Luiz Thiago Silva

DESENVOLVIMENTO DE UM ANALISADOR DE CÓDIGO ANSI-C

Orientador: Prof. MSc. João Carlos Giacomin

LAVRAS
MINAS GERAIS – BRASIL
2006

Luiz Thiago Silva

DESENVOLVIMENTO DE UM ANALISADOR DE CÓDIGO ANSI-C

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do curso de Pós-Graduação *Lato Sensu* Administração em Redes Linux, para a obtenção do título de especialista.

Orientador: Prof. MSc. João Carlos Giacomin

LAVRAS
MINAS GERAIS – BRASIL
2006

Luiz Thiago Silva

DESENVOLVIMENTO DE UM ANALISADOR DE CÓDIGO ANSI-C

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do curso de Pós-Graduação *Lato Sensu* Administração em Redes Linux, para a obtenção do título de especialista.

Aprovada em ___ de _____ de_____

_____
Prof. MSc. Douglas Machado Tavares

_____
Prof. MSc. Fernando Cortez Sica

_____
Prof. MSc. João Carlos Giacomin
(Orientador)

LAVRAS
MINAS GERAIS – BRASIL
2006

Agradeço aos meus pais pelo apoio e compreensão, aos meus irmãos e minha namorada Paula.

RESUMO

Esse trabalho aborda o desenvolvimento e análise do projeto de um analisador de códigos, que visa oferecer ao usuário iniciante da linguagem C, um diálogo amigável, reportando as mensagens de erro de programação no idioma português. O pré-compilador é uma ferramenta auxiliar no ensino da linguagem C. É importante destacar, que o pré-compilador trata alguns tipos de erros comumente encontrados nesta linguagem, e não todos os possíveis erros. Dessa forma, espera-se que este trabalho possa contribuir para a comunidade Linux e especialmente para o ensino da linguagem C.

SUMÁRIO

LISTA DE FIGURAS

1 - INTRODUÇÃO

Em diversas linguagens de programação, muitos alunos e programadores iniciantes na linguagem C, têm dificuldade em saber se estão utilizando somente funções do padrão ANSI-C. Traduz-se, portanto, em dificuldades para entender e interpretar as mensagens de erros exibidas durante a compilação de algum programa, principalmente pelo fato dos diversos compiladores disponíveis hoje exibirem tais mensagens no idioma inglês.

A importância de se utilizar apenas funções do padrão ANSI-C faz com que os programas se tornam portáveis, isto é, se tornam independentes de plataforma, podendo um mesmo programa ser compilado em diferentes computadores e sobre diferentes sistemas operacionais.

Esse trabalho tem como proposta a análise do projeto de desenvolvimento de um analisador de códigos para linguagem C, onde foram levadas em consideração as principais finalidades que são:

- O nome das bibliotecas e funções indicando suas relações com o padrão ANSI-C;
- Os protótipos, parâmetros e chamadas das funções implementadas no programa-fonte;
- A ambigüidade de variáveis, tipos e suas localizações

dentro do escopo do programa e,

- Erros de sintaxe, a falta e/ou excesso de parênteses, chaves e colchetes, variáveis utilizadas sem a prévia declaração e falta de ponto e vírgula no final das linhas de comando.

O projeto analisado visa propor um ambiente de pré-compilação, que ofereça ao usuário iniciante na linguagem de programação C, um diálogo mais amigável, reportando mensagens de erros no idioma português, estas provenientes de diversas análises realizadas no programa fonte.

Aho (1995) afirma que 60% dos erros encontrados em um código fonte de um programa criado na linguagem Pascal são erros de pontuação, 20% de operadores e operandos, 15% de palavras chave e 5% outros tipos de erros. A partir destas estatísticas pode-se considerar que grande parte dos erros encontrados em código fonte na linguagem C, são erros simples de pontuação.

Tendo como base este quadro, foi proposto o desenvolvimento de um pré-compilador padrão ANSI-C que gere mensagens de erro no idioma português. Este programa foi desenvolvido utilizando-se apenas funções do padrão ANSI-C, de forma que pudesse ser compilado para diferentes plataformas, como Linux e Windows.

De acordo com os autores do projeto, torna-se relevante no ensino desta linguagem, quando os alunos têm os primeiros contatos com C, e começam a desenvolver pequenos programas.

Este trabalho foi estruturado em seis capítulos, além da introdução

e considerações finais. No segundo capítulo, são apresentados os apectos históricos relevantes da Linguagem C.

O terceiro capítulo traz os fundamentos e conceitos sobre a Análise Léxica e a Análise Sintática e faz uma abordagem à última das etapas das análises que é a Semântica.

O quarto capítulo apresenta as Linguagens Formais e Autômatos, pela qual foi planejada e projetada a base do analisador de código analisado.

O quinto capítulo faz uma abordagem sobre a estrutura adotada no projeto do analisador em questão.

No sexto capitulo são apresentados os resultados obtidos e a seguir a conclusão do trabalho.

2 - LINGUAGEM DE PROGRAMAÇÃO C

A linguagem C é uma linguagem genérica de alto nível. Foi desenvolvida por programadores para programadores tendo como meta características de flexibilidade e portabilidade.

C é uma linguagem que nasceu juntamente com o aparecimento da teoria de linguagem estruturada e do computador pessoal. Assim tornou-se rapidamente popular entre os programadores.

C é uma linguagem vitoriosa como ferramenta na programação de qualquer tipo de sistema (sistemas operacionais, planilhas eletrônicas, processadores de texto, gerenciadores de banco de dados, processadores gráficos, sistemas de transmissão de dados, para solução de problemas de engenharia ou física etc.). Foi utilizada para desenvolver o sistema operacional UNIX (antigamente desenvolvido em Assembly), e para desenvolver novas linguagens, entre elas a linguagem C++ e Java (SCHILDT, 1996).

A linguagem C foi desenvolvida a partir da necessidade de escrever programas que utilizassem as potencialidades da linguagem de máquina, mas de uma forma mais simples e portátil do que esta.

2.1 - Características da Linguagem C

A linguagem C pertence a uma família de linguagens cujas características são:

- Portabilidade entre máquinas e sistemas operacionais;
- Pequeno tamanho da sua definição;
- Subdivisão do código e grande utilização de funções;
- Alguma conversão automática entre tipos de dados;
- Modularidade;
- Programas estruturados;
- Disponibilidade de operadores para programação de baixo nível;
- Total interação com o sistema operacional;
- Compilação separada;
- Utilização fácil e extensa de apontadores para aceder memória, vetores, estruturas e funções;
- Possibilidade de usar construções de alto nível;
- Possibilidade de utilizar operadores de baixo nível;
- Produção de código executável eficiente;
- Disponibilidade de compiladores em praticamente todos os sistemas de computação;
- Confiabilidade, regularidade, simplicidade, facilidade de uso;
- Pode ser usada para os mais variados propósitos.

2.2 - Utilização

Atualmente, nos USA, C é a linguagem mais utilizada pelos programadores, por permitir, dadas suas características, a escrita de programas típicos do Assembler, BASIC, COBOL e Clipper, sempre com maior eficiência e portabilidade, como podemos constatar pelos exemplos abaixo relacionados:

- Sistema Operacional: UNIX (Sistema Operacional executável em micro computadores e em mainframes).
- Montadores: Clipper.
- Planilhas: 1,2,3 e Excel (A planilha eletrônica com maior volume de vendas mundial).
- Banco de Dados: dBase III, IV e Access.
- InfoStar: Editor de Texto muito utilizado nos USA no Sistema Operacional UNIX.
- Utilitários: FormTool (Editor de formulário).
- Aplicações Gráficas: Efeitos Especiais de filmes com Star Trek e Star War.
- Linguagens como o Power Builder e o Visual Basic.

No Brasil, C foi utilizada por empresas especializadas na elaboração de vinhetas e outros efeitos especiais.

3 - ASPECTOS DE UM COMPILADOR

Compiladores são programas que convertem expressões de uma determinada linguagem de programação para a linguagem de montagem.

Segundo Price e Toscani (2001) compiladores são tradutores que mapeiam programas escritos em linguagem de alto nível para linguagem simbólica ou linguagem de máquina.

Em sua estrutura o compilador possui duas fases: a fase de análise, como análise léxica, sintática e semântica, as quais são de grande importância para o reconhecimento de linguagem e verificação de erros (PRICE e TOSCANI, 2001); e a fase de síntese que é responsável pela geração de código intermediário e sua otimização. Como esse trabalho se limita à identificação de erros em código fonte não será abordada a geração de código intermediário e a otimização de código.

