

Compiladores

Projeto e implementação de um compilador - parte 5

Rubem Kalebe Santos

2 de dezembro de 2015

1. Introdução

Neste relatório serão apresentados os conceitos que fundamentam o compilador implementado e como tais conceitos foram aplicados. Além disso, serão apresentados os resultados obtidos.

Nos anexos deste relatório é possível encontrar as especificações da linguagem. No Apêndice A é apresentada a descrição da linguagem, no B, as palavras reservadas, no C, os operadores válidos e no D, a gramática da linguagem.

2. Tradução dirigida pela sintaxe

Existem duas notações para associar regras semânticas às produções: definições dirigidas pela sintaxe e esquemas de tradução. As definições dirigidas pela sintaxe são especificações de alto nível para as traduções, sendo que escondem muitos detalhes de implementação e liberam o programador de especificar exatamente a ordem na qual as traduções têm lugar. Os esquemas de tradução indicam a ordem na qual as regras semânticas são avaliadas e assim permitem que alguns detalhes de implementação sejam evidenciados.

Uma definição dirigida pela sintaxe é uma generalização de uma gramática livre de contexto na qual cada símbolo gramatical possui um conjunto associado de atributos, partitionados em dois subconjuntos, chamados de atributos sintetizados e atributos herdados daquele símbolo gramatical. Um atributo pode representar qualquer tipo de informação como uma cadeia, um número, um tipo, uma localização de memória, etc. O valor de um atributo em um nó da árvore gramatical é definido por uma regra semântica associada à produção daquele nó.

Numa definição dirigida pela sintaxe, cada produção gramatical $A \rightarrow \alpha$ tem associada a si um conjunto de regras semânticas da forma $b := f(c_1, c_2, \dots, c_k)$, onde f é uma função e vigora uma das duas situações seguintes, mas não ambas:

- a) b é um atributo sintetizado de A e c_1, c_2, \dots, c_k são atributos pertencentes aos símbolos gramaticais da produção, ou
- b) b é um atributo herdado, pertencente a um dos símbolos gramaticais do lado direito da produção, e c_1, c_2, \dots, c_k são atributos pertencentes aos símbolos gramaticais da produção.

Tanto para atributos herdados quanto para atributos sintetizados, o atributo b depende dos atributos c_1, c_2, \dots, c_k . As funções nas regras semânticas serão frequentemente escritas como expressões e chamadas de procedimentos ou fragmentos de programas.

Uma gramática de atributos consiste em uma definição dirigida pela sintaxe, na qual as funções das regras semânticas não têm efeitos colaterais, ou seja, não alteram seus parâmetros ou variável não local.

Um atributo é dito sintetizado se seu valor num nó da árvore gramatical é determinado a partir dos valores dos atributos dos filhos daquele nó. Os atributos sintetizados possuem a propriedade de que podem ser avaliados durante um único percurso *bottom-up* da

árvore gramatical. Uma definição dirigida pela sintaxe que use exclusivamente atributos sintetizados é dita uma definição S-atribuída.

Um atributo herdado é aquele cujo valor a um nó de uma árvore gramatical é definido em termos do pai e/ou irmãos daquele nó. Os atributos herdados são convenientes para expressar a dependência de uma construção de linguagem de programação no contexto em que a mesma figurar.

As regras semânticas definem dependências entre os atributos dos símbolos. Essas dependências podem-se representar graficamente através de um grafo (de dependências) orientado, onde os vértices são os atributos e as arestas as dependências. Sendo assim, a aresta $\langle X.a, Y.b \rangle$ indica que $Y.b$ depende de $X.a$.

3. Implementação

Nesta seção serão discutidas as questões relativas a aplicação dos conceitos teóricos vistos anteriormente no compilador projetado.

3.1 Tabela de símbolos

Foram adotadas três tabelas de símbolos no compilador: uma tabela para tipos, uma tabela para variáveis e uma tabela para subprogramas.

A tabela para tipos é única e persiste durante toda a execução do programa e armazena, além dos tipos primitivos, todos os tipos declarados pelo usuário. A tabela de subprograma é semelhante a tabela para tipos, quanto a duração e quantidade. Ela armazena entradas de subprogramas, onde cada entrada é definida pelo identificador do subprograma e armazena o tipo de retorno e os tipos dos parâmetros.

A tabela de variáveis se comporta como uma pilha durante o processamento de escopos aninhados. Por isso, decidiu-se por utilizar uma tabela de variáveis para cada escopo, onde constrói-se uma nova tabela para cada escopo e vincula-se essas tabelas de escopo mais interno com o mais externo. Dessa forma, a operação para verificar a declaração de uma variável varre tabelas mais externas se não conseguir encontrar um nome na tabela corrente. Citamos dois fatores que motivaram essa escolha:

1. Por ser mais próxima ao raciocínio humano;
2. Utilizando esta abordagem, abandonar o escopo requer menor esforço, pois as declarações não precisam ser removidas uma a uma.

As tabelas de subprogramas e de variáveis são implementadas utilizando-se árvores binárias de busca através do contêiner `map` da biblioteca padrão do C++. As operações de inserção e busca ocorrem em tempo logarítmico e os *maps* associam um lexema a uma entrada, de subprograma ou de variável. A diferença é o que a entrada armazena: um subprograma armazena tipo de retorno e uma lista dos tipos dos parâmetros e uma variável armazena apenas o seu tipo. Um lexema é representado simplesmente por uma *string*.

Essa implementação é bastante atual e de fácil entendimento, o que facilita futuras modificações.

