CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO

ANA CLARA CUNHA LOPES
GUSTAVO DE ASSIS XAVIER

Compiladores
Trabalho parte 3 - Analisador Semântico

Belo Horizonte - MG
25 de julho de 2025

SUMÁRIO

1. INTRODUÇÃO..... 3
2. ANALISADOR LÉXICO..... 4
2.1. FORMA DE USO DO COMPILADOR..... 4
2.2. ABORDAGEM UTILIZADA NA IMPLEMENTAÇÃO..... 4
2.3. PRINCIPAIS CLASSES DA APLICAÇÃO..... 5
2.4. DETALHES DA IMPLEMENTAÇÃO..... 6
2.5. RESULTADOS DOS TESTES..... 7
3. ANALISADOR SINTÁTICO..... 8
3.1. GRAMÁTICA..... 8
3.2. IMPLEMENTAÇÃO..... 10
3.3. RESULTADO DOS TESTES..... 10
4. ANALISADOR SEMÂNTICO..... 12
4.1. IMPLEMENTAÇÃO..... 12
4.1.1. TABELA DE SÍMBOLOS..... 12
4.1.2. REGRAS SEMÂNTICAS..... 13
4.2. VISÃO GERAL DO CÓDIGO..... 16
4.3. INSTRUÇÕES DE USO DO COMPILADOR SEMÂNTICO..... 19
4.4. TESTES:..... 20
4.4.1. TESTE 1:..... 20
4.4.2. TESTE 2:..... 21
4.4.3. TESTE 3:..... 22
4.4.4. TESTE 4:..... 23
4.4.5. TESTE 5:..... 23
4.4.6. TESTE EXTRA:..... 25
5. CONCLUSÃO..... 26

Observação: As partes destacadas em vermelho são as referentes à etapa três do trabalho. Além disso, a cada trabalho a gente resume a parte anterior, então nem todas as informações do último estão dispostas aqui, principalmente os testes antigos.

1. INTRODUÇÃO

Este relatório apresenta o desenvolvimento de um analisador léxico e sintático de um compilador para uma linguagem de programação específica, implementado como parte de um trabalho acadêmico. O objetivo é descrever como o analisador sintático funciona, a abordagem utilizada na implementação, destacando as principais classes e seus propósitos, e os resultados dos testes realizados, incluindo os programas-fontes analisados e as saídas do compilador.

2. ANALISADOR LÉXICO

2.1. FORMA DE USO DO COMPILADOR

O analisador sintático do compilador foi desenvolvido em Java e executado em um ambiente de desenvolvimento integrado (VS-Code). A seguir, descrevemos os passos para executá-lo:

1. **Preparação do programa-fonte:** o usuário deve criar um arquivo de texto (testeX.txt) contendo o código-fonte da linguagem especificada. Este arquivo pode ser salvo na pasta testes, junto aos demais arquivos.

2. **Compilação do projeto:** Com o terminal dentro da pasta **compilador,** faça:

   javac analisadorlexico/*.java interpretador/value/*.java *.java

3. **Execução do compilador:** após compilar, para executar o programa basta ir com o terminal para pasta COMPILADORES e executar:

   java compilador.exe

   Há o passo a passo também no README, nos arquivos.

4. **Interpretação da saída:** a saída do compilador lista os tokens identificados no formato Token(TIPO, "valor", linha=N), onde TIPO é o tipo do token (ex.: PROGRAM, IDENTIFIER), valor é o lexema identificado e N é a linha onde o lexema foi encontrado. Linhas vazias são ignoradas na contagem lógica. Se houver erro, uma mensagem acima do problema.

2.2. ABORDAGEM UTILIZADA NA IMPLEMENTAÇÃO

A implementação do compilador foi realizada em Java, focando inicialmente na etapa de análise léxica, que é responsável por transformar o código-fonte em uma sequência de tokens. A abordagem utilizada é baseada em um analisador léxico determinístico, que processa o código caractere por caractere, identificando tokens com base em regras predefinidas para a linguagem.

2.3. PRINCIPAIS CLASSES DA APLICAÇÃO

A aplicação é composta por três “classes”, cada uma com um propósito específico:

- **exe:** esta classe contém o método main, sendo o ponto de entrada da aplicação. É responsável por iniciar o compilador e coordenar a execução do processo de análise léxica. Ela instancia a classe AnalisadorLexico, passa o conteúdo do arquivo de entrada e exibe a saída gerada (tokens ou mensagens de erro) no console.

- **LexicalAnalysis:** esta classe é o núcleo do compilador, responsável pela análise léxica do código-fonte. Ela lê o arquivo de entrada, processa o texto caractere por caractere e gera uma sequência de tokens. Possui métodos para:
    - Pular espaços em branco e quebras de linha, ajustando a contagem de linhas (ignorando linhas vazias na contagem lógica).
    - Identificar palavras-chave, identificadores, constantes, literais, operadores e símbolos.
    - Gerar tokens com informações sobre tipo, valor e linha.
    - Reportar erros léxicos, como caracteres inválidos ou literais mal formados.

