

PCS2056

Linguagens e Compiladores

Relatório Final

Professor: Ricardo Luis de Azevedo Rocha

Grupo:

| | |
|---------------------------------|---------|
| Filipe Morgado Simões de Campos | 5694101 |
|---------------------------------|---------|

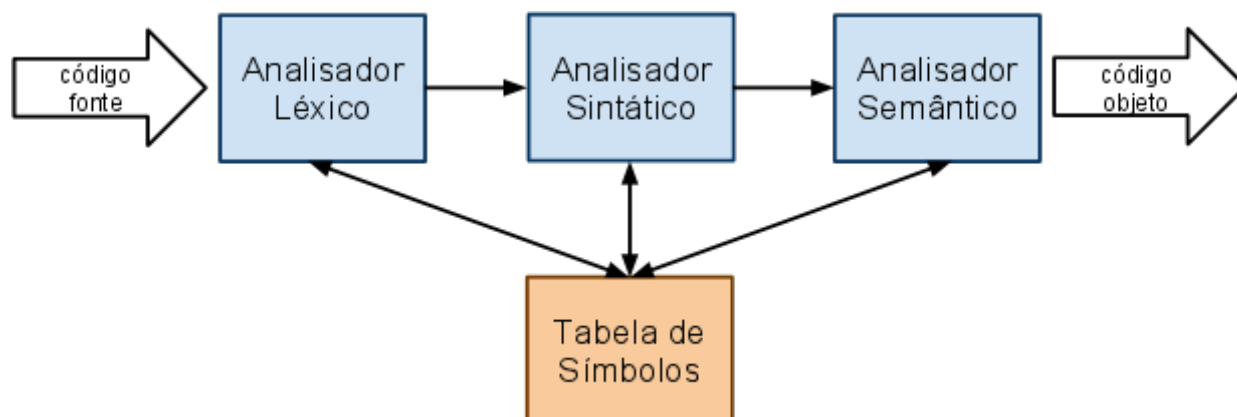
| | |
|----------------------|---------|
| Rafael Barbolo Lopes | 5691262 |
|----------------------|---------|

1. Introdução

Este relatório descreve o desenvolvimento do compilador *Educational Compiler*. Este compilador foi construído com objetivo didático e, por isso, requisitos de velocidade e otimização de código-fonte foram desconsiderados. Ele é um compilador escrito em C, que compila a linguagem *Edu*, e gera o código objeto que é representado pela linguagem simbólica da MVN, que roda na máquina virtual Java (JVM), usada na disciplina PCS2024 - Laboratório de Fundamentos de Engenharia de Computação.

O compilador foi desenvolvido inteiramente em ambiente Unix, no sistema operacional Mac OS X, sendo que os recursos computacionais necessários para a sua execução são simples e pouco custosos. O compilador é dirigido por sintaxe e executado em um único passo, com analisador sintático baseado em Autômato de Pilha Estruturado (APE). O código do compilador pode ser acessado pelo link que segue: https://github.com/barbolo/educational_compiler.

Sua arquitetura geral pode ser observada na imagem a seguir:



O desenvolvimento do compilador foi dividido nas seguintes fases:

1. Especificações gerais do compilador
2. Especificação da linguagem fonte
3. Especificação da linguagem de saída
4. Projeto e implementação do analisador léxico
5. Projeto e implementação do analisador sintático
6. Projeto e implementação do analisador semântico e gerador de código

A primeira fase foi a mais simples e definiu os conceitos gerais do compilador apresentados nesta introdução. Na segunda fase, especificação da linguagem fonte, foram definidas a linguagem de programação *Edu* e sua gramática, que foi descrita em notação BNF e notação de Wirth. Na terceira fase, foi compreendida e especificada a linguagem simbólica de saída.

Na quarta fase, que se refere ao analisador léxico, foram implementados os *tokens* e o transdutor que faz parsing do código fonte de entrada. Na quinta fase, que se refere ao analisador sintático, foi implementado um reconhecedor da linguagem *Edu* baseado no APE visto em aula.

Por fim, na sexta fase, foram implementadas as ações semânticas e suas integrações com os analisadores léxico e sintático, permitindo que o código objeto fosse gerado pelo compilador.

2. Especificação da linguagem fonte: Edu

2.1 Descrição informal

| Construção | Descrição informal |
|--|---|
| <i>programa</i> | <i>main { declarações comandos }</i> |
| <i>declarações</i> | <i>tipo_variável identificador ;</i> |
| <i>comando de atribuição</i> | <i>Identificador = expressão ;</i> |
| <i>comando condicional 1</i> | <i>If (condição) { comandos }</i> |
| <i>comando condicional 2</i> | <i>If (condição) { comandos } else { comandos }</i> |
| <i>comando iterativo</i> | <i>while (condição) { comandos }</i> |
| <i>comando de entrada</i> | <i>scan (tipo, identificador) ;</i> |
| <i>comando de saída</i> | <i>print (texto) ;</i> |
| <i>função</i> | <i>tipo_retorno identificador (argumentos) { declarações <i>begin</i> comandos retorno }</i> |
| <i>argumentos</i> | <i>tipo_variável identificador</i> |
| <i>estrutura de dados homogênea</i> | <i>tipo_variável identificador[inteiro] ; identificador[inteiro] = valor ;</i> |
| <i>estrutura de dados heterogênea</i> | <i>struct identificador { declarações }</i> |

2.2 Exemplo de programa

```
int soma(int a, int b) {
```

```

        return a + b;
    }

void imprime_x(int x) {
    print( "valor de x: " + x + ".");
}

main {
    int a, int b, int x;

    a = 10;
    b = 5;
    x = soma(a, b);
    imprime_x(x);
}

```

2.3 Gramática em notação BNF

```

<programa> ::= structs <estruturas> functions <funções> main { <declarações> begin
    <comandos>}

<estruturas> ::= <estrutura> | <estrutura> <estruturas>
<declarações> ::= <tipo> <variável>; | <tipo> <variável>, <declarações> |  $\epsilon$ 
<variável> ::= <id> | <vetor> | <matriz> | <acesso_estrutura>
<vetor> ::= <id>[<inteiro>]
<matriz> ::= <id>[<inteiro>][<inteiro>]
<acesso_estrutura> ::= <id>.<id>

<estrutura> ::= struct <id> { <declarações> }

<funções> ::= <função> | <função> <funções> |  $\epsilon$ 
<função> ::= <tipo_retorno> <id>(argumentos) {<declarações> begin <comandos>
    <retorno>}
<tipo_retorno> ::= <tipo> | void
<argumentos> ::= <tipo> <variável> | <tipo> <variável>, <argumentos> |  $\epsilon$ 
<retorno> ::= return <expressão>; |  $\epsilon$ 
<chamada_função> ::= <id>(argumentos)

<comandos> ::= <comando> | <comando> <comandos> |  $\epsilon$ 
<comando> ::= <atribuição> | <condicional> | <iteração> | <entrada> | <saída> |
    <chamada_função>;
<atribuição> ::= <variável> = <expressão>;
<condicional> ::= if (<condição>) {<comandos>} |
    if (<condição>) {<comandos>} else {<comandos>}
<iteração> ::= while (<condição>) {<comandos>}
<entrada> ::= scan( <tipo_primitivo> , <variavel> );
<saída> ::= print(<texto>);

<id> ::= <letra><letras_ou_dígitos>
<tipo_primitivo> ::= int | float | char | boolean
<tipo> ::= <tipo_primitivo> | my_struct <id>

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

```

```

        pow(<expressão>,<expressão>) | <termo>
<termo> ::= <termo> * <fator> | <termo> / <fator> | <fator>
<fator> ::= (<expressão>) | <sinal><valor>

<condição> ::= <expressão> <operador_booleano> <expressão> | <expressão_booleana>
<operador_booleano> ::= < | <= | > | >= | == | !=
<expressão_booleana> ::= <condição> <operador_lógico> <condição> | <booleano> |
        (<condição>) | NOT <expressão_booleana>
<operador_lógico> ::= AND | OR

<texto> ::= <string> | <string> + <texto> | <conteúdo> | <conteúdo> + <texto>
<conteúdo> ::= <expressão> | <condição>

<sinal> ::= - | E
<valor> ::= <variável> | <número> | <chamada_função> | <booleano>
<número> ::= <inteiro> | <decimal>
<inteiro> ::= <dígito> | <dígito><inteiro>
<decimal> ::= <inteiro>.<inteiro>
<booleano> ::= true | false
<string> = " <caracteres> "

<letras_ou_dígitos> ::= <letra_ou_dígito> | <letra_ou_dígito> <letras_ou_dígitos>
<letra_ou_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
<caracteres> ::= <letra> | <dígito> | + | - | / | * | =

```

2.4 Gramática em notação de Wirth

```

programa = "structs" estruturas "functions" funções "main" "{" declarações "begin"
comandos "}".

