

**LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO
DO RIO GRANDE DO NORTE**

Fábio Henrique Ferreira Silva

**LIPP – PROTÓTIPO DE UM COMPILADOR PARA UMA LINGUAGEM DE
PROGRAMAÇÃO EM PORTUGUÊS**

**NATAL
2006**

FARN

ICAD DE FARMACIA DO GRANDE DO NORTE
UNIDADE DE FARMACIA O DESENVOLVIMENTO
DO FARMACIA DO NORTE

Edição Henrique Ferreira Silva

UNIDADE DE FARMACIA DO GRANDE DO NORTE
UNIDADE DE FARMACIA DO GRANDE DO NORTE

RB. 29308

FARN
BIBLIOTECA
Nº Reg. M000618
28/05/07

004.43
5582
2006
Ex. 01

EXTRA
2006

FÁBIO HENRIQUE FERREIRA SILVA

LIPP – PROTÓTIPO DE UM COMPILADOR PARA UMA LINGUAGEM DE PROGRAMAÇÃO EM PORTUGUÊS

LIPP – PROTÓTIPO DE UM COMPILADOR PARA UMA LINGUAGEM DE PROGRAMAÇÃO EM PORTUGUÊS

Monografia apresentada ao curso de graduação em Sistemas da Informação da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte como requisito obrigatório para obtenção do grau de Bacharel em Sistemas da Informação

Orientadora: Bartira P. Falcão Dantas Rocha.
M.Sc.

**NATAL
2006**

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Silva, Fábio Henrique Ferreira

LIPP – protótipo de um compilador para uma linguagem de programação em português / Fábio Henrique Ferreira Silva. – Natal: FARN, 2006.
73f.

Orientador: Bartira P. Falcão Dantas Rocha (M.Sc.)

Monografia (Bacharelado em Sistemas de Informação) – Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte.

1. Linguagem de Programação – Monografia. 2. Linguagem LIPP – compilação – Monografia. I. Título.

RN/FARN/BC

CDU 004.43

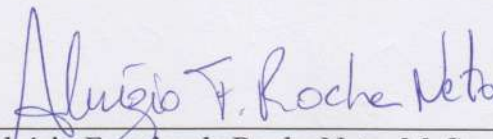
FÁBIO HENRIQUE FERREIRA SILVA

**LIPP – PROTÓTIPO DE UM COMPILADOR PARA UMA LINGUAGEM DE
PROGRAMAÇÃO EM PORTUGUÊS**

Monografia apresentada ao curso de Bacharelado em Sistemas da Informação da Faculdade
Natalense para o Desenvolvimento do Rio Grande do Norte e aprovada em 16 de dezembro de
2006 à banca examinadora composta pelos seguintes membros:



Bartira P. Falcão Dantas Rocha, M. Sc
Orientadora


Aluizio Ferreira da Rocha Neto, M. Sc
Membro Examinador
Ricardo Kleber Martins Galvão, M. Sc
Membro Examinador

AGRADECIMENTOS

Agradeço a todas as pessoas que doaram os seus conhecimentos e esforços para a realização deste trabalho. Especialmente a professora Maria F. Falcão Duarte Rocha, orientadora deste trabalho pela atenção, paciência e apoio para a realização e conclusão do mesmo.

À minha esposa Aline dedico este trabalho pelo apoio constante e compreensão pelos momentos de ausência em função desta empreitada. Às minhas filhas Beatriz e Mariana pelos finais de semanas perdidos.

Aos meus pais pela educação que me deram e me fizeram chegar até aqui.

AGRADECIMENTOS

Agradeço à todos aqueles que direta ou indiretamente contribuíram para a realização deste trabalho. Especialmente a professora Bartira P. Falcão Dantas Rocha, orientadora deste trabalho pela atenção, presteza e suporte para a realização e conclusão do mesmo.

RESUMO

Este trabalho visa reunir informações sobre a construção de compiladores e o processo de definição de uma linguagem de programação, com o objetivo de subsidiar o desenvolvimento de um protótipo de um compilador e a definição de uma linguagem de programação, que utilize o idioma português do Brasil como base para seus comandos. Esta linguagem e este protótipo, servirão de ponto de partida para o desenvolvimento de um compilador e uma linguagem completa, que poderão ser utilizados como ferramentas de apoio à disciplina de construção de algoritmos e lógica de programação dos cursos de bacharelado em sistemas da informação e licenciatura em sistemas da informação da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte, bem como cursos correlatos de outras instituições.

Palavras-Chave: Compiladores, Linguagem LIPP, Linguagem de Programação, Compilação

RESUMEM

Este trabalho tiene como objetivo para juntar la información sobre la construcción de compiladores y del proceso de la definición de un lenguaje de programación, con el objetivo para subvencionar el desarrollo de un arquetipo de un compilador y de la definición de un lenguaje de programación, que utiliza la lengua portuguesa del Brasil como base para sus comandos. Esta lenguaje y este arquetipo, servirán del punto de partida para el desarrollo de un compilador y de una lengua completa, que podrían ser utilizados como herramientas de la ayuda a la disciplina de la construcción de algoritmos y de la lógica de la programación de los cursos del bacharelado en los sistemas de la información y del licenciatura en los sistemas de la información de la universidad Natalense para o Desenvolvimento do Rio Grande do Norte, así como correlatos de los cursos de otras instituciones.

Palabras Clave: Lenguaje LIPP, Lenguaje de programación, Recompilador, Compilación

SUMÁRIO

1 INTRODUÇÃO.....	10
1.1 PROBLEMA DE PESQUISA.....	10
1.2 OBJETIVO GERAL.....	11
1.2.1 Objetivos Específicos.....	11
1.3 JUSTIFICATIVA.....	11
1.4 METODOLOGIA.....	12
2 REFERENCIAL TEÓRICO.....	13
2.1 LINGUAGEM, ALFABETO E PALAVRAS.....	13
2.2 ÁRVORES GRAMATICAIIS.....	14
2.3 AMBIGÜIDADE.....	15
2.4 ASSOCIATIVIDADE DOS OPERADORES.....	15
2.5 PRECEDÊNCIA DE OPERADORES.....	15
2.6 COMPILADORES.....	16
2.7 ANÁLISE LÉXICA.....	18
2.7.1 Classes, Padrões E Símbolos.....	18
2.8 ANÁLISE SINTÁTICA.....	19
2.9 ANÁLISE SEMÂNTICA.....	20
2.10 GERAÇÃO DE CÓDIGO.....	20
2.10.1 Geração de Código Intermediário.....	20
2.11 OTIMIZAÇÃO DE CÓDIGO.....	21
2.12 GERAÇÃO DO CÓDIGO ALVO.....	21
3 A LINGUAGEM LIPP.....	23
3.1 GRAMÁTICA DA LINGUAGEM LIPP.....	23
3.1.1 Comentários.....	23
3.1.2 Tipos de Dados.....	24

3.1.3 Variáveis.....	24
3.1.4 Atribuição.....	24
3.1.6 Operadores Relacionais.....	25
3.1.7 Estrutura de um programa LIPP.....	25
3.1.8 Controle de Fluxo.....	25
3.1.9 Estrutura Condicional.....	25
3.1.10 Estrutura de Repetição.....	26
3.1.11 Entrada e Saída de Dados.....	26
3.1.12 Gramática LIPP em EBNF.....	27
4 O COMPILADOR.....	29
4.1 FLEX.....	29
4.1.1 O Arquivo de Entrada Flex.....	30
4.2 YACC.....	33
4.2.1 O Arquivo de Entrada do Yacc.....	33
4.3 MÁQUINA VIRTUAL.....	36
4.4 GERAÇÃO DO CÓDIGO.....	37
5 CONCLUSÃO.....	39
REFERÊNCIAS.....	40
ANEXO.....	41

LISTA DE ILUSTRAÇÕES

Figura 1 – Um compilador.	16
Figura 2 - Fases de um compilador	17
Figura 3 – Interação entre o analisador léxico e o analisador sintático	19
Quadro 1 – Exemplo de símbolos	18
Quadro 2 – Exemplo de otimização de código	21
Quadro 3 – Operadores Aritméticos.....	24
Quadro 4 – Operadores Relacionais.....	25
Quadro 5 – Metacaracteres	30
Quadro 6 – Exemplos de uso dos metacaracteres	30
Quadro 7 - Instruções RO	37
Quadro 8 - Instruções RM	37

1.1 PROBLEMA DE PESQUISA

1 INTRODUÇÃO

Nos cursos da área de informática uma das matérias onde há mais dificuldade de aprendizagem é a de algoritmos e lógica de programação. A programação, seja estruturada ou orientada à objetos ou eventos, representa uma nova forma de pensar e elaborar soluções para problemas utilizando técnicas e metodologias muitas vezes desconhecidas pelos alunos.

Disciplinas na área de programação são normalmente divididas em dois enfoques: teórico e prático. No enfoque teórico são, habitualmente, discutidos conceitos como a lógica e a estrutura de comandos, enquanto no enfoque prático esses conceitos são testados e verificados de acordo com o que foi visto na teoria. Observa-se que a dificuldade de aprendizado aumenta quando os alunos vão ao laboratório e deparam-se com toda a complexidade abordada em sala de aula, além de ter que aprender a utilizar uma ferramenta apropriada para a compilação ou interpretação dos programas.

O raciocínio lógico é a base para o desenvolvimento de programas, é importante que os alunos consigam ver no algoritmo o funcionamento da lógica que eles aprenderam.

Com o objetivo de facilitar o aprendizado geralmente utiliza-se o Portugol como linguagem para discussão dos conceitos e elaboração dos algoritmos durante as aulas teóricas, porém, nas aulas práticas ainda adota-se o uso de ferramentas cujo idioma padrão é o inglês. Neste cenário, o professor precisa desviar-se do conteúdo específico de algoritmos e ensinar o mínimo da linguagem que usará em suas aulas práticas. A diferença do que se vê em sala de aula – Portugol – com o que se pratica – Inglês – aumenta a dificuldade de entendimento do assunto que está sendo lecionado.

Para resolver este problema, este trabalho propõe-se a desenvolver o protótipo de um compilador que usará o idioma Português do Brasil como padrão. Por usar o idioma nativo dos alunos busca-se proporcionar uma melhor assimilação da matéria, pois, o que é discutido na teoria será aplicado na prática sem a necessidade de conversões ou traduções.

1.1 PROBLEMA DE PESQUISA

As linguagens de programação atuais usam o idioma inglês como padrão e são utilizadas durante as aulas práticas como ferramentas de apoio à disciplina. Muitos alunos dos

tiveram contato com uma linguagem de programação. Toda a interação do aluno com a ferramenta de apoio precisará ser ensinada pelo professor, uma vez que a mesma por ser em outro idioma não torna-se intuitiva. Este treinamento desvia o professor do conteúdo da matéria e também demanda parte do tempo destinado à disciplina.

A falta de uma linguagem em Português que possa ser usada para exemplos práticos prejudica a boa assimilação dos conceitos discutidos em sala de aula e demanda maior tempo de aprendizagem.

Como facilitar o aprendizado das disciplinas algoritmos e lógica de programação?

1.2 OBJETIVO GERAL

O objetivo deste trabalho é desenvolver um protótipo de compilador cuja linguagem de programação possua uma gramática semelhante a da linguagem Pascal, mas que utilize termos da língua portuguesa do Brasil em seu vocabulário, visando facilitar o ensino e o aprendizado de algoritmos e da lógica de programação pelos alunos dos cursos de computação.

1.2.1 Objetivos Específicos

- Realizar um estudo detalhado das etapas de construção de compiladores.
- Analisar as ferramentas adequadas para a construção de compiladores.
- Definir uma linguagem que será usada como referência na concepção do compilador.
- Implementar e testar o protótipo do compilador proposto.

1.3 JUSTIFICATIVA

Este trabalho visa facilitar o aprendizado das matérias algoritmos e lógica de programação pelos alunos dos períodos iniciais dos cursos de sistemas da informação. Estas matérias são a base para o entendimento de quaisquer linguagens computacionais, portanto, é imprescindível que os alunos tenham uma ferramenta adequada para praticarem o que vêm em sala de aula.

O tempo de aprendizagem será reduzido e a assimilação do conteúdo da disciplina será

idioma dos alunos, elimina-se a necessidade de traduzir os exemplos e exercícios para a linguagem que seria utilizada nas aulas práticas.

2.1 LINGUAGEM, ALFABETO E PALAVRAS

1.4 METODOLOGIA

A metodologia utilizada neste trabalho é de pesquisa aplicada, uma vez que, a conclusão do mesmo gerará um aplicativo que será utilizado com a finalidade de resolver o problema descrito anteriormente. A abordagem adotada foi a de pesquisa qualitativa, pois baseou-se na observação do dia-a-dia dos alunos do curso de Bacharelado em Sistemas da Informação da FARN para a delimitação do problema. Para alcançar o objetivo foi utilizada uma pesquisa exploratória através de levantamento bibliográfico e documental.

Uma linguagem de programação pode ser definida pela sua sintaxe e por sua semântica. A sintaxe de uma linguagem de programação define a sua estrutura e os seus componentes, enquanto a semântica dá o significado aos componentes. Por exemplo, a sintaxe para o comando `if` da linguagem C é:

`if ( <expressão> ) <instrução>`

A semântica deste comando é que se o valor atual de expressão for verdadeiro, a instrução será executada.