3.2 Atributos herdados e sintetizados

Os atributos herdados e sintetizados são processados da maneira como descritos anteriormente. Em especial, para o processamento dos atributos sintetizados foi utilizado um tipo de dado com dois campos: tipo e lexema. No campo tipo é armazenado o tipo associado ao lexema que é processado. Com isso, alguns não-terminais são declarados serem desse tipo e retornam variáveis configuradas de acordo com seu contexto.

Por exemplo, ao ser processado o lexema 2 ele é associado ao tipo `int`. Quando opera-se `2 + 3.5` o lexema 3.5 é associado ao tipo `double` e, por regras da linguagem, a soma em questão é do tipo `double`. Essa expressão seria válida se fosse associada a uma declaração de variável inteira, com base nas regras de coerção da linguagem, da qual a expressão de ponto flutuante seria convertida em um valor inteiro. Repare que o valor associado ao identificador da variável é um atributo hergado e o tipo da expressão numérica é um atributo sintetizado.

4. Resultados computacionais

O compilador foi implementado de forma a funcionar de maneira idêntica a outros compiladores. Para executá-lo basta chamá-lo e passar um código fonte como parâmetro. O compilador irá processar tal fonte e, se ele não conter erros, será enviada a versão traduzida do fonte escrito em Samekh para C++ simplificado, que passará então pelo compilador g++. Ele é responsável por gerar o executável de nome `a.out`.

Nas figuras 1, 2, 3 e 4 são apresentados alguns *prints* da execução do compilador.

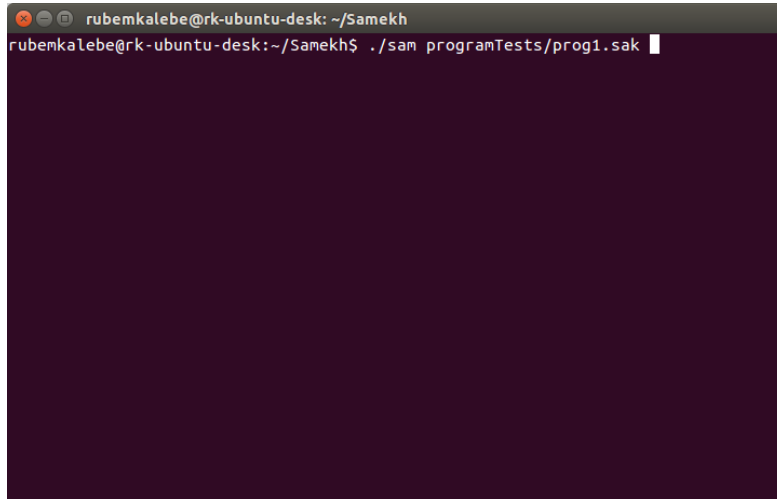
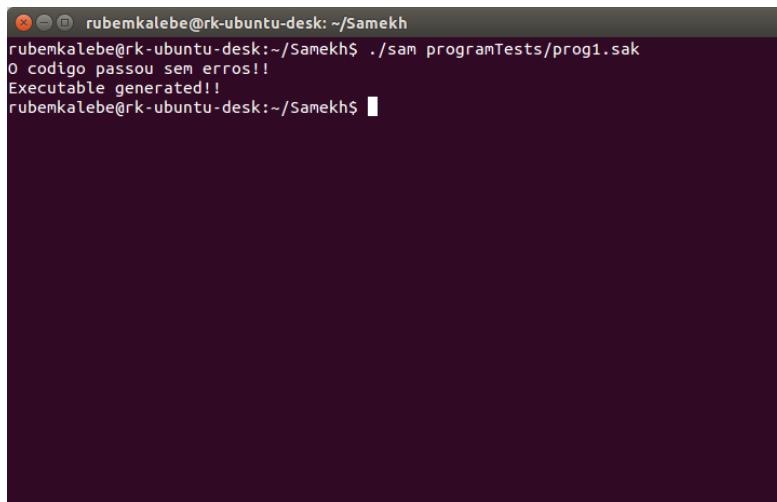
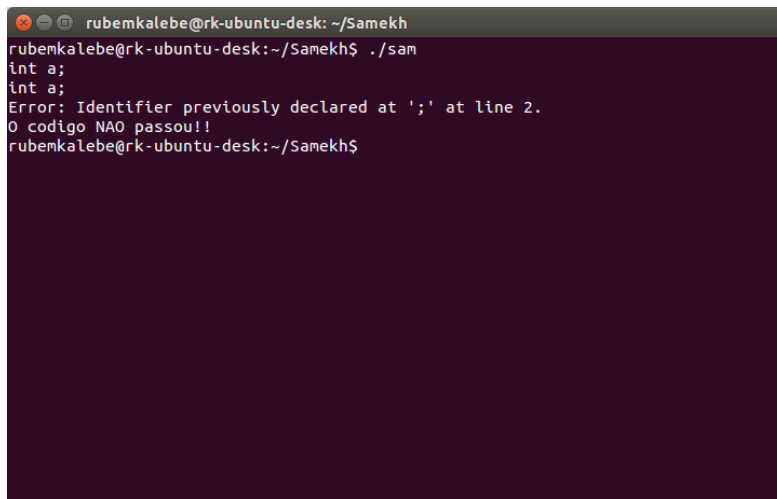
A screenshot of a terminal window with a dark background. The window title is 'rubemkalebe@rk-ubuntu-desk: ~/Samekh'. The prompt is 'rubemkalebe@rk-ubuntu-desk:~/Samekh\$'. The command entered is './sam programTests/prog1.sak'. The cursor is at the end of the command line.

Figura 1: Exemplo de chamado ao compilador.



```
rubemkalebe@rk-ubuntu-desk: ~/Samekh
rubemkalebe@rk-ubuntu-desk:~/Samekh$ ./sam programTests/prog1.sak
0 código passou sem erros!!
Executable generated!!
rubemkalebe@rk-ubuntu-desk:~/Samekh$
```

Figura 2: Exemplo de execução, com sucesso, do compilador.

