Compiladores - Agoravai

Valério N. R. Júnior

Agosto 2021

1 Especificação da linguagem

1.1 Estrutura geral

Um programa escrito na linguagem Agoravai precisa conter uma função chamada `main` que recebe um *array* de **strings** (argumentos da linha de comando) e retorna um código de erro (0 caso o programa seja executado com sucesso). Comentários iniciam com `//` e o restante da linha em que aparecem é ignorado. Um bloco de código é uma região delimitada por `{` e `}`. Variáveis podem ser declaradas globalmente (fora de qualquer função/bloco) ou localmente (dentro de uma função ou bloco), assim como subprogramas. O ambiente de referenciamento da variável é o bloco a qual ela pertence no caso de váriaveis locais, e qualquer ponto do programa para as globais. Para referenciar qualquer variável em um determinado ponto do programa é necessário que essa variável tenha sido declarada em um ponto anterior. Funções e variáveis podem ser declaradas

1.1.1 Nomes

A tabela1 mostra as palavras reservadas da linguagem. Identificadores de variáveis e funções devem iniciar com uma letra ou _ e podem conter letras, dígitos ou _, sem limite de caracteres e obviamente não podem estar contidos no conjunto das palavras reservadas.

1.1.2 Declarações

Variáveis Para declarar uma variável é necessário especificar o tipo e o nome dela. Opcionalmente, é possível inicializá-la com uma expressão cujo resultado seja do mesmo tipo da variável ou pode sofrer uma coerção para ele. Também é

bool	break	char	if	else
false	float	for	int	in
print	return	scan	skip	string
true	void	while		

Tabela 1: Palavras reservadas

possível declarar mais de uma variável do mesmo tipo em uma única sentença, separando-as por vírgula. Ex.: `int a;`, `int a = 1`, `int b = 2, c = 4;` e `int d = b + c`.

Subprogramas Subprogramas podem ser procedimentos ou funções. A sintaxe de declaração de um subprograma é mostrada na listagem 1 O retorno da função deve ser uma expressão compatível com o tipo declarado na assinatura da função. Para mais informações sobre subprogramas ver seção 1.4.

Listing 1: Declaração de subprogramas

```
// Procedimento
void my_procedure(int a) {
   ...
}

// Funcao
int my_func(int a) {
   ...
   return (int) ...;
}
```

A listagem 2 mostra exemplos de construções da linguagem.

Listing 2: Estrutura básica da linguagem Agoravai

```
//
// Agoravai
//

// Comentarios comecam com "//". Quando um comentario eh
   red↪ detectado
// o resto da linha  ignorado

// Ponto de entrada do programa
int main (string[] args) {
   // Declaracao de variaveis
   bool b1 = true, b2;
   string message = "Ola mundo!";
   int[] a1 = [0, 1, 2, 3, 4, 5, 6, 7, 8]; // array de
      red↪ inteiros com tamanho 9 (implicito)
   int[9] a2; // array de inteiros com tamanho 9 (
      red↪ explicito)

   // Condicional
   if (b1 && b2) {
      print("b1 eh igual a b2");
   }
```

```
   else {

   }

   for(int i : [1, 3, ..., 9)) { // for num intervalo
      red↪ numerico
      print("%4d\n" % i);
   }

   for(char c : message) { // for numa string
      print(c);
   }

   int j = 0;
   for(int i : a1) { // for num array
      a2[j] = i;
      j = j + 1;
   }

   if(a1 == a2) { // verdade
      print("a1 eh igual a a2\n");
   }

   // Declaracao de funcoes
   int f (int a) {
      return 2 * a;
   }

   int result = f(20 + a[2]);

   return 0;
}
```

1.2 Especificação de tipos

1.2.1 bool

Representa valores booleanos.

Constantes Os únicos valores possíveis são `true` (verdadeiro) e `false` (falso).

Operadores A tabela 2 mostra os operadores para valores do tipo **bool**. Considere que `a` e `b` são duas variáveis **bool**.

Coerções A tabela 3 mostra as possíveis coerções do tipo **bool**.

Operador	Associatividade	Precedência	Descrição
`==`	Esquerda	1	`a` é igual a `b`?
`!=`	Esquerda	1	`a` é diferente de `b`?
`&&`	Esquerda	2	`a` e `b` são verdadeiros?
`||`	Esquerda	2	`a` ou `b` são verdadeiros?
`!`	Direita	3	o inverso de `a`
`++`	Esquerda	2	concatena a **string** que representa o valor de `a`

Tabela 2: Operadores do tipo **bool**

Tipo	Descrição
int	Retorna `1` para `true` e `0` para `false`
string	Retorna `"true"` para `true` e `"false"` para `false`

Tabela 3: Coerções do tipo bool

1.2.2 char

Representa um caractere. O caractere deve pertencer à tabela ASCII. Caracteres de controle são representados por uma sequência de escape (ver tabela **??**)

Constantes Um caractere ou uma sequência de escape entre `'` (apóstrofos). Ex.: `'a'`, `'0'`, `'
n'`.

Operadores A tabela 4 mostra os operadores para valores do tipo **char**. Considere que `a` e `b` são duas variáveis **char**.

