ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Departamento de Engenharia de Computação e Sistemas Digitais

Linguagens e Compiladores

RELATÓRIO FINAL

Autores: Erika Midori Kato Priscilla Barreira Avegliano Rodrigo Gioia Guimarães Rodrigo Seisho Hanashiro	Data de emissão: 02/12/2005
Professor: Ricardo Luis de Azevedo da Rocha	Disciplina: PCS-2056

SUMÁRIO

1 Introdução

O objetivo deste projeto é modelar e construir um compilador que utilize uma linguagem de programação imperativa. Assim, baseando-se na arquitetura de Von Neumann (1- busca próxima instrução, 2 - decodifica a instrução, 3 - executa a instrução, e 4- incrementa o contador de instruções), as instruções serão executadas de maneira seqüencial para se otimizar a velocidade de processamento.

O projeto pode ser dividido em quatro partes principais:

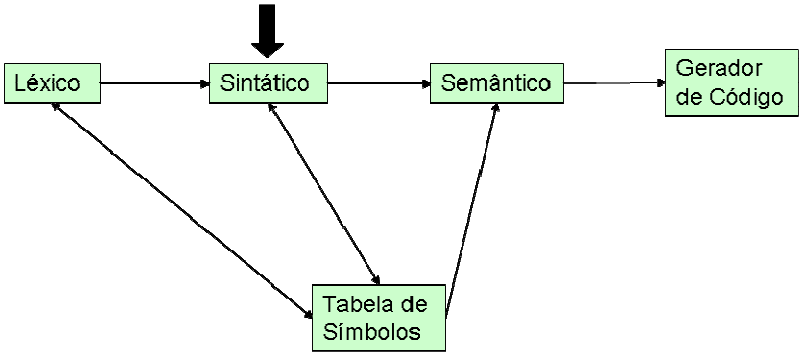
1. Definição da linguagem: para que se possa executar um programa, este deve ser escrito em uma linguagem compreensível ao compilador. Portanto, nesta fase, será definida uma gramática e quais serão os comandos aceitos e suas respectivas funções.

2. Analisador Léxico: definida a linguagem, o analisador léxico é a parte responsável por receber os caracteres do arquivo fonte e agrupá-los em pequenos grupos denominados átomos (*tokens*). Cada um desses átomos terá duas propriedades:
    a. Classe: define o tipo (identificador, número, etc.);
    b. Valor: define o local em que se encontra o valor associado.

3. Analisador Sintático: este módulo é responsável por verificar se o código analisado está gramaticalmente correto. Além de fazer essa análise, este módulo será o "cérebro" do compilador, solicitando os *tokens* para o analisador léxico e chamando o analisador semântico para interpretar os comandos e gerar o código-objeto.

4. Analisador Semântico: este módulo tem como funções principais analisar restrições quanto à utilização dos identificadores, verificar a compatibilidade de tipos, efetuar a tradução do programa e gerar o código-objeto.

O compilador projetado será dirigido por sintaxe, ou seja, o módulo de análise sintática vai gerenciar as atividades, fazendo requisições aos outros módulos. Dessa forma, a integração e comunicação entre os módulos podem ser representadas pelo diagrama abaixo:



Temos que, de acordo com o teorema de Böhm-Jacopini, para que seja possível resolver qualquer tipo de problema computacional, a linguagem desenvolvida deve possui uma instrução de teste condicional (if) e outra de repetição condicional (while). Dessa forma, pode-se dizer que a linguagem criada para o projeto atende aos requisitos citados no teorema.

2 Definição da Linguagem de Alto-Nível

A linguagem de alto nível criada para a construção deste compilador aceita programas formados por uma seqüência de comandos, separados por ponto e vírgula, sendo que o comando vazio é uma das opções válidas.

Além do comando vazio, outros 6 comandos são disponíveis:

- Declaração de variável
- Atribuição de valor a uma variável
- Comandos de decisão
- Comandos de laço
- Comandos de leitura e impressão
- Chamada de função

A declaração de uma variável tem como objetivo definir o tipo e o nome da variável que está sendo criada. Essa medida permite a detecção de alguns erros do programador através da verificação de compatibilidade dos tipos dos utilizados.

O comando de atribuição envolve depositar o valor associado a uma expressão em uma variável, explicitada à esquerda do comando. A avaliação das expressões segue as convenções usuais, sendo efetuada da esquerda para a direita. Tem-se ainda que as potenciações possuem precedência sobre os produtos e divisões, e estes precedência sobre as somas e subtrações, precedências estas alteráveis através da utilização de parênteses. As expressões aceitam somente números e identificadores de variáveis.

Os comandos de decisão implementam saltos condicionais e podem apresentar duas formas:

1. “IF”: este comando envolve uma comparação entre expressões, efetuadas através dos operadores “>” (maior que), “<” (menor que), “>=” (maior ou igual a), “<=” (menor ou igual a), “= =” (igual a) e “!=” (diferente de). As operações são avaliadas e comparadas, de acordo com o operador escolhido. Caso a comparação seja verdadeira, o comando que se encontra entre as palavras “then” e “else” será executado. Caso contrário, o comando após o “else” será executado.

2. “SWITCH”: este comando trata os valores possíveis para uma variável. Esta é comparada a cada valor que se encontra após a palavra “case”, e no caso de a

comparação ser verdadeira, o comando que se encontra após o valor testado é executado.

Os comandos de laço implementam repetições condicionais, e podem apresentar duas formas:

1. "WHILE": este comando testa uma comparação para decidir se irá realizar o comando que segue a palavra "do". Esta ação solicitada será executada repetidamente até que a condição de teste não mais seja atendida.

2. "FOR": este comando tenta unificar três diferentes ações, que são representadas entre parênteses, e que geralmente também estão presentes junto com o comando "while", inclusive ele mesmo. O primeiro parâmetro do comando "for" trata uma atribuição, que geralmente é utilizada para a inicialização de uma variável. Já o segundo parâmetro comporta-se como o próprio "while", realizando a comparação. Finalmente, o terceiro parâmetro trata o passo, necessário para que o laço não seja repetido infinitamente.

Os comandos de leitura e impressão promovem, respectivamente, a entrada e saída de dados com relação a um meio externo. O comando de leitura captura dados e preenche o valor de uma variável, especificada após a palavra "scan". Já o comando de impressão permite duas opções: (a) a impressão do resultado de uma expressão, ou (b) a impressão de um conjunto de caracteres quaisquer, situados entre aspas.

Finalmente, as funções podem ser consideradas pequenos sub-programas, que recebem um conjunto de parâmetros e que são chamados pelo programa principal para executar uma dada ação. As funções são chamadas através do comando "call", que especifica o nome da função requisitada e os parâmetros que devem ser passados para sua execução.

2.1 Descrição Informal da Linguagem

Nesta seção, foi feita uma descrição simplificada das principais funcionalidades desenvolvidas na linguagem. As palavras marcadas em negrito indicam palavras reservadas da linguagem.

→ Funções:
func *tipo nome_função* (*tipo1 parâmetro1*, *tipo2 parâmetro2 ,..., tipoN parâmetroN*)
comando1;
comando2;
...
comandoN;

return *valor* ;
endfunc;

→ Repetição condicional:
while *condição* **do**
comando1;
comando2;
...
comandoN;
endwhile;

for (*atribuição; condição; passo***)**
comando1;
comando2;
...
comandoN;
endfor;

→ Decisão:
if *condição* **then**
comando1;
comando2;
...
comandoN;
else
comando(N+1);
comando(N+2);
...
comandoM;
endif;

→ Decisão de casos:
switch *variável*
case *valor1*
comando1;
comando2;
...
comandoN;
break
case *valor2*
comando(N+1);
comando(N+2);
...
comandoM;

```
                break
        ...
        case    valor2
                comando(M+1);
                comando(M+2);
                ...
                comandoP;
                break
endswitch;
```

→ Impressão:
print *"sentença_a_ser_impressa_na_tela"* **;**
ou
print *variável* **;**

→ Leitura de entrada:
scan *variável* **;**

→ Chamada de função:
call *nome_função* **(** *parâmetro1* , *parâmetro2, ..., parâmetroN***);**

→ Programa Principal:

```
main
        comando1;
        comando2;
        ...
        comandoN;
end
```

2.2 Descrição BNF da Linguagem

Nesta seção, foi feita a descrição formal da linguagem através da notação BNF.

<letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<numero> ::= <digito> <numero> | <digito>

<programa> ::= <funcao> main <seq_comandos> end

<funcao> ::**=** func <tipo> <identificador> (<var>) <seq_comandos> return (<identificador> | <numero>) endfunc <funcao> | func <tipo> <identificador> (<var>) <seq_comandos> return (<identificador> | <numero>) endfunc | ε

<seq_comandos> ::= <comando> ; <seq_comandos> | ε

<comando> ::= <var> | <atribuicao> | <decisao> | <loop> | <leitura> | <impressao> | <chamada>

<var> ::= <tipo> <identificador>

<tipo> ::= int | double | string | boolean

<identificador> ::= <nome> | <estrutura>

<nome> ::= <letra> <nome_aux>

<nome_aux> ::= <letra> | <digito> | <nome_aux> <letra> | <nome_aux> <digito> | ε

<estrutura> ::= <nome> [<numero>] [<numero>] | <nome> [<numero>]

<atribuicao> ::= <identificador> = <expressao>

<expressao> ::= <expressao> + <termo> | <expressao> - <termo> | <termo>

<termo> ::= <termo> * <fator> | <termo> / <fator> | <fator>

<fator> ::= <fator> ^ <pot> | <pot>

<pot> ::= <identificador> | <numero> | (<expressão>)

<decisao> ::= <case> | <if>

<if> ::= if <comparacao> then <seq_comandos> else <seq_comandos> endif | if <comparacao> then <seq_comandos> endif

<comparacao> ::= <expressao> <comparador> <expressao> | <boolean>

<boolean> ::= <bool> and <boolean> | <bool> or <boolean> | <bool>

<bool> ::= (<boolean>) | <identificador> | not <identificador>

<comparador> ::= == | != | < | > | <= | >=

<case> ::= switch <identificador> <aux_case> endswitch

<aux_case> ::= case <identificador> <seq_comandos> break <aux_case> | case <identificador> <seq_comandos> break

<loop> ::= <while> | <for>

<while> ::= while <comparacao> do <seq_comandos> endwhile

<for> ::= for(<atribuicao> ; <comparacao> ; <expressao>) <seq_comandos> endfor

<leitura> ::= scan <identificador>

<impressao> ::= print <expressao> | print " <expressao> "

<chamada> ::= call <nome> (<aux_cham>)

<aux_cham> ::= <identificador> | <identificador> , <aux_cham> | ε

2.3 Descrição da Linguagem em Notação de Wirth

Nesta seção, foi feita a descrição da linguagem através da notação de Wirth.

letra = “a” | “b” | “c” | “d” | “e” | “f” | “g” | “h” | “i” | “j” | “k” | “l” | “m” | “n” | “o” | “p” | “q” | “r” | “s” | “t” | “u” | “v” | “w” | “x” | “y” | “z”

| “A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” | “I” | “J” | “K” | “L” | “M” | “N” | “O” | “P” | “Q” | “R” | “S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z”.

digito = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”.

numero = digito {digito} .

programa = {funcao} “main” seq_comandos “end”.

funcao = “func” tipo nome “(” var {“,”var} “)” seq_comandos “return” (identificador | numero) “endfunc”

seq_comandos = {comando “;”} .

comando = var | atribuicao | decisao | loop | leitura | impressao | chamada .

var = tipo identificador .

tipo = “int” | “double” | “string” | “boolean”.

identificador = nome | estrutura .

nome = letra {letra | digito}.

estrutura = nome “[” numero “]” [“[” numero “]”] .

atribuicao = identificador “=” expressão .

expressao = termo { (“+” | “-”) termo } .

termo = fator { (“*” | “/”) fator }.

fator = pot { “^” pot }.

pot = identificador | numero | “(” expressão “)” .

decisao = case | if .

if = “if” comparacao “then” seq_comando [“else” seq_comando] “endif” .

comparacao = expressao comparador expressao | boolean .

boolean = bool { (“and” | “or”) bool }.

bool = “(” boolean “)” | [“not”] identificador .

comparador = “= =” | “!=” | “<” | “>” | “<=” | “>=” .

case = “switch” (identificador | numero) “case” identificador seq_comandos “break” {“case” identificador seq_comandos “break”} “endswitch” .

loop = while | for .

while = “while” comparacao “do” seq_comandos “endwhile” .

for = “for(” atribuicao “;” comparacao “;” expressao “)” seq_comandos “endfor”.

leitura = “scan” identificador .

impressao = “print” (expressao | “”” expressao “”””) .

chamada = “call” nome “(” [identificador {“,” identificador}] “)” .