```

```

estruturas = {estrutura}.
declarações = [tipo variável {" "," tipo variável"};"].
variável = id | vetor | matriz | acesso_estrutura.
vetor = id["inteiro"].
matriz = id["inteiro"]""["inteiro"].
acesso_estrutura = id"."id;

```

```

estrutura = "struct" id "{" declarações "}".

```

```

funções = {função}.
função = tipo_retorno id "(" argumentos ")" "{" declarações "begin" comandos
retorno "}".
tipo_retorno = tipo | "void".
argumentos = [tipo variável {" "," tipo variável }].
retorno = ["return" expressão ";"].
chamada_função = id "(" argumentos ")".

```

```

comandos = {comando}.
comando = atribuição | condicional | iteração | entrada | saída | chamada_função.

```

```

atribuição = variável "=" expressão ";"
condicional = "if" "(" condição ")" "{" comandos "}" ["else" "{" comandos "}"].
iteração = "while" "(" condição ")" "{" comandos "}".
entrada = "scan" "(" tipo_primitivo "," variavel ")" ";"
saída = "print" "(" texto ")" ";"

id = letra{letra | dígito}.
tipo_primitivo = "int" | "float" | "char" | "boolean".
tipo = tipo_primitivo | "my_struct" id.

expressão = expressão ("+"|"-" ) termo | "pow" "(" expressão, expressão ")" | termo.
termo = termo ("*"|" /") fator.
fator = "(" expressão ")" | sinal valor.

condição = expressão operador_booleano expressão | expressão_booleana.
operador_booleano = "<" | "<=" | ">" | ">=" | "==" | "!=".
expressão_booleana = condição operador_lógico condição | booleano | "(" condição ")" |
    "NOT" expressão_booleana.
operador_lógico = "AND" | "OR".

texto = (string|expressao|condicao) {"+" (string|expressao|condicao)}.
sinal = ["-"].
valor = variável | número | chamada_função | booleano.
número = inteiro | decimal.
inteiro = dígito{dígito}.
decimal = inteiro"."inteiro.
booleano = "true" | "false".
string = "" caracteres ""

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".
caracteres = {letra | dígito | "+" | "-" | "*" | "/" | "="}.

```

2.5 Lista de palavras reservadas

| | |
|-----------|---------|
| structs | boolean |
| functions | pow |
| struct | not |
| my_struct | and |
| main | or |
| begin | true |
| void | false |
| return | print |
| if | int |
| else | float |
| while | char |
| scan | |

2.5 Lista de símbolos compostos por mais de um caracter ASCII

<= >= == !=

2.6 Gramática reduzida

Para implementar o APE com um pequeno número de máquinas, a gramática descrita em Wirth será reduzida, agrupando os não-terminais essenciais à linguagem. O primeiro passo para reduzir a gramática é descrevê-la em notação de Wirth, eliminando recursões à direita ou à esquerda, e representar todos os não-terminais através de iterações.

obs: para simplificar a construção do compilador, foram removidas as estruturas de dados homogêneas e heterogêneas; identificador, inteiro, booleano e string serão considerados não-terminais pois já são reconhecidos pelo léxico.

```
programa = {função} "main" "{" declarações {comando} "}".
declarações = [argumentos ";"].
função = tipo_retorno id "(" argumentos ")" "{" declarações {comando} retorno "}".
argumentos = [tipo id {" "," tipo id }].
retorno = ["return" expressão ";"].
chamada_função = "call" id "(" [expressao ] ")".
comando = atribuição | condicional | iteração | entrada | saída | chamada_função.
atribuição = id "=" (expressão|chamada_função) ";".
condicional = "if" "(" condição ")" "{" {comando} "}" ["else" "{" {comando} "}"].
iteração = "while" "(" condição ")" "{" {comando} "}".
entrada = "scan" "(" tipo "," id ")" ";".
saída = "print" "(" (string|expressao|condicao) {"+" (string|expressao|condicao)}
      ")" ";".
expressão = termo {"+"|"-"} termo}.
termo = fator {"*"|" /"} fator}.
fator = "(" expressão ")" | id | inteiro.
condição = (expressão comparador expressão) | booleano | expressão.
comparador = "<" | "<=" | ">" | ">=" | "==" | "!=".
tipo = "int" | "char" | "boolean".
tipo_retorno = tipo | "void".
```

Reduzindo essa gramática, obtemos a notação de Wirth final com as definições dos não-terminais essenciais:

```
programa = {( "int" | "char" | "boolean" | "void") id "(" [( "int" | "char" | "boolean")
id {" "," ( "int" | "char" | "boolean") id } ] ")" "{" [( "int" | "char" | "boolean") id
{" "," ( "int" | "char" | "boolean") id } ] ";"} {comando} ["return" expressao ";"] {" "}
  "main" "{" [( "int" | "char" | "boolean") id {" "," ( "int" | "char" | "boolean") id } ]
  ";"} {comando} {" }".
```

```

comando = (id "=" (expressao | (id "(" [expressao] ")") ";" )
  | ("if" "(" (condicao) ")" "{" {comando} "}" ["else" "{" {comando} "}"])
  | ("while" "(" (condicao) ")" "{" {comando} "}")
  | ("scan" "(" ("int" | "char" | "boolean") "," id ")" ";" )
  | ("print" "(" (string|condicao) {"+" (string|condicao)} ")" ";" )
  | ("call" id "(" expressao ")").

expressao = (("(" expressao ")" | id | inteiro) {"*" | "/" } ("(" expressao ")" |
id | inteiro)) {"+" | "-"} (("(" expressao ")" | id | inteiro) {"*" | "/" } ("("
expressao ")" | id | inteiro))}.

condicao = (expressao ("<" | "<=" | ">" | ">=" | "==" | "!=") expressao) | booleano |
expressao.

```

3. Especificação da linguagem de saída

O código objeto gerado pelo compilador *Educational Compiler* pertence à linguagem simbólica que foi utilizada na disciplina PCS2024 - Laboratório de Fundamentos de Engenharia de Computação.

As instruções, pseudoinstruções e regras dessa linguagem estão descritas nos itens seguintes.

3.1 Instruções

O conjunto de instruções disponibilizadas pela linguagem simbólica da Máquina de von Neumann usada é apresentado na tabela abaixo. São 16 instruções, incluindo desvios, operações aritméticas, transferência de dados e entrada e saída.

| Instrução | Interpretação |
|-------------|---|
| JP X | Desvia incondicionalmente para a posição X. |
| JZ X | Desvia para a posição X se o conteúdo do acumulador for zero. |
| JN X | Desvia para a posição X se o conteúdo do acumulador for negativo. |
| LV X | Carrega o valor de X no acumulador. |
| + X | Soma ao acumulador o conteúdo da posição X. |
| - X | Subtrai do acumulador o conteúdo da posição X. |
| * X | Multiplica com o acumulador o conteúdo da posição X. |
| / X | Divide o acumulador pelo conteúdo da posição X. |
| LD X | Carrega o conteúdo da posição X no acumulador. |

| | |
|-------------|---|
| MM X | Armazena o conteúdo do acumulador na posição X. |
| SC X | Chamada de subprograma. Desvia para a posição X. |
| RS X | Retorno de subprograma. Desvia para a posição X. |
| HM X | Termina o programa |
| GD X | Aciona dispositivo de entrada indicado por X e transfere dado do dispositivo para o acumulador. |
| PD X | Aciona dispositivo de saída indicado por X e transfere dado do acumulador para o dispositivo. |
| OS | Esta instrução não faz nada. |

3.2 Pseudoinstruções

As pseudoinstruções representam operações simples que são executadas pela linguagem simbólica da Máquina de von Neumann usada e são apresentadas na tabela abaixo.

| Pseudoinstrução | Interpretação |
|-----------------|---|
| @ X | Define que o código que está abaixo dessa instrução deve ser colocado a partir da posição X. Usado para determinar a posição inicial do código. |
| \$ X | Reserva área de dados de tamanho X. |
| # X | Final físico do código, sendo X o endereço de execução. |
| X K Y | Define constante X com o valor Y. |
| /X | Valor hexadecimal de X. |
| =X | Valor decimal de X. |

3.3 Regras

- Cada instrução é composta de um mnemônico e seu operando;
- Entre os elementos de uma linha de código, deve haver ao menos um espaço;
- Os operandos podem ser representados por seu valor numérico ou simbólico (rótulo);
- Se o primeiro caracter de uma linha for um espaço, esta linha não caracteriza um rótulo;
- À direita de um ponto-e-vírgula (;), todo texto é ignorado (comentário).

3.4 Memória

A memória da MVN tem tamanho de 4k. Para melhor organização do código objeto, foi estabelecido que a área de dados iniciará na posição 200.

4. Projeto e implementação do analisador léxico

O analisador léxico é um módulo responsável pela leitura do(s) arquivo(s) onde o código fonte está localizado; ele realiza o *parsing* desse arquivo, extraindo o que são chamados de *tokens* (átomos).

O analisador léxico lê cada um dos caracteres do código fonte e realiza agrupamento deles em *tokens*, que são classificados em tipos de informações importantes do programa que deve ser compilado/interpretado (um *token* pode ser, por exemplo, uma palavra reservada ou um identificador de variável). Durante esse processo, o analisador pode realizar conversões desses *tokens* (por exemplo, conversão para tipo numérico), e também elimina caracteres que não fazem parte do programa (espaços em branco, comentários, etc).