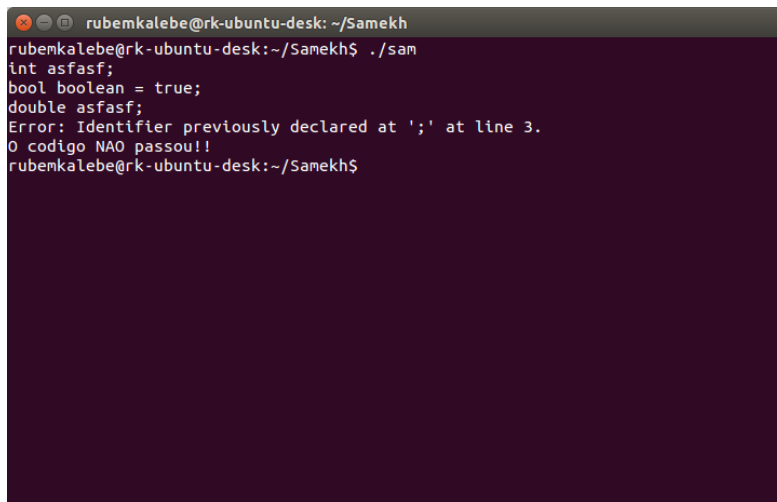


```
rubemkalebe@rk-ubuntu-desk: ~/Samekh
rubemkalebe@rk-ubuntu-desk:~/Samekh$ ./sam
int a;
int a;
Error: Identifier previously declared at ';' at line 2.
0 código NAO passou!!
rubemkalebe@rk-ubuntu-desk:~/Samekh$
```

Figura 3: Exemplo de erro de declaração duplicada.

Referências

- [ASU95] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compiladores: Princípios, técnicas e ferramentas. *LTC, Rio de Janeiro, Brasil*, 1995.

A terminal window with a dark purple background. The prompt is 'rubemkalebe@rk-ubuntu-desk: ~/Samekh'. The user enters './sam'. The output shows a compilation error: 'Error: Identifier previously declared at ';' at line 3.' followed by '0 codigo NAO passou!!'. The prompt returns to 'rubemkalebe@rk-ubuntu-desk:~/Samekh\$'.

```
rubemkalebe@rk-ubuntu-desk: ~/Samekh
rubemkalebe@rk-ubuntu-desk:~/Samekh$ ./sam
int asfasf;
bool boolean = true;
double asfasf;
Error: Identifier previously declared at ';' at line 3.
0 codigo NAO passou!!
rubemkalebe@rk-ubuntu-desk:~/Samekh$
```

Figura 4: Outro exemplo de erro de declaração duplicada.

Apêndice A – Descrição da linguagem

Nesta seção será discutida a estruturação da linguagem Samekh.

Características da linguagem

As características da linguagem para a qual será implementada o compilador são:

- Segue o paradigma imperativo;
- Possui um tipo genérico alocado dinamicamente;
- Blocos delimitados por palavras reservadas;
- Comandos parecidos com C/C++/Java, que são linguagens populares, o que facilita a migração de programadores de alguma dessas linguagens;
- Fortemente tipada;
- Linguagem voltada para desenvolvimento de sistemas.

Sintaxe da linguagem

A sintaxe da linguagem é definida usando-se uma Gramática Livre de Contexto e a notação utilizada é a BNF. A gramática da linguagem é apresentada no Apêndice D.

Estrutura da linguagem

Os próximos tópicos abordarão mais detalhes sobre a linguagem proposta.

- **Nomes**

As principais características quanto aos identificadores da linguagem são descritas a seguir:

- Um *id* não tem tamanho máximo definido;
- O primeiro caractere deve ser letra ou sinal de sublinha ('_'). Os demais caracteres podem ser letras, números ou sinal de sublinha, portanto, identificadores como 1peso ou @email estão errados;
- Os *ids* não devem coincidir com palavras reservadas da linguagem, por exemplo, a palavra *switch* será uma palavra reservada;
- A linguagem fará distinção entre letras maiúsculas e minúsculas (*case-sensitive*). Essa característica implica em uma variável `pesoPaciente` diferente de `PesoPaciente`.

- **Tipos primitivos**

Visando facilitar a escrita de programas, a nossa linguagem disponibilizará os seguintes tipos primitivos:

Nome	Bytes	Faixa de valores
bool	1	<i>true</i> ou <i>false</i>
int	4	-2.147.483.648 a 2.147.483.647
long	8	9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
double	8	2,2e-308 a 1,8e+308

Tabela 1: Tabela com informações sobre os tipos primitivos da linguagem

As variáveis primitivas serão alocadas estaticamente com a quantidade de bytes informada na tabela 1. Haverá também um tipo dinâmico chamado de *auto* (ao realizar operações com variáveis desse tipo, será necessário uma consulta de compatibilidade, que será feita com o operador `typeof`, para saber como realizar essa operação). Além disso também existe o tipo *string* que armazena uma cadeia de caracteres, sem tamanho pré-definido.

Os números de ponto flutuante são representados de acordo com o padrão **IEEE 754**.

