

Departamento de Engenharia de Computação e Sistemas Digitais

**PCS 2056 – Linguagens e Compiladores**

**Relatório Final**

05/12/2011

Bruno Pezzolo dos Santos 5948816

Carla Guillen Gomes 5691366

Sumário

[Introdução 3](#_Toc310697939)

[Definição da Linguagem 4](#_Toc310697940)

[Descrição Informal 4](#_Toc310697941)

[Funções 4](#_Toc310697942)

[Repetição Condicional 5](#_Toc310697943)

[Decisão 5](#_Toc310697944)

[Entrada 5](#_Toc310697945)

[Saída 5](#_Toc310697946)

[Descrição Formal 5](#_Toc310697947)

[Notação BNF 5](#_Toc310697948)

[Notação Wirth 7](#_Toc310697949)

[Descrição Reduzida 9](#_Toc310697950)

[Análise Léxica 11](#_Toc310697951)

[Autômatos 12](#_Toc310697952)

[Implementação 15](#_Toc310697953)

[Análise Sintática 18](#_Toc310697954)

[Submáquinas do Autômato de Pilha Estruturado (APE) 18](#_Toc310697955)

[Implementação 23](#_Toc310697956)

[Definição do Ambiente de Execução 25](#_Toc310697957)

[Instruções da linguagem de saída 25](#_Toc310697958)

[Pseudoinstruções da linguagem de saída 26](#_Toc310697959)

[Características gerais 26](#_Toc310697960)

[Chamada de subrotina 28](#_Toc310697961)

[Retorno de subrotina 28](#_Toc310697962)

[Exemplo de programa traduzido 28](#_Toc310697963)

[Análise Semântica 30](#_Toc310697964)

[Testes 30](#_Toc310697965)

[Conclusão 30](#_Toc310697966)

# Introdução

Este documento descreve o compilador desenvolvido pelos alunos Bruno Pezzolo dos Santos e Carla Guillen Gomes na disciplina PCS2056 – Linguagens e Compiladores. O compilador foi desenvolvido com fins didáticos, sendo desconsideradas características como otimização de código-fonte, recuperação de erros e desempenho.

O compilador foi escrito na linguagem C e gera código objeto em linguagem simbólica composta por mnemônicos, aceita pela MVN fornecida. A MVN (Máquina de von Neumann) simula o Modelo de von Neumann como um processador simples rodando na máquina virtual Java (JVM).

A arquitetura do compilador é orientada à sintaxe, ou seja, o módulo de análise sintática gerencia as atividades através de requisições aos outros módulos, conforme representação a seguir:

Código objeto

Analisador Sintático

Analisador Léxico

Código fonte

Tokens

Tabela de símbolos

Solicita token

Analisador Semântico

As fases do desenvolvimento do compilador (definição da linguagem, análise léxica, análise sintática, definição do ambiente de execução e análise semântica) serão descritas nos próximos capítulos.

# Definição da Linguagem

A linguagem de alto nível criada é baseada na linguagem C. Os programas são formados por sequências de comandos separados por ponto e vírgula. Os programas se iniciam com a palavra **program**, seguida pelas definições de structs, declarações de variáveis, structs e/ou vetores e funções. Por fim, a estrutura **main {**<declarações> <comandos>**}** contém o programa principal a ser executado.

Um exemplo de programa está representado a seguir:

|  |
| --- |
| program  function int fatorial\_iterativo(int n) {    declare int fatorial;  fatorial = 1;   while (n > 0) {  fatorial = fatorial \* n;     n = n - 1;  }   return fatorial; }   main {  declare int fat;  fat = fatorial\_iterativo(10);  output fat; } |

## Descrição Informal

A seguir, apresenta-se a representação informal das principais funcionalidades definidas para a linguagem.

### Funções

|  |
| --- |
| **function** tipo identificador **(** parâmetro\_1**,** ..., parâmetro\_n **)** **{**  declaração\_1**;**  ...  declaração\_n**;**  comando\_1**;**  ...  comando\_n**;**  **}** |

### Repetição Condicional

|  |
| --- |
| **while** (condição) {  comando\_1**;**  ...  comando\_n**;**  } |

### Decisão

|  |
| --- |
| **if** (condição) {  comando\_1**;**  ...  comando\_n**;**  } |
| **if** (condição) {  comando\_1**;**  ...  comando\_n**;**  } **else** {  comando\_1**;**  ...  comando\_n**;**  } |

### Entrada

|  |
| --- |
| **input** identificador **;** |

### Saída

|  |
| --- |
| **output** identificador **;** |

## Descrição Formal

Para formalização da sintaxe definida, foram usadas as notações BNF e Wirth. A descrição formal da gramática nas duas notações encontra-se a seguir.