Operador	Associatividade	Precedência	Descrição
`==`	Esquerda	1	`a` é igual a `b`?
`!=`	Esquerda	1	`a` é diferente de `b`?
`<`	Esquerda	1	`a` é menor que `b`?
`<=`	Esquerda	1	`a` é menor ou igual a `b`?
`>`	Esquerda	1	`a` é maior que `b`?
`>=`	Esquerda	1	`a` é maior ou igual a `b`?
`++`	Esquerda	2	**string** resultado da concatenação de `a` e `b`

Tabela 4: Operadores do tipo **char**

Coerções A tabela 5 mostra as possíveis coerções do tipo **char**.

1.2.3 int

Representa números inteiros.

Tipo	Descrição
int	Retorna o valor do caractere na tabela ASCII
string	Retorna uma **string** composta unicamente pelo caractere

Tabela 5: Coerções do tipo **char**

Constantes Sequência de dígitos (0 a 9), sem zeros à esquerda. Podem conter sinal ou não.

Operadores A tabela 6 mostra os operadores para valores do tipo **int**. Considere que `a` e `b` são duas variáveis **int**.

Operador	Associatividade	Precedência	Descrição
==	Esquerda	1	`a` é igual a `b`?
!=	Esquerda	1	`a` é diferente de `b`?
<	Esquerda	1	`a` é menor que `b`?
<=	Esquerda	1	`a` é menor ou igual a `b`?
>	Esquerda	1	`a` é maior que `b`?
>=	Esquerda	1	`a` é maior ou igual a `b`?
+	Esquerda	2	soma de `a` e `b`
-	Esquerda	2	subtração de `a` por `b`
*	Esquerda	3	produto de `a` e `b`
/	Esquerda	3	divisão de `a` por `b`
%	Esquerda	4	resto da divisão de `a` por `b`
++	Esquerda	2	concatena a **string** que representa o valor de `a`
+ (unário)	Direita	4	não faz nada
- (unário)	Direita	4	oposto de `a`

Tabela 6: Operadores do tipo **int**

Coerções A tabela 7 mostra as possíveis coerções do tipo **int**.

Tipo	Descrição
bool	Retorna `true` para números diferentes de 0 e `false` para 0
float	Retorna o número de ponto flutuante mais próximo
string	Retorna a **string** com a representação do número

Tabela 7: Coerções do tipo **int**

1.2.4 float

Representa números reais.

Constantes Sequência de dígitos (0 a 9) representando a parte inteira, seguida de `.` e por fim uma sequência de dígitos representando a parte decimal

Operadores A tabela 8 mostra os operadores para valores do tipo **float**. Considere que `a` e `b` são duas variáveis **float**.

Operador	Associatividade	Precedência	Descrição
==	Esquerda	1	`a` é igual a `b`?
!=	Esquerda	1	`a` é diferente de `b`?
<	Esquerda	1	`a` é menor que `b`?
<=	Esquerda	1	`a` é menor ou igual a `b`?
>	Esquerda	1	`a` é maior que `b`?
>=	Esquerda	1	`a` é maior ou igual a `b`?
+	Esquerda	2	soma de `a` e `b`
−	Esquerda	2	subtração de `a` por `b`
*	Esquerda	3	produto de `a` e `b`
/	Esquerda	3	divisão de `a` por `b`
++	Esquerda	2	concatena a **string** que representa o valor de `a`
+ (unário)	Direita	4	não faz nada
− (unário)	Direita	4	oposto de `a`

Tabela 8: Operadores do tipo **float**

Coerções A tabela 9 mostra as possíveis coerções do tipo **float**.

Tipo	Descrição
string	Retorna a **string** com a representação do número

Tabela 9: Coerções do tipo **float**

1.2.5 string

Representa uma cadeia de caracteres.

Constantes

Operadores A tabela 10 mostra os operadores para valores do tipo **string**. Considere que `a` e `b` são duas variáveis **string**.

Coerções A tabela 11 mostra as possíveis coerções do tipo **string**.

Operador	Associatividade	Precedência	Descrição
==	Esquerda	1	`a` é igual a `b`?
!=	Esquerda	1	`a` é diferente de `b`?
<	Esquerda	1	`a` vem antes de `b`?
<=	Esquerda	1	`a` vem antes ou é igual a `b`?
>	Esquerda	1	`a` vem depois de `b`?
>=	Esquerda	1	`a` vem depois ou é igual a `b`?
++	Esquerda	2	concatenação de `a` e `b`

Tabela 10: Operadores do tipo **string**

Tipo	Descrição
char[]	Retorna um *array* de **char** onde os elementos são os caracteres da **string**

Tabela 11: Coerções do tipo **string**

1.2.6 Arrays

Representa uma coleção homôgenea de valores com tamanho fixo. A declaração de um array é feita de um tipo seguido de `[]`. Ex.: **string[]**, **int[]** etc. O tamanho pode ser explícito ou inferido a partir do valor inicial. Se for explícito só pode ser iniciado com um array de tamanho igual ou menor. Caso seja omitido, o array deverá ser inicializado e terá o tamanho do valor inicial.

Constantes Um array do tipo T é escrito como uma sequência de elementos do tipo T, separados por vírgula e entre colchetes. Ex.: `[0, 1, 2, 3]`, `['a', 'b', 'c', 'd', 'e']`. Também é possível criar um array com a sintaxe `[<inicio> ... <fim>]`, que cria um array com os elementos de inicio até fim (aberto) onde início e fim devem ser valores de tipos numéricos ou caracteres (coerção para **int**).

Operadores A tabela 12 mostra os operadores para *arrays*. Considere que `a` e `b` são dois *arrays*.

