Escola Politécnica
da Universidade de
São Paulo

Relatório Final

PCS-2056 : Linguagens e Compiladores

Autores:
Eduardo Russo
Helio Kazuo Nagamachi

Professor:
Ricardo Rocha

9 de dezembro de 2010

Sumário

1 Introdução

Compiladores são programas capazes de traduzir o texto fonte de uma linguagem específica para o texto objeto de uma outra linguagem específica, normalmente a segunda linguagem é o formato entendido por um processador para executar programas.

Nas aulas foram vistos métodos e técnicas para analisar sobre o ponto de vista sintático o texto fonte de um programa, e técnicas para a geração de código executável.

O objetivo deste projeto é exercitar os conceitos vistos em aula para modelar e construir um compilador que utilize uma linguagem definida pelos alunos e gere código objeto para a MVN.

2 Definição da linguagem de alto nível

A definição da linguagem em notação de WIRTH apresenta-se no código 1 a seguir:

Código 1: Gramática definida.

```
1  programa = lista_import  lista_tipos lista_funcoes.
2  lista_import = {"import" "string" ";"}.
3  lista_funcoes = {declaracao_funcao}.
4  lista_tipos = {declaracao_tipo}.
5
6  declaracao_variavel = tipo [indice_vetor_matriz] "identificador" {"," "
       identificador"} ";".
7  declaracao_tipo = "typedef" "identificador" "{" { declaracao_variavel }
       "}".
8
9
10 declaracao_funcao =  (tipo | "void" )"identificador" "(" [tipo [
       indice_vetor_matriz] "identificador" {","tipo [indice_vetor_matriz] "
       identificador" } ]")" bloco_codigo .
11
12
13 indice_vetor_matriz = "[" ("identificador" | "numero") "]" {"[" ("
       identificador" | "numero") "]"}.
14
15 chamada_funcao_procedimento = "identificador" "(" [ "identificador" { "," "
       identificador"  } ] ")".
16
17
18 rotulo = "identificador" ":".
19 desvio = ("continue" ["identificador"]) | ("break" ";").
20 condicional = "if" "(" expressao ")" bloco_codigo ["else"  bloco_codigo ].
21 laco = "while" "("expressao")" bloco_codigo.
22
23 bloco_codigo = "{" {declaracao_variavel} { atribuicao | condicional | laco
       | desvio | retorno | chamada_funcao_procedimento";" }  "}".
24
25 retorno = "return" ("identificador" | chamada_funcao_procedimento |
       expressao | expressao) ";".
26
27
28
```

```
29 atribuicao = "identificador" "=" (expressao| chamada_funcao_procedimento |
       "identificador" ) ";".
30
31 expressao = {("!" | "-")} ( (termo ) | (termo {operadores (termo | "("
       expressao ")")} )| ("(" expressao ")" )) | chamada_funcao_procedimento.
32
33 termo = "identificador" | "true" | "false" | "Char" | "numero".
34
35 operadores = "+" | "-" | "/" | "*" | "||" | "&&" | "==" | ">=" | "<=" |
       "!=" | "<" | ">".
36
37 booleano = "true" | "false".
38
39 tipo = "bool" | "int" | "String" | "identificador" | "char".
```

Na definição da gramática não há detalhamento de como são os identificadores, números e caracteres. Eles são reconhecidos pelo analisador léxico e já são recebidos pelo analisador sintático como tokens – ou átomos – prontos e apropriados, por isso, na gramática aparecem como elementos finais.

Uma breve descrição de como cada um dos elementos citados acima, são esperados é mostrado a seguir:

- **Identificador:** pelo menos uma letra, seguida de letras e números. Não pode ser uma palavra reservada.
- **Número:** números inteiros sem sinal, e de ponto flutuante, no entanto como a MVN só trabalha com inteiros, somente a parte inteira é considerada.
- **Caractere:** uma letra entre aspas simples.
- **String:** Começa com aspas duplas, uma sequência qualquer de caracteres e termina com aspas duplas.

