INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE MINAS GERAIS - *CAMPUS* BAMBUÍ
BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

Henrique Araujo Miranda
Julia Gabriella Corrêa Silva
Johnattan Silva Ferreira

TRABALHO FINAL COMPILADORES

Bambuí - MG
2023

SUMÁRIO

1 Linguagem Rubrum

A linguagem de programação Rubrum, se destaca pela abordagem única, utilizando o idioma latim como base para as palavras reservadas, proporcionando uma experiência diferenciada para programadores iniciantes. Seu objetivo é fornecer um ponto de partida acessível para quem deseja entrar no mundo da programação.

Ao contrário de algumas linguagens mais avançadas, Rubrum utiliza um escopo simples, fornecendo uma base sólida para o desenvolvimento do raciocínio lógico e compreensão das estruturas básicas presentes em qualquer linguagem de programação. Suas estruturas básicas incluem laços, condicionais, declaração de variáveis, operadores aritméticos, relacionais e lógicos.

Assim como o latim é um idioma clássico e é a base para muitas outras línguas, a linguagem Rubrum pretende ser uma base sólida para a aprendizagem de programação, incentivando a compreensão dos princípios fundamentais críticos em qualquer ambiente de desenvolvimento de software.

A seguir, é mostrado a Tabela 1 de tokens da linguagem, que se trata de uma estrutura de dados que armazena informações sobre cada token identificado. Para cada token, a tabela contém detalhes como a expressão regular, tipo e a descrição do token. A tabela é empregada na fase de implementação da análise léxica, na qual o autômato finito determinístico (AFD) é responsável por reconhecer e aceitar tanto palavras quanto outros símbolos.

Observação:

* = 0 ou mais repetições

+ = 1 ou mais repetições

Tabela 1 - Tokens

Expressão Regular	**Token**	**Descrição**
[0-9]		Dígitos
[a-z | A-Z]		Caracteres
(a-z) (a-z | A-Z | 0-9)*	<id, >	Identificador
(0-9)+	<num, >	Constante Numérica
“ (a-z | A-Z | 0-9)* ”	<literal, “”>	Constante Literal

SATUS	<SATUS, >	Palavra Reservada
LITTERAE	<LITTERAE, >	Palavra Reservada
NUMERUS	<NUMERUS, >	Palavra Reservada
SI	<SI, >	Palavra Reservada
ALIUD	<ALIUD, >	Palavra Reservada
DUM	<DUM, >	Palavra Reservada
LEGERE	<LEGERE, >	Palavra Reservada
SCRIBERE	<SCRIBERE, >	Palavra Reservada
REDITUS	<REDITUS, >	Palavra Reservada
!		Comentário
; = |	<|,>	Símbolo especial
(	<(,>	Símbolo especial
)	<),>	Símbolo especial
[	<[,>	Símbolo especial
]	<],>	Símbolo especial
+	<+,>	Operador Aritmético
-	<-,>	Operador Aritmético
/	</,>	Operador Aritmético
*	<*,>	Operador Aritmético
<	<<,>	Operador Relacional
>	<>,>	Operador Relacional
<=	<<=,>	Operador Relacional
>=	<>=,>	Operador Relacional
=	<=,>	Comando de Atribuição
AUT	<AUT,>	Operador lógico
ET	<ET,>	Operador lógico
NO	<NO,>	Operador lógico