Operador	Associatividade	Precedência	Descrição
==	Esquerda	1	`a` é igual a `b`?
!=	Esquerda	1	`a` é diferente de `b`?
+	Esquerda	2	concatenação de `a` e `b`

Tabela 12: Operadores de *arrays*

Coerções A tabela 13 mostra as possíveis coerções de *arrays*.

Tipo	Descrição
string	Retorna uma **string** com a representação em **string** de cada elemento do *array*, separados por vírgu

Tabela 13: Coerções de *arrays*

Para *arrays* e **strings**, é possível acessar/alterar o i-ésimo elemento através de `[<indice>]`.

1.3 Instruções

1.3.1 Estrutura condicional

Estruturas condicionais são sentenças que realizam desvios no fluxo do programa dependendo do valor de uma expressão booleana. A seguir são apresentadas as estruturas condicionais de uma via e de duas vias.

De uma via A listagem 3 mostra a estrutura condicional de uma via. Se `expressao` for `true`, o fluxo do programa segue para a sentença ou bloco após o `)`. Caso contrário, o fluxo é desviado para depois da sentença ou bloco seguinte.

Listing 3: Estrutura condicional de uma via

```
if (<expressao)
   <sentenca>
if (<expressao>) {
   <lista de sentencas>
}
```

De duas vias A listagem 4 mostra a estrutura condicional de duas vias. A diferença para a estrutura de uma via é a palavra `else` após o bloco ou sentença da 1ª via. De maneira semelhante, o bloco ou sentença que aparece diretamente após o `else` só será executado(a) se a expressão da 1ª via for `false`.

Listing 4: Estrutura condicional de uma via

```
if (<expresso>) {
   ...
}
else if (<expresso>) { // Como a sentenca if vem
   red↪ depois do else, o else mais abaixo
   red↪ corresponde a esse if
   ...
}
else {
   ...
```

```
}
```

1.3.2 Estrutura iterativa

As estruturas iterativas repetem um bloco ou sentença. Há dois tipos de desvio incondicional dentro de uma estrutura dessa. O comando **break** desvia para depois do laço mais próximo e o comando **skip** desvia do ponto atual no bloco para o teste e consequentemente a próxima iteração. Nos dois casos mostrados a seguir a avaliação dos laços é pré-teste.

Listing 5: Estrutura iterativa com controle lógico

com controle lógico

```
while(<expresso>) {
    ...
}
```

por contador A iteração por contador é mostrada na listagem **??**. Com ela é possível iterar sobre intervalos (*strings*, *arrays* e intervalos numéricos). `tipo do contador` representa o tipo de cada elemento do conjunto (`char` para *strings* e tipo correspondente do *array* ou intervalo numérico) e `nome do contador` representa o nome que poderá ser usado pelo programador para se referir ao contador. Para declarar um intervalo numérico é necessário no mínimo uma expressão do tipo correspondente. Nesse caso entende-se que o valor inicial é 0 e o incremento igual a 1. A sintaxe utilizada é demonstrada na listagem6. Também é possível especificar um valor inicial e um valor de incremento. O valor de incremento é deduzido a partir da diferença entre o primeiro e segundo valores. Caso o segundo valor seja omitido, pressupõe-se que o valor de incremento é 1. Os tipos permitidos nesse tipo de construção são: `char`, `int` e `float`.

Listing 6: Estrutura iterativa por contador

```
for(<tipo do contador> <nome do contador> in <
    red↪ intervalo>)
    <sentenca>

for(<tipo do contador> <nome do contador> in <
    red↪ intervalo>) {
    <lista de sentencas>
}

// Ex.:
for(int i in [1, 2, 3]) {
    ...
}
```

```
for(char c in "Ola mundo") {
    ...
}

// Notacao de intervalo para tipos numericos
// [<inicio>, ..., <fim>) de <inicio> ate <fim> - 1
// [<inicio>, <segundo>, ..., <fim>) de inicio ate fim
    red↪  - 1, de g em g, onde g eh igual a (<segundo>
    red↪  - <inicio>)
for (int i in [1, ..., 20)) { // 1, 2, 3, ..., 18 e 19
    ...
}
for (int i in [1, 5, ..., 20] { // 1, 5, 10
    ...
}
```

1.3.3 Entrada

A entrada é feita através do comando **scan**. Ele recebe uma lista com as variáveis que receberão os valores lidos, na ordem em que são passados. Opcionalmente é possível passar uma **string** como formato para ser lido imediatamente à esquerda.

Listing 7: Comando `scan`

```
int a;
string s;

scan(a, "%[^\n]" s);
```

1.3.4 Saída

A saída é feita através do comando **print**. Ele recebe uma lista com as expressões que serão impressas, na ordem em que são passadas. Opcionalmente é possível passar uma **string** como formato para ser impresso imediatamente à esquerda.

Listing 8: Comando `print`

```
int a = 10;
string s = "agora vai";

print(a, "%20" s);
```

1.3.5 Atribuição

A atribuição (=) é um operador com precedência 0 e associatividade à direita. O operando à esquerda deve ser o identificador de uma variável e à direita uma expressão que resulte num tipo compatível com o da variável.

1.4 Subprogramas

É possível declarar subprogramas localmente ou globalmente. A passagem de parâmetros para subprogramas é feita em modo de entrada. Os parâmetros são passados por valor. A sintaxe para declaração de subprogramas foi mostrada na seção1.1.2. Procedimentos são declarados da mesma forma que funções, exceto que no lugar do tipo de retorno possuem a palavra `void`.