3 Análise léxica

A análise léxica é a primeira feita sobre os programas escritos. Basicamente, separa o texto fonte em átomos que podem ser utilizados nas etapas posteriores da compilação.

Na análise léxica, todo o conteúdo irrelevante para a compilação – espaços em branco, marcadores de nova linha e comentários – é descartado.

3.1 Tokens

A análise léxica do compilador se dá basicamente pelo reconhecimento dos tokens definidos para a linguagem, são eles:

- **Identificador**
- **Número**
- **Palavra Reservada**

- **Caractere**
- **Comentário**
- **String**

Para o reconhecimento dos átomos, lançou-se mão do uso de autômatos finitos. Estes aceitam caracteres como entrada e, ao chegar em um estado final, podem emitir um token.

Os tokens foram modelados com os três campos a seguir:

- **Valor**
- **Tipo**
- **Linha**

Os valores possíveis para o token dependem diretamente do seu tipo, como pode ser visto na tabela 1

Tabela 1: *Tipos de tokens e valores.*

Tipo de token	**Valores Possíveis**
Número	0 ,1 , 2 ,3 , 4 ,...
Palavra reservada	Número associado à palavra reservada
Identificador	Inicialmente o valor é um numero qualquer, depois é associado a um ID
Comentário	O valor assumido é sempre 0
Caractere	O valor é o equivalente em código ascii do caractere
String	Valor correspondente a uma entrada em uma tabela de constantes

3.2 Autômatos

Os autômatos para o reconhecimento dos tokens foram modelados da seguinte forma: todos herdam da classe automata.java – código 2.

O campo **State** representa o estado do autômato. Os estados são membros de uma enumeração pública que tem uma única propriedade: se o estado é final ou não – código 3.

Cada um dos autômatos implementa de forma diferente o método *processChar(char a)*, que deve retornar um booleano ao final do método. O booleano a ser retornado deve ter valor "verdadeiro" caso o autômato possa consumir o caractere fornecido, e "falso" caso contrário. O analisador léxico deve, então, verificar se o autômato chegou a um estado final ou não e pedir a emissão do token caso o estado atual seja um estado final.

Dependendo do estado e de como ocorreu o processamento, o autômato pode ser solicitado a emitir um átomo baseado no seu estado atual.

3.3 Analisador Léxico

O analisador é a classe que tem o contato direto com o arquivo de texto fonte e sua função é obter os tokens do texto fonte a ser compilado.

Em sua estrutura, há uma lista com todos os autômatos, e, associado a cada um deles, um booleano que indica se a aquele autômato já retornou "falso" alguma vez na análise do token atual.

O funcionamento do analisador léxico é descrito da seguinte forma: Ao ser iniciado, o analisador cria os meios para leitura de caracteres um a um do arquivo fonte e lê o primeiro caractere do arquivo.

Quando o método *getNexttoken(table)* é chamado, o analisador sintático executa os seguintes passos:

1. Verifica se o último caractere é de espaço em branco (*whitespace*). Se for, lê o próximo até achar um que não seja.
2. Verifica se o último caractere é o marcador de fim de arquivo. Se sim, retorna o token de EOF.
3. Pede a todos os autômatos para processarem o caractere. Aqueles que retornarem "falso" são marcados como autômatos desabilitados para o próximo caractere.
4. Lê o próximo caractere.
5. Aos autômatos restantes, oferece o caractere para processamento.
6. Verifica se ainda há autômatos habilitados.
7. Se sim, volta a repetir desde o item 4.
8. Se não há autômatos habilitados, verifica o último autômato a ser desativado.
9. Se esse autômato está em estado final, recupera o token, reinicializa os autômatos e retorna o token.
10. Se o último autômato não estiver em um estado final, isso caracteriza um estado de erro. O analisador sintático pára o processo – por exemplo, na presença de um caractere que não seja ASCII no nome de um identificador.

Se o token obtido no processo for de comentário, o analisador léxico o descarta e avança para o próximo.

Se o token obtido for um identificador, o analisador léxico recupera o nome do identificador no autômato de palavras e insere a palavra na tabela de símbolos passada. O índice retornado pela tabela de símbolo é o valor que o token vai assumir.