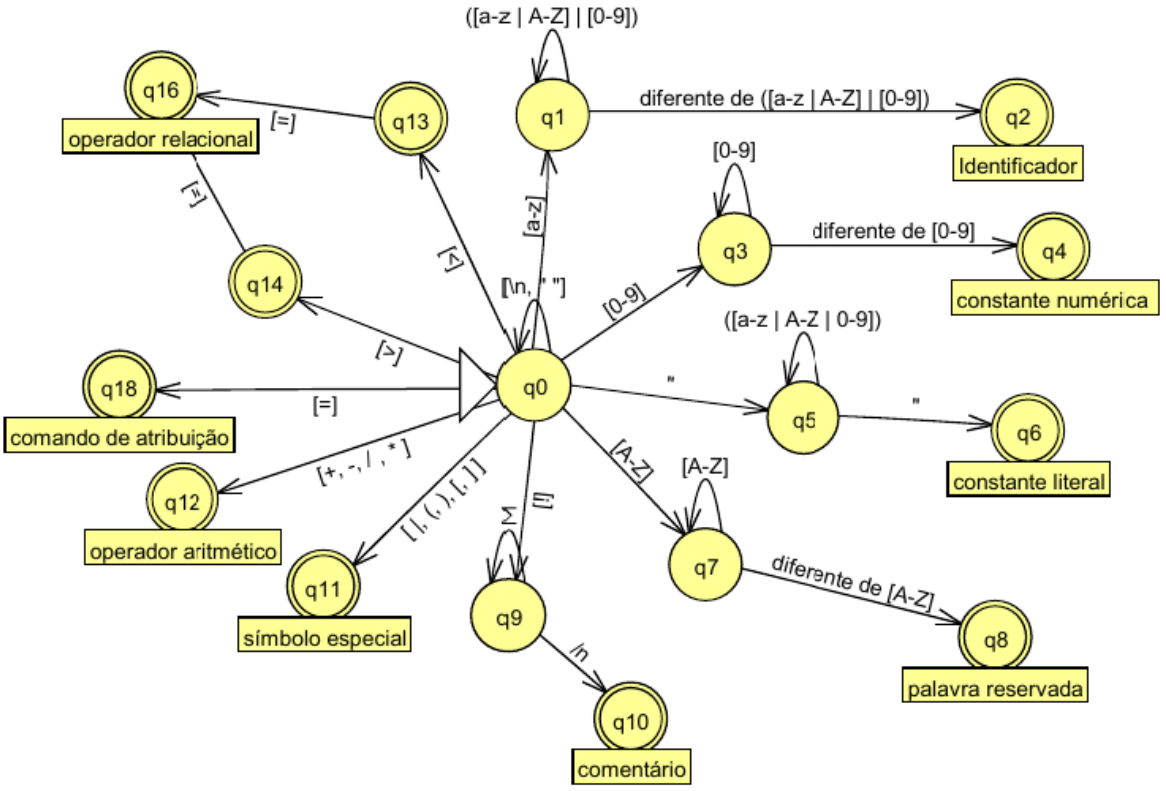
VERUM	<VERUM, >	Booleano
FALSUS	<FALSUS, >	Booleano

Fonte: Elaborado pelos autores, 2023.

2 Autômato

Os AFD possuem, para cada combinação de estado e entrada, uma única transição aplicável. Desempenham um papel crucial na implementação da análise léxica para identificação de palavras e outros símbolos específicos. Neste trabalho, atua como uma ferramenta determinística, aceitando sequências de caracteres baseadas nas regras pré definidas enquanto percorre o código-fonte. Essa abordagem facilita a identificação eficiente e precisa dos tokens, auxiliando nos estágios iniciais do processo de compilação. Na Figura 1 a seguir, é mostrado o autômato da linguagem Rubrum.

Figura 1 - Autômato da Linguagem Rubrum



Fonte: Elaborado pelos autores, 2023.

3 Gramática

A gramática em compiladores desempenha um papel crucial na análise e interpretação de linguagens de programação. Trata-se de um conjunto formal de regras que define a estrutura sintática de uma linguagem, permitindo que o compilador compreenda e processe corretamente o código-fonte. Portanto, entender e definir gramáticas de forma eficaz é essencial para o desenvolvimento de compiladores robustos e eficientes, garantindo a precisão na transformação do código-fonte em código executável.

A linguagem Rubrum utiliza a classe de gramática preditiva LL(1), que permite análise utilizando apenas um símbolo de entrada por vez. Sua principal vantagem está na simplicidade de implementação em analisadores sintáticos preditivos, que são eficientes e podem ser construídos de forma recursiva descendente. A tabela de análise preditiva é essencial nesse processo, mapeando não-terminais e terminais para as produções correspondentes. A Tabela 2 a seguir, demonstra a estrutura gramatical da linguagem.