O funcionamento básico de um compilador consiste em receber uma codificação correspondente de um determinado programa, verificar possíveis erros léxicos, sintáticos e semânticos e gerar uma nova codificação correspondente ao programa objeto. Este apenas será gerado quando não houver erros pendentes. O novo programa torna-se independente visto que não precisa do compilador para ser executado.

Esse processo pode ser visualizado na Figura 1.

Figura 1 – **Processo básico de um compilador**[1]

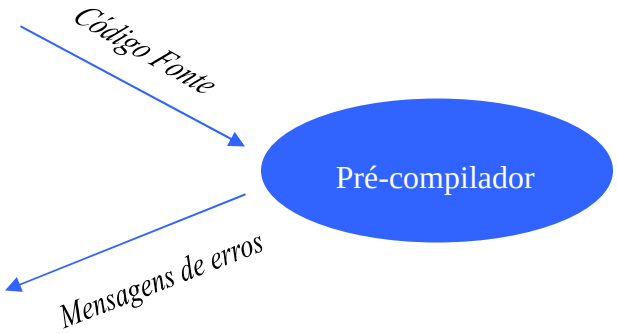


Figura 2 – **Processo básico do pré-compilador**

[1] JOSE NETO, João. *Introdução à compilação*. Rio de Janeiro: LTC, 1987. 222 p.

Pelo projeto proposto não haveria a geração do programa objeto, com isso a estrutura básica do analisador de códigos, que pode ser vista na Figura 2, se difere em algumas partes da estrutura de um compilador completo.

3.1 - Análise Léxica

A análise léxica é a primeira fase do compilador. A função do analisador léxico, também denominado *scanner*, é:
"Fazer a leitura do programa fonte, caractere a caractere, e traduzi-lo para uma seqüência de símbolos léxicos, também chamados *tokens"* (PRICE e TOSCANI, 2001).

Os símbolos léxicos citados acima são referentes ao conjunto de palavras reservadas, variáveis, constantes e funções que compõem uma linguagem de programação, podendo ser também uma variável ou função declarada pelo programador. Como exemplos de *tokens* para linguagem C tem-se o comando *for*, a função *printf( )*, o identificador *int*.

Após serem identificados no código fonte os *tokens* são classificados segundo sua categoria a partir de um cabeçalho pré-estabelecido. Se o *token* for identificado como uma variável ou função será usada uma tabela de símbolos para o armazenamento de caracteres e suas informações (AHO, *et al.*,1995).

Um compilador usa uma tabela de símbolos para controlar as informações de escopo e das amarrações a respeito de nomes. A tabela de

símbolos é pesquisada a cada vez que um nome é encontrado no código fonte (AHO, *et al.*,1995).

Esta tabela de símbolos pode vir a ser feita através de listas lineares que são mais simples de se implementar, porém perdem desempenho quando há um grande número de consultas e de valores dentro da tabela. Outra forma de se implementar é através de tabelas *hash* que são mais rápidas, porém exigem um maior esforço de programação (PRICE e TOSCANI, 2001).

Uma *Palavra, Cadeia de Caracteres* ou *Sentença* sobre um alfabeto é uma seqüência finita de símbolos justapostos.

A *palavra vazia,* representada pelo símbolo , é uma palavra sem símbolo. Se representa um alfabeto, então * denota o conjunto de todas as palavras possíveis sobre, analogamente, + representa o conjunto de todas as palavras sobre excetuando-se a palavra vazia, ou seja, + = Z* - { }. (MENEZES, 2000).

Um *Alfabeto* é um conjunto finito de *Símbolos.* Portanto, um conj unto vazio também é considerado um alfabeto. Um *símbolo* ou *caractere* é uma entidade abstrata básica a qual não é definida formalmente. Letras e dígitos são exemplos de símbolos frequentemente usados. (Menezes, 2000).

Ferreira (2004) define *linguagem* como o uso da palavra articulada ou escrita como meio de expressão e comunicação entre pessoas.

Entretanto, esta definição não é suficientemente precisa para permitir o desenvolvimento matemático de uma teoria sobre linguagens. (MENEZES, 2000).

É interessante observar que o mesmo programa fonte é visto pelos analisadores léxico e sintático como sentenças de linguagens diferentes.

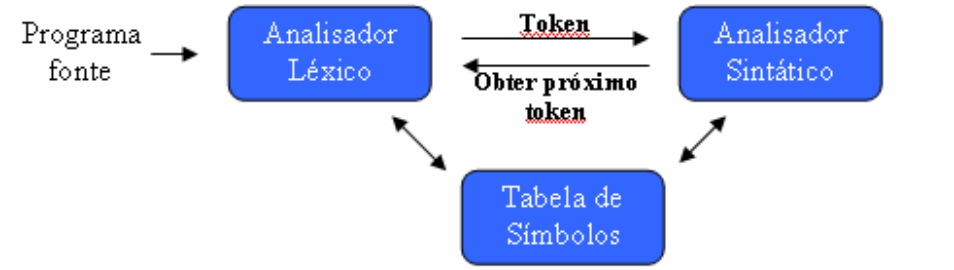


Figura 3: **Interação do analisador léxico com o analisador sintático** (Aho, 1986)

Para o analisador léxico, o programa fonte é uma seqüência de palavras de uma *linguagem regular.* Para o analisador sintático, essa seqüência de *tokens* constitui uma sentença de uma *linguagem livre do contexto* (sentenças formadas por expressões e blocos aninhados, por exemplo). (PRICE e TOSCANI, 2001).

Os analisadores sintáticos serão abordados no próximo tópico desse estudo. Uma *linguagem* é um conjunto de palavras formadas por símbolos de um determinado alfabeto. Os símbolos constituem uma *linguagem regular.*

As linguagens regulares e as livres do contexto são as mais simples, segundo a classificação proposta por Chomsky[1], dentro da Teoria das Linguagens Formais. No que diz respeito ao contexto da tradução de linguagens de programação, as linguagens são usualmente apresentadas

através de gramáticas ou de algoritmos (autômatos) que as reconhecem. (Price, 2001).

3.2 - Análise Sintática

Toda linguagem de programação deve possuir um conjunto de regras que definem a estrutura sintática da linguagem (PRICE e TOSCANI, 2001). Estas regras são utilizadas para validar a estrutura gramatical da linguagem em questão.

Segundo Price e Toscani, o analisador sintático funciona da seguinte maneira:

Dada uma gramática livre de contexto, **G,** e uma sentença (programa fonte) **s**, o objetivo do analisador sintático é verificar se a sentença **s** pertence à linguagem gerada por **G**. O analisador sintático, também chamado *parser*, recebe do analisador léxico a seqüência de *tokens* que constitui a sentença **s** e produz como resultado uma árvore de derivação para **s**, se a sentença é válida, ou emite uma mensagem de erro, caso contrário.

A verificação da sentença ocorre após o analisador sintático receber a cadeia de *tokens* do analisador léxico.

O ideal para um analisador sintático é continuar a análise até o final do código fonte, mesmo sendo encontrados erros ao longo do código.

Para ilustrar o conceito acima temos o seguinte exemplo na linguagem C.

for(int = 4)

Apesar de todos identificadores existirem na linguagem C, a expressão está sintaticamente incorreta, pois não conferem com a estrutura gramatical da linguagem.

3.3 - Análise Semântica

Nesta fase há uma verificação de erros semânticos no programa fonte. A principal tarefa da análise semântica é a verificação de tipos.

Nela o compilador verifica se cada operador recebe os operandos que são permitidos pelo código fonte.

O objetivo da análise semântica é trabalhar nesse nível de inter-relacionamento entre as partes distintas do programa.

As tarefas básicas desempenhadas durante a análise semântica incluem a *verificação de tipos,* a *verificação do fluxo de controle* e a *verificação da unicidade da declaração de variáveis e funções.* (RICARTE, 2003).

Para tornar as ações semânticas mais efetivas, pode-se associar variáveis aos símbolos (terminais e não-terminais) da gramática. Assim, os símbolos gramaticais passam a conter atributos (ou parâmetros) capazes de armazenar valores durante o processo de reconhecimento.

Toda vez que uma regra de produção é usada no processo de reconhecimento de uma sentença, os símbolos gramaticais dessa regra são “alocados” juntamente com seus atributos.

Isto é, cada referência a um símbolo gramatical, numa regra, faz com que uma cópia desse símbolo seja criada, juntamente com seus atributos. Analogamente a árvore de derivação da sentença sob análise, é como se a cada nó da árvore (símbolo gramatical) correspondesse a uma instanciação de um símbolo e de suas variáveis. (PRICE, 2001).