1.5 Exemplos de programas

1.5.1 Olá mundo

```
// Ola mundo
//
// Exibe a mensagem "Ola mundo!"
//
// Criado por Valerio Nogueira em 31/08/2021

int main(string[] args) {
   print("Ola mundo!");

   return 0;
}
```

1.5.2 Série de Fibonacci

```
// Serie de Fibonacci
//
// Dado um valor limite lista todos os numeros
// da serie de Fibonacci menores que ele
// separados por virgula
//
// Criado por Valerio Nogueira em 31/08/2021

void fibonacci(int limite) {
   int a = 1;
   int b = 1;
   int c = a + b;
```

```
   print(a, ", ", b);
   while (c < limite) {
      print(c, ", ");
      a = b;
      b = c;
      c = a + b;
   }
}

int main(string[] args) {
   int limite;
   scan(limite);

   fibonacci(limite);

   return 0;
}
```

1.5.3 Shell sort

```
// Shell sort
//
// Ordena um array de inteiros
//
// Criado por Valerio Nogueira em 01/08/2021 (Adaptado de
   red↪  https://panda.ime.usp.br/panda/static/
   red↪ pythonds_pt/05-OrdenacaoBusca/OShellSort.html)

int[] shell_sort(int n, int[] values) {
   int c = n / 2;
   while(c > 0) {
      for(int start in [0 ... c]) {
         gap_insertion_sort(n, values, start, c);

         for(int i in [start + c ... n] by c) {
            int current_value = values[i];
            int position = i;

            while (position >= c && values[position - c]
               red↪ > current_values) {
               values[position] = values[position - c];
               position = position - c;
            }
         }
```

```
         c = c/2;
      }
   }

   return values;
}

int main(string[] args) {
   int n;
   int[] values;

   scan(n);
   for(int i in [0 ... n]) {
      scan(values[i]);
   }

   values = shell_sort(n, values);

   print(values);

   return 0;
}
```

2 Especificação dos tokens

Os analisadores léxico e sintático serão implementados na linguagem C++. A máquina de estados do analisador léxico (único implementado no momento) foi feita utilizando a ferramenta *Finite State Machine Designer* (https://www.cs.unc.edu/ otternes/comp455/fsm_designer/) e através de um *script Python* um arquivo contendo a tabela de transições é gerado e utilizado no programa principal. A tabela 14 mostra os nomes simbólicos dos *tokens* acompanhados de seus valores numéricos. A figura 1 mostra a máquina de estados construída. A ferramenta possibilita a exportação no formato *json*, que é utilizado para a geração dos arquivos.

Tabela 14: Enumeração das categorias dos *tokens*.

Nome simbólico	Valor numérico
Assignment	0
Bool	1
Break	2
Character	3
CloseBraces	4
CloseBrackets	5

CloseParenthesis	6
Comma	7
Dot	8
Ellipsis	9
Else	10
EndOfLine	11
Float	12
For	13
Identifier	14
If	15
In	16
Integer	17
OpAdd	18
OpAnd	19
OpConcatenate	20
OpDiv	21
OpEq	22
OpGt	23
OpGte	24
OpLt	25
OpLte	26
OpMod	27
OpMul	28
OpNeq	29
OpNot	30
OpOr	31
OpSub	32
OpenBraces	33
OpenBrackets	34
OpenParenthesis	35
Print	36
Return	37
Scan	38
SemiColon	39
Skip	40
String	41
TypeBool	42
TypeChar	43
TypeFloat	44
TypeInt	45
TypeString	46
Void	47
While	48

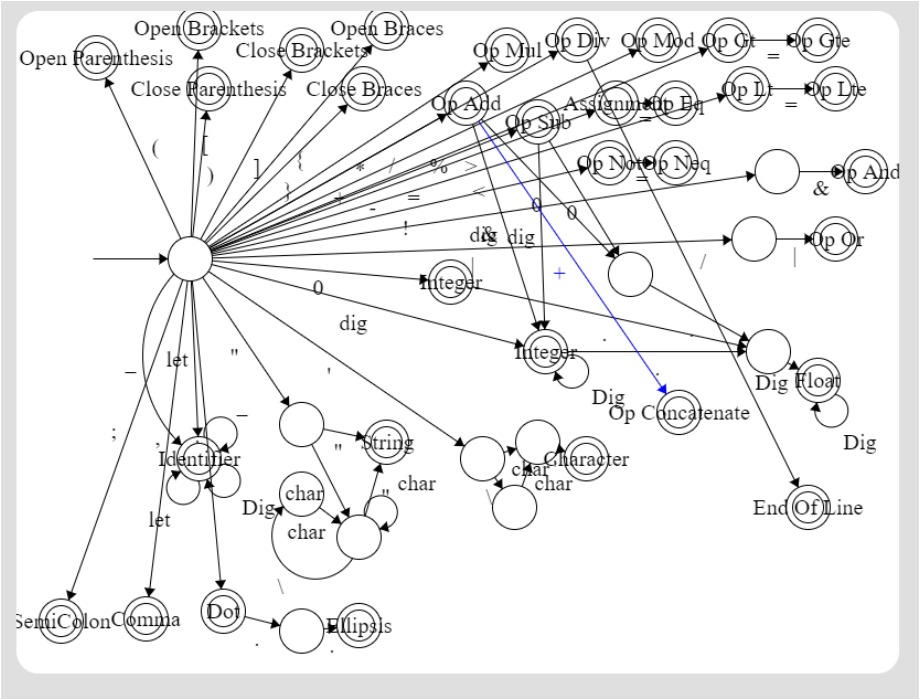

Figura 1: Máquina de estados do autômato

Padrão	Expressão regular
dig	`'[1-9]'`
Dig	`'[0-9]'`
let	`'[a-zA-Z]'`
char	`[\n-~]`

Tabela 15: Expressões regulares auxiliares

Tabela 16: Expressões regulares das categorias dos *tokens*.