Para especificar a sintaxe de uma linguagem utiliza-se uma gramática livre de contexto que é definida por Delamero (2004, p. 11) como uma quadrupla (Σ, T, S, P) , onde:

Σ – é um conjunto não vazio de símbolos não terminais;

T – é o alfabeto sobre o qual a linguagem é definida;

S – é o símbolo inicial da gramática, $S \in \Sigma$;

P – é o conjunto de produções da forma $A \rightarrow \alpha$, onde $A \in \Sigma$ e $\alpha \in (\Sigma \cup \Sigma^*)$.

Alto (1993, p. 12) destaca que "Uma gramática descreve naturalmente a estrutura hierárquica de muitas construções das linguagens de programação". Por exemplo, um comando `if-else` em C, possui a forma:

2 REFERENCIAL TEÓRICO

2.1 LINGUAGEM, ALFABETO E PALAVRAS

Antes de descrevermos as fases de um compilador precisamos definir alguns conceitos básicos sobre linguagens formais e como elas podem ser definidas.

Uma linguagem é formada a partir de palavras contidas em um alfabeto. Um alfabeto é um conjunto finito de símbolos, como por exemplo o alfabeto da língua portuguesa que é constituído pelos símbolos de A até Z, que normalmente chamamos de letras. A partir do alfabeto obtém-se as palavras que são uma seqüência ou cadeias destes símbolos.

Formalmente, uma linguagem é qualquer conjunto de cadeias formadas a partir de uma alfabeto dado. Aho (1995, p. 42) afirma que “O termo linguagem denota qualquer conjunto de cadeias sobre algum alfabeto fixo”.

Uma linguagem de programação pode ser definida pela sua sintaxe e por sua semântica. A sintaxe de uma linguagem de programação define a sua estrutura e os seus componentes, enquanto a semântica dá o significado aos componentes. Por exemplo, a sintaxe para o comando `if` da linguagem C é:

`if (<expressão>) <instrução>`

A semântica deste comando é que: se o valor atual da expressão for verdadeiro, a instrução será executada.

Para especificar a sintaxe de uma linguagem utiliza-se uma gramática livre de contexto que é definida por Delamaro (2004, p 11) como uma quadrúpla (Ω, Σ, S, P) , onde

Ω – é um conjunto não vazio de símbolos não terminais;

Σ – é o alfabeto sobre o qual a linguagem é definida;

S – é o símbolo inicial da gramática, $S \in \Omega$;

P – é o conjunto de produções da forma $A \rightarrow \alpha$, onde $A \in \Omega$ e $\alpha \in (\Sigma \cup \Omega)^*$.

Aho (1995, p. 12) destaca que “Uma gramática descreve naturalmente a estrutura hierárquica de muitas construções das linguagens de programação”. Por exemplo, um comando *if-else* em C, possui a forma:

2.1 **if** (expressão) comando **else** comando

Ou seja, o comando é uma concatenação da palavra-chave **if**, um parênteses à esquerda, uma expressão, um parênteses à direita, um comando, a palavra-chave **else** e um outro comando.

Usando-se a variável *expr* a fim de denotar uma expressão e a variável *cmd* para um comando, esta regra de estruturação pode ser expressa como:

$$\text{cmd} \rightarrow \text{if}(\text{expr}) \text{cmd} \text{ else } \text{cmd}$$

2.4 ASSOCIATIVIDADE DOS OPERADORES

onde a seta deve ser lida como “pode ter a forma”. Tal regra é chamada de uma produção. Numa produção os elementos léxicos, como a palavra-chave **if** e os parênteses, são chamados de lexemas. As variáveis *expr* e *cmd* representam seqüências de lexemas e são chamadas de não-terminais.

2.2 ÁRVORES GRAMATICAIIS

Uma árvore gramatical mostra como o símbolo de partida de uma gramática deriva uma cadeia de linguagem. Se um não-terminal *A* possui uma produção $A \rightarrow XYZ$, então uma árvore gramatical pode ter um nó interno rotulado *A*, com três filhos rotulados *X*, *Y* e *Z* da esquerda para a direita. (AHO, 1995, p.13)

Uma árvore gramatical possui as seguintes propriedades:

1. A raiz é rotulada pelo símbolo de partida.
2. Cada folha é rotulada por um símbolo ou por λ .
3. Cada nó interno é rotulado por um não-terminal.
4. Se *A* é um não-terminal rotulando algum nó interno e $X_1, X_2, \dots, X_n$ são os rótulos dos filhos daquele nó, da esquerda para a direita, então $A \rightarrow X_1, X_2, \dots, X_n$ é uma produção. Se $A \rightarrow \lambda$, então um nó rotulado *A* deve possuir um único filho rotulado λ .

2.3 AMBIGÜIDADE

Uma gramática é dita ambígua, quando ela tem mais de uma árvore gramatical gerando a mesma cadeia de símbolos. Uma cadeia com mais de uma árvore possui usualmente mais de um significado, Aho (1995, p 14) destaca que “é preciso projetar para as aplicações de compilação, gramáticas não ambíguas ou ambíguas com regras adicionais para resolver as ambigüidades”.

2.4 ASSOCIATIVIDADE DOS OPERADORES

A associatividade de operadores define como os operadores capturam os operandos em uma expressão. Uma linguagem de programação precisa saber o que fazer quando uma expressão $9 + 5 + 2$ aparece no código fonte.

Aho (1995, p. 14) menciona que “Na maioria das linguagens de programação, os quatro operadores aritméticos – de adição, subtração, multiplicação e divisão -, são associativos à esquerda.”, o exemplo acima é equivalente à $(9 + 5) + 2$, pois o operando 5 é capturado pelo operador + localizado do lado esquerdo.

2.5 PRECEDÊNCIA DE OPERADORES

Além da associatividade, uma linguagem de programação precisa se preocupar com a precedência dos operadores. Na matemática sabemos que a multiplicação e a divisão são executadas antes da adição e da subtração e esta regra precisa ser obedecida pelo programa que será gerado por um compilador.

Na expressão $9 + 5 * 2$, observando a regra da associatividade ela seria equivalente à $(9 + 5) * 2$, porém a multiplicação segundo a matemática tem precedência em relação a adição, a expressão também poderia ser interpretada como $9 + (5 * 2)$. Temos aqui uma ambigüidade que precisa ser resolvida.

A solução para este problema é obtida através de uma tabela de precedência de operadores.

2.6 COMPILADORES

Compiladores são programas que traduzem uma linguagem de alto nível para uma outra linguagem, que pode ser também de alto nível ou uma linguagem de máquina. Neste último caso, o programa gerado é normalmente executável e independente de aplicações auxiliares. Segundo Aho (1995, p 1) “um compilador é um programa que lê um programa escrito numa linguagem - a linguagem fonte - e o traduz num programa equivalente em outra linguagem - a linguagem alvo.”

Podemos dividir a compilação em duas partes, a análise e a síntese. A análise divide o programa fonte em partes constituintes e cria uma representação intermediária do mesmo. A síntese constrói o programa alvo desejado a partir do programa intermediário. A figura 1 mostra uma representação básica de um compilador.



Figura 1 – Um compilador.
Fonte: (Aho, 1995)

A análise consiste de três fases; a análise linear ou léxica onde o programa é lido da esquerda para a direita e os caracteres agrupados em classes. A análise hierárquica ou sintática onde as classes são agrupadas hierarquicamente e a análise semântica que verifica os erros entre os componentes da linguagem fonte assegurando uma combinação significativa entre eles e informações de tipo.

Além das fases de análise um compilador possui ainda a geração do código intermediário, a otimização, a geração do código alvo, a geração da tabela de símbolos e a manipulação de erros.

A figura 2 mostra um compilador com todas as suas fases.

2.3 ANÁLISE LÉXICA

A análise léxica é a primeira fase de um compilador. Sua função principal é a de ler os caracteres de entrada e produzir uma sequência de tokens que serão utilizados na análise sintática. Ou seja, ela separa e identifica os símbolos que compõem o programa fonte. Outra função do analisador léxico é retirar os comentários e espaços em branco, as operações sobre a forma de tabulação, etc.

2.3.1 Classes

Na análise léxica, os tokens são classificados em "palavras-chaves" e "identificadores" com significados específicos.

As palavras-chaves são um conjunto de palavras reservadas que possuem funções específicas. Este conjunto de palavras é definido pelo compilador e é parte integrante de um padrão. O padrão reconhece cada palavra do conjunto. Um exemplo de palavras-chaves em Pascal é:

Figura 2 – Fases de um compilador
Fonte: Aho (1995)

const pi = 3.1416;

Simbolo	Classe
const	Palavra reservada
pi	Identificador
=	Operador
3.1416	Literal

Quadro 1 – Exemplo de símbolos

2.7 ANÁLISE LÉXICA

A análise léxica é a primeira fase de um compilador. Sua função principal é a de ler os caracteres de entrada e produzir uma sequência de símbolos que serão utilizados na análise sintática, ou seja, ela separa e identifica os elementos que compõem o programa fonte. Outra função do analisador léxico é retirar do programa fonte os comentários e espaços em brancos, os últimos sob a forma de tabulações, avanços de linha, etc.

2.7.1 Classes, Padrões E Símbolos

Na análise léxica usaremos os termos “**classes**”, “**padrões**” e “**símbolos**” com significados específicos.

As **classes** são um conjunto de cadeias de entrada que possuem funções semelhantes. Este conjunto de cadeias é descrito por uma regra denominada de um **padrão**. O padrão reconhece cada cadeia do conjunto. Um **símbolo** é um conjunto de caracteres no programa-fonte que é reconhecido pelo padrão de alguma classe. Por exemplo, em Pascal:

```
const pi = 3,1416;
```

Símbolo	Classe
const	Palavra reservada
Pi	Identificador
“=”	Operador
3,1416	Número

Quadro 1 – Exemplo de símbolos.

2.8 ANÁLISE SINTÁTICA

A análise sintática é a segunda fase no processo de compilação, a sua função é garantir a estrutura da linguagem, ou seja, que os componentes estejam na ordem especificada pelas regras gramaticais definidas para a mesma. A sintaxe das linguagens de programação pode ser definidas por gramáticas livres de contexto ou pela notação BNF (Forma de Backus-Naur).

A análise sintática pode ser classificada em ascendente ou descendente. Na análise descendente a árvore sintática é construída partindo-se da raiz em direção as folhas, enquanto na ascendente o processo é o inverso.

O analisador sintático recebe uma cadeia de componentes do analisador léxico e verifica se a mesma pode ser gerada pela gramática da linguagem. Caso a cadeia seja aceita o analisador sintático irá gerar a árvore sintática para a linguagem.

Esta fase também é responsável pelo tratamento dos erros sintáticos, conforme Aho (1995, p. 73)

“Freqüentemente, boa parte da detecção e recuperação de erros num compilador gira em torno da fase de análise sintática. Isto porque os erros ou são sintáticos por natureza ou são expostos quando o fluxo de tokens provenientes do analisador léxico desobedece às regras gramaticais que definem a linguagem de programação.”

A figura 3 temos a interação entre o analisador léxico e o sintático, onde o analisador léxico lê o programa fonte, verifica se os símbolos pertencem a linguagem e repassa-os para o analisador sintático.

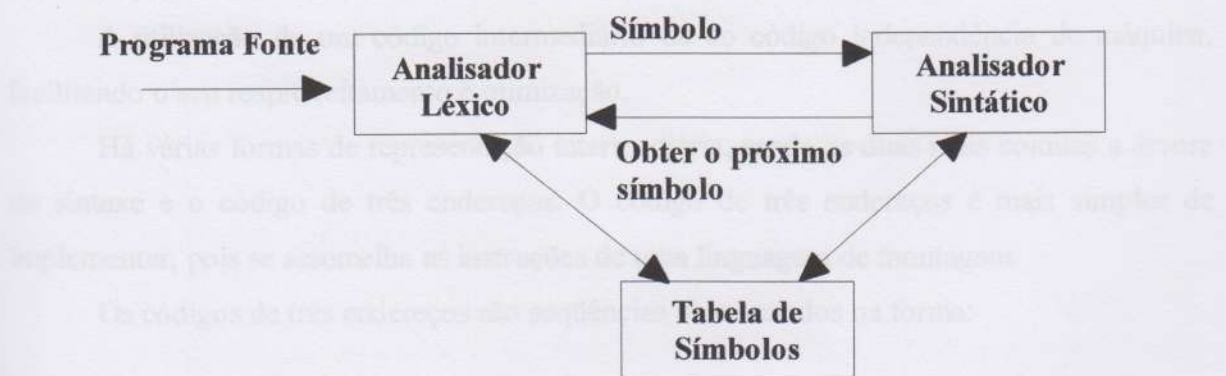


Figura 3 – Interação entre o analisador léxico e o analisador sintático.
Fonte: Aho (1995).

2.9 ANÁLISE SEMÂNTICA

A análise semântica é responsável pela verificação da árvore sintática em busca de inconsistências semânticas. Nesta fase são verificados os tipos, as declarações de variáveis, as funções e procedimentos.