A classe utiliza um mapa de palavras reservadas para associar palavras-chave a tipos de tokens específicos.

- **Token:** esta classe representa um token gerado pelo analisador léxico. Cada token possui quatro atributos: o tipo (um enum que define categorias como PROGRAM, IDENTIFIER, SEMICOLON, etc.), o valor (que é representado por uma determinada classe derivada da classe Value, pois isso ajudará nas próximas implementações) o lexema (o “texto” correspondente no código-fonte) e a linha onde foi encontrado. A classe é usada para estruturar a saída do analisador léxico e facilita a integração com futuras etapas do compilador (como a análise sintática).

- **Value<?> e seus filhos:** Considerando a disciplina de LP, essa classe pode nos ajudar a padronizar valores e fazer conversões, facilitando a implementação dos demais analisadores.

2.4. DETALHES DA IMPLEMENTAÇÃO

A análise léxica foi implementada utilizando uma abordagem baseada em máquina de estados implícita, onde o método nextToken() da classe LexicalAnalysis lê caracteres consecutivamente e decide qual token gerar com base em padrões predefinidos:

- Palavras-chave e identificadores são identificados verificando se o caractere inicial é uma letra ou underline, seguido por letras, dígitos ou underline.
- Constantes numéricas são processadas verificando dígitos e, no caso de números reais, a presença de um ponto decimal seguido por mais dígitos.
- Constantes de caracteres e literais são identificados por aspas simples e duplas, respectivamente.
- Símbolos e operadores (como: +, *, - , etc.) são tratados por um switch que verifica o caractere atual e, em alguns casos, o próximo caractere, como por exemplo: ==, >=, etc.).

Os casos no geral são dados pelo diagrama (feito do Draw.io):

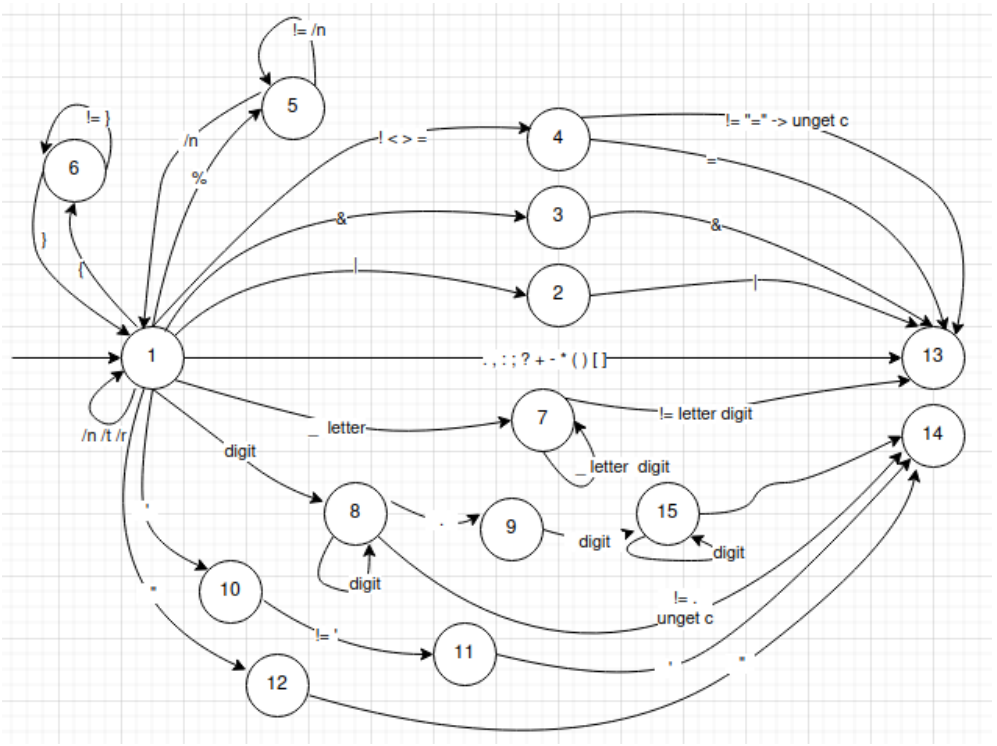


Diagrama de Estados do Analisador Léxico.