4 Analise sintática

A análise sintática se utiliza de um Autômato de Pilha Estruturado (APE) que é descrito por [Neto, 1987], estrutura que utiliza uma pilha e alguns autômatos finitos para realizar a tarefa de análise sintática.

4.1 Estrutura

O analisador sintático possuí, basicamente, um APE e acesso a códigos que permitem o *parse* de arquivos de configuração dos autômatos finitos.

4.1.1 Autômato Finito

O autômato finito utilizado para a análise sintática possuí um vetor de booleanos que denota os seus estados. Se uma posição do vetor possuir o booleano "verdadeiro", quer dizer que aquele estado é final. Há também um vetor com as transições do autômato.

4.1.2 Transição

A transição pode ser de dois tipos: transição normal ou a chamada de outro autômato.

Se a transição for normal, ela possuí o próximo estado e o token que a ativa.

Se a transição for uma chamada, ela contém o número do autômato a ser chamado e o estado de retorno, que é um estado no autômato atual que será o estado corrente após a execução do segundo autômato.

4.1.3 Pilha

A pilha é a estrutura utilizada pelo APE para armazenar a informação de autômato e estado quando a transição de um autômato a ser executada é a chamada de outro autômato.

Quando esse tipo de transição ocorre, o autômato corrente passa a ser o autômato denotado pela transição e a pilha guarda qual o autômato que realizou a chamada e qual o estado que esse deve voltar após a execução do segundo autômato.

4.2 Parse

A gramática em notação de Wirth foi reduzida de forma a ficarmos com 3 autômatos: um de programa, um de bloco de código e o terceiro de expressões.

A gramática foi transformada em autômatos finitos determinísticos mínimos com o uso da ferramenta criada por Hugo Baraúna e Fabio Yamate [Baraúna and Yamate, 2009], um meta compilador de definições de gramática em notação Wirth.

Um exemplo da saída para um dos autômatos encontra-se na seção de anexo. As transições do autômato de bloco estão nos anexos – código **??**. A primeira linha do arquivo é sempre igual, indicando que o estado inicial do autômato é o estado zero, a segunda linha mostra quais são os estados finais e as outras linhas indicam as transições. Se o elemento da transição representar um token é uma transição normal, caso seja o nome de um autômato, indica a chamada para o outro autômato.

Conforme o arquivo de configuração é lido, descobre-se ao final a quantidade de estados do autômato.

4.3 Análise Sintática

A análise sintática se dá com a requisição dos tokens um a um do sintático para o léxico.

O analisador gerencia o APE e, na eventualidade de um erro, o processo para e uma mensagem de erro é apresentada. Caso o átomo que marca o fim do texto fonte seja recebido, o analisador deve verificar se não há dados na pilha de autômatos e se o autômato atual é o primeiro autômato – no caso o autômato referente a programa – e o estado corrente desse autômato é final. Caso essas condições sejam atendidas, o programa é válido sob o ponto de vista sintático.

5 Definição do ambiente de execução

O ambiente de execução é composto por uma coleção de bibliotecas escritas em baixo nível, assembly da MVN para prover algumas funcionalidades básicas. São elas:

- função lógica OU
- função lógica E
- função comparativa $>$
- função comparativa $<$
- função comparativa $>=$
- função comparativa $<=$
- função comparativa $==$
- função comparativa $!=$
- função lógica !
- função print de strings
- função de print de inteiros

A ideia é manter esses arquivos com essas funções em arquivos separados, pelo uso da biblioteca do montador, ligador e relocador, um único arquivo do programa principal pode ser gerado.

5.1 Registros de Ativação

A implementação da MVN não tem uma pilha em seu ambiente de execução, isso implica na impossibilidade de realizar chamadas de função recursivamente. Um modo de fazer isso é utilizando-se de registros de ativação para contornar o problema.

6 Análise Semântica

Na análise semântica é realizada a geração de código e outras verificações que não podem ser feitas apenas com a definição da gramática

6.1 Introdução das ações semânticas

As ações semânticas foram introduzidas na gramática, em sua notação de Wirth. Para podermos continuar a utilizar a ferramenta

6.2 Estruturas

Para algumas ações semânticas foram criadas estruturas de dados específicas para auxiliar na sua função.