2.3.1 Descrição Reduzida

Para permitir a implementação do autômato de pilha estruturado, método que será utilizado para a construção do analisador sintático, a descrição em notação de Wirth mostrada anteriormente foi reduzida, agrupando os não-terminais essenciais à linguagem.

programa = {“func” tipo nome “(” (tipo identificador) {“,” (tipo identificador) } “)” seq_comandos “return” (identificador | numero) “endfunc”} “main” seq_comandos “end”.

seq_comandos = { (
tipo identificador
| identificador “=” expressão
| “if” (expressao comparador expressao | boolean) “then” seq_comandos [“else” seq_comandos] “endif”
| “switch” identificador “case” identificador seq_comandos “break” {“case” identificador seq_comandos “break”} “endswitch”
| “while” (expressao comparador expressao | boolean) “do” seq_comandos “endwhile”
| “for(” (identificador “=” expressão) “;” (expressao comparador expressao | boolean) “;” expressao “)” seq_comandos “endfor”
| “scan” identificador
| “print” (expressao | “””” expressao “””)
| “call” nome “(” [identificador {“,” identificador}] “)”)
“;”}.

boolean = (“(” boolean “)” | [“not”] identificador) { (“and” | “or”) (“(” boolean “)” | [“not”] identificador) }.

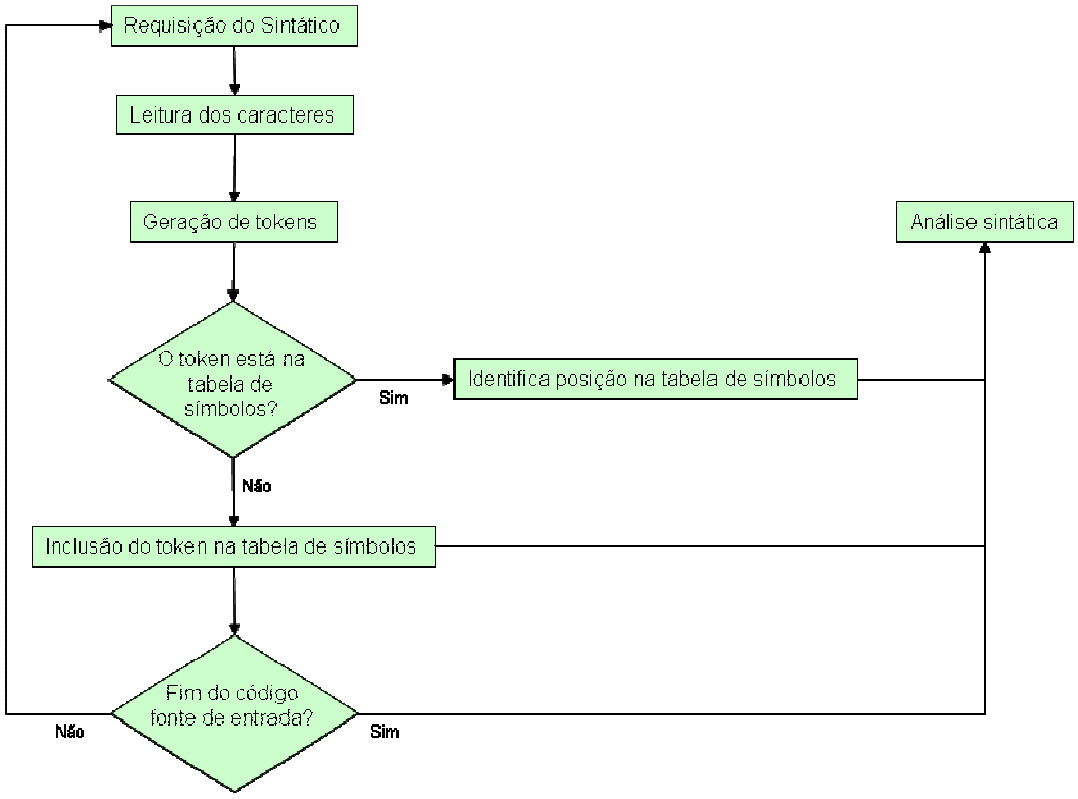
expressao = (((identificador | numero | “(” expressão “)”) { “^” (identificador | numero | “(” expressão “)”) }) { (“*” | “/”) ((identificador | numero | “(” expressão “)”) { “^” (identificador | numero | “(” expressão “)”) }) }) { (“+” | “-”) (((identificador | numero | “(” expressão “)”) { “^” (identificador | numero | “(” expressão “)”) }) { (“*” | “/”) ((identificador | numero | “(” expressão “)”) { “^” (identificador | numero | “(” expressão “)”) }) }) } .

3 Análise Léxica

O módulo de análise léxica é o responsável por fazer a interface entre o texto-fonte e os programas encarregados de sua análise e tradução. Sua missão fundamental é a de, a partir do texto de entrada, fragmentá-lo em seus componentes básicos, identificando trechos elementares completos e com identidade própria.

Uma vez identificadas, estas partículas do texto-fonte devem ser classificadas segundo o tipo a que pertencem. O analisador léxico mapeia a cadeia de entrada em uma cadeia de partículas definidas pelos pares (classe, valor). Cada uma dessas partículas é chamada de *token* (ou átomo), onde a classe representa a informação utilizada pelo analisador sintático, que é o tipo do *token*, e o valor indica a informação utilizada pela analisador semântico.

A figura a seguir indica a seqüência de ações do analisador léxico:



A linguagem utilizada apresenta os seguintes tipos de *tokens*:

Terminal	Tipo do átomo	Informação Complementar
Palavra reservada	A própria palavra reservada	Exemplos: if, else, while, print, scan, call, etc.
ID	<<identificador>>	Índice do identificador na tabela de símbolos
NUM	<<número>>	O valor numérico associado ao número
OPER	O próprio operador	Irrelevante
COMP	O próprio comparador	Irrelevante
DELIM	O próprio delimitador	Irrelevante
ASPA	" (aspas)	Irrelevante
END_CMD	; (ponto-e-vírgula)	Irrelevante
COMA	, (vírgula)	Irrelevante
TIPO_INT	<<tipo>>	Irrelevante
TIPO_CHAR	<<tipo>>	Irrelevante
TIPO_BOOLEAN	<<tipo>>	Irrelevante

Esses *tokens* são identificados pelo analisador léxico e, após validados, são enviados ao analisador sintático. Apresentamos a seguir as representações formais desses átomos através de uma linguagem regular:

- Identificador: (letra) (letra | dígito)*
- Comparador: == | != | < | > | <= | >=
- Operador: - | + | * | / | ^
- Delimitador: [|] | (|)
- Comentário: @ (letra | dígito)*@

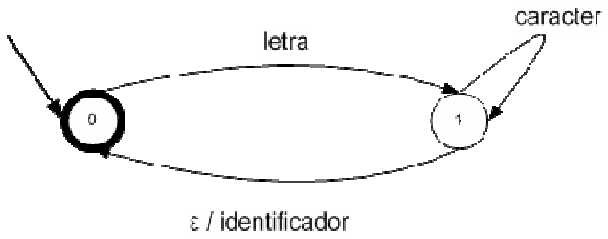
Por questões de simplificação, foram utilizadas as seguintes equivalências:

- letra = A | B |…| Z | a | b |…| z
- dígito = 0 | 1 | 2 |...| 9
- número = (dígito)*

Para melhor compreensão do analisador léxico, os autômatos de reconhecimento foram montados separadamente, e a sua unificação num autômato único é mostrada ao final.

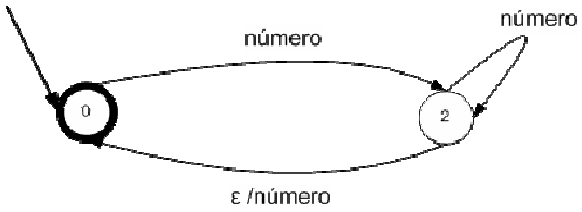
3.1 Identificador

O *token* de identificador é caracterizado por uma cadeia de caracteres iniciada por uma letra.



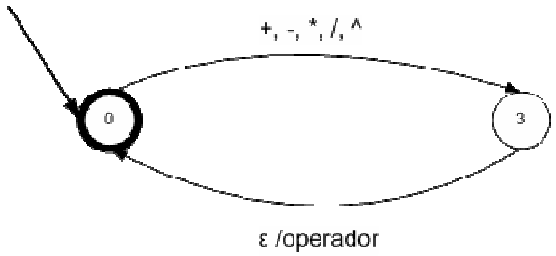
3.2 Número

O *token* de número identifica cadeias de algarismos.



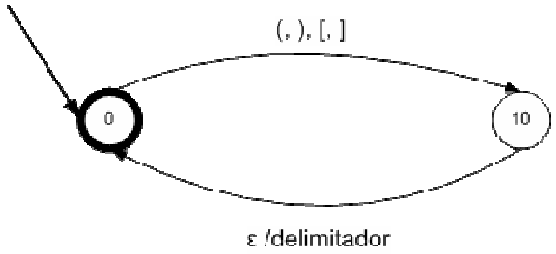
3.3 Operadores

Para executar as operações matemáticas, o léxico reconhece os operadores básicos: adição (+), subtração (-), multiplicação (*), divisão (/) e potenciação (^).



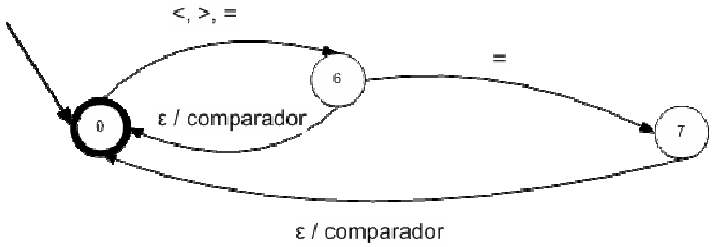
3.4 Delimitador

Os delimitadores, utilizados em operações matemáticas e para passar os parâmetros de vetores e matrizes, também possuem uma rotina para serem reconhecidos.