Cabe ao analisador semântico verificar a coerência entre a declaração e uso dos identificadores, a compatibilidade entre operandos e identificar as referências não resolvidas, tais como, ponteiros para registros inexistentes.

A análise semântica é responsável por verificar aspectos do programa relacionados ao significado de cada comando. Um programa pode estar correto de acordo com as regras de sintaxe definida pela gramática, mas apresentar problemas relacionados com a semântica da linguagem. (DELAMARO, 2004, p. 199).

A análise semântica é feita percorrendo-se a árvore sintática gerada na fase de análise sintática.

2.10 GERAÇÃO DE CÓDIGO

2.10.1 Geração de Código Intermediário

Um compilador pode implementar a fase de geração de código intermediário. Esta fase é responsável por gerar uma representação intermediária do programa que será utilizada pela fase de geração do código final.

A utilização de um código intermediário dá ao código independência de máquina, facilitando o seu reaproveitamento e otimização.

Há várias formas de representação intermediária, sendo as duas mais comuns a árvore de sintaxe e o código de três endereços. O código de três endereços é mais simples de implementar, pois se assemelha as instruções de uma linguagem de montagem.

Os códigos de três endereços são seqüências de comandos na forma:

x := y op z

onde **x**, **y** e **z** são variáveis ou constantes definidas pelo programador ou variáveis temporárias geradas pelo compilador, **op** define um operador e **:=** define uma atribuição.

2.11 OTIMIZAÇÃO DE CÓDIGO

A otimização do código visa melhorar o código gerado tornando-o mais eficiente. Isto é feito eliminando instruções desnecessárias ou redundantes, sub-expressões comuns, código morto, efetuar transformações algébricas entre várias outras oportunidades de otimização que podem aparecer no programa gerado.

Como exemplo de otimização podemos usar uma transformação algébrica na expressão $x = a + b * c$, que pode ser transformada em $x = b * c + a$, de acordo com a propriedade comutativa da soma. A troca correspondente no código seria:

De	Para
Load b	Load b
Mult c	Mult c
Store t1	Add a
Load a	Store x
Add t1	
Store x	

Quadro 2 – Exemplo de otimização de código

Com a transformação elimina-se as manipulações da variável temporária **t1** e das instruções que a manipulariam.

2.12 GERAÇÃO DO CÓDIGO ALVO

A geração do código alvo é a fase final de um compilador. Esta fase recebe o código intermediário otimizado na fase de otimização e produz como saída o programa alvo.

O código alvo poderá ter como saída uma linguagem de máquina, uma linguagem

compilador é a geração do código alvo, consistindo normalmente de código de máquina ou código de montagem”.

Nesta fase as localizações de memória são alocadas para as variáveis, as instruções intermediárias são traduzidas para instruções de máquina e as variáveis são atribuídas aos registradores.

No projeto do gerador do código deve-se levar em consideração a ordem na qual as computações são realizadas, o gerenciamento da memória, o custo das instruções, e outras características que visam o desenvolvimento de um gerador de código eficiente.

Esta é uma das fases mais complexas no projeto de construção de compiladores porque ela requer o conhecimento das instruções de máquina específicas de cada processador para o qual o código alvo será gerado.

2.1 GRAMÁTICA DA LINGUAGEM LPP

A gramática da linguagem LPP foi descrita, segundo a notação EBNF (Extended Backus - Naur Form). A EBNF é uma extensão da BNF que foi inventada por J. Backus e P. Naur para descrever a sintaxe da linguagem Algol 60 em 1960.

A LPP possui comentários, comando de repetição, estruturas de fluxo e expressões lógicas.

2.1.1 Comentários

A LPP como as outras linguagens são usadas em programas escritos em linguagem Pascal usa-se as chaves “{” e “}” para comentários de uma linha ou o “/* */” para comentários de múltiplas linhas. Na LPP foi adotado o estilo de múltiplas linhas de C.

Um comentário em LPP deverá ser escrito assim:

/* Isto é um comentário */

ou

/* Um exemplo de comentário de
múltiplas linhas */

3 A LINGUAGEM LIPP

LIPP significa Linguagem de Programação em Português, ela foi definida para usar o português do Brasil como base para seus comandos. O objetivo da linguagem é facilitar o aprendizado da lógica de programação e da construção de algoritmos, disciplinas presentes nos diversos cursos de informática. A gramática de LIPP é simples e conta com poucos comandos e apenas dois tipos de dados, o inteiro e o lógico. A compilação da LIPP gerará um arquivo com código para uma máquina virtual que será a responsável pela execução do programa. A máquina virtual não foi desenvolvida neste trabalho, ela foi adaptada para reconhecer os programas LIPP.

3.1 GRAMÁTICA DA LINGUAGEM LIPP

A gramática da linguagem LIPP foi descrita usando a notação EBNF (Extension Backus – Naur Form). A EBNF é uma extensão da BNF que foi inventada por J. Backus e P. Naur para descrever a sintaxe da linguagem Algol 60 em 1960.

A LIPP possui comentários, comando de repetição, controle de fluxo e expressões básicas.

3.1.1 Comentários

A forma como os comentários são usados em programas varia de linguagem para linguagem, em Pascal usa-se as chaves “{}”, em Perl o numeral “#”, em C pode-se usar o “//” para o comentário de uma linha ou o “/* */” para comentários de múltiplas linhas. Na LIPP foi adotado o estilo de múltiplas linhas do C.

Um comentário em LIPP deverá ser escrito assim:

/* Isto é um comentário */

ou

**/* Um exemplo de comentário de
múltiplas linhas */**

3.1.2 Tipos de Dados

Como o objetivo da LIPP é ser uma linguagem base para o protótipo de um compilador, ela conta com apenas dois tipos de dados, o inteiro e o lógico.

3.1.3 Variáveis

Variáveis são definidas na gramática como identificadores. Os identificadores devem começar com uma letra e podem conter números na formação de seus nomes, não são permitidos caracteres especiais nem espaços na formação do nome. Uma variável também não poderá ter o mesmo nome de uma palavra reservada. Não há declaração explícita de variáveis, o seu tipo é determinado na primeira vez em que ela é usada.

3.1.4 Atribuição

A operação de atribuição é denotada através do símbolo “<=” (seta), amplamente utilizado nas disciplinas de construção de algoritmos e lógica de programação.

Exemplo:

salario <= salario * 2;

3.1.5 Operadores Aritméticos

Os operadores aritméticos básico definidos na LIPP.

<i>Operação</i>	<i>Símbolo</i>
Adição	+
Subtração	-
Multiplicação	*
Divisão	/

3.1.6 Operadores Relacionais

Os operadores relacionais reconhecidos pela LIPP são:

<i>Operador</i>	<i>Símbolo</i>
Maior que	>
Menor que	<
Igual	=
Diferente	!=
Maior ou Igual	>=
Menor ou Igual	<=

Quadro 4 – Operadores Relacionais

3.1.7 Estrutura de um programa LIPP

Um programa em LIPP é uma seqüência de comandos terminados por ponto-e-vírgula. Não há marcação de início ou fim do programa nem procedimentos ou funções.

3.1.8 Controle de Fluxo

Um controle de fluxo básico foi implementado através de estruturas condicionais e de repetição. As estruturas condicionais utilizam o comando **SE**, enquanto as estruturas de repetição utilizam o comando **REPETIR**.

3.1.9 Estrutura Condicional

Em LIPP o **SE** forma a estrutura condicional de um programa. Se uma expressão for verdadeira a seqüência de comandos imediatamente após o **ENTAO** será executada, caso contrário a execução passa para os comandos após o **SENAO**.

A sintaxe de um comando SE é:

ou

SE expressão ENTAO comandos SENAO comandos FIM

Exemplo de uma estrutura SE ENTAO SENAO:

se salario < 100 entao

salario <- salario * 1.20;

senao

salario <- salario * 1.10;

fim;

3.1.10 Estrutura de Repetição

A estrutura de repetição implementada na LIPP é a **REPETIR**. Ela é uma estrutura básica de repetição onde os comandos são executados no mínimo uma vez e uma condição é testada para que sejam executados novamente ou não.

A sintaxe do comando REPETIR é:

REPETIR comandos ATE expressao

Exemplo de uma estrutura REPETIR:

repetir

salario <- salario * 1.20;

ate salario > 100;

3.1.11 Entrada e Saída de Dados

Na LIPP a entrada de dados é feita através do comando **LEIA** que espera como parâmetro uma variável, e a saída de dados é feita através do comando **ESCREVA** que também espera uma variável.

Exemplo de uso dos comandos LEIA e ESCRIVA.

número = dígito { dígito }

leia x; = Mandar ler

escreva x;

3.1.12 Gramática LIPP em EBNF

A seguir é apresentada a gramática da LIPP construída a partir das definições apresentadas anteriormente:

```

programa = sequenciaComandos
identificador = letra { letra | dígito }
letra = a | b | c | ... | z | A | B | C | ... | Z
dígito = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
listaIdentificador = identificador { “,” identificador }
seqüênciaComandos = comando { “;” comando }
comando = (cmdAtribuição | cmdSe | cmdRepetir | cmdLeia | comandoEscreva)
cmdAtribuição = variável “<-” expressão
cmdRepetir = repetir sequenciaComandos ate expressão
cmdSe = se expressão entao comando [ senao comando ] fim
cmdLeia = leia identificador
cmdEscreva = escreva identificador
expressao = expressaoSimples < expressaoSimples | expressaoSimples >
expressaoSimples | expressaoSimples = expressaoSimples | expressaoSimples <=
expressaoSimples | expressaoSimples >= expressaoSimples | expressaoSimples !=
expressaoSimples | expressaoSimples
expressaoSimples = expressaoSimples + termo | expressaoSimples - termo | termo
termo = termo * fator | termo / fator | fator
fator = identificador | número | cadeia | “(“ expressão “)”
operadorRelacional = “=” | “<” | “<” | “>” | “>” | “<=” | “<=”

```

operadorMultiplicação = "*" | "/"

número = dígito { dígito }

variável = identificador

do compilador para a LPT foram utilizados os utilitários FLEX e YACC. Estes utilitários aplicam a metodologia de compiladores, especificamente as fases de análise léxica e sintática.

Baseado em um arquivo com a especificação da linguagem, o flex produz o código-fonte em C referente ao analisador léxico e o yacc produz o código correspondente ao analisador sintático.

4.1 FLEX

O FLEX é uma evolução do LEX, ele processa um arquivo texto contendo expressões regulares e ações associadas a elas para reconhecer ou não um símbolo. O FLEX produzirá um arquivo contendo o código C que define um procedimento chamado `yylex`. Este procedimento é uma implementação de um AFD (Automato Finito Determinístico) que será utilizado pelo analisador sintático. O arquivo será gerado com o nome `lex.yy.c` que deverá ser compilado com os demais fontes para gerar o compilador.

As expressões regulares no FLEX são compostas dos caracteres exibidos na tabela 5 e exemplos de uso estão na tabela 6. Elas nos permitem especificar padrões que serão associados a ações que por sua vez retornarão os símbolos que fazem parte da linguagem. Um exemplo de expressão regular para definir um identificador é:

`letra(letra | dígito)*`

Este exemplo diz que, um identificador deve ser formado por uma letra, seguida por zero ou mais letras ou dígitos. C, CONT, CONTI, CI são identificadores válidos, enquanto IC, SVAR não são identificadores válidos.

4 O COMPILADOR

Para o desenvolvimento do compilador para a LIPP foram utilizados os utilitários FLEX e YACC. Estes utilitários agilizam a construção de compiladores, especificamente as fases de análise léxica e sintática.

Baseado em um arquivo com a especificação da linguagem, o flex produz o código fonte em C referente ao analisador léxico e o yacc produz o código correspondente ao analisador sintático.

4.1 FLEX

O FLEX é uma evolução do LEX, ele processa um arquivo texto contendo expressões regulares e ações associadas a elas para reconhecer ou não um símbolo. O FLEX produzirá um arquivo contendo o código C que define um procedimento chamado yylex. Este procedimento é uma implementação de um AFD (Automato Finito Determinístico) que será utilizado pelo analisador sintático. O arquivo será gerado com o nome lex.yy.c que deverá ser compilado com os demais fontes para gerar o compilador.

As expressões regulares no FLEX são compostas dos metacaracteres exibidos na tabela 5 e exemplos de uso estão na tabela 6. Elas nos permitem especificar padrões que serão associados a ações que por sua vez retornarão os símbolos que fazem parte da linguagem. Um exemplo de expressão regular para definir um identificador é:

letra(letra | dígito)*

Este exemplo diz que, um identificador deve ser formado por uma letra, seguido por zero ou mais letras ou dígitos. C, CONT, CONT1, C1 são identificadores válidos, enquanto 1C, 5VAR não são identificadores válidos.