6.3 Ações

Uma ação semântica é executada quando o ocorre a transição a que ela está associada. Por necessidade, cada autômato, pode ter uma ação final, que é executada apenas quando este retorna para outro autômato.

6.3.1 push_term

Ação Semântica da máquina de expressão que empilha o operando na pilha de operandos.

6.3.2 push_op

Ação semântica da máquina de expressão que empilha o operador na pilha de operadores, faz verificação com o topo atual da pilha para que as operações aritméticas possam ser realizadas de acordo com sua prioridade.

Seu funcionamento é enumerado a seguir:

1. Verifica o topo da pilha de operadores.
2. Se o operador que está no topo da pilha é mais prioritário do que o novo operador, desempilha dois operandos e o operador, gera o código da operação e gera uma *label* temporária que irá receber o resultado da operação realizada. Essa *label* é empilhada na pilha de operandos e volta para o passo inicial.
3. Se o topo está vazio, ou o operador que está no topo tem prioridade menor ou igual que o novo operador, simplesmente empilha o novo operador.

6.3.3 Expressão - ação final

Quando a máquina de expressão deve retornar de sua chamada, deve, também, gerar o código correspondente à expressão presente na pilha. A geração de código é bastante simples, uma vez que a complexidade de lidar com a precedência de operações já foi resolvida. Essa ação pára no momento em que um token de abertura de parênteses aparece na pilha de operandos, indicando que a ação deve simplesmente guardar o resultado em uma variável temporária para outro processamento de expressão. Essa variável temporária é empilhada na pilha de operandos e, se não houver mais dados para serem processados, o resultado é armazenado em uma nova variável e também estará à disposição para outros comandos no acumulador.

6.3.4 nova_var

Ação semântica que associa uma nova *label* à variável criada.

6.3.5 atribuicao

Atribui um valor a variável que está à esquerda no comando, atualmente apenas inteiros, booleanos e caracteres podem ser atribuídos.

6.3.6 fecha_escopo

Emite uma *label* para o final do escopo corrente, no caso de *while* e *else*, eles podem emitir uma instrução que leva até essa *label* antes mesmo dela ser sido declarada, por isso, deve verificar se há alguma desse tipo presente em uma pequena pilha para essa finalidade.

6.3.7 novo_escopo

Emite uma *label* correspondente ao novo escopo. Pode ser utilizada pela instrução *while*, que, ao chegar ao seu final, faz um desvio incondicional para essa *label*.

7 Implementação e Resultados

A implementação do compilador permite ao usuário as funcionalidades de mostrar na tela um inteiro de que esteja contido em uma variável, expressões aritméticas, considerando prioridade e expressões booleanas, sem consideração a respeito das expressões boolenas.

Infelizmente as bibliotecas do montador, relocador e lingador apresentam algum erro em sua implementação e acabam inserindo no código binário final no meio do arquivo a linha **0000 0002**, um desvio incondicional para o endereço 2 de memória, arruinando os resultados.

Referências

[Baraúna and Yamate, 2009] Baraúna, H. and Yamate, F. (2009). Compiler::wirth. `http://radiant-fire-72.heroku.com/`.

[Neto, 1987] Neto, J. J. (1987). *Introdução à Compilação*. Livros Técnicos e Científicos, Rio de Janeiro RJ, 1st edition.

