Ministério da Educação

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Campus Toledo

COENC - ENGENHARIA DE COMPUTAÇÃO

CURSO DE COMPILADORES

EDUARDO BIF PITOL

GUSTAVO TREVIZAN

GABRIELA RAMOS

DOCUMENTAÇÃO DE ESPECIFICAÇÃO AO DE REQUISITOS: COMPILADOR MORDOMO

TOLEDO

2023

EDUARDO BIF PITOL
GUSTAVO TREVIZAN
GABRIELA RAMOS

DOCUMENTAÇÃO DE ESPECIFICAÇÃO AO DE REQUISITOS: COMPILADOR MORDOMO

Documentação do Compilador apresentado ao Curso de Compiladores da Universidade Tecnológica Federal do Paraná.
Orientador: Leandro Augusto Ensina Universidade Tecnológica Federal do Paraná.

TOLEDO
2023

SUMÁRIO

1. INTRODUÇÃO

A construção de um compilador é um processo essencial para a criação de software. Um compilador é uma ferramenta poderosa que traduz o código-fonte de um programa em uma linguagem de programação para um código executável em outra linguagem, geralmente em linguagem de máquina. Essa transformação permite que os programas sejam executados diretamente pelo hardware do computador.

A construção de um compilador envolve várias etapas complexas, que exigem um profundo conhecimento teórico e prático da linguagem de programação em questão, bem como dos princípios subjacentes à compilação.

O presente trabalho tem por objetivo a elaboração de um compilador chamado Mordomo, cujas características da linguagem suportada estão fortemente atreladas a falas típicas de um mordomo muito educado. Além disso, a sua gramática é inspirada em alguns aspectos da linguagem Python e C++.

1.1 CONTEXTUALIZAÇÃO

A construção de um compilador requer habilidades em áreas como teoria da computação, linguagens formais, algoritmos e estruturas de dados. É um processo desafiador, mas extremamente gratificante, pois permite a criação de software eficiente e robusto, que pode ser executado em uma ampla gama de plataformas e arquiteturas.

O processo começa com a análise léxica, na qual o código-fonte é dividido em tokens significativos, como palavras-chave, identificadores e símbolos especiais.

Em seguida, ocorre a análise sintática, na qual a estrutura gramatical do código é verificada para garantir que esteja de acordo com a sintaxe definida pela linguagem. Essa etapa é fundamental para garantir a correção do programa e evitar erros de sintaxe.

Após a análise sintática, é realizada a análise semântica, na qual são verificadas as regras de semântica da linguagem. Isso inclui a verificação de tipos de dados, escopo de variáveis e outras restrições impostas pela linguagem.

Uma vez que todas as análises tenham sido concluídas com sucesso, o compilador gera o código intermediário ou otimizado, que é uma representação do programa em uma forma mais próxima da linguagem de máquina, mas ainda em um formato legível para humanos.

O Mordomo faz tudo isso de forma intuitiva e modesta, passando pelos mais diversos desafios da construção do compilador até o momento. Ele foi composto por várias fases, incluindo análise léxica, análise sintática, análise semântica, geração de código intermediário e geração de código de máquina. Essas fases trabalham juntas de forma harmoniosa para transformar o código-fonte em um código executável.

1.2 DEFINIÇÕES DE CONCEITOS

Definição de conceitos importantes quando se trata de um compilador, sendo eles teóricos mas que possuem aplicações reais em diversas partes do compilador. Abaixo, se encontram os conceitos e suas respectivas funcionalidades:

- Análise: A análise é o processo de examinar um programa fonte para identificar os elementos, estrutura e propriedades semânticas. É uma etapa crucial no processo de compilação, onde o código fonte é examinado para garantir sua corretude e conformidade com a linguagem de programação utilizada.
- Autômato: Um autômato é um modelo matemático utilizado na análise léxica para reconhecer padrões em sequências de símbolos. Ele é uma representação abstrata de uma máquina que pode estar em diferentes estados e transitar entre eles de acordo com a entrada recebida.
- Gramática: A gramática é um conjunto de regras que define a estrutura sintática de uma linguagem. Ela descreve como as diferentes partes do código devem ser organizadas e quais construções são válidas na linguagem. As regras gramaticais especificam como os símbolos podem ser combinados para formar expressões, comandos e outros elementos da linguagem.
- Lexema: Um lexema é uma sequência de caracteres que representa uma unidade léxica reconhecida pelo analisador léxico. Por exemplo, em uma linguagem de programação, um lexema pode ser uma palavra-chave (como "if" ou "while"), um identificador (nome de variável), um número, um operador ou um símbolo especial.
- Lexer: Um lexer, também conhecido como analisador léxico ou scanner, é responsável por dividir o código fonte em lexemas e identificar os tokens correspondentes. Ele percorre o código fonte caractere por caractere, reconhecendo e agrupando os lexemas de acordo com as regras da linguagem.
- Parser: Um parser, também chamado de analisador sintático, é responsável por verificar a estrutura gramatical do programa fonte. Ele utiliza as regras da gramática

da linguagem para analisar a sequência de tokens produzidos pelo analisador léxico e construir uma árvore sintática que representa a estrutura hierárquica do código.