- **Tipos definidos pelo usuário**

Na nossa linguagem será possível criar novos tipos a partir dos seguintes comandos:

- **struct:** permite agrupar dados de tipos diferentes numa única variável, formando um novo tipo de dado, podendo acessar campos específicos através do operador `'.'`. Ex.:

```
1 struct Aluno
2     string nome;
```

```

3     int idade;
4     int matricula;
5 endstruct
6
7 // Exemplos de uso
8 Aluno a; // Declaracao de variavel
9 a.idade = 18; // Atribuicao de valor
10 a.matricula = a.idade + 2005; // Atribuicao e acesso

```

- **union:** é parecido com as *structs*, a diferença é que todos os elementos ocupam o mesmo espaço de memória. Uma grande vantagem do uso de *union* é a possibilidade de economizar espaço, já que as suas variáveis são guardadas no mesmo espaço (para isso será levado em conta o tamanho da variável que ocupa o maior espaço). Ex.:

```

1 union ip_address
2     int s_long;
3     string s_byte[4];
4 endunion

```

- **enum:** são conjuntos de constantes organizados por identificadores e em ordem de declaração, ou seja, o que é declarado primeiro terá um valor menor. Mas, se necessário, é possível alterar o valor de uma constante dentro da estrutura. Ex.:

```

1 enum Semana
2     seg, ter, qua, qui, sex, sab, dom; // note o ',' no final
3 endenum

```

• Arrays

Os *arrays* na linguagem proposta permitem armazenar uma sequência de dados de mesmo tipo (homogêneo). É possível alocar uma quantidade de memória para eles de maneira estática ou dinâmica. O número máximo de elementos da estrutura deve ser especificado no momento da declaração e cada elemento pode ser acessado individualmente através do seu índice (um número inteiro positivo que vai de 0 a $n - 1$, onde n é o tamanho do *array*). Também será possível inicializar o *array*, no momento da declaração, com elementos pertencentes ao mesmo. Uma grande vantagem será a presença de um atributo *length*, que retorna o tamanho do *array*.

Acompanhe alguns exemplos de uso:

```
1 int array[5];
2 double vetor[3] = {0.25, 1.75, 10.75};
3 double a = vetor[2];
4 for(int i = 0; i < vetor.length; i++)
```

- **Tipos recursivos**

Em nossa linguagem utilizaremos tipos recursivos para construir matrizes (que serão vetores de vetores), árvores, listas encadeadas. Para implementar elas, será necessário o uso de ponteiros ou vetores como primitivas.

Os ponteiro são representados com o símbolo '*' antes da variável, por exemplo:

```
1 int *var;
2 var = alloc(sizeof(int));
3 free(var);
```

Onde **var** é uma variável ponteiro para um tipo **int**. Isso é representado pelo símbolo *. Na linha **2** podemos ver a função **alloc**, usada para alocar memória de acordo com o parâmetro inserido (retorna um endereço de memória). Ainda na linha **2** o argumento **sizeof(int)** é usado para retornar a quantidade(em bytes) de memória a ser alocada, neste caso, para um **int**. Na ultima linha é utilizada a função **free()** para liberar a memória alocada.

A linguagem também oferecerá tipos referência, como em C++, que podem ser declarados de maneira parecida com ponteiros, só que com o uso do operador &. Também é necessário que a variável que será referenciada seja inicializada antes. Acompanhe um exemplo:

```
1 int result = 0;
2 int &ref_result = result;
3
4 ref_result = 100;
5 println("result value: " + result);
6 println("ref_result value: " + ref_result);
```

O programa deverá imprimir na tela o valor 100, em ambas as funções, já que **ref_result** funciona como um “nome alternativo” para **result**, então qualquer alteração naquela será refletida nesta.

- **Conversões de tipo**

A linguagem possuirá tipagem forte sendo necessário declarar explicitamente o tipo da variável que está alocando e não é permitido armazenar dados de outros tipos diferentes do declarado para evitar erros de leitura por parte do compilador.

Todos os elementos e construções estarão sujeitos à verificação de tipo e esta ocorrerá na etapa de compilação e otimização do código.

Serão permitidas conversões implícitas e explícitas entre os tipos numéricos desde que o novo tipo comporte o tamanho, em bytes, do anterior (alargamento). Ex: `int` → `long/double`.

- **Blocos**

Os blocos serão delimitados por palavras reservadas. Qualquer corpo de um bloco deve terminar com a palavra reservada específica para fechar o bloco em questão. A seguir uma lista com as palavras de fechamento de blocos:

- | | |
|----------------------------|-----------------------------|
| – <code>endcase</code> | – <code>endprocedure</code> |
| – <code>endenum</code> | – <code>endstruct</code> |
| – <code>endfor</code> | – <code>endunion</code> |
| – <code>endfunction</code> | – <code>endwhile</code> |
| – <code>endif</code> | |

Por exemplo, para o bloco `if` será usado o `endif` para marcar o final da execução do mesmo.

Também serão permitidos blocos aninhados. Com isso, a regra de visibilidade será afetada quando uma variável em um bloco mais interno é declarada com o mesmo nome de outra de um bloco mais externo, fazendo com que a variável mais externa fique inacessível dentro daquele bloco. Essa visibilidade é determinada com o auxílio do ambiente de referenciamento.

- **Operadores**

Os operadores suportados em nossa linguagem serão apresentados no Apêndice C, por questões de estética.

Vale destacar que a linguagem suportará operadores curto-circuito quando os elementos envolvidos forem do tipo booleano, e contará com operadores diferentes para operações com ou sem curto-circuito. Ex: `a && b` (operação sem curto-circuito) e `a &&& b` (operação com curto-circuito).

- **Estruturas de controle**

- **Seleção:** `if`, `else`, similar a C/C++/Java.
- **Iteração:** `for`, `while`, também similar a C/C++/Java.

Devido à possibilidade de erros de utilização optamos por não disponibilizar instruções de desvio na linguagem e os *escapes* que poderão ser utilizados serão o `break` e `continue`, similar a C/C++/Java.