<i>Metacaracteres</i>	<i>Comparação</i>
.	Quaisquer caracteres exceto “nova-linha”
\n	“Nova-linha”
*	Zero ou mais cópias da expressão anterior.
+	Uma ou mais cópias da expressão anterior
?	Zero ou uma cópia da expressão anterior
^	O que estiver no início da linha
\$	O que estiver no final da linha
()	Um grupo
[]	Classe de caracteres
	Seleção “OU”
“ ”	O que estiver entre as aspas

Quadro 5 – Metacaracteres

<i>Exemplo</i>	<i>Resultado</i>
abc	abc
abc*	ab abc abcc abccc abcccc ...
abc+	abc abcc abccc abcccc ...
a(bc)+	abc abcb abcbcb abcbcbcb ...
[abc]	Um dos caracteres: a b c
a b	a ou b
a(bc)?	a abc
[^ab]	Qualquer coisa que não termine em ab

Quadro 6 – Exemplos de uso dos metacaracteres

4.1.1 O Arquivo de Entrada Flex

O arquivo que especifica as regras para o FLEX é um arquivo texto simples dividido em três seções. As seções são separadas pelos caracteres %% conforme o exemplo:

definições...

%%

regras

%%

sub-rotinas

A seção de definição vem antes do primeiro `%%`. Nela deverão ficar quaisquer códigos C a serem inseridos externamente a qualquer função e deverão ser delimitados por `%{ e %}`.

Nesta seção também deverão ficar as definições de nomes das expressões regulares que serão usadas. O nome da expressão deve aparecer na primeira coluna seguido da expressão regular separada por um ou mais espaços.

Exemplo da seção definição:

```
%{
/* Este é um comentário que será inserido no início do arquivo lex.yy.c .
Ele será inserido exatamente como está, bem como todas as demais
linhas que estão entre os delimitadores %{ e %} */

#include <stdio.h>

int numero = 1;

}%

letra      [a-zA-Z]

digito     [0-9]

identificador {letra}+{digito}*

%%

regras

%%
```

As palavras **letra**, **digito** e **identificador** são nomes para as expressões regulares que a seguem.

A próxima seção, é a seção de regras, elas são compostas por uma sequência de expressões regulares seguidas pela ação a ser executada, quando ocorrer o casamento com a expressão correspondente. As regras são incluídas entre os delimitadores **%%** como a seguir:

```
%%
{letra}      { printf("A letra %s foi encontrada !\n", yytext); }
{identificador} { printf("O identificador %s foi encontrado !\n", yytext); }
%%
```

A última seção é a das sub-rotinas, ela é opcional e contém código auxiliar para as ações contidas na seção de regras.

Um trecho do arquivo lex para a LIPP vem a seguir, o conteúdo completo está no anexo I.

Seção de definições

```
%{
#include "globais.h"
#include "util.h"
#include "varredura.h"
char simboloLiteral[TAMMAXSIMB+1];
}%

digito      [0-9]
numero      {digito}+
letra       [a-zA-Z]
identificador {letra}+{digito}*
novaLinha   \n
espBranco   [ \t]+
```

Seção de regras

%%

```
"se"          {return SE;}
"entao" {return ENTAO;}
"senao" {return SENAO;}
"fim"       {return FIM;}
"enquanto"  {return ENQUANTO;}
"faca"      {return FACA;}
"leia"      {return LEIA;}
```

Para gerar o arquivo `lex.yy.c` basta usar o comando: **flex nome_arquivo.lex**.

4.2 YACC

O YACC é um programa utilizado para gerar analisadores sintáticos para compiladores, assim como o FLEX ele utiliza um arquivo de entrada contendo a especificação da linguagem e produz um arquivo de saída composto pelo código C do analisador. O arquivo gerado será o **y.tab.c**.

4.2.1 O Arquivo de Entrada do Yacc

O formato do arquivo com a especificação para o YACC é semelhante a do FLEX e também se divide em três seções.

definições

%%

regras

%%

rotinas auxiliares

A primeira seção é a de definições e conterá informações sobre os símbolos, tipos de dados e regras gramaticais. esta seção poderá ficar vazia.

Um exemplo desta seção seria:

```
%{
/* Este é um comentário que será inserido no início do arquivo y.tab.c . Ele será
   inserido exatamente como está */

#include <stdio.h>

int numero = 1;

}%

%token NUMERO SE SENAO ENQUANTO

%%
```

Um item importante no exemplo acima é o “**%token**”, após ele deverão ser relacionados todos os símbolos que serão usados na linguagem. Mais de um “**%token**” pode aparecer no arquivo, sendo um em cada linha. Os símbolos relacionados devem ser separados por espaço.

Na seção de regras ficam as regras gramaticais em uma forma BNF modificada, junto com ações em código C que serão executadas quando uma regra for reconhecida. Uma regra gramatical no YACC sempre termina com ponto-e-vírgula.

```
%%

expressao :      expressao “+” termo {$$ = $1 + $3;} |
               expressao “-” termo {$$ = $1 - $3;} |
               termo {$$ = $1;};

termo:         termo “*” fator {$$ = $1 * $3;} |
               termo “/” fator {$$ = $1 / $3;} |
               fator {$$ = $1;};

fator:         NUMERO      {$$ = $1;} |
               “(“ expressao “)”      {$$ = $2;};

%%
```

Os caracteres “\$\$” representam o símbolo que foi reconhecido e que está do lado esquerdo da regra gramatical, enquanto os caracteres “\$1”, “\$2” são pseudovariáveis que representam os valores de cada símbolo à direita da regra.

A última seção é onde pode-se adicionar códigos auxiliares para o analisador, esta seção pode ficar vazia.

Para gerar o analisador sintático com YACC, basta após ter criado o arquivo de especificação digitar o comando: **yacc nome_arquivo.yac**.

Para que o analisador gerado pelo YACC interaja com o analisador léxico gerado pelo FLEX é necessário produzir um arquivo com as definições dos símbolos usados por ele. Para obter este arquivo use o comando: **yacc -d nome_arquivo.yac**. Este comando além do arquivo **y.tab.c** irá produzir o arquivo **y.tab.h** que deverá ser incluído através de um cláusula **#include** no arquivo **.lex**.

Um trecho do arquivo yacc para a linguagem LIPP vem abaixo, o arquivo completo encontra-se na seção de anexos.

Seção de Definições

```
%{
#define YYPARSER /* diferencia a saída Yacc de outros códigos */

#include "globais.h"

#include "lexico.h"

#include "sintaxe.h"

#define YYSTYPE noArvore *

static char * nomeSalvo; /* para uso nas atribuições */

static int numLinhaSalvo; /* idem */

static noArvore * arvoreSalva; /* armazena a arvore sintática para retorno */

%}

%token SE ENTAO FIM ENQUANTO FACA LEIA ESCREVA

%token ID NUMERO PARESQUERDO PARDIREITO PONTOVIRGULA
```

Seção das regras

%%

cmd_leia : LEIA ID

{ \$\$ = novoNoDeclara(cLeia);

\$\$->atributo.nome = copiaCaracteres(simboloLiteral); };

cmd_escreva : ESCREVA expressao

{ \$\$ = novoNoDeclara(cEscreva);

\$\$->filho[0] = \$2; };

4.3 MÁQUINA VIRTUAL

O compilador LIPP gerará código para uma máquina virtual. A escolha pela utilização de uma máquina virtual foi devido a complexidade de implementação de um gerador de código para uma arquitetura real. Essa complexidade requeria um tempo de desenvolvimento maior, tornando inviável a entrega deste trabalho no prazo previsto.

A máquina virtual utilizada se baseia na máquina de Louden, ela é composta por uma memória para instruções, uma memória de dados e oito registradores de uso geral.

O conjunto de instruções da máquina é listado nas tabelas abaixo. Há dois formatos básico de instruções, um para instruções apenas entre registradores que será referenciada por RO e outra para instruções entre os registradores e a memória, que é referenciada por RM.

As instruções RO têm o formato: codop r,s,t onde, codop é a instrução a ser executada e r,s,t são registradores.

As instruções RM têm o formato: codop r,d(s) onde, r e s são registradores e d é um deslocamento na memória.

<i>Codop</i>	<i>Efeito</i>
HALT	Interrompe a execução
IN	Reg[r] entrada de dados em r
OUT	Reg[r] saída de dados
ADD	$\text{Reg[r]} = \text{reg[s]} + \text{reg[t]}$
SUB	$\text{Reg[r]} = \text{reg[s]} - \text{reg[t]}$
MUL	$\text{Reg[r]} = \text{reg[s]} * \text{reg[t]}$
DIV	$\text{Reg[r]} = \text{reg[s]} / \text{reg[t]}$

Quadro 7 Instruções RO

<i>Codop</i>	<i>Efeito</i>
LD	$\text{Reg[r]} = \text{dMem[a]}$ carrega r com o valor de memória em a
LDA	$\text{Reg[r]} = a$ carrega e endereça a diretamente em r
LDC	$\text{Reg[r]} = d$ carrega constante d diretamente em r-s
ST	$\text{dMem[a]} = \text{reg[r]}$ armazena r na posição de memória a
JLT	Se $(\text{reg[r]} < 0)$ $\text{reg[PC_REG]} = a$
JLE	Se $(\text{reg[r]} \leq 0)$ $\text{reg[PC_REG]} = a$
JGE	Se $(\text{reg[r]} \geq 0)$ $\text{reg[PC_REG]} = a$
JGT	Se $(\text{reg[r]} > 0)$ $\text{reg[PC_REG]} = a$
JNE	Se $(\text{reg[r]} \neq 0)$ $\text{reg[PC_REG]} = a$

Quadro 8 Instruções RM

4.4 GERAÇÃO DO CÓDIGO

O compilador LIPP gera código para a máquina virtual descrita anteriormente. Os códigos gerados são gravados em um arquivo cujo nome é o mesmo do programa fonte, mas cujo sufixo é .mv. O código gerado é semelhante aos códigos da linguagem ASSEMBLER e são os relacionados na seção sobre a máquina virtual.

O gerador de código percorre a árvore sintática, identifica se o nó lido é uma expressão ou uma declaração e executa a geração de código apropriada. A tabela de símbolos é acessada no caso em que um acesso a uma variável é necessário.

Os exemplos abaixo mostram um programa em LIPP e um trecho do código gerado pelo compilador para o programa.

/* Programa em LIPP */

leia x;

se x > 0 entao

fatorial <- 1;

repetir

fatorial <- fatorial * x;

x <- x - 1;

ate x != 0;

escreva fatorial;

fim;

0: LD 6,0(0)

1: ST 0,0(0)

2: IN 0,0,0

3: ST 0,0(5)

4: LD 0,0(5)

5: ST 0,0(6)

6: LDC 0,0(0)

7: LD 1,0(6)

5 CONCLUSÃO

REFERÊNCIAS

Um dos grandes desafios que alunos iniciantes nos cursos de informática enfrentam é o descobrimento de que a forma como as soluções são idealizadas a fim de serem automatizadas requerem uma nova forma de pensamento e uma reestruturação de idéias em um formato totalmente novo. Para que uma determinada tarefa seja automatizada deve-se, além de outras coisas, elaborar o algoritmo que representa a codificação a ser realizada. Um dos problemas existentes é justamente o uso de linguagens de programação, assim como as ferramentas de codificação, os compiladores, que fazem uso da língua inglesa. Apesar de saber-se que nos dias de hoje todas as linguagens de programação usam o idioma inglês como base para sua gramática, para alunos que estão iniciando no curso de informática, trabalhar com uma ferramenta na sua própria língua seria de grande ajuda.

Outro ponto importante é o fato de que muitas vezes o aluno de um curso de informática sabe utilizar um compilador, domina uma determinada linguagem, mas não conhece em detalhes as etapas que envolvem o processo de criação de uma linguagem de programação, assim como de um compilador. Ou seja, conhece a prática mas não a teoria, ou conhece a teoria mas não sabe como ela ocorre na prática.

Este trabalho é o ponto de partida para um projeto completo de um compilador. O protótipo permitiu unir a teoria com a prática dando uma visão completa do processo de construção de um compilador e da definição de um linguagem. O compilador completo e a linguagem que está sendo desenvolvida a partir deste protótipo, visam contribuir na melhoria da aprendizagem dos alunos das disciplinas de algoritmos e lógica de programação, pois, tanto o compilador quanto a linguagem poderão ser usados como ferramentas de auxílio nas salas de aulas.

REFERÊNCIAS.

AHO, Alfred V.; SETHI, Ravi.; ULLMAN, Jeffrey D. **Compiladores: Princípios, Técnicas e Ferramentas**. Rio de Janeiro: LTC – Livros Técnicos e Científicos, 1995.

DELAMARO, Márcio E. **Como construir um compilador: Utilizando Ferramentas Java**. São Paulo: Novatec Editora Ltda, 2004.

LOUDEN, Kenneth C. **Compiladores: Princípios e Práticas**. São Paulo: Pioneira Thomson Learning, 2004.

SALIBA, Walter L. Caram. **Técnicas de programação: Uma abordagem estruturada**. São Paulo: Makron, McGraw-Hill, 1992.

SEBESTA, Robert W. **Conceitos de linguagens de programação**. 4. ed. Porto Alegre: Bookman, 2000.

TERRY, P.D. **Compilers and compilers generators: an introduction with C++**. Rhodes University, 1996. Disponível em: <<http://www.scifac.ru.ac.za/compilers>>. Acesso em 13 Jun 2006.