Categoria	Expressão regular
Arrow	`->`
Assignment	`=`
Bool	`true|false`
Break	`break`
Character	`'{char}'`
CloseBraces	`}`
CloseBrackets	`]`
CloseParentheses	`)`

Comma	,
Dot	.
Ellipsis	...
Else	else
EndOfLine	//
Float	[+-]?{Dig}+.{Dig}+
For	for
Func	func
Identifier	[_{let}][{let}{Dig}_]*
If	if
In	in
Integer	[+-]?0|({dig}{Dig}*)
OpAdd	+
OpAnd	&&
OpConcatenate	++
OpDiv	/
OpEq	==
OpGt	>
OpGte	>=
OpLt	<
OpLte	<=
OpMod	%
OpMul	*
OpNeq	!=
OpNot	!
OpOr	||
OpSub	-
OpenBraces	{
OpenBrackets	[
OpenParentheses	(
Print	print
Return	return
Scan	scan
SemiColon	;
Skip	skip
String	\"({char}|(\\{char}))*\"
TypeBool	bool
TypeChar	char
TypeFloat	float
TypeInt	int
TypeString	string
Void	void
While	while

3 Analisador léxico

3.1 Resultados nos exemplos

3.1.1 Olá mundo

Listing 9: Resultado do analisador no arquivo ola_mundo.agrvai

```
`// Ola mundo`
`//`
`// Exibe a mensagem "Ola mundo!"`
`//`
`// Criado por Valerio Nogueira em 31/08/2021`
``
`int main(string[] args) {`
          [0007, 0001] (0045, TypeInt) {int}
          [0007, 0005] (0014, Identifier) {main}
          [0007, 0009] (0035, OpenParenthesis) {(}
          [0007, 0010] (0046, TypeString) {string}
          [0007, 0016] (0034, OpenBrackets) {[}
          [0007, 0017] (0005, CloseBrackets) {]}
          [0007, 0019] (0014, Identifier) {args}
          [0007, 0023] (0006, CloseParenthesis) {)}
          [0007, 0025] (0033, OpenBraces) {{}
` print("Ola mundo!");`
          [0008, 0005] (0036, Print) {print}
          [0008, 0010] (0035, OpenParenthesis) {(}
          [0008, 0011] (0041, String) {"Ola mundo!"}
          [0008, 0023] (0006, CloseParenthesis) {)}
          [0008, 0024] (0039, SemiColon) {;}
``
` return 0;`
          [0010, 0005] (0037, Return) {return}
          [0010, 0012] (0017, Integer) {0}
          [0010, 0013] (0039, SemiColon) {;}
`}`
          [0011, 0001] (0004, CloseBraces) {}}
```

