
      }
```

```
    }
    p = p->prox;
  }
  return 0;
}



int semanticAnalysis(TipoLista *hashList){
  int i;
  int j;
  int checkMainFlag = 1;
  int checkDecScopeFlag;
  int checkSameVarFuncFlag;

  for(i = 0;i<211;i++){
      if(&hashList[i]!=NULL){
        // Check Existance of void variables
        int checkVoidFlag;
        checkVoidFlag = checkVoid(hashList, i);
        if (checkVoidFlag)
          printf("Semantic error at line %d: Variable declared as
              void\n ", checkVoidFlag);
        //Check Existance of main function
       if (!checkMain(hashList, i))
         checkMainFlag = 0;
        //Check doubl  e var/vet declarations in a same scope
       checkDecScopeFlag = checkDecScope(hashList,i);
       if(checkDecScopeFlag)
            printf("Semantic error at line %d: double declaration at
                a same scope\n ", checkDecScopeFlag);
  //Check variables and functions with similar names
       checkSameVarFuncFlag = checkSameVarFunc(hashList,i);
       if(checkSameVarFuncFlag)
            printf("Semantic error at line %d: variable and function
                with same name\n ", checkSameVarFuncFlag);
      }
    }
    if (checkMainFlag)
      printf("Semantic error: main function not present or different
          from 'void main(void)'\n ");
  return 0;
}

int main() {
/*
extern int yydebug;
yydebug = 1;
*/
FILE *f_in;
FILE *f_out;
int i;
int w;
int flag;
int buf[100000];
char escopo[30];  //  escopo da funcao
char nomeID[20];  //  nome do ID
char tipoID[3];   //  tipo nenhum <var, fun, vet>
char tipoData[10]; //  tipo de dados <int, float, void>
char nomeIDAnt[20];
```

```
int line = 1;
int hash = 0;
// Alocando o vetor estatico e inicializando ponteiros com NULL
TipoLista vetor[211]; //lista de listas

for(i = 0; i < 211; i++) {
  vetor[i].start = NULL;
}

for (i=0;i<=100000;i++) buf[i] = 0;

abrirArq();

int token;
int cont = 0;               //  contador de chaves inicia com zero no
   IDS.escopo global
strcpy(nomeID, "nome");
strcpy(escopo, "global"); //  inicia laco com escopo global
strcpy(tipoData, "void"); //  tipo void por default
strcpy(tipoID, "non");    //  tipo nenhum <var, fun, vet>
flag = 0;
w = 0;


// Inserindo funcoes predefinidas int input() e void output()
insere(vetor, escopo, "input", "func", "int", -1, 39);
insere(vetor, escopo, "output", "func", "void", -1, 34);

while ((token=yylex()) != '\0') {
  buf[w] = token;
  /*printf("%d\t", token);*/
  w++;
    switch(token) {

      case VOID:
        strcpy(tipoData, "void");
        flag = 2;
    //    puts(tipoData);
      break;

      case FLOAT:
        strcpy(tipoData, "float");
        flag = 1;
  //      puts(tipoData);
      break;

      case INT:
        strcpy(tipoData, "int");
        flag = 1;
  //      puts(tipoData);
      break;

      case LCAPSULE:
        cont++;
        flag = 0;
      break;

      case RCAPSULE:
        cont--;
        if(cont == 0) strcpy(escopo,"global");
```

```
        break;

        case ID:
         hash = string2int(nomeID);
          strcpy(nomeID, yytext);
          token = yylex();
          buf[w] = token;
          /*printf("%d\t", token);*/
          w++;
          // IDS.linhas[0] = noline;
          if(token == LPAREN) {
            strcpy(tipoID, "func");
            if(strcmp(escopo,"global") == 0) {
              strcpy(escopo, nomeID);
            }
            if(flag==0&&!(functionLookup(vetor, nomeID)))
              printf("\nSemantic Error: Non declared function called,
                  '%s()'. Line %d\n", nomeID, lineno);
          } else {
            if(token == LBRACK) {
              // vetor
              // printf("Vetor\n");
              strcpy(tipoID, "vet");
              if(flag == 0 && !(buscaVariavel(vetor, nomeID, escopo)))
                printf("\nSemantic error: bad declaration for Variable
                    '%s'. Line %d\n", nomeID, lineno);