### Notação BNF

|  |
| --- |
| <programa> ::=  program  <definições> <declarações> <funções>  main { <declarações> <comandos> }  <definições> ::= <definição> <definições> | ε  <definição> ::= typedef struct <nome\_da\_estrutura> { <declarações> }  <nome\_da\_estrutura> ::= <identificador>  <declarações> ::= <declarações\_variáveis> | <declarações\_vetores> | <declarações\_structs> | <declarações> <declarações> | ε  <declarações\_variáveis> ::= declare <declaração\_variável> ; | declare <declaração\_variável>, <declarações\_variáveis>  <declaração\_variável> ::= <tipo> <identificador>  <declarações\_vetores> ::= declare <declaração\_vetor> ; | declare <declaração\_vetor>, <declarações\_vetores>  <declaração\_vetor> ::= <tipo> <identificador>[<número>]  <declarações\_structs> ::= declare <declaração\_struct> ; | declare <declaração\_struct>, <declarações\_structs>  <declaração\_struct> ::= struct <nome\_da\_estrutura> <identificador>  <funções> ::= function <tipo> <identificador> ( <parâmetros> ) { <declarações> <comandos> }  <parâmetros> ::= <declaração\_variável> | <declaração\_variável>, <parâmetros> | ε  <comandos> ::= <comando\_atribuição> | <comando\_condicional> | <comando\_iterativo> | <comando\_entrada> | <comando\_saída> | <comando\_chamada> | <comando\_retorno> | <comandos> <comandos> | ε  <comando\_atribuição> ::= <atribuição\_variável> | <atribuição\_agregados> | <atribuição\_vetor>  <atribuição\_variável> ::= <identificador> = <expressão> ;  <atribuição\_vetor> ::= <identificador>[<número>] = <expressão> ;  <atribuição\_agregados> ::= <identificador> = { <expressões> } ;  <comando\_condicional> ::= if ( <expressão> ) { <comandos> } |  if ( <expressão> ) { <comandos> } else { <comandos> }  <comando\_iterativo> ::= while ( <expressão> ) { <comandos> }  <comando\_entrada> ::= input <identificador> ;  <comando\_saída> ::= output <identificador> ;  <comando\_chamada> ::= <identificador> ( <argumentos> ) ;  <comando\_retorno> = return <expressão> ;  <expressões> ::= <expressão> | <expressão>, <expressões>  <expressão> ::= <expressão\_lógica> | <expressão\_aritmética> | <identificador> | <elemento\_do\_vetor> | <elemento\_struct> | <chamada\_função>  <expressão\_lógica> ::= <booleano> | <expressão> <operador\_lógico> <expressão> | not <expressão\_lógica>  <expressão\_aritmética> ::=<número> | <expressão> <operador\_aritmético> <expressão>  <elemento\_do\_vetor> ::= <identificador>[<numero>]   <elemento\_struct> ::= <identificador>.<identificador>  <chamada\_função> ::= <identificador> ( <argumentos> )  <argumentos> ::= <expressões>  <tipo> ::= int | boolean  <identificador> ::= <letra> | <letra><letra\_digito>  <letra\_dígito> ::= <letra> | <dígito>  <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 | \_  <dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  <booleano> ::= true | false  <operador\_lógico> ::= > | < | == | and | or  <operador\_aritmético> ::= + | - | \* | / |

### Notação Wirth

|  |
| --- |
| programa = "program" {definicao} {declaracao} {funcao} "main" "{" {declaracao} {comando} "}" .  definicao = "typedef" "struct" nome\_estrutura "{" {declaracao} "}" .  declaracao = declaracao\_variaveis | declaracao\_vetores | declaracao\_structs .  declaracao\_variaveis = "declare" declaracao\_variavel{"," declaracao\_variavel  } ";"  declaracao\_variavel = tipo identificador .  declaracao\_vetores = "declare" declaracao\_vetor {"," declaracao\_vetor } ";" .  declaracao\_vetor = tipo identificador "[" numero "]" .  declaracao\_structs = "declare" declaracao\_struct {"," declaracao\_struct } ";" .  declaracao\_struct = "struct" nome\_estrutura identificador ";"  funcao = "function" tipo identificador "(" [parametros] ")" "{" {declaracao}{comando} "}" .  parametros = declaracao\_variavel {"," declaracao\_variavel} .  comando = comando\_atribuicao | comando\_condicional | comando\_iterativo | comando\_entrada | comando\_saida | comando\_chamada | comando\_retorno .  comando\_atribuicao = atribuicao\_variavel | atribuicao\_vetor | atribuicao\_agregado .  atribuicao\_variavel = identificador "=" expressao ";" .  atribuicao\_vetor = identificador "[" numero "]" "=" expressao ";" .  atribuicao\_agregado = identificador "=" "{" expressao{"," expressao} "};" .  comando\_condicional = "if" "(" expresao ")" "{" {comando} "}" ["else" "{" {comando}"}"] .  comando\_iterativo = "while" "(" expresao ")" "{" {comando}"}" .  comando\_entrada = "input" identificador ";" .  comando\_saida = "output" identificador ";" .  comando\_chamada = identificador "(" [argumentos] ")" ";" .  comando\_retorno = "return" expressao ";" .  expressao = expressao\_logica | expressao\_aritmetica | identificador | elemento\_do\_vetor | elemento\_struct | chamada\_funcao .  expressao\_logica = booleano | "not" expressao\_logica | expressao operador\_logico expressao .  expressao\_aritmetica = numero | expressao operador\_aritmetico expressao .  elemento\_do\_vetor = identificador "[" numero "]" .  elemento\_struct = identificador "." identificador .  chamada\_funcao = identificador "(" [argumentos] ")" .  argumentos = expressao {"," expressao} .  booleano = "true" | "false" .  tipo = "int" | "boolean" .  nome\_estrutura = identificador .  identificador = letra{letra|digito} .  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} .   operador\_logico = ">" | "<" | "==" | "and" | "or" .  operador\_aritmetico = "+" | "-" | "\*" | "/" . |

## Descrição Reduzida

Para que a análise sintática fosse implementada usando um autômato de pilha estruturado, a descrição em notação de Wirth foi reduzida através do agrupamento de não-terminais. Além disso, as recursões à esquerda foram eliminadas. O resultado da redução é mostrado a seguir.

|  |
| --- |
| programa = "program" { "typedef" "struct" identificador "{" { "declare" ( ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } | "struct" identificador identificador {"," "struct" identificador identificador } ) ";" } "}" } { "declare" ( ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } | "struct" identificador identificador {"," "struct" identificador identificador } ) ";" } { "function" ( "int" | "boolean" ) identificador "(" [ ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } ] ")" "{" { "declare" ( ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } | "struct" identificador identificador {"," "struct" identificador identificador } ) ";" } {comando} "}" } "main" "{" { "declare" ( ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } | "struct" identificador identificador {"," "struct" identificador identificador } ) ";"} {comando} "}" .  comando = identificador ( "=" ( expressao | "{" expressao{"," expressao} "}" ) | "[" numero "]" "=" expressao | "(" [ expressao {"," expressao} ] ")" ) ";" | "if" "(" expressao ")" "{" {comando} "}" ["else" "{" {comando}"}"] | "while" "(" expressao ")" "{" {comando}"}" | "input" identificador ";" | "output" identificador ";" | "return" expressao ";" .  expressao = ( "true" | "false" | "not" expressao | numero | identificador [ "[" numero "]" | "." identificador | "(" [ expressao {"," expressao} ] ")" ] ) [ ( ">" | "<" | "==" | "and" | "or" | "+" | "-" | "\*" | "/" ) expressao ] .  identificador = letra{letra|digito} .  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} . |

# Análise Léxica

O módulo de análise léxica é responsável por converter o texto fonte em componentes chamados *tokens* ou átomos. Fornecendo, então, uma representação mais adequada para realização da análise e tradução pelos outros módulos do compilador.