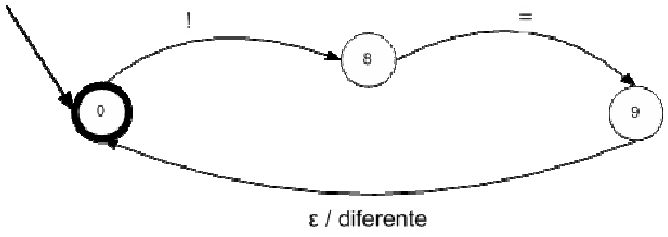
3.5 Comparador

No compilador desenvolvido, os comparadores são utilizados nas rotinas condicionais. Assim, eles também devem ser reconhecidos como *tokens*.



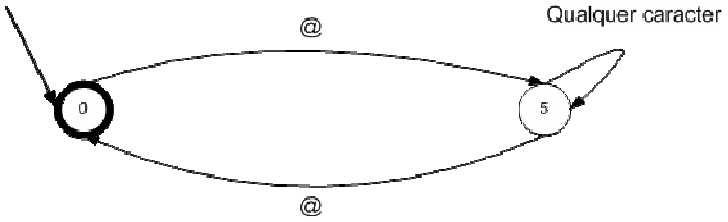
3.6 Diferente

No caso do comparador de diferença, foi necessário criar um novo reconhecedor, pois o caractere '!' não pode aparecer separado do caractere '='.



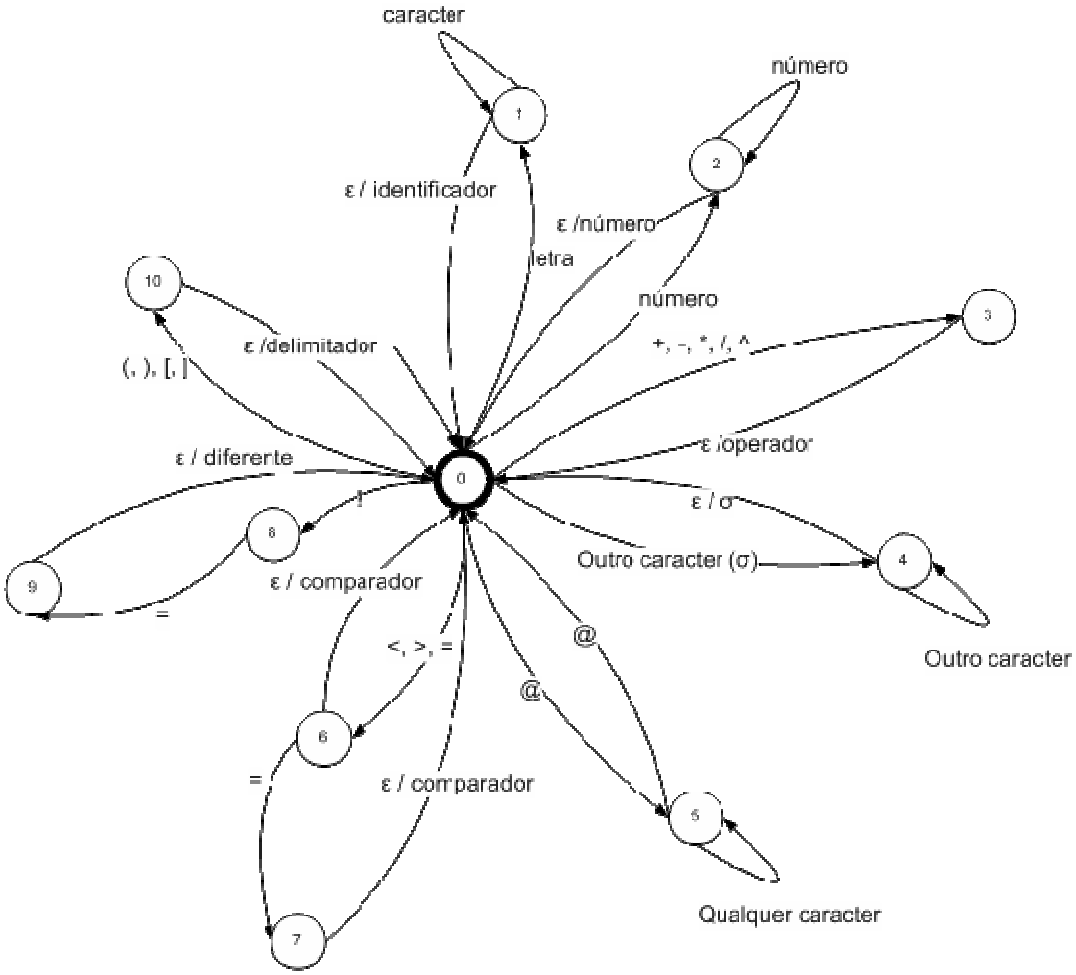
3.7 Comentário

Para que o programador possa inserir seus comentários no código, foi criado um meio para identificar essa função: os comentários devem ser inseridos entre dois caracteres '@'.



3.8 Autômato Final

Assim, unindo todos os reconhecedores num só diagrama, o analisador léxico projetado teria a seguinte representação:

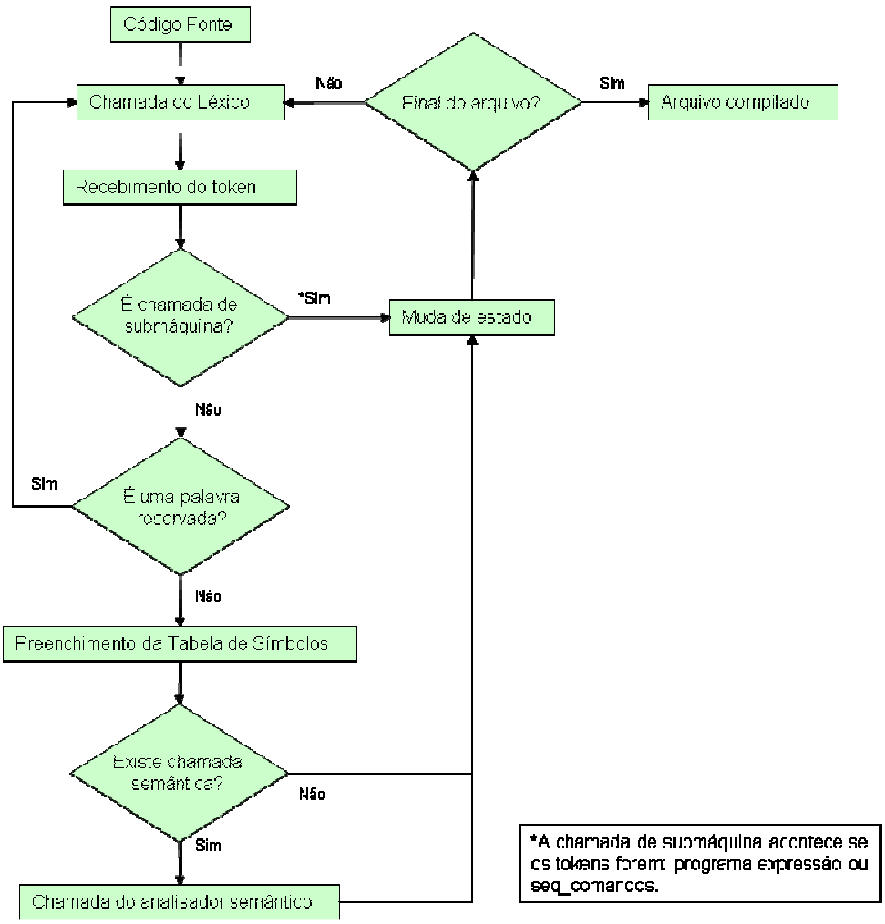


4 Análise Sintática

O analisador sintático é o módulo mais importante deste compilador, pois ele é orientado por sintaxe, ou seja, é o analisador sintático que controla as atividades do compilador. A função principal deste módulo é promover a análise da seqüência com que os átomos componentes do texto-fonte se apresentam e, a partir disso, montar sua árvore de sintaxe, com base na gramática da linguagem-fonte.

Outras funções tipicamente atribuídas ao analisador sintático são detecção de erros de sintaxe, recuperação de erros, correção de erros, ativação do analisador léxico, ativação de rotinas de análise semântica, ativação de rotinas de síntese do código-objeto, entre outras.

A figura a seguir mostra o diagrama de funcionamento do analisador sintático:



Neste projeto, definida a linguagem e gerado o analisador léxico, passou-se a trabalhar no módulo de análise sintática. Foi utilizado o método de construção baseado em autômato de pilha estruturado (APE).

4.1 Geração Automática dos Autômatos

Na primeira etapa para a criação do APE, foram utilizadas as expressões fundamentais criadas a partir da descrição reduzida em notação de Wirth mostrada anteriormente. A definição das expressões seguiu o critério dos não-terminais com recursividade central, sendo possível concluir que as máquinas finais tratariam a seqüência de comandos, as expressões, as operações booleanas e o programa principal.

Em seguida, foi utilizado o algoritmo de definição dos estados, seguindo as regras de transição determinadas pela notação de Wirth. Os estados encontrados podem ser vistos nas expressões abaixo:

(SC) seq_comandos =

$\bullet_0$ { $\bullet_1$ ($\bullet_1$ T $\bullet_4$ I $\bullet_5$

| $\bullet_1$ I $\bullet_6$ “=” $\bullet_7$ E $\bullet_8$

| $\bullet_1$ “If” $\bullet_9$ ($\bullet_9$ E $\bullet_{11}$ C $\bullet_{12}$ E $\bullet_{13}$ | $\bullet_9$ B $\bullet_{14}$) $\bullet_{10}$ “then” $\bullet_{15}$ SC $\bullet_{16}$ [$\bullet_{16}$ ”else” $\bullet_{18}$ SC $\bullet_{19}$] $\bullet_{17}$ “endif” $\bullet_{20}$

| $\bullet_1$ “switch” $\bullet_{21}$ I $\bullet_{22}$ “case” $\bullet_{74}$ I $\bullet_{23}$ SC $\bullet_{24}$ “break” $\bullet_{25}$ { $\bullet_{26}$ “case” $\bullet_{27}$ SC $\bullet_{28}$ “break” $\bullet_{29}$ } $\bullet_{26}$ “endswitch” $\bullet_{30}$

| $\bullet_1$ “while” $\bullet_{31}$ ($\bullet_{31}$ E $\bullet_{33}$ C $\bullet_{34}$ E $\bullet_{35}$ | $\bullet_{31}$ B $\bullet_{36}$) $\bullet_{32}$ “do” $\bullet_{37}$ SC $\bullet_{38}$ “endwhile” $\bullet_{39}$

| $\bullet_1$ “for” $\bullet_{40}$ “(“ $\bullet_{41}$ ($\bullet_{41}$ I $\bullet_{43}$ “=” $\bullet_{44}$ E $\bullet_{45}$) $\bullet_{42}$ “;” $\bullet_{46}$ ($\bullet_{46}$ E $\bullet_{48}$ C $\bullet_{49}$ E $\bullet_{50}$ | $\bullet_{46}$ B $\bullet_{51}$) $\bullet_{47}$ “;” $\bullet_{52}$ E $\bullet_{53}$ “)” $\bullet_{54}$ SC $\bullet_{55}$ "endfor” $\bullet_{56}$

| $\bullet_1$ “scan” $\bullet_{57}$ I $\bullet_{58}$

| $\bullet_1$ “print” $\bullet_{59}$ ($\bullet_{59}$ E $\bullet_{61}$ | $\bullet_{59}$ “"” $\bullet_{62}$ E $\bullet_{63}$ “"” $\bullet_{64}$) $\bullet_{60}$

| $\bullet_1$ “call” $\bullet_{65}$ NM $\bullet_{66}$ “(“ $\bullet_{67}$ [$\bullet_{68}$ I $\bullet_{69}$ { $\bullet_{70}$ “,” $\bullet_{71}$ I $\bullet_{72}$ } $\bullet_{70}$] $\bullet_{68}$ ”)” $\bullet_{73}$) $\bullet_2$ “;” $\bullet_3$ } $\bullet_1$.

