Documentação do Compilador Coins

Este documento detalha a implementação do compilador para a linguagem "Coins", conforme as especificações do Trabalho Discente Efetivo (TDE) da disciplina de Compiladores.

1. Especificação da Linguagem Coins

A linguagem Coins é uma linguagem procedural simples, criada para fins educacionais, abrangendo conceitos fundamentais de compiladores. Ela suporta declarações de variáveis, atribuições, operações aritméticas, lógicas e de comparação, estruturas de controle de fluxo (condicionais e laços de repetição), definição de procedimentos e funções (com retorno), e comentários de linha única.

1.1. Elementos da Linguagem

Palavras-chave e Tipos de Dados:

- **Tipos:** `inteiro`, `real`, `texto`
- **Controle:** `se`, `senao`, `enquanto`
- **Sub-rotinas:** `procedimento` (sem retorno), `funcao` (com retorno)
- **Retorno:** `retorna`

Operadores:

- **Aritméticos:** `+`, `-`, `*`, `/`, `%`
- **Lógicos:** `&&` (and), `||` (or), `!` (not)
- **Comparação:** `==`, `!=`, `>=`, `<=`, `>`, `<`

Declaração e Atribuição de Variáveis:

Variáveis são declaradas com um tipo seguido pelo nome da variável e um ponto e vírgula. Múltiplas variáveis do mesmo tipo podem ser declaradas na mesma linha, separadas por vírgulas.

Exemplos:

```
 inteiro x;
real valor, preco;
texto nome;
```

Atribuições são realizadas utilizando o operador `=`. O tipo do valor atribuído deve ser compatível com o tipo da variável.

Exemplos:

```
 x = 10;
valor = 3.14;
nome = "João";
```

Estruturas de Controle de Fluxo:

- **Condicional (`se` / `senao`):** Permite a execução condicional de blocos de código.

  Exemplo: `coins se (x > 5) { // comandos se a condição for verdadeira } senao { // comandos se a condição for falsa }`

- **Loop (`enquanto`):** Permite a repetição de um bloco de código enquanto uma condição for verdadeira.

  Exemplo: `coins enquanto (x < 10) { x = x + 1; }`

Sub-rotinas:

- **Procedimentos (`procedimento`):** Blocos de código que não retornam valor.

  Exemplo: `coins procedimento exibirMensagem() { // comandos }`

- **Funções (`funcao`):** Blocos de código que retornam um valor de um tipo específico.

  Exemplo: `coins funcao soma(inteiro a, inteiro b) retorna inteiro { retorna a + b; }`

Comentários:

Comentários de uma linha iniciam com `//` e se estendem até o final da linha.

Exemplo:

```
 // Este é um comentário
inteiro idade; // Declaração de idade
```

2. Implementação do Analisador Léxico

O analisador léxico é responsável por converter o código-fonte em uma sequência de tokens, ignorando espaços em branco e comentários. Ele também gera uma tabela de símbolos em formato HTML.