Dessa maneira, o analisador léxico efetua a leitura do código fonte, e com o conhecimento das especificações da linguagem agrega os caracteres lidos do código em unidades que contenham um sentido completo para a linguagem, os *tokens*, descartando os caracteres que não possuem significado para a linguagem (por exemplo, caracteres de tabulação, na linguagem C).

Porém, mais informações são necessárias para as análises subsequentes, como o tipo de informação que está sendo representada pelo *token* (como, por exemplo, a classificação do átomo como um "identificador", "palavra reservada" ou "número"), além disso, um valor para o *token* também é necessário em outras etapas da compilação, esse valor pode ser um valor númerico do *token*, um índice em uma tabela de símbolos ou o próprio texto. Sendo também papel do analisador léxico realizar essas tarefas.

A tabela abaixo mostra os tipos de átomos reconhecidos pelo analisador léxico, definidos através de expressões regulares.

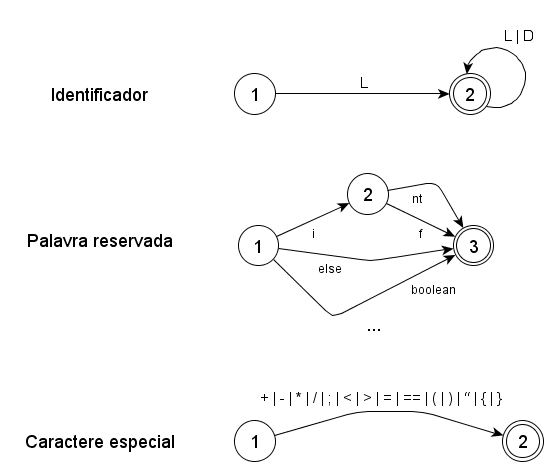
|  |  |
| --- | --- |
| **Classe do átomo** | **Expressão regular** |
| Identificador | [letra][letra | dígito]\* |
| Palavra reservada | if | else | int | boolean | string | false | true | while | main | void | return | def | and | or | not |
| Caractere especial | + | - | \* | / | ; | < | > | = | == | ( | ) | “ | { | } |
| Número | [dígito][dígito]\* |
| String | [“][qualquer\_caractere\_ascii][”] |
| Espaçador | space | \n | \t |
| Comentário | [#][qualquer\_caractere\_ascii]\*[\n] |

O analisador léxico implementado realiza uma simples detecção de erros, retornando *tokens* inválidos quando o conjunto de caracteres lidos não corresponde a nenhuma das expressões representadas acima.

## Autômatos

Para facilitar a representação do reconhecimento dos tokens pelo analisador léxico, as expressões regulares foram convertidas em autômatos finitos equivalentes, representados a seguir, onde:

L = A | B | … | Z | a | … | z  
D = 0 | 1 | .. | 9

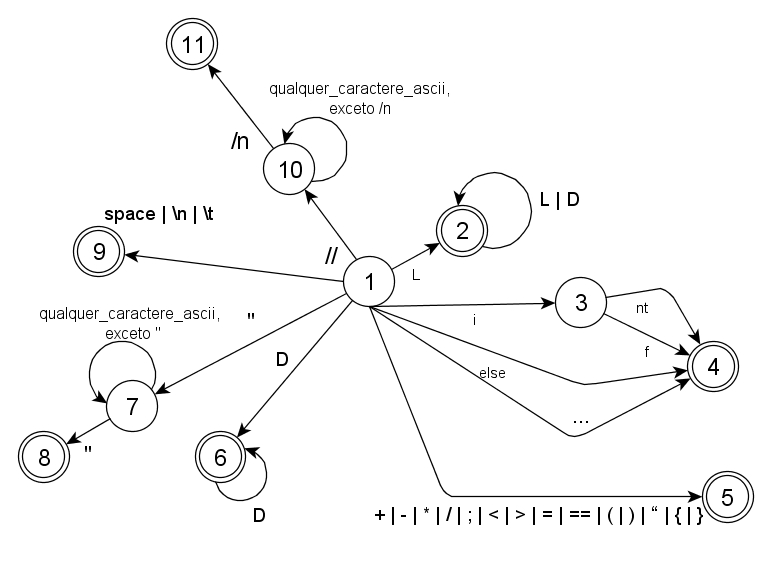






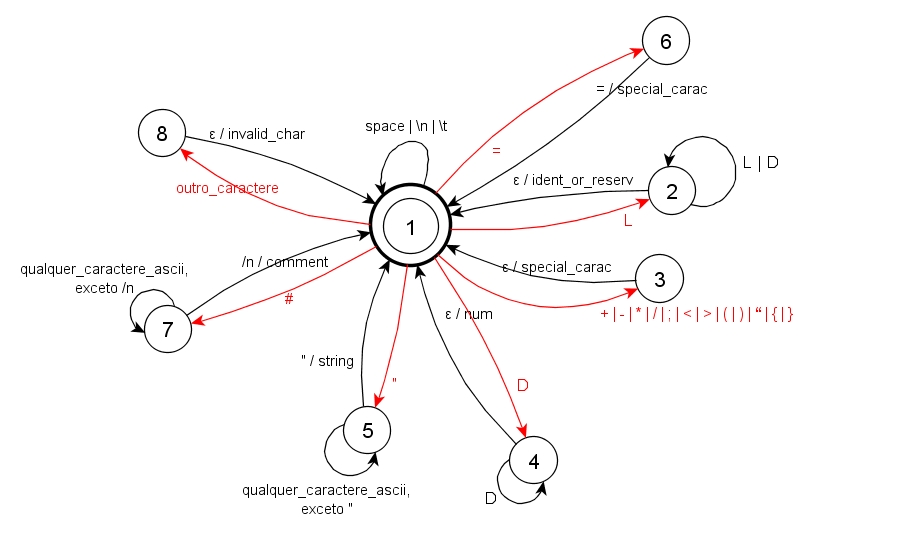


Estes autômatos reconhecem as correspondentes linguagens por eles definidas. Um autômato que reconhece todas essas linguagens, partindo de um mesmo estado inicial, mas apresentando um estado diferenciado para cada tipo de átomo é mostrado a seguir.