- Síntese: A síntese é a etapa do compilador que converte o código fonte em uma representação intermediária ou código objeto. Nessa fase, as estruturas sintáticas e semânticas do programa são processadas e transformadas em uma forma que possa ser executada pelo computador.
- Tabela de Símbolos: A tabela de símbolos é uma estrutura de dados utilizada para armazenar informações sobre identificadores encontrados no programa fonte. Ela é criada durante a análise léxica e pode ser reutilizada nas etapas subsequentes do compilador. A tabela de símbolos registra informações como nomes de variáveis, tipos de dados, escopo e outras propriedades relacionadas aos identificadores.
- Tokens: Os tokens são unidades léxicas reconhecidas pelo analisador léxico. Eles representam elementos individuais do código fonte, como palavras-chave, operadores, delimitadores, identificadores, números e literais. Os tokens são utilizados pelo analisador sintático para construir a estrutura do programa e realizar a análise gramatical.

2. DESCRIÇÃO DO CÓDIGO

Seção dedicada para o entendimento do código em suas funções básicas. De simples entendimento mas necessária para que seja possível escrever códigos de acordo com o padrão estabelecido no projeto, o qual deve ser seguido para que a compilação seja executada de forma esperada.

2.1 ESTRUTURA BÁSICA E CARACTERÍSTICAS DA LINGUAGEM

2.1.1 Operações matemáticas:

Exemplos onde uma variável recebe a operação entre duas variáveis ou um inteiro.

Adição:

```
x = a + 2
```

Subtração:

```
x = 4 - b
```

Divisão:

```
x = 10 / 2
```

Multiplicação:

```
x = a * b
```

Resto:

```
x = a % b
```

2.1.2 Comparações matemáticas:

Operações utilizadas em estrutura de controles em que compara uma variável com outra variável ou atributo, utilizada nas estruturas de controles.
Igualdade:

```
x == “hello”
```

Maior ou igual:

```
x >= 10
```

Menor ou igual:

```
x <= z
```

Maior:

```
x > 0
```

Menor:

```
x < 5
```

Diferença:

```
x != y
```

2.1.3 Definição de variável:

Mantém as palavras reservadas de tipos de variáveis inalteradas. Porém, na linguagem ‘Mordomo’ é obrigatório a inicialização da variável:

Tipo inteiro:

```
int x = 10
```

Tipo ponto flutuante:

```
float y = 1.5
```

Tipo linha de caracteres:

```
string w = "hello"
```

Tipo caracter:

```
char z = 'a'
```

Tipo booleano:

```
"bool a = True"
```

2.1.4 Definição de função:

Utiliza-se a palavra reservada ‘def’, como em python, para indicar criação de uma função, pode ou não retornar algo:

```
def exemplo1(){
  #code
}
```

Ou

```
def exemplo2(int x, string y){
    #code
    return x
}
```

2.1.5 Estruturas de controles:

Exemplo de usos de estrutura de controles:

if:
Utiliza-se a palavra reservada ‘senhor, caso’ seguida de uma comparação

```
senhor, caso x > y {
    #code
}
```

else:
Utiliza-se a palavra reservada ‘senao, receio que’.

```
senao, receio que{
     #code
}
```

while:
Utiliza-se a palavra reservada ‘durante tal ordem, ’ seguido de uma comparação

```
durante tal ordem, a != b{
    #code
}
```

for:
Utiliza-se a palavra reservada ‘senhor, voce tem das’ e a palavra ‘ate’ junto com inteiros para determinar o loop.

```
senhor, voce tem das 1 ate 5{
}
```

3. REQUISITOS ESPECÍFICOS

Importante citar os requisitos mínimos para que o compilador possa ser executado em uma máquina. Esses requisitos englobam tanto aspectos de hardware quanto de software. Em relação ao hardware, é necessário que a máquina possua uma capacidade de processamento adequada para executar as operações exigidas pelo compilador, bem como memória suficiente para armazenar o programa e os dados temporários durante o processo de compilação.