Também é papel do analisador léxico relatar erros durante sua execução, tratar macros, e criar e preencher estruturas de dado que guardam referência de localização de cada *token* no código fonte.

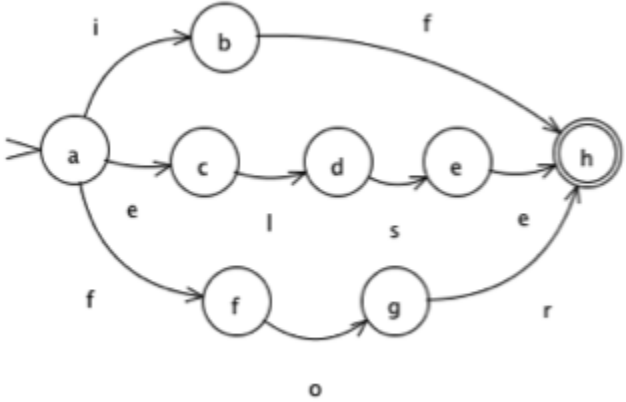
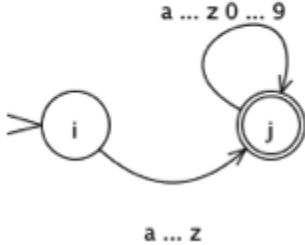
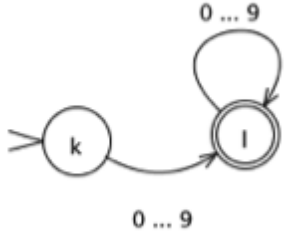
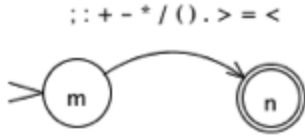
4.1 Átomos

Os átomos da linguagem estão definidos na tabela abaixo:

| Tipo de Átomo | Expressão Regular |
|---|--------------------------------------|
| palavra reservada | if else while ... |
| identificador | [a-z][a-z0-9]* |
| número inteiro | [0-9][0-9]* |
| senal de pontuação, operação ou separação | [;][:][+][-][*] \/ [(][)][\.] > = <] |
| senal composto | \: = |
| espaçador | (\n t)* |
| comentário | %[^\\n]*\\n |

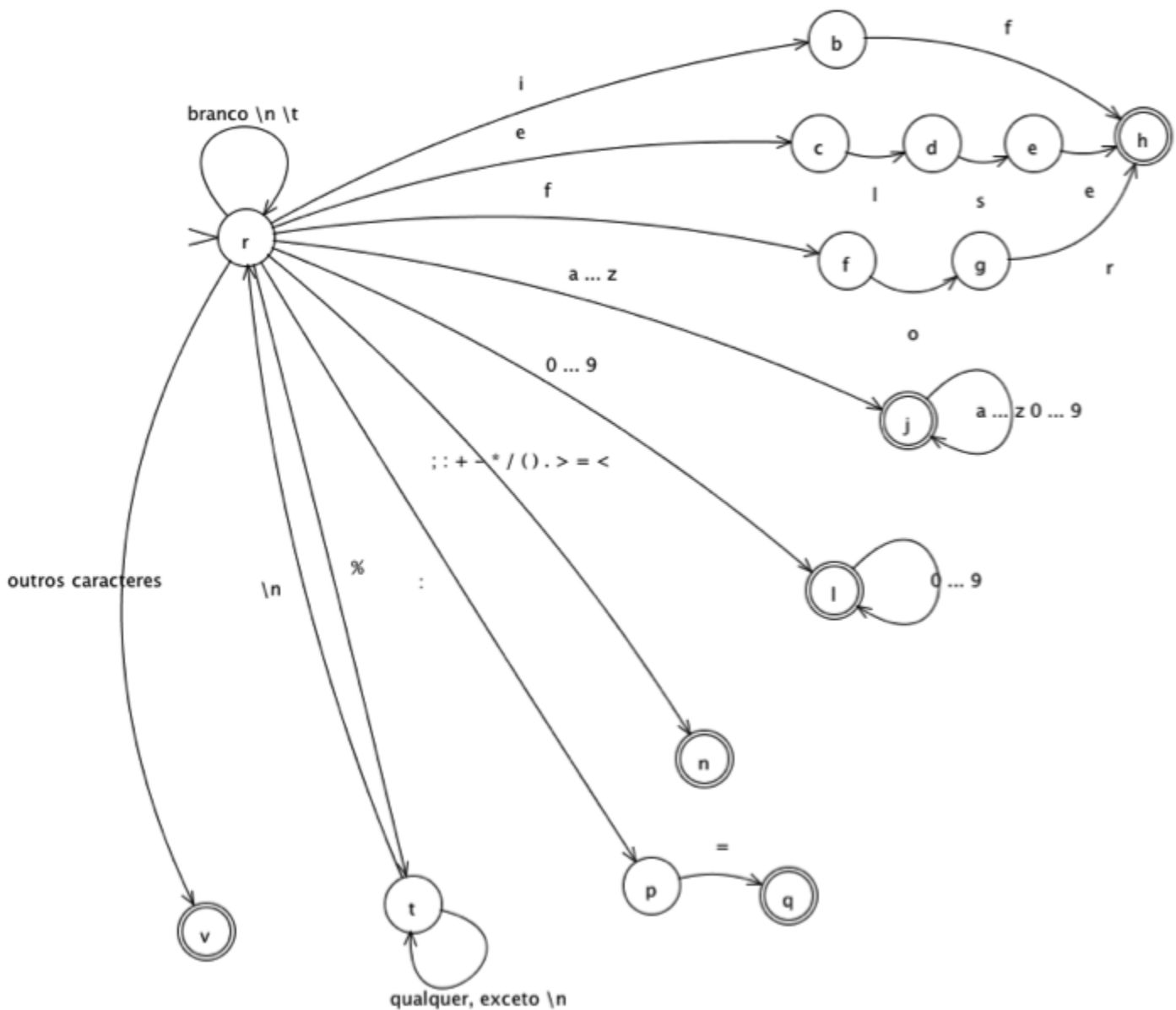
4.2 Autômatos Finitos

Os autômatos finitos que reconhecem os átomos da linguagem, estão apresentados abaixo:

| Tipo de Átomo | Autômato Finito |
|---|--|
| palavra reservada |  <p>The diagram shows a finite automaton with eight states: a, b, c, d, e, f, g, and h. State 'a' is the start state, indicated by an incoming arrow from the left. State 'h' is the final state, indicated by a double circle. The transitions are as follows: 'a' to 'b' on input 'i', 'a' to 'c' on input 'e', 'a' to 'f' on input 'f', 'b' to 'h' on input 'f', 'c' to 'd' on input 'l', 'd' to 'e' on input 's', 'e' to 'h' on input 'e', 'f' to 'g' on input 'o', and 'g' to 'h' on input 'r'.</p> |
| identificador |  <p>The diagram shows a finite automaton with two states: 'i' and 'j'. State 'i' is the start state, indicated by an incoming arrow from the left. State 'j' is the final state, indicated by a double circle. There is a self-loop on state 'j' labeled 'a ... z 0 ... 9', and a transition from state 'i' to state 'j' labeled 'a ... z'.</p> |
| número inteiro |  <p>The diagram shows a finite automaton with two states: 'k' and 'l'. State 'k' is the start state, indicated by an incoming arrow from the left. State 'l' is the final state, indicated by a double circle. There is a self-loop on state 'l' labeled '0 ... 9', and a transition from state 'k' to state 'l' labeled '0 ... 9'.</p> |
| sinal de pontuação, operação ou separação |  <p>The diagram shows a finite automaton with two states: 'm' and 'n'. State 'm' is the start state, indicated by an incoming arrow from the left. State 'n' is the final state, indicated by a double circle. There is a single transition from state 'm' to state 'n' labeled with the symbols ';; + - * / () . > = <'.</p> |