(E) expressao =

•0 (•0 (•0 (•0 I •2 | •0 N •3 | •0 “(“ •4 E •5 “)” •6) •1 { •2´ “^” •3´ (•3´ I •5´ | •3´ N •6´ | •3´ “(“ •7 E •8 “)” •9) •4´ } •2´) •2´ { •10 (•10 “*” •12 | •10 “/” •13) •11 (•11 (•11 I •15 | •11 N •16 | •11 “(“ •17 E •18 “)” •19) •14 { •20 “^” •21 (•21 I •23 | •21 N •24 | •21 “(“ •25 E •26 “)” •27) •22 } •20) •20 } •10) •10 { •10´ (•10´ “+” •29 | •10´ “-” •30) •28 (•28 (•28 (•28 I •32 | •28 N •33 | •28 “(“ •34 E •35 “)” •36) •31 { •32 “^” •33 (•33 I •35 | •33 N •36 | •33 “(“ •37 E •38 “)” •39) •34 } •32) •32 { •40 (•40 “*” •42 | •40 “/” •43) •41 (•41 (•41 I •45 | •41 N •46 | •41 “(“ •47 E •48 “)” •49) •44 { •50 “^” •51 (•51 I •53 | •51 N •54 | •51 “(“ •55 E •56 “)” •57) •52 } •50) •50 } •40) •40 } •10´ .

(B) boolean =

•0 (•0 “(”•2 B •3 “)”•4 | •0 [•0 “not” •6] •5 I •7) •1 { •8 (•8 “and” •10 | •8 “or” •11) •9 (•9 “(”•13 B •14 “)”•15 | •9 [•9 “not” •17] •16 I •18) •12 } •8 .

programa =

•0 { •1 “func” •2 T •3 NM •4 “(“ •5 (•5 T •6 I •7) •7 { •8 “,” •9 (•9 T •10 I •11) •11 } •8 “)” •12 SC •13 “return” •18 (•18 I •20 | •18 N •21) •19 “endfunc” •14 } •1 “main” •15 SC •16 “end” •17 .

4.2 Redução dos Autômatos

Para a redução dos autômatos gerados na etapa anterior, foram montadas tabelas de transição para cada um deles. O processo utilizado seguiu os seguintes passos:

1. Eliminação das transições em vazio;
2. Eliminação dos estados não-acessíveis;
3. Eliminação dos estados equivalentes.

As tabelas finais de cada expressão estão indicadas a seguir. A legenda utilizada foi:

Estado final antes da redução
Transições equivalentes
Estado a ser eliminado por não ser acessível
Estado a ser eliminado por equivalência direta
Estado a ser eliminado por equivalência indireta
Estado final após a redução

4.2.1 Programa

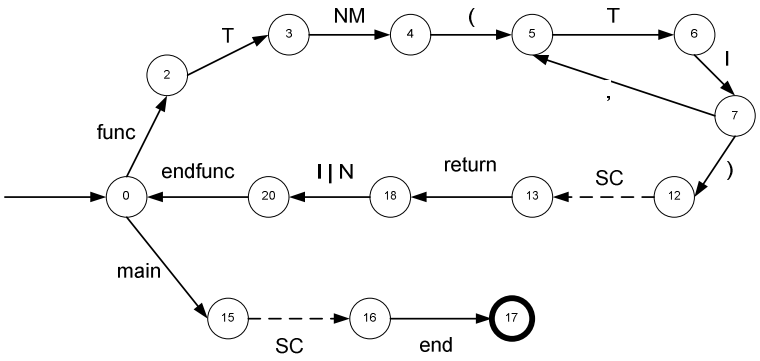
O autômato inicial possuía 22 estados. Após a análise, as equivalências encontradas foram as seguintes:

Equivalências
0, 1 e 14
7, 8 e 11
5 e 9
6 e 10
20 e 21

A tabela utilizada para a redução do autômato de programa pode ser vista na figura a seguir:

		Entrada														REDUÇÃO	
		func	NM	T	I	(	,	)	SC	return	N	endfunc	main	end	ε	Acessível	Considerado
Estado	0	2											15		1	1	2
	1	2											15				
	2			3												3	4
	3		4													5	6
	4					5										7	8
	5			6												9	10
	6				7											11	12
	7						9	12							8	13	14
	8						9	12									
	9			10												15	16
	10				11											17	18
	11						9	12							8	19	20
	12								13							15	21
	13									18						22	23
	14	2											15		1	28	29
	15								16							3	30
	16													17		31	32
	17															33	34
	18				20						21					24	25
	19											14					
	20											14			19	26	27
	21											14			19	26	35

O diagrama do autômato correspondente ficou da seguinte forma:



4.2.2 Seqüência de Comandos (SC)

O autômato inicial possuía 75 estados. Após a análise, as equivalências encontradas foram as seguintes:

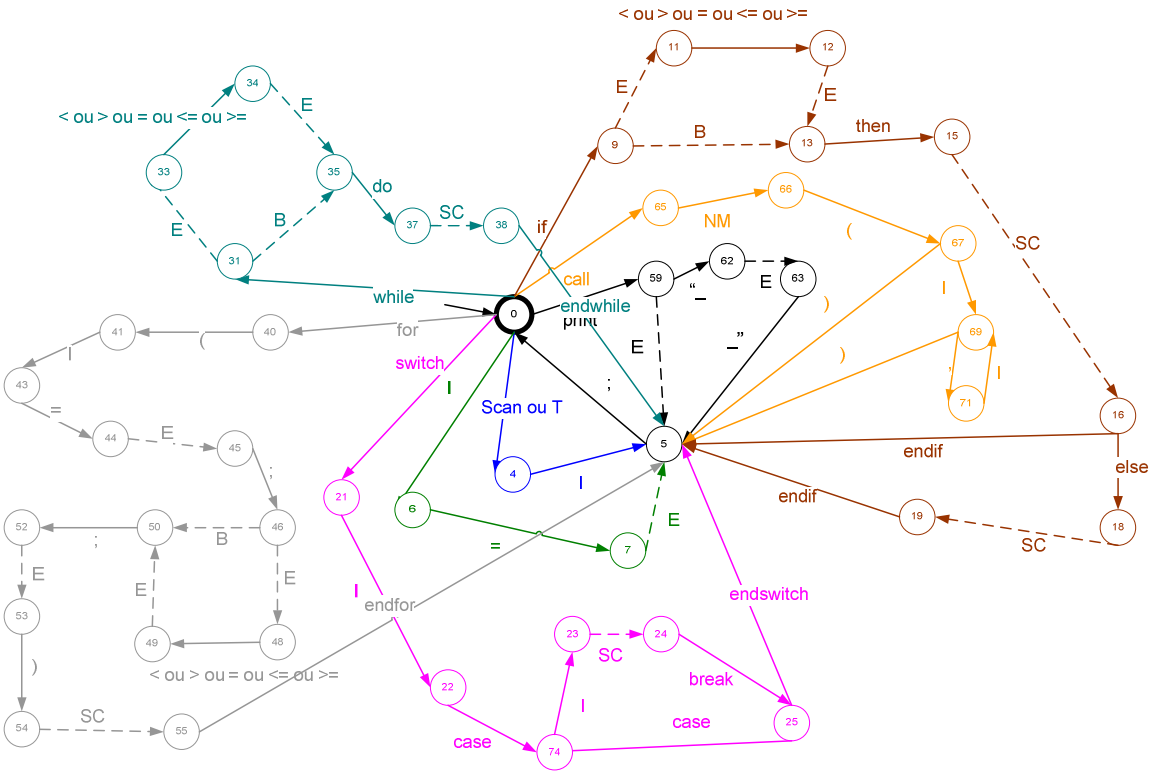
Equivalências
4 e 57
17 e 19
23 e 27
24 e 28
42 e 45
69 e 72
0, 1 e 3
10, 13 e 14
25, 26 e 29
32, 35 e 36
47, 50 e 51
2, 5, 8, 20, 30, 39, 56, 58, 60, 61, 64 e 73

A tabela utilizada para a redução do autômato de seqüência de comandos pode ser vista na figura a seguir:

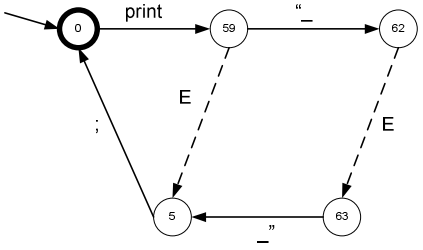
Estado	Entradas																																REDUÇÃO	
SC	T	I	=	E	if	C	B	then	SC	else	endif	switch	case	break	endswitch	while	do	endwhile	for	(	;	)	endfor	scan	print	" _	_ "	call	NM	-	ε	Acessível	Considerado
0	4	6			9							21				31			40					57	59			65			1	1	2
1	4	6			9							21				31			40					57	59			65					
2																					3												
3	4	6			9							21				31			40					57	59			65			1	7	8
4		5																														3	4
5																					3										2	5	6
6			7																													3	9
7				8																												10	11
8																					3										2	12	13
9				11			14																									3	14
10								15																									
11						12																										15	16
12				13																												17	18
13								15																							10	19	20
14								15																							10	15	31
15									16																							21	22
16										18	20																				17	23	24
17											20																						
18									19																							25	26
19											20																				17	27	28
20																					3										2	29	30
21		22																														3	32
22													74																			33	34
23									24																							35	36
24														25																		37	38
25													74		30																26	39	40
26													27		30																		
27									28																							41	42
28														29																		43	44
29													27		30																26	45	46
30																					3										2	41	47
31				33			36																									3	48
32																	37																
33						34																										49	50
34				35																												51	52
35																	37														32	53	54
36																	37														32	49	61
37									38																							55	56
38																		39														57	58
39																					3										2	59	60
40																				41												3	62
41		43																														63	64
42																					46												
43			44																													65	66
44				45																												67	68
45																					46										42	69	70
46				48			51																									71	72
47																					52												
48						49																										73	74
49				50																												75	76
50																					52										47	77	78
51																					52										47	73	89
52				53																												79	80
53																						54										81	82
54									55																							83	84
55																							56									85	86

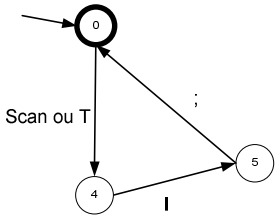
SC	Entradas																															REDUÇÃO	
	T	I	=	E	if	C	B	then	SC	else	endif	switch	case	break	endswitch	while	do	endwhile	for	(	;	)	endfor	scan	print	" _	" _	call	NM	-	ε	Acessível	Considerado
56																					3										2	87	88
57		58																														3	90
58				61																	3										2	91	92
59																										62						3	93
60																					3										2		
61																					3										60	94	95
62				63																												94	96
63																											64					97	98
64																					3										60	99	100
65																													66			3	101
66																				67												102	103
67		69																				73									68	104	105
68																						73											
69																						73								71	70	106	107
70																						73								71	68		
71		72																														108	109
72																														71	70	110	111
73																					3										2	112	113
74		23																															