Todas as etapas do compilador “Mordomo” foram desenvolvidas totalmente baseadas na linguagem de programação *Python,* linguagem gratuita e aberta ao público, de sintaxe simplificada, onde a maior necessidade de atenção se dá indentação, visto que a mesma é crucial para reconhecimento das funções necessárias para o desenvolvimento. Entretanto, o

projeto poderia ter sido realizado baseado em linguagens como: C++, Java, PHP, entre outras, fica a critério do desenvolvedor o que é mais confortável e benéfico para o projeto.

3.1 FUNCIONALIDADE

O compilador desenvolvido recebe um código fonte escrito na linguagem “Mordomo” e, a partir disso, inicia o seu processo de compilação. A partir de sua leitura, suas informações relevantes são captadas e armazenadas, as quais serão usadas futuramente durante as análises respectivas. Em cada processo do compilador é possível obter um retorno de saída para acompanhar o processo, e assim, se aprofundar em seu entendimento e processamento caso seja necessário. A partir de tudo isso, o código é totalmente convertido para que seja entendível pelo compilador e, finalmente, retornando ao utilizador erros(caso existam).

3.2 DESEMPENHO

Desenvolvida de maneira focada na sua simplicidade e entendimento, seus processos são realizados via funções, as quais transitam entre si para realizar todas as verificações. Sua complexidade depende do tamanho máximo do código fonte, a qual pode ser definida como $O(n^2)$, visto que são realizadas diversas checagens ao longo do programa, praticamente a cada token analisado.

4. FORMALIZAÇÃO DA LINGUAGEM

A seguir, alguns quesitos formais sobre a linguagem.

4.1 BNF

BNF (Backus-Naur Form) é uma notação formal usada para descrever a sintaxe de uma linguagem de programação ou de uma gramática formal. É uma forma de representação gramatical que define as regras de produção da linguagem de forma recursiva. A BNF usa símbolos não terminais, que representam elementos da linguagem, e símbolos terminais, que são as unidades básicas da linguagem, como palavras-chave e operadores. As regras de produção especificam como os símbolos não terminais podem ser combinados para formar expressões válidas na linguagem.

4.2 EBNF

EBNF (Extended Backus-Naur Form) é uma extensão da notação BNF (Backus-Naur Form) que oferece recursos adicionais para descrever de forma mais expressiva a sintaxe de uma linguagem. A EBNF permite o uso de construções como repetições, opções e grupos, facilitando a representação de estruturas gramaticais mais complexas. Além disso, a EBNF também permite o uso de metassímbolos para representar elementos que não são caracteres literais, como símbolos especiais e sequências de caracteres. Abaixo se encontra a EBNF base para produção de toda a gramática usada pelo compilador:

REGRAS DE PRODUÇÃO	
<S>	<define_var> <S> | <func> {S}
<expressao>	<operaçao> {<expressao>} | <define_var> {<expressao>} | <estrut_control> {<expressao>} | return_struct
<estrut_control>	<if> | <for> | <while> | <else>
<operaçao>	<operaçao_atr> | <operaçao_comp>
<operaçao_atr>	<var> = <int> <operator> <var> | <var> = <var> <operator> <int> | <var> = <int> <operator> <int> | <var> = <var> <operator> <var> | <var> = <var> | <var> = <atr>
<operaçao_comp>	<var> <comparaçao> <atr> | <var> <comparaçao> <var>
<define_var>	<type> <var> = <var> | <type> <var> = <atr>
<var>	<letra> <var_name_final>
<var_name_next>	<letra> {<var_name_next>} | <int> {<var_name_next>}
<atr>	<char> | <int> | <string_format> | <bool> | <float_format>
<int>	<digito> {<int>}
<string_format>	<aspas> <string> <aspas>
<string>	<letra> {<string>} | <digito> {<string>}
<float_format>	<int>.<int>
<func>	<def> <var> <parent_op> <escopo> <parent_op> <chaves_op> <expressao> <chaves_cls>
<escopo>	{<type> <var> {<virgula> <escopo>}}

<return_struct>	<return> <atr> | <return> <var>
<if>	<if_word> <operaçao_comp> <chaves_op> <expressao> <chaves_cls>
<else>	<else_word> <chaves_op> <expressao> <chaves_cls>
<for>	<for_word> <var> <for_word_ate> <int> <chaves_op> <expressao> <chaves_cls> | <for_word> <var> <for_word_ate> <var> <chaves_op>< expressao> <chaves_cls>
<while>	<while_word> <operaçao_comp> <chaves_op> <expressao> <chaves_cls>
<type>	int | float | char | bool | string
<char>	<aspas_simples> <letra> <aspas_simples>
Terminais	
<comparaçao>	== | != | <= | >= | < | >
<letra>	a | b ... y | z | A | B ... Y | Z
<operator>	+ | - | / | * | % | =
<digito>	0 | 1 | 2 ... 8 | 9
<bool>	True | False
<chaves_op>	{
<chaves_cls>	}
<parent_op>	(
<parent_cls>	)
<virgula>	,
<return>	return
<if_word>	senhor, caso
<else_word>	senao, receio que
<for_word>	senhor, voce tem das
<for_word_ate>	ate
<while_word>	durante tal ordem,
<def>	def