onde:  
L = A | B | … | Z | a | … | z  
D = 0 | 1 | .. | 9

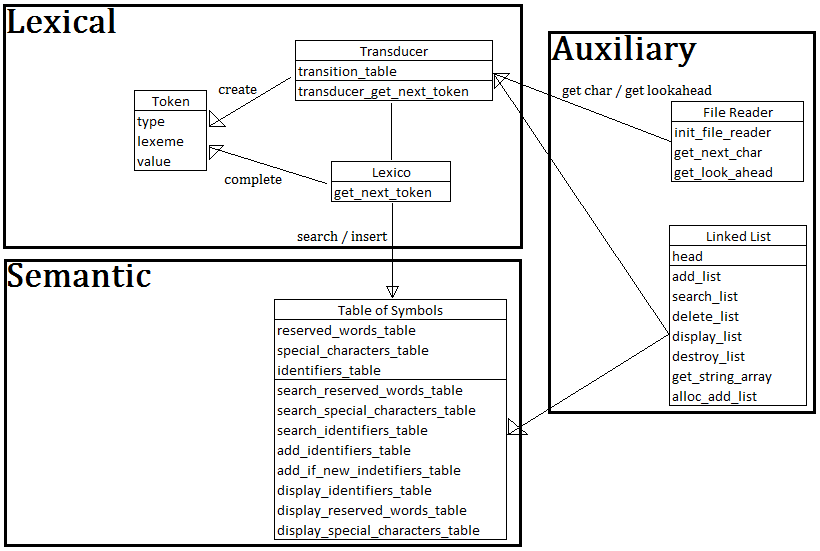
Por fim, o autômato foi convertido em transdutor, que emite como saída o átomo encontrado ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais um átomo do texto.



A partir deste transdutor é possível observar uma simplificação realizada: o átomo **ident\_or\_reserv** representa um átomo que começa com uma letra seguida por letras ou dígitos. Este átomo pode ser do tipo **identificador** ou **palavra\_reservada**, porém o transdutor indefere estes dois tipos, deixando esta tarefa para uma rotina posterior, ainda no analisador léxico. Maiores detalhes da implementação são expostos na próxima seção.

## Implementação

O analisador léxico foi implementado como uma sub-rotina que é chamada pelo analisador sintático sempre que um novo *token* for necessário. A arquitetura básica da implementação é a que segue:



Os arquivos token.h e token.c representam um *token* na linguagem, contendo seu tipo, lexema (ou seja, o texto que representa o *token*) e seu valor. Neles são definidos todos os tipos possíveis de tokens existentes na linguagem. Vale observar que para alguns tipos (*string, comment, ...*) o atributo *lexeme* é utilizado no lugar do *value*.

Os arquivos lexico.h e lexico.c representam o analisador léxico. Na implementação realizada, o analisador léxico faz requisições de tokens para o transdutor, que retorna um token com um tipo inicial (não necessariamente válido para a linguagem). É então, papel do analisador léxico completar o token retornado com o restante dos dados necessários, ou seja, calcular o valor dos tokens, bem como os tipos exatos de cada um, retornando o valor para quem fez a requisição. Além disso, o analisador também é responsável pelas inserções nas tabelas de identificadores, verificando a existência, ou não, de um identificador, e inserindo apropriadamente.

Os arquivos table\_of\_symbols.h e table\_of\_symbols.c representam as tabelas de símbolos existentes no compilador. Três tabelas de símbolos são utilizadas, a de palavras reservadas, a de caracteres especiais e a de identificadores. Por terem tamanho fixo, as tabelas de palavras reservadas e caracteres especiais foram implementadas através de matrizes, inicializadas com todos os lexemas possíveis. Já a tabela de identificadores, que apresenta tamanho variável, foi implementada através de uma lista ligada. Todas as tabelas apresentam funções de busca, e a tabela de identificadores apresenta também funções para inserção.

Os arquivos file\_reader.c e file\_reader.h implementam rotinas auxiliares de leitura. Dado um arquivo de entrada há funções para acesso ao caractere sendo lido e ao próximo caractere a ser lido, essas funções são get\_next\_char() e get\_look\_ahead(). O arquivo de entrada é determinado ao inicializar o file\_reader através da função void init\_file\_reader(char \*path). Esta inicialização é feita na função *main*.

Os arquivos transducer.c e transducer.h implementam o transdutor representado pela figura mostrada na questão 6. Para isto, existe uma matriz de transição (denominada transition\_table) onde as linhas representam os estados atuais e, dado um caractere de entrada (representado pelas colunas), o valor da célula correspondente indica o próximo estado. Os estados são representados por números inteiros e correspondem aos estados da figura do transdutor.

A função transducer\_get\_next\_token() percorre o arquivo de entrada (através de funções auxiliares) e a cada caractere lido atualiza seu estado. Quando o transdutor volta para o estado 1 (inicial), ele atualiza uma variável global token preenchendo seu *type* e *lexeme* (o preenchimento do campo *value* e eventual atualização do type é feito posteriormente por rotinas do lexico). Para possibilitar as transições vazias do transdutor é usado o caractere look\_ahead. Quando, usando o lookahead, é observado que o token incompleto se tornaria inválido, o token é atualizado e o transdutor volta ao estado inicial.

Um teste foi realizado comprovando o correto funcionamento do analisador léxico. A entrada e saída correspondente são mostradas a seguir.

|  |  |
| --- | --- |
| **Entrada\_teste.txt** | |
| int a = 1; int b=2; string s = "olá!?"; # comentário qualquer int a = 2,2;  123 # $ {typedef} | |
| **Saída** | **Tabelas de símbolos** |
| https://lh3.googleusercontent.com/_TEZweKcluFI4g2BBCfnrdtsGSDhFSdF-ra0Wk-bR0iZoNPkP7S5WmP7ICV4PtFPHF7is8QTXrQyL75dyVLdH-QSzMGD4292zl4cZCTfj1shd5r1yO8 tokens reconhecidos no formato: lexeme  TYPE (value) | https://lh5.googleusercontent.com/lUGKRGWhbUoexhajo7gVG4ttRhVqvqhG7lwt45oF9CgOA5txVZZJj_CYOVVMqW4oCjPsLhCctg_f3w7QtauK5uIISnvocKsSYpP_qhH1bQUxuoYE_yU |

# Análise Sintática

A análise sintática é responsável por verificar se o código a ser analisado corresponde à gramática da linguagem-fonte.