Código Fonte do Analisador Léxico (`analisador_lexico.py`)

```python


import re
import html


token_specs = [
    ("TIPO", r"\b(inteiro|real|texto)\b"),
    ("SE", r"\bse\b"),
    ("SENAO", r"\bsenao\b"),
    ("ENQUANTO", r"\benquanto\b"),
    ("PROCEDIMENTO", r"\bprocedimento\b"),
    ("FUNCAO", r"\bfuncao\b"),
    ("RETORNA", r"\bretorna\b"),
    ("ID", r"\b[a-zA-Z_][a-zA-Z0-9_]*\b"),
    ("NUMERO", r"\b[0-9]+(?:\.[0-9]+)?\b"),
    ("STRING", r"\"[^\"]*\""),
    ("OP_ARIT", r"[+\-*/%]"),
    ("OP_LOGICO", r"(&&|\|\||!)"),
    ("OP_COMP", r"(==|!=|>=|<=|>|<)"),
    ("IGUAL", r"="),
    ("PONTO_VIRGULA", r";"),
    ("VIRGULA", r","),
    ("ABRE_PAREN", r"\("),
    ("FECHA_PAREN", r"\)"),
    ("ABRE_CHAVE", r"\{"),
    ("FECHA_CHAVE", r"\}"),
    ("COMENTARIO", r"//.*\n"), # Inclui a quebra de linha para
consumir o comentário completo
    ("SKIP", r"[ \t\n]+"),
    ("MISMATCH", r"."),
]

tok_regex = "|".join(f"(?P<{name}>{regex})" for name, regex in
token_specs)

tabela_simbolos = {}

def analise_lexica(codigo):
```

```
    tokens_gerados = []
    for match in re.finditer(tok_regex, codigo):
        tipo = match.lastgroup
        valor = match.group(tipo)

        if tipo == "SKIP" or tipo == "COMENTARIO":
            continue
        elif tipo == "MISMATCH":
            tokens_gerados.append(("ERRO_LEXICO", valor))
        else:
            tokens_gerados.append((tipo, valor))
            if tipo == "ID" and valor not in tabela_simbolos:
                tabela_simbolos[valor] = {"tipo": "indefinido",
"valor": ""}
    return tokens_gerados

def salvar_html():
    with open("tabela_simbolos.html", "w", encoding="utf-8") as
f:
        f.write("<html><head><meta
charset=\'UTF-8\'><title>Tabela de Símbolos</title></
head><body>\n")
        f.write("<h2>Tabela de Símbolos</h2>\n")
        f.write("<table border=\'1\
angle<tr><th>Identificador</th><th>Tipo</th><th>Valor</th></
tr>\n")
        for nome, info in tabela_simbolos.items():
            f.write(f"<tr><td>{html.escape(nome)}</
td><td>{info["tipo"]}</td><td>{html.escape(str(info["valor"]))}
</td></tr>\n")
        f.write("</table></body></html>\n")
    print("Tabela salva em tabela_simbolos.html!")

if __name__ == "__main__":
    try:
        with open("codigo.txt", "r", encoding="utf-8") as file:
            codigo_fonte = file.read()
            if not codigo_fonte.strip():
                print("⚠ O arquivo codigo.txt está vazio!")
            else:
                tokens = analise_lexica(codigo_fonte)
                print("Tokens gerados:")
                for token in tokens:
                    print(token)
                salvar_html()
    except FileNotFoundError:
        print("❌ Arquivo codigo.txt não encontrado!")
```

Exemplo de Tokens Gerados

Para o código de exemplo `codigo.txt`:

```
 inteiro x, y;
y = 10 + 10;
x = 5 + 5;
real z;
z = 10.1;
texto nome;
nome = "João";
```

O analisador léxico gera a seguinte sequência de tokens:

```
 ("TIPO", "inteiro")
("ID", "x")
("VIRGULA", ",")
("ID", "y")
("PONTO_VIRGULA", ";")
("ID", "y")
("IGUAL", "=")
("NUMERO", "10")
("OP_ARIT", "+")
("NUMERO", "10")
("PONTO_VIRGULA", ";")
("ID", "x")
("IGUAL", "=")
("NUMERO", "5")
("OP_ARIT", "+")
("NUMERO", "5")
("PONTO_VIRGULA", ";")
("TIPO", "real")
("ID", "z")
("PONTO_VIRGULA", ";")
("ID", "z")
("IGUAL", "=")
("NUMERO", "10.1")
("PONTO_VIRGULA", ";")
("TIPO", "texto")
("ID", "nome")
("PONTO_VIRGULA", ";")
("ID", "nome")
("IGUAL", "=")
("STRING", "\"João\"")
("PONTO_VIRGULA", ";")
```

3. Implementação do Analisador Sintático e Semântico

O analisador sintático verifica se a sequência de tokens está de acordo com a gramática da linguagem e constrói uma Árvore Sintática Abstrata (AST). O analisador semântico, integrado ao sintático, realiza a verificação de tipos e escopo.