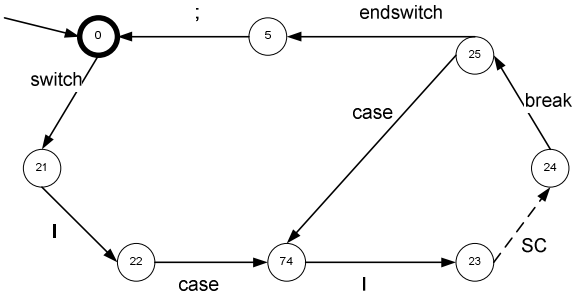
O diagrama do autômato correspondente ficou da seguinte forma:

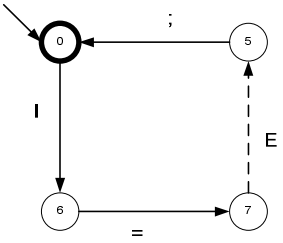


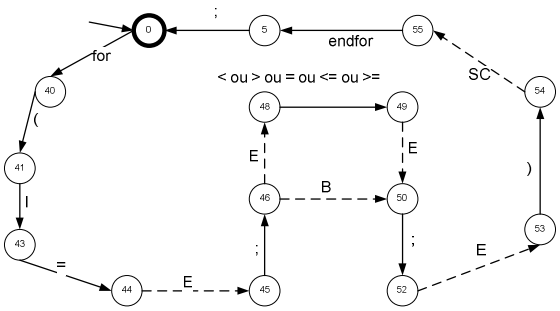
Para maior clareza, cada comando foi desenhado separadamente, conforme as figuras abaixo:

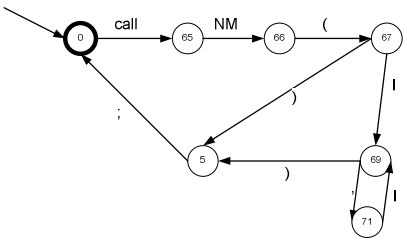

print
"
E
E
;
"

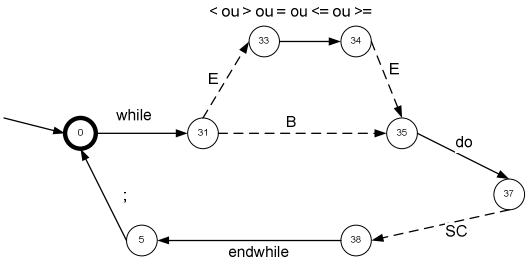

Scan ou T
;
I

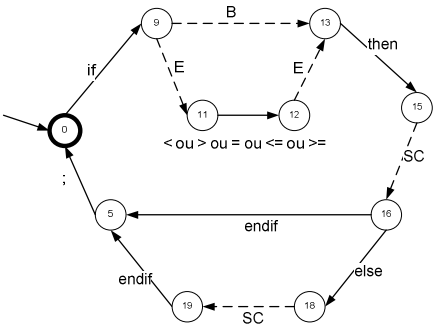

;
endswitch
switch
case
break
I
case
I
SC


;
I
E
=


;
endfor
for
< ou > ou = ou <= ou >=
SC
(
E
E
B
)
I
;
E
=
E
;


call
NM
(
)
I
;
)
I


< ou > ou = ou <= ou >=
while
E
E
B
do
;
SC
endwhile


B
if
then
E
E
< ou > ou = ou <= ou >=
SC
;
endif
endif
else
SC

4.2.3 Expressão (E)

O autômato inicial possuía 58 estados. Após a análise, as equivalências encontradas foram as seguintes:

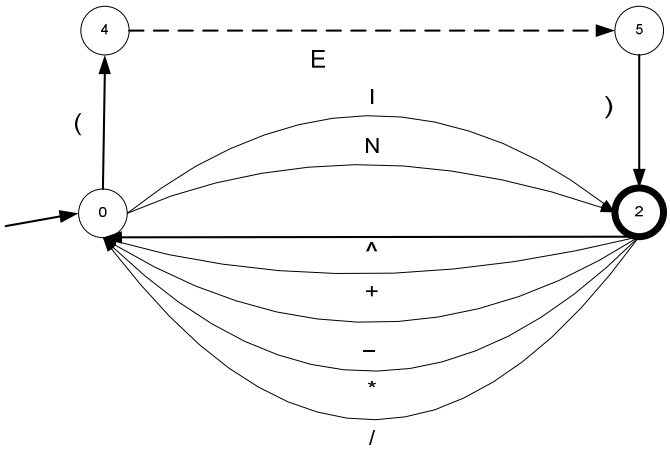
Equivalências
0, 3, 12, 15, 21 e 29
4, 7, 17 e 25
5, 8, 18 e 26
15, 32 e 45
42 e 51
48 e 56

A tabela utilizada para a redução do autômato de expressão pode ser vista na figura a seguir:

Estado	Entradas											REDUÇÃO	
E	I	N	(	E	)	^	*	/	+	-	ε	Acessível	Considerado
0	2	3	4									1	2
1						3'	12	13	29	30	2'		
2						3'	12	13	29	30	1	3	4
3						3'	12	13	29	30	1	3	66
4				5								3	67
5					6							68	69
6						3'	12	13	29	30	1	70	71
2'						3'	12	13	29	30	10		
3'	5'	6'	7									5	6
4'						3'	12	13	29	30	2'		
5'						3'	12	13	29	30	4'	7	8
6'						3'	12	13	29	30	4'	7	9
7				8								7	10
8					9							72	73
9						3'	12	13	29	30	4'	74	75
10							12	13	29	30	10'		
10'									29	30			
11	15	16	17										
12	15	16	17								11	5	11
13	15	16	17								11	5	65
14						21	12	13	29	30	20		
15						21	12	13	29	30	14	12	13
16						21	12	13	29	30	14	12	59
17				18								12	60
18					19							61	62
19						21	12	13	29	30	14	63	64
20						21	12	13	29	30	10		
21	23	24	25									14	15
22						21	12	13	29	30	20		
23						21	12	13	29	30	22	16	17
24						21	12	13	29	30	22	16	18
25				26								16	19
26					27							20	21
27						21	12	13	29	30	22	22	23
28	32	33	34										
29	32	33	34								28	14	24
30	32	33	34								28	14	58
31						33	42	43	29	30	32		
32						33	42	43	29	30	40	25	26
33	35	36	37									25	48
34				35		33	42	43	29	30	32	25	57
35					36							49	50
36						33	42	43	29	30	31	49	51
37				38								49	52
38					39							53	54
39				35		33	42	43	29	30	34	55	56
40							42	43	29	30	10'		

Estado	Entradas											REDUÇÃO	
	I	N	(	E	)	^	*	/	+	-	ε	Acessível	Considerado
41	45	46	47										
42	45	46	47								41	27	28
43	45	46	47								41	27	47
44						51	42	43	29	30	50		
45						51	42	43	29	30	44	29	30
46						51	42	43	29	30	44	29	41
47				48								29	42
48					49							43	44
49						51	42	43	29	30	44	45	46
50						51	42	43	29	30	40		
51	53	54	55									31	32
52						51	42	43	29	30	50		
53						51	42	43	29	30	52	33	34
54						51	42	43	29	30	52	33	35
55				56								33	36
56					57							37	38
57						51	42	43	29	30	52	39	40

O diagrama do autômato correspondente ficou da seguinte forma:



4.2.4 Boolean (B)

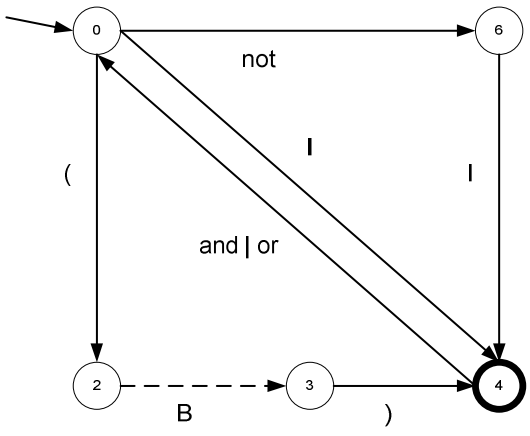
O autômato inicial possuía 19 estados. Após a análise, as equivalências encontradas foram as seguintes:

Equivalências
0, 10 e 11
2 e 13
3 e 14
4, 7, 15 e 18
6 e 17

A tabela utilizada para a redução do autômato de boolean pode ser vista na figura a seguir:

		Entrada								REDUÇÃO	
		(	)	B	not	I	and	or	ε	Acessível	Considerado
Estado	0	2			6	7			5	1	2
	1						10	11	8		
	2			3						3	4
	3		4							5	6
	4						10	11	1	7	8
	5					7	10	11	1		
	6					7			5	3	17
	7						10	11	1	3	18
	8						10	11			
	9	13			17	18			16		
	10	13			17	18			9	3	9
	11	13			17	18			9	3	16
	12						10	11	8		
	13			14						10	11
	14		15							12	13
	15						10	11	12	14	15
	16					18					
	17					18			16	10	19
	18						10	11	12	20	21

O diagrama do autômato correspondente ficou da seguinte forma:



5 Definição do Ambiente de Execução

Para que seja possível executar um programa-objeto, é preciso ter um ambiente que dê suporte às suas diversas funções, que não foram explicitamente implementadas pelo código-objeto. Tais funções englobam principalmente o gerenciamento de memória, a comunicação com o sistema operacional e com outros programas, e a execução de funções especiais, usualmente representadas por pacotes de rotinas, da biblioteca do sistema ou de aplicação.

O ambiente de execução deste projeto utiliza uma memória de 4k posições, sendo que as primeiras 3k posições são reservadas à execução do programa, e o restante das 1k posições podem ser utilizadas dinamicamente.

Algumas das rotinas básicas necessárias ao ambiente de execução são:

- #NEWLINE: esta rotina promove a impressão de uma linha do dispositivo de entrada/saída utilizado pelos comandos de entrada e de saída da linguagem.
- #CONVERT: esta rotina tem como função converter para a notação decimal o conteúdo do acumulador, preparando-o para impressão.
- #READ: esta rotina é responsável pela leitura de dados, fazendo a conversão de uma cadeia de caracteres (que devem ser interpretados como um número decimal) para a notação utilizada pelo acumulador, armazenando nele o resultado obtido.
- #STOP: esta rotina faz a interface com o sistema operacional, acionando uma chamada de supervisor para parar o processamento e retornar o controle ao sistema.

Ainda deveriam ser acrescentadas à lista rotinas que tratassem sub-rotinas, passagem de parâmetros, expressões aritméticas, etc. Assim, seria necessária a criação de pilhas para que seja possível armazenar endereços de retorno de sub-rotinas, tratamento de prioridade de operações nas expressões aritméticas, tratamento de escopos, entre outras funções.

5.1 Montador e Máquina Virtual

O montador tem como função transformar o código-objeto gerado pelo compilador em código de máquina. Dessa forma, é possível executar o programa, carregando esse código de máquina na máquina virtual.

6 Estruturas de Dados e Algoritmos do Compilador

Para que seja possível a implementação deste compilador, as estruturas utilizadas foram a tabela de símbolos e algumas pilhas de alto-nível.