4 – PROJETO DE UM ANALISADOR

O analisador desenvolvido foi projetado com base em autômatos finitos para validação de algumas estruturas, tais como, validação de variáveis e funções, diretivas de pré-processador, estruturas de controle de fluxo e validação comentários.

Cada autômato deverá ser implementado de forma independente. Com isso os módulos produzidos durante a implementação podem ser reutilizados.

Dentro das estruturas de controle de fluxo o analisador não faz a verificação de condição, apenas verifica o inicio e fim da estruturas.

As estruturas foram ilustradas e representadas através de diagramas de transição de estados para o entendimento do leitor. O uso de autômatos (especificamente a representação por grafos) para projetos de compiladores é de grande importância, porque permite a visualização do problema através de estruturas gráficas facilmente compreensíveis para profissionais da computação. Além disso, os autômatos podem ser usados como documentação do projeto.

4.1 - Diretivas do Pré-Processador

As diretivas do pré-processador são comandos que não são compilados como o restante do código, são executadas antes do processo de compilação, são precedidas do símbolo #.

Pode-se incluir diversas instruções do compilador no código-fonte de um programa em C. Elas são chamadas de diretivas do pré-processador e, embora não sejam realmente parte da linguagem de programação C, expandem o escopo do ambiente de programação em C. (SCHILDT, 1996).

4.1.1 Diretiva #include

A diretiva #include permite que o compilador utilize rotinas contidas em outro código fonte. O nome do arquivo adicional deve estar entre aspas ou símbolos de maior e menor.

Forma geral:

#include "biblioteca"
#include <biblioteca>

Exemplo:

#include "arqs/biblio.h"
#include <stdio.h>

Em um dos exemplos há a de utilização da biblioteca padrão ANSI-C *stdio* . As duas formas de especificação com aspas e sinais de maior e menor se diferenciam pelo fato da primeira ser possível incluir o caminho de onde o arquivo se encontra, a segunda utiliza apenas o nome da biblioteca.

A seguir é mostrado o autômato que representa a diretiva #include.

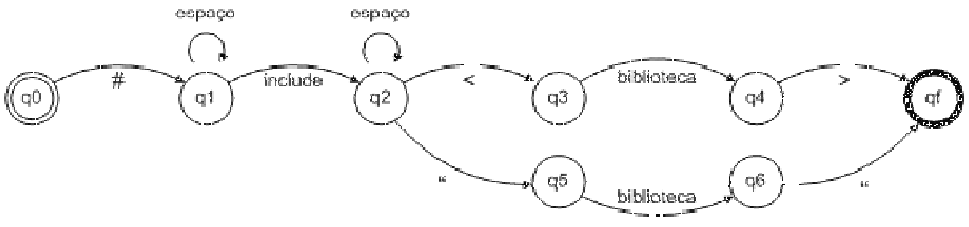


Figura 4: **Módulo *include***

4.1.2 Diretiva #define

A diretiva #define especifica um identificador e atribui a este um valor, quando o identificador é encontrado no código é substituído pelo valor atribuído na diretiva. O padrão ANSI C refere-se ao identificador como um nome de macro e ao processo de substituição como substituição de macro. (SCHILDT, 1996).

Forma geral:

#define nome_da_macro caracteres

Exemplo:

#define tamanho 10

No exemplo acima foi especificado a macro *tamanho* de valor *10.*

Neste comando observa-se a ausência de ponto e vírgula no final da linha. Pode haver qualquer número de espaços entre o identificador e a *string*, mas, assim que a *string* começar, será terminada apenas por uma nova linha. (SCHILDT, 1996).

Abaixo é mostrado o autômato que representa a diretiva #define.

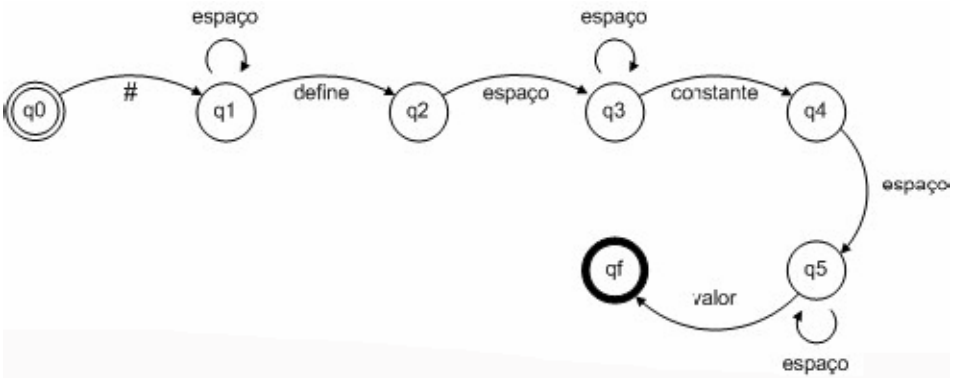


Figura 5: **Módulo *define***

4.2 Tipos de Variáveis

Uma variável é uma posição alocada na memória que pode possuir diferentes valores conforme a execução de programa.

Forma geral :

tipo lista_de_variáveis;

Exemplo :

int soma,cont=2;

No exemplo acima foram declaradas duas variáveis do inteiro, a variável *cont* possui um valor atribuído em sua declaração.

Uma variável não pode ser igual a uma palavra-chave de C e não deve ter o mesmo nome que as funções escritas no programa ou as que estão nas bibliotecas ANSI C. (SCHILDT, 1996).

A seguir é mostrado o autômato que representa as variáveis da linguagem C.

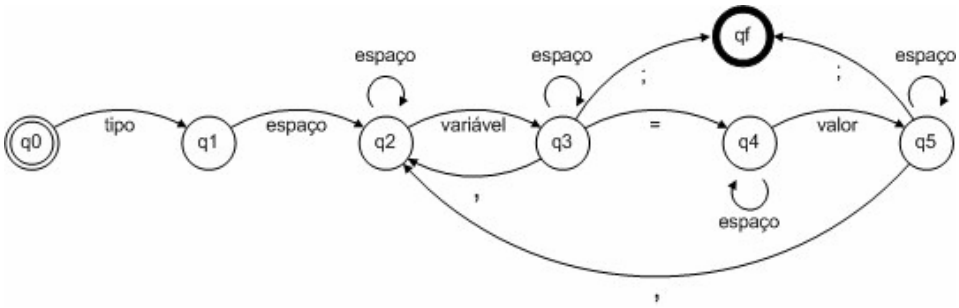


Figura 6: **Módulo variáveis**

4.3 Estruturas de Dados

Uma estrutura de dados agrupa diversas variáveis a um conjunto comum. Quando este conjunto é declarado é possível atribuir e acessar os conteúdo de todas variáveis.

As variáveis que compreendem a estrutura são chamadas membros da estrutura. (SCHILDT, 1996).

Forma geral:

```
struct nome_estrutura {
   tipo lista_de_variáveis;
   tipo lista_de_variáveis;
   (...)
} variáveis_estrutura;
```

Exemplo:

```
struct pontos{
int max,min;
float media;
}eixos;
```

O exemplo acima cria uma estrutura chamada pontos contendo três variáveis duas do tipo inteiro e uma do tipo real. A variável da estrutura é chamada eixos, está pode ser omitida.

A figura 8 mostra o autômato que representa a estrutura de dados da linguagem C.

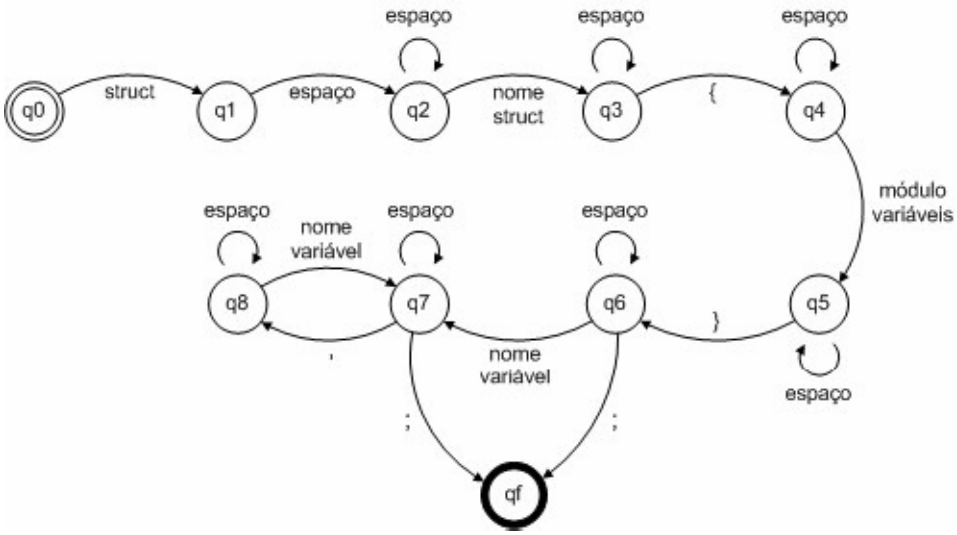


Figura 7: **Módulo *struct***

4.4 Definição de Funções

Funções são os blocos de construção de C e o local onde toda a atividade do programa ocorre. Elas representam uma das características mais importantes da linguagem. (SCHILDT, 1996).

