UNIVERSIDADE ESTADUAL DE MARINGÁ
Centro de Tecnologia
Departamento de Informática

INFORMÁTICA

9926 - Introdução à Compilação

Projeto: Criação de um compilador orientado a objetos

Data:10/05/2025

Professor: Felippe Fernandes da Silva

RA	**Discentes**
123756	Renan Batista Meyring Junior
125710	Heiber Marcos Rissi Fertunani
126619	Gabriel Galieta Baldasso

1- Visão Geral

Este relatório técnico documenta o desenvolvimento de um compilador para a linguagem fictícia Element, projetada com fins didáticos e inspirada em linguagens modernas que combinam características imperativas, tipagem estática e elementos iniciais de orientação a objetos.
O projeto tem como objetivo explorar, por meio da implementação prática, todas as etapas do processo de compilação — da análise léxica até a geração de código intermediário. Para isso, foi utilizada a ferramenta ANTLR (Another Tool for Language Recognition), que automatiza as fases de análise léxica e sintática com base em uma gramática formal, permitindo concentrar os esforços de desenvolvimento na análise semântica e na geração de código.
A linguagem Element foi definida para oferecer uma sintaxe concisa e pedagógica, suportando os principais elementos de linguagens reais, como:

- Tipos primitivos (inteiro, ponto flutuante, caractere, string, booleano)
- Declaração e chamada de funções (com e sem retorno)
- Comandos de entrada e saída
- Vetores genéricos
- Estruturas de controle (condicionais, loops)
- Classes e tipos definidos pelo usuário
- Chamadas de métodos por instância (objeto.método)

O compilador gerado é capaz de processar programas escritos em Element, realizar a validação semântica completa, identificar erros como uso indevido de tipos ou variáveis não declaradas, e produzir código intermediário e código em linguagem LLVM IR — aproximando-se do funcionamento de compiladores reais

O compilador executa as seguintes etapas:

- Análise semântica com escopos;
- Inferência e compatibilidade de tipos;
- Geração de código intermediário;
- Geração de código LLVM (formato LLVM IR);
- Exportação da árvore sintática em JSON para depuração.

2- Tecnologias Utilizadas

- Linguagem de implementação: C# (.NET 8)
- ANTLR 4.13.2 (gerador de lexer/parser)
- Serialização JSON nativa do .NET (System.Text.Json)
- Terminal/Console para interação
- LLVM textual (saida.ll)
- Visual Studio / VS Code

3- Definição da Linguagem (Element.g4)

A linguagem Element é uma linguagem de programação fictícia criada com fins educacionais, com influência simbólica da Tabela Periódica dos Elementos. Cada palavra-chave é representada por um símbolo químico, favorecendo a memorização e o uso criativo da linguagem.

A linguagem Element suporta:

```CSharp
H, He, Li, Be, B, C, N, O, F, Ne,
Na, Mg, Al, Si, P, S, Cl, Ar, K, Ca,
Sc, Ti, V, Cr, Mn, Fe, Co, Ni, Cu, Zn,
Ga, Ge, As, Se, Br, Kr, Rb, Sr, Y, Zr,
Nb, Mo, Tc, Ru, Rh, Pd, Ag, Cd, In, Sn,
Sb, Te, I, Xe, Cs, Ba, La, Ce, Pr, Nd,
Pm, Sm, Eu, Gd, Tb, Dy, Ho, Er, Tm, Yb,
Lu, Hf, Ta, W, Re, Os, Ir, Pt, Au, Hg,
Tl, Pb, Bi, Po, At, Rn, Fr, Ra, Ac, Th,
Pa, U, Np, Pu, Am, Cm, Bk, Cf, Es, Fm,
Md, No, Lr, Rf, Db, Sg, Bh, Hs, Mt, Ds,
Rg, Cn, Nh, Fl, Mc, Lv, Ts, Og
```

3.1. Palavras-chave e Identificadores

A linguagem utiliza símbolos químicos como palavras-chave, tipos e comandos. Identificadores de usuário (variáveis, funções, objetos) devem começar com letra minúscula.