Em compiladores orientados por sintaxe, como é o caso do compilador desenvolvido, o analisador sintático é responsável por controlar as atividades do compilador, sendo responsável por efetuar chamadas de funções léxicas para o recebimento de tokens conforme os mesmos vão sendo consumidos.

A partir dos tokens gerados pelo analisador léxico, o analisador sintático analisa a sequência de recebimentos dos tokens e monta a árvore sintática.

## Submáquinas do Autômato de Pilha Estruturado (APE)

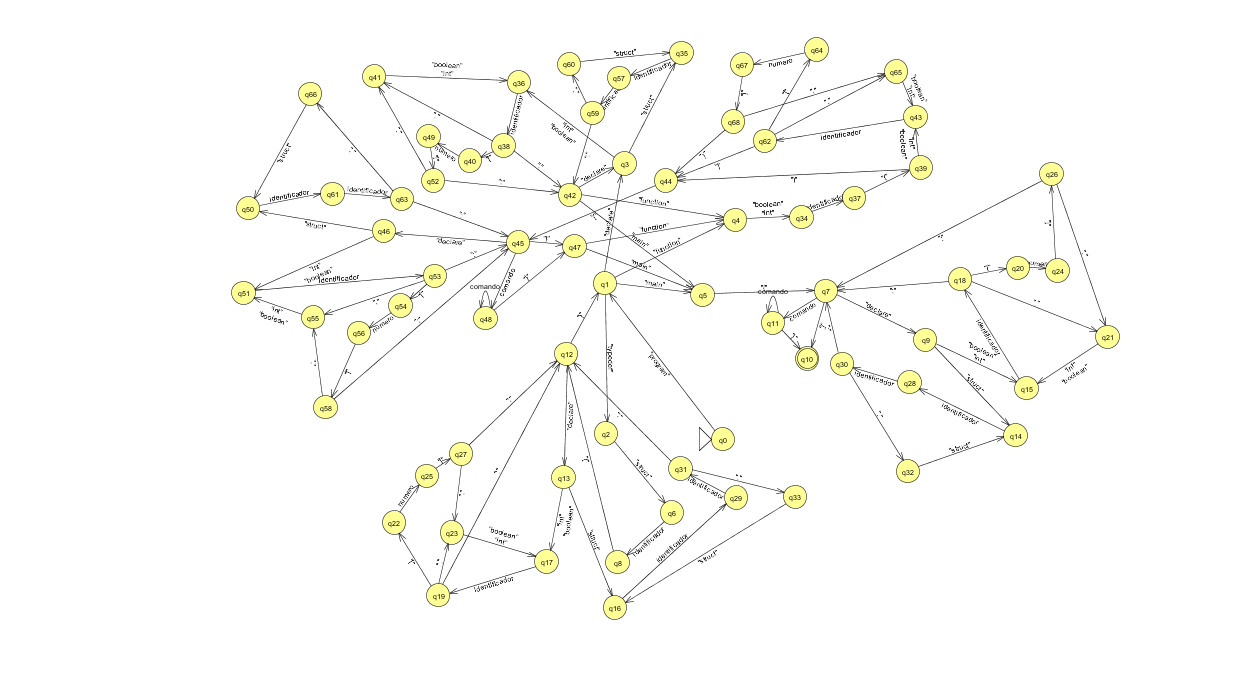
A implementação do analisador sintático do compilador desenvolvido foi baseada no autômato de pilha estruturado (APE) tornando-se necessária a tradução dos três terminais essenciais (PROGRAMA, COMANDO e EXPRESSAO) em autômatos.

Para esta tradução, foi utilizado o gerador de autômados criado por Hugo Baraúna e Fábio Yamate, disponível em: <http://radiant-fire-72.heroku.com/>. Ao inserir descrição reduzida da linguagem, obteve-se as seguintes listas de transições:

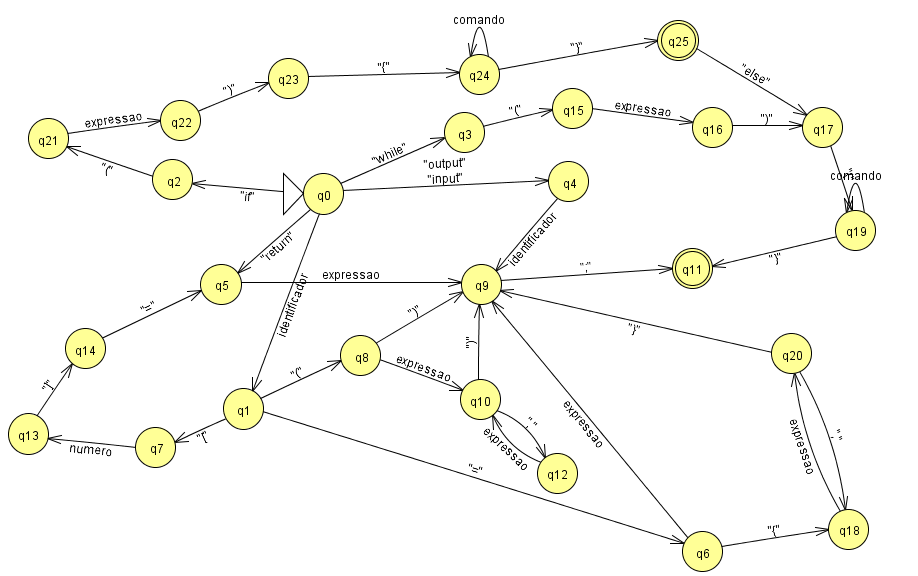
|  |  |  |
| --- | --- | --- |
| **PROGRAMA** | **COMANDO** | **EXPRESSAO** |
| initial: 0  final: 10  (0, "program") -> 1  (1, "typedef") -> 2  (1, "declare") -> 3  (1, "function") -> 4  (1, "main") -> 5  (2, "struct") -> 6  (3, "struct") -> 35  (3, "int") -> 36  (3, "boolean") -> 36  (4, "int") -> 34  (4, "boolean") -> 34  (5, "{") -> 7  (6, identificador) -> 8  (7, "declare") -> 9  (7, "}") -> 10  (7, comando) -> 11  (8, "{") -> 12  (9, "struct") -> 14  (9, "int") -> 15  (9, "boolean") -> 15  (11, "}") -> 10  (11, comando) -> 11  (12, "declare") -> 13  (12, "}") -> 1  (13, "struct") -> 16  (13, "int") -> 17  (13, "boolean") -> 17  (14, identificador) -> 28  (15, identificador) -> 18  (16, identificador) -> 29  (17, identificador) -> 19  (18, "[") -> 20  (18, ",") -> 21  (18, ";") -> 7  (19, "[") -> 22  (19, ",") -> 23  (19, ";") -> 12  (20, numero) -> 24  (21, "int") -> 15  (21, "boolean") -> 15  (22, numero) -> 25  (23, "int") -> 17  (23, "boolean") -> 17  (24, "]") -> 26  (25, "]") -> 27  (26, ",") -> 21  (26, ";") -> 7  (27, ",") -> 23  (27, ";") -> 12  (28, identificador) -> 30  (29, identificador) -> 31  (30, ",") -> 32  (30, ";") -> 7  (31, ",") -> 33  (31, ";") -> 12  (32, "struct") -> 14  (33, "struct") -> 16  (34, identificador) -> 37  (35, identificador) -> 57  (36, identificador) -> 38  (37, "(") -> 39  (38, "[") -> 40  (38, ",") -> 41  (38, ";") -> 42  (39, "int") -> 43  (39, "boolean") -> 43  (39, ")") -> 44  (40, numero) -> 49  (41, "int") -> 36  (41, "boolean") -> 36  (42, "declare") -> 3  (42, "function") -> 4  (42, "main") -> 5  (43, identificador) -> 62  (44, "{") -> 45  (45, "declare") -> 46  (45, "}") -> 47  (45, comando) -> 48  (46, "struct") -> 50  (46, "int") -> 51  (46, "boolean") -> 51  (47, "function") -> 4  (47, "main") -> 5  (48, "}") -> 47  (48, comando) -> 48  (49, "]") -> 52  (50, identificador) -> 61  (51, identificador) -> 53  (52, ",") -> 41  (52, ";") -> 42  (53, "[") -> 54  (53, ",") -> 55  (53, ";") -> 45  (54, numero) -> 56  (55, "int") -> 51  (55, "boolean") -> 51  (56, "]") -> 58  (57, identificador) -> 59  (58, ",") -> 55  (58, ";") -> 45  (59, ",") -> 60  (59, ";") -> 42  (60, "struct") -> 35  (61, identificador) -> 63  (62, "[") -> 64  (62, ",") -> 65  (62, ")") -> 44  (63, ",") -> 66  (63, ";") -> 45  (64, numero) -> 67  (65, "int") -> 43  (65, "boolean") -> 43  (66, "struct") -> 50  (67, "]") -> 68  (68, ",") -> 65  (68, ")") -> 44 | initial: 0  final: 11, 25  (0, identificador) -> 1  (0, "if") -> 2  (0, "while") -> 3  (0, "input") -> 4  (0, "output") -> 4  (0, "return") -> 5  (1, "=") -> 6  (1, "[") -> 7  (1, "(") -> 8  (2, "(") -> 21  (3, "(") -> 15  (4, identificador) -> 9  (5, expressao) -> 9  (6, expressao) -> 9  (6, "{") -> 18  (7, numero) -> 13  (8, expressao) -> 10  (8, ")") -> 9  (9, ";") -> 11  (10, ",") -> 12  (10, ")") -> 9  (12, expressao) -> 10  (13, "]") -> 14  (14, "=") -> 5  (15, expressao) -> 16  (16, ")") -> 17  (17, "{") -> 19  (18, expressao) -> 20  (19, "}") -> 11  (19, comando) -> 19  (20, ",") -> 18  (20, "}") -> 9  (21, expressao) -> 22  (22, ")") -> 23  (23, "{") -> 24  (24, "}") -> 25  (24, comando) -> 24  (25, "else") -> 17 | initial: 0  final: 1, 4  (0, "true") -> 1  (0, "false") -> 1  (0, "not") -> 2  (0, expressao) -> 3  (0, numero) -> 1  (0, identificador) -> 4  (2, expressao) -> 1  (3, ">") -> 2  (3, "<") -> 2  (3, "==") -> 2  (3, "and") -> 2  (3, "or") -> 2  (3, "+") -> 2  (3, "-") -> 2  (3, "\*") -> 2  (3, "/") -> 2  (4, "[") -> 5  (4, ".") -> 6  (4, "(") -> 7  (5, numero) -> 10  (6, identificador) -> 1  (7, expressao) -> 8  (7, ")") -> 1  (8, ",") -> 9  (8, ")") -> 1  (9, expressao) -> 8  (10, "]") -> 1 |

Para uma melhor visualização dos autômatos, foi utilizado o programa JFLAP, obtendo-se as representações a seguir, onde cada figura representa uma submáquina.

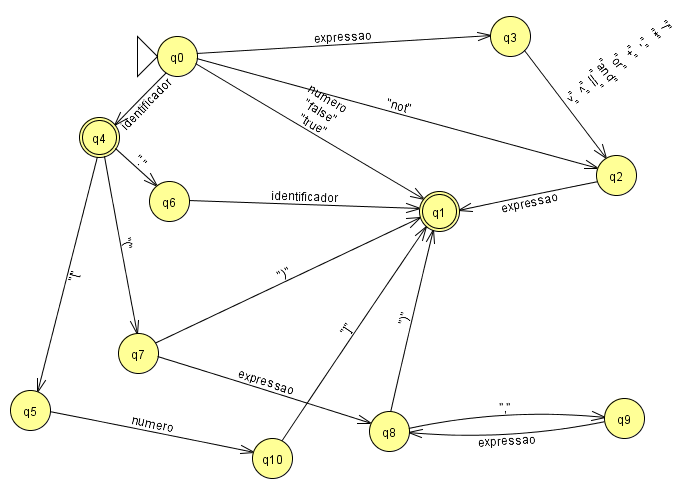
PROGRAMA



COMANDO



EXPRESSAO



## Implementação

O analisador sintático foi implementado como uma rotina do programa principal. A princípio, são inicializados todos os recursos utilizados pela análise sintática (o analisador léxico, tabela de transição e o autômato de pilha estruturado).