Forma geral:

tipo nome_da_função (<parâmetros>) {

<escopo_da_função>

}

Exemplo:

```
float  media(int a,int b){
   float calc;
```

```
    calc=(a+b)/2;
    return(calc);
}
```

Pela forma geral *tipo* é o tipo de retorno da função, *nome_da_função* é o nome que irá referenciar a função ao longo código do programa, *parâmetros* são o conjunto de variáveis de entrada na função, este conjunto pode ser vazio. O *escopo* é conteúdo que interno da função, que é executado quando a função é chamada.

A seguir é apresentado o autômato que representa a declaração de funções da linguagem C.

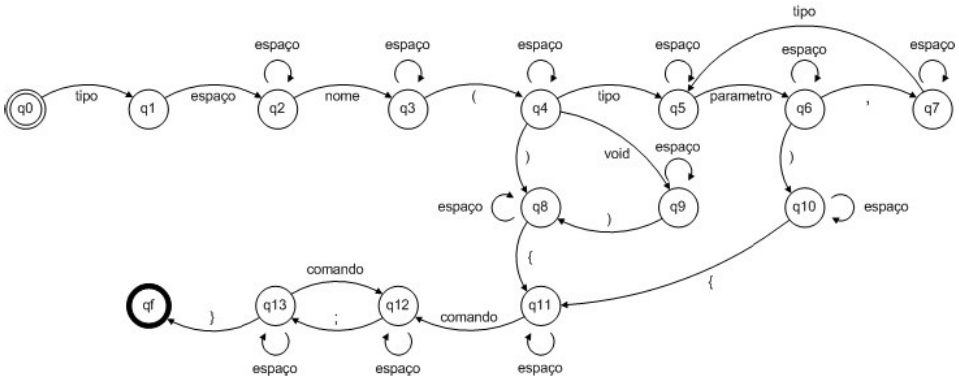


Figura 8: **Módulo *function***

4.5 Protótipos de Funções

Um protótipo de função é exatamente a declaração de uma função que será utilizada ao longo do programa. O protótipo existe para que o compilador tome conhecimento que a função que vai ser utilizada foi implementada ao longo do código.

O protótipo da função será bem parecido com o cabeçalho da função posteriormente implementada.

Forma geral:

tipo nome_da_função (<parametros>);

Exemplo:

Float media(int a,int b);

Na figura 9 é mostrado o autômato que representa a declaração de protótipos de funções da linguagem C.

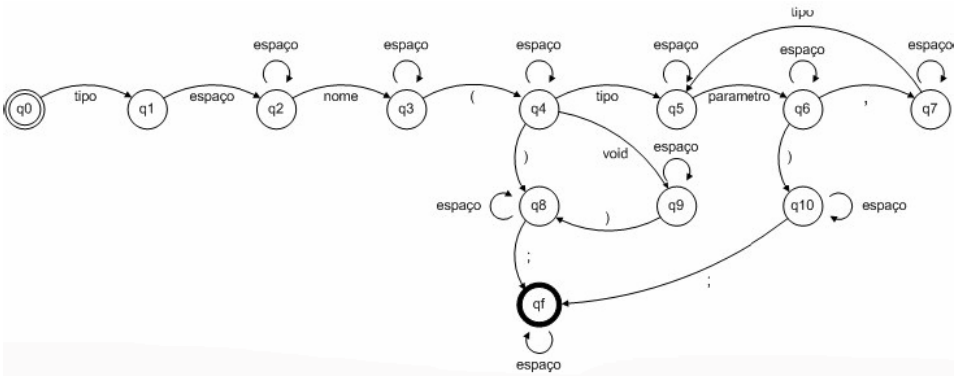


Figura 9: **Módulo protótipo da função**

4.6 Estruturas de Controle de Fluxo

A função destas estruturas é gerenciar o fluxo informação no código através de condições lógicas.

As estruturas de controle nesta linguagem são: if-else, switch, for, while e do-while.

4.6.1 Estrutura If-Else

Nesta estrutura haverá tomada de decisões com relação a qual parte do programa será executada

Na estrutura if se o resultado da expressão lógica for verdadeiro, o escopo interno da estrutura é acessado.

Forma geral:

if (condição)
<escopo>;

if (condição)
<escopo1>;
else
<escopo2>;

Existindo o *else* se a condição for verdadeira o *escopo1* é acessado caso contrário o *escopo2* é acessado.

. A figura 10 mostra a representação da estrutura de controle if-else da linguagem C.

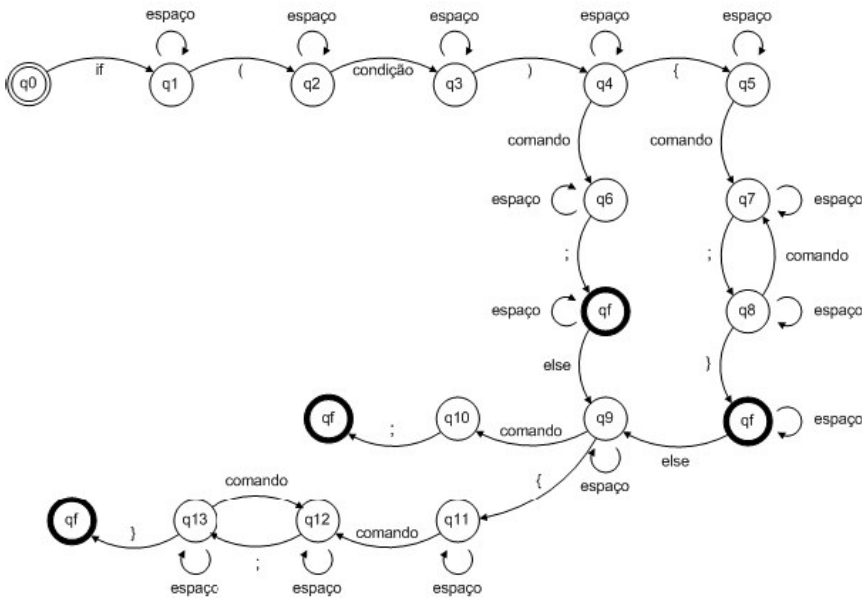


Figura 10: **Módulo *if-else***

4.6.2 Switch

O comando switch é uma estrutura de decisão múltipla que testa se uma expressão casa um de vários valores inteiros *constantes*, e desvia de acordo com o resultado. (KERNIGHAN, 1988). Sendo assim a estrutura *switch* não aceita expressões lógicas, aceita somente constantes.

Forma geral:

switch (variável) {
case condição1 : comando1;

case condição2 : comando2;

case condição3 : comando3;

(...)

condiçãoN : comandoN;

}

No exemplo acima o comando testa a variável e executa a linha cujo o *case* corresponda a ao mesmo valor da variável.

A seguir é mostrado o autômato que representa a estrutura de controle switch da linguagem C.

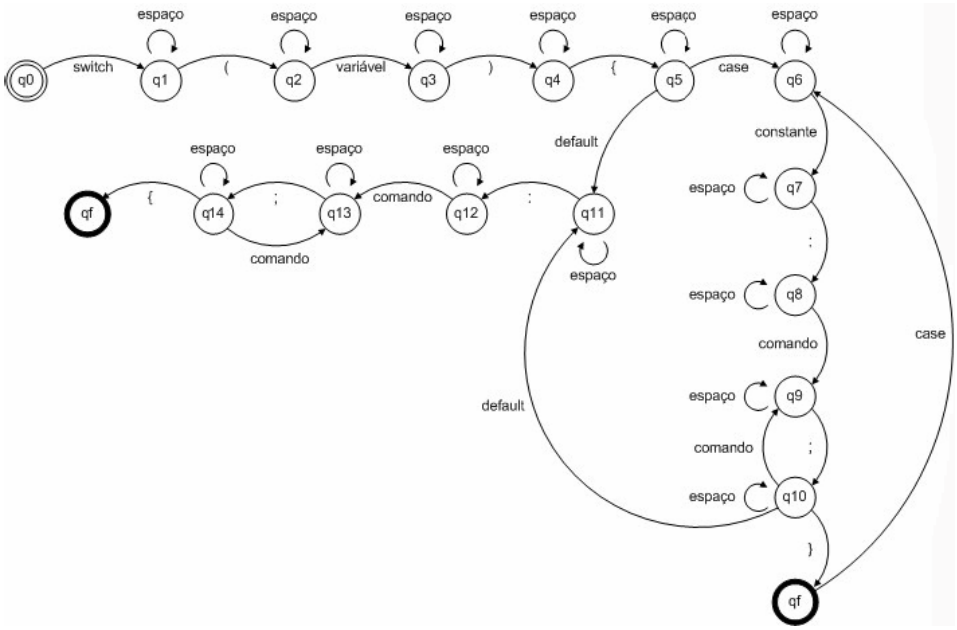


Figura 11: **Módulo *switch***

4.6.3 Estrutura For

O comando *for* é uma das três estruturas de *loops* de repetição da linguagem C, na maioria da vezes este comando é utilizado quando se tem um número determinado *loops* a serem feitos.