<aspas>	"
<aspas_simples>	'

4.3 TABELA DE TOKENS

Formalidade para conversão do código fonte para tokens, os quais poderão ser lidos pelo compilador e futuramente, interpretados durante os processos de análise. Abaixo encontra-se uma tabela para cada tipo de variável, onde cada linha é composta pelo seu tipo, gramática de produção e token, respectivamente:

Descrição	Cadeia	TOKEN
Nome de variável	(a-z, A-Z)(a-z, A-Z, 0-9)*	TK.ID
Número	(0-9)*	TK.INT
Caracter	‘(a-z, A-Z)’	TK.CHAR
linha de caracteres	“ASCII”	TK.STRING
Número decimal	(0-9)+.(0-9)+	TK.FLOAT
booleano	True|False	TK.BOOL
Abre chaves	{	TK.CHAVES_op
Fecha chaves	}	TK.CHAVES_cls
Abre parênteses	(	TK.PARENTESES_op
Fecha parênteses	)	TK.PARENTESES_cls
Operadores matemáticos	- | + | * | / | % | =	TK.OPERADOR
Comparadores matemático	==|!=|<=|>=|<|>	TK.COMPARADOR
Caractere de comentário single line	#	TK.COMMENT
Caracter ‘,’	,	TK.VIRGULA

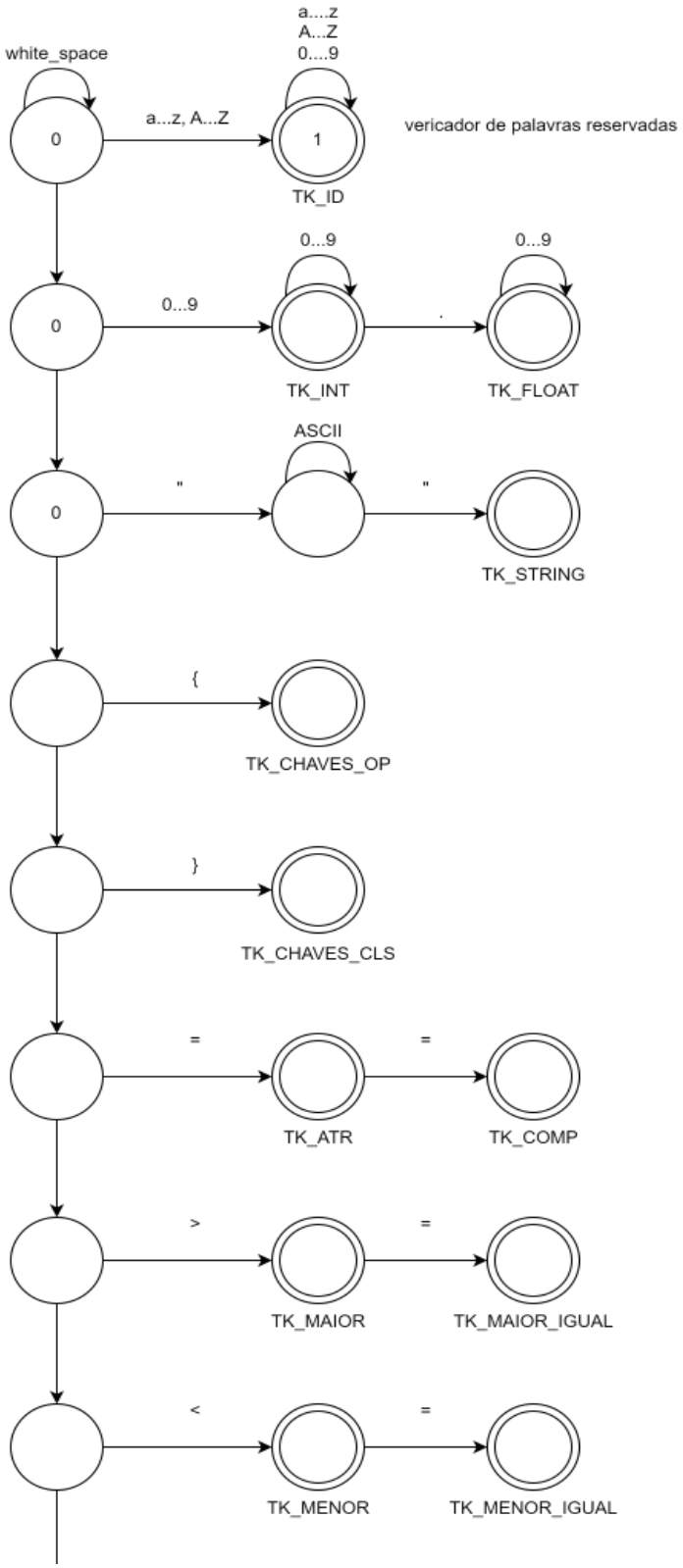
Tabela de palavras reservadas;