ANEXO

FONTES DO COMPIADOR

ARQUIVO DE ESPECIFICAÇÃO DO FLEX

```
*****
* Arquivo: fop.lex                                     *
* Especificação Lex para o compilador LPT              *
* Trabalho de final de curso - BSI - FARN              *
* Fabio Henrique F. Silva                               *
*****
```

```
%include "global.h"
%include "util.h"
%include "lexico.h"
char simbolo[1000];[TAMMAXSIMB+1]
```

ANEXOS

```
digito [0-9]
numero [digito]+
letra [a-zA-Z]
identificador {letra}+{digito}*
separador [ ]
operacao [ + ]
opcao [ ]
```

```
%c
```

```
"sc"      {return SLC;}
"entao"   {return ENTAO;}
"entoao"  {return ENTAO;}
"fin"     {return FIM;}
"enquanto" {return ENQUANTO;}
"faça"    {return FAÇA;}
"até"     {return ATÉ;}
"escreva" {return ECREVA;}
"repita"  {return REPETIR;}
"para"    {return PARA;}
"até"     {return ATE;}
"passo"   {return PASSO;}
"inicio"  {return INICIO;}
"fim"     {return FIM;}
```

ANEXO

FONTES DO COMPILADOR

ARQUIVO DE ESPECIFICAÇÃO DO FLEX

```

/*****
/* Arquivo: lipp.lex */
/* Especificação Lex para a linguagem LIPP */
/* Trabalho de final de curso - BSI - FARN */
/* Fábio Henrique F. Silva */
*****/

%{
#include "globais.h"
#include "util.h"
#include "lexico.h"
char simboloLiteral[TAMMAXSIMB+1];
}%

digito    [0-9]
numero    {digito}+
letra     [a-zA-Z]
identificador {letra}+{digito}*
novaLinha \n
espBranco [\t]+
aspas     ""

%%

"se"      {return SE;}
"entao"   {return ENTAO;}
"senao"   {return SENAO;}
"fim"     {return FIM;}
"enquanto" {return ENQUANTO;}
"faca"    {return FACA;}
"leia"    {return LEIA;}
"escreva" {return ESCREVA;}
"repetir" {return REPETIR;}
"para"    {return PARA;}
"ate"     {return ATE;}
"passo"   {return PASSO;}
"inicio"  {return INICIO;}
"fim"     {return FIM;}

```

```

" <="      {return MENORIGUAL;}
" >="      {return MAIORIGUAL;}
" !=      {return DIFERENTE;}
" <-"      {return ATRIBUICAO;}
" ="      {return IGUAL;}
" >"      {return MAIOR;}
" <"      {return MENOR;}
" +"      {return MAIS;}
" -"      {return MENOS;}
" *"      {return MULTIPLICA;}
" /"      {return DIVISAO;}
" ("      {return PARESQUERDO;}
" )"      {return PARDIREITO;}
" ;"      {return PONTOVIRGULA;}
{numero}   {return NUMERO;}
{identificador} {return ID;}
{novaLinha} {numLinha++;}
{espBranco} {/* ignora espaços em branco */}
"/*"
    { char c;
      int OK = NAO;
      do
      { c = input();
        if (c == EOF) break;
        if (c == '\n') numLinha++;
        if (c == '*') {
          c = input();
          if (c == '/') OK = SIM;
        }
      } while (!OK);
    }
    {return ERRO;}
<<EOF>>    {return ENDFILE;}
%%
int yywrap(void) { return 1; }

TipoSimbolo lerSimbolo(void)
{ static int primeiraVez = SIM;
  TipoSimbolo simboloAtual;
  if (primeiraVez)
  { primeiraVez = NAO;
    numLinha++;
    yyin = arqFonte;
    yyout = arqSaida;
  }
  simboloAtual = yylex();
  strncpy(simboloLiteral,yytext,TAMMAXSIMB);
  if (RastreiaLex) {
    fprintf(arqSaida,"\t%d: ",numLinha);

```

```

    imprimirSimbolo(simboloAtual,simboloLiteral);
}
return simboloAtual;
}

```

ARQUIVO DE ESPECIFICAÇÃO DO YACC

```

/*****/
/* Arquivo: lipp.yac */
/* Arquivo de especificação para o Yacc */
/* Trabalho de final de curso - BSI - FARN */
/* Fábio Henrique F. Silva */
/*****/
%{
#define YYPARSER /* diferencia a saída Yacc de outros códigos */

#include "globais.h"
#include "util.h"
#include "lexico.h"
#include "sintaxe.h"

#define YYSTYPE NoArvore *
static char * nomeSalvo; /* para uso nas atribuições */
static int numLinhaSalvo; /* idem */
static NoArvore * arvoreSalva; /* armazena a arvore sintática para retorno */

%}

%token SE ENTAO SENAO INICIO FIM ENQUANTO FACA LEIA ESCREVA ID
NUMERO MAIOR
%token IGUAL MENOR MAIS MENOS MULTIPLICA DIVISAO PARESQUERDO
PARDIREITO PONTOVIRGULA
%token ATRIBUICAO ERRO MAIORIGUAL MENORIGUAL DIFERENTE PARA ATE
PASSO REPETIR

%% /* Gramatica para LIPP */

programa : seq_comandos
        { arvoreSalva = $1; }
        ;

seq_comandos : seq_comandos PONTOVIRGULA comandos
        { YYSTYPE t = $1;
          if (t != NULL)
            { while (t->primo != NULL)
              { t = t->primo;
                t->primo = $3;
                $$ = $1; }
            }
          else $$ = $3;
        }

```

```

    }
    | comandos { $$ = $1; }
    ;
comandos : cmd_se { $$ = $1; }
    | cmd_enquanto { $$ = $1; }
    | cmd_atribuicao { $$ = $1; }
    | cmd_leia { $$ = $1; }
    | cmd_escreva { $$ = $1; }
    | cmd_repetir { $$ = $1; }
    ;
cmd_se : SE expressao ENTAO seq_comandos FIM
    { $$ = novoNoDeclara(cSe);
      $$->filho[0] = $2;
      $$->filho[1] = $4;
    }
    | SE expressao ENTAO seq_comandos SENAO seq_comandos FIM
    { $$ = novoNoDeclara(cSe);
      $$->filho[0] = $2;
      $$->filho[1] = $4;
      $$->filho[2] = $6;
    }
    ;
cmd_enquanto : ENQUANTO expressao FACA seq_comandos FIM
    { $$ = novoNoDeclara(cEnquanto);
      $$->filho[0] = $2;
      $$->filho[1] = $4;
    }
    ;
cmd_atribuicao : ID { nomeSalvo = copiaCaracteres(simboloLiteral);
                    numLinhaSalvo = numLinha; }
    ATRIBUICAO expressao
    { $$ = novoNoDeclara(cAtribuicao);
      $$->filho[0] = $4;
      $$->atributo.nome = nomeSalvo;
      $$->numLinha = numLinhaSalvo;
    }
    ;
cmd_leia : LEIA ID
    { $$ = novoNoDeclara(cLeia);
      $$->atributo.nome =
        copiaCaracteres(simboloLiteral);
    }
    ;
cmd_escreva : ESCREVA expressao
    { $$ = novoNoDeclara(cEscreva);
      $$->filho[0] = $2;
    }
    ;

```

```

cmd_repetir : REPETIR seq_comandos ATE expressao
    { $$ = novoNoDeclara(cRepetir);
      $$->filho[0] = $2;
      $$->filho[1] = $4;
    }
;

expressao : expressao_simples MENOR expressao_simples
    { $$ = novoNoExp(cOp);
      $$->filho[0] = $1;
      $$->filho[1] = $3;
      $$->atributo.op = MENOR;
    }
| expressao_simples IGUAL expressao_simples
    { $$ = novoNoExp(cOp);
      $$->filho[0] = $1;
      $$->filho[1] = $3;
      $$->atributo.op = IGUAL;
    }
| expressao_simples MAIOR expressao_simples
    { $$ = novoNoExp(cOp);
      $$->filho[0] = $1;
      $$->filho[1] = $3;
      $$->atributo.op = MAIOR;
    }
| expressao_simples MAIORIGUAL expressao_simples
    { $$ = novoNoExp(cOp);
      $$->filho[0] = $1;
      $$->filho[1] = $3;
      $$->atributo.op = MAIORIGUAL;
    }
| expressao_simples MENORIGUAL expressao_simples
    { $$ = novoNoExp(cOp);
      $$->filho[0] = $1;
      $$->filho[1] = $3;
      $$->atributo.op = MENORIGUAL;
    }
| expressao_simples DIFERENTE expressao_simples
    { $$ = novoNoExp(cOp);
      $$->filho[0] = $1;
      $$->filho[1] = $3;
      $$->atributo.op = DIFERENTE;
    }
| expressao_simples { $$ = $1; }
;

expressao_simples : expressao_simples MAIS termo
    { $$ = novoNoExp(cOp);
      $$->filho[0] = $1;
      $$->filho[1] = $3;
    }

```

```

    $$->atributo.op = MAIS;
  }
  | expressao_simples MENOS termo
  { $$ = novoNoExp(cOp);
    $$->filho[0] = $1;
    $$->filho[1] = $3;
    $$->atributo.op = MENOS;
  }
  | termo { $$ = $1; }
;

termo : termo MULTIPLICA fator
  { $$ = novoNoExp(cOp);
    $$->filho[0] = $1;
    $$->filho[1] = $3;
    $$->atributo.op = MULTIPLICA;
  }
  | termo DIVISAO fator
  { $$ = novoNoExp(cOp);
    $$->filho[0] = $1;
    $$->filho[1] = $3;
    $$->atributo.op = DIVISAO;
  }
  | fator { $$ = $1; }
;

fator : PARESQUERDO expressao PARDIREITO
  { $$ = $2; }
  | NUMERO
  { $$ = novoNoExp(cConstante);
    $$->atributo.val = atoi(simboloLiteral);
  }
  | ID { $$ = novoNoExp(cId);
    $$->atributo.nome =
      copiaCaracteres(simboloLiteral);
  }
;

%%

int yyerror(char * mensagem)
{ fprintf(arqSaida,"Erro de sintaxe na linha %d: %s\n",numLinha,mensagem);
  fprintf(arqSaida,"Símbolo atual: ");
  imprimirSimbolo(yychar,simboloLiteral);
  Erro = SIM;
  return 0;
}

/* yylex chama lerSimbolo para compatibilizar a saída do Yacc/Bison
 * com versões anteriores do analisador léxico da LIPP

```

```
static int yylex(void)
{ return lerSimbolo(); }
*/
```

```
NoArvore * sintaxe(void)
{ yyparse();
  return arvoreSalva;
}
```

CODIGO.H

```

/*****
/* Arquivo: codigo.h */
/* Interface das funções para codigo.c */
/* Trabalho de final de curso - BSI - FARN */
/* Fábio Henrique F. Silva */
/*
/* Baseado no compilador TINY do livro Construção */
/* de Compiladores - Principios e Práticas, autor */
/* Kenneth C. Louden - Tradução 2004 */
*****/

```

```
#ifndef _CODIGO_H_
#define _CODIGO_H_
```

```
/* cp = contador de programa */
#define cp 7
```

```
/* pm = "ponteiro de memória" aponta para o topo
 * da memória (para armazenamento temporário)
 */
```

```
#define pm 6
```

```
/* pg = "ponteiro global" aponta para o final da
 * memória para armazenamento de variáveis (global)
 */
```

```
#define pg 5
```

```
/* acumulador */
```

```
#define ac 0
```

```
/* 2o acumulador */
```

```
#define ac1 1
```

```
/* utilitários de emissão de código */
```

```
/* O procedimento emitirComentario grava o comentário
 * no arquivo de código
```

```

*/
void emitirComentario( char * comentario );

/* O procedimento emitirRO emite uma instrução Apenas Registrador
* op = código operação
* r = registrador alvo
* s = 1o registrador de origem
* t = 2o registrador de origem
* c = um comentário a ser gravado se o rastreiaCod é SIM
*/
void emitirRO( char *op, int r, int s, int t, char *c);

/* O procedimento emitirRM emite uma instrução Registrador-Para-Memoria
* op = código operação
* r = registrador alvo
* d = deslocamento
* s = registrador base
* c = um comentário a ser gravado se o rastreiaCod é SIM
*/
void emitirRM( char * op, int r, int d, int s, char *c);

/* A função emitirSalto salta "distancia" localizações de código
* para retorno posterior. Ela também retorna a posição atual do código
*/
int emitirSalto( int distancia);

/* O procedimento emitirCopia copia para loc a localização saltada anteriormente */
void emitirCopia( int loc);

/* O procedimento emitirRecuperar recupera a posição atual do código para a posição mais
* alta não emitida.
*/

void emitirRecuperar(void);

/* O procedimento emitirRM_Abs converte uma referência absoluta para uma
* referência cp-relativa quando emitir uma instrução registrador-para-memoria da LIPP
* op = código operação
* r = registrador alvo
* a = a localização absoluta na memória
* c = um comentário a ser gravado se o rastreiaCod é SIM
*/
void emitirRM_Abs( char *op, int r, int a, char * c);

#endif