- **Comandos**

Principais comandos:

- | | |
|----------------------------|------------------------------|
| – Atribuição, | – Alloc, realloc, free, |
| – Declaração de variáveis, | – Print, println, read, |
| – If, continue, break, | – Invocação de subprogramas, |
| – For, while, | – Return. |

A maioria desses são similares as linguagens populares. Os comandos mais diferenciados são:

- `alloc()`: aloca um bloco de *bytes* consecutivos na memória do computador e devolve o endereço desse bloco. O número de bytes é especificado no argumento da função;
- `realloc()`: realoca um espaço de memória;
- `free()`: libera a porção de memória alocada por `alloc`. A instrução `free (ptr)` avisa ao sistema que o bloco de *bytes* apontado por `ptr` está livre;
- `print()` e `println()`: imprimem na tela uma *string* que é passada ao procedimento como argumento entre parênteses. As *strings* podem ser concatenadas com outras *strings* ou com outras variáveis de tipos primitivos (que terão seu valor convertido implicitamente para *string*). A diferença entre os dois procedimentos é que o último imprime a *string* com um ‘\n’ no final;
- `read()`: lê e armazena os dados vindos da entrada padrão, de acordo com um formato (como no `scanf()`), nas variáveis colocadas como parâmetros adicionais.

Alguns desses comandos são explorados na seção de exemplos.

- **Subprogramas**

A principal diferença entre uma função e um procedimento está no fato de que uma função obrigatoriamente retorna um valor, enquanto que um procedimento não retorna valor algum, ou seja o procedimento apenas executa uma ação. Portanto usaremos notações diferentes para diferenciá-los.

A linguagem utilizará duas formas de passagem de parâmetros: por valor e por referência. Para diferenciar, na passagem de parâmetro por referência será usado o operador `&`.

- **Categorias de variáveis**

A alocação das variáveis será **estática** para variáveis globais e variáveis estáticas. Será **automática** para variáveis locais e parâmetros de funções (tempo de vida nesse caso é igual ao escopo). Será **dinâmica explícita** quando requisitado pelo programador com o uso de comandos específicos, ficando a cargo dele gerenciar e liberar a memória posteriormente. Por fim a variável genérica será dinamicamente implícita.

- **Resolução de escopo**

O escopo da nossa linguagem será estático e apresentará o operador de escopo ‘: :’, que funcionará como em C.

- **Regras de Semântica estática**

Alguns erros só podem ser identificados através de regras semânticas. As principais regras serão:

- Uma variável não deve ser usada sem antes ser declarada;
- Uma variável só deve ser declarada, no máximo, uma vez em um mesmo bloco;
- Subprogramas devem ser chamados com a quantidade e tipos corretos de parâmetros;
- Em uma instrução de atribuição, o tipo da variável do lado esquerdo da atribuição deve ser compatível com o tipo da expressão do lado direito;
- Uma variável dinâmica genérica recebe qualquer tipo de variável, mas o contrário não se aplica.

Apêndice B – Palavras reservadas da linguagem

- | | | | |
|------------|----------------|------------|----------|
| • auto | • elsif | • endwhile | • static |
| • bool | • endcase | • enum | • string |
| • break | • endenum | • for | • struct |
| • case | • endfor | • if | • typeof |
| • const | • endfunction | • int | • union |
| • continue | • endif | • long | • when |
| • do | • endprocedure | • null | • while |
| • double | • endstruct | • return | |
| • else | • endunion | • sizeof | |

Apêndice C – Operadores da linguagem

- Operadores aritméticos:

Operador	Sintaxe
Adição	$a + b$
Adição unária	$+a$
Atribuição por adição	$a += b$
Subtração	$a - b$
Subtração unária	$-a$
Atribuição por subtração	$a -= b$
Multiplicação	$a * b$
Atribuição por multiplicação	$a *= b$
Divisão	a / b
Atribuição por divisão	$a /= b$
Resto	$a \% b$
Atribuição por resto	$a \% = b$

Tabela 2: Tabela com operadores aritméticos.

- Operadores comparativos/relacionais:

Operador	Sintaxe
Menor que	$a < b$
Menor ou igual que	$a \leq b$
Maior que	$a > b$
Maior ou igual que	$a \geq b$
Diferente de	$a \neq b$
Igual a	$a == b$

Tabela 3: Tabela com operadores comparativos/relacionais.

- Operadores lógicos:

Operador	Sintaxe
Não lógico	$!a$
E lógico	$a \&\& b$
Ou lógico	$a b$
E lógico com curto-circuito	$a \&\&\& b$
Ou lógico com curto-circuito	$a b$

Tabela 4: Tabela com operadores lógicos.

- Operadores lógicos sobre *bits*:

Operador	Sintaxe
Complemento	$\sim a$
<i>E</i>	$a \& b$
Atribuição por <i>e</i>	$a \&= b$
<i>Ou</i>	$a b$
Atribuição por <i>ou</i>	$a = b$
<i>Ou exclusivo</i>	$a \wedge b$
Atribuição por <i>ou exclusivo</i>	$a \wedge= b$

Tabela 5: Tabela com operadores lógicos sobre *bits*.

- Operadores de deslocamento de *bits*:

Operador	Sintaxe
Deslocamento à esquerda	$a \ll b$
Atribuição de deslocamento à esquerda	$a \ll= b$
Deslocamento à direita	$a \gg b$
Atribuição de deslocamento à direita	$a \gg= b$

Tabela 6: Tabela com operadores de deslocamento de *bits*.