8 Anexos

Código 2: Automata.java

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5
6  package automaton;
7
8  import lex.Token;
9
10 /**
11  *
12  * @author Helio
13  */
14 public abstract class Automata {
15
16     protected State currentState = State.INITIAL;
17
18
19     /**
20      * The automaton receives the character, if it can still
21      * process another char returns true, if it is on the final state or
22      * reached an error state, returns false.
23      * @param a the character to be consumed
24      * @return true if it can continue to receive more chars, or false if
          it
25      * is in the final state or an error state.
26      */
27     public abstract boolean processChar(char a);
28
29
30     /**
31      * Gets the current state of the automaton.
32      * @return the state
33      */
34     public State getState(){
35       return this.currentState;
36     };
37
38     /**
39      * Resets the automata to the initial state.
40      */
41     public void resetAutomata(){
42         this.currentState = State.INITIAL;
43     }
44
45     /**
46      * Gets the token to the machine state, if none, null should be
          returned
47      * This method should be used only if the automata reaches a final
          state;
48      * @return the token
```

```
49          */
50      public abstract Token getToken();
51
52
53
54      @Override
55      public abstract String toString();
56
57  }
```

Código 3: State.java

```
1   /*
2    * To change this template, choose Tools | Templates
3    * and open the template in the editor.
4    */
5
6   package automaton;
7
8   /**
9    *
10   * @author Helio
11   */
12  public enum  State {
13      //Default states for every automata.
14      INITIAL,  ERROR,
15      //States for the comment automata.
16      COMMENT_START, COMMENT_LINE , COMMENT_BLOCK , COMMENT_BLOCK_END,
            COMMENT_END(true),
17      //States for number automata,
18      INTEGER(true), FLOAT(true), FLOAT_STARTED_BY_DOT,
19      //States for String,
20      STRING_CONTENT, STRING_FINAL(true),
21      //For reserved words and identifiers...
22      POSSIBLE_RESERVED_WORD, POSSIBLE_OPERATOR, RESERVED_WORD(true),
            IDENTIFIER(true),
23      //Characters
24      ONE_QUOTE, POSSIBLE_CHAR ,POSSIBLE_ESCAPE_CHAR ,CHAR(true),
25      ;
26
27      private boolean finalState;
28      private State(){
29          this.finalState = false;
30      }
31      private State(boolean finalState){
32          this.finalState = finalState;
33      }
34
35      public boolean isFinalState(){
36          return this.finalState;
37      }
38  }
```

Código 4: Autômato de bloco de Código.

```
initial: 0
final: 13
(0, "{") -> 1
(1, new_context) -> 2
(2, "bool") -> 3
(2, "int") -> 3
(2, "String") -> 3
(2, "identificador") -> 4
(2, "char") -> 3
(2, "if") -> 5
(2, "while") -> 6
(2, "continue") -> 7
(2, "break") -> 8
(2, "return") -> 9
(2, "}") -> 10
(3, "identificador") -> 11
(3, "[") -> 12
(4, "identificador") -> 11
(4, "[") -> 12
(4, "=") -> 18
(4, "(") -> 19
(5, "(") -> 25
(6, "(") -> 31
(7, "identificador") -> 29
(7, "if") -> 5
(7, "while") -> 6
(7, "continue") -> 7
(7, "break") -> 8
(7, "return") -> 9
(7, "}") -> 10
(8, ";") -> 16
(9, "identificador") -> 15
(9, expressao) -> 8
(10, end_block_code) -> 13
(11, new_var) -> 21
(12, "identificador") -> 14
(12, "numero") -> 14
(14, "]") -> 3
(15, ";") -> 16
(15, "(") -> 19
(16, "identificador") -> 17
(16, "if") -> 5
(16, "while") -> 6
(16, "continue") -> 7
(16, "break") -> 8
(16, "return") -> 9
(16, "}") -> 10
(17, "=") -> 18
(17, "(") -> 19
(18, expressao) -> 8
(19, "identificador") -> 20
(19, ")") -> 8
(20, ",") -> 22
(20, ")") -> 8
(21, ",") -> 23
```

```
(21, ";") -> 24
(22, "identificador") -> 20
(23, "identificador") -> 11
(24, fim_vars) -> 2
(25, expressao) -> 26
(26, ")") -> 27
(27, bloco_codigo) -> 28
(28, "identificador") -> 17
(28, "if") -> 5
(28, "else") -> 30
(28, "while") -> 6
(28, "continue") -> 7
(28, "break") -> 8
(28, "return") -> 9
(28, "}") -> 10
(29, "identificador") -> 17
(29, "=") -> 18
(29, "if") -> 5
(29, "(") -> 19
(29, "while") -> 6
(29, "continue") -> 7
(29, "break") -> 8
(29, "return") -> 9
(29, "}") -> 10
(30, bloco_codigo) -> 16
(31, expressao) -> 32
(32, ")") -> 30
```

Código 5: Autômato de expressão.