3.1.2 Olá mundo

Listing 10: Resultado do analisador no arquivo serie_fibonacci.agrvai

```
`// Serie de Fibonacci`
`//`
`// Dado um valor limite lista todos os numeros`
`// da serie de Fibonacci menores que ele`
```

```
`// separados por virgula`
`//`
`// Criado por Valerio Nogueira em 31/08/2021`
``
`void fibonacci(int limite) {`
          [0009, 0001] (0047, Void) {void}
          [0009, 0006] (0014, Identifier) {fibonacci}
          [0009, 0015] (0035, OpenParenthesis) {(}
          [0009, 0016] (0045, TypeInt) {int}
          [0009, 0020] (0014, Identifier) {limite}
          [0009, 0026] (0006, CloseParenthesis) {)}
          [0009, 0028] (0033, OpenBraces) {{}
` int a = 1;`
          [0010, 0005] (0045, TypeInt) {int}
          [0010, 0009] (0014, Identifier) {a}
          [0010, 0011] (0000, Assignment) {=}
          [0010, 0013] (0017, Integer) {1}
          [0010, 0014] (0039, SemiColon) {;}
` int b = 1;`
          [0011, 0005] (0045, TypeInt) {int}
          [0011, 0009] (0014, Identifier) {b}
          [0011, 0011] (0000, Assignment) {=}
          [0011, 0013] (0017, Integer) {1}
          [0011, 0014] (0039, SemiColon) {;}
` int c = a + b;`
          [0012, 0005] (0045, TypeInt) {int}
          [0012, 0009] (0014, Identifier) {c}
          [0012, 0011] (0000, Assignment) {=}
          [0012, 0013] (0014, Identifier) {a}
          [0012, 0015] (0018, OpAdd) {+}
          [0012, 0017] (0014, Identifier) {b}
          [0012, 0018] (0039, SemiColon) {;}
``
` print(a, ",␣", b);`
          [0014, 0005] (0036, Print) {print}
          [0014, 0010] (0035, OpenParenthesis) {(}
          [0014, 0011] (0014, Identifier) {a}
          [0014, 0012] (0007, Comma) {,}
          [0014, 0014] (0041, String) {",␣"}
          [0014, 0018] (0007, Comma) {,}
          [0014, 0020] (0014, Identifier) {b}
          [0014, 0021] (0006, CloseParenthesis) {)}
          [0014, 0022] (0039, SemiColon) {;}
` while (c < limite) {`
          [0015, 0005] (0048, While) {while}
          [0015, 0011] (0035, OpenParenthesis) {(}
```

```
          [0015, 0012] (0014, Identifier) {c}
          [0015, 0014] (0025, OpLt) {<}
          [0015, 0016] (0014, Identifier) {limite}
          [0015, 0022] (0006, CloseParenthesis) {)}
          [0015, 0024] (0033, OpenBraces) {{}
` print(c, ",␣");`
          [0016, 0009] (0036, Print) {print}
          [0016, 0014] (0035, OpenParenthesis) {(}
          [0016, 0015] (0014, Identifier) {c}
          [0016, 0016] (0007, Comma) {,}
          [0016, 0018] (0041, String) {",␣"}
          [0016, 0022] (0006, CloseParenthesis) {)}
          [0016, 0023] (0039, SemiColon) {;}
` a = b;`
          [0017, 0009] (0014, Identifier) {a}
          [0017, 0011] (0000, Assignment) {=}
          [0017, 0013] (0014, Identifier) {b}
          [0017, 0014] (0039, SemiColon) {;}
` b = c;`
          [0018, 0009] (0014, Identifier) {b}
          [0018, 0011] (0000, Assignment) {=}
          [0018, 0013] (0014, Identifier) {c}
          [0018, 0014] (0039, SemiColon) {;}
` c = a + b;`
          [0019, 0009] (0014, Identifier) {c}
          [0019, 0011] (0000, Assignment) {=}
          [0019, 0013] (0014, Identifier) {a}
          [0019, 0015] (0018, OpAdd) {+}
          [0019, 0017] (0014, Identifier) {b}
          [0019, 0018] (0039, SemiColon) {;}
` }`
          [0020, 0005] (0004, CloseBraces) {}}
`}`
          [0021, 0001] (0004, CloseBraces) {}}
``
`int main(string[] args) {`
          [0023, 0001] (0045, TypeInt) {int}
          [0023, 0005] (0014, Identifier) {main}
          [0023, 0009] (0035, OpenParenthesis) {(}
          [0023, 0010] (0046, TypeString) {string}
          [0023, 0016] (0034, OpenBrackets) {[}
          [0023, 0017] (0005, CloseBrackets) {]}
          [0023, 0019] (0014, Identifier) {args}
          [0023, 0023] (0006, CloseParenthesis) {)}
          [0023, 0025] (0033, OpenBraces) {{}
` int limite;`
```

```
          [0024, 0005] (0045, TypeInt) {int}
          [0024, 0009] (0014, Identifier) {limite}
          [0024, 0015] (0039, SemiColon) {;}
` scan(limite);`
          [0025, 0005] (0038, Scan) {scan}
          [0025, 0009] (0035, OpenParenthesis) {(}
          [0025, 0010] (0014, Identifier) {limite}
          [0025, 0016] (0006, CloseParenthesis) {)}
          [0025, 0017] (0039, SemiColon) {;}
``
` fibonacci(limite);`
          [0027, 0005] (0014, Identifier) {fibonacci}
          [0027, 0014] (0035, OpenParenthesis) {(}
          [0027, 0015] (0014, Identifier) {limite}
          [0027, 0021] (0006, CloseParenthesis) {)}
          [0027, 0022] (0039, SemiColon) {;}
``
` return 0;`
          [0029, 0005] (0037, Return) {return}
          [0029, 0012] (0017, Integer) {0}
          [0029, 0013] (0039, SemiColon) {;}
`}`
          [0030, 0001] (0004, CloseBraces) {}}
```