2.5. RESULTADOS DOS TESTES

Foram realizados testes com diferentes programas-fonte para verificar o funcionamento do analisador léxico. Todos os testes que deram erro foram corrigidos adequando esses testes para a análise sintática. Um teste pequeno é colocado abaixo para exemplificar o funcionamento do analisador léxico:

```
program

     float: raio, area$ = 0.0;

begin
repeat
   in(raio);
   char: resposta;
   if (raio > 0.0) then
     area =  3. * raio * raio;
     out (area);
   end;
   out ("Deseja continuar?);
   in (resp);

until (resp == 'N' || resp == 'n');

end
```

```
Analisando: teste2.txt
Token: PROGRAM      Lexema: program  Linha: 1
Token: FLOAT_CONST  Lexema: float    Linha: 2
Token: COLON        Lexema: :        Linha: 2
Token: IDENTIFIER   Lexema: raio     Linha: 2
Token: COMMA        Lexema: ,        Linha: 2
Token: IDENTIFIER   Lexema: area     Linha: 2
Erro ao analisar léxico: Na linha 2->Caracter não permitido.
```

3. ANALISADOR SINTÁTICO

3.1. GRAMÁTICA

Para a implementação da linguagem, algumas alterações e definições na gramática foram feitas, são elas:

program ::= program [decl-list] begin stmt-list end **“;”**

decl-list ::= decl {decl}

decl ::= type “:” ident-list **“;”**

ident-list ::= identifier {"," identifier}

type ::= int | float | char

stmt-list ::= stmt {stmt}

stmt ::= assign-stmt **“;”** | if-stmt **“;”** | while-stmt **“;”** | repeat-stmt **“;”** | read-stmt **“;”** | write-stmt **“;”**

assign-stmt ::= identifier "=" simple_expr

if-stmt ::= if condition then [decl-list] stmt-list **[else [decl-list] stmt- list]** end

while-stmt ::= stmt-prefix [decl-list] stmt-list end

stmt-prefix ::= while condition do

repeat-stmt ::= repeat [decl-list] stmt-list stmt-suffix

stmt-suffix ::= until condition

read-stmt ::= in "(" identifier ")"

write-stmt ::= out "(" writable ")"

writable ::= simple-expr | literal

condition ::= expression

expression ::= simple-expr [relop simple-expr]

simple-expr ::= term {addop term}

term ::= factor-a {mulop factor-a}

fator-a ::= = ["!" | "-"] factor

factor ::= identifier | constant | "(" expression ")"

relop ::= "==" | ">" | ">=" | "<" | "<=" | "!="

addop ::= "+" | "-" | ||

mulop ::= "*" | "/" | &&

constant ::= integer_const | float_const | char_const

Com isso, nossa linguagem traz alguns destaques, são eles:

- Todos os comandos terminam com “;”. Sendo que por comando se entende toda a estrutura do comando. Por exemplo, o comando if é dado por:

  if condition **then** [decl-list] stmt-list **end**

  Sendo assim, após o **end** haverá um “;”:

  if condition **then** [decl-list] stmt-list **end;**

  No formato de quebra de linha:

  if condition **then**
  [decl-list]
  stmt-list
  end;

  Isso inclui o último “end” do programa , por motivos de padronização.

- Em cada bloco, todas declarações devem ser sempre feitas antes dos demais comandos, ou seja, podemos ter:

  if condition **then**
  int : a, b;
  a = 2;
  b = 2;
  end;

  Mas não podemos ter:

  if condition **then**
  int : a;
  a = 2;
  int : b;
  b = 2;
  end;

- Comparações aninhadas devem ser organizadas com “( )”. Por exemplo, a seguinte comparação gera erros:

  if (a > b && a > 2)

  Ela deve ser organizados de forma:

  if ((a > b) && (a > 2))

3.2. IMPLEMENTAÇÃO

Para a implementação da gramática foi usado um parser LL(1). Por isso, as ambiguidades e recursão à esquerda foram corrigidas. Por exemplo, a representação correta do if seria:

if-stmt ::= if condition then [decl-list] stmt-list [else-part] end |

else-part ::= else [decl-list] stmt-list

Que pode ser simplificada por:

if-stmt ::= if condition then [decl-list] stmt-list [else declaration stmt- list] end

Já com relação a recursão a esquerda, temos que uma regra como:

simple-expr ::= term | simple-expr addop term

Se torna:

simple-expr ::= term {addop term}

Através da transformação dada por:

A ::= A α | β

Onde β = termo e A = simple-expr e α = addop term, que inserindo uma variável de "ajuda" se torna:

A ::= β A'

A' ::= α A' | ε

Que se trata de β seguido por um número indeterminado de α. Isso pode ser simplificada por:

A ::= β {α}

Da mesma forma os demais casos.