```
initial: 0
final: 8, 9
(0, "!") -> 1
(0, "-") -> 1
(0, "identificador") -> 2
(0, "true") -> 3
(0, "false") -> 3
(0, "Char") -> 3
(0, "numero") -> 3
(0, "(") -> 4
(1, push_op) -> 5
(2, push_term) -> 8
(2, "(") -> 15
(3, push_term) -> 8
(4, push_op) -> 6
(5, "!") -> 1
(5, "-") -> 1
(5, "identificador") -> 3
(5, "true") -> 3
(5, "false") -> 3
(5, "Char") -> 3
(5, "numero") -> 3
(5, "(") -> 4
(6, expressao) -> 7
(7, ")") -> 9
(8, "-") -> 10
```

```
(8, "+") -> 10
(8, "/") -> 10
(8, "*") -> 10
(8, "&&") -> 10
(8, "==") -> 10
(8, ">=") -> 10
(8, "<=") -> 10
(8, "!=") -> 10
(8, "<") -> 10
(8, ">") -> 10
(8, "||") -> 10
(10, push_op) -> 11
(11, "identificador") -> 3
(11, "true") -> 3
(11, "false") -> 3
(11, "Char") -> 3
(11, "numero") -> 3
(11, "(") -> 12
(12, push_op) -> 13
(13, expressao) -> 14
(14, ")") -> 8
(15, "identificador") -> 16
(15, ")") -> 9
(16, ")") -> 9
(16, ",") -> 17
(17, "identificador") -> 16
```

Código 6: Autômato de programa.

```
initial: 0
final: 8, 13, 19
(0, "import") -> 1
(0, "typedef") -> 2
(0, "identificador") -> 3
(0, "bool") -> 3
(0, "int") -> 3
(0, "String") -> 3
(0, "char") -> 3
(0, "void") -> 3
(1, "string") -> 4
(2, "identificador") -> 12
(3, "identificador") -> 5
(4, ";") -> 6
(5, "(") -> 7
(6, fim_import) -> 8
(7, "identificador") -> 9
(7, "bool") -> 9
(7, "int") -> 9
(7, "String") -> 9
(7, "char") -> 9
(7, ")") -> 10
(8, "import") -> 1
(8, "typedef") -> 2
(8, "identificador") -> 3
(8, "bool") -> 3
(8, "int") -> 3
```

```
(8, "String") -> 3
(8, "char") -> 3
(8, "void") -> 3
(9, "identificador") -> 17
(9, "[") -> 18
(10, new_func) -> 11
(11, bloco_codigo) -> 13
(12, "{") -> 14
(13, "identificador") -> 3
(13, "bool") -> 3
(13, "int") -> 3
(13, "String") -> 3
(13, "char") -> 3
(13, "void") -> 3
(14, "identificador") -> 15
(14, "bool") -> 15
(14, "int") -> 15
(14, "String") -> 15
(14, "char") -> 15
(14, "}") -> 16
(15, "identificador") -> 21
(15, "[") -> 22
(16, fim_new_type) -> 19
(17, ",") -> 24
(17, ")") -> 10
(18, "identificador") -> 20
(18, "numero") -> 20
(19, "typedef") -> 2
(19, "identificador") -> 3
(19, "bool") -> 3
(19, "int") -> 3
(19, "String") -> 3
(19, "char") -> 3
(19, "void") -> 3
(20, "]") -> 9
(21, new_var) -> 25
(22, "identificador") -> 23
(22, "numero") -> 23
(23, "]") -> 15
(24, "identificador") -> 9
(24, "bool") -> 9
(24, "int") -> 9
(24, "String") -> 9
(24, "char") -> 9
(25, ";") -> 14
(25, ",") -> 26
(26, "identificador") -> 21
```