Então, a rotina pede o próximo token ao analisador léxico e, enquanto não terminar de ler o arquivo ou encontrar um erro na análise sintática, o analisador executa um passo do seu autômato de pilha estruturado.

O autômato de pilha estruturado (em spa.h e spa.c) tem sua principal função como sendo a função step, ilustrado abaixo:

|  |
| --- |
| **int** **spa\_step**() {  **int** machine\_token\_type = spa\_convert\_token\_to\_machine\_type();  **if** (machine\_token\_type == MTTYPE\_INVALID) **return** 0;  **return** transition\_current\_machine\_with\_token(machine\_token\_type);  } |

O autômato de pilha estruturado (APE) utiliza autômatos finitos que representam os três não terminais da linguagem (PROGRAMA, COMANDO, EXPRESSAO). Primeiramente ele converte o *token* retornado pelo analisador sintático, para que ele seja reconhecido nas transições dos autômatos finitos da linguagem.

Então, o passo do APE realiza uma transição da máquina atual para o *token* convertido. Essa transição possui três possibilidades. O primeiro caso consiste na existência de uma transição na máquina corrente do estado atual com o *token* recebido. Esse caso é tratado na função “transition\_to\_next\_state(int next\_state)”:

|  |
| --- |
| **void** **transition\_to\_next\_state** (**int** next\_state) {  semantico\_tbd();  current\_machine.current\_state = next\_state;  } |

Nesse caso, o mais simples deles, apenas ocorre uma mudança de estado da máquina atual. Além disso, a transição também desencadeia uma ação semântica, as ações semânticas serão descritas no capítulo referente à análise semântica.

O segundo caso, é quando a transição de estado é uma chamada de submáquina. Esse caso é tratado na seguinte função:

|  |
| --- |
| **void** **call\_machine**(**int** machine\_type) {  semantico\_tbd();  current\_machine.current\_state =  current\_machine.machine\_calls[current\_machine.current\_state][machine\_type];  spa\_stack\_push(current\_machine, spa\_stack);  current\_machine = machines\_array[machine\_type];  current\_machine.current\_state = 0;  } |

Para que seja possível o correto funcionamento do APE, é necessário que se guarde a máquina atual e o seu estado corrente em uma pilha. Para isso, foi implementada uma pilha em “*spa\_stack.h*” e “*spa\_stack.c*”, utilizada pelo APE para empilhar a máquina corrente antes de transicionar para a próxima. Além disso, a transição também desencadeia uma ação semântica.

O último caso existente é aquele em que a o autômato chegou em seu estado final, mas ainda existem máquinas empilhadas. Nesse caso, o que ocorre é um retorno à máquina anterior, pela função:

|  |
| --- |
| **void** **return\_machine**() {  semantico\_tbd();  current\_machine = spa\_stack\_pop(spa\_stack);  } |

Nesse caso, é desencadeada uma ação semântica, e, além disso, é retornado a máquina que se encontra no topo da pilha, para continuar a execução do APE de maneira adequada.

Todas as informações das máquinas utilizadas pelo APE se encontram nos arquivos “*machines.h*” e “*machines.c*”. Sendo estes modelos das máquinas, com suas tabelas de transição de estados, estados finais e transições que representam chamadas de máquinas.

# Definição do Ambiente de Execução

O ambiente em questão consiste da arquitetura do computador-alvo que é, neste caso, a MVN disponibilizada. A MVN (Máquina de von Neumann) simula o Modelo de von Neumann como um processador simples composto pelos seguintes elementos: Memória, Acumulador e Registradores Auxiliares.

Também faz parte do ambiente de execução o Montador disponibilizado. Portanto, a linguagem de saída do compilador desenvolvido não é a linguagem de máquina da MVN, mas sim, uma linguagem simbólica composta por mnemônicos e que lida com rótulos, operandos e sub-rotinas para endereçamento dentro de um programa.

Esta linguagem simbólica será descrita nos itens a seguir.

## Instruções da linguagem de saída

|  |  |  |  |
| --- | --- | --- | --- |
| **Operação** | **Mnemônico** | **Operando** | **Descrição** |
| Jump | JP | Endereço/Rótulo de desvio | Desvio incondicional |
| Jump if Zero | JZ | Endereço/Rótulo de desvio | Desvio se valor no acumulador é zero |
| Jump if Negative | JN | Endereço/Rótulo de desvio | Desvio se valor no acumulador é negativo |
| Load Value | LV | Constante de 12 bits | Deposita uma constante no acumulador |
| Add | + | Endereço/Rótulo do operando | Soma o conteúdo do acumulador com o operando |
| Subtract | - | Endereço/Rótulo do subtraendo | Subtração do conteúdo do acumulado com o subtraendo |
| Multiply | \* | Endereço/Rótulo do multiplicador | Multiplicação do conteúdo do acumulado com o multiplicador |
| Divide | / | Endereço/Rótulo do divisor | Divisão do conteúdo do acumulador com o divisor |
| Load | LD | Endereço/Rótulo do dado | Copia valor contido no endereço de memória para acumulador |
| Move to Memory | MM | Endereço/Rótulo de destino do dado | Copia valor do acumulador para a memória |
| Subroutine Call | SC | Endereço//Rótulo do subprograma | Desvio para subprograma |
| Return from Subroutine | RS | Endereço/Rótulo de retorno | Retorno de subprograma |
| Halt Machine | HM | Endereço/Rótulo do desvio | Parada |
| Get Data | GD | Dispositivo de E/S | Entrada |
| Put Data | PD | Dispositivo de E/S | Saída |
| Operating System | OS | Constante | Chamada de supervisor |