Código Fonte do Analisador Sintático (`analisador_sintatico.py`)

```
import json
from analisador_lexico import analise_lexica, tabela_simbolos,
salvar_html
from gerador_codigo import CodeGenerator

class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.current_token_index = 0
        self.current_token =
self.tokens[self.current_token_index] if self.tokens else None
        self.ast = {"type": "Programa", "body": []}
        self.scope_stack = [{}]

    def advance(self):
        self.current_token_index += 1
        if self.current_token_index < len(self.tokens):
            self.current_token =
self.tokens[self.current_token_index]
        else:
            self.current_token = None

    def match(self, expected_type):
        if self.current_token and self.current_token[0] ==
expected_type:
            value = self.current_token[1]
            self.advance()
            return value
        else:
            self.error(f"Erro de sintaxe: Esperado
{expected_type}, encontrado {self.current_token[0] if
self.current_token else 'EOF'}")

    def error(self, message):
        raise Exception(message)

    def enter_scope(self):
        self.scope_stack.append({})

    def exit_scope(self):
        self.scope_stack.pop()
```

```
    def declare_variable(self, name, var_type):
        current_scope = self.scope_stack[-1]
        if name in current_scope:
            self.error(f"Erro semântico: Variável '{name}' já
declarada neste escopo.")
        current_scope[name] = {"type": var_type}
        tabela_simbolos[name] = {"tipo": var_type, "valor": ""}

    def get_variable_type(self, name):
        for scope in reversed(self.scope_stack):
            if name in scope:
                return scope[name]["type"]
        self.error(f"Erro semântico: Variável '{name}' não
declarada.")

    def check_type_compatibility(self, expected_type,
actual_type, operation="atribuição"):
        if expected_type == "inteiro" and actual_type == "real":

print(f"Aviso semântico: Conversão implícita de real para
inteiro em {operation}. Pode haver perda de dados.")
            return True
        elif expected_type == "real" and actual_type ==
"inteiro":
            return True
        elif expected_type == actual_type:
            return True
        else:
            self.error(f"Erro semântico: Incompatibilidade de
tipos em {operation}. Esperado {expected_type}, encontrado
{actual_type}.")

    def parse(self):
        self.programa()
        return self.ast

    def programa(self):
        self.declaracoes()
        self.comandos()

    def declaracoes(self):
        while self.current_token and self.current_token[0] ==
"TIPO":
            node = {"type": "Declaracao", "declarations": []}
            var_type = self.match("TIPO")
            while True:
                var_name = self.match("ID")
                self.declare_variable(var_name, var_type)
                node["declarations"].append({"name": var_name,
"type": var_type})
                if self.current_token and self.current_token[0]
```

```
== "VIRGULA":
                    self.match("VIRGULA")
                else:
                    break
            self.match("PONTO_VIRGULA")
            self.ast["body"].append(node)

    def comandos(self):
        while self.current_token and self.current_token[0] in
["ID", "SE", "ENQUANTO", "PROCEDIMENTO", "FUNCAO"]:
            if self.current_token[0] == "ID":
                self.atribuicao()
            elif self.current_token[0] == "SE" or
self.current_token[0] == "ENQUANTO":
                self.estrutura_controle()
            elif self.current_token and self.current_token[0] in
["PROCEDIMENTO", "FUNCAO"]:
                self.chamada_subrotina()

    def atribuicao(self):
        node = {"type": "Atribuicao"}
        var_name = self.match("ID")
        node["variable"] = var_name
        expected_type = self.get_variable_type(var_name)
        self.match("IGUAL")
        value_node = self.expressao()
        actual_type = self.get_expression_type(value_node)
        self.check_type_compatibility(expected_type,
actual_type)
        node["value"] = value_node
        self.match("PONTO_VIRGULA")
        self.ast["body"].append(node)

    def estrutura_controle(self):
        if self.current_token and self.current_token[0] == "SE":
            self.condicional()
        elif self.current_token and self.current_token[0] ==
"ENQUANTO":
            self.repeticao()
        else:
            self.error("Erro de sintaxe: Esperado 'se' ou
'enquanto'")

    def condicional(self):
        node = {"type": "Condicional"}
        self.match("SE")
        self.match("ABRE_PAREN")
        condition_node = self.expressao()
        condition_type =
self.get_expression_type(condition_node)
        if condition_type not in ["inteiro", "real", "boolean"]
and not (condition_node["type"] == "BinaryExpression" and
```

```
condition_node["operator"] in ["==", "!=", ">", "<", ">=", "<=",
"&&", "||", "!"]):
            print(f"Aviso semântico: Condição de tipo
inesperado: {condition_type}. Esperado tipo booleano ou
numérico.")
        node["condition"] = condition_node
        self.match("FECHA_PAREN")
        self.match("ABRE_CHAVE")
        self.enter_scope()
        node["consequent"] = []
        original_body = self.ast["body"]
        self.ast["body"] = node["consequent"]
        self.comandos()
        self.ast["body"] = original_body
        self.exit_scope()
        self.match("FECHA_CHAVE")
        if self.current_token and self.current_token[0] ==
"SENAO":
            self.match("SENAO")
            self.match("ABRE_CHAVE")
            self.enter_scope()
            node["alternate"] = []
            original_body = self.ast["body"]
            self.ast["body"] = node["alternate"]
            self.comandos()
            self.ast["body"] = original_body
            self.exit_scope()
            self.match("FECHA_CHAVE")
        self.ast["body"].append(node)

    def repeticao(self):
        node = {"type": "Repeticao"}
        self.match("ENQUANTO")
        self.match("ABRE_PAREN")
        condition_node = self.expressao()
        condition_type =
self.get_expression_type(condition_node)
        if condition_type not in ["inteiro", "real", "boolean"]
and not (condition_node["type"] == "BinaryExpression" and
condition_node["operator"] in ["==", "!=", ">", "<", ">=", "<=",
"&&", "||", "!"]):
            print(f"Aviso semântico: Condição de tipo
inesperado: {condition_type}. Esperado tipo booleano ou
numérico.")
        node["condition"] = condition_node
        self.match("FECHA_PAREN")
        self.match("ABRE_CHAVE")
        self.enter_scope()
        node["body"] = []
        original_body = self.ast["body"]
        self.ast["body"] = node["body"]
        self.comandos()
```

```
        self.ast["body"] = original_body
        self.exit_scope()
        self.match("FECHA_CHAVE")
        self.ast["body"].append(node)

    def expressao(self):