Palavra-chave	Função	Significado
P	Impressão (print)	Exibe mensagem ou valor
L	Entrada (input)	Lê valor para uma variável
Fn	Declaração de função	Define uma função
Rn	Retorno de função	Retorna valor
Cl	Declaração de classe	Define uma classe
Au, Cu	Condicional (if/else)	if, else
W	Repetição (while)	Laço while
V	Tipo void (sem retorno)	Indica ausência de retorno

3.2. Tipos Primitivos

A linguagem suporta tipos primitivos representados por siglas químicas:

Símbolo	Equivalente	Tipo
I	int	Inteiro

Fl	float	Real
Ch	char	Caractere
S	string	Cadeia
B	bool (O/N)	Booleano

Observação:Os valores booleanos são O (verdadeiro) e N (falso).

3.3. Arrays Genéricos

Vetores podem ser definidos por Ar<T>, onde T é um tipo primitivo:

```CSharp
Ar<I> numeros;
numeros[0] = 10;
```

3.4. Estruturas da Linguagem

Estrutura de decisão - IF/ELSE

Au (IF)

```
Au(var1 > 37){
        var2 += 40;
        P(“funciona”);
}Cu{
        P(“erro”);
}
```

Au	(	var1	>	37
)	{	var2	+=	40
;	P	(	“funciona”	)
;	}	Cu	{	P
(	“erro”	)	}	

Identificadores	var1, var2

Constantes (númericas)	37, 40
constantes (cadeias)	“Funciona”
operadores	>, +=
palavras-chave	Au, Cu, P

Classe	Padrão	Sigla
Identificadores	Cadeia de caracteres começando com letra minúscula	id
constantes (numéricas)	Sequência de digitos	nun
constantes (cadeias)	Cadeia de caracteres envolta por aspas	cadeia
operadores	o próprio lexema	op
palavras-chaves	o próprio lexema	o próprio lexema

- <Au>
- <id, “var1”>
- <id, “var2”>
- <num, 37>
- <num, 40>
- <cadeia, “funciona”>
- <Cu>
- <P>
- <cadeia, “erro”>

Estrutura de Repetição - WHILE

W (While)

```
W ( var1 < 3 ) {
        P ( "rodando" ) ;
        var1 += 1 ;
}
```

W	(	var1	<	3
)	{	P	(	“rodando”
)	;	var1	+=	1
;	}			

Identificadores	var1
Constantes (númericas)	3, 1
constantes (cadeias)	“rodando”
operadores	<, +=
palavras-chave	W, P

Classe	Padrão	Sigla
Identificadores	Cadeia de caracteres começando com letra minúscula	id
constantes (numéricas)	Sequência de digitos	nun
constantes (cadeias)	Cadeia de caracteres envolta por aspas	cadeia
operadores	o próprio lexema	op
palavras-chaves	o próprio lexema	o próprio lexema

<W>
<id, "var1">
<num, 3>
<cadeia, "rodando">
<id, "var1">
<num, 1>
<P>
<op, "<">
<op, "+=">

Tipo V (Void): Usado para funções sem retorno.

Exemplo:

Fn V exibir() { P("Texto"); }

Comando de entrada (L): Similar ao scanf, usado para ler valores do usuário.

Exemplo:

L(valor);

Comando de saída (P): Equivalente ao print, permite exibir mensagens e valores.

Exemplo:

P("Resultado:"); P(resultado);

Classes (Cl): A linguagem agora permite a definição de classes com atributos e métodos.

Exemplo:

```
CSharp
Cl Pessoa {
    S nome;
    I idade;

    Fn S falar() {
        Rn "Olá!";
    }
}
```

Chamadas de método com objeto.metodo(): Adicionada capacidade de fazer chamadas como p.falar();.

Arrays genéricos (Ar<T>): Vetores podem ser declarados com tipo específico.