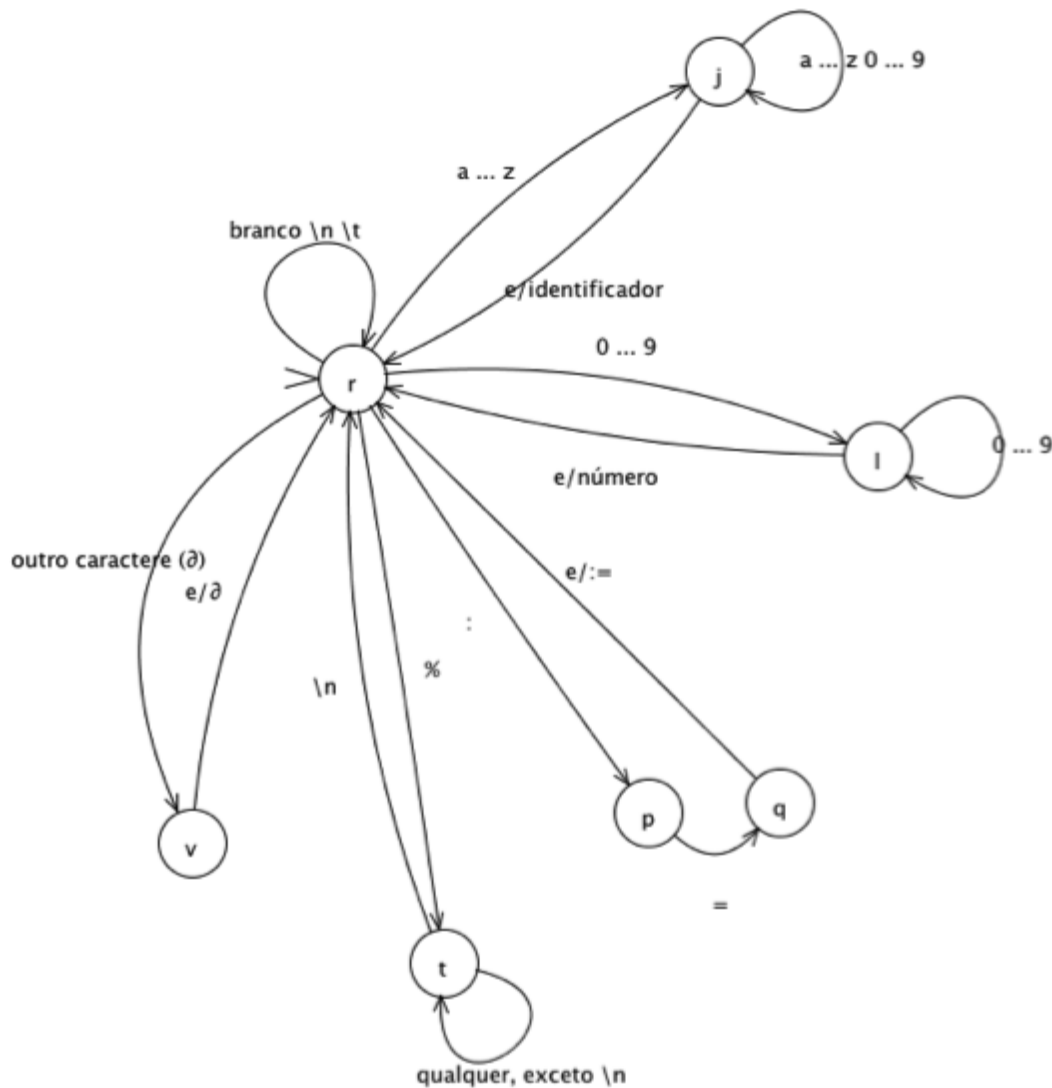
| | |
|-------------------|--|
| sinal composto | <pre> graph LR start(()) --> o((o)) o -- ":" --> p((p)) p -- "=" --> q(((q))) </pre> |
| espaçador | <pre> graph LR start(()) --> r(((r))) r -- "branco \n \t" --> r </pre> |
| comentário | <pre> graph LR start(()) --> s((s)) s -- "%" --> t(((t))) t -- "\n" --> s t -- "qualquer, exceto \n" --> t </pre> |
| outros caracteres | <pre> graph LR start(()) --> u((u)) u -- "outros caracteres" --> v(((v))) </pre> |

4.3 Autômato finito final



4.4 Transdutor

Para conseguir desenhar o autômato transdutor no software utilizado, foi necessário remover as transições que iam para o autômato que aceitava sinal de pontuação, operação ou separação e as que iam para o autômato que aceitava palavras reservadas. Apesar disso, pode-se entender o raciocínio para criação do transdutor através do autômato final desenhado. O resultado pode ser visto abaixo:



O transdutor foi abstraído como uma tabela de transições. Essa tabela possui 4 colunas: estado atual, entrada, próximo estado, saída. A implementação desta tabela foi feita através de uma matriz de uma estrutura de dados que guarda próximo estado e saída. Desta forma, é possível acessar próximo estado e saída conhecendo apenas o estado atual e a entrada.

É importante deixar claro que a tabela de transições se apresentou como um meio para implementar o transdutor com flexibilidade. Como a linguagem de programação do código fonte ainda não foi definida, é importante que o código seja flexível para aceitar novas regras de transições do transdutor.

O transdutor é uma instância que tem um método que consome uma entrada e atualiza seu estado atual, retornando o tipo de token que está sendo lido pelo analisador léxico. Para que este método funcione, é necessário conhecer o caractere de entrada atual e o seguinte (*lookahead*), de maneira que transições com cadeia vazia sejam resolvidas e não sejam

retornados resultados inválidos. Este método é chamado pelo analisador léxico enquanto está lendo um token e analisando seu tipo e valor.

4.5 Considerações sobre a implementação

4.5.1 Transdutor

O transdutor foi abstraído como uma tabela de transições. Essa tabela possui 4 colunas: estado atual, entrada, próximo estado, saída. A implementação desta tabela foi feita através de uma matriz de uma estrutura de dados que guarda próximo estado e saída. Desta forma, é possível acessar próximo estado e saída conhecendo apenas o estado atual e a entrada.

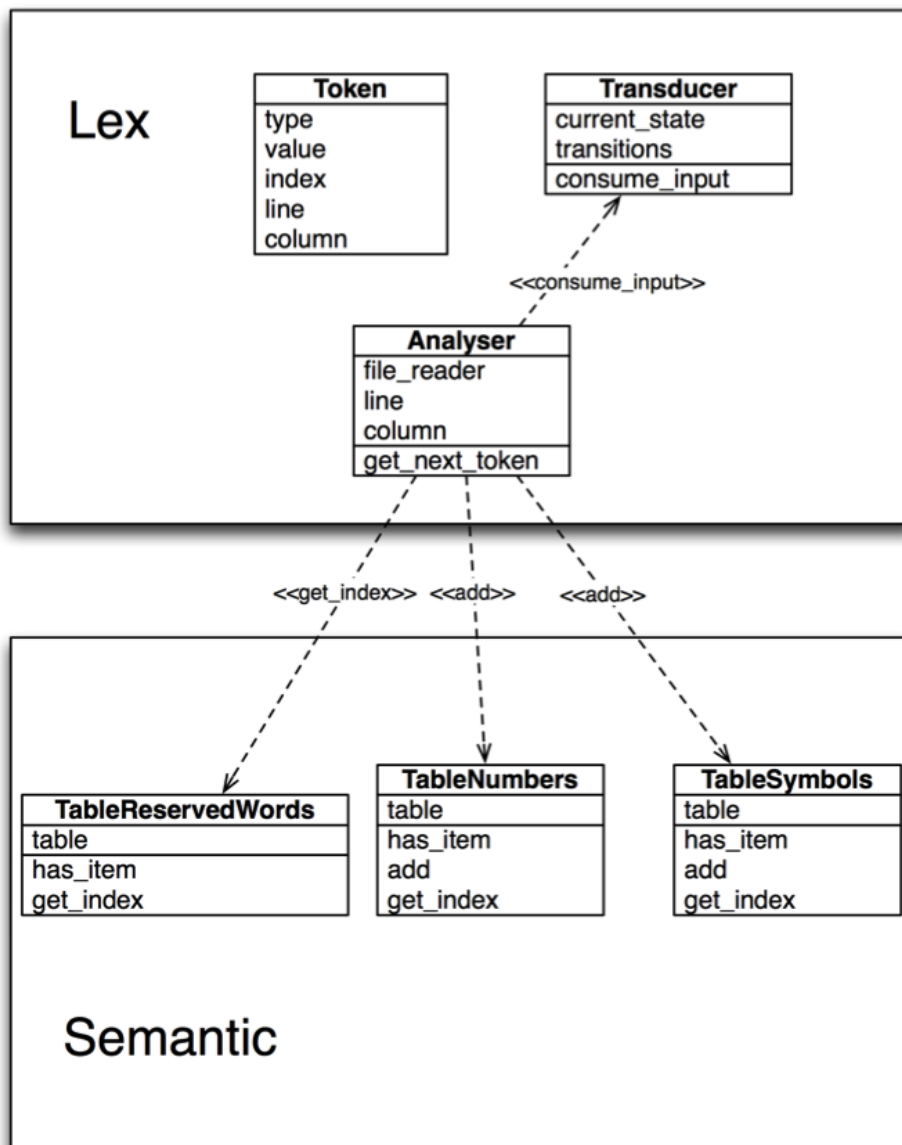
É importante deixar claro que a tabela de transições se apresentou como um meio para implementar o transdutor com flexibilidade. Como a linguagem de programação do código fonte ainda não foi definida, é importante que o código seja flexível para aceitar novas regras de transições do transdutor.

O transdutor é uma instância que tem um método que consome uma entrada e atualiza seu estado atual, retornando o tipo de token que está sendo lido pelo analisador léxico. Para que este método funcione, é necessário conhecer o caractere de entrada atual e o seguinte (*lookahead*), de maneira que transições com cadeia vazia sejam resolvidas e não sejam retornados resultados inválidos. Este método é chamado pelo analisador léxico enquanto está lendo um token e analisando seu tipo e valor.