        node = self.termo()
        while self.current_token and self.current_token[0] in
["OP_ARIT", "OP_LOGICO", "OP_COMP"]:
            operator = self.match(self.current_token[0])
            right = self.termo()
            left_type = self.get_expression_type(node)
            right_type = self.get_expression_type(right)
            result_type = self.infer_type(left_type, right_type,
operator)
            node = {"type": "BinaryExpression", "operator":
operator, "left": node, "right": right, "_type": result_type}
        return node

    def termo(self):
        node = self.fator()
        while self.current_token and self.current_token[0] in
["OP_ARIT", "OP_LOGICO", "OP_COMP"]:
            operator = self.match(self.current_token[0])
            right = self.fator()
            left_type = self.get_expression_type(node)
            right_type = self.get_expression_type(right)
            result_type = self.infer_type(left_type, right_type,
operator)
            node = {"type": "BinaryExpression", "operator":
operator, "left": node, "right": right, "_type": result_type}
        return node

    def fator(self):
        if self.current_token and self.current_token[0] ==
"NUMERO":
            value = self.match("NUMERO")
            if "." in value:
                return {"type": "Literal", "value": value,
"_type": "real"}
            else:
                return {"type": "Literal", "value": value,
"_type": "inteiro"}
        elif self.current_token and self.current_token[0] ==
"ID":
            value = self.match("ID")
            var_type = self.get_variable_type(value)
            return {"type": "Identifier", "name": value,
"_type": var_type}
        elif self.current_token and self.current_token[0] ==
"STRING":
            value = self.match("STRING")
```

```
            return {"type": "Literal", "value": value, "_type":
"texto"}
        elif self.current_token and self.current_token[0] ==
"ABRE_PAREN":
            self.match("ABRE_PAREN")
            node = self.expressao()
            self.match("FECHA_PAREN")
            return node
        else:
            self.error(f"Erro de sintaxe: Esperado NUMERO, ID,
STRING ou ABRE_PAREN, encontrado {self.current_token[0] if
self.current_token else 'EOF'}")

    def get_expression_type(self, node):
        if "_type" in node:
            return node["_type"]
        elif node["type"] == "Identifier":
            return self.get_variable_type(node["name"])
        elif node["type"] == "Literal":
            if isinstance(node["value"], str) and
node["value"].startswith("\""):
                return "texto"
            elif "." in str(node["value"]):
                return "real"
            else:
                return "inteiro"
        elif node["type"] == "BinaryExpression":
            left_type = self.get_expression_type(node["left"])
            right_type = self.get_expression_type(node["right"])
            return self.infer_type(left_type, right_type,
node["operator"])
        elif node["type"] == "ChamadaSubrotina":
            # For now, assume procedures return void and
functions return their declared type
            # This needs to be properly handled when function
declarations are parsed
            return "void" # Placeholder
        return "unknown"

    def infer_type(self, type1, type2, operator):
        if operator in ["+", "-", "*", "/"]:
            if type1 == "texto" or type2 == "texto":

self.error(f"Erro semântico: Operação aritmética com tipo texto
não permitida: {type1} {operator} {type2}")
            if type1 == "real" or type2 == "real":
                return "real"
            return "inteiro"
        elif operator == "%" :
            if type1 != "inteiro" or type2 != "inteiro":
                self.error(f"Erro semântico: Operador de módulo
(%) só pode ser usado com inteiros: {type1} {operator} {type2}")
```

```
            return "inteiro"
        elif operator in ["&&", "||", "!"]:
            if type1 not in ["inteiro", "real", "boolean"] or
type2 not in ["inteiro", "real", "boolean"]:

self.error(f"Erro semântico: Operação lógica com tipo não
booleano/numérico não permitida: {type1} {operator} {type2}")
            return "boolean"
        elif operator in ["==", "!=", ">", "<", ">=", "<="]:
            if type1 == "texto" and type2 == "texto":
                return "boolean"
            if type1 in ["inteiro", "real"] and type2 in
["inteiro", "real"]:
                return "boolean"
            self.error(f"Erro semântico: Comparação de tipos
incompatíveis: {type1} {operator} {type2}")
        return "unknown"

    def chamada_subrotina(self):
        node = {"type": "ChamadaSubrotina"}
        subroutine_type_token = self.current_token[0]
        if subroutine_type_token == "PROCEDIMENTO":
            node["subroutine_type"] = self.match("PROCEDIMENTO")
        elif subroutine_type_token == "FUNCAO":
            node["subroutine_type"] = self.match("FUNCAO")
        else:

self.error("Erro de sintaxe: Esperado 'procedimento' ou
'funcao'")

        node["name"] = self.match("ID")

        if self.current_token and self.current_token[0] ==
"ABRE_PAREN":
            self.match("ABRE_PAREN")
            node["arguments"] = self.argumentos()
            self.match("FECHA_PAREN")
        else:
            node["arguments"] = []

        self.match("PONTO_VIRGULA")
        self.ast["body"].append(node)

    def argumentos(self):
        args = []
        if self.current_token and self.current_token[0] !=
"FECHA_PAREN":
            args.append(self.expressao())
            while self.current_token and self.current_token[0]
== "VIRGULA":
                self.match("VIRGULA")
                args.append(self.expressao())
```

```
        return args


if __name__ == "__main__":
    try:
        with open("codigo.txt", "r", encoding="utf-8") as file:
            codigo_fonte = file.read()
            if not codigo_fonte.strip():
                print("⚠ O arquivo codigo.txt está vazio!")
            else:
                tokens = analise_lexica(codigo_fonte)
                print("Tokens gerados:")
                for token in tokens:
                    print(token)

                parser = Parser(tokens)
                ast = parser.parse()

                with open("ast.json", "w", encoding="utf-8") as
ast_file:
                    json.dump(ast, ast_file, indent=4)
                print("AST salva em ast.json!")

                generator = CodeGenerator(ast)
                generated_code = generator.generate()
                with open("codigo_gerado.py", "w",
encoding="utf-8") as code_file:
                    code_file.write(generated_code)
                print("Código Python gerado e salvo em
codigo_gerado.py!")

                salvar_html()

    except FileNotFoundError:
        print("❌ Arquivo codigo.txt não encontrado!")
    except Exception as e:
        print(f"❌ Erro durante a análise sintática/semântica:
{e}")
```