            } else {
              // variavel
              strcpy(tipoID, "var");
              if(flag == 0 && !(buscaVariavel(vetor, nomeID, escopo)))
                printf("\nSemantic error: bad declaration for Variable
                    '%s'. Line %d\n", nomeID, lineno);

            }
          }

          hash = string2int(nomeID)%211;
          int newID = checkExistance(vetor, nomeID, lineno, hash,
             escopo, flag);
          if(!newID){
            insere(vetor, escopo, nomeID, tipoID, tipoData, lineno,
               hash);
          }
          if(token==SEMI) flag = 0;
        break;

        case SEMI:
          flag = 0;
          break;
    }
}


f_out = fopen("out.txt", "w");
i = 0;
  while (buf[i] != 0) {
    switch(buf[i]) {
    case INT: fprintf(f_out, "INT\n"); break;
```

```
  case FLOAT: fprintf(f_out, "FLOAT\n"); break;
  case IF:    fprintf(f_out, "IF\n"); break;
  case ELSE:  fprintf(f_out, "ELSE\n"); break;
  case RETURN:fprintf(f_out, "RETURN\n"); break;
  case VOID:  fprintf(f_out, "VOID\n"); break;
  case WHILE: fprintf(f_out, "WHILE\n"); break;
  case PLUS:  fprintf(f_out, "PLUS\n"); break;
  case MINUS: fprintf(f_out, "MINUS\n"); break;
  case TIMES: fprintf(f_out, "TIMES\n"); break;
  case OVER:  fprintf(f_out, "OVER\n"); break;
  case LT:    fprintf(f_out, "LT\n"); break;
  case LET:   fprintf(f_out, "LET\n"); break;
  case HT:    fprintf(f_out, "HT\n"); break;
  case HET:   fprintf(f_out, "HET\n"); break;
  case EQ:    fprintf(f_out, "EQ\n"); break;
  case NEQ:   fprintf(f_out, "NEQ\n"); break;
  case ASSIGN:fprintf(f_out, "ASSIGN\n"); break;
  case SEMI:  fprintf(f_out, "SEMI\n"); break;
  case COMMA: fprintf(f_out, "COMMA\n"); break;
  case LPAREN:fprintf(f_out, "LPAREN\n"); break;
  case RPAREN:fprintf(f_out, "RPAREN\n"); break;
  case LBRACK:fprintf(f_out, "LBRACK\n"); break;
  case RBRACK:fprintf(f_out, "RBRACK\n"); break;
  case LCAPSULE:      fprintf(f_out, "LCAPSULE\n"); break;
  case RCAPSULE:      fprintf(f_out, "RCAPSULE\n"); break;
  case NUMI:  fprintf(f_out, "NUMI\n"); break;
  case NUMF:  fprintf(f_out, "NUMF\n"); break;
  case ID:    fprintf(f_out, "ID\n"); break;
  }
i++;
}

// Verificacao de atribuicoes
abrirArq();
lineno = 1;
strcpy(escopo, "global"); //  inicia laco com escopo global
while ((token=yylex()) != '\0') {

    switch(token) {

      case ASSIGN:
          if(verState == 0) {
            verState = 1; // passa a procurar lado direito
          }
      break;

      case SEMI:
          if(verState == 1) {
            verState = 0; // volta a procurar lado esquerdo
          }

      break;

      case ID:
          if(verState == 0) {
            strcpy(L_side,yytext);
            if(functionLookup(vetor, L_side)) {
              strcpy(escopo, yytext);
            }
          } else {
```