Exemplo:

Ar<I> numeros; numeros[0] = 10;

3.5. Regras Léxicas (Tokens)

Classe	Padrão	Exemplo
Identificadores	Iniciam com letra minúscula ([a-z])	var1, texto
Números	Dígitos ([0-9]+)	37, 40, 1
Cadeias	Entre aspas ("...")	"teste", "erro"
Caracteres	Entre aspas simples ('c')	'z'
Booleanos	O e N	O, N
Operadores	+, -, *, /, =, +=, <, >	+=, <
Palavras-chave	Letras maiúsculas conforme gramática	P, L, Au, W

3.6. Exemplo de código

Exemplo do código processado no Program.cs para criação de classes:

```
CSharp
Cl Pessoa {
         S nome;
        I idade;

       Fn S falar() {
       Rn ""Olá!"";
    }
 }
Pessoa p;
P(p.falar());

Ar<I> notas;
 notas[0] = 10;
 P(notas[0]);
```

Seguir o link para testes de todas as funcinalidade :
https://docs.google.com/document/d/1tqecGv35MaYYQSPnKf-Ju9FjjEiSdjUvWh8HlySgLIo/edit?usp=sharing

4- Análise Léxica, Sintática e Semântica

Após a geração da gramática com o ANTLR, o compilador executa três fases centrais da análise do código-fonte: léxica, sintática e semântica. Essas fases garantem que o programa seja bem formado, coerente e semanticamente válido antes de gerar o código intermediário LLVM.

4.1 Análise Léxica (Lexer)

- **Objetivo**

Converter o código-fonte em uma sequência de tokens, que são os menores elementos significativos da linguagem.

- **Implementação**

Arquivo gerado: ElementLexer.cs

Baseado nas regras léxicas definidas no arquivo Element.g4.

- **Exemplos de tokens definidos**

Token	Representação	Exemplo
KW_FN	Palavra-chave de função	Fn
KW_RETURN	Palavra-chave de retorno	Rn
KW_PRINT	Impressão	P
KW_SCANF	Leitura	L
…	…	…

- **Características**

Ignora comentários (//, /* */) e espaços em branco.

Detecta erros como caracteres inválidos, nomes de variáveis ilegais, tokens malformados.

4.2 Análise Sintática (Parser)

- **Objetivo**

Validar se os tokens estão organizados de forma correta segundo a sintaxe da linguagem.

- **Implementação**

Arquivo gerado: ElementParser.cs regras definidas no arquivo Element.g4 (como prog, statement, expression, functionCall, etc).

- **Exemplo de regras**

```
None
functionCall: IDENTIFIER LPAREN argumentList? RPAREN;
returnStatement: KW_RETURN expression?;
```

```
classDeclaration: KW_CLASS IDENTIFIER block;
expression: logicalOrExpr;
```

- **Funcionalidades Suportadas**

1. Declaração e chamada de funções com ou sem retorno.
2. Atribuições, expressões aritméticas e booleanas.
3. Estruturas de controle (Au, Cu, W, Fe).
4. Vetores e acesso a índices.
5. Suporte completo a chamadas de métodos com objeto.metodo() após ajustes na gramática e listener.
6. Declaração de classes e variáveis de tipo classe.

- **Detecta erros como:**

1. Falta de ponto e vírgula.
2. Parênteses ou chaves desbalanceadas.
3. Uso incorreto de expressões ou estruturas.
4. Tokens em posições inválidas (Pessoap).

4.3 Análise Semântica

- **Objetivo**

Garantir a coerência lógica do programa:

1. Tipos compatíveis.
2. Variáveis e funções declaradas antes do uso.
3. Tipos corretos de retorno.
4. Chamada correta de métodos e funções.

- **Implementação**

Arquivo: SemanticAnalyzer.cs a árvore sintática é percorrida com ParseTreeWalker.

Verificações realizadas:

Categoria	Exemplo
Declaração antes do uso	x = 5; (erro se x não foi declarado)
Redeclaração	I x = 5; I x = 10; (erro)
Compatibilidade de tipos	I x = "texto"; (erro)
Retorno de funções	Fn I f() { Rn "a"; } (erro: retorna string, espera int)
Chamadas de funções	soma(1); (erro se soma espera 2 parâmetros)
Leitura e escrita	L(nome); P(idade); — verifica se variáveis

	foram declaradas
Vetores	vetor[0] = 5; — verifica tipo e declaração
Objetos e métodos	p.falar(); — detecta p como Pessoa e valida existência de falar

Além disso, no caso específico de vetores, foi adicionada uma verificação semântica que impede atribuições de valores incompatíveis com o tipo definido para o vetor. Isso evita erros em tempo de execução e reforça a integridade do sistema de tipos.