Exemplo de AST Gerada

Para o código de exemplo `codigo.txt`:

```
 inteiro x, y;
y = 10 + 10;
x = 5 + 5;
real z;
z = 10.1;
```

```
texto nome;
nome = "João";
```

A AST gerada (`ast.json`) é a seguinte:


```
{
    "type": "Programa",
    "body": [
        {
            "type": "Declaracao",
            "declarations": [
                {
                    "name": "x",
                    "type": "inteiro"
                },
                {
                    "name": "y",
                    "type": "inteiro"
                }
            ]
        },
        {
            "type": "Atribuicao",
            "variable": "y",
            "value": {
                "type": "BinaryExpression",
                "operator": "+",
                "left": {
                    "type": "Literal",
                    "value": "10",
                    "_type": "inteiro"
                },
                "right": {
                    "type": "Literal",
                    "value": "10",
                    "_type": "inteiro"
                },
                "_type": "inteiro"
            }
        },
        {
            "type": "Atribuicao",
            "variable": "x",
            "value": {
                "type": "BinaryExpression",
                "operator": "+",
                "left": {
                    "type": "Literal",
                    "value": "5",
                    "_type": "inteiro"
```

```
                },
                "right": {
                    "type": "Literal",
                    "value": "5",
                    "_type": "inteiro"
                },
                "_type": "inteiro"
            }
        }
    ]
}
```

4. Implementação do Gerador de Código

O gerador de código traduz a AST para um código Python equivalente, que pode ser executado diretamente.