```
                if(verState == 1) {
                  while(token != SEMI && token != LPAREN) {
                    strcpy(R_side,yytext);
                    if(functionLookup(vetor, R_side)){
                      if(!functionType(vetor, R_side))
                        voidFlag = 1;
                    }

                    token = yylex();
                    if((token == MINUS) || (token == PLUS) || (token ==
                        TIMES) || (token == OVER)) token == yylex();
                  }
                  verState = 0;
                }
              }
            break;
        }
    }


    printf("\nParser running...\n");
    abrirArq();
    if (yyparse()==0) printf("\nSyntax Analysis OK\n");
    else printf("\nERROR in Syntax Analysis\n");

    printTree();
    if(voidFlag)
      printf("Semantic error: Void attribution to variable\n");

    printf("Running Semantic Analysis...\n");
    semanticAnalysis(vetor);
      printf("Semantic Analysis Finished\n");


    printf("Finished.\n");

    printf("Name(ID)  Type(ID)  Type(Data)   Scope    Appears in lines\
       n");
    for(i = 0;i<211;i++){
        if(&vetor[i]!=NULL)
        printWTable(vetor, i);
    }

    return 0;
}
```

Algoritmo 2: cminus.y - Análise Sintática

```
%{
//GLC para gerar parser para C-

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>


#define YYDEBUG 0

/*************************************************/
/***********   Syntax tree for parsing ************/
/*************************************************/

typedef struct treeNode{
  char *str;
  int lineno;
  struct treeNode *child;
  struct treeNode *sibling;
}TreeNode;

TreeNode * tree;// Declaracao da arvore
TreeNode * allocateToken(char const* token);
TreeNode * allocateNode(char const* node);
TreeNode * addChild(TreeNode* node, TreeNode* newChild);
TreeNode * addSibling(TreeNode* first, TreeNode* newSibling);

static int indentNo = 0;

#define YYSTYPE TreeNode *
#define INDENT indentNo+=2
#define UNINDENT indentNo-=2

static char * savedName; /* for use in assignments */
static int savedLineNo;  /* ditto */
static TreeNode * savedTree; /* stores syntax tree for later return
   */
// char * strExp;
std::string strExp;



using namespace std;

extern "C"
{

  ofstream writeTree;
  int yylex();
  int yyparse();
        void abrirArq();
  void effPrintTree(TreeNode * tree);
  void printTree();
  int yywrap() {
```

```
    return 1;
  }
}


extern char* yytext;
extern int yylineno;
FILE *arq;

void yyerror(char*);
%}

%start programa
%token INT       300
%token FLOAT     301
%token IF        302
%token ELSE      303
%token RETURN    304
%token VOID      305
%token WHILE     306
%token PLUS      307
%token MINUS     308
%token TIMES     309
%token OVER      310
%token LT                311
%token LET           312
%token HT                313
%token HET           314
%token EQ                315
%token NEQ               316
%token ASSIGN    317
%token SEMI        318
%token COMMA       319
%token LPAREN    320
%token RPAREN      321
%token LBRACK    322
%token RBRACK    323
%token LCAPSULE  324
%token RCAPSULE  325
%token NUMI          326
%token NUMF          327
%token ID                  328
%token NEWLINE   329
%token ERROR     331

%%

programa        :       /* Entrada Vazia */
                                        | declaration-list
          {
            $$ = allocateNode("programa");
            addChild($$,$1);
            tree = $$;
          }
                                        ;

declaration-list        : declaration-list declaration
                  {$$ = allocateNode("declaration-list");
                   addChild($$,$1);
                   addChild($$,$2);}
```