Forma geral:

```
for (inicialização; condição; inc/dec){
        <escopo>
}
```

Exemplo:

```
for(i=0;i<=10;i++)
{ <escopo>
}
```

Pode-se observar que o comando *for* possui três tipos de expressões. Em geral, a primeira e a terceira expressão são atribuições ou chamadas de função e a segunda expressão é uma expressão relacional.

Na maioria dos casos usa-se a primeira expressão para inicialização de variáveis, a segunda como expressão lógica e a terceira como incremento ou decremento de variável.

Qualquer uma das três pode ser omitida, embora os ponto-e-vírgulas devam permanecer.

A figura 12 mostra o autômato que representa a estrutura de controle for da linguagem C.

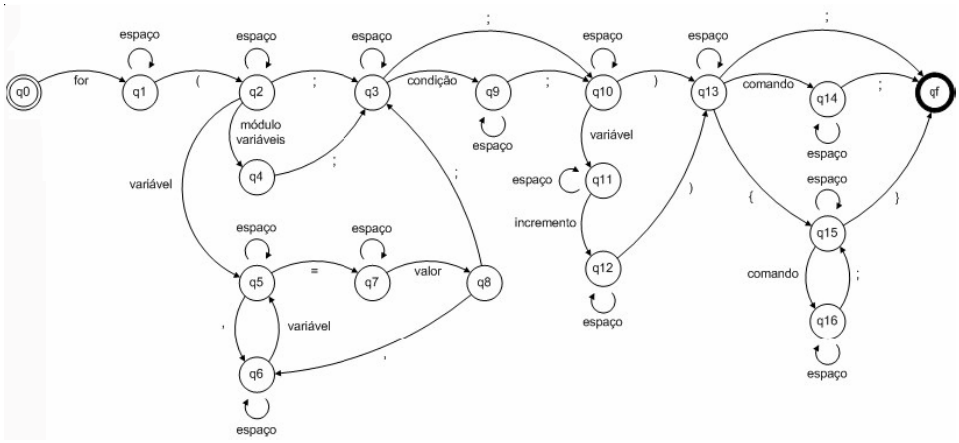


Figura 12: **Módulo *for***

4.6.4 Estrutura While

O comando *while* assim como o comando *for* faz com que uma determinado código que esteja dentro de seu escopo seja executado diversas vezes até que uma condição não seja mais satisfeita.

Forma geral:

while (condição)
<escopo>

Exemplo:

while(x=y)
{escopo}

Abaixo é mostrado o autômato que representa a estrutura de controle *while* da linguagem C.

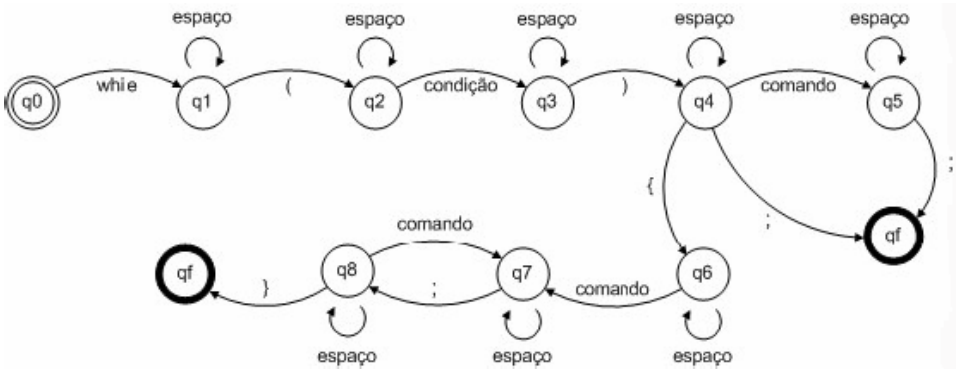


Figura 13: **Módulo *while***

4.6.5 Estrutura do-while

A estrutura do-while cria um ciclo repetido até que a condição seja não seja mais satisfeita, diferentemente das estruturas *for* e *while* esta estrutura garante que o condigo seja executado pelo menos uma vez, pois o teste só é feito no final da estrutura.

Forma geral:

do
<escopo>;
while (condição);

A seguir é mostrado o autômato que representa a estrutura de controle while da linguagem C.

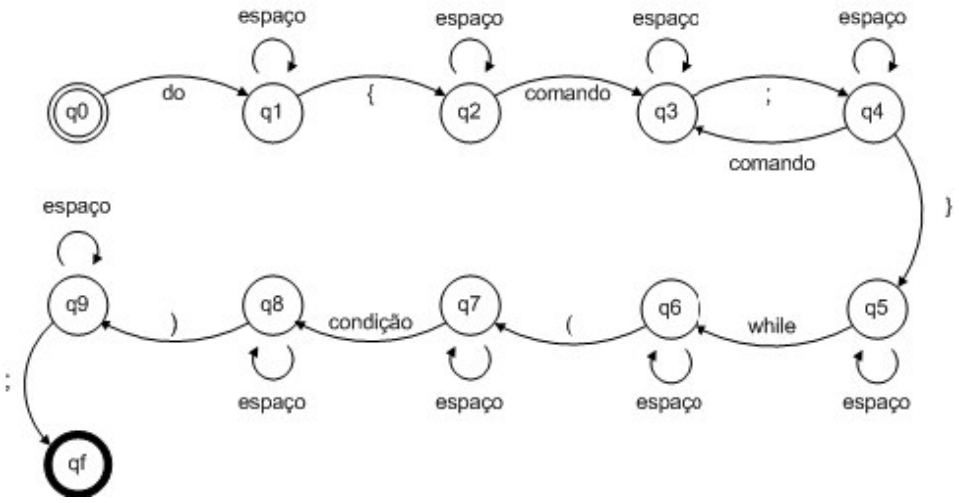


Figura 14: **Módulo *do-while***

4.7 Comentários

Os comentários na linguagem C são colocados entre /* e */ e não sendo considerados durante o processo de compilação. É válido lembrar que os comentários de uma linha precedidos de // não são padrão ANSI-C, portanto não será abordado.

Forma geral:

/ sequência de caracteres*/*

Exemplo:

/ Isso é uma comentário */*

Abaixo é mostrado o autômato que representa os comentários na linguagem C.

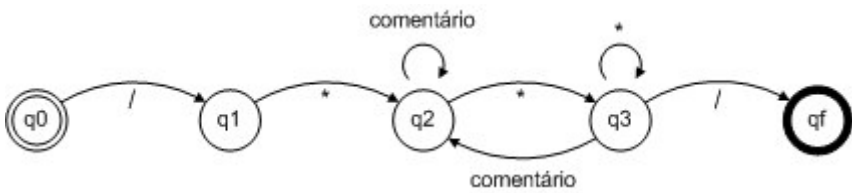


Figura 15: **Módulo comentário**

5 - O DESENVOLVIMENTO DE UM ANALISADOR

A implementação do analisador se baseia na análise da estrutura gramatical da linguagem de programação C, para cada *token* é feita uma verificação que define a qual tipo de identificador este pertence. Após conhecer a classe do identificador é acionado uma rotina específica para sua validação.

Se o *token* não pertence à classe alguma ou está definido forma incorreta, fora dos padrões da linguaguem é gerada uma mensagem de erro imediatamente.

No projeto do analisador proposto, as estruturas gramaticais a serem implementadas são: as diretivas de pré-processador, tipos de variáveis, funções do padrão ANSI (definidas em bibliotecas), e estruturas de controle de fluxo. Maiores detalhes de estrutura de dados em C podem ser encontrados em (SCHILDT, 1996).