4.5.2 Módulos em C

O analisador léxico, bem como todo o compilador que será desenvolvido, foi implementado utilizando-se a linguagem de programação C. Após uma etapa inicial de estudos teóricos, que foram apresentados nas primeiras questões e portanto não serão reintroduzidos aqui, prosseguimos com o seu desenvolvimento e testes para validar seu funcionamento. Logo, cabe apresentar a estrutura do compilador até então, descrevendo seu funcionamento e os testes realizados.

O programa foi estruturando em dois grupos para essa fase do compilador. O primeiro dele, representa o analisador léxico e suas estruturas auxiliares que são o foco dessa atividade. Já o segundo, reserva espaço para a etapa futura, o analisador semântico, que pôde ser contemplado com algumas estruturas que também são utilizadas pelo analisador léxico. A interação entre esses módulos ficará mais clara no decorrer da descrição do funcionamento do sistema.



O primeiro grupo é composto por uma dupla de arquivos que representa um token (token.h e token.c), uma outra dupla para implementar o transdutor (transducer.h e transducer.c) e, por último, a dupla de arquivos que coordena o processo do analisador léxico (analyser.h e analyser.c).

No arquivo token.h foi definida a estrutura de dados de um token e todos os tipos possíveis a que este poderia pertencer. Além dessa estrutura e dos tipos de tokens que são utilizados por todo o analisador léxico, foi declarada a variável global que representa o token no compilador.

Para o arquivo token.c restou a implementação da função *token_type_name* que é responsável por retornar o nome do tipo do token que está sendo processado pelo analisador léxico. Ela foi

utilizada na impressão desse nome na saída do programa.

O arquivo *transducer.h* e o arquivo *transducer.c* já tiveram seu funcionamento explicado no item 4.5.1. Vale apenas acrescentar que seu funcionamento é solicitado pelo *analyser* através da função *transducer_consume_input* em que são passados os valores do caractere atual e do próximo (*lookahead*) e a função retorna o código do tipo do token com base na consulta da matriz *transducer_transitions* que representa o transdutor final.

O arquivo *analyser.h* contém apenas a declaração da função *get_next_token* que foi implementada no arquivo *analyser.c*. Essa função é responsável pelo funcionamento da parte do compilador responsável pela análise léxica como já foi explicado na questão 8. Durante esse procedimento, é feito o uso da função *read_next_char*, que retorna os caracteres do arquivo que contém o código fonte, e das estruturas anteriormente indicadas do grupo semântico para a consulta e adição de valores nas tabelas semânticas.

O segundo grupo, contém as tabelas semânticas que estão implementadas nos arquivos *tables.h*, *tables.c*, *symboltable.h* e *symboltable.c*. Elas são: tabela de símbolos, tabela de números, tabela de palavra reservadas e a tabela de caracteres especiais.

Uma biblioteca foi criada para representar uma string, *strings.h*, Apêndice XI, e *strings.c*, Apêndice XII. A utilizamos apenas para comparar se uma string é igual a outra através da função *strcmp*, e também para retornar uma cópia de uma string com a função *strcpy*.

Por fim, a última biblioteca realiza a leitura dos caracteres dos arquivos, *reader.h* e *reader.c*. Vale comentar que essa possui uma estrutura representando uma cabeça de leitura que possui os caracteres atual, anterior e futuro (*lookahead*), a posição (linha e coluna) do caractere atual e o ponteiro para o arquivo sendo lido.

Para consolidar o desenvolvimento dessa etapa do compilador, testamos a saída do léxico para o seguinte código de entrada:

```
b
b
b == a = 10120 + 15;
> >=
< <=

3

&

bc = 10
```

d == ; 29123 7

10

10

15

bc

bc

=

for

else

A saída encontrada, que também era a esperada, foi:

Padrão da saída para cada token:

valor (indicie na tabela semântica) :: nome do tipo (identificador do tipo) :: linha ,
coluna

| | | | | | | | | | |
|--------|-------|------|----|----------------|-----|----|------|------------|----|
| Token: | b | (0) | :: | identifier | (2) | :: | line | 1, column | 1 |
| Token: | b | (0) | :: | identifier | (2) | :: | line | 2, column | 1 |
| Token: | b | (0) | :: | identifier | (2) | :: | line | 3, column | 1 |
| Token: | == | (0) | :: | special char | (4) | :: | line | 3, column | 3 |
| Token: | a | (1) | :: | identifier | (2) | :: | line | 3, column | 6 |
| Token: | = | (1) | :: | special char | (4) | :: | line | 3, column | 8 |
| Token: | 10120 | (0) | :: | integer number | (3) | :: | line | 3, column | 10 |
| Token: | + | (2) | :: | special char | (4) | :: | line | 3, column | 16 |
| Token: | 15 | (1) | :: | integer number | (3) | :: | line | 3, column | 18 |
| Token: | ; | (3) | :: | special char | (4) | :: | line | 3, column | 20 |
| Token: | > | (4) | :: | special char | (4) | :: | line | 4, column | 1 |
| Token: | >= | (5) | :: | special char | (4) | :: | line | 4, column | 3 |
| Token: | < | (6) | :: | special char | (4) | :: | line | 5, column | 1 |
| Token: | <= | (7) | :: | special char | (4) | :: | line | 5, column | 3 |
| Token: | 3 | (2) | :: | integer number | (3) | :: | line | 7, column | 1 |
| Token: | & | (-1) | :: | invalid (-1) | | :: | line | 9, column | 1 |
| Token: | bc | (2) | :: | identifier | (2) | :: | line | 11, column | 1 |
| Token: | = | (1) | :: | special char | (4) | :: | line | 11, column | 4 |

| | | | | | | | | |
|--------|-------|-----|----|--------------------|----|------|------------|----|
| Token: | 10 | (3) | :: | integer number (3) | :: | line | 11, column | 6 |
| Token: | d | (3) | :: | identifier (2) | :: | line | 13, column | 1 |
| Token: | == | (0) | :: | special char (4) | :: | line | 13, column | 3 |
| Token: | ; | (3) | :: | special char (4) | :: | line | 13, column | 6 |
| Token: | 29123 | (4) | :: | integer number (3) | :: | line | 13, column | 8 |
| Token: | 7 | (5) | :: | integer number (3) | :: | line | 13, column | 14 |
| Token: | 10 | (3) | :: | integer number (3) | :: | line | 15, column | 1 |
| Token: | 10 | (3) | :: | integer number (3) | :: | line | 17, column | 1 |
| Token: | 15 | (1) | :: | integer number (3) | :: | line | 19, column | 1 |
| Token: | bc | (2) | :: | identifier (2) | :: | line | 21, column | 1 |
| Token: | bc | (2) | :: | identifier (2) | :: | line | 23, column | 1 |
| Token: | = | (1) | :: | special char (4) | :: | line | 25, column | 1 |
| Token: | for | (1) | :: | reserved word (1) | :: | line | 27, column | 1 |
| Token: | else | (2) | :: | reserved word (1) | :: | line | 29, column | 1 |

5. Projeto e implementação do analisador sintático

Como segundo e principal bloco do compilador dirigido por sintaxe desse projeto, temos o analisador sintático. Podemos dizer que ele é o principal bloco, pois é responsável por todo o gerenciamento e chamadas das rotinas existentes no compilador, tanto do léxico quanto do semântico. Sua função principal está relacionada à análise e verificação da sequência dos átomos provindo do analisador léxico quando este lê o código fonte. Assim é de sua responsabilidade verificar se esses átomos estão de acordo com a sintaxe da linguagem a ser compilada.

Em teoria, o produto dessa etapa de compilação é a árvore sintática. Porém, para a implementação, o que de fato ocorre é que não existe a necessidade de se sintetizar toda essa árvore para então passarmos para a etapa do analisador semântico, tudo isso pode ser feito de uma vez só. Ou seja, geram-se “pedaços” da árvore necessários para que as ações semânticas corretas possam ser chamadas e a geração de código efetuada com sucesso.

A análise sintática engloba, em geral, as seguintes funções :

- Identificação de sentenças;
- Detecção de erros de sintaxe;
- Recuperação de erros;
- Correção de erros;
- Montagem da árvore abstrata da sentença;
- Comando da ativação do analisador léxico;
- Comando do modo de operação do analisador léxico;
- Ativação de rotinas da análise referente às dependências de contexto da

- linguagem;
- Ativação de rotinas de análise semântica;
- Ativação de rotinas de síntese do código objeto.

O analisador sintático foi construído tomando como base a já apresentada e discutida gramática reduzida da linguagem Edu.

O passo seguinte constitui em utilizar o gerador de autômatos criado por Hugo Baraúna e Fábio Yamate, localizado em: <http://radiant-fire-72.herokuapp.com/>. Para tal, deve-se colocar a gramática reduzida em notação de Wirth na entrada do gerador, e esse deve produzir como saída os autômatos já otimizados que representam cada submáquina do APE que reconhece a linguagem descrita pela gramática dada. Tais autômatos nos são transmitidos na forma de XML com as transições correspondentes às máquinas que o APE deve utilizar.