```
                                                                    |
                                                                        decl
                    {$$ = allocateNode("declaration-list");
                     addChild($$,$1);}
                                                                    ;


declaration             : var-declaration
                {$$ = allocateNode("declaration");
                 addChild($$,$1);}
                | fun-declaration
                {$$ = allocateNode("declaration");
                 addChild($$,$1);}
                                                          ;

var-declaration : type-specifier var-list SEMI
                  {$$ = allocateNode("var-declaration");
                   addChild($$,$1);
                   addChild($$,$2);
                   $3 = allocateNode("SEMI");
                   addChild($$,$3);}
                                                                ;


var-list: var-list COMMA variable
           {$$ = allocateNode("var-list");
            addChild($$,$1);
            $2 = allocateNode("COMMA");
            addChild($$,$2);
            addChild($$,$3);}
           | variable
           {$$ = allocateNode("var-list");
            addChild($$,$1);}
           ;

variable: ID
          {$$ = allocateNode("variable");
           $1 = allocateToken("ID");
           addChild($$,$1);}
          | ID LBRACK NUMI RBRACK
          {$$ = allocateNode("variable");
           $1 = allocateToken("ID");
           $2 = allocateNode("LBRACK");
           $3 = allocateToken("NUMI");
           $4 = allocateNode("RBRACK");
           addChild($$,$1);
           addChild($$,$2);
           addChild($$,$3);
           addChild($$,$4);}
          ;


type-specifier  : INT
                  {$$ = allocateNode("type-specifier");
                   $1 = allocateNode("INT");
                   addChild($$,$1);}
                                                                |
                                                                   FLOAT
```

```
                {$$ = allocateNode("type-specifier");
                 $1 = allocateNode("FLOAT");
                 addChild($$,$1);}
                                                                    |
                                                                       VOID

                {$$ = allocateNode("type-specifier");
                 $1 = allocateNode("VOID");
                 addChild($$,$1);}
                                                                    ;

fun-declaration : type-specifier ID LPAREN params RPAREN compound-
    stmt
                {$$ = allocateNode("fun-declaration");
                 addChild($$,$1);
                 $2 = allocateToken("ID");
                 $3 = allocateNode("LPAREN");
                 addChild($$,$2);
                 addChild($$,$3);
                 addChild($$,$4);
                 $5 = allocateNode("RPAREN");
                 addChild($$,$5);
                 addChild($$,$6);}
                | type-specifier ID LPAREN RPAREN compound-stmt
                {$$ = allocateNode("fun-declaration");
                 addChild($$,$1);
                 $2 = allocateToken("ID");
                 $3 = allocateNode("LPAREN");
                 $4 = allocateNode("RPAREN");
                 addChild($$,$2);
                 addChild($$,$3);
                 addChild($$,$4);
                 addChild($$,$5);}
                ;

params  : param-list
        {$$ = allocateNode("params");
         addChild($$,$1);}
        | VOID
        {$$ = allocateNode("params");
         $1 = allocateNode("VOID");
         addChild($$,$1);}
        ;

param-list  : param-list COMMA param
            {$$ = allocateNode("param-list");
             addChild($$,$1);
             $2 = allocateNode("COMMA");
             addChild($$,$2);
             addChild($$,$3);}
            | param
            {$$ = allocateNode("param-list");
             addChild($$,$1);}
            ;

param       : type-specifier ID
            {$$ = allocateNode("param");
             addChild($$,$1);
             $2 = allocateToken("ID");
```

```
                addChild($$,$2);}
              | type-specifier ID LBRACK RBRACK
              {$$ = allocateNode("param");
               addChild($$,$1);
               $2 = allocateToken("ID");
               $3 = allocateNode("LBRACK");
               $4 = allocateNode("RBRACK");
               addChild($$,$2);
               addChild($$,$3);
               addChild($$,$4);}
              ;

compound-stmt : LCAPSULE local-declarations statement-list RCAPSULE
               {$$ = allocateNode("compound-stmt");
                $1 = allocateNode("LCAPSULE");
                addChild($$,$1);
                addChild($$,$2);
                addChild($$,$3);
                $4 = allocateNode("RCAPSULE");
                addChild($$,$4);}
               ;

local-declarations  : local-declarations var-declaration
                      {
                        $$ = allocateNode("local-declarations");
                        addChild($$,$1);
                        addChild($$,$2);
                      }
                    | /* empty */
                    {$$ = allocateNode("local-declarations");}
                    ;