Como foi visto anteriormente, todas as estruturas gramaticais da linguagem C foram modeladas através do uso de autômatos finitos, sendo assim, para cada tipo de estrutura tem-se o autômato correspondente.

Um algoritmo correspondente ao autômato deverá ser acionado a partir do momento que for verificada uma estrutura da linguagem C no código fonte do programador. A seguir, apresenta-se um exemplo para ilustrar o processo.

Exemplo:

float media;

Quando o *scanner* do analisador encontrar a palavra reservada **float,** imediatamente é verificado que se trata de uma declaração de variável, a partir daí o algoritmo de verificação de variáveis é acionado para validação sintática da linha.

Foi utilizada somente a linguagem C para o desenvolvimento do projeto, não sendo usado nenhum analisador léxico ou sintático existente.

Um dos motivos que levaram os autores a esta decisão foi pela flexibilidade do projeto e pela possibilidade de se adquirir um conhecimento profundo sobre técnicas de desenvolvimento de compiladores.

O nome dado ao programa executável do analisador de códigos foi "prec". A chamada deste programa é feita da seguinte maneira:

prec nomedoarqvuivo.c

Onde *nomedoarquivo.c* é o nome do arquivo fonte a ser analisado pelo pré-compilador.

Se houver identificação de erros, os mesmo são exibidos no console para o usuário.

A construção do pré-compilador foi dividida em três etapas, cada qual com objetivos distintos que serão detalhados.

Portanto, esse capítulo tem como finalidade, abordar toda a estrutura utilizada para o desenvolvimento da implementação de um analisador de códigos para a linguagem de programação C, baseando-se sempre no padrão ANSI desta linguagem. O objetivo de se adotar este

padrão, é o fato de tornar o pré-compilador portável para diversas plataformas de sistemas operacionais, bastando para tal, apenas compilá-lo na plataforma desejada.

Este analisador foi compilado e executado sobre os sistemas operacionais Linux e Windows, utilizando-se diversos tipos de processadores.

Para a utilização do programa digita-se o nome do arquivo executável chamado *prec* e coloca-se em seguida o caminho do arquivo a ser verificado.

Exemplo:

prec meusarqs/progs/teste.c

O analisador fará uma leitura de todo o código contido no arquivo cujo nome foi passado como parâmetro e verificará a existência de um conjunto de erros comuns de programação. Os erros encontrados serão reportados ao usuário na tela do computador.

5.1 - Primeira fase

Esta fase teve como objetivo realizar algumas das tarefas da análise léxica, formatando o código fonte, de forma que o mesmo fique adequado para realização das etapas posteriores (Sintática e Semântica).

Toda a formatação do código original é feita em um novo arquivo

que será o utilizado nas etapas posteriores, mantendo o arquivo original sem alterações.

O exemplo a seguir ilustra o processo:

prec teste.c

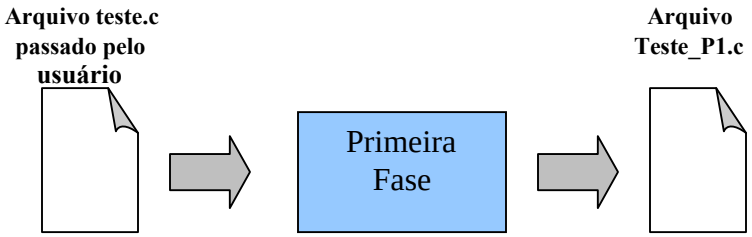


Figura 16 **– Processo da primeira fase**

Após passar o arquivo *teste.c* por parâmetro para o pré-compilador, inicia-se a primeira etapa, após a qual, é gerado um novo arquivo chamado *teste_P1.c*, demonstrado na figura 16.

As regras de formatação aplicados no novo arquivo são:

- Colocação de espaçamento antes e depois dos seguintes tokens: (,), [,], , , &&, ||, &, *, +=, =, -=, *=, /, /=, %, %=, --, ++, -, +, !, >, <, >=, <=, !=, ==, ., ?;
- Colocar espaçamento antes dos seguintes tokens: **;** , **“** , **‘** ;
- Colocar espaçamento depois dos seguintes tokens: # ;
- Eliminar espaços iniciais como tabulações, linhas em branco, etc;

- Eliminar comentários;
- Se houver mais de um comando por linha, colocá-los em linhas separadas, deixando apenas um comando por linha;
- Se houver o token ‘{’ na mesma linha de algum comando, passar o token para a próxima linha, o mesmo se aplica para o token ‘}’;

A seguir há um exemplo de código gerado na primeira fase. A primeira imagem apresenta um arquivo codificado na linguagem C.

```
#include<stdio.h>
/* Programa que exibe o fatorial de um numero */
int numero,res;

/*funcao que calcula o fatorial */
int fat(int num)
{
    int cont,resultado=1;
    for(cont=1;cont<=num;cont++)
       resultado=resultado*cont;
    return resultado;
}
/* funcao principal /*
void main(void)
{
   printf("Forneca um numero: ");
   scanf("%d",&numero);
   res=fat(numero);
   if (numero>=0)
     printf("O fatorial de %d = %d",numero,res);
   else
     printf("Nao existe fatorial para este numero !");
}
```

Figura 17 – **Arquivo *teste.c***

Após submeter o arquivo ao analisador tem-se o seguinte resultado.

```
# include < stdio.h >
int numero , res ;

int fat ( int num )
{
int cont , resultado = 1 ;
for ( cont = 1 ; cont <= num ; cont ++  )
resultado = resultado * cont ;
return resultado ;
}

void main ( void )
{
printf (  "Forneca um numero: "  )  ;
scanf (  "%d"  ,  & numero )  ;
res = fat ( numero )  ;
if  ( numero >= 0 )
printf (  "O fatorial de %d = %d"  , numero , res )  ;
else
printf (  "Nao existe fatorial para este numero !"  )  ;
}
```

Figura 18 **– Arquivo *teste_P1.c***

Após a primeira etapa é gerado um novo arquivo com as alterações aplicadas, o arquivo inicial não é excluído. Deste modo às próximas etapas utilizaram apenas o novo arquivo.

5.2 - Segunda fase

Esta etapa teve por objetivo a organização e validação dos *tokens*. Para a realização desta tarefa foram criadas diversas bibliotecas com objetivos distintos. Houve também a implementação de funções responsáveis por validação de declarações de variáveis e funções.

Nesta etapa há um *scanner* que varre todo arquivo gerado na primeira etapa procurando e validando palavras reservadas da linguagem C. Este *scanner* também procura e valida declarações de funções, variáveis e constantes. Quando não há erro nestas declarações, são inseridas informações sobre as mesmas em uma tabela de símbolos.

Os objetivos desta etapa foram:

- Criar uma tabela com as definições do padrão ANSI-C, são elas: Bibliotecas, Funções, Tipos, Palavras Reservadas, etc;
- Criar uma tabela para o armazenamento das variáveis declaradas;
- Criar uma tabela para o armazenamento das funções declaradas;
- Criar uma tabela para o armazenamento das constantes declaradas;
- Criar uma tabela para o armazenamento das bibliotecas instanciadas;
- Criar uma tabela para o armazenamento das mensagens de erro;
- Identificar e validar declarações de variáveis;
- Identificar e validar declarações de funções;

- Identificar e validar declarações de constantes;
- Identificar e validar declarações de bibliotecas;
- Nas declarações de funções, se o tipo de retorno for diferente de “void”, verificar se há retorno de valores;
- Exibir mensagens de erro de acordo com as validações mencionadas anteriormente;

As tabelas de armazenamento de variáveis declaradas, funções declaradas e constantes citadas no texto acima se referem a tabelas de símbolos que serão geradas dinamicamente. Estas foram implementadas através de listas lineares de registro, pois não haverá a necessidade de armazenamento de um grande número de variáveis, já que o pré-compilador é feito para iniciantes na linguagem C, cujos programas serão pequenos e estarão contidos em um único arquivo fonte.

As demais tabelas são estáticas, ou seja, já possuem as informações previamente definidas em seu escopo.

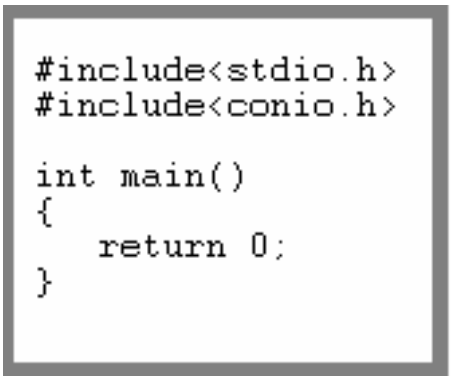
As tabelas estáticas criadas foram:

- Bibliotecas padrão ANSI-C;
- Funções de bibliotecas ANSI(terceira etapa);
- Mensagens de erro;
- Palavras reservadas da linguagem;

As bibliotecas padrão ANSI-C armazenadas são: assert.h, ctype.h, errno.h, float.h, limits.h, locate.h, math.h, setjmp.h, signal.h, stdarg.h, stddef.h, stdio.h, stdlib.h, string.h e time.h.