- Outros operadores:

Operador	Sintaxe
Atribuição	$a = b$
Chamada de função	$a()$
Elemento de arranjo	$a[]$
Operador de indireção	$*a$
Referência	$\&a$
Membro de ponteiro	$a \rightarrow b$
Membro de identificador	$a.b$
Indireção de membro de ponteiro	$a \rightarrow *b$
Indireção de membro de identificador	$a.*b$
Conversão de tipo de dados	$(\text{tipo}) a$
Vírgula	a , b
Resolução de escopo	$a :: b$
Tamanho de	$\text{sizeof } (a)$
Identificador de tipo	$\text{typeof } (a)$

Tabela 7: Tabela com outros operadores.

Apêndice D – Gramática da linguagem

$\langle \text{program} \rangle ::= \langle \text{external_declaration} \rangle$
| $\langle \text{program} \rangle \langle \text{external_declaration} \rangle$

$\langle \text{translation_unit} \rangle ::= \langle \text{program_file} \rangle$

$\langle \text{program_file} \rangle ::= \langle \text{declarations} \rangle$

$\langle \text{declarations} \rangle ::= \langle \text{declaration} \rangle$
| $\langle \text{declarations} \rangle \langle \text{declaration} \rangle$

$\langle \text{declaration} \rangle ::= \langle \text{function_declaration} \rangle$
| $\langle \text{procedure_declaration} \rangle$
| $\langle \text{type_declaration} \rangle$
| $\langle \text{variable_declaration} \rangle$

$\langle \text{function_declaration} \rangle ::= \text{'function'} \langle \text{type_specifier} \rangle \langle \text{subprogram_declarator} \rangle \langle \text{subprogram_body} \rangle$
 'endfunction'

$\langle \text{procedure_declaration} \rangle ::= \text{'procedure'} \langle \text{subprogram_declarator} \rangle \langle \text{subprogram_body} \rangle \text{'endprocedure'}$

$\langle \text{subprogram_declarator} \rangle ::= \text{IDENTIFIER '('} \langle \text{parameter_list} \rangle \text{'})'}$
| $\text{IDENTIFIER '(' '}'$

$\langle \text{parameter_list} \rangle ::= \langle \text{parameter} \rangle$
| $\langle \text{parameter_list} \rangle \text{' , ' } \langle \text{parameter} \rangle$

$\langle \text{parameter} \rangle ::= \langle \text{type_specifier} \rangle \langle \text{declarator_name} \rangle$
| $\text{'const'} \langle \text{type_specifier} \rangle \langle \text{declarator_name} \rangle$

$\langle \text{subprogram_body} \rangle ::= \langle \text{block} \rangle$
| ' ; '

$\langle \text{block} \rangle ::= \langle \text{local_variable_declarations_and_statements} \rangle$

$\langle \text{local_variable_declarations_and_statements} \rangle ::= \langle \text{local_variable_declarations_or_statements} \rangle$
| $\langle \text{local_variable_declarations_and_statements} \rangle \langle \text{local_variable_declarations_or_statements} \rangle$

$\langle \text{local_variable_declarations_or_statements} \rangle ::= \langle \text{local_variable_declaration_statement} \rangle$
| $\langle \text{statement} \rangle$

$\langle \text{local_variable_declaration_statement} \rangle ::= \langle \text{type_specifier} \rangle \langle \text{variable_declarators} \rangle \text{' ; '}$
| $\text{'static'} \langle \text{type_specifier} \rangle \langle \text{variable_declarators} \rangle \text{' ; '}$

$\langle \text{statement} \rangle ::= \langle \text{assignment_expression} \rangle \text{' ; '}$
| $\langle \text{selection_statement} \rangle$

```

| <iteration_statement>
| <jump_statement>
| <read_statement>
| <print_statement>
| <println_statement>

<selection_statement> ::= 'if' '(' <expression> ')' <block> 'elseif' <elsif_staments> 'endif'
| 'if' '(' <expression> ')' <block> 'else' <block> 'endif'
| 'if' '(' <expression> ')' <block> 'endif'

<elsif_staments> ::= <elsif_statement>
| <elsif_staments> 'elseif' <elsif_statement>
| 'else' <block>

<elsif_statement> ::= '(' <expression> ')' <block>

<iteration_statement> ::= 'while' '(' <expression> ')' <block> 'endwhile'
| 'for' '(' <for_init> <for_expr> <for_incr> ')' <block> 'endfor'
| 'for' '(' <for_init> <for_expr> ')' <block> 'endfor'

<for_init> ::= <expression_statements> ';'
| <local_variable_declaration_statement>
| ';'

<for_expr> ::= <expression> ';'
| ';'

<for_incr> ::= <expression_statements>

<expression_statements> ::= <expression_statement>
| <expression_statements> ',' <expression_statement>

<expression_statement> ::= <expression>

<jump_statement> ::= 'break' IDENTIFIER ';'
| 'break' ';'
| 'continue' IDENTIFIER ';'
| 'continue' ';'
| 'return' <expression> ';'
| 'return' ';'

<read_statement> ::= 'read' '(' STRING_LITERAL ',' <argument_list> ')' ';'
| 'read' '(' STRING_LITERAL ')' ';'

<print_statement> ::= 'print' '(' <expression> ')' ';'

<println_statement> ::= 'println' '(' <expression> ')' ';'

```


$\langle \text{type_declaration} \rangle ::= \text{'struct' IDENTIFIER } \langle \text{variable_declarations} \rangle \text{'endstruct'}$
 $\quad | \text{'union' IDENTIFIER } \langle \text{discriminant} \rangle \langle \text{union_body} \rangle \text{'endunion'}$
 $\quad | \text{'enum' IDENTIFIER } \langle \text{enumerator_list} \rangle \text{'endenum'}$

$\langle \text{variable_declarations} \rangle ::= \langle \text{variable_declaration} \rangle$
 $\quad | \langle \text{variable_declarations} \rangle \langle \text{variable_declaration} \rangle$