statement-list      : statement-list statement
                      {
                        $$ = allocateNode("statement-list");
                        addChild($$,$1);
                        addChild($$,$2);
                      }
                    | /* empty */
                    {$$ = allocateNode("statement-list");}
                    ;

statement           : expression-stmt
                      {
                        $$ = allocateNode("statement");
                        addChild($$,$1);
                      }
                    | compound-stmt
                      {
                        $$ = allocateNode("statement");
                        addChild($$,$1);
                      }
                    | selection-stmt
                      {
                        $$ = allocateNode("statement");
                        addChild($$,$1);
                      }
                    | iteration-stmt
                      {
                        $$ = allocateNode("statement");
```

```
                        addChild($$,$1);
                      }
                    | return-stmt
                      {
                        $$ = allocateNode("statement");
                        addChild($$,$1);
                      }
                    ;

expression-stmt     : expression SEMI
                      {
                        $$ = allocateNode("expression-stmt");
                        addChild($$,$1);
                        $2 = allocateNode("SEMI");
                        addChild($$,$2);
                      }
                    | SEMI
                      {
                        $$ = allocateNode("expression-stmt");
                        $1 = allocateNode("SEMI");
                        addChild($$,$1);
                      }
                    ;

selection-stmt      : IF LPAREN comparative-expression RPAREN
                      statement
                      {
                        $$ = allocateNode("selection-stmt");
                        $1 = allocateNode("IF");
                        addChild($$,$1);
                        $2 = allocateNode("LPAREN");
                        addChild($$,$2);
                        addChild($$,$3);
                        $4 = allocateNode("RPAREN");
                        addChild($$,$4);
                        addChild($$,$5);
                      }
                    | IF LPAREN comparative-expression RPAREN
                       statement ELSE statement
                      {
                        $$ = allocateNode("selection-stmt");
                        $1 = allocateNode("IF");
                        addChild($$,$1);
                        $2 = allocateNode("LPAREN");
                        addChild($$,$2);
                        addChild($$,$3);
                        $4 = allocateNode("RPAREN");
                        addChild($$,$4);
                        addChild($$,$5);
                        $6 = allocateNode("ELSE");
                        addChild($$,$6);
                        addChild($$,$7);
                      }
                    ;

iteration-stmt      : WHILE LPAREN comparative-expression RPAREN
                      statement
                      {
                        $$ = allocateNode("iteration-stmt");
                        $1 = allocateNode("WHILE");
```

```
                        addChild($$,$1);
                        $2 = allocateNode("LPAREN");
                        addChild($$,$2);
                        addChild($$,$3);
                        $4 = allocateNode("RPAREN");
                        addChild($$,$4);
                        addChild($$,$5);
                      }
                    ;

return-stmt         : RETURN SEMI
                      {
                        $$ = allocateNode("return-stmt");
                        $1 = allocateNode("RETURN");
                        addChild($$,$1);
                        $2 = allocateNode("SEMI");
                        addChild($$,$2);
                      }

                    | RETURN expression SEMI
                      {
                        $$ = allocateNode("return-stmt");
                        $1 = allocateNode("RETURN");
                        addChild($$,$1);
                        addChild($$,$2);
                        $3 = allocateNode("SEMI");
                        addChild($$,$3);
                      }
                    ;

expression          : var ASSIGN expression
                      {
                        $$ = allocateNode("expression");
                        addChild($$,$1);
                        $1 = allocateNode("ASSIGN");
                        addChild($$,$2);
                        addChild($$,$3);
                      }
                    | simple-expression
                      {
                        $$ = allocateNode("expression");
                        addChild($$,$1);
                      }
                    ;

var                 : ID
                      {
                        $$ = allocateNode("var");
                        $1 = allocateToken("ID");
                        addChild($$,$1);
                      }
                    | ID LBRACK expression RBRACK
                      {
                        $$ = allocateNode("var");
                        $1 = allocateToken("ID");
                        addChild($$,$1);
                        $2 = allocateNode("LBRACK");
                        addChild($$,$2);
                        addChild($$,$3);
                        $4 = allocateNode("RBRACK");
```