A partir desse ponto o analisador é capaz de verificar se uma biblioteca declarada no código fonte pertence ou não ao padrão ANSI através da consulta da tabela de bibliotecas ANSI.

Dado o código abaixo:



```
#include<stdio.h>
#include<conio.h>

int main()
{
   return 0;
}
```

Figura 19 – **Arquivo *teste2.c***

O seguinte resultado é exibido após a execução do analisador na figura 20.

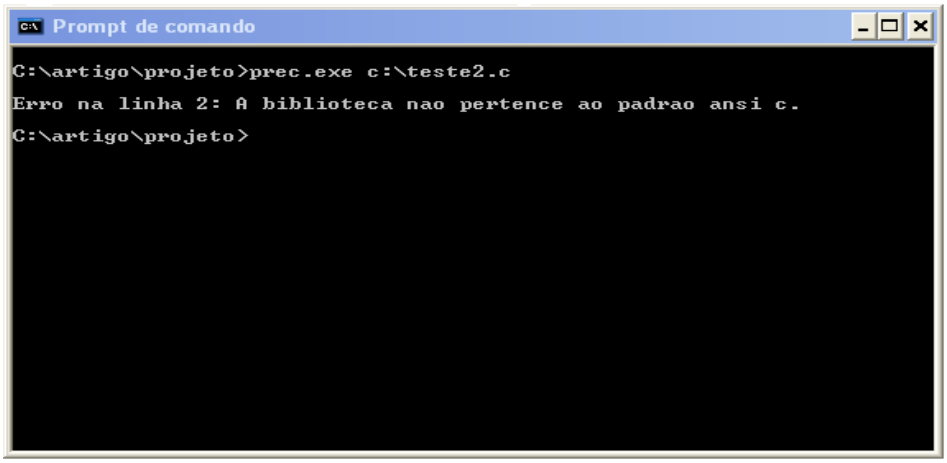


Figura 20: **Execução arquivo teste2.c**

Para as trinta e quatro palavras reservadas da linguagem C, também foi criada uma tabela com as mesmas características da tabela de função. Informações sobre palavras reservadas na linguagem C podem ser obtidas em O'Hare (2004).

Para exibição de mensagens de erro para o usuário, foi criada uma tabela contendo quarenta e sete mensagens. Ocorrendo um erro no programa fonte é verificado a que código pertencem tal erro e partir deste código faz-se a busca nesta tabela. Outra característica importante, é que todas as mensagens de erro foram desenvolvidas no idioma português.

Para verificação de ambigüidade na declaração de variáveis e funções foram criadas tabelas de símbolos que armazenam suas

informações no momento da declaração. A partir da criação destas tabelas é possível obter as seguintes informações:

- A variável ou função declarada não é palavra reservada da linguagem;
- A variável ou função não foi declarada anteriormente;
- A variável é global ou local

Antes de serem inseridas nestas tabelas, as variáveis e funções são analisadas sintaticamente de acordo com especificações da linguagem, projetadas no quarto capítulo.

Na figura a seguir há uma exemplificação do processo ocorrido nesta etapa.

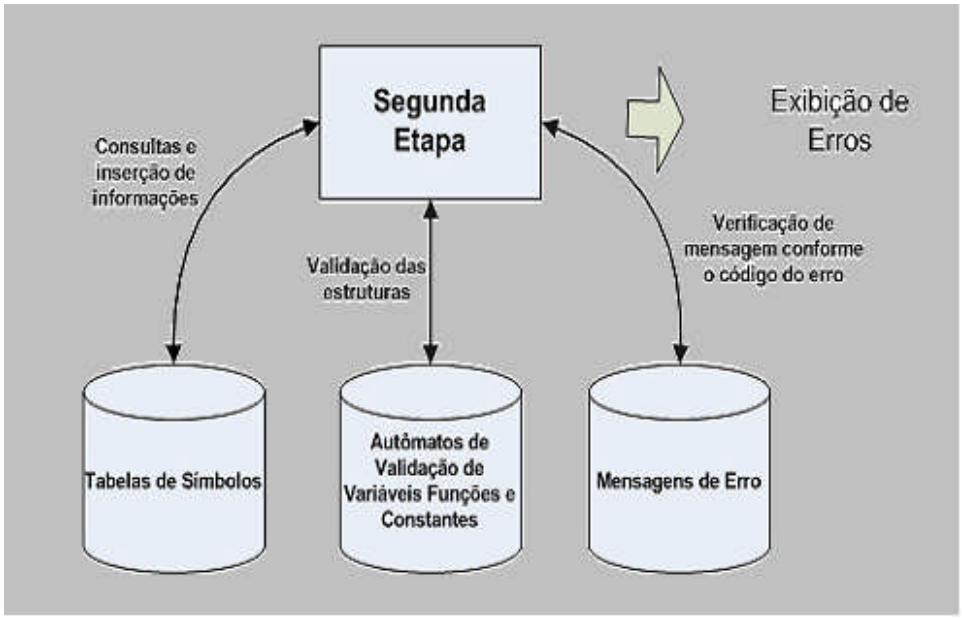


Figura 21 – **Esquema da segunda etapa**

5.3 – Terceira fase

Nesta fase o código é varrido novamente para a verificação de emprego correto do ponto e vírgula, utilização correta de variáveis, funções e constantes. A verificação de variáveis, constantes e funções são feitas por pesquisa nas tabelas de símbolos criadas na segunda etapa. Exemplificando, quando é encontrado um *token*, é verificado se este corresponde a uma função, variável, constante ou palavra reservada da linguagem C.

A terceira fase do analisador tem como objetivo, receber por parâmetro o arquivo fonte gerado pela primeira etapa, e a partir desse arquivo, analisá-lo a fim de verificar algumas regras necessárias.

As regras implementadas nesta etapa são mostradas a seguir:

- Identificar e validar a obrigatoriedade de ponto e vírgula no final das linhas de comando;
- Verificação de ponto e virgulo onde é definido o cabeçalho das funções.
- Verificar paridade de parênteses, chaves, colchetes e aspas duplas;
- Identificação de utilização de variáveis, e posteriormente validar se as mesmas foram previamente declaradas;
- Identificar a utilização de funções, e posteriormente validar se as mesmas foram previamente declaradas;

Para que o analisador consiga identificar se uma determinada função pertence ou não a uma biblioteca do padrão ANSI-C, foi necessário construir uma tabela de funções do padrão ANSI-C.

Também foi descrito neste capítulo, a forma com a qual o analisador utiliza e manipula as tabelas de símbolos de variáveis, de funções, de constantes e de bibliotecas.

6 - RESULTADOS

Este capítulo tem por finalidade apresentar os resultados obtidos com este projeto. Será apresentado alguns exemplos de códigos analisados pelo programa.

Após ser submetido o código abaixo ao analisador tem-se os seguintes resultados na Figura 22.

```
#include<stdio.h>

#define tam 10
/* Declaracao de variaveis */
int main()
{
     int i =;
     int x,y,z;
     float 1teste;
     int vetor[tam];
     float vetor2[tan];
     char y;

     return 1;
}
```

Figura 22 – **Código analisado**

No código acima pode-se observar alguns erros bem evidentes:

- Na linha 6 não há atribuição para variável **i**;
- Na linha 8 **1teste** não é um nome válido para variável;
- Na linha 10 a constante **tan** não existe;

- Na linha 11 a variável y já havia sido declarada na linha 7;

A seguir apresentado o resultados programa em execução.