Descrição	Cadeia	TOKEN

Tipo - int	int	TK.KEYWORD
Tipo - char	char	TK.KEYWORD
Tipo - float	float	TK.KEYWORD
Tipo - string	string	TK.KEYWORD
Tipo - bool	bool	TK.KEYWORD
if	senhor, caso	TK.KEYWORD
else	senao, receio que	TK.KEYWORD
while	durante tal ordem	TK.KEYWORD
for	senhor, você tem das	TK.KEYWORD
dita até onde vai o loop do for	até	TK.KEYWORD
retorno	return	TK.KEYWORD
Definição de criação de função	def	TK.KEYWORD

4.4 DIAGRAMA DE ESTADOS

Autômato de produção para os tokens da linguagem:


a....z
A...Z
0....9
white_space
0
a...z, A...Z
1
vericador de palavras reservadas
TK_ID
0...9
0...9
0
0...9
.
TK_INT
TK_FLOAT
ASCII
0
"
"
TK_STRING
{
TK_CHAVES_OP
}
TK_CHAVES_CLS
=
=
TK_ATR
TK_COMP
>
=
TK_MAIOR
TK_MAIOR_IGUAL
<
=
TK_MENOR
TK_MENOR_IGUAL

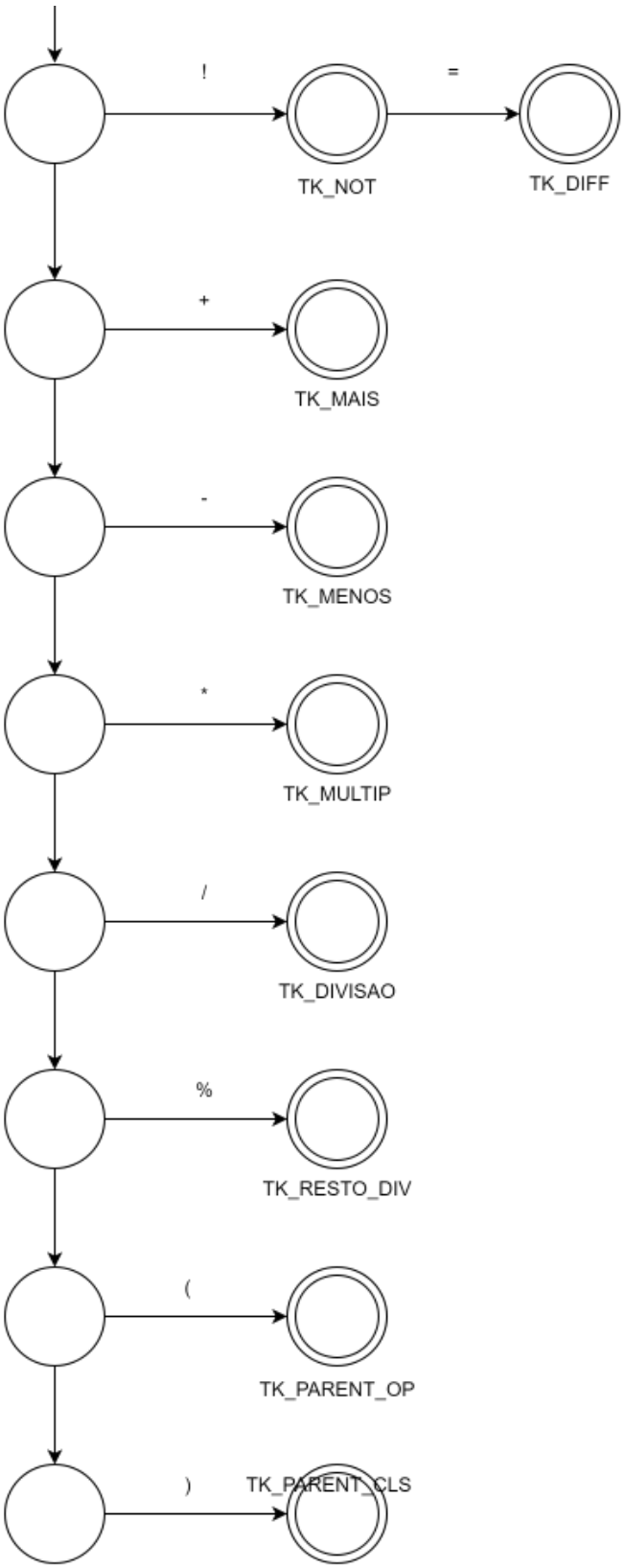

!
=
TK_NOT
TK_DIFF
+
TK_MAIS
-
TK_MENOS
*
TK_MULTIP
/
TK_DIVISAO
%
TK_RESTO_DIV
(
TK_PARENT_OP
)
TK_PARENT_CLS

figura 1.

5. ANÁLISES

Como já foi descrito previamente, um compilador é composto por três fases:

- Análise léxica;
- Análise sintática;
- Análise semântica.

Com base nisso, essa seção tem como base explicar cada fase e demonstrar o que foi feito em código.

5.1 ANÁLISE LÉXICA

A análise léxica é a primeira fase do processo de compilação. Seu objetivo é analisar o código-fonte em uma sequência de tokens significativos. O analisador léxico, também conhecido como scanner, percorre o código-fonte caracter por caracter, identificando palavras-chave, identificadores, números, símbolos e outros elementos lexicais da linguagem de programação. Os tokens resultantes são armazenados em uma lista e enviados para a próxima fase, a análise sintática.

Neste trabalho, foi usado um autômato(figura 1) como base para classificar os elementos do código fonte. Esse método foi escolhido devido à sua simplicidade e facilidade de implementação. É importante observar que não foram definidas transições para as palavras reservadas no autômato, pois elas são tratadas como identificadores e serão posteriormente classificadas em palavras reservadas na tabela de símbolos.

Durante o processo de análise léxica, o código-fonte é percorrido linha por linha e os tokens são identificados e criados através do scanner. Ao final dessa etapa, uma lista de tokens é gerada, além de uma tabela contendo informações como entrada, nome, tipo, tipo_data, valor, escopo e linha. É nessa etapa que são detectados erros de léxicos que resultam na geração de tokens inválidos para a sintaxe da linguagem.

Aqui está uma breve visão geral do código:

- O dicionário token_patterns define expressões regulares para diferentes tipos de token, como palavras-chave, identificadores, inteiros, floats, caracteres, strings, operadores e outros.

- A função search é usada para evitar duplicidade na tabela de símbolos (quando um identificador é chamado mais de uma vez). ela funciona procurando este token na tabela de símbolos com base em seu lexema e tipo.
- A função create_symbol é usada para criar um símbolo e adicioná-lo à tabela de símbolos.
- A função build_symbol_table itera sobre os tokens e constrói a tabela de símbolos criando símbolos para palavras-chave, identificadores e outros tipos com base em certas condições e regras.
- A função tokenize recebe o código-fonte como entrada e o tokeniza correspondendo às expressões regulares definidas em token_patterns. Ela retorna uma lista de tokens.
- A função tokenize_file lê o código-fonte de um arquivo e chama as funções tokenize e build_symbol_table para tokenizar o código e construir a tabela de símbolos.

Para usar o código, é necessário fornecer um arquivo de código-fonte válido .txt (por exemplo, "source_code.txt") e chamar a função tokenize_file. Após tokenizar o código e construir a tabela de símbolos, você pode iterar sobre as listas tokens e symbol_table para acessar os tokens individuais e as entradas de símbolos, respectivamente.

5.1.1 Exemplos práticos:

Abaixo alguns exemplos práticos de códigos e suas saídas:

Código 1:

```
def test(){
   int y = 5
}
```

Saída:

entrada: 1 | Name: test | Type: TK.ID | data_type: None | value: None | Scope: global | line: 1
entrada: 2 | Name: int | Type: TK.KEYWORD | data_type: int | value: None | Scope: test | line: 2
entrada: 3 | Name: y | Type: TK.ID | data_type: None | value: None | Scope: test | line: 2
entrada: 4 | Name: 5 | Type: TK.INT | data_type: int | value: 5 | Scope: test | line: 3

Código 2:

```
int x = 10
```

```
def test(){
    int y = 5
    y = x
}
```

Saída:

entrada: 1 | Name: int | Type: TK.KEYWORD | data_type: int | value: None | Scope: global | line: 1
entrada: 2 | Name: x | Type: TK.ID | data_type: None | value: None | Scope: global | line: 1
entrada: 3 | Name: 10 | Type: TK.INT | data_type: int | value: 10 | Scope: global | line: 2
entrada: 4 | Name: test | Type: TK.ID | data_type: None | value: None | Scope: global | line: 2
entrada: 5 | Name: int | Type: TK.KEYWORD | data_type: int | value: None | Scope: test | line: 3
entrada: 6 | Name: y | Type: TK.ID | data_type: None | value: None | Scope: test | line: 3
entrada: 7 | Name: 5 | Type: TK.INT | data_type: int | value: 5 | Scope: test | line: 4