3.3. RESULTADO DOS TESTES

Foram realizados testes com diferentes programas-fonte para verificar o funcionamento do analisador sintático. Todos os testes que deram erro foram corrigidos adequando esses testes para a análise semântica. Um teste pequeno é colocado abaixo para exemplificar o funcionamento do analisador léxico:

INPUT (erros declarados em vermelho):

INPUT ANTES DA CORREÇÃO.	INPUT CORRIGIDO:
program int: a,b,c; float: result; char: ch; begin	program int: a,b,c; float: result; char: ch; begin

```
        out("Digite o valor de a:");
        in (a);
        out("Digite o valor de c:");
        read (ch);
        b = 10;
        result = (a * ch)/(b 5 - 345.27);
        out("O resultado e: ");
        out(result);
        result = result + ch;
end
```

OUTPUT DO PRIMEIRO ERRO:

```
Analisando: teste1.txt
[INFO] Analise sintatica iniciada.
[ERROR] Linha 09: Esperado 'END', encontrado 'READ'. / Token atual: read (linha: 9).
[RESULTADO] Analise sintatica falhou.
```

```
        out("Digite o valor de a:");
        in (a);
        out("Digite o valor de c:");
        in(ch);
        b = 10;
        result = (a * ch)/(b - 345.27);
        out("O resultado e: ");
        out(result);
        result = result + ch;
end;
```

4. ANALISADOR SEMÂNTICO

O analisador semântico é a parte do compilador responsável pela análise semântica do código. O principal intuito da análise semântica é a verificação de tipos das variáveis e constantes nas operações e comandos (verificação de tipos), além do controle de declarações de variáveis, acusando erros como tentar acessar uma variável não declarada ou declarar uma mesma variável duas vezes (verificação de unicidade). Além disso, o analisador semântico também é responsável pela verificação da classe de algum objeto, destinando regras semânticas a partir da classe (verificação de classe). Para fazer tudo isso, o analisador semântico é responsável por reunir informações das variáveis, comandos, funções, constantes e etc. e as salva na tabela de símbolos.

4.1. IMPLEMENTAÇÃO

O analisador semântico desenvolvido neste trabalho é baseado em tradução dirigida pela sintaxe, ou seja, abordagem na qual ações semânticas são executadas durante a análise sintática. Assim sendo, na construção da árvore sintática para determinada entrada, alguns nós da árvore terão regras semânticas atribuídas.

Regras semânticas são um conjunto de regras lógicas que regem os comportamentos dos programas com relação aos tipos de dados que são tratados neles.

A estrutura do analisador sintático é dado por um conjunto de métodos, simulando cada regra sintática, de forma que sempre que devemos achar uma estrutura denotada por outra regra, chamamos o método referente a ela para pegar a estrutura. Para uma análise simplesmente sintática, não é necessário retornar nada em cada método, mas para a análise semântica, é necessário retornar as informações até o local onde elas devem ser tratadas.

Nesse sentido, se faz necessário uma representação mais geral das construções da linguagem, que é chamada de expressões de tipo, que pode ser desde uma variável sozinha como uma operação binária entre duas variáveis, e que foi usada na nossa implementação.

4.1.1. TABELA DE SÍMBOLOS

A tabela de símbolos é o local onde é guardado as informações sobre as palavras reservadas e as variáveis, assim como o tipo de cada uma. Para fazer isso, cada variável da nossa linguagem é associada a um tipo, também um objeto criado (ex: IntegerValue,

CharValue…), dessa forma, a tabela é um map que contém o nome da variável, denotado pelo lexema do token, e por seu tipo.

Para modificar a tabela, há dois métodos, um para inserir uma variável (put) e um para pegar as informações de determinada variável através do nome (get). Nesse métodos, há restrições de forma que se tentar declarar uma variável já declarada irá gerar erro, assim como tentar obter uma não declarada também irá gerar erro.

Na implementação proposta, cada bloco de código (do if, while e repeat) possui sua própria tabela de símbolos. Isso permite controlar o escopo das variáveis, garantindo que declarações internas a um bloco não interfiram em variáveis externas.

Para isso, a estrutura da Tabela de Símbolos foi projetada de forma encadeada, de modo que cada nova tabela criada em um bloco aponta para a tabela do escopo imediatamente superior. Durante a resolução de identificadores, a busca ocorre de forma recursiva até que o símbolo seja encontrado ou determinado como indefinido.

Dessa forma, ao declarar todas as variáveis, garantimos a unicidade no código.

4.1.2. REGRAS SEMÂNTICAS

Para implementar as regras semânticas no analisador sintático, além do uso de expressões de tipo para os retornos, também foi necessário a criação de um tipo de valor, BooleanType, que denota o valor de comparações feitas entre os demais tipos, por exemplo uma expressão: int > int gera um BooleanType.

As regras semânticas são mostradas em alto nível, para cada produção, pela tabela 1. Nessa tabela também é mostrada onde é gerado novos escopos.

Tabela 1 - Regras Semênticas por produção.