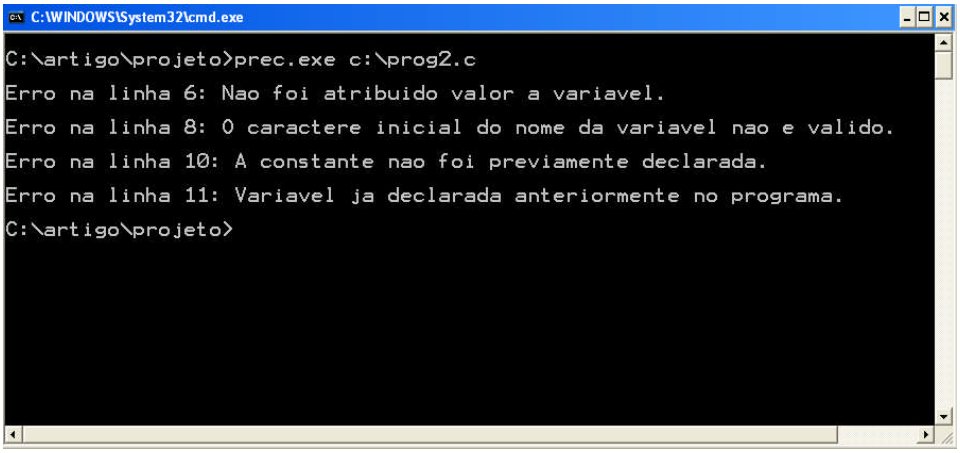


Figura 23 – **Tela de resultados**

Abaixo é apresentado mais um exemplo.

```
#include <stdio.h>
#include <conio.h>

int void(int ?x)
{
    cont=cont++;
    return cont;
}
int main()
{
return 1;
}
```

Figura 24 – **Código analisado**

No exemplo acima pode-se identificar uma serie de erros:

- Na linha 2 a biblioteca **conio.h** não faz parte da linguagem C padrão Ansi;
- Na linha 3 é declarado uma função com nome de **void** que é palavra reservada na linguagem C;
- Na linha 3 **?x** é um nome inválido para variável;
- Na linha 5 a variável **cont** não foi declarada;

Após passar o código para o analisador é exibido o seguintes resultados conforme a figura a seguir:

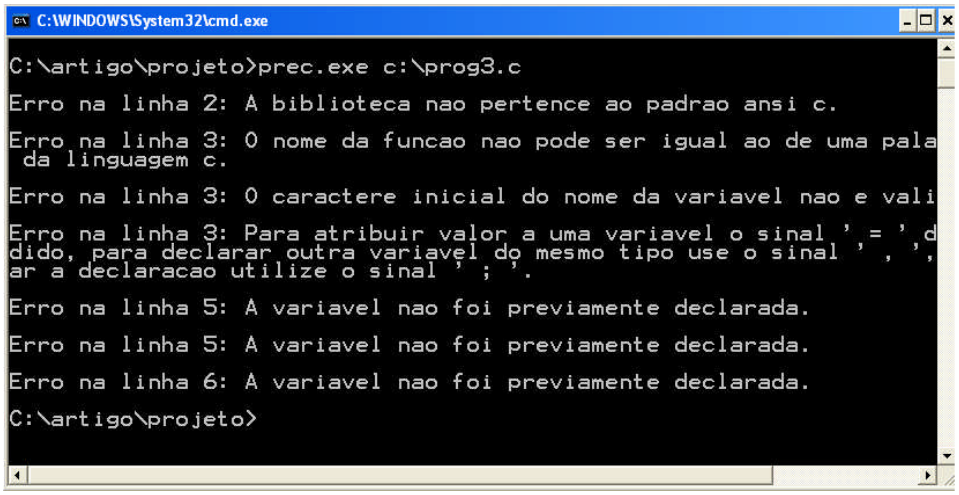


Figura 25 – **Tela de resultados**

Outro exemplo pode ser visto a seguir:

```
#include <stdio.h>

int main()
{
   int z,x=5;
   float l;
   y=x;
   z=3
   if(x==z))
printf("Teste");
   l= sqrt(x);
   return 1;
}
```

Figura 26: **Código analisado**

No exemplo acima pode ser encontrado os seguintes erros:

- Na linha 6 a variável **y** não foi declarada;
- Na linha 7 não há ponto e vírgula no final da linha;
- Na linha 8 não há paridade entre parênteses;
- Na linha 10 não há biblioteca declarada para a função **sqrt;**

Como resultado, obtém-se a tela mostrada na figura 27, a seguir:

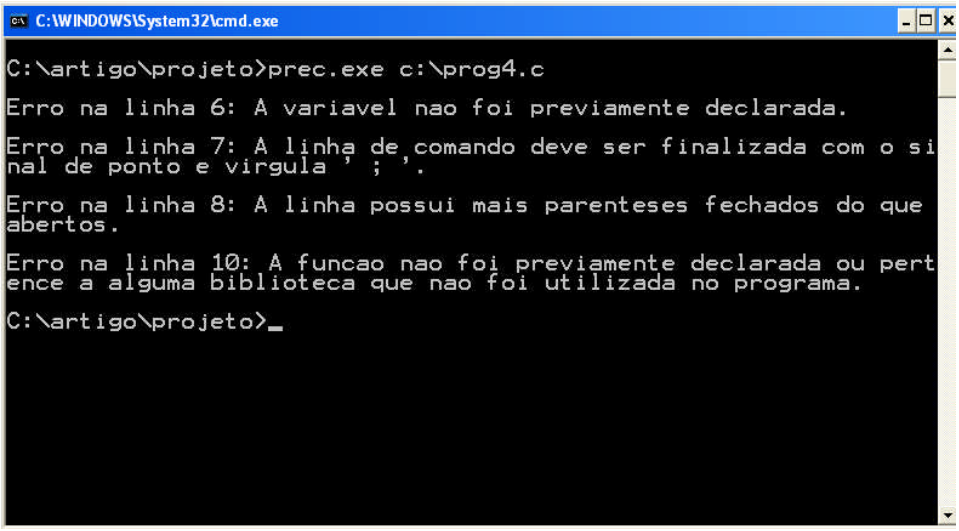


Figura 27 – **Tela de resultados**

7 - CONSIDERAÇÕES FINAIS

O desenvolvimento de um analisador de códigos é de grande importância para o aprendizado dos alunos, pois permite um conhecimento aprofundado na área de construção de compiladores.

Grande parte dos objetivos para o desenvolvimento do analisador de códigos deve ser implementada como: verificação na sintaxe de declaração de variáveis, constantes e funções; verificação e validação de bibliotecas do padrão ANSI quando uma de suas funções é utilizada; validação de variáveis, funções ou constantes verificando se estas foram previamente declaradas; verificação de ponto e vírgula no final de linha; verificação de paridade entre parênteses.

O analisador de códigos gera as mensagens de erro exibindo a linha em que houve sua ocorrência, porém o arquivo tomado como referência é o arquivo gerado na primeira etapa e não o arquivo original, devendo ser corrigido em trabalhos futuro.

O analisador de códigos possui sete bibliotecas e o programa principal. As bibliotecas *ansic.h, listbibl.h, listvar.h, listcons.h, listfunc.h, msgerros.h, utils.h.*

O analisador de códigos não depende de nenhum software auxiliar ou sistema operacional específico sendo totalmente independente.

Alguns módulos propostos no início do projeto não foram implementados, porém estão modelados através de autômatos e podem ser

desenvolvidos em um trabalho futuro. Estas estruturas são: *for, while-do, do-while, if, if-else*, *struct e typedef*.

Para o desenvolvimento do projeto analisado, é necessário, ainda, a implementação das estruturas citadas acima, bem como a validação de tipos em nível semântico.

8 – REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V; SETHI, Ravi; ULLMAN, Jeffrey D. *Compiladores:* princípios, técnicas e ferramentas. Rio de Janeiro: LTC, 1995. 344 p.

C PROGRAMMING LANGUAGE Disponível em: http://www.wikipedia .org /wiki /C_programming_language. Acesso em 28/06/2006.

JOSE NETO, João. *Introdução à compilação*. Rio de Janeiro: LTC, 1987. 222 p.

MENEZES, Paulo Fernando Blauth. *Linguagens formais e autômatos***.** 4.ed. Porto Alegre: Sagra Luzzatto, 2000. 165 p.

MIZRAHI. Victorine Viviane, Treinamento em Linguagem C – Curso Completo – Módulo 1 e 2, Makron Books.

O DESENVOLVIMENTO DA LINGUAGEM C. Disponível em: http://cm. Bell . labs.com /cm/cs/who/dmr/chistPT.html. Acesso em 28/06/2006

PRICE, Ana Maria de; TOSCANI, Simão Sirineo. *Implementação de linguagens de programação:* compiladores. 2.ed. Porto Alegre: Sagra Luzzato, 2001. 194 p.

RITCHIE, Dennis M. O Desenvolvimento da linguagem C., 1996.

SCHILDT, Herbert. *C Completo e total.* 3.ed. São Paulo: Makron Books, 1996. 827 p.

SCHILDT, Herbert. Turbo C: Guia do Usuário, São Paulo, McGraw-Hill, 1988.

The Development of the C Language. Disponível em: http://cm.bell .labs . com/ cm/cs/who/dmr/chist.html. Acesso em 28/06/2006.