5.1.2 Erros léxicos:

A seguir um exemplo de uma código com erro léxico e sua saída:

Código:

```
def test(){
    int y = $
}
```

Saída:

“ValueError: Invalid token '$' at line 2”

5.2 ANÁLISE SINTÁTICA

A análise sintática é a segunda fase do processo de compilação. Ela recebe a lista de tokens produzidos pela análise léxica e constrói uma árvore sintática ou uma representação estruturada do código-fonte. Essa árvore sintática, também conhecida como árvore de análise, mostra a estrutura gramatical do programa. O analisador sintático, também chamado de parser, utiliza regras gramaticais e técnicas como a análise descendente ou ascendente para verificar se o código está de acordo com a sintaxe da linguagem. Caso ocorram erros sintáticos, são gerados mensagens de erro indicando as violações encontradas.

Como já há a saída do código da análise léxica a disposição, é possível criar uma árvore sintática, porém, antes disso é necessário a eliminação de retrocessos para que ocorra a validação sintática, ou seja, é cabível o conhecimento de antemão das regras sintáticas aplicáveis.

Sendo assim, as regras gramaticais são determinadas de forma a evitar recursão à esquerda e garantir que não haja duas regras começando ou levando a um mesmo terminal, evitando assim ambiguidades e garantir que a análise seja determinística. Dessa forma, ao observar o próximo token de entrada, é possível determinar exatamente qual regra aplicar.

A análise sintática descendente não recursiva é baseada em um conjunto de regras gramaticais (produções) que descrevem a estrutura sintática da linguagem. Durante esta análise, o analisador percorre a lista de tokens, lendo os tokens sequencialmente, e utiliza as regras da bnf e as informações do próximo token para decidir qual derivação aplicar em cada passo. A cada derivação, um nodo é criado na parse tree com base na produção feita, seu objetivo é ir encontrando as derivações ideais que gerem a derivação correta da parse tree de acordo com as regras da linguagem.

No código que foi implementado, a função principal é “parse_S()”, que representa a regra de produção inicial da gramática da linguagem. Essa função verifica se o próximo token corresponde a um def (criação de uma função) ou a um tipo de variável (como int, float, char, bool ou string), que corresponde a definição de uma variável. Com base nisso, chama funções específicas para analisar a definição de função ou a definição de variável.

Outras funções são implementadas para analisar diferentes elementos da linguagem, como expressões, estruturas de controle (como if, else, for e while), operações (atribuição e comparação), tipos de variáveis, entre outros.

Cada função de análise sintática retorna um objeto Node que representa um nó da árvore de análise sintática. A árvore de análise sintática é construída conforme o código é analisado, e cada nó da árvore representa uma produção da gramática da linguagem.

5.2.1 Exemplos práticos:

Abaixo alguns exemplos práticos de códigos e suas saídas:

Código 1:

```
def test(){
   int y = 5
}
```

Saída: (parse_tree níveis representados na horizontal)

```
<S>
 <func>
  def
  <var_name>
   test
  (
  <escopo>

  )
  {
  <expressao>
   <define_var>
    <type>
     int
    <var_name>
     y
    =
    <atr>
     5
  }
```

Código 2:

```
int x = 10
def test(){
   int y = 5
   y = x
}
```

Saída: (parse_tree níveis representados na horizontal)

```
<S>
 <define_var>
  <type>
   int
  <var_name>
   x
  =
  <atr>
   10
 <S>
  <func>
   def
   <var_name>
    test
   (
   <escopo>
```

```
)
{
<expressao>
  <define_var>
    <type>
      int
    <var_name>
      y
    =
    <atr>
      5
  <expressao>
    <operacao>
      <operacao_atr>
        <var_name>
          y
        =
        <var_name>
          x
}
```

5.2.2 Erros Sintáticos:

A seguir um exemplo de uma código com erro léxico e sua saída:

Código:

```
#erro de inicialização de variável
def test(){
   int y
}
```

Saída:

“SyntaxError: line 4, Expected '=', but found '}'”