Produção.	Regra Semântica.
program ::= program [decl-list] begin stmt-list end **“;”**	-
decl-list ::= decl {decl}	-
decl ::= type “:” ident-list **“;”**	Se (variável já declarada) então Gerar_ Erro senão Declarar_Variável.
ident-list ::= identifier {"," identifier}	-
type ::= int | float | char	-
stmt-list ::= stmt {stmt}	-

stmt ::= assign-stmt **“;”** | if-stmt **“;”** | while-stmt **“;”** | repeat-stmt **“;”** | read-stmt **“;”** | write-stmt **“;”**	-
assign-stmt ::= identifier "=" simple_expr	Se (variável.tipo == expressão.tipo) então atribuir Senão Gerar_ Erro.
if-stmt ::= if condition then [decl-list] stmt-list **[else [decl-list] stmt- list]** end	Criação de novo escopo para decl-list. O escopo é desfeito e volta para o seu anterior após a execução do comando.
while-stmt ::= stmt-prefix [decl-list] stmt-list end	Criação de novo escopo para decl-list.
stmt-prefix ::= while condition do	-
repeat-stmt ::= repeat [decl-list] stmt-list stmt-suffix	Criação de novo escopo para decl-list.
stmt-suffix ::= until condition	-
read-stmt ::= in "(" identifier ")"	Se (variável já declarada) então Prosseguir Senão Gerar_ Erro.
write-stmt ::= out "(" writable ")"	-
writable ::= simple-expr | literal	-
condition ::= expression	Se (expression.type == BoolType) então Prosseguir Senão Gerar_ Erro.
expression ::= simple-expr [relop simple-expr]	Conjunto de regras I
simple-expr ::= term {addop term}	Conjunto de regras II.
term ::= factor-a {mulop factor-a}	Conjunto de regras III.
fator-a ::= = ["!" | "-"] factor	Conjunto de regras IV.
factor ::= identifier | constant | "(" expression ")"	-
relop ::= "==" | ">" | ">=" | "<" | "<=" | "!="	-
addop ::= "+" | "-" ||	-
mulop ::= "*" | "/" | &&	-
constant ::= integer_const | float_const | char_const	-

Considerando que qualquer caso não descrito resulte em erro, temos:

- Conjunto de regras I:

Comparações binárias feitas com "==" | ">" | ">=" | "<" | "<=" | "!=" podem ser feitas entre os seguintes tipos e geram os seguintes tipos:

Float com Float -> Bool

Float com Int -> Bool

Int com Int -> Bool

Char com Int -> Bool

Char com Char -> Bool

Bool com Bool -> Bool

- Conjunto de regras II:

Operações binárias feitas por "+" | "-" podem ser feitas entre os seguintes tipos e geram os seguintes tipos:

Float com Float -> Float

Float com Int -> Float

Int com Int -> Int

Char com Int -> Int

Char com Char -> Int

Operações binárias feitas por “||” podem ser feitas entre os seguintes tipos e geram os seguintes tipos:

Bool com Bool -> Bool

- Conjunto de regras III:

Operações binárias feitas por "*"podem ser feitas entre os seguintes tipos e geram os seguintes tipos:

Float com Float -> Float

Float com Int -> Float

Int com Int -> Int

Char com Int -> Int

Char com Char -> Int

Operações binárias feitas por "/" podem ser feitas entre os seguintes tipos e geram os seguintes tipos:

Float com Float -> Float

Float com Int -> Float

Int com Int -> Float

Char com Int -> Float

Char com Char -> Float

Operações binárias feitas por "&&" podem ser feitas entre os seguintes tipos e geram os seguintes tipos:

Bool com Bool -> Bool

- Conjunto de regras IV:

Operações unárias feitas por “-” pode ser feita para os seguinte tipos e geram os seguintes tipos:

Float -> Float

Int -> Int

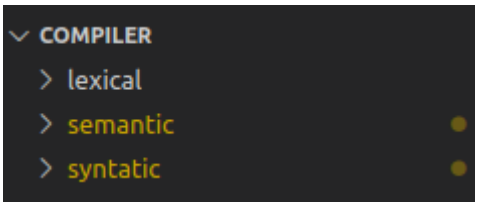
Operações unárias feitas por “!” pode ser feita para os seguinte tipos e geram os seguintes tipos:

Bool -> Bool

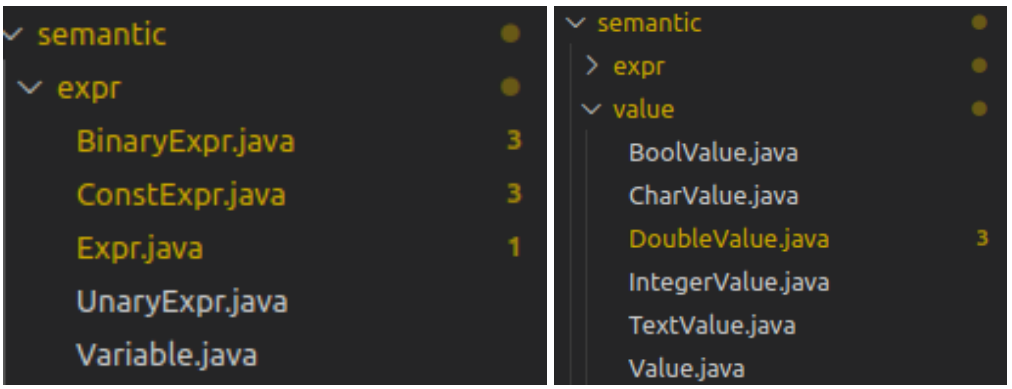
Mais detalhes da implementação são gerados no tópico “Visão Geral do Código”, onde há uma breve explicação do funcionamento do código feito.