3.1.3 Shell sort

Listing 11: Resultado do analisador no arquivo shell_sort.agrvai

```
`// Shell sort`
`//`
`// Ordena um array de inteiros`
`//`
`// Criado por Valerio Nogueira em 01/08/2021 (Adaptado
   red↪ de https://panda.ime.usp.br/panda/static/
   red↪ pythonds_pt/05-OrdenacaoBusca/OShellSort.html)`
``
`int[] shell_sort(int n, int[] values) {`
          [0007, 0001] (0045, TypeInt) {int}
          [0007, 0004] (0034, OpenBrackets) {[}
          [0007, 0005] (0005, CloseBrackets) {]}
          [0007, 0007] (0014, Identifier) {shell_sort}
          [0007, 0017] (0035, OpenParenthesis) {(}
          [0007, 0018] (0045, TypeInt) {int}
          [0007, 0022] (0014, Identifier) {n}
          [0007, 0023] (0007, Comma) {,}
```

```
          [0007, 0025] (0045, TypeInt) {int}
          [0007, 0028] (0034, OpenBrackets) {[}
          [0007, 0029] (0005, CloseBrackets) {]}
          [0007, 0031] (0014, Identifier) {values}
          [0007, 0037] (0006, CloseParenthesis) {)}
          [0007, 0039] (0033, OpenBraces) {{}
` int c = n / 2;`
          [0008, 0005] (0045, TypeInt) {int}
          [0008, 0009] (0014, Identifier) {c}
          [0008, 0011] (0000, Assignment) {=}
          [0008, 0013] (0014, Identifier) {n}
          [0008, 0015] (0021, OpDiv) {/}
          [0008, 0017] (0017, Integer) {2}
          [0008, 0018] (0039, SemiColon) {;}
` while(c > 0) {`
          [0009, 0005] (0048, While) {while}
          [0009, 0010] (0035, OpenParenthesis) {(}
          [0009, 0011] (0014, Identifier) {c}
          [0009, 0013] (0023, OpGt) {>}
          [0009, 0015] (0017, Integer) {0}
          [0009, 0016] (0006, CloseParenthesis) {)}
          [0009, 0018] (0033, OpenBraces) {{}
` for(int start in [0 ... c]) {`
          [0010, 0009] (0013, For) {for}
          [0010, 0012] (0035, OpenParenthesis) {(}
          [0010, 0013] (0045, TypeInt) {int}
          [0010, 0017] (0014, Identifier) {start}
          [0010, 0023] (0016, In) {in}
          [0010, 0026] (0034, OpenBrackets) {[}
          [0010, 0027] (0017, Integer) {0}
          [0010, 0029] (0009, Ellipsis) {...}
          [0010, 0033] (0014, Identifier) {c}
          [0010, 0034] (0005, CloseBrackets) {]}
          [0010, 0035] (0006, CloseParenthesis) {)}
          [0010, 0037] (0033, OpenBraces) {{}
` gap_insertion_sort(n, values, start, c);`
          [0011, 0013] (0014, Identifier) {
             red↪ gap_insertion_sort}
          [0011, 0031] (0035, OpenParenthesis) {(}
          [0011, 0032] (0014, Identifier) {n}
          [0011, 0033] (0007, Comma) {,}
          [0011, 0035] (0014, Identifier) {values}
          [0011, 0041] (0007, Comma) {,}
          [0011, 0043] (0014, Identifier) {start}
          [0011, 0048] (0007, Comma) {,}
          [0011, 0050] (0014, Identifier) {c}
```

```
          [0011, 0051] (0006, CloseParenthesis) {)}
          [0011, 0052] (0039, SemiColon) {;}
``
` for(int i in [start + c ... n] by c) {`
          [0013, 0013] (0013, For) {for}
          [0013, 0016] (0035, OpenParenthesis) {(}
          [0013, 0017] (0045, TypeInt) {int}
          [0013, 0021] (0014, Identifier) {i}
          [0013, 0023] (0016, In) {in}
          [0013, 0026] (0034, OpenBrackets) {[}
          [0013, 0027] (0014, Identifier) {start}
          [0013, 0033] (0018, OpAdd) {+}
          [0013, 0035] (0014, Identifier) {c}
          [0013, 0037] (0009, Ellipsis) {...}
          [0013, 0041] (0014, Identifier) {n}
          [0013, 0042] (0005, CloseBrackets) {]}
          [0013, 0044] (0014, Identifier) {by}
          [0013, 0047] (0014, Identifier) {c}
          [0013, 0048] (0006, CloseParenthesis) {)}
          [0013, 0050] (0033, OpenBraces) {{}
` int current_value = values[i];`
          [0014, 0017] (0045, TypeInt) {int}
          [0014, 0021] (0014, Identifier) {current_value}
          [0014, 0035] (0000, Assignment) {=}
          [0014, 0037] (0014, Identifier) {values}
          [0014, 0043] (0034, OpenBrackets) {[}
          [0014, 0044] (0014, Identifier) {i}
          [0014, 0045] (0005, CloseBrackets) {]}
          [0014, 0046] (0039, SemiColon) {;}
` int position = i;`
          [0015, 0017] (0045, TypeInt) {int}
          [0015, 0021] (0014, Identifier) {position}
          [0015, 0030] (0000, Assignment) {=}
          [0015, 0032] (0014, Identifier) {i}
          [0015, 0033] (0039, SemiColon) {;}
` `
` while (position >= c && values[position - c] >
   red↪ current_values) {`
          [0017, 0017] (0048, While) {while}
          [0017, 0023] (0035, OpenParenthesis) {(}
          [0017, 0024] (0014, Identifier) {position}
          [0017, 0033] (0024, OpGte) {>=}
          [0017, 0036] (0014, Identifier) {c}
          [0017, 0038] (0019, OpAnd) {&&}
          [0017, 0041] (0014, Identifier) {values}
          [0017, 0047] (0034, OpenBrackets) {[}
```

```
          [0017, 0048] (0014, Identifier) {position}
          [0017, 0057] (0032, OpSub) {-}
          [0017, 0059] (0014, Identifier) {c}
          [0017, 0060] (0005, CloseBrackets) {]}