```
                        addChild($$,$4);
                      }
                    ;

simple-expression   : comparative-expression
                      {
                        $$ = allocateNode("simple-expression");
                        addChild($$,$1);
                      }
                    | additive-expression
                      {
                        $$ = allocateNode("simple-expression");
                        addChild($$,$1);
                      }
                    ;

comparative-expression: additive-expression relop additive-expression
                        {
                          $$ = allocateNode("comparative-expression")
                              ;
                          addChild($$,$1);
                          addChild($$,$2);
                          addChild($$,$3);
                        }
                    ;

relop               : LET
                      {
                        $$ = allocateNode("relop");
                        $1 = allocateNode("LET");
                        addChild($$,$1);
                      }
                    | LT
                      {
                        $$ = allocateNode("relop");
                        $1 = allocateNode("LT");
                        addChild($$,$1);
                      }
                    | HT
                      {
                        $$ = allocateNode("relop");
                        $1 = allocateNode("HT");
                        addChild($$,$1);
                      }
                    | HET
                      {
                        $$ = allocateNode("relop");
                        $1 = allocateNode("HET");
                        addChild($$,$1);
                      }
                    | EQ
                      {
                        $$ = allocateNode("relop");
                        $1 = allocateNode("EQ");
                        addChild($$,$1);
                      }
                    | NEQ
                      {
                        $$ = allocateNode("relop");
                        $1 = allocateNode("NEQ");
```

```
                        addChild($$,$1);
                      }
                    ;

additive-expression : additive-expression addop term
                      {
                        $$ = allocateNode("additive-expression");
                        addChild($$,$1);
                        addChild($$,$2);
                        addChild($$,$3);
                      }
                    | addop term
                      {
                        $$ = allocateNode("additive-expression");
                        addChild($$,$1);
                        addChild($$,$2);
                      }
                              | term
                      {
                        $$ = allocateNode("additive-expression");
                        addChild($$,$1);
                      }
                    ;

addop               : PLUS
                      {
                        $$ = allocateNode("addop");
                        $1 = allocateNode("PLUS");
                        addChild($$,$1);
                      }
                    | MINUS
                      {
                        $$ = allocateNode("addop");
                        $1 = allocateNode("MINUS");
                        addChild($$,$1);
                      }
                    ;

term                : term mulop factor
                      {
                        $$ = allocateNode("term");
                        addChild($$,$1);
                        addChild($$,$2);
                        addChild($$,$3);
                      }
                    | factor
                      {
                        $$ = allocateNode("term");
                        addChild($$,$1);
                      }
                    ;

mulop               : TIMES
                      {
                        $$ = allocateNode("mulop");
                        $1 = allocateNode("TIMES");
                        addChild($$,$1);
                      }
                    | OVER
                      {
```

```
                    $$ = allocateNode("mulop");
                    $1 = allocateNode("OVER");
                    addChild($$,$1);
                  }
                ;

factor          : LPAREN expression RPAREN
                  {
                    $$ = allocateNode("factor");
                    $1 = allocateNode("LPAREN");
                    addChild($$,$1);
                    addChild($$,$2);
                    $3 = allocateNode("RPAREN");
                    addChild($$,$3);
                  }
                | var
                  {
                    $$ = allocateNode("factor");
                    $1 = allocateNode("var");
                    addChild($$,$1);
                  }
                | call
                  {
                    $$ = allocateNode("factor");
                    $1 = allocateNode("call");
                    addChild($$,$1);
                  }
                | NUMI
                  {
                    $$ = allocateNode("factor");
                    $1 = allocateToken("NUMI");
                    addChild($$,$1);
                  }
                | NUMF
                  {
                    $$ = allocateNode("factor");
                    $1 = allocateToken("NUMF");
                    addChild($$,$1);
                  }
                ;

call            : ID LPAREN args RPAREN
                  {
                    $$ = allocateNode("call");
                    $1 = allocateToken("ID");
                    addChild($$,$1);
                    $2 = allocateNode("LPAREN");
                    addChild($$,$2);
                    addChild($$,$3);
                    $4 = allocateNode("RPAREN");
                    addChild($$,$4);
                  }
                ;

args            : arg-list
                  {
                    $$ = allocateNode("args");
                    addChild($$,$1);
                  }
                | /* empty */
```