4.2. VISÃO GERAL DO CÓDIGO

O compilador, é composto basicamente de 3 estruturas principais, o analisador léxico, sintático e semântico.



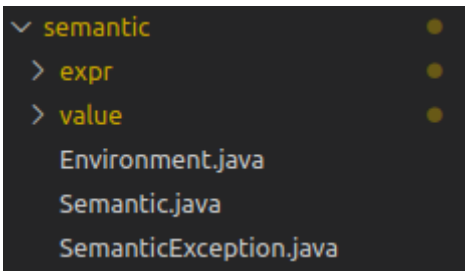
O papel dos analisadores léxico e sintático já foram previamente discutidos. A inserção de um analisador semântico traz consigo algumas implementações. Como a criação de classes de valores para facilitar (já implementada em versões passadas), e de expressões de tipo, que representam construções da linguagem, como discutido anteriormente.



Como visto na imagem, a expressão de tipo pode representar construções de operações binárias, unárias, uma simples variável ou uma constante. Cada expressão é associada com um Value, dessa forma é possível, através das regras semânticas, fazer a verificação de tipos. A forma como a verificação é feita é mostrada no método abaixo, da classe BinaryExpr:

```
// && e ||
private Value<?> andorOp(Value<?> v1, Value<?> v2) {
    // Só pode se ambos os tipos são booleanos
    // Bool com Bool -> Bool
    if((v1 instanceof BoolValue) &&  (v2 instanceof BoolValue)){
        return new BoolValue(value:null);
    }
    // Se não é erro
    else{
        throw new SemanticException(super.getLine()
        ,mensagem:"Tipo não permitido para comparação de && ou ||");
    }
}
```

Além disso, como discutido anteriormente, o analisador semântico tem acesso a tabela de símbolos. Mas nessa implementação, ela não é a do léxico, que guarda informações das palavras chaves, ela é uma tabela própria do semântico, que guarda informações apenas das variáveis. A classe responsável por guardar a tabela e seus métodos é a environment, enquanto que a classe semantic guarda uma instância dessa classe, que denota a escopo global.



Um exemplo de como a análise semântica é imposta no analisador sintático é mostrado no trecho de código a seguir:

```
public class SyntaticAnalysis {
// fator-a ::= factor | ! factor | "-" factor
    private Expr procFactorA() {

        Expr expr = null;
        int line = current.line;

        if (match(NOT, SUB)) {
            switch (previous.type) {
                case NOT:
                    expr = new UnaryExpr(line, procFactor(), UnaryExpr.Op.Not);
                    break;
                case SUB:
                    expr = new UnaryExpr(line, procFactor(), UnaryExpr.Op.Neg);
                    break;
                default:
                    break;
            }
        } else {
            expr = procFactor();
        }

        return expr;
    }
```

Como podemos ver, temos expressões de tipo que vem de leituras mais “baixas” da árvore sintática, que vêm através de chamadas como procFactor(). Ao termos tanto a operação como as expressões de tipo que serão utilizadas, podemos gerar uma nova expressão de tipo referente a operação. O valor dessa nova expressão de tipo é definido através da própria classe referente a operação. Por exemplo, nesse trecho de código, passamos os parâmetros para um método da classe de operação unitária, e ela estabelece qual será o novo tipo de acordo com as regras presentes nessa classe(regras semânticas). Regras simples foram implementadas diretamente no código do sintático, como a verificação de que se é uma condição (se o valor da expressão é um BoolValue) feita no trecho abaixo.

```
// condition ::= expression
    private BoolValue procCondition() {
        // Preciso saber se a expressão que eu achei é um valor booleano
        // Sendo que valores booleanos só são gerados a partir de comparações
        // == != > >= < <=

        Expr expr = procExpression();
        Value v = expr.expr();

        if(v instanceof BoolValue){
            return new BoolValue(value:null);
        }

        throw new SemanticException(current.line, mensagem:"Não é uma condição");
    }
```

Dessa forma, aplicando todas as regras semânticas no analisador sintático, podemos fazer a análise semântica do código.

4.3. INSTRUÇÕES DE USO DO COMPILADOR SEMÂNTICO

O analisador semântico do compilador foi desenvolvido em Java e executado em um ambiente de desenvolvimento integrado (VS-Code). A seguir, descrevemos os passos para executá-lo:

- **Preparação do programa-fonte:** o usuário deve criar um arquivo de texto (testeX.txt) contendo o código-fonte da linguagem especificada. Este arquivo deve ser salvo na pasta testes, o programa lerá ele automaticamente, porém, somente um arquivo de teste deve ser colocado na pasta por vez, pois a tabela de símbolos não é reiniciada de um para outro (logo, geraria erros).