- **Suporte a escopo**

Variáveis dentro de funções ou blocos são armazenadas por escopo com uso de pilha (Stack<Dictionary<string, string>>). O escopo é empilhado ao entrar em blocos e desempilhado ao sair.

- **Exibe erros como:**

"Atribuição inválida: não é possível atribuir 'S' para variável do tipo 'I'."

"Função 'mostrarMensagem' chamada sem estar declarada."

"Função 'falar' chamada em objeto do tipo incorreto."

5- Geração de Código Intermediário LLVM

Após a análise semântica, o compilador realiza a geração de código:

- **LLVM IR (Intermediate Representation):**

Geração feita pelo arquivo LLVMCodeGenerator.cs. que percorre a árvore sintática gerada pelo ANTLR e traduz as estruturas da linguagem Element para instruções equivalentes em LLVM IR.

- **Funcionalidades Suportadas**

Tipos suportados:

I → i32 (inteiro)

Fl → double (float)

Ch → i8 (caractere)

S → ponteiros para i8 (string simulada)

B → i1 (booleano)

Ar<...> → arrays (com suporte básico)

Declaração, atribuição e leitura de variáveis.

- **Funções**

Suporte completo a funções com:

Retorno tipado: I, Fl, S, Ch, B

Retorno V (void)

Parâmetros com múltiplos tipos

Corpo com escopo local e retorno com Rn

Chamada a funções com verificação de número e tipo de argumentos.

- **Operações Suportadas**

Aritméticas: +, -, *, / com I e Fl

Relacionais: ==, !=, <, <=, >, >=

Booleanas: &&, ||, !

Conversão implícita de I para Fl quando necessário.

- **Comandos de Controle de Fluxo**

Au (...) { ... } — if

Cu { ... } — else

W (...) { ... } — while

Fe (...) { ... } — for

- **Entrada e Saída**

P(expr); → traduzido para chamada de printf

L(variavel); → traduzido para chamada de scanf

- **Classes e Objetos**

Declaração de classes com Cl

Instanciação de objetos com Classe nome;

Detecção básica do tipo da variável de instância

Suporte preliminar para chamada de métodos como obj.metodo() (parseado e reconhecido na semântica)

Arquivo: saida.ll.

6- Estrutura e Funcionamento do Compilador

6.1 O que é ANTLR?

ANTLR (Another Tool for Language Recognition) é uma ferramenta poderosa usada para gerar automaticamente analisadores léxicos e sintáticos a partir de uma gramática escrita em sua notação própria (.g4). Ele gera código em diversas linguagens, e neste projeto foi utilizado com saída para C#.

6.2 O que ele gera?

A partir do arquivo **Element.g4**, o ANTLR gera automaticamente:

- Lexer
  - Responsável por quebrar o código-fonte em unidades léxicas (tokens).
  - Por exemplo: reconhece palavras-chave como **I, Fl, P,** operadores como =, +, ;, e identificadores como **a**, **soma**.
- Parser
  - Responsável por organizar os tokens conforme as regras sintáticas da linguagem.
  - Valida se as sequências de tokens formam estruturas corretas (declarações, expressões, blocos, etc.).
- Listeners e Visitors
  - São classes que permitem percorrer a árvore gerada na análise sintática e implementar funcionalidades adicionais, como a **análise semântica**, geração de código, etc.

6.3 Fluxo de Execução

1. **Criação da Gramática (Element.g4)**
   a. Define as regras léxicas (tokens) e regras sintáticas da linguagem.
   b. Serve como entrada para o ANTLR gerar os analisadores.
   c. A gramática foi ampliada para suportar:
      i. Tipos primitivos (I, Fl, Ch, S, B
      ii. Vetores (Ar<T>)
      iii. Tipo V (void) para funções sem retorno
      iv. Classes (Cl) e métodos
      v. Comandos de leitura (L) e escrita (P)
      vi. Funções com ou sem retorno (Fn)
      vii. Chamadas do tipo p.método() com suportada pela geração de código
2. **Geração dos Analisadores com ANTLR**
   a. Comando básico no terminal para gerar arquivos C#:

```CSharp
java -jar antlr-4.13.2-complete.jar -Dlanguage=CSharp Element.g4 -o Generated
```

b. O que acontece:
   i. O ANTLR cria uma pasta Generated contendo:
   ii. ElementLexer.cs
   iii. ElementParser.cs
   iv. ElementBaseListener.cs
   v. ElementListener.cs
   vi. E outros arquivos auxiliares.