          [0017, 0062] (0023, OpGt) {>}
          [0017, 0064] (0014, Identifier) {current_values
             red↪ }
          [0017, 0078] (0006, CloseParenthesis) {)}
          [0017, 0080] (0033, OpenBraces) {{}
` values[position] = values[position - c];`
          [0018, 0021] (0014, Identifier) {values}
          [0018, 0027] (0034, OpenBrackets) {[}
          [0018, 0028] (0014, Identifier) {position}
          [0018, 0036] (0005, CloseBrackets) {]}
          [0018, 0038] (0000, Assignment) {=}
          [0018, 0040] (0014, Identifier) {values}
          [0018, 0046] (0034, OpenBrackets) {[}
          [0018, 0047] (0014, Identifier) {position}
          [0018, 0056] (0032, OpSub) {-}
          [0018, 0058] (0014, Identifier) {c}
          [0018, 0059] (0005, CloseBrackets) {]}
          [0018, 0060] (0039, SemiColon) {;}
` position = position - c;`
          [0019, 0021] (0014, Identifier) {position}
          [0019, 0030] (0000, Assignment) {=}
          [0019, 0032] (0014, Identifier) {position}
          [0019, 0041] (0032, OpSub) {-}
          [0019, 0043] (0014, Identifier) {c}
          [0019, 0044] (0039, SemiColon) {;}
` }`
          [0020, 0017] (0004, CloseBraces) {}}
` }`
          [0021, 0013] (0004, CloseBraces) {}}
``
` c = c/2;`
          [0023, 0013] (0014, Identifier) {c}
          [0023, 0015] (0000, Assignment) {=}
          [0023, 0017] (0014, Identifier) {c}
          [0023, 0018] (0021, OpDiv) {/}
          [0023, 0019] (0017, Integer) {2}
          [0023, 0020] (0039, SemiColon) {;}
` }`
          [0024, 0009] (0004, CloseBraces) {}}
` }`
          [0025, 0005] (0004, CloseBraces) {}}
``
```

```
` return values;`
          [0027, 0005] (0037, Return) {return}
          [0027, 0012] (0014, Identifier) {values}
          [0027, 0018] (0039, SemiColon) {;}
`}`
          [0028, 0001] (0004, CloseBraces) {}}
``
`int main(string[] args) {`
          [0030, 0001] (0045, TypeInt) {int}
          [0030, 0005] (0014, Identifier) {main}
          [0030, 0009] (0035, OpenParenthesis) {(}
          [0030, 0010] (0046, TypeString) {string}
          [0030, 0016] (0034, OpenBrackets) {[}
          [0030, 0017] (0005, CloseBrackets) {]}
          [0030, 0019] (0014, Identifier) {args}
          [0030, 0023] (0006, CloseParenthesis) {)}
          [0030, 0025] (0033, OpenBraces) {{}
` int n;`
          [0031, 0005] (0045, TypeInt) {int}
          [0031, 0009] (0014, Identifier) {n}
          [0031, 0010] (0039, SemiColon) {;}
` int[] values;`
          [0032, 0005] (0045, TypeInt) {int}
          [0032, 0008] (0034, OpenBrackets) {[}
          [0032, 0009] (0005, CloseBrackets) {]}
          [0032, 0011] (0014, Identifier) {values}
          [0032, 0017] (0039, SemiColon) {;}
``
` scan(n);`
          [0034, 0005] (0038, Scan) {scan}
          [0034, 0009] (0035, OpenParenthesis) {(}
          [0034, 0010] (0014, Identifier) {n}
          [0034, 0011] (0006, CloseParenthesis) {)}
          [0034, 0012] (0039, SemiColon) {;}
` for(int i in [0 ... n]) {`
          [0035, 0005] (0013, For) {for}
          [0035, 0008] (0035, OpenParenthesis) {(}
          [0035, 0009] (0045, TypeInt) {int}
          [0035, 0013] (0014, Identifier) {i}
          [0035, 0015] (0016, In) {in}
          [0035, 0018] (0034, OpenBrackets) {[}
          [0035, 0019] (0017, Integer) {0}
          [0035, 0021] (0009, Ellipsis) {...}
          [0035, 0025] (0014, Identifier) {n}
          [0035, 0026] (0005, CloseBrackets) {]}
          [0035, 0027] (0006, CloseParenthesis) {)}
```

```
          [0035, 0029] (0033, OpenBraces) {{}
` scan(values[i]);`
          [0036, 0009] (0038, Scan) {scan}
          [0036, 0013] (0035, OpenParenthesis) {(}
          [0036, 0014] (0014, Identifier) {values}
          [0036, 0020] (0034, OpenBrackets) {[}
          [0036, 0021] (0014, Identifier) {i}
          [0036, 0022] (0005, CloseBrackets) {]}
          [0036, 0023] (0006, CloseParenthesis) {)}
          [0036, 0024] (0039, SemiColon) {;}
` }`
          [0037, 0005] (0004, CloseBraces) {}}
``
` values = shell_sort(n, values);`
          [0039, 0005] (0014, Identifier) {values}
          [0039, 0012] (0000, Assignment) {=}
          [0039, 0014] (0014, Identifier) {shell_sort}
          [0039, 0024] (0035, OpenParenthesis) {(}
          [0039, 0025] (0014, Identifier) {n}
          [0039, 0026] (0007, Comma) {,}
          [0039, 0028] (0014, Identifier) {values}
          [0039, 0034] (0006, CloseParenthesis) {)}
          [0039, 0035] (0039, SemiColon) {;}
``
` print(values);`
          [0041, 0005] (0036, Print) {print}
          [0041, 0010] (0035, OpenParenthesis) {(}
          [0041, 0011] (0014, Identifier) {values}
          [0041, 0017] (0006, CloseParenthesis) {)}
          [0041, 0018] (0039, SemiColon) {;}
``
` return 0;`
          [0043, 0005] (0037, Return) {return}
          [0043, 0012] (0017, Integer) {0}
          [0043, 0013] (0039, SemiColon) {;}
`}`
          [0044, 0001] (0004, CloseBraces) {}}
```