5.3 ANÁLISE SEMÂNTICA

A análise semântica é uma etapa crucial no processo de compilação, pois verifica se as regras semânticas da linguagem de programação estão sendo obedecidas. Durante essa etapa, o compilador examina o significado e as relações entre os elementos do programa, garantindo a consistência e a correção semântica. Isso inclui a verificação de tipos de dados, assegurando que as operações sejam realizadas de maneira coerente, além de lidar com

questões de escopo, como a visibilidade de variáveis e a resolução de nomes. Em resumo, a parte semântica é dedicada para estudar o sentido das palavras e o contexto aplicado, onde os principais pontos analisados são:

- Análise contextual;
- Verificação do tipo de variáveis diante de operações;
- Verificação de pertencimento de escopo.
- Verificação de duplicidade ou nao declaramento de variáveis

Esse levantamento de dados é todo armazenado na **Tabela de Símbolos**, fazendo ajustes necessários, de acordo com a lógica usada para análise, para que assim possa ser analisada finalmente. Erros semânticos, como o uso de uma variável não declarada ou a aplicação de uma operação inválida, geralmente são detectados nessa etapa e podem resultar em mensagens de erro específicas para o programador.

Em suma, a análise semântica desempenha um papel essencial na verificação da corretude semântica do programa fonte, garantindo que ele esteja de acordo com as regras e restrições da linguagem de programação.

A seguir está uma breve visão geral do código:

- A classe “SemanticError” é uma exceção personalizada para erros semânticos.
- A função “update_table_atr” atualiza os atributos de uma entrada na tabela de símbolos, como tipo e valor, com base em informações de outros símbolos.
- A função “update_table” atualiza o tipo de uma entrada na tabela de símbolos e atribui um valor simbólico aos identificadores de funções e variáveis utilizadas como parâmetros.
- A função “verify_type” verifica se dois tipos de dados são compatíveis. Se os tipos forem incompatíveis, uma exceção “SemanticError” é lançada.
- A função “verify_duplicity” verifica se uma variável já foi declarada anteriormente no escopo atual. Se a duplicidade for encontrada, uma exceção “SemanticError” é lançada.
- A função “verify_var_existence” verifica se uma variável existe na tabela de símbolos e se está acessível no escopo atual. Se a variável não existir, uma exceção “SemanticError” é lançada.

- A função “verify_define_var” realiza verificações semânticas específicas para a definição de variáveis, como verificação de duplicidade e verificação de tipo. Ela chama as funções auxiliares conforme necessário.
- As funções “verify_comp_op” e “verify_atr_op” realizam verificações semânticas para operações de comparação e atribuição, respectivamente. Elas verificam a existência de variáveis, a compatibilidade de tipos e lançam exceções “SemanticError” quando apropriado.
- A função “pre_order_traversal” percorre a árvore de análise sintática em pré-ordem, chama as funções de verificação semântica relevantes para cada nó e mantém o controle do escopo atual usando a lista “scope_track” e do nó pai usando a variável father.

Após a execução do código, são realizadas verificações semânticas percorrendo a árvore de análise sintática. Em caso de erros semânticos, uma exceção “SemanticError” é lançada com uma mensagem de erro apropriada.

5.3.1 Exemplos práticos:

Abaixo alguns exemplos práticos de códigos e suas saídas:

Código 1:

```
def test(){
   int y = 5
}
```

Saída: (tabela de símbolos atualizada)

entrada: 1, Name: test, Type: TK.ID, data_type: def, value: 0 Scope: global
entrada: 2, Name: int, Type: TK.KEYWORD, data_type: int, value: None Scope: test
entrada: 3, Name: y, Type: TK.ID, data_type: int, value: 5 Scope: test
entrada: 4, Name: 5, Type: TK.INT, data_type: int, value: 5 Scope: test

Código 2:

```
int x = 10
def test(){
   int y = 5
```

```
    y = x
}
```

Saída: (tabela de símbolos atualizada)

entrada: 1, Name: int, Type: TK.KEYWORD, data_type: int, value: None Scope: global
entrada: 2, Name: x, Type: TK.ID, data_type: int, value: 10 Scope: global
entrada: 3, Name: 10, Type: TK.INT, data_type: int, value: 10 Scope: global
entrada: 4, Name: test, Type: TK.ID, data_type: def, value: 0 Scope: global
entrada: 5, Name: int, Type: TK.KEYWORD, data_type: int, value: None Scope: test
entrada: 6, Name: y, Type: TK.ID, data_type: int, value: 5 Scope: test
entrada: 7, Name: 5, Type: TK.INT, data_type: int, value: 5 Scope: test

5.3.2 Erros Semânticos:

A seguir um exemplo de uma código com erro léxico e sua saída:

Código:

```
#char recebendo inteiro
int x = 10
def test(){
    char y = 5
    y = x
}
```

Saida:

“SemanticError: Tipos de dado incompativeis: char, int, line: 3”