6.1 Tabela de Símbolos

Uma vez extraídos do texto-fonte pelo analisador léxico, os identificadores são coletados em uma tabela, e a cada um é associado um código único. A cada ocorrência de um identificador no texto do programa-fonte, a tabela de símbolos é consultada ou alterada. Pelo fato de um ponto qualquer do programa poder pertencer a escopos (domínios) diferentes

CAMPO	TIPO	OBJETIVO
VALOR	*String*	Informação Complementar
TOKEN	*Int*	Tipo do átomo
TIPO	*Int*	Identificação de variável
PROXIMO	*Ponteiro*	Tratamento de escopo
ANTERIOR	*Ponteiro*	Tratamento de escopo

Token - Código único que identifica o tipo de átomo que foi recolhido pelo analisador léxico, podendo ser Identificador (**ID**), Números (**NUM**), Palavras reservadas (**IF, ELSE, MAIN, FUNC, THEN, ASPA, WHILE, ENDIF, END, DO, ENDWHILE, PRINT, SCAN, CALL**), Delimitadores: "(", ")", "[", "]" (**DELIM**), Operadores: "+", "-", "/", "*" (**OPER**), Comparadores: ">", "<", "=" (COMP), Fim de comando: ";" (END_CMD), Separadores: "," (**COMA**), Tipos: int, char, boolean (**TIPO_INT, TIPO_CHAR, TIPO_BOOLEAN**)

Valor - Representa uma informação complementar relativa ao átomo coletado. Varia de acordo com cada tipo

Tipo - Responsável por identificar qual dos átomos recolhidos são variáveis e qual o tipo de dado a ele associado: Inteiro (int), Caractere (char), Booleana (boolean).

Para o tratamento de diferentes escopos no programa, definidos por interações ou funções distintas, optou-se pela utilização de tabelas de símbolos aninhadas. Neste caso, a cada mudança de escopo, cria-se uma nova tabela de símbolos que será única para este domínio específico, sendo assim todos os valores nela presentes serão próprios do escopo

(não interferindo nas demais regiões do código nem na tabela referenciada). Visando esta nova manipulação, duas novas estruturas são acrescentadas na tabela de símbolos:

Próximo – Atributo específico para átomos do tipo "função" (delimitadores de escopo). É um ponteiro para uma outra estrutura do tipo tabela de símbolos. Indica que a partir deste ponto em diante, todos os novos átomos a serem recolhidos serão inseridos nesta outra tabela (limitando portanto os escopos)

Anterior – Atributo considerado no fim de cada tabela referenciada. Será utilizado para retornar ao escopo de ordem superior (definido pelo fim de uma função). Delimita o domínio dos átomos considerados.

A função abaixo cria uma tabela de símbolos e atributos:

```
1. struct token insere (char s[], int token, int *indice, struct entrada *tab_simbolos){
2. struct token resultado;
3. resultado = busca (tab_simbolos, *indice, s);
4. if (resultado.posicao != -1)
        i. return resultado;
5. tab_simbolos [*indice].token = token;
6. tab_simbolos [*indice].lexptr = strdup(s);
7. printf ("lexptr: %s, token: %d\n",
        i. tab_simbolos [*indice].lexptr, tab_simbolos [*indice].token);
8. resultado.tkval = token;
9. resultado.posicao = *indice;
10. (*indice)++;
11. return resultado;

    }
```

A estrutura descrita acima mostra a inserção dos *tokens* recebidos na tabela de símbolos. Para realizar essa inserção utilizamos um índice, que identifica a linha da tabela. As colunas são representadas pelo tipo de dado a ser armazenado. Dessa forma, a tabela seria da seguinte forma:

ID	token	lexptr
1	x	m
2	y	n
3	z	o
…	…	…

A estrutura *resultado•posição* indica se a posição está sendo utilizada ou não. Se ela estiver vazia (linha 4), preenche a tabela com os dados coletados (linhas 5 e 6). Depois de adicionado os dados na tabela o índice é incrementado.

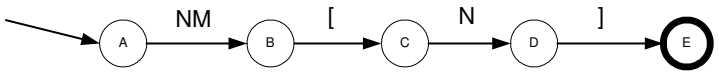
A função abaixo busca os dados na tabela de símbolo:

```
1. struct token busca (struct entrada *tab_simbolo, int size, char *palavra){
2. int i;
3. struct token resultado;
4. resultado.posicao = -1;
5. for (i=0; i<size; i++){
6. if (!strcmp (tab_simbolo [i].lexptr, palavra)){
7. resultado.posicao = i;
8. resultado.tkval = tab_simbolo [i].token;
   }
   }
9. return resultado;
   }
```

Inicializado com valor de -1 (linha 4), o *resultado•posicao*, indica a linha da tabela que está sendo lida. Então, incrementando-se o valor de i (linha 5), lê-se a tabela até se encontrar o dado desejado. Se o símbolo for encontrado (linha 6), retorna-se a posição e seu respectivo valor.

6.2 Agregados Homogêneos

Por simplificação de projeto, em nível de modelagem, considerou-se as estruturas do tipo vetor (agregados homogêneos) como componentes do grupo de identificadores. Para a etapa de implementação, porém, tais estruturas receberam um tratamento diferenciado. Considera-se cada token específico que compõem estas estruturas: NM (nome do vetor), [ ] (delimitadores), N (tamanho do vetor).



Assim que são declarados, considera-se seu tamanho e tipo, previamente considerados, para alocar na memória um espaço suficiente para armazenamento de suas posições. Uma vez delimitada através do processo de linearização de vetor, consegue-se apontar cada posição

específica do mesmo diretamente na memória, através da fórmula (calculada em tempo de execução):

VET[N] = VET[1] + N*TAMANHO(tipo)

VET[N]: Endereço da posição N do vetor a ser apontada na memória;
VET[1]: Endereço da posição inicial do vetor na memória (conhecido em tempo de compilação);
N: Índice da posição desejada do vetor (conhecido em tempo de execução);
TAMANHO(tipo): Tamanho do tipo de dado que será armazenado pelo vetor (conhecido em tempo de compilação). Ex. *int* – tamanho de 32 bits

Para geração do código de máquina correspondente ao cálculo da expressão acima representada, toda vez que uma estrutura do tipo vetor é declarada, gera-se uma estrutura equivalente a seguinte (correspondente no alto nível):

```
        TEMP    /00
        MEMO    /4000
        END     /00
        INC     /02
        <AÇÕES>
        LD   XX
        ....
        MM   TEMP
        </ACÕES>
        LD
MEMO
        +       END
        +       INC
        MM      EXEC
        LD      TEMP
```

Variável Temporária
Inicio da posição de armazenagem de dados
Off-set
Incremento de acordo com o tamanho do dado armazenado

Ações realizadas terão seu valor armazenado em TEMP

Carrega o endereço inicial na memória
Soma o off-set
Soma o tamanho do dado a ser armazenado
Nova posição do vetor a ser considerada

6.3 Pilhas de Alto-Nível

O projeto desse compilador utiliza-se de 5 pilhas. A função de cada uma delas está descrita abaixo:

1. ***Pilha de operandos***: empilha todos os operandos das expressões aritméticas. Esses operandos são desempilhados de acordo com a análise da precedência dos operadores encontrados na pilha de operadores.

2. ***Pilha de operadores***: empilha todos os operadores das expressões aritméticas, incluindo parênteses. É utilizado o método de *look ahead* para verificar se o próximo termo a ser colocado na pilha é prioritário sobre o que está atualmente no topo. Se for prioritário, esse valor é empilhado. Se não, desempilha-se o operador juntamente com os dois valores do topo da pilha de operando e realiza-se a operação.

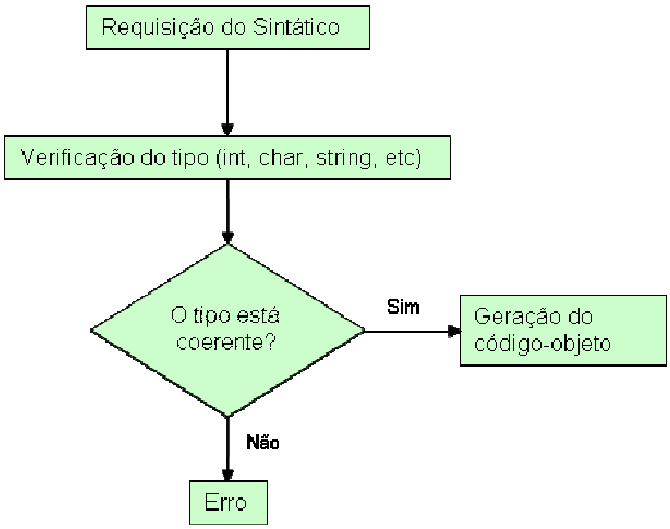
3. ***Pilha de escopos***: nessa pilha são colocados os valores da variável *rot_escopo*. Assim, toda vez que uma nova função é chamada, incrementa-se e empilha-se o valor da variável. A todas as variáveis dessa função, são concatenados os valores de *rot_escopo*. Quando o analisador sintático receber o *token* de identificação de fim de função (“ENDFUNC”) esse valor é desempilhado. Fazendo isso, evita-se que variáveis iguais que pertençam a escopos diferentes sejam igualmente identificadas na tabela de símbolos.

4. ***Pilha de funções***: os nomes das funções vão sendo empilhados nesta estrutura a cada vez que as funções forem chamadas, e desempilhados quando ocorre o retorno das mesmas, identificado pela presença do *token* de fim de função. Caso o programador faça uma função chamar outra, o *token* “ENDFUNC” deve funcionar corretamente.

5. ***Pilha de comando***: nessa pilha são colocados os valores da variável *rot_comando* que fazem referência aos “IF”s e “WHILE”s que são utilizados no código-fonte. Assim, toda vez que um desses comandos é lido, incrementa-se o valor da variável, concatena-se ao rótulo do comando e empilha-se o valor. Quando o sintático receber “ENDIF” e “ENDWHILE”, esse valor é desempilhado e, novamente, concatenado ao rótulo. Dessa forma, garantimos que o compilador reconheça à estrutura corretamente.

7 Semântica Dinâmica

O analisador semântico tem seu funcionamento indicado pelo diagrama a seguir:



7.1 Rotinas Semânticas

As rotinas semânticas necessárias para este projeto estão descritas a seguir.

7.1.1 Expressão Aritmética

Expressao		Entradas									
		I	N	(	E	)	+	-	*	/	SAÍDA
Estado	0	2 / **A**	2 / **A**	4 / **B**							
	2						0 / **D**	0 / **D**	0 / **E**	0 / **E**	
	4				5						
	5					6 / C					**F**

A: ***Exp_00_02( )***

Empilha o identificador ou número recebido na pilha de operandos.
Verifica se o tipo da variável a ser utilizada é consistente com os tipos aceitos para o cálculo de expressões (no caso, somente variáveis declaradas como ***int*** poderão ser usadas na estruturação de expressões).

B: ***Exp_00_04( )***

Empilha o operador "(" recebido na pilha de operadores.

C: ***Exp_05_02( )***

Verifica o elemento armazenado no topo da pilha de operadores. Caso seja o símbolo "(", apenas o desempilha, caso contrário executa a rotina de geração de código (GeraCodigoExp).

D: ***Exp_02_00 ( )***

Verifica o elemento armazenado no topo da pilha de operadores. Caso seja o símbolo "+", "-", "/", "*", executa a rotina de geração de código (GeraCodigoExp), caso contrário empilha o operador recebido ("+" ou "-") na pilha de operadores.

E: ***Exp_02_00_Priority ( )***

Verifica o elemento armazenado no topo da pilha de operadores. Caso seja o símbolo "/", "*", executa a rotina de geração de código (GeraCodigoExp), caso contrário empilha o operador recebido ("*" ou "/") na pilha de operadores.