```

CODIGO.C

```

/*****
/* Arquivo: codigo.c */
/* Funções de chamada da máquina virtual */
/* Trabalho de final de curso - BSI - FARN */
/* Fábio Henrique F. Silva */
/*
/* Baseado no compilador TINY do livro Construção */
/* de Compiladores - Principios e Práticas, autor */
/* Kenneth C. Louden - Tradução 2004 */
*****/

#include "globais.h"
#include "codigo.h"

/* Localização da instrução atual na MV */
static int emitirLoc = 0 ;

/* Localização mais alta na MV para uso em conjunto com emitirSalto,
emitirCopia, and emitirRecuperar
*/
static int emitirLocAlta = 0;

/* Procedimento emitirComentario coloca uma linha de comentário no arquivo
* de código gerado.
*/
void emitirComentario( char * c )
{ if (RastreiaCod) fprintf(arqCodigo,"* %s\n",c);}

/* Procedimento emitirRO emite uma instrução de registro
* op = codop
* r = registrador de destino
* s = 1o registrador fonte
* t = 2o registrador fonte
* c = comentário a ser gravado se RastreiaCod é SIM
*/
void emitirRO( char *op, int r, int s, int t, char *c)
{ fprintf(arqCodigo,"%3d: %5s %d,%d,%d",emitirLoc++,op,r,s,t);
  if (RastreiaCod) fprintf(arqCodigo,"\t%s",c) ;
  fprintf(arqCodigo,"\n") ;
  if (emitirLocAlta < emitirLoc) emitirLocAlta = emitirLoc ;
} /* emitirRO */

/* Procedimento emitirRM emite uma instrução de registro para memória
* op = codop
* r = registrador de destino
* d = deslocamento

```

```

* s = registrador base
* c = comentário a ser gravado se RastreiaCod é SIM
*/
void emitirRM( char * op, int r, int d, int s, char *c)
{ fprintf(arqCodigo,"%3d: %5s %d,%d(%d) ",emitirLoc++,op,r,d,s);
  if (RastreiaCod) fprintf(arqCodigo,"\t%s",c) ;
  fprintf(arqCodigo,"\n") ;
  if (emitirLocAlta < emitirLoc) emitirLocAlta = emitirLoc ;
} /* emitirRM */

/* Função emitirSalto salta localizações do código. O salto é definido por "distancia".
* Ela também pode retornar a posição atual.
*/
int emitirSalto( int distancia)
{ int i = emitirLoc;
  emitirLoc += distancia ;
  if (emitirLocAlta < emitirLoc) emitirLocAlta = emitirLoc ;
  return i;
} /* emitirSalto */

/* Procedimento emitirCopia copia até loc = uma localização anteriormente
* saltada.
*/
void emitirCopia( int loc)
{ if (loc > emitirLocAlta)
  emitirComentario("A localização extrapola a localização mais alta em emitirCopia");
  emitirLoc = loc ;
} /* emitirCopia */

/* Procedimento emitirRecuperar recupera a posição atual do código
* para a posição mais alta anterior
*/
void emitirRecuperar(void)
{ emitirLoc = emitirLocAlta;}

/* Procedimento emitirRM_Abs converte uma referência absoluta para uma
* referência relativa a pc quando emitir uma instrução de registrador para memória
* op = codop
* r = registrador de destino
* a = uma localização absoluta na memória
* c = comentário a ser gravado se RastreiaCod é SIM
*/
void emitirRM_Abs( char *op, int r, int a, char * c)
{ fprintf(arqCodigo,"%3d: %5s %d,%d(%d) ",
          emitirLoc,op,r,a-(emitirLoc+1),cp);
  ++emitirLoc ;
  if (RastreiaCod) fprintf(arqCodigo,"\t%s",c) ;
  fprintf(arqCodigo,"\n") ;
}

```

```

    if (emitirLocAlta < emitirLoc) emitirLocAlta = emitirLoc ;
} /* emitirRM_Abs */

```

GERACODIGO.H

```

/*****
/* Arquivo: geracodigo.h
/* Interface das funções para codigo.c
/* Trabalho de final de curso - BSI - FARN
/* Fábio Henrique F. Silva
/*
/* Baseado no compilador TINY do livro Construção
/* de Compiladores - Principios e Práticas, autor
/* Kenneth C. Louden - Tradução 2004
*****/

#ifndef _GCOD_H_
#define _GCOD_H_

/* O procedimento geraCodigo gera o código para um arquivo de código
 * analisando a árvore sintática. O segundo parâmetro (arqCodigo) é o nome
 * do arquivo de código que é impresso como comentário no arquivo.
 */
void geraCodigo(NoArvore * arvoreSintatica, char * arqCodigo);

#endif

```

GERACODIGO.C

```

/*****
/* Arquivo: geracodigo.c
/* Gerador de código para o compilador
/* Trabalho de final de curso - BSI - FARN
/* Fábio Henrique F. Silva
/*
/* Baseado no compilador TINY do livro Construção
/* de Compiladores - Principios e Práticas, autor
/* Kenneth C. Louden - Tradução 2004
*****/

#include "globais.h"
#include "tabsim.h"
#include "codigo.h"
#include "geracodigo.h"

/* tmpDesloc é o deslocamento na memória para temps
   Ele é decrementado a cada vez que um temp é armazenado
   e incrementado quando ele é carregado denovo.

```

```

*/
static int tmpDesloc = 0;

/* prototipo para o gerador de código recursivo interno */
static void gCod (NoArvore * arvore);

/* Procedimento geraDeclaracao gera código para um nó declaração */
static void geraDeclaracao( NoArvore * arvore)
{ NoArvore * p1, * p2, * p3;
  int localSalvo1,localSalvo2,localAtual;
  int loc;
  switch (arvore->cInd.declaracao) {

    case cSe :
      if (RastreiaCod) emitirComentario("-> se") ;
      p1 = arvore->filho[0] ;
      p2 = arvore->filho[1] ;
      p3 = arvore->filho[2] ;
      /* gera código para expressão de testes */
      gCod(p1);
      localSalvo1 = emitirSalto(1) ;
      emitirComentario("se: salta para o senao a partir daqui");
      /* recurso na parte entao */
      gCod(p2);
      localSalvo2 = emitirSalto(1) ;
      emitirComentario("se: salta para o fim a partir daqui");
      localAtual = emitirSalto(0) ;
      emitirCopia(localSalvo1) ;
      emitirRM_Abs("JEQ",ac,localAtual,"se: salta para o senao");
      emitirRecuperar() ;
      /* recurso na parte senao */
      gCod(p3);
      localAtual = emitirSalto(0) ;
      emitirCopia(localSalvo2) ;
      emitirRM_Abs("LDA",cp,localAtual,"salta para o fim") ;
      emitirRecuperar() ;
      if (RastreiaCod) emitirComentario("<- se") ;
      break; /* se */

    case cRepetir:
      if (RastreiaCod) emitirComentario("-> repete") ;
      p1 = arvore->filho[0] ;
      p2 = arvore->filho[1] ;
      localSalvo1 = emitirSalto(0);
      emitirComentario("repete: salta após o corpo retornar para cá");
      /* gera o código para o corpo */
      gCod(p1);
      /* gera o código para teste */

```

```

gCod(p2);
emitirRM_Abs("JEQ",ac,localSalvo1,"repete: salta de volta para o corpo");
if (RastreiaCod) emitirComentario("<- repete");
break; /* repete */

case cEnquanto:
    if (RastreiaCod) emitirComentario("<-> enquanto");
    p1 = arvore->filho[0];
    p2 = arvore->filho[1];
    emitirComentario("enquanto: salta após o corpo retornar para cá");
    /* gera o código para teste */
    localSalvo1 = emitirSalto(0);
    gCod(p1);
    emitirRM_Abs("JEQ",ac,localSalvo1,"enquanto: salta de volta para o corpo");
    /* gera o código para o corpo */
    gCod(p2);
    if (RastreiaCod) emitirComentario("<- enquanto");
    break; /* enquanto */

case cAtribuicao:
    if (RastreiaCod) emitirComentario("<-> atribuição");
    /* gera o código para rhs */
    gCod(arvore->filho[0]);
    /* armazena o valor */
    loc = procurarDec(arvore->atributo.nome);
    emitirRM("ST",ac,loc,pg,"atribuição: armazena valor");
    if (RastreiaCod) emitirComentario("<- atribuição");
    break; /* atribuição */

case cLeia:
    emitirRO("IN",ac,0,0,"le um valor inteiro");
    loc = procurarDec(arvore->atributo.nome);
    emitirRM("ST",ac,loc,pg,"leia: armazena valor");
    break;

case cEscreva:
    /* gera o código da expressão a escrever */
    gCod(arvore->filho[0]);
    /* grava */
    emitirRO("OUT",ac,0,0,"escreve ac");
    break;

default:
    break;
}
} /* geraDeclaracao */

/* Procedimento geraExpressao gera código para uma expressão */
static void geraExpressao( NoArvore * arvore)
{ int loc;

```

```

NoArvore * p1, * p2;
switch (arvore->cInd.expressao) {

case cConstante :
    if (RastreiaCod) emitirComentario("-> Constante") ;
    /* gera código para carregar uma constante inteiro usando LDC */
    emitirRM("LDC",ac,arvore->atributo.val,0,"carrega constante");
    if (RastreiaCod) emitirComentario("<- Constante") ;
    break; /* cConstante */

case cId :
    if (RastreiaCod) emitirComentario("-> Id") ;
    loc = procurarDec(arvore->atributo.nome);
    emitirRM("LD",ac,loc,pg,"carrega valor id");
    if (RastreiaCod) emitirComentario("<- Id") ;
    break; /* cId */

case cOp :
    if (RastreiaCod) emitirComentario("-> Op") ;
    p1 = arvore->filho[0];
    p2 = arvore->filho[1];
    /* gera código para ac = argumento esquerdo */
    gCod(p1);
    /* gera o código para pegar o operando esquerdo */
    emitirRM("ST",ac,tmpDesloc--,pm,"op: pega a esquerda");
    /* gera o código para ac = operando direito */
    gCod(p2);
    /* carrega o operando esquerdo */
    emitirRM("LD",ac1,++tmpDesloc,pm,"op: carrega o operando esquerdo");
    switch (arvore->atributo.op) {
        case MAIS :
            emitirRO("ADD",ac,ac1,ac,"op +");
            break;
        case MENOS :
            emitirRO("SUB",ac,ac1,ac,"op -");
            break;
        case MULTIPLICA :
            emitirRO("MUL",ac,ac1,ac,"op *");
            break;
        case DIVISAO :
            emitirRO("DIV",ac,ac1,ac,"op /");
            break;
        case MENOR :
            emitirRO("SUB",ac,ac1,ac,"op <") ;
            emitirRM("JLT",ac,2,cp,"br se verdadeiro") ;
            emitirRM("LDC",ac,0,ac,"caso falso") ;
            emitirRM("LDA",cp,1,cp,"salto incondicional") ;
            emitirRM("LDC",ac,1,ac,"caso verdadeiro") ;
    }
}

```

```

        break;
    case IGUAL :
        emitirRO("SUB",ac,ac1,ac,"op =") ;
        emitirRM("JEQ",ac,2,cp,"br se verdadeiro");
        emitirRM("LDC",ac,0,ac,"caso falso") ;
        emitirRM("LDA",cp,1,cp,"salto incondicional") ;
        emitirRM("LDC",ac,1,ac,"caso verdadeiro") ;
        break;
    case MAIOR :
        emitirRO("SUB",ac,ac1,ac,"op =") ;
        emitirRM("JGT",ac,2,cp,"br se verdadeiro");
        emitirRM("LDC",ac,0,ac,"caso falso") ;
        emitirRM("LDA",cp,1,cp,"salto incondicional") ;
        emitirRM("LDC",ac,1,ac,"caso verdadeiro") ;
        break;
    case MENORIGUAL :
        emitirRO("SUB",ac,ac1,ac,"op <=") ;
        emitirRM("JLE",ac,2,cp,"br se verdadeiro");
        emitirRM("LDC",ac,0,ac,"caso falso") ;
        emitirRM("LDA",cp,1,cp,"salto incondicional") ;
        emitirRM("LDC",ac,1,ac,"caso verdadeiro") ;
        break;
    case MAIORIGUAL :
        emitirRO("SUB",ac,ac1,ac,"op >=") ;
        emitirRM("JGE",ac,2,cp,"br se verdadeiro");
        emitirRM("LDC",ac,0,ac,"caso falso") ;
        emitirRM("LDA",cp,1,cp,"salto incondicional") ;
        emitirRM("LDC",ac,1,ac,"caso verdadeiro") ;
        break;
    case DIFERENTE :
        emitirRO("SUB",ac,ac1,ac,"op !=") ;
        emitirRM("JNE",ac,2,cp,"br se verdadeiro");
        emitirRM("LDC",ac,0,ac,"caso falso") ;
        emitirRM("LDA",cp,1,cp,"salto incondicional") ;
        emitirRM("LDC",ac,1,ac,"caso verdadeiro") ;
        break;

    default:
        emitirComentario("ERRO: operador inválido");
        break;
} /* caso op */
if (RastreiaCod) emitirComentario("<- Op") ;
break; /* cOp */

default:
    break;
}
} /* geraExpressao */