Tabela 2 - Gramática da Linguagem

MAIN	->	SATUS '(' DECLARA BLOCO ')'
DECLARA	->	VAR DECLARA | ε
VAR	->	TYPE ID (, ID)* '|' | TYPE ATR
TYPE	->	LITTERAE | NUMERUS
ATR	->	ID = (TEXTO | EXP) '|'
BLOCO	->	CMD BLOCO | ε
CMD	->	IN | OUT | LOOP | IF | EXP | ATR
IN	->	LEGERE [ID] '|'
OUT	->	SCRIBERE [ID | TEXTO] '|' | REDITUS [ID] '|'
LOOP	->	DUM [EXP OP EXP | BOOL] '(' BLOCO ')'
IF	->	SI [EXP OP EXP] '(' BLOCO ')' (ALIUD BLOCO)?
OP	->	LOGICO | ARITMETICO | RELACIONAL
LOGICO	->	AUT | ET | NO

ARITMETICO	->	N0 | N1
N0	->	- | +
N1	->	* | /
RELACIONAL	->	> | < | >= | <=
EXP	->	TERM EXP'
EXP'	->	N0 EXP | ε
TERM	->	FATOR TERM'
TERM'	->	N1 TERM | ε
FATOR	->	NUMERO | ID | '(' EXP ')'
TEXTO	->	" (LETRA | NUMERO)* "
LETRA	->	(a..z | A..Z)*
ID	->	(a..z)(A..Z | a..z | 0..9)*
NUMERO	->	(0..9)+
BOOL	->	VERUM | FALSUS

Fonte: Elaborado pelos autores, 2023.

4 Tabela First e Follow

O conjunto First e Follow desempenha um papel crucial na implementação de compiladores, sendo essencial para a análise sintática eficiente de linguagens de programação. O conjunto First são todos os terminais que podem ser o primeiro terminal para aquele dado símbolo. Essa informação é valiosa para o compilador, pois permite antecipar os possíveis símbolos iniciais de uma produção.

Por outro lado, o conjunto Follow de uma variável não-terminal representa os terminais que podem aparecer imediatamente após dado símbolo, ou seja, à sua direita em alguma derivação. Essa informação é crucial para determinar onde uma produção pode ser encerrada e como continuar a análise. Além disso, a análise preditiva, que é frequentemente baseada nos conjuntos First e Follow, permite a criação de analisadores sintáticos mais simples e eficientes. A Tabela 3 a seguir, mostra a tabela do First e Follow da linguagem.

Tabela 3 - Conjuntos First e Follow da Linguagem

Variável	**First**	**Follow**
MAIN	{SATUS}	{ $ }
DECLARA	{LITTERAE, NUMERUS, ε}	{LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,)}
VAR	{LITTERAE, NUMERUS}	{LITTERAE, NUMERUS, LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,)}
TYPE	{LITTERAE, NUMERUS}	{(a..z)(A..Z | a..z | 0..9)*}
ATR	{(a..z)(A..Z | a..z | 0..9)*}	{LITTERAE, NUMERUS, LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,)}
BLOCO	{LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*, ε}	{) }
CMD	{LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*}	{LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,)}
IN	{LEGERE}	{LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,)}
OUT	{SCRIBERE, REDITUS}	{LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,)}
LOOP	{DUM}	{LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,)}
IF	{SI}	{LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,)}
OP	{AUT, ET, NO, -, +, *, /, >, <, >=, <=}	{(0..9)+, (a..z)(A..Z | a..z | 0..9)*}
LOGICO	{AUT, ET, NO}	{(0..9)+, (a..z)(A..Z | a..z | 0..9)*}