$\langle \text{variable_declaration} \rangle ::= \langle \text{modifiers} \rangle \langle \text{type_specifier} \rangle \langle \text{variable_declarators} \rangle \text{';'}$
 $\quad | \langle \text{type_specifier} \rangle \langle \text{variable_declarators} \rangle \text{';'}$

$\langle \text{modifiers} \rangle ::= \langle \text{modifier} \rangle$
 $\quad | \langle \text{modifiers} \rangle \langle \text{modifier} \rangle$

$\langle \text{modifier} \rangle ::= \text{'const'}$
 $\quad | \text{'static'}$

$\langle \text{type_specifier} \rangle ::= \langle \text{type_name} \rangle$

$\langle \text{type_name} \rangle ::= \langle \text{primitive_type} \rangle$
 $\quad | \langle \text{qualified_name} \rangle$

$\langle \text{primitive_type} \rangle ::= \text{'auto'}$
 $\quad | \text{'bool'}$
 $\quad | \text{'int'}$
 $\quad | \text{'long'}$
 $\quad | \text{'double'}$
 $\quad | \text{'string'}$

$\langle \text{discriminant} \rangle ::= \text{'(' } \langle \text{type_specifier} \rangle \langle \text{declarator_name} \rangle \text{'}'}$
 $\quad | \epsilon$

$\langle \text{enumerator_list} \rangle ::= \langle \text{enumerator} \rangle$
 $\quad | \langle \text{enumerator_list} \rangle \text{' ,' } \langle \text{enumerator} \rangle$

$\langle \text{enumerator} \rangle ::= \text{IDENTIFIER '=' } \langle \text{expression} \rangle$
 $\quad | \text{IDENTIFIER}$

$\langle \text{qualified_name} \rangle ::= \text{IDENTIFIER}$

$\langle \text{union_body} \rangle ::= \langle \text{variable_declarations} \rangle \langle \text{short_case_statement} \rangle$
 $\quad | \langle \text{variable_declarations} \rangle$

$\langle \text{short_case_statement} \rangle ::= \text{'case' IDENTIFIER } \langle \text{choices} \rangle \text{'endcase'}$

$\langle \text{choices} \rangle ::= \langle \text{choice} \rangle$
 $\quad | \langle \text{choices} \rangle \langle \text{choice} \rangle$

$\langle \text{choice} \rangle ::= \text{'when' IDENTIFIER 'then' } \langle \text{variable_declarations} \rangle$
 $\langle \text{variable_declarators} \rangle ::= \langle \text{variable_declarator} \rangle$
 $\quad | \quad \langle \text{variable_declarators} \rangle \text{' , ' } \langle \text{variable_declarator} \rangle$
 $\langle \text{variable_declarator} \rangle ::= \langle \text{declarator_name} \rangle$
 $\quad | \quad \langle \text{declarator_name} \rangle \text{' = ' } \langle \text{variable_initializer} \rangle$
 $\langle \text{declarator_name} \rangle ::= \text{IDENTIFIER}$
 $\quad | \quad \langle \text{declarator_name} \rangle \langle \text{dim_exprs} \rangle$
 $\langle \text{variable_initializer} \rangle ::= \langle \text{expression} \rangle$
 $\quad | \quad \text{' { ' ' } '}$
 $\quad | \quad \text{' { ' } \langle \text{array_initializers} \rangle \text{' } \text{' } \text{' } \text{' }$
 $\langle \text{array_initializers} \rangle ::= \langle \text{variable_initializer} \rangle$
 $\quad | \quad \langle \text{array_initializers} \rangle \text{' , ' } \langle \text{variable_initializer} \rangle$
 $\langle \text{expression} \rangle ::= \langle \text{assignment_expression} \rangle$
 $\langle \text{assignment_expression} \rangle ::= \langle \text{conditional_or_expression} \rangle$
 $\quad | \quad \langle \text{unary_expression} \rangle \langle \text{assignment_operator} \rangle \langle \text{assignment_expression} \rangle$
 $\langle \text{unary_expression} \rangle ::= \langle \text{arithmetic_unary_operator} \rangle \langle \text{cast_expression} \rangle$
 $\langle \text{logical_unary_expression} \rangle ::= \langle \text{postfix_expression} \rangle$
 $\quad | \quad \langle \text{logical_unary_operator} \rangle \langle \text{unary_expression} \rangle$
 $\langle \text{postfix_expression} \rangle ::= \langle \text{primary_expression} \rangle$
 $\langle \text{primary_expression} \rangle ::= \langle \text{qualified_name} \rangle$
 $\quad | \quad \langle \text{not_just_name} \rangle$
 $\langle \text{arithmetic_unary_operator} \rangle ::= \text{' + '}$
 $\quad | \quad \text{' - '}$
 $\langle \text{logical_unary_operator} \rangle ::= \text{' ~ '}$
 $\quad | \quad \text{' ! '}$
 $\langle \text{conditional_or_expression} \rangle ::= \langle \text{conditional_and_expression} \rangle$
 $\quad | \quad \langle \text{conditional_or_expression} \rangle \text{' || ' } \langle \text{conditional_and_expression} \rangle$
 $\quad | \quad \langle \text{conditional_or_expression} \rangle \text{' ||| ' } \langle \text{conditional_and_expression} \rangle$
 $\langle \text{conditional_and_expression} \rangle ::= \langle \text{inclusive_or_expression} \rangle$
 $\quad | \quad \langle \text{conditional_and_expression} \rangle \text{' \&\& ' } \langle \text{inclusive_or_expression} \rangle$
 $\quad | \quad \langle \text{conditional_and_expression} \rangle \text{' \&\&\& ' } \langle \text{inclusive_or_expression} \rangle$