```
                        {
                                $$ = allocateNode("args");
                            }
                        ;

arg-list                : arg-list COMMA expression
                          {
                            $$ = allocateNode("arg-list");
                            addChild($$,$1);
                            $2 = allocateNode("COMMA");
                            addChild($$,$2);
                            addChild($$,$3);
                          }
                        | expression
                          {
                            $$ = allocateNode("arg-list");
                            addChild($$,$1);
                          }
                        ;

%%


// void reset(char arg[]) //100%
// {
//      int i, max = strlen(arg);
//
//      for(i = 0; i < max; i++)
//              arg[i] = '\0';
// }


TreeNode * allocateToken(char const *token)
{
        //reset(strExp);
        //strncpy(strExp, yytext, sizeof(strExp));
  strExp.erase();
  strExp = yytext;
        TreeNode *branch = allocateNode(token);
        //puts( yytext );
//      sprintf(strExp, "%s", yytext);
        TreeNode *leaf = allocateNode(strExp.c_str());//because it
           must be a char const*
  // TreeNode *leaf = allocateNode("galeto");
//      addChild(branch, leaf);
        return branch;
}

TreeNode * allocateNode(char const *node)
{

        TreeNode *newNode = (TreeNode*)malloc(sizeof(TreeNode));
        newNode->lineno = yylineno;

        newNode->str = (char*) calloc(sizeof(char),20);
        strcpy(newNode->str, node);

        newNode->child = NULL;
        newNode->sibling = NULL;
  /*printf("alocou no str->%s\n", newNode->str);*/
```

```
        return newNode;
}

TreeNode* addSibling(TreeNode* first, TreeNode* newSibling){
        if(first->sibling == NULL){first->sibling = newSibling;}
        else{first->sibling = addSibling(first->sibling, newSibling)
            ;}
        return first;
}

TreeNode* addChild(TreeNode* node, TreeNode* childNode){
        if(node->child!=NULL){node->child = addSibling(node->child,
            childNode);}
        else{node->child = childNode;}
        return node;
}

TreeNode* freeTree(TreeNode * tree){
        if(tree != NULL)
        {
                if(tree->sibling != NULL){tree->sibling = freeTree(
                    tree->sibling);}

                if(tree->child != NULL){tree->child = freeTree(tree->
                    child);}

                if(tree->child == NULL && tree->sibling == NULL)
                {
                        free(tree);
                        return NULL;
                }
        }
}

/* printSpaces indents by printing spaces */
static void printSpaces(void)
{
        int i;
        for (i=0;i<indentNo;i++)
        {
                //fprintf(arq, "        ");
                printf("_");
        }
}

/* procedure printTree prints a syntax tree to the
 * listing file using indentation to indicate subtrees
 */
void effPrintTree(TreeNode * tree)
{
        INDENT;
        while (tree != NULL)
        {
                printSpaces();
                //fprintf(arq, "%s\n",tree->str);
    printf("%s\n",tree->str);

                tree = tree->child;

                while(tree != NULL)
```

```
                {
                        effPrintTree(tree);
                        tree = tree->sibling;
                }
                UNINDENT;
        }
}

void printTree()
{
  printf("Imprimindo arvore sintatica...\n");
        arq = fopen("syntaticTree.xls", "w");
        effPrintTree(tree);
        fclose(arq);
}

void yyerror (char* s)  /* Called by yyparse on error */
{
        extern char* yytext;
        cout << s << ": " << yytext << endl << "At line: " <<
           yylineno <<  endl;
  strExp = (char*) calloc(sizeof(char),40);
}
```