ARITMETICO	{-, +, *, /}	{(0..9)+, (a..z)(A..Z | a..z | 0..9)*}
N0	{-, +}	{(0..9)+, (a..z)(A..Z | a..z | 0..9)*}
N1	{*, /}	{(0..9)+, (a..z)(A..Z | a..z | 0..9)*}
RELACIONAL	{>, <, >=, <=}	{(0..9)+, (a..z)(A..Z | a..z | 0..9)*}
EXP	{(0..9)+, (a..z)(A..Z | a..z | 0..9)*}	{|, LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,), AUT, ET, NO, +, -, *, /, >, <, >=, <=,], *, /}
EXP'	{-, +, ε}	{|, LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,), AUT, ET, NO, +, -, *, /, >, <, >=, <=,], *, /}
TERM	{(0..9)+, (a..z)(A..Z | a..z | 0..9)*}	{|, LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,), AUT, ET, NO, +, -, *, /, >, <, >=, <=,], *, /}
TERM'	{*, /, ε}	{|, LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,), AUT, ET, NO, +, -, *, /, >, <, >=, <=,], *, /}
FATOR	{(0..9)+, (a..z)(A..Z | a..z | 0..9)*}	{LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,), AUT, ET, NO, +, -, *, /, >, <, >=, <=,], *, /}
TEXTO	{“}	{LITTERAE, NUMERUS, LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,]}
LETRA	{(a..z | A..Z)*}	{ “ }
ID	{(a..z)(A..Z | a..z | 0..9)*}	{|, =, LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,), AUT, ET, NO, +, -, *, /, >, <, >=, <=,], *, /}

NUMERO	{(0..9)+}	{|, LEGERE, SCRIBERE, REDITUS, DUM, SI, (0..9)+, (a..z)(A..Z | a..z | 0..9)*,), AUT, ET, NO, +, -, *, /, >, <, >=, <=,], *, /, “}
BOOL	{VERUM, FALSUS}	{ [}

Fonte: Elaborado pelos autores, 2023.

5 Compilador

Um compilador é um programa que traduz o código-fonte de uma linguagem de programação para um código executável, passando por etapas como análise léxica, sintática, semântica e geração de código intermediário, culminando na produção de um código final em linguagem de máquina.

Essa tradução permite que os desenvolvedores escrevam em linguagens de alto nível, tornando o código mais compreensível, enquanto o compilador o converte para uma forma executável pelo hardware do computador. Essa abordagem facilita o desenvolvimento de software eficiente e portátil, sendo essencial no ciclo de desenvolvimento de aplicativos.

As estratégias criadas nas etapas de análise léxica, sintática e semântica do compilador apresentado neste trabalho foram baseadas nos materiais e vídeos fornecidos pelo professor da disciplina.

5.1 Requisitos Funcionais

Requisitos funcionais são especificações que detalham as operações e comportamentos que um sistema deve realizar para atender às necessidades dos usuários. Fornecem uma base clara para o desenvolvimento, teste e validação do sistema, delineando as ações que o sistema deve ser capaz de executar. Essa documentação é fundamental para garantir que o produto final atenda efetivamente às expectativas e necessidades dos usuários. Neste trabalho, os requisitos do front-end foram divididos nas três etapas principais, como mostra a Tabela 4 a seguir.

Tabela 4 - Requisitos Funcionais da Linguagem

Etapa	Função	Requisito Funcional
Análise Léxica	Verificar e reconhecer os Tokens	Deve receber o arquivo de entrada e identificar os tokens.
		Deve classificar e retornar os tokens para palavras-chave, identificadores, símbolos especiais, operadores lógicos, relacionais e aritméticos na linguagem.
		Deve possibilitar que os tokens produzidos sejam empregados nas fases seguintes do processo de compilação.
		Deve ignorar comentários, espaçamentos, quebras de linha e tabulações.
		Deve detectar erros léxicos, como palavras reservadas, símbolos e operadores.
Análise Sintática	Verificar a estrutura da gramática	Deve analisar as configurações dos tokens em conformidade com a gramática estabelecida.
		Deve ser capaz de antecipar um único caminho a partir da leitura de um token.
		Deve detectar erros sintáticos, como a ordem dos tokens, símbolos mal estruturados e palavras reservadas fora de sequência.
Análise Semântica	Verificar os possíveis erros não tratados nas fases anteriores	Deve ser capaz de não declarar duas variáveis com o mesmo identificador.
		Deve verificar se a variável foi declarada antes de usar.
		Deve impedir que um carácter receba número e vise versa na declaração.