F: ***Exp_Saida( )***

Verifica o elemento armazenado no topo da pilha de operadores. Caso a pilha não esteja vazia, executa a rotina de geração de código (GeraCodigoExp). Lança o valor final da expressão calculada na variável temporária TEMP_EX. Gera:
"LD TEMP#N ;"
"MD TEMP_EX ;"

G: ***GeraCodigoExp( )***

i – Desempilha o valor contido no topo da pilha de operadores
ii – Desempilha os 2 elementos do topo da pilha de operandos (A e B)
iii – Gera o código específico para a operação encontrada no topo da pilha de operadores

Gera: "LD A ● ESP ;"

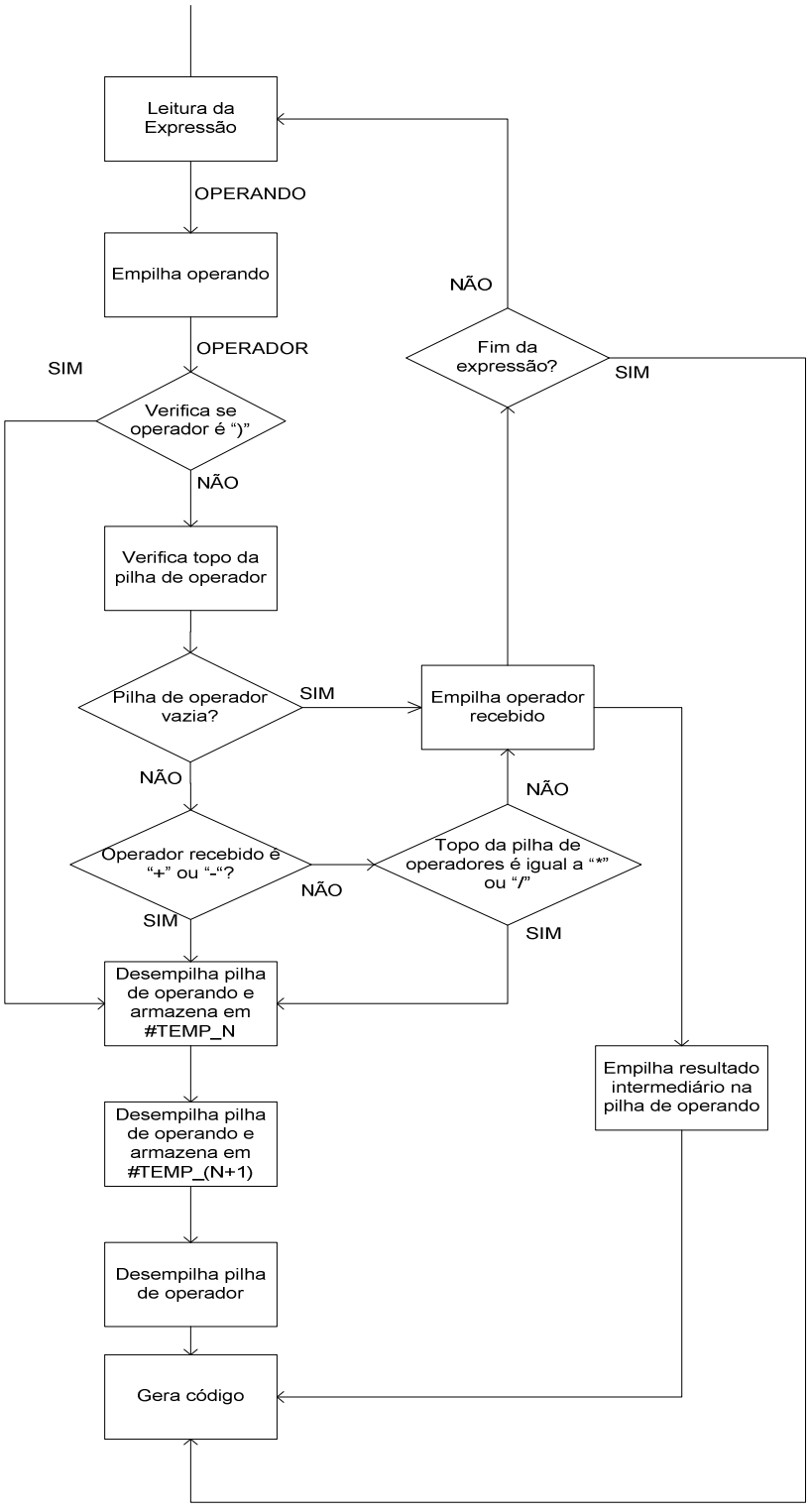
Caso (i) = "+" gera: "+ B ● ESP ;"
Caso (i) = "-" gera: "- B ● ESP ;"
Caso (i) = "*" gera: "* B ● ESP ;"
Caso (i) = "/" gera: "/ B ● ESP ;"

vi – Incrementa o contador de variáveis temporárias (TEMP#N)

Gera: "MD TEMP#N ;"

v – Empilha o valor de TEMP#N na pilha de operandos

O fluxograma abaixo apresenta a estruturação da rotina semântica de tradução de expressão aritmética:


Leitura da Expressão
OPERANDO
Empilha operando
OPERADOR
SIM
Verifica se operador é ")"
NÃO
Verifica topo da pilha de operador
Pilha de operador vazia?
SIM
Empilha operador recebido
NÃO
NÃO
Operador recebido é "+" ou "-"?
NÃO
Topo da pilha de operadores é igual a "*" ou "/"
SIM
SIM
Desempilha pilha de operando e armazena em #TEMP_N
Empilha resultado intermediário na pilha de operando
Desempilha pilha de operando e armazena em #TEMP_(N+1)
Desempilha pilha de operador
Gera código
NÃO
Fim da expressão?
SIM

7.1.2 Programa

Programa		Entrada													
		func	NM	T	I	(	,	)	SC	return	N	endfunc	main	end	SAIDA
Estado	0	2											15 / **A**		
	2			3											
	3		4 / **D**												
	4					5									
	5			6											
	6				7 / **B**										
	7						5	12							
	12								13						
	13									18					
	15								16						
	16													17 / **C**	
	17														
	18				20 / **D**						20 / **E**				
	20											0 / **F**			

OBS: ***FSP*** é um *stack pointer* que aponta para o topo da pilha de nome de funções.

A: ***Pro_00_02( )***

Gera cabeçalho de entrada para sinalizar o início do programa e começar o armazenamento correto das instruções de entrada nas posições adequadas de memória.
Gera: “ @/0”
“MAIN: SO = 0 ;”

B: ***Pro_06_07 ( )***

Para cada identificador caracterizado como parâmetro de entrada (I), carrega o valor específico armazenado na memória em cada argumento de entrada (representado pelo nome da função – NM – e um número que é fornecido pela variável arg_aux, incrementada a cada I recebido, para ser utilizado na seqüência de comandos dentro da função.
Gera: “LD NM ● arg_aux ;”
“MM I ;”
Zera a variável arg_aux para ser considerada na chamada de outras funções.

C: ***Pro_16_17 ( )***

Esta rotina está associada ao final do reconhecimento do programa.

Gera: "END: SO = 0 ;"
" HM ;"

Para cada um dos identificadores, declarados como variáveis, aloca-se neste ponto as áreas de memória necessárias. Posteriormente, faz o mesmo para as variáveis temporárias (TEMP#N, onde N é definido pelo valor armazenado no contador de variáveis temporárias). Para isso percorre-se as tabelas de símbolos em busca das entradas que correspondem a variáveis (coluna tipo). Elas terão seu valor concatenado ao número referente ao topo da pilha de escopos ESP (o qual é, a cada mudança de escopo, incrementado e empilhado – entrada de função), sendo representadas como rótulos na linguagem de máquina.

Gera: " @/3000"

Para cada variável encontrada, concatena-se o valor de ESP e gera-se:

" I ● ESP : JP /0 "

D: ***Pro_03_04 ( )***

Gera cabeçalho de entrada com o nome da função (NM) para sinalizar o início da estrutura previamente declarada. Incrementa o valor de FSP e empilha o nome da função (NM). Incrementa o valor de ESP (stack pointer da pilha de escopos), de rot_var (rótulo de variáveis) e empilha seu valor no topo da pilha de rótulos de escopo.

Gera: "NM SO = 0 ;"

E: ***Pro_18_20 ( )***

Armazena o valor final de retorno gerado pela função (um identificador I, representando uma variável, ou um número N) em uma posição específica da memória. Guarda o valor de "I" ou "N" em aux.

Gera: "LD aux"
"MM RETORNO ● FSP"

F: ***Pro_20_00 ( )***

Desempilha o valor armazenado no topo da pilha funções (apontado por FSP) para gerar o comando de retorno da sub-rotina para o programa chamador.

Gera: "RS NM"

7.1.3 Print, Scan, Declaração e Atribuição

PRINT / SCAN / DECLARAÇAO		Entrada									
		scan	T	I	;	=	E	print	"	"	SAIDA
Estado	0	4 / A	4 / A	6 / E				59			G
	4			5 / B							
	5				0						
	6					7					
	7						5 / F				
	59						5 / C		62		
	62						63 / D				
	63									5	

OBS: ESP é um stack pointer que aponta para o topo da pilha de rótulos de variáveis (delimitação de escopos). rot_var é variável numérica que será incrementada a cada mudança de escopo, seu valor será concatenado a cada variável identificada. O sinal ● indica concatenação.

A: ***Com_00_04 ( )***

Guarda o "token" recebido ("scan" ou "tipo") em uma variável auxiliar ("aux" no alto nível)

B: ***Com_04_05 ( )***

Ao receber um identificador (I), verifica o valor da variável auxiliar ("aux") para gerar a ação de entrada de informação ou declaração de variável.
Caso aux = "scan"

Gera: " GD = 0 ;"
"MD I ● ESP ;"

Caso aux = "tipo" (T), não realiza nenhuma ação semântica.

C: ***Com_59_05 ( )***

Recupera o valor da expressão e gera o comando de impressão de caracteres.
Gera: "LD TEMP_EX ;"
" PD = 1 ;"

D: ***Com_62_63 ( )***

Recupera o valor literal da expressão (a partir da tabela de símbolos) e gera o comando de impressão de caracteres. Cada caractere que compõe a string total será destacado e impresso separadamente pela máquina em formato ASCII.

Ex. “OI!”
Gera: “LD /00f4 ;”
“ PD = 1;”
“LD /0094 ;”
“ PD = 1;”
“LD /0012 ;”
“ PD = 1;”

E: ***Com_00_06 ( )***

Armazena o nome da variável de entrada em “entAux” (I).

F: ***Com_07_05 ( )***

Verifica se o tipo da variável a ser utilizada (do lado direito da igualdade) é consistente com o tipo encontrado do lado esquerdo da expressão (no caso, somente variáveis declaradas como int e char poderão ser usadas na estruturação de atribuições).
Faz a atribuição direta do valor da expressão para a variável armazenada em “entAux”.
Gera: “LD TEMP_EX”
“MD entAux”

7.1.4 Call

CALL		Entrada							
		call	NM	(	)	I	;	,	SAIDA
Estado	0	65							
	5						0 / C		
	65		66 / A						
	66			67					
	67				5	69 / B			
	69				5			71	
	71					69			

A: ***CALL_00_65 ( )***

Empilha o nome da função encontrada (NM) na pilha de rótulo de funções, cujo topo é identificado por FSP.

Para cada parâmetro declarado na função, armazena o valor do argumento em uma posição de memória que será utilizada diretamente pela seqüência de comandos da função. A variável arg_aux é incrementada a cada parâmetro encontrado e seu valor é concatenado ao nome da função para gerar um novo rótulo de posição de memória.
Para cada argumento de função, gera:
"MM FSP ● arg_aux"

B: ***CALL_67_69 ( )***

Função para chamar a sub-rotina específica que fora declarada.
Gera: "SC FSP"

C: ***CALL_05_00 ( )***

Recupera o valor retornado pela função para ser utilizado posteriormente pelo programa chamador.
Gera: "LD TEMP ● FSP"
Desempilha o valor armazenado em FSP para considerarmos o próximo nome de função armazenado na pilha e zera o valor de arg_aux.