```

```

/* Procedimento gCod gera código recursivamente pela árvore transversa */
static void gCod( NoArvore * arvore)
{ if (arvore != NULL)
  { switch (arvore->cindnoh) {
    case cDeclaracao:
      geraDeclaracao(arvore);
      break;
    case cExpressao:
      geraExpressao(arvore);
      break;
    default:
      break;
  }
  gCod(arvore->primo);
}
}

/*****
/* a função principal do gerador de código */
*****/
/* O procedimento geraCodigo gera código para um arquivo atravessando a
* a árvore sintática, o segundo parâmetro (codArquivo) é o nome do arquivo do código
* e é usado para gravar o nome do arquivo como um comentário.
*/
void geraCodigo(NoArvore * arvoreSintatica, char * codArquivo)
{ char * s = malloc(strlen(codArquivo)+7);
  strcpy(s,"Arquivo: ");
  strcat(s,codArquivo);
  emitirComentario("Compilação LIPP para a máquina virtual");
  emitirComentario(s);
  /* gera o prólogo padrão */
  emitirComentario("Prólogo padrão:");
  emitirRM("LD",pm,0,ac,"carrega o maxendereço da localização 0");
  emitirRM("ST",ac,0,ac,"limpa a localização 0");
  emitirComentario("Fim do prólogo padrão.");
  /* gera o código para o programa LIPP */
  gCod(arvoreSintatica);
  /* fim */
  emitirComentario("Fim da execução.");
  emitirRO("HALT",0,0,0,"");
}

```

MAIN.C

```

/*****
/* Arquivo: main.c */
/* Programa principal para o compilador LIPP */

```

```

/* Trabalho de final de curso - BSI - FARN */
/* Fábio Henrique F. Silva */
/* */
/* Baseado no compilador TINY do livro Construção */
/* de Compiladores - Principios e Práticas, autor */
/* Kenneth C. Louden - Tradução 2004 */
/*****/

#include "globais.h"
#include "util.h"
/* extern int yydebug;
yydebug = 1; */
/* SEM_SINTAXE = SIM gera só o analisador léxico */
#define SEM_SINTAXE NAO
/* SEM_SEMANTICA = SIM gera só o analisador sintático */
#define SEM_SEMANTICA NAO
/* SEM_CODIGO = SIM gera um compilador sem gerar o código */
#define SEM_CODIGO NAO
#if SEM_SINTAXE
#include "lexico.h"
#else
#include "sintaxe.h"
#endif
#if !SEM_SEMANTICA
#include "semantica.h"
#endif
#if !SEM_CODIGO
#include "geracodigo.h"
#endif
#endif
#endif

/* aloca as variáveis globais */
int numLinha = 0;
FILE * arqFonte;
FILE * arqSaida;
FILE * arqCodigo;

/* aloca e define as opções de rastreamento */
int EcoaFonte = NAO;
int RastreiaLex = NAO;
int RastreiaSin = NAO;
int RastreiaSem = NAO;
int RastreiaCod = SIM;

int Erro = NAO;

main( int argc, char * argv[] )
{ NoArvore * arvoreSintatica;
  char pgm[120]; /* nome dos arquivos arqFonte arqCodigo */

```

```

if (argc != 2)
{ fprintf(stderr,"use: %s <nome_arquivo>\n",argv[0]);
  exit(1);
}
strcpy(pgm,argv[1]) ;
if (strchr (pgm, '.') == NULL)
  strcat(pgm, ".lip");
arqFonte = fopen(pgm,"r");
if (arqFonte==NULL)
{ fprintf(stderr,"O arquivo %s não foi encontrado !\n",pgm);
  exit(1);
}
arqSaida = stdout; /* exibe o arqSaida */
fprintf(arqSaida,"\nCOMPILAÇÃO LIPP: %s\n",pgm);
#if SEM_SINTAXE
  fprintf(stderr,"Iniciando analise lexica !\n");
  while (lerSimbolo()!=ENDFILE);
#else
  fprintf(stderr,"Iniciando analise sintatica !\n");
  arvoreSintatica = sintaxe();
  if (RastreiaSin) {
    fprintf(arqSaida,"\nÁrvore sintática:\n");
    imprimirArvore(arvoreSintatica);
  }
#endif
fprintf(stderr,"Iniciando analise semantica !\n");
if (! Erro)
{ if (RastreiaSem) fprintf(arqSaida,"\nCriando a tabela de símbolos...\n");
  construirTabSim(arvoreSintatica);
  if (RastreiaSem) fprintf(arqSaida,"\nChecando tipos...\n");
  checaTipo(arvoreSintatica);
  if (RastreiaSem) fprintf(arqSaida,"\nChecagem de tipos concluída.\n");
}
if (! Erro)
{ char * codArquivo;
  int fnlen = strlen(pgm);
  codArquivo = (char *) calloc(fnlen+4, sizeof(char));
  strncpy(codArquivo,pgm,fnlen);
  strcat(codArquivo, ".mv");
  arqCodigo = fopen(codArquivo,"w");
  if (arqCodigo == NULL)
  { printf("Não foi possível abrir o %s\n",codArquivo);
    exit(1);
  }
  geraCodigo(arvoreSintatica,codArquivo);
  fclose(arqCodigo);
}
fclose(arqFonte);

```

```

return 0;
}

```

SEMANTICA.H

```

/*****
/* Arquivo: semantica.h
/* Interface para o analisador semântico
/* Trabalho de final de curso - BSI - FARN
/* Fábio Henrique F. Silva
/*
/* Baseado no compilador TINY do livro Construção
/* de Compiladores - Principios e Práticas, autor
/* Kenneth C. Louden - Tradução 2004
*****/

#ifndef _SEMANTICA_H_
#define _SEMANTICA_H_

/* A função construirTabSimbolos contrói a tabela de símbolos */
void construirTabSim(NoArvore *);

/* O procedimento checaTipo executa uma checagem de tipos em pós-ordem na
 * árvore sintática
 */
void checaTipo(NoArvore *);

#endif

```

SEMANTICA.C

```

/*****
/* Arquivo: semantica.c
/* Analisador semantico
/* Trabalho de final de curso - BSI - FARN
/* Fábio Henrique F. Silva
/*
/* Baseado no compilador TINY do livro Construção
/* de Compiladores - Principios e Práticas, autor
/* Kenneth C. Louden - Tradução 2004
*****/

#include "globais.h"
#include "tabsim.h"
#include "semantica.h"

/* localização das variáveis na memória */
static int local = 0;

```

```

/* O procedimento caminha é uma rotina para uma árvore de sintaxe
* recursiva genérica. Ele aplica preProc em preordem e posProc em posordem
* na árvore apontada por t
*/

```

```

static void caminha( NoArvore * t,
                    void (* preProc) (NoArvore *),
                    void (* posProc) (NoArvore *) )
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXFILHOS; i++)
        caminha(t->filho[i],preProc,posProc);
    }
    posProc(t);
    caminha(t->primo,preProc,posProc);
  }
}

```

```

/* procNula não faz nada
*/

```

```

static void procNula(NoArvore * t)
{ if (t==NULL) return;
  else return;
}

```

```

/* O procedimento inserirNo insere identificadores armazenados em t
* na tabela de símbolos.
*/

```

```

static void inserirNo( NoArvore * t)
{ switch (t->cindnoh)
  { case cDeclaracao:
    switch (t->cInd.declaracao)
    { case cAtribuicao:
      case cLeia:
        if (procurarDec(t->atributo.nome) == -1)
          /* se não estiver na tabela, insere-a */
          inserirDec(t->atributo.nome,t->numLinha,local++);
        else
          /* se já estiver na tabela acrescenta apenas o número da linha */
          inserirDec(t->atributo.nome,t->numLinha,0);
        break;
      default:
        break;
    }
    break;
  case cExpressao:
    switch (t->cInd.expressao)

```

```

{ case cId:
  if (procurarDec(t->atributo.nome) == -1)
    /* se nao estiver na tabela, insere-a */
    inserirDec(t->atributo.nome,t->numLinha,local++);
  else
    /* se já estiver na tabela acrescenta apenas o número da linha */
    inserirDec(t->atributo.nome,t->numLinha,0);
  break;
default:
  break;
}
break;
default:
  break;
}
}

/* A função construirTabSim contrói a tabela de símbolos
 * pela préordem transversa da árvore de sintaxe
 */
void construirTabSim(NoArvore * arvoreSintatica)
{ caminha(arvoreSintatica,inserirNo,procNula);
  if (RastreiaSem)
  { fprintf(arqSaida,"\nTabela de símbolos:\n\n");
    imprimirTabSim(arqSaida);
  }
}

static void tipoErro(NoArvore * t, char * mensagem)
{ fprintf(arqSaida,"Erro de tipo na linha %d: %s\n",t->numLinha,mensagem);
  Erro = SIM;
}

/* O procedimento checaNo executa uma checagem de tipos em um nó da árvore */
static void checaNo(NoArvore * t)
{ switch (t->cindnoh)
  { case cExpressao:
    switch (t->cInd.expressao)
    { case cOp:
      if ((t->filho[0]->tipo != Integer) ||
          (t->filho[1]->tipo != Integer))
        tipoErro(t,"Op aplicada a um não inteiro");
      if ((t->atributo.op == IGUAL) || (t->atributo.op == MENOR) ||
          (t->atributo.op == MENORIGUAL) || (t->atributo.op == MAIOR) ||
          (t->atributo.op == MAIORIGUAL) || (t->atributo.op == DIFERENTE))
        t->tipo = Boolean;
      else
        t->tipo = Integer;
    }
  }
}

```

```

        break;
    case cConstante:
    case cId:
        t->tipo = Integer;
        break;
    default:
        break;
    }
    break;
case cDeclaracao:
    switch (t->cInd.declaracao)
    { case cSe:
        if (t->filho[0]->tipo == Integer)
            tipoErro(t->filho[0], "O teste SE não é booleano");
        break;
        case cAtribuicao:
            if (t->filho[0]->tipo != Integer)
                tipoErro(t->filho[0], "Atribuição de um valor não inteiro");
            break;
        case cEscreva:
            if (t->filho[0]->tipo != Integer)
                tipoErro(t->filho[0], "Escrita de um valor não inteiro");
            break;
        default:
            break;
    }
    break;
default:
    break;
}
}
}

```

/* O procedimento checaTipo executa uma checagem de tipos em pósordem
 * na árvore sintática
 */

```

void checaTipo(NoArvore * arvoreSintatica)
{
    caminha(arvoreSintatica, procNula, checaNo);
}

```

TABSIM.H

```

/*****/
/* Arquivo: tabsim.h */
/* Interface para a tabela de símbolos */
/* Trabalho de final de curso - BSI - FARN */
/* Fábio Henrique F. Silva */

```

```

/*                                                                    */
/* Baseado no compilador TINY do livro Construção                    */
/* de Compiladores - Principios e Práticas, autor                    */
/* Kenneth C. Louden - Tradução 2004                                  */
/*****                                                                    */

#ifndef _TABSIM_H_
#define _TABSIM_H_

/* O procedimento inserirDec insere números de linhas e
 * localizações de memória na tabela de símbolos
 * local = localização de memória inserida apenas uma vez
 * caso contrário será ignorada
 */
void inserirDec( char * nome, int numLinha, int local );

/* A função procurarDec retorna a localização de memória
 * de uma variável ou -1 se não encontrá-la.
 */
int procurarDec ( char * nome );

/* O procedimento imprimirTabSim imprime a tabela de símbolos no arquivo
 * de saída.
 */
void imprimirTabSim(FILE * arqSaida);

#endif

```

TABSIM.C

```

/*****                                                                    */
/* Arquivo: tabsim.c                                                    */
/* Tabela de símbolos para a LIPP                                       */
/* Trabalho de final de curso - BSI - FARN                             */
/* Fábio Henrique F. Silva                                              */
/*                                                                    */
/* Baseado no compilador TINY do livro Construção                    */
/* de Compiladores - Principios e Práticas, autor                    */
/* Kenneth C. Louden - Tradução 2004                                  */
/*****                                                                    */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tabsim.h"

```

```

/* TAMANHO é o tamanho da tabela hash */
#define TAMANHO 211

/* SHIFT é a potência de dois usada como multiplicadora na
função hash */
#define SHIFT 4

/* função hash */
static int hash ( char * chave )
{ int temp = 0;
  int i = 0;
  while (chave[i] != '\0')
  { temp = ((temp << SHIFT) + chave[i]) % TAMANHO;
    ++i;
  }
  return temp;
}

/* A lista do número de linhas do código fonte no qual a variável é referenciada */
typedef struct ListaLinhasReg
{ int numLinhas;
  struct ListaLinhasReg * proximo;
} * ListaLinhas;

/* O registro para cada variável no balde, incluindo nome, localização na memória
* e a lista do número de linhas na qual elas aparecem no código fonte.
*/
typedef struct BaldeReg
{ char * nome;
  ListaLinhas linhas;
  int memloc ; /* localização da variável na memória */
  struct BaldeReg * proximo;
} * Balde;

/* tabela hash */
static Balde tabelaHash[TAMANHO];

/* O procedimento inserirDec insere o número da linha e a localização de
* memória na tabela de símbolos loc = localização de memória é incluída
* apenas uma vez.
*/
void inserirDec( char * nome, int numLinhas, int loc )
{ int h = hash(nome);
  Balde l = tabelaHash[h];
  while ((l != NULL) && (strcmp(nome,l->nome) != 0))
    l = l->proximo;
  if (l == NULL) /* a variável ainda não está na tabela */
  { l = (Balde) malloc(sizeof(struct BaldeReg));