$\langle \text{inclusive_or_expression} \rangle ::= \langle \text{exclusive_or_expression} \rangle$
 $\quad | \quad \langle \text{inclusive_or_expression} \rangle ' | ' \langle \text{exclusive_or_expression} \rangle$

$\langle \text{exclusive_or_expression} \rangle ::= \langle \text{and_expression} \rangle$
 $\quad | \quad \langle \text{exclusive_or_expression} \rangle '^' \langle \text{and_expression} \rangle$

$\langle \text{and_expression} \rangle ::= \langle \text{equality_expression} \rangle$
 $\quad | \quad \langle \text{and_expression} \rangle '&' \langle \text{equality_expression} \rangle$

$\langle \text{equality_expression} \rangle ::= \langle \text{relational_expression} \rangle$
 $\quad | \quad \langle \text{equality_expression} \rangle '==' \langle \text{relational_expression} \rangle$
 $\quad | \quad \langle \text{equality_expression} \rangle '!=' \langle \text{relational_expression} \rangle$

$\langle \text{relational_expression} \rangle ::= \langle \text{shift_expression} \rangle$
 $\quad | \quad \langle \text{relational_expression} \rangle '<' \langle \text{shift_expression} \rangle$
 $\quad | \quad \langle \text{relational_expression} \rangle '>' \langle \text{shift_expression} \rangle$
 $\quad | \quad \langle \text{relational_expression} \rangle '<=' \langle \text{shift_expression} \rangle$
 $\quad | \quad \langle \text{relational_expression} \rangle '>=' \langle \text{shift_expression} \rangle$

$\langle \text{shift_expression} \rangle ::= \langle \text{additive_expression} \rangle$
 $\quad | \quad \langle \text{shift_expression} \rangle '\ll' \langle \text{additive_expression} \rangle$
 $\quad | \quad \langle \text{shift_expression} \rangle '\gg' \langle \text{additive_expression} \rangle$

$\langle \text{additive_expression} \rangle ::= \langle \text{multiplicative_expression} \rangle$
 $\quad | \quad \langle \text{additive_expression} \rangle '+' \langle \text{multiplicative_expression} \rangle$
 $\quad | \quad \langle \text{additive_expression} \rangle '-' \langle \text{multiplicative_expression} \rangle$

$\langle \text{multiplicative_expression} \rangle ::= \langle \text{cast_expression} \rangle$
 $\quad | \quad \langle \text{multiplicative_expression} \rangle '*' \langle \text{cast_expression} \rangle$
 $\quad | \quad \langle \text{multiplicative_expression} \rangle '/' \langle \text{cast_expression} \rangle$
 $\quad | \quad \langle \text{multiplicative_expression} \rangle '%' \langle \text{cast_expression} \rangle$

$\langle \text{cast_expression} \rangle ::= \langle \text{unary_expression} \rangle$
 $\quad | \quad '(' \langle \text{primitive_type_expression} \rangle ')' \langle \text{cast_expression} \rangle$
 $\quad | \quad '(' \langle \text{user_type_expression} \rangle ')' \langle \text{cast_expression} \rangle$
 $\quad | \quad '(' \langle \text{expression} \rangle ')' \langle \text{logical_unary_expression} \rangle$

$\langle \text{primitive_type_expression} \rangle ::= \langle \text{primitive_type} \rangle$
 $\quad | \quad \langle \text{primitive_type} \rangle \langle \text{dims} \rangle$

$\langle \text{user_type_expression} \rangle ::= \langle \text{qualified_name} \rangle \langle \text{dims} \rangle$

$\langle \text{assignment_operator} \rangle ::= '='$
 $\quad | \quad '+='$
 $\quad | \quad '-='$
 $\quad | \quad '*='$

- | ‘/=’
- | ‘%=’
- | ‘&=’
- | ‘|=’
- | ‘^=’
- | ‘<=<=’
- | ‘>=>=’

$\langle not_just_name \rangle ::= \langle complex_primary \rangle$

$\langle complex_primary \rangle ::= ‘(’ \langle expression \rangle ‘)’$

$\langle complex_primary_no_parenthesis \rangle ::= \text{BOOL_LITERAL}$

- | OCTAL
- | DECIMAL
- | HEX
- | FLOATING_POINT
- | ‘null’
- | STRING_LITERAL
- | $\langle array_access \rangle$
- | $\langle field_access \rangle$
- | $\langle subprogram_call \rangle$

$\langle array_access \rangle ::= \langle qualified_name \rangle ‘[’ \langle expression \rangle ‘]’$

- | $\langle complex_primary \rangle ‘[’ \langle expression \rangle ‘]’$

$\langle field_access \rangle ::= \langle postfix_expression \rangle \text{DOT IDENTIFIER}$

$\langle subprogram_call \rangle ::= \langle subprogram_access \rangle ‘(’ \langle argument_list \rangle ‘)’$

- | $\langle subprogram_access \rangle ‘(’ ‘)’$

$\langle subprogram_access \rangle ::= \langle complex_primary_no_parenthesis \rangle$

- | $\langle qualified_name \rangle$

$\langle argument_list \rangle ::= \langle expression \rangle$

- | $\langle argument_list \rangle ‘,’ \langle expression \rangle$

$\langle dim_exprs \rangle ::= \langle dim_expr \rangle$

- | $\langle dim_exprs \rangle \langle dim_expr \rangle$

$\langle dim_expr \rangle ::= ‘[’ \langle expression \rangle ‘]’$

$\langle dims \rangle ::= ‘[’ ‘]’$

- | $\langle dims \rangle ‘[’ ‘]’$