- **Compilação do projeto:** Com o terminal dentro da pasta **compiler (**que possui o README**),** faça:

  javac -d . $(find . -name "*.java")

- **Execução do compilador:** após compilar, para executar o programa, ainda no mesmo terminal, digite:

  java compiler.exe

  Há o passo a passo também no README, nos arquivos.

- **Interpretação da saída:** Caso o programa seja corretamente lido, será gerado a mensagem “Análise semântica concluída com êxito”, caso haja algum erro, o motivo do primeiro erro encontrado e sua linha serão impressas no terminal.

4.4. TESTES:

Para validar o analisador semântico implementado, uma série de testes foi executado. O objetivo é verificar se todas as regras semânticas, de verificação de tipos e de escopo de variáveis, são aplicadas corretamente pelo compilador.

Cada teste a seguir apresentará um código-fonte, a saída esperada do compilador e uma breve análise do resultado, demonstrando o comportamento do analisador semântico tanto em cenários de sucesso quanto na identificação de erros.

4.4.1. TESTE 1:

INPUT:	**OUTPUT**:

program
int: a,b,c;
float: result;
char: ch;
begin
out("Digite o valor de a:");
in (a);
out("Digite o valor de c:");
in (ch);
b = 10;
result = (a * ch)/(b - 345.27);
out("O resultado e: ");
out(result);
result = result + ch;
end;
	Analisando: teste1.txt
	
	Erro ao analisar código 14: Operação semântica inválida: Operandos não permitido para comparação de + - ou *

Esse erro ocorreu pois na linha 14 temos result = result + ch;, no entanto, ch é um Char enquanto “result” é um float, e operações entre esses dois não é permitido na linguagem. Substituindo “ch” por “b”, o programa fica semanticamente correto.

INPUT CORRIGIDO:	**OUTPUT COM CÓDIGO CORRIGIDO**:

program
int: a,b,c;
float: result;
char: ch;
begin
out("Digite o valor de a:");
in (a);
out("Digite o valor de c:");
in (ch);
b = 10;
result = (a * ch)/(b - 345.27);
out("O resultado e: ");
out(result);
result = result + b;
end;
	Analisando: teste1.txt
	Análise Semântica realizada com sucesso

O código fonte acima, não apresenta nenhum erro semântico.

4.4.2. TESTE 2:

INPUT:	**OUTPUT:**

program
float: raio, area;
begin
repeat
char: resposta;
in(raio);
if (raio > 0.0) then
area = 3.0 * raio * raio;
out (area);
end;
out ("Deseja continuar?");
in (resp);
until ((resp == 'N') || (resp == 'n'));
end;
	Analisando: teste2.txt
	Erro ao analisar código 12: Operação semântica inválida: Variável não declarada: resp

Erro devido a variável “resp” nunca ter sido declarada, substituindo resp por “resposta” obtivemos sucesso no procedimento.

INPUT CORRIGIDO:	**OUTPUT COM CÓDIGO CORRIGIDO**:

program
float: raio, area;
begin
repeat
char: resposta;
in(raio);
if (raio > 0.0) then
area = 3.0 * raio * raio;
out (area);
end;
out ("Deseja continuar?");
in (resposta);
until ((resposta == 'N') || (resposta == 'n'));
end;
	Analisando: teste2.txt
	
	Analise concluida com sucesso!

O código fonte acima, não apresenta nenhum erro semântico.

4.4.3. TESTE 3:

INPUT:	**OUTPUT:**

program
int: a, b, aux;
begin
in (a);
in(b);
if (a>b) then
int: aux;
aux = b;
b = a;
a = aux;
end;
out(a);
out(b);
end;
	Analisando: teste3.txt
	
	Analise concluida com sucesso!

O código compila sem erros. Como podemos ver, a variável aux foi declarada duas vezes, mas o compilador, ao entrar no bloco if, cria um novo escopo com uma tabela de símbolos própria e vazia. A declaração int: aux; dentro do if é então validada apenas contra essa nova tabela local. Como ela não contém nenhuma variável aux, a declaração é permitida, criando uma variável completamente nova que existe apenas dentro daquele bloco.

4.4.4. TESTE 4:

INPUT:	**OUTPUT**:

program
int: a, b, c, maior, outro;
begin
repeat
out("A");
in(a);
out("B");
in(b);
out("C");
in(c);
%Verifica o maior
if ((a>b) && (a>c)) then
  in(c);
else
maior = a;
if (b>c) then
  maior = b;
else
  maior = c;
end;
end;
out("Maior valor:");
out (maior);
out ("Outro ");
in(outro);
until (outro == 0);
end;
	Analisando: teste4.txt
	
	Análise Semântica realizada com sucesso

O código fonte acima, não apresenta nenhum erro semântico.