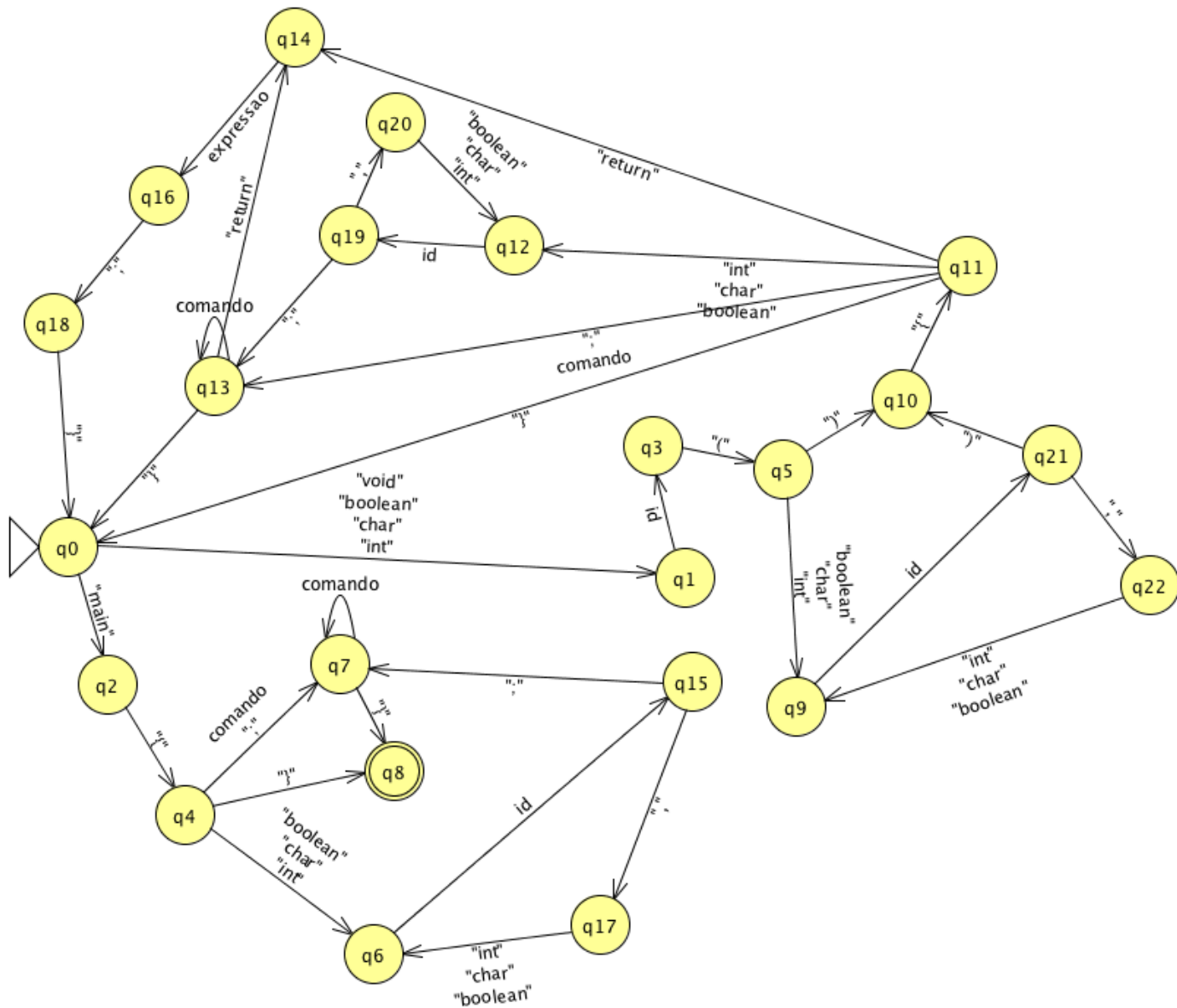
Abaixo, seguem as transições e os autômatos gerados para cada submáquina.

Máquina **Programa**:

Transições:

| | |
|---------------------|-----------------------|
| initial: 0 | (11, "boolean") -> 12 |
| final: 8 | (11, ";") -> 13 |
| (0, "int") -> 1 | (11, comando) -> 13 |
| (0, "char") -> 1 | (11, "return") -> 14 |
| (0, "boolean") -> 1 | (11, "}") -> 0 |
| (0, "void") -> 1 | (12, id) -> 19 |
| (0, "main") -> 2 | (13, comando) -> 13 |
| (1, id) -> 3 | (13, "return") -> 14 |
| (2, "{") -> 4 | (13, "}") -> 0 |
| (3, "(") -> 5 | (14, expressao) -> 16 |
| (4, "int") -> 6 | (15, ",") -> 17 |
| (4, "char") -> 6 | (15, ";") -> 7 |
| (4, "boolean") -> 6 | (16, ";") -> 18 |
| (4, ";") -> 7 | (17, "int") -> 6 |
| (4, comando) -> 7 | (17, "char") -> 6 |
| (4, "}") -> 8 | (17, "boolean") -> 6 |
| (5, "int") -> 9 | (18, "}") -> 0 |
| (5, "char") -> 9 | (19, ",") -> 20 |
| (5, "boolean") -> 9 | (19, ";") -> 13 |
| (5, ")") -> 10 | (20, "int") -> 12 |
| (6, id) -> 15 | (20, "char") -> 12 |
| (7, comando) -> 7 | (20, "boolean") -> 12 |
| (7, "}") -> 8 | (21, ",") -> 22 |
| (9, id) -> 21 | (21, ")") -> 10 |
| (10, "{") -> 11 | (22, "int") -> 9 |
| (11, "int") -> 12 | (22, "char") -> 9 |
| (11, "char") -> 12 | (22, "boolean") -> 9 |

Autômato:



Máquina **Comando**

Transições:

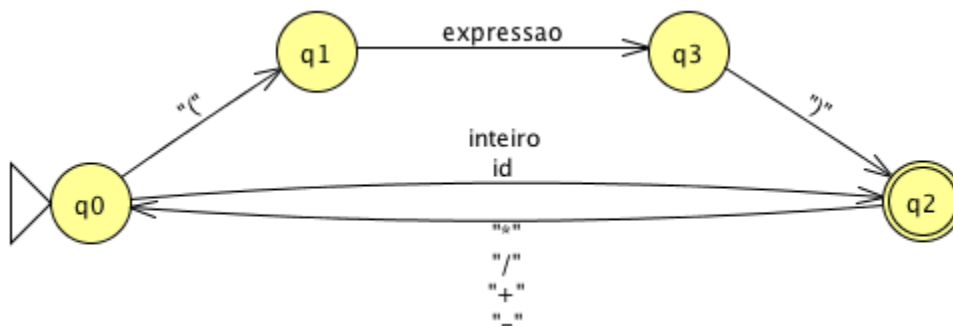
| | |
|---|--|
| initial: 0 final: 12, 22 (0, id) -> 1 (0, "if") -> 2 (0, "while") -> 3 (0, "scan") -> 4 (0, "print") -> 5 | (15, ")") -> 10 (16, condicao) -> 17 (16, string) -> 17 (17, ")") -> 10 (17, "+") -> 16 (18, condicao) -> 19 (19, ")") -> 20 |
|---|--|

Máquina **Expressão**

Transições:

| | |
|---|--|
| initial: 0 final: 2 (0, "(") -> 1 (0, id) -> 2 (0, inteiro) -> 2 (1, expressao) -> 3 | (2, "*") -> 0 (2, "/") -> 0 (2, "+") -> 0 (2, "-") -> 0 (3, ")") -> 2 |
|---|--|

Autômato:

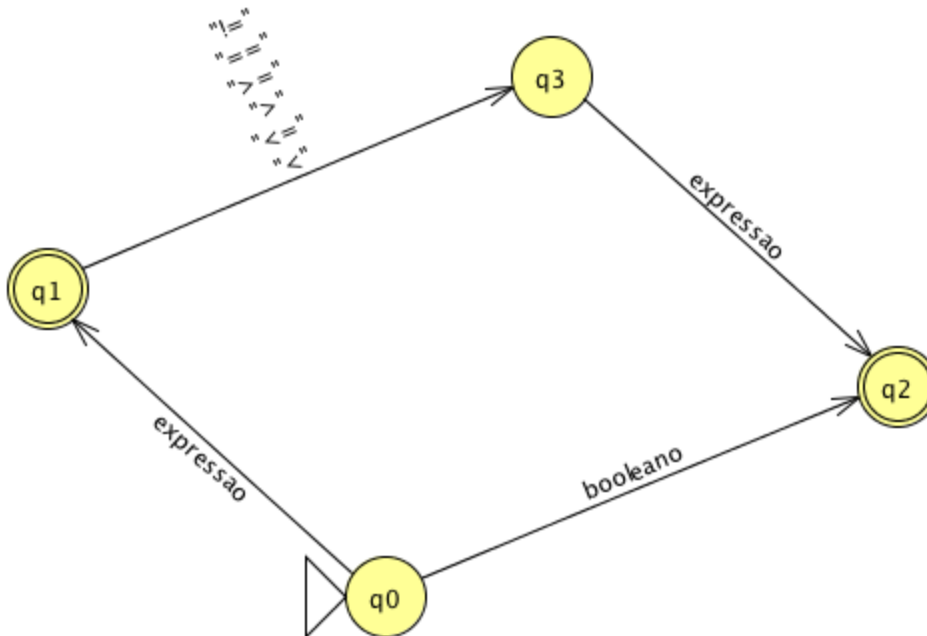


Máquina **Condição**

Transições:

| | |
|---|---|
| initial: 0 final: 1, 2 (0, expressao) -> 1 (0, booleano) -> 2 (1, "<") -> 3 (1, "<=") -> 3 | (1, ">") -> 3 (1, ">=") -> 3 (1, "==") -> 3 (1, "!=") -> 3 (3, expressao) -> 2 |
|---|---|

Autômato:



Para a implementação do analisador sintático, foram desenvolvidos três conjuntos (arquivo .c e arquivo .h) de arquivos, sendo eles:

- syntatic: nesses arquivos residem a estrutura lógica central do compilador. É aqui que foi implementada a rotina responsável pelo gerenciamento e pela chamada das outras rotinas dos grandes blocos do compilador. Seu funcionamento será explicado abaixo;
- machine: nesses arquivos encontra-se a estrutura de dados que representa cada submáquina do APE de nosso compilador e também uma rotina para a inicialização das transições dessas máquinas;
- ape: aqui, foi desenvolvida a estrutura de dados que representa o autômato de pilha estruturado bem como as funções necessárias para seu funcionamento;
- ape_stack: esses arquivos existem para implementar a pilha utilizada no APE. Logo, existe a estrutura de dados responsável por tal representação e as rotinas de *push* e *pop*.

A rotina inicial do nosso compilador é responsável por uma série de chamadas preparatórias até que se chegue na rotina do analisador sintático. Elas são, *init_reader* que inicia o leitor do arquivo com o código fonte; *initialize_writer* que inicializa a funcionalidade de escrita que será utilizada para escrever o código de máquina após a compilação; *initialize_transitions* que inicia as transições do transdutor utilizado no analisar léxico; *initialize_semantic* que inicia o analisador semântico; *init_semantic_actions* que é responsável por inicializar as ações semânticas; *init_ape_machines* que inicia os APE utilizados pelo sintático e por fim a função *run_sintatic* é chamada pra dar início à rotina mestre do sintático.

Após ser chamada, a rotina *run_sintatic* começa requisitando o primeiro átomo do analisador léxico. Após esse passo, entra-se em um loop que executa as seguintes ações:

- *ape_consume_token*, essa rotina é responsável por fazer funcionar o APE responsável pela análise sintática do código fonte, bem como as chamadas das ações semânticas necessárias em cada transição dos autômatos das submáquinas do APE. Essa função também alertará o compilador da ocorrência de erro sintático, caso ela ocorra, a execução do compilador é interrompida;
- checa-se a ocorrência de erro semântico, caso ela seja verdadeira a execução do compilador é interrompida;
- requisita-se o próximo átomo do analisador léxico.

Por fim, é realizada uma etapa de checagem para verificar se o APE está em seu estado final e se a pilha do APE está vazia. Se sim, chama-se a função do módulo semântico *write_variables* que realiza a escrita do código referente à área de dados como as variáveis declaradas durante o código, as constantes utilizadas e as variáveis temporárias. Essas informações são recuperadas consultando as tabelas criadas durante a compilação do código fonte.

6. Projeto e implementação do analisador semântico e gerador de código

Para implementar o analisador semântico, foram necessárias estruturas de dados especiais. Tanto elas quanto as ações semânticas serão especificadas nos itens a seguir.

6.1 Estruturas de dados

Tabela de símbolos

A tabela de símbolos possui os campos:

- nome do identificador;
- tipo do identificador (rótulo ou variável);
- valor (pode ser inteiro, char ou booleano).

Além disso, cada tabela de símbolos usada no compilador pode guardar referência para uma tabela de símbolos pai ou filha, de maneira que escopos de variáveis e rótulos possam ser criados no analisador semântico.

Esta estrutura é responsável por gerenciar os símbolos declarados no código fonte, sejam eles nomes de funções (rótulos) ou de variáveis. Por este motivo, existem métodos que permitem acessar tais tabelas, inserindo ou atualizando dados. Também existe um método chamado "create_new_scope" que cria uma nova tabela de símbolos, ligando-a à atual por meio do relacionamento de pai e filho.

Tabela de palavras reservadas

Esta estrutura é uma tabela que guarda a lista de palavras reservadas da linguagem Edu. Ao

inicializar o compilador, esta tabela é inicializada recebendo os valores de todas as palavras reservadas da linguagem. Após a inicialização, seu conteúdo é constante e ela é utilizada apenas para leitura. Conforme os tokens são consumidos, esta tabela é utilizada para verificar se um token lido representa um identificador ou uma palavra reservada.

Tabela de constantes

No código fonte podem ser declaradas constantes do tipo inteiro, booleano ou char. Quando o analisador léxico lê um token de um destes tipos (por exemplo, 10 para inteiro), este token é adicionado na tabela de constantes que cria um id para cada constante reconhecida no código fonte.

Pilhas de operandos e operadores

As pilhas de operadores e operandos são pilhas comuns que também possuem um método que retorna o elemento do topo sem removê-lo. Também são oferecidos métodos para verificar o tamanho da pilha e se ela está vazia. Estas duas pilhas são utilizadas durante a compilação de expressões.

Variáveis complementares

Foram utilizadas variáveis complementares que permitiram a declaração de variáveis temporárias (usadas, por exemplo, durante a compilação de uma expressão) e a declaração de rótulos da linguagem em comandos de laços ou condicionais. Algumas dessas variáveis também permitiram que certas ações semânticas fossem executadas, por exemplo, guardando resultados de operações ou endereços de retornos. Como exemplos destas variáveis, citam-se CONTATEMP e CONTAIF, que são contadores de variáveis temporárias e rótulos gerados durante a compilação.

Registro de Ativação (Stack Frame)

O Registro de Ativação consiste de uma pilha que guarda o ambiente de execução do compilador. Ela é utilizada, por exemplo, quando uma função é chamada. Neste momento, o ambiente de execução (variáveis, pilhas e tabelas) é empilhado na Stack Frame e um novo ambiente de execução é criado, evitando que conflitos entre endereços de variáveis, tipos ou declarações ocorram.

6.2 Principais rotinas semânticas

default action

Esta rotina é uma função sem retorno, não modifica nada no compilador e não gera código. Esta é a rotina padrão executada em todas as transições de estados do APE que não possuem

rotinas semânticas definidas.

create_new_scope

Esta rotina cria um novo escopo (tabela de símbolos aninhada com a atual). Ela é utilizada quando um escopo é declarado no código fonte, por exemplo entrando em um laço ou em uma área de comando condicional.

declaring_type

Esta rotina é chamada quando um tipo está sendo declarado (por exemplo, na ocorrência de um tipo de dado como int, char ou bool).

declaring_function

Esta rotina é chamada quando uma função está sendo declarada. Ela atualiza a tabela de símbolos com o rótulo da função declarada. Caso o rótulo já exista na tabela de símbolos, uma mensagem de erro é retornada.

declaring_variable

Esta rotina é chamada quando uma variável está sendo declarada. Ela atualiza a tabela de símbolos com o nome do identificador da variável. Caso o identificador já exista na tabela de símbolos, uma mensagem de erro é retornada.

comecando_atribuicao

Esta rotina é utilizada para armazenar em uma variável complementar o nome de um identificador que receberá a atribuição de uma expressão.

atribuicao_finalizada

Esta rotina é chamada depois que uma expressão é calculada, de maneira que gera o código responsável por armazenar o resultado da expressão na variável correspondente à atribuição.

definir_nome_da_variavel_retornada_por_expressao

Esta rotina é executada para armazenar o nome da última variável retornada em um cálculo de expressão.

jj_expressao_X (X = 1..13)

Cada rotina jj_expressao_X é a implementação da ação semântica X para a máquina de expressão que é apresentada no livro "Introdução à Compilação" do João José Neto.

jj_gera_codigo

Esta rotina é a implementação da ação semântica GERACÓDIGO descrita no livro "Introdução à Compilação" do João José Neto. Ela é responsável por gerar os códigos de cálculo da expressão.

jj_gera

Esta rotina é a implementação da ação semântica GERA descrita no livro "Introdução à Compilação" do João José Neto. Ela é responsável por gerar o código de uma operação em MVN.

jj_erro

Esta rotina é a implementação da ação semântica ERRO descrita no livro "Introdução à Compilação" do João José Neto. Ela é responsável por imprimir mensagens de erro e contabilizar a quantidade de erros semânticos existentes no código fonte.

save_stack_frame

Esta rotina empilha o ambiente de execução atual no Stack Frame, e cria um novo ambiente de execução.

acao_if_X (X = 1..5)

As rotinas acao_if_X são utilizadas para gerar códigos responsáveis por executar comandos condicionais na MVN. São calculados resultados de condições, são realizados desvios e criação de rótulos.

init_semantic_actions

Esta rotina inicializa vetores de funções que serão acionados para cada transição que o APE realizar.