Fonte: Elaborado pelos autores, 2023.

5.2 Requisitos Não Funcionais

Os requisitos não funcionais referem-se a critérios que descrevem características do sistema que não estão relacionadas diretamente com as operações específicas, mas que impactam sua eficácia, desempenho e usabilidade. Esses requisitos são fundamentais para assegurar que o sistema não apenas execute as funções necessárias, mas também atenda aos padrões de qualidade e desempenho estabelecidos. Na Tabela 5 a seguir, é possível verificar os requisitos não funcionais da linguagem.

Tabela 5 - Requisitos Não Funcionais da Linguagem

Propriedade	**Medida**
Eficiência	O tempo de compilação deve ser proporcional à complexidade e tamanho do código.
Portabilidade	O compilador deve ser capaz de gerar código executável que seja compatível com diferentes plataformas e arquiteturas.
Usabilidade	O compilador deve fornecer mensagens de erros claros, contendo tipo de erro, sendo eles léxico, sintático, semântico e a linha que se encontra.
Confiabilidade	Ser robusto em relação a entradas inesperadas e garantir a consistência do processo de compilação.
Funcionalidade	Garantir que o compilador suporte recursão de maneira eficiente, permitindo a implementação de algoritmos e estruturas de dados recursivas.

Fonte: Elaborado pelos autores, 2023.

5.3 Análise Léxica

Ao construir o analisador léxico, a abordagem adotada envolveu a criação de um autômato finito determinístico diretamente no código. Foi implementado por meio de estruturas condicionais que representam cada estado do autômato. Cada estado

está associado a um padrão específico, como uma palavra-chave, identificador, operador ou símbolo, e as transições entre os estados são determinadas pela entrada do código-fonte.

Esta técnica permitiu a identificação eficiente de tokens durante a análise de código. Depois que um token é reconhecido, informações importantes sobre ele são registradas, como seu tipo, texto e a linha no arquivo de onde foi encontrado. Esses dados são armazenados para uso posterior na detecção de erros durante a análise sintática ou semântica.

O resultado proporciona uma maneira organizada e estruturada de realizar a análise léxica, permitindo a identificação e classificação precisa de tokens no código-fonte, além de facilitar a localização e correção de eventuais erros durante etapas subsequentes do processo de compilação.

5.4 Análise Sintática

Ao construir o analisador sintático, foi necessário criar uma gramática da linguagem Rubrum, baseado na gramática preditiva. Este tipo de analisador é descendente (top-down), ou seja, inicia a análise a partir do primeiro símbolo da raiz da árvore e desce para as folhas, fazendo a leitura dos tokens da entrada da esquerda para a direita.

Esta escolha significa que a gramática utilizada deve ser preditiva, permitindo que previsões precisas sejam feitas com base no próximo símbolo de entrada. A abordagem adotada permite eliminar a recursividade à esquerda e o analisador usa apenas um símbolo anterior para tomar decisões de análise. Esta estratégia simplifica a implementação e é muito eficiente em termos de desempenho, facilitando a construção de analisadores de forma clara e estruturada.

5.5 Análise Semântica

A construção do analisador semântico foi feita para complementar a análise sintática e é incorporada como extensão no mesmo arquivo. Esta abordagem permite uma integração mais estreita entre as duas etapas do processo de compilação, simplificando a detecção de erros e realizando verificações semânticas mais avançadas.

Essa etapa no compilador não apenas adiciona uma camada adicional de verificação de erros, mas também contribui para a robustez e confiabilidade do software gerado. A abordagem integrada entre análise sintática e semântica facilita a identificação precoce de problemas e a entrega de mensagens de erro claras, promovendo a qualidade e a eficácia do processo de compilação.

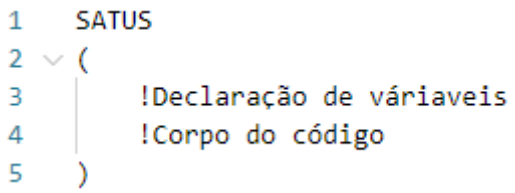
6 Manual

Neste manual, são apresentados exemplos de aplicação da linguagem Rubrum, que ilustram as diversas funcionalidades.