Código Fonte do Gerador de Código (`gerador_codigo.py`)

```
class CodeGenerator:
    def __init__(self, ast):
        self.ast = ast
        self.code = []
        self.indent_level = 0

    def generate(self):
        self.visit(self.ast)
        return "\n".join(self.code)

    def indent(self):
        return "    " * self.indent_level

    def visit(self, node):
        method_name = f"visit_{node["type"]}"
        visitor = getattr(self, method_name, self.generic_visit)
        visitor(node)

    def generic_visit(self, node):
        raise Exception(f"Nenhum método visit_{node["type"]}
implementado.")

    def visit_Programa(self, node):
        for child_node in node["body"]:
            self.visit(child_node)

    def visit_Declaracao(self, node):
        # Declarações em Python não precisam de tipo explícito
        # Apenas inicializamos as variáveis para garantir que
```

```
existam
        for declaration in node["declarations"]:
            var_name = declaration["name"]
            var_type = declaration["type"]
            if var_type == "inteiro":
                self.code.append(f"{self.indent()}{var_name} =
0")
            elif var_type == "real":
                self.code.append(f"{self.indent()}{var_name} =
0.0")
            elif var_type == "texto":
                self.code.append(f"{self.indent()}{var_name} =
\"\"")

    def visit_Atribuicao(self, node):
        var_name = node["variable"]
        value = self.visit_expression(node["value"])
        self.code.append(f"{self.indent()}{var_name} = {value}")

    def visit_BinaryExpression(self, node):
        left = self.visit_expression(node["left"])
        right = self.visit_expression(node["right"])
        operator = node["operator"]
        return f"({left} {operator} {right})"

    def visit_Literal(self, node):
        if node["_type"] == "texto":
            return f"\"" + node["value"].replace("\"", "") +
f"\"" # Remove aspas extras se já houver
        return str(node["value"])

    def visit_Identifier(self, node):
        return node["name"]

    def visit_Condicional(self, node):
        condition = self.visit_expression(node["condition"])
        self.code.append(f"{self.indent()}if {condition}:")
        self.indent_level += 1
        for consequent_node in node["consequent"]:
            self.visit(consequent_node)
        self.indent_level -= 1
        if "alternate" in node:
            self.code.append(f"{self.indent()}else:")
            self.indent_level += 1
            for alternate_node in node["alternate"]:
                self.visit(alternate_node)
            self.indent_level -= 1

    def visit_Repeticao(self, node):
        condition = self.visit_expression(node["condition"])
        self.code.append(f"{self.indent()}while {condition}:")
        self.indent_level += 1
```

```
        for body_node in node["body"]:
            self.visit(body_node)
        self.indent_level -= 1

    def visit_ChamadaSubrotina(self, node):
        func_name = node["name"]
        args = ", ".join([self.visit_expression(arg) for arg in
node["arguments"]])
        self.code.append(f"{self.indent()}{func_name}({args})")

    def visit_expression(self, node):
        if node["type"] == "BinaryExpression":
            return self.visit_BinaryExpression(node)
        elif node["type"] == "Literal":
            return self.visit_Literal(node)
        elif node["type"] == "Identifier":
            return self.visit_Identifier(node)
        else:
            self.error(f"Tipo de expressão desconhecido:
{node["type"]}")
```

Exemplo de Código Python Gerado

Para o código de exemplo `codigo.txt`:

```
inteiro x, y;
y = 10 + 10;
x = 5 + 5;
real z;
z = 10.1;
texto nome;
nome = "João";
```

O código Python gerado (`codigo_gerado.py`) é o seguinte:

```
x = 0
y = 0
y = (10 + 10)
x = (5 + 5)



```python
x = 0
y = 0
z = 0.0
nome = ""
y = (10 + 10)
```
```

```
x = (5 + 5)
z = 10.1
nome = "João"
```

5. Testes e Exemplos

Para testar o compilador, execute o arquivo `analisador_sintatico.py`. Ele irá ler o `codigo.txt`, gerar a AST, realizar a análise semântica, gerar o `codigo_gerado.py` e atualizar a `tabela_simbolos.html`.

Para executar o código gerado, basta rodar o arquivo `codigo_gerado.py` com um interpretador Python.

6. Conclusão

Este projeto demonstra as fases iniciais de um compilador para a linguagem "Coins", incluindo análise léxica, sintática, semântica e geração de código. As ferramentas e técnicas utilizadas são fundamentais para a compreensão do funcionamento de compiladores e processadores de linguagem.