7. Exemplo de compilação

Foi gerado um exemplo de compilação para demonstrar o correto funcionamento do compilador.

Para o código fonte abaixo:

```
main {  
    int a, int b, int resultado;
```

```

a = 20/5 + 4*2 - 1;
b = a - 50;

if (a + 2*b > 2*a + b) {
    resultado = 1;
} else {
    resultado = 2;
}
}

```

Foi gerado o código objeto a seguir:

| | | | |
|------------|----|-------|--------------------------------------|
| | @ | /0 | |
| | SC | main | |
| main | K | /0 | |
| | LD | K_0 | ; AC = K_0 |
| | / | K_1 | ; AC = AC/K_1 |
| | MM | TEMP1 | ; TEMPX = AC |
| | LD | K_2 | ; AC = K_2 |
| | * | K_3 | ; AC = AC*K_3 |
| | MM | TEMP2 | ; TEMPX = AC |
| | LD | TEMP1 | ; AC = TEMP1 |
| | + | TEMP2 | ; AC = AC+TEMP2 |
| | MM | TEMP3 | ; TEMPX = AC |
| | LD | TEMP3 | ; AC = TEMP3 |
| | - | K_4 | ; AC = AC-K_4 |
| | MM | TEMP4 | ; TEMPX = AC |
| | LD | TEMP4 | ; carrega constante no acumulador |
| | MM | a | ; armazena conteúdo do acumulador na |
| variável a | | | |
| | LD | a | ; AC = a |
| | - | K_5 | ; AC = AC-K_5 |
| | MM | TEMP5 | ; TEMPX = AC |
| | LD | TEMP5 | ; carrega constante no acumulador |
| | MM | b | ; armazena conteúdo do acumulador na |
| variável b | | | |
| ROT_1 | LD | K_3 | ; AC = K_3 |
| | * | b | ; AC = AC*b |
| | MM | TEMP6 | ; TEMPX = AC |
| | LD | a | ; AC = a |
| | + | TEMP6 | ; AC = AC+TEMP6 |
| | MM | TEMP7 | ; TEMPX = AC |
| | LD | K_3 | ; AC = K_3 |
| | * | a | ; AC = AC*a |
| | MM | TEMP8 | ; TEMPX = AC |

```

LD      TEMP8      ;      AC = TEMP8
+      b           ;      AC = AC+b
MM      TEMP9      ;      TEMPX = AC
LD      TEMP7      ;      Carrega conteúdo da expressão à
esquerda da condição no acumulador
-      TEMP9      ;      Subtrai conteúdo da expressão à
direita da condição do acumulador
JN      ROT_2      ;      Desvia se AC < 0
JZ      ROT_2      ;      Desvia se AC = 0
LD      K_4      ; carrega constante no acumulador
MM      resultado  ; armazena conteúdo do acumulador na variável
resultado
JP      ROT_3      ;      Desvia para rótulo de saída do
comando condicional
ROT_2   LD      K_3      ; carrega constante no acumulador
MM      resultado  ; armazena conteúdo do acumulador na variável
resultado
ROT_3   #      main
HM      /0

```

```

@      /200      ; começo da área de dados
a      K          =0      ; declaração da variável a
b      K          =0      ; declaração da variável b
resultado K      =0      ; declaração da variável resultado
K_0    K          =20     ; declaração da constante K_0
K_1    K          =5      ; declaração da constante K_1
K_2    K          =4      ; declaração da constante K_2
K_3    K          =2      ; declaração da constante K_3
K_4    K          =1      ; declaração da constante K_4
K_5    K          =50     ; declaração da constante K_5
TEMP1  K          =0      ; declaração da variável temporária TEMP1
TEMP2  K          =0      ; declaração da variável temporária TEMP2
TEMP3  K          =0      ; declaração da variável temporária TEMP3
TEMP4  K          =0      ; declaração da variável temporária TEMP4
TEMP5  K          =0      ; declaração da variável temporária TEMP5
TEMP6  K          =0      ; declaração da variável temporária TEMP6
TEMP7  K          =0      ; declaração da variável temporária TEMP7
TEMP8  K          =0      ; declaração da variável temporária TEMP8
TEMP9  K          =0      ; declaração da variável temporária TEMP9

```

Executando o código na mvn:

1. Área de dados na memória após carregar o código de saída na MVN:

```

[200]: 00 00 00 00 00 00 00 14 00 05 00 04 00 02 00 01
[210]: 00 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

[220]: 00 00 00 00

2. Execução do programa:

| MAR | MDR | IC | IR | OP | OI | AC |
|------|------|------|------|------|------|------|
| ---- | ---- | ---- | ---- | ---- | ---- | ---- |
| 0000 | A002 | 0004 | A002 | 000A | 0002 | 0000 |
| 0004 | 8206 | 0006 | 8206 | 0008 | 0206 | 0014 |
| 0006 | 7208 | 0008 | 7208 | 0007 | 0208 | 0004 |
| 0008 | 9212 | 000A | 9212 | 0009 | 0212 | 0004 |
| 000A | 820A | 000C | 820A | 0008 | 020A | 0004 |
| 000C | 620C | 000E | 620C | 0006 | 020C | 0008 |
| 000E | 9214 | 0010 | 9214 | 0009 | 0214 | 0008 |
| 0010 | 8212 | 0012 | 8212 | 0008 | 0212 | 0004 |
| 0012 | 4214 | 0014 | 4214 | 0004 | 0214 | 000C |
| 0014 | 9216 | 0016 | 9216 | 0009 | 0216 | 000C |
| 0016 | 8216 | 0018 | 8216 | 0008 | 0216 | 000C |
| 0018 | 520E | 001A | 520E | 0005 | 020E | 000B |
| 001A | 9218 | 001C | 9218 | 0009 | 0218 | 000B |
| 001C | 8218 | 001E | 8218 | 0008 | 0218 | 000B |
| 001E | 9200 | 0020 | 9200 | 0009 | 0200 | 000B |
| 0020 | 8200 | 0022 | 8200 | 0008 | 0200 | 000B |
| 0022 | 5210 | 0024 | 5210 | 0005 | 0210 | FFD9 |
| 0024 | 921A | 0026 | 921A | 0009 | 021A | FFD9 |
| 0026 | 821A | 0028 | 821A | 0008 | 021A | FFD9 |
| 0028 | 9202 | 002A | 9202 | 0009 | 0202 | FFD9 |
| 002A | 820C | 002C | 820C | 0008 | 020C | 0002 |
| 002C | 6202 | 002E | 6202 | 0006 | 0202 | FFB2 |
| 002E | 921C | 0030 | 921C | 0009 | 021C | FFB2 |
| 0030 | 8200 | 0032 | 8200 | 0008 | 0200 | 000B |
| 0032 | 421C | 0034 | 421C | 0004 | 021C | FFBD |
| 0034 | 921E | 0036 | 921E | 0009 | 021E | FFBD |
| 0036 | 820C | 0038 | 820C | 0008 | 020C | 0002 |
| 0038 | 6200 | 003A | 6200 | 0006 | 0200 | 0016 |
| 003A | 9220 | 003C | 9220 | 0009 | 0220 | 0016 |
| 003C | 8220 | 003E | 8220 | 0008 | 0220 | 0016 |
| 003E | 4202 | 0040 | 4202 | 0004 | 0202 | FFEF |
| 0040 | 9222 | 0042 | 9222 | 0009 | 0222 | FFEF |
| 0042 | 821E | 0044 | 821E | 0008 | 021E | FFBD |
| 0044 | 5222 | 0046 | 5222 | 0005 | 0222 | FFCE |
| 0046 | 2050 | 0050 | 2050 | 0002 | 0050 | FFCE |
| 0050 | 820C | 0052 | 820C | 0008 | 020C | 0002 |
| 0052 | 9204 | 0054 | 9204 | 0009 | 0204 | 0002 |
| 0054 | C000 | 0000 | C000 | 000C | 0000 | 0002 |

3. Área de dados na memória após carregar o código de saída na MVN:

```
[200]: 00 0B FF D9 00 02 00 14 00 05 00 04 00 02 00 01  
[210]: 00 32 00 04 00 08 00 0C 00 0B FF D9 FF B2 FF BD  
[220]: 00 16 FF EF
```

4. Conclusões

Verifica-se que após a execução do programa, os seguintes valores são obtidos:

a = 11

b = -39

resultado = 2

Desta forma, pode-se concluir que o código objeto foi gerado corretamente a partir do código fonte.