6.1 Estrutura do programa

Para iniciar um programa na linguagem Rubrum, é necessário adicionar a palavra-chave STATUS e em seguida parênteses. Dentro deles, todo o escopo do código será descrito, como mostrado na estrutura abaixo:

Figura 2 - Estrutura da Linguagem Rubrum



Fonte: Elaborado pelos autores, 2023.

Os comentários podem ser adicionados utilizando o símbolo “!” no início da frase. É fundamental observar que a ordem da estrutura deve seguir a demonstração acima, com as declarações de variáveis precedendo os comandos. Este padrão garante uma organização coerente e facilita a compreensão do código.

6.2 Tipos de dados

As variáveis são áreas de memória reservadas pelo compilador para armazenar informações, e podem ser categorizadas em três tipos distintos: NUMERUS (utilizado para valores inteiros), LITTERAE (utilizado para strings) e BOOL (utilizado para valores booleanos).

6.3 Declaração e Atribuição

Na Figura a seguir, serão apresentadas as formas de declarações de variáveis, bem como métodos para atribuição de valores. É essencial observar que é preciso realizar a declaração de uma variável antes de utilizá-la no corpo do código.

Para declarar variáveis, é necessário escrever um tipo seguido por um identificador. Já para realizar a atribuição, utiliza-se o símbolo "=", sendo importante observar que, ao lidar com variáveis do tipo LITTERAE, o conteúdo deve estar entre aspas. Além disso, após o final de cada linha, é necessário inserir o símbolo pipe "|" para indicar a finalização.

Figura 3 - Declaração e Atribuição da Linguagem Rubrum

```
1   SATUS
2   (
3       !Declaração de variáveis
4
5       NUMERUS x|
6       LITTERAE a|
7
8       NUMERUS y,z|
9       LITTERAE b,c|
10
11      NUMERUS w = 23|
12      LITTERAE d = "Salve mundo"|
13
14
15      !Corpo do código
16
17      y = 1 + (2 * 3)|
18      b = "Salve mundo 1"|
19  )
```

Fonte: Elaborado pelos autores, 2023.

6.4 Entrada e Saída

Para a interação com o usuário, a linguagem possui duas funções específicas: LEGERE, responsável por ler dados fornecidos pelo usuário através de uma variável, e SCRIBERE, utilizada para escrever as informações de volta ao usuário. Essas operações fundamentais desempenham um papel essencial na dinâmica.

Figura 4 - Entrada e Saída da Linguagem Rubrum

```
1   SATUS
2   (
3       !Declaração de variáveis
4
5       LITTERAE palavra|
6
7
8       !Corpo do código
9
10      LEGERE[palavra] | !"palavra" recebe o conteúdo digitado
11      SCRIBERE[palvra]| !conteudo de "palavra" é escrito na tela
12  )
```

Fonte: Elaborado pelos autores, 2023

6.5 Comandos SI, ALIUD e DUM

Na linguagem Rubrum, a execução dos comandos dentro da estrutura do "SI" ocorre somente se a condição entre parênteses for satisfeita. A representação dessa estrutura está ilustrada na Figura 5 abaixo, entre as linhas 7 e 14, sendo o uso de "ALIUD" opcional. Se a condição dentro dos parênteses do comando "SI" não for atendida, os comandos dentro de "ALIUD" serão executados.

Essa construção estabelece uma clara analogia com os comandos condicionais mais comuns, como "IF" e "ELSE", presentes em diversas linguagens de programação.

No intervalo entre as linhas 15 e 20, é encontrado o comando de repetição "DUM", que se correlaciona diretamente com o comando "WHILE" de outras linguagens. É importante ressaltar que o decremento da variável de controle necessita ser realizado manualmente dentro do corpo da função.