4.4.5. TESTE 5:

INPUT:	**OUTPUT**:

program
int: pontuacao, pontuacaoMaxina, disponibilidade;
char: pontuacaoMinima;
begin
disponibilidade = 'S';
pontuacaoMinima = 50;
pontuacaoMaxima = 100;
out("Pontuacao Candidato: ");
in(pontuacao);
out("Disponibilidade Candidato: ");
in(disponibilidade);
 	
{ Comentario
grande
	Analisando: teste5.txt
	
	Erro ao analisar código 05: Operação semântica inválida: Não são do mesmo tipo

while (pontuacao>0 && (pontuação <= pontuacaoMaxima) do
int: cont;
cont = cont + 1;
if ((pontuação > pontuacaoMinima) && (disponibilidade==1)) then
out("Candidato aprovado")
else
out("Candidato reprovado")
end
out("Pontuacao Candidato: ");
in(pontuacao);
out("Disponibilidade Candidato: ");
in(disponibilidade);
end
out (cont);
end}
end;	

Esse erro acontece pois na linha 5, é feito: disponibilidade = 'S';, o que não é permitido já que ‘S’ é Char e disponibilidade é Int. Atribuição em tipos diferentes não é permitido. Existem outras atribuições erradas no programa também, corrigindo todas, e também o erro de digitação de pontuacaoMaxina, obtemos:

INPUT CORRIGIDO:	**OUTPUT COM CÓDIGO CORRIGIDO**:

program
int: pontuacao, pontuacaoMaxima, pontuacaoMinima;
char: disponibilidade;
begin
disponibilidade = 'S';
pontuacaoMinima = 50;
pontuacaoMaxima = 100;
out("Pontuacao Candidato: ");
in(pontuacao);
out("Disponibilidade Candidato: ");
in(disponibilidade);
 	
{ Comentario
grande
while (pontuacao>0 && (pontuação<= pontuacaoMaxima) do
int: cont;
cont = cont + 1;
if ((pontuação > pontuacaoMinima) && (disponibilidade==1)) then
out("Candidato aprovado")
else
out("Candidato reprovado")
end
out("Pontuacao Candidato: ");
in(pontuacao);
	Analisando: teste5.txt
	
	Analise concluida com sucesso!

out("Disponibilidade
Candidato: ");
in(disponibilidade);
end
out (cont);
end}
end;	

O código fonte acima, não apresenta nenhum erro semântico.

4.4.6. TESTE EXTRA:

INPUT:	**OUTPUT**:

program
int: x, y;
float: media;
char: opcao;
% Teste de comentario de linha inicial
begin
{ Teste de comentario em bloco
sobre o programa
}
out("Digite o valor de x: ");
in(x);
out("Digite o valor de y: ");
in(y);
media = (x + y) / 2.0;
out("Media dos valores: ");
out(media);
if (media >= 5.0) then
out("Media suficiente!");
else
out("Media insuficiente!");
end;
out("Digite a opcao (S/N): ");
in(opcao);
while ((opcao != 'N') && (opcao != 'n')) do
out("Digite novos valores para x e y: ");
in(x);
in(y);
media = (x + y) / 2.0;
out("Nova media: ");
out(media);
out("Digite a opcao (S/N): ");
in(opcao);
end;
end;
	Analisando: teste6.txt
	Analise concluida com sucesso!

A análise desse teste não apresentou nenhum erro semântico.

5. CONCLUSÃO

Este trabalho mostrou como um compilador foi construído em etapas. Começamos com o analisador léxico, depois o sintático e, por fim, o semântico. O projeto foi todo feito em Java, usando o programa VS-Code.

A primeira parte foi o analisador léxico, que pegou o código escrito e o quebrou em pequenas partes, chamadas tokens. Em seguida, o analisador sintático verificou se esses tokens estavam na ordem certa, de acordo com as regras da linguagem.

A última etapa, detalhada neste relatório, foi o analisador semântico. Ele foi adicionado para verificar se o código fazia sentido. Para isso, as regras de lógica foram checadas ao mesmo tempo em que a estrutura do código era analisada. Foi criada uma tabela de símbolos para guardar as informações das variáveis e controlar onde elas podiam ser usadas. Com isso, o analisador semântico garantiu que as operações matemáticas estivessem certas, que os tipos de variáveis fossem compatíveis e que nenhuma variável fosse usada ou declarada de forma errada.

No final, o resultado foi um compilador que funciona para a linguagem criada e que consegue fazer as três primeiras fases da compilação. O projeto ajudou a entender na prática como um compilador é feito, desde o início, com a leitura do código, até a verificação das regras mais complexas de lógica que fazem um programa funcionar corretamente.