7.1.5 If

IF		Entrada												
		if	E	<	>	==	>=	<=	;	SC	then	else	endif	SAIDA
Estado	0	9/**A**												
	5								0					
	9		11/**B**											
	11			12/**C**	12/**C**	12/**C**	12/**C**	12/**C**						
	12		13/**B**											
	13										15/**D**			
	15									16				
	16											18/**E**	5/**F**	
	18									19				
	19												5/**F**	

OBS: RSP é um stack pointer que aponta para o topo da pilha de rótulos de comando (rot_comando). O sinal "●" indica concatenação.

A: Incrementa-se o valor de *rot_comando* e empilha.

B: IF_09_11 ( )

i - Carrega o valor correspondente da última expressão calculada.
ii – Incrementa valor do N.
iii - Armazena no campo #TEMP_COND Nº.
Gera: "LD #TEMP_EX" ;
Gera: "MM#TEMP_COND Nº" ;
Guarda o valor gerado pela máquina "Expressão" em uma variável temporária.

C: IF_11_12 ( )

Guarda o sinal de comparação utilizado no teste condicional em uma variável chamada comparador.

D: IF_13_15 ( )

i - Realiza o teste.

Se comparador = ">"
Gera:
"LD #TEMP_COND 1 ;"
"- #TEMP_COND 2 ;"
"JN ELSE● RSP ;"
"JZ ELSE● RSP ;"

Se comparador = "<"
Gera:
"LD #TEMP_COND 2 ;"
"- #TEMP_COND 1 ;"
"JN ELSE ● RSP;"
"JZ ELSE● RSP ;"

Se comparador = ">="
Gera:
"LD #TEMP_COND 1 ;"
"- #TEMP_COND 2 ;"
"JN ELSE● RSP ;"

Se comparador = "<="

Gera:
"LD #TEMP_COND 2 ;"
"- #TEMP_COND 1 ;"
"JN ELSE● RSP ;"

Se comparador = "= ="
Gera:
"LD #TEMP_COND 1 ;"
"- #TEMP_COND 2 ;"
"JN ELSE● RSP ;"
"JP ELSE● RSP ;"

Se comparador = "!="
Gera:
"LD #TEMP_COND 1 ;"
"- #TEMP_COND 2 ;"
"JZ ELSE● RSP ;"

Realiza-se o teste para saber se a condição é satisfeita ou não. Dependendo do resultado, pode-se haver um salto na próxima instrução.

E: IF_16_18 ( )

Gera: "ELSE ● RSP SO=0 ;"
Desempilha o valor de RSP.

F: IF_16_05 ( )

Gera: "JP ENDIF● RSP ;"
Desempilha o valor de RSP.

OBS: Na pilha de rótulos de comando, são colocados os valores de rot_comando que fazem referência aos "IF"s, "ELSE"s e "WHILE"s que são utilizados no código-fonte. Assim, toda vez que um desses comandos é lido, incrementa-se o valor da variável rot_comando, concatena-o ao rótulo e empilha esse valor.

7.1.6 While

WHILE		Entrada											
		while	E	<	>	==	>=	<=	;	SC	do	endwhile	SAIDA
Estado	0	31 / A											
	5								0				
	31		33 / B										
	33			34 / C	34 / C	34 / C	34 / C	34 / C					
	34		35 / B										
	35										37 / D		
	37									38			
	38											5 / E	

A: Incrementa-se o valor de rot_comando e empilha.

B: WHILE_31_33 ( )

i - Carrega o valor correspondente da última expressão calculada.
ii – Incrementa valor do N.
iii - Armazena no campo #TEMP_COND Nº.
Gera: “LD #TEMP_EX” ;
Gera: “MM #TEMP_COND Nº” ;
Guarda o valor gerado pela máquina “Expressão” em uma variável temporária.

C: WHILE_33_34 ( )

Guarda o sinal de comparação utilizado no teste condicional em uma variável chamada comparador.

D: WHILE_35_37 ( )

i - Realiza o teste.
Se comparador = “>”
Gera:
“LD #TEMP_COND 2 ;”
“- #TEMP_COND 1 ;”
“JN DO ;”
“JP ENDWHILE● RSP ;”

Se comparador = “<”

Gera:
“LD #TEMP_COND 1 ;”
“- #TEMP_COND 2 ;”
“JN DO ;”
“JP ENDWHILE● RSP ;”

Se comparador = “>=”
Gera:
“LD #TEMP_COND 2 ;”
“- #TEMP_COND 1 ;”
“JN DO ;”
“JZ DO ;”
“JP ENDWHILE● RSP ;”

Se comparador = “<=”
Gera:
“LD #TEMP_COND 1 ;”
“- #TEMP_COND 2 ;”
“JN DO ;”
“JZ DO ;”
“JP ENDWHILE● RSP ;”

Se comparador = “= =”
Gera:
“LD #TEMP_COND 1 ;”
“- #TEMP_COND 2 ;’
“JZ DO ;”
“JP ENDWHILE● RSP ;”

Se comparador = “!=”
Gera:
“LD #TEMP_COND 1 ;”
“- #TEMP_COND 2 ;”
“JZ ENDWHILE● RSP ;’
“JP DO ;”

Realiza-se o teste para saber se a condição é satisfeita ou não. Dependendo do resultado, pode-se haver um salto na próxima instrução.

E: WHILE_38_05 ( )

Gera:
"JP WHILE ;"
"ENDWHILE SO = 0 ;"
Desempilha o valor de RSP.

8 Referências

- Neto, J.J. – Introdução à compilação
- Aho, A.V.; Sethi, R.; Ullman J. – Compilers: Principles, Techniques and Tools
- Sagra; Luzzatto – Compiladores: Implementação de Linguagens de Programação

Anexo I – Teste de Geração de Código

Programa de Teste:

```
main
      int variavel;
      variavel = (1 + 2 - 3 * 7);
end
```

Tabela de Símbolos gerada a partir do programa acima. Seus campos são:
(******* Tab simbolos: valor token tipo)***

```
***** Conteudo da tabela de funcao 0 *****
********** Tab simbolos: if 258 0
********** Tab simbolos: then 259 0
********** Tab simbolos: else 260 0
********** Tab simbolos: endif 261 0
********** Tab simbolos: main 265 0
********** Tab simbolos: end 266 0
********** Tab simbolos: while 272 0
********** Tab simbolos: do 271 0
********** Tab simbolos: endwhile 273 0
********** Tab simbolos: print 274 0
********** Tab simbolos: scan 275 0
********** Tab simbolos: int 276 0
********** Tab simbolos: boolean 277 0
********** Tab simbolos: char 278 0
********** Tab simbolos: , 279 0
********** Tab simbolos: call 280 0
********** Tab simbolos: func 267 0
********** Tab simbolos: return 268 0
********** Tab simbolos: endfunc 269 0
********** Tab simbolos: variavel 257 276
********** Tab simbolos: ; 270 0
********** Tab simbolos: = 264 0
********** Tab simbolos: ( 262 0
********** Tab simbolos: 1 256 0
********** Tab simbolos: + 263 0
********** Tab simbolos: 2 256 0
********** Tab simbolos: - 263 0
********** Tab simbolos: 3 256 0
********** Tab simbolos: * 263 0
********** Tab simbolos: 7 256 0
********** Tab simbolos: ) 262 0
```

Arquivo Assembly gerado a partir do programa de testes com base nas rotinas semânticas

```
                  @ /0
MAIN:             SO=0;
            LD    7;
            *     3;
            MD TEMP0
            LD #TEMP0
            -     2;
            MD TEMP1
            LD #TEMP1
            +     1;
            MD TEMP2
            LD TEMP2
            MM VARIAVEL
END:              SO=0;
                  HM;
                  @ /3000
VARIAVEL: JP      /00
```

Anexo II – Teste de Compilação Sintática

Programa gerado contendo todos os comandos implementados para validação das rotinas sintáticas, léxicas e verificação de escopo.

```
@ funcao sabao @
@ nao faz nada @
func int sabao (int var)
      int agape;
      print agape;
      return bla
endfunc

@ programa bonito @
@ ele comeca aqui @
main
      int variavel;
      int var;
      int tnirp;

      if variavel != var
      then
            while 1 < 2
            do
                  print tnirp;
                  variavel = (1 + 2 - 3);
                   bla = call sabao(y);
            endwhile;
      else
            print 3;
      endif;
end
@ e ele acaba aqui @
```

Tabela de Símbolos gerada a partir do programa acima. Seus campos são:
(******* Tab simbolos: valor token tipo)***

```
***** Conteudo da tabela de funcao 0 *****
********** Tab simbolos: if 258 0
********** Tab simbolos: then 259 0
********** Tab simbolos: else 260 0
********** Tab simbolos: endif 261 0
********** Tab simbolos: main 265 0
********** Tab simbolos: end 266 0
********** Tab simbolos: while 272 0
********** Tab simbolos: do 271 0
********** Tab simbolos: endwhile 273 0
********** Tab simbolos: print 274 0
********** Tab simbolos: scan 275 0
********** Tab simbolos: int 276 0
********** Tab simbolos: boolean 277 0
********** Tab simbolos: char 278 0
********** Tab simbolos: , 279 0
********** Tab simbolos: call 280 0
********** Tab simbolos: func 267 0
********** Tab simbolos: return 268 0
********** Tab simbolos: endfunc 269 0
********** Tab simbolos: variavel 257 276
********** Tab simbolos: ; 270 0
********** Tab simbolos: var 257 276
********** Tab simbolos: tnirp 257 276
********** Tab simbolos: != 264 0
********** Tab simbolos: 1 256 0
********** Tab simbolos: < 264 0
********** Tab simbolos: 2 256 0
********** Tab simbolos: = 264 0
********** Tab simbolos: ( 262 0
********** Tab simbolos: + 263 0
********** Tab simbolos: - 263 0
********** Tab simbolos: 3 256 0
********** Tab simbolos: ) 262 0
********** Tab simbolos: bla 257 0
********** Tab simbolos: sabao 257 0
********** Tab simbolos: y 257 0
```

```
***** Conteudo da tabela de funcao 1 *****
********** Tab simbolos: if 258 0
********** Tab simbolos: then 259 0
********** Tab simbolos: else 260 0
********** Tab simbolos: endif 261 0
********** Tab simbolos: main 265 0
********** Tab simbolos: end 266 0
********** Tab simbolos: while 272 0
********** Tab simbolos: do 271 0
********** Tab simbolos: endwhile 273 0
********** Tab simbolos: print 274 0
********** Tab simbolos: scan 275 0
********** Tab simbolos: int 276 0
********** Tab simbolos: boolean 277 0
********** Tab simbolos: char 278 0
********** Tab simbolos: , 279 0
********** Tab simbolos: call 280 0
********** Tab simbolos: func 267 0
********** Tab simbolos: return 268 0
********** Tab simbolos: endfunc 269 0
********** Tab simbolos: sabao 257 0
********** Tab simbolos: ( 262 0
********** Tab simbolos: var 257 0
********** Tab simbolos: ) 262 0
********** Tab simbolos: agape 257 276
********** Tab simbolos: ; 270 0
********** Tab simbolos: bla 257 0
```