```

```

l->nome = nome;
l->linhas = (ListaLinhas) malloc(sizeof(struct ListaLinhasReg));
l->linhas->numLinhas = numLinhas;
l->memloc = loc;
l->linhas->proximo = NULL;
l->proximo = tabelaHash[h];
tabelaHash[h] = l; }
else /* a variável está na tabela, inclui apenas o número da linha */
{ ListaLinhas t = l->linhas;
  while (t->proximo != NULL) t = t->proximo;
  t->proximo = (ListaLinhas) malloc(sizeof(struct ListaLinhasReg));
  t->proximo->numLinhas = numLinhas;
  t->proximo->proximo = NULL;
}
} /* inserirDec */

/* A função procurarDec retorna a localização de memória
 * de uma variável ou -1 se não a encontrar.
 */
int procurarDec ( char * nome )
{ int h = hash(nome);
  Balde l = tabelaHash[h];
  while ((l != NULL) && (strcmp(nome,l->nome) != 0))
    l = l->proximo;
  if (l == NULL) return -1;
  else return l->memloc;
}

/* O procedimento imprimirTabSim imprime uma listagem formatada do
 * conteúdo da tabela de símbolos no arquivo de saída
 */
void imprimirTabSim(FILE * arqSaida)
{ int i;
  fprintf(arqSaida,"Variável    Local    Número linhas\n");
  fprintf(arqSaida,"-----    -----    ----- \n");
  for (i=0;i<TAMANHO;++i)
  { if (tabelaHash[i] != NULL)
    { Balde l = tabelaHash[i];
      while (l != NULL)
      { ListaLinhas t = l->linhas;
        fprintf(arqSaida,"%-14s ",l->nome);
        fprintf(arqSaida,"%-8d ",l->memloc);
        while (t != NULL)
        { fprintf(arqSaida,"%4d ",t->numLinhas);
          t = t->proximo;
        }
        fprintf(arqSaida,"\n");
        l = l->proximo;
      }
    }
  }
}

```

```

    }
  }
}
} /* imprimirTabSim */

```

UTIL.H

```

/*****
/* Arquivo: util.h */
/* Funções utilitárias para o compilador LIPP */
/* Trabalho de final de curso - BSI - FARN */
/* Fábio Henrique F. Silva */
/*
/* Baseado no compilador TINY do livro Construção
/* de Compiladores - Principios e Práticas, autor
/* Kenneth C. Louden
*****/

#ifndef _UTIL_H_
#define _UTIL_H_

/* Procedimento imprimirSimbolo imprime um símbolo
 * e o lexema para o arquivo de saída
 */
void imprimirSimbolo( TipoSimbolo, const char* );

/* A função novoNoDeclara cria um novo nó para declarações na árvore de sintaxe */
NoArvore * novoNoDeclara(cIndDeclara);

/* A função novoNoExp cria um novo nó para expressões na árvore de sintaxe */
NoArvore * novoNoExp(cIndExpressao);

/* A função copiaCaracteres aloca e faz uma nova cópia de uma cadeia
 * caracteres existente
 */
char * copiaCaracteres( char * );

/* O procedimento imprimirArvore imprime uma árvore sintática no arquivo de saída
 * usando indentações para indicar subárvores.
 */
void imprimirArvore( NoArvore * );

#endif

```

UTIL.C

```

/*****
/* Arquivo: util.c */

```

```

/* Funções utilitárias para uso com a LIPP      */
/* Trabalho de final de curso - BSI - FARN      */
/* Fábio Henrique F. Silva                      */
/* Baseado nos exemplo do livro                 */
/* Construção de Compiladores: Principios e Práticas*/
/* Autor: Kenneth C. Louden                    */
/*****

```

```

#include "globais.h"
#include "util.h"

```

```

/* Procedimento imprimirSimbolo imprime um símbolo
 * e o lexema para o arquivo de saída
 */

```

```

void imprimirSimbolo( TipoSimbolo simbolo, const char* simboloLiteral )

```

```

{ switch (simbolo)
{ case SE:
  case ENTAO:
  case SENAO:
  case INICIO:
  case FIM:
  case ENQUANTO:
  case FACA:
  case LEIA:
  case REPETIR:
  case ATE:
  case ESCREVA:
    fprintf(arqSaida,
      "Palavra reservada: %s\n",simboloLiteral);
    break;
  case ATRIBUICAO: fprintf(arqSaida,"<\n"); break;
  case MAIOR: fprintf(arqSaida,">\n"); break;
  case MAIORIGUAL: fprintf(arqSaida,">=\n"); break;
  case MENORIGUAL: fprintf(arqSaida,"<=\n"); break;
  case DIFERENTE: fprintf(arqSaida,"!=\n"); break;
  case MENOR: fprintf(arqSaida,"<\n"); break;
  case IGUAL: fprintf(arqSaida,"=\n"); break;
  case PARESQUERDO: fprintf(arqSaida,"(\n"); break;
  case PARDIREITO: fprintf(arqSaida,")\n"); break;
  case PONTOVIRGULA: fprintf(arqSaida,";\n"); break;
  case MAIS: fprintf(arqSaida,"+\n"); break;
  case MENOS: fprintf(arqSaida,"-\n"); break;
  case MULTIPLICA: fprintf(arqSaida,"*\n"); break;
  case DIVISAO: fprintf(arqSaida,"/\n"); break;
  case ENDFILE: fprintf(arqSaida,"EOF\n"); break;
  case NUMERO:
    fprintf(arqSaida,
      "NUMERO, val= %s\n",simboloLiteral);

```

```

    break;
case ID:
    fprintf(arqSaida,
        "Identificador: %s\n", simboloLiteral);
    break;
case ERRO:
    fprintf(arqSaida,
        "ERRO: %s\n", simboloLiteral);
    break;
default: /* executado se o símbolo não for reconhecido */
    fprintf(arqSaida, "Símbolo inválido: %d\n", simbolo);
}
}

```

/* A função novoNoDeclara cria um novo nó para declarações na árvore de sintaxe */

```

NoArvore * novoNoDeclara(cIndDeclara cInd)
{ NoArvore * noDeclaracao = (NoArvore *) malloc(sizeof(NoArvore));
  int i;
  if (noDeclaracao==NULL)
    fprintf(arqSaida, "Erro de memória na linha %d\n", numLinha);
  else {
    for (i=0; i<MAXFILHOS; i++) noDeclaracao->filho[i] = NULL;
    noDeclaracao->primo = NULL;
    noDeclaracao->cindnoh = cDeclaracao;
    noDeclaracao->cInd.declaracao = cInd;
    noDeclaracao->numLinha = numLinha;
  }
  return noDeclaracao;
}

```

/* A função novoNoExp cria um novo nó para expressões na árvore de sintaxe */

```

NoArvore * novoNoExp(cIndExpressao cInd)
{ NoArvore * noExpressao = (NoArvore *) malloc(sizeof(NoArvore));
  int i;
  if (noExpressao==NULL)
    fprintf(arqSaida, "Erro de memória na linha %d\n", numLinha);
  else {
    for (i=0; i<MAXFILHOS; i++) noExpressao->filho[i] = NULL;
    noExpressao->primo = NULL;
    noExpressao->cindnoh = cExpressao;
    noExpressao->cInd.expressao = cInd;
    noExpressao->numLinha = numLinha;
    noExpressao->tipo = Void;
  }
  return noExpressao;
}

```

```

/* A função copiaCaracteres aloca e faz uma nova cópia de uma cadeia
 * caracteres existente
 */
char * copiaCaracteres(char * caracteres)
{ int n;
  char * tmp;
  if (caracteres==NULL) return NULL;
  n = strlen(caracteres)+1;
  tmp = malloc(n);
  if (tmp==NULL)
    fprintf(arqSaida,"Erro de memória na linha %d\n",numLinha);
  else strcpy(tmp,caracteres);
  return tmp;
}

/* A variável qtdEspacos é usada por imprimirArvore para armazenar a quantidade
 * de espaços na indentação
 */
static qtdEspacos = 0;

/* macros para aumentar ou diminuir a indentação */
#define INDENTA qtdEspacos+=2
#define RECUA qtdEspacos-=2

/* imprimirEspacos indenta imprimindo espaços */
static void imprimirEspacos(void)
{ int i;
  for (i=0;i<qtdEspacos;i++)
    fprintf(arqSaida," ");
}

/* O procedimento imprimirArvore imprime uma árvore sintática no arquivo de saída
 * usando indentações para indicar subárvores.
 */
void imprimirArvore( NoArvore * arvore )
{ int i;
  INDENTA;
  while (arvore != NULL) {
    imprimirEspacos();
    if (arvore->cindnoh==cDeclaracao)
      { switch (arvore->cInd.declaracao) {
        case cSe:
          fprintf(arqSaida,"Se\n");
          break;
        case cEnquanto:
          fprintf(arqSaida,"Enquanto\n");
          break;

```

```

    case cAtribuicao:
        fprintf(arqSaida,"Atribuição para: %s\n",arvore->atributo.nome);
        break;
    case cLeia:
        fprintf(arqSaida,"Leia: %s\n",arvore->atributo.nome);
        break;
    case cEscreva:
        fprintf(arqSaida,"Escreva\n");
        break;
    case cRepetir:
        fprintf(arqSaida,"Repetir\n");
        break;
    default:
        fprintf(arqSaida,"Nó Expressão Inválido cind\n");
        break;
}
}
else if (arvore->cindnoh==cExpressao)
{ switch (arvore->cInd.expressao) {
    case cOp:
        fprintf(arqSaida,"Op: ");
        imprimirSimbolo(arvore->atributo.op,"\\0");
        break;
    case cConstante:
        fprintf(arqSaida,"Const: %d\n",arvore->atributo.val);
        break;
    case cId:
        fprintf(arqSaida,"Id: %s\n",arvore->atributo.nome);
        break;
    default:
        fprintf(arqSaida,"Nó Expressão Inválido cind\n");
        break;
    }
}
else fprintf(arqSaida,"Nó inválido cind\n");
for (i=0;i<MAXFILHOS;i++)
    imprimirArvore(arvore->filho[i]);
arvore = arvore->primo;
}
RECUA;
}

```

GLOBALIS.H

```

/*****
/* Arquivo: globais.h */
/* Tipos globais e variáveis para o compilador LIPP */
/* Trabalho de final de curso - BSI - FARN */

```

```

/* Fábio Henrique F. Silva */
/*
/* Baseado no compilador TINY do livro Construção
/* de Compiladores - Principios e Práticas, autor
/* Kenneth C. Louden
/*
/*****/

#ifndef _GLOBAIS_H_
#define _GLOBAIS_H_

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#ifndef YYPARSER

/* Inclui a tabela de símbolos */
#include "simbolos.h"

#define ENDFILE 0
#define FIMARQ 0

#endif

#ifndef FALSE
#define FALSE 0
#endif

#ifndef TRUE
#define TRUE 1
#endif

#ifndef NAO
#define NAO 0
#endif

#ifndef SIM
#define SIM 1
#endif

/* MAXRES = número de palavras reservadas */
#define MAXRES 9

typedef int TipoSimbolo;

extern FILE* arqFonte; /* arquivo com o código fonte */
extern FILE* arqSaida; /* arquivo com informações diversas */

```

```

extern FILE* arqCodigo; /* arquivo com o código para a máquina virtual */

extern int numLinha; /* número de linhas no fonte */

/*****
/*****  Árvore de sintaxe  *****/
/*****/

typedef enum {cDeclaracao,cExpressao} cIndNoh;
typedef enum {cSe,cEnquanto,cAtribuicao,cLeia,cEscreva,cRepetir} cIndDeclara;
typedef enum {cOp,cConstante,cId} cIndExpressao;

/* TipoExp é usado para checagem de tipos */
typedef enum {Void,Integer,Boolean} TipoExp;

#define MAXFILHOS 3

typedef struct noArvore
{ struct noArvore * filho[MAXFILHOS];
  struct noArvore * primo;
  int numLinha;
  cIndNoh cindnoh;
  union { cIndDeclara declaracao; cIndExpressao expressao;} cInd;
  union { TipoSimbolo op;
          int val;
          char * nome; } atributo;
  TipoExp tipo;
} NoArvore;

/*****
/*****  Opções para rastreamento  *****/
/*****/

/* EcoaFonte = SIM ecoa o fonte no arquivo de
 * saída com o número das linhas.
 */
extern int EcoaFonte;

/* RastreiaLex = SIM grava informações no arquivo de
 * saída sobre cada símbolo que é reconhecido.
 */
extern int RastreiaLex;

/* RastreiaSin = SIM grava a Árvore sintática no
 * arquivo de saída na forma linearizada indentando os filhos
 */
extern int RastreiaSin;

```

```
/* RastreiaTab = SIM grava informações sobre a tabela de
 * símbolos no arquivo de saída.
 */
extern int RastreiaSem;

/* RastreiaCod = SIM insere comentários no arquivo de código
 * gerado.
 */
extern int RastreiaCod;

/* Erro = SIM evita passos adicionais se ocorrer um erro */
extern int Erro;
#endif
```