## Pseudoinstruções da linguagem de saída

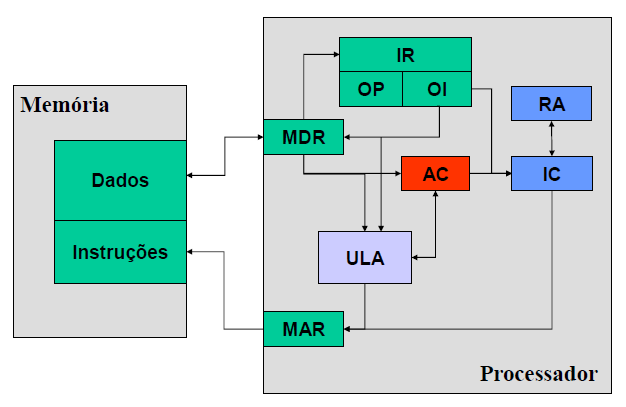
|  |  |
| --- | --- |
| **Pseudoinstrução** | **Descrição** |
| @ | Recebe um operando numérico, define o endereço da instrução seguinte, uma origem absoluta para o código a ser gerado |
| K | Define área preenchida por uma constante, o operando numérico tem o valor da constante de 2 bytes (em hexadecimal) |
| $ | Define um bloco de memória com número especificado de bytes, o operando numérico define o tamanho da área a ser reservada (em bytes) |
| # | Define o fim físico do texto fonte |
| & | Define uma origem relocável para o código a ser gerado, o operando é o endereço em que o próximo código se localizará (relativo à origem do código corrente) |
| > | Define endereço simbólico de entrada (Entry Point) |
| < | Define um endereço simbólico que referencia um entry-point externo |

## Características gerais

O ambiente de execução é composto pela simulação de um processador muito simples. Esse simulador apresenta um conjunto de elementos de armazenamento e dados, são eles: memória principal, acumulador e registradores auxiliares.

Primeiramente, na memória principal são armazenados as instruções dos programas e os dados utilizados por eles. Já o acumulador é um registrador especial que é utilizado para a realização das operações aritméticas e lógicas, por exemplo, a realização da instrução “+ VALOR” pode ser interpretada como “some o conteúdo indicado pelo rótulo ‘VALOR’ ao acumulador”.

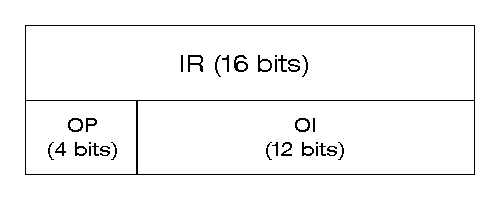
Por fim, os registradores auxiliares são utilizados em operações intermediárias. A tabela abaixo apresenta uma relação dos registradores auxiliares da MVN e suas funções. A arquitetura do simulador de MVN utilizado como ambiente de execução está representada a figura a seguir.



Por fim, os registradores auxiliares são utilizados em operações intermediárias. A tabela abaixo apresenta uma relação dos registradores auxiliares da MVN e suas funções.

|  |  |
| --- | --- |
| **Registrador Auxiliar** | **Utilização** |
| Registrador de Dados da Memória (MDR) | Utilizado para tráfego de dados entre a memória e outros elementos da MVN |
| Registrador de Endereço de Memória (MAR) | Contém a origem ou destino dos dados que se encontram no MDR |
| Registrador de Endereço de Instrução (IC) | Armazena a próxima instrução a ser executada pela máquina |
| Registrador de Instrução (IR) | Representa a instrução em execução, é composto de duas parcelas: o código de operação (OP) e o operando da instrução (OI). |

O registrador de Instrução (IR) da MVN armazena a instrução em execução corrente. A parcela OP é composta por 4 bits que codificam a instrução já a parcela OI complementa a instrução com o dado ou endereço parâmetro da instrução.



A memória possui capacidade de 4K de armazenamento e endereços de 12 bits. O acesso à variáveis é feito diretamente pelo endereço de memória, aleatoriamente. E a codificação numérica é feita em complemento de dois, com o bit mais significativo sendo indicador de sinal.

Duas instruções serão descritas em maiores detalhes para entender melhor o funcionamento do simulador, a de chamada de subrotina, retorno de subrotina.

### Chamada de subrotina

Para realização da chamada de (ou desvio para) subrotina é necessário armazenar o endereço da próxima instrução que seria executada antes do desvio, a fim de manter a ordem de execução do programa.

Dessa forma o conteúdo do registrador de instrução (IC) é armazenado no endereço de memória passado como operando da instrução de chamada de subrotina, salvando a posição de retorno da subrotina.

Então, é colocado no IC o endereço apontado pelo operando da instrução incrementado de um. Fazendo assim o desvio de execução do programa para o início da execução da subrotina.

### Retorno de subrotina

Devido ao método como foi implementada na MVN a chamada de subrotina, o retorno da subrotina se dá simplesmente trocando o conteúdo do registrador de instrução (IC) com o conteúdo da posição de memória apontada pelo operando da instrução.

## Exemplo de programa traduzido

A seguir é apresentado um exemplo da tradução feita manualmente de um programa escrito na linguagem definida para melhor entendimento do ambiente de execução.

**Código-fonte**

|  |
| --- |
| program function int fatorial\_iterativo(int n) {   declare int fat;    }  main {   declare int fat;      declare int n;      n = 5;       fat = 1;   while (n > 0) {         fat = fat \* n;         n = n - 1;   }   output fat; } |
|  |

**Código-objeto**

|  |
| --- |
| @ /0 ; área de código     JP main  zero        K =0 ; constante zero  main        LD k5            ;             MM n             ;              LD K1            ;             MM fat           ;  while        LD zero          ;             +  n             ;             -  K0            ;             JN endwhile      ;             JZ endwhile      ;             LD fat           ;             \*  n             ;              MM fat           ;             LD n             ;             -  K1            ;             MM fat           ;             JP while         ;  endwhile     LD fat           ;             MM output\_number ;             SC output        ;             HM /00           ;             #  main      @ /200 ; área de dados K5   K =5   ; declaracao de constante K1   K =1    ; declaracao de constante K0   K =0    ; declaracao de constante n    K =0    ; declaração de constante fat   K =0 ; declaração da variável fatorial |

# Análise Semântica

# Testes

# Conclusão