Figura 5 - Comandos da Linguagem Rubrum

```
SATUS
(
    LITTERAE a = "Salve mundo"|
    NUMERUS x = 10|
    NUMERUS y = 10|

    SI[x >= 10]
    (
        REDITUS[x]|
    )
    ALIUD
    (
        SCRIBERE[a]|
    )
    DUM[y < 10]
    (
        SCRIBERE["Valor de y"]
        SCRIBERE[y]|
        y = y - 1|
    )
)
```

Fonte: Elaborado pelos autores, 2023

6.6 Erros

Durante a análise do código escrito pelo usuário, o compilador possui a capacidade de identificar erros, fornecendo informações detalhadas sobre o tipo de erro ocorrido, como erro léxico, sintático e semântico. Além disso, o compilador indica a linha específica onde o erro foi detectado, facilitando a correção por parte do desenvolvedor.

Essa funcionalidade é crucial para o processo de desenvolvimento de software, pois permite uma abordagem eficiente na depuração do código, garantindo a detecção e resolução rápida de potenciais problemas.

Na Figura 6 a seguir, é apresentado um exemplo de erro léxico que são associados a erros de digitação em comandos, palavras-chave ou símbolos. Neste caso específico, o erro ocorreu na declaração da variável, onde foi escrito de forma errada a palavra-chave do tipo, que deveria ser "LITTERAE”.

Figura 6 - Exemplo de Erro Léxico da Linguagem Rubrum

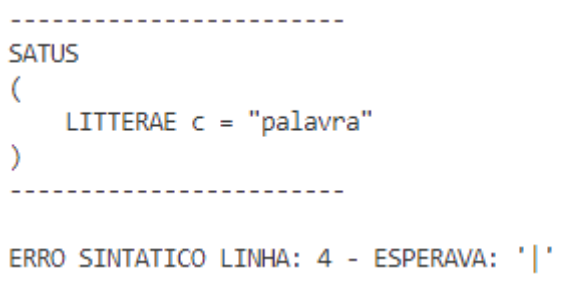
```
------------------------
SATUS
(
    LITTER c = "palavra"|
)
------------------------

ERRO LÉXICO LINHA: 3 - PALAVRA RESERVADA: LITTER
```

Fonte: Elaborado pelos autores, 2023

Na Figura 7 abaixo, é mostrado um exemplo de erro sintático que são relacionados à estrutura do código, como a disposição inadequada dos comandos ou a omissão de parênteses e símbolos essenciais. Neste caso, o erro surgiu devido à ausência do símbolo pipe "|" obrigatório ao final da linha.

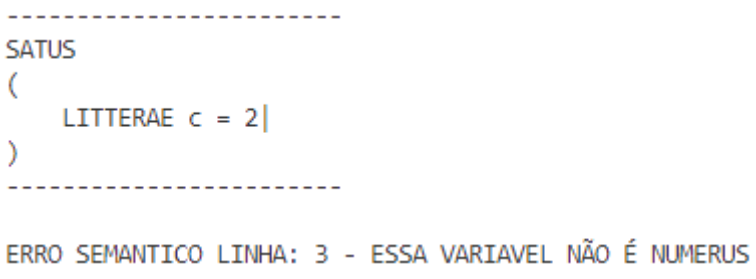
Figura 7 - Exemplo de Erro Sintático da Linguagem Rubrum



Fonte: Elaborado pelos autores, 2023

Na Figura 8 a seguir, é apresentado um exemplo de erro semântico que são associados ao significado do código, como a utilização de uma variável não definida. Neste caso específico, houve a tentativa de atribuir um número a uma variável declarada literal.

Figura 8 - Exemplo de Erro Semântico da Linguagem Rubrum



Fonte: Elaborado pelos autores, 2023

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, A. V. ET AL.. Compiladores: princípios, técnicas e ferramentas. 2. ed. São Paulo: Pearson Addison Wesley, 2008. Acesso em: 12 dez. 2023.

PROFESSOR ISIDRO. Compiladores - Curso Completo. YouTube, 23 de set. de 2020. Disponível em: https://www.youtube.com/watch?v=gxlxHYv-9oo&list=PLjcmNukBom6--0we1zrpoUE2GuRD-Me6W&index=1. Acesso em: 12 dez. 2023.