3. Implementação em C# (Program.cs)

O código C# realiza as seguintes etapas:

a. Fase Léxica

```CSharp
AntlrInputStream inputStream = new AntlrInputStream(code);
ElementLexer lexer = new ElementLexer(inputStream);
```

   i. O lexer recebe o código-fonte e converte em uma sequência de tokens.

b. Fase Sintática

```CSharp
CommonTokenStream tokens = new CommonTokenStream(lexer);
ElementParser parser = new ElementParser(tokens);
```

   i. O parser lê os tokens e valida se estão organizados de acordo com as regras da gramática (prog, statement, expression, etc.).
   ii. Gera uma árvore sintática (Parse Tree).

c. Remoção e Adição de Listeners de Erro

```CSharp
parser.RemoveErrorListeners();
parser.AddErrorListener(new ConsoleErrorListener<IToken>());
```

   i. Remove listeners padrão do ANTLR e adiciona um listener que imprime erros no console.

d. Parsing (Construção da Árvore Sintática)

```CSharp
var tree = parser.prog();
```

   i. Executa a regra inicial prog e gera a árvore sintática completa do código.

e. Geração da Árvore Sintática em JSON

```CSharp
var jsonTree = BuildJsonTree(tree);
File.WriteAllText("arvore.json", JsonSerializer.Serialize(jsonTree, new JsonSerializerOptions { WriteIndented = true }));
```

i. A árvore sintática é percorrida e transformada em uma estrutura JSON.
ii. O arquivo arvore.json é salvo e pode ser visualizado por qualquer leitor de JSON, ajudando a entender a estrutura do código analisado.

4. Análise Semântica (SemanticAnalyzer.cs)

```CSharp
var walker = new ParseTreeWalker();
var semanticAnalyzer = new SemanticAnalyzer();
walker.Walk(semanticAnalyzer, tree);
```

a. O walker percorre a árvore sintática.
b. A classe SemanticAnalyzer verifica:
   i. Se as variáveis foram declaradas antes de serem usadas.
   ii. Se funções são chamadas corretamente após serem declaradas.
   iii. Se não há variáveis ou funções duplicadas no mesmo escopo.
   iv. Se tipos de dados são compatíveis em atribuições
   v. Se o número e tipo dos argumentos de funções está correto
   vi. Se vetores são utilizados e indexados corretamente
   vii. Se chamadas de métodos (obj.metodo()) seguem a estrutura parcial implementada
   viii. Se o tipo V (void) é respeitado
   ix. Se return está presente quando necessário
c. Se erros forem encontrados, eles são exibidos no console.

5. Geração de Código Intermediário LLVM (LLVMCodeGenerator.cs)

```CSharp
var llvmGen = new LLVMCodeGenerator();
walker.Walk(llvmGen, tree);
File.WriteAllLines("saida.ll", llvmGen.LLVMCode);
```

a. A última etapa gera código intermediário no formato LLVM IR, mais próximo de código de máquina.
b. O LLVM é salvo no arquivo saida.ll com instruções como:

```None
%a = alloca i32
store i32 5, i32* %a
```

```
%tmp = load i32, i32* %a
```

6. Resultado Final no Console
    a. O console exibe:
        i. O código-fonte processado.
        ii. Mensagem sobre a geração bem-sucedida do arquivo arvore.json.
        iii. Se houve ou não erros semânticos.
        iv. Mensagem de sucesso para geração LLVM
            1. saida.ll
        v. Mensagem final: "Processamento concluído."

7- Passo a Passo da Execução Prática

1. Gerar os analisadores com ANTLR:

```
CSharp
java -jar antlr-4.13.2-complete.jar -Dlanguage=CSharp Element.g4 -o Generated
```

2. Compilar o projeto com o .NET:

```
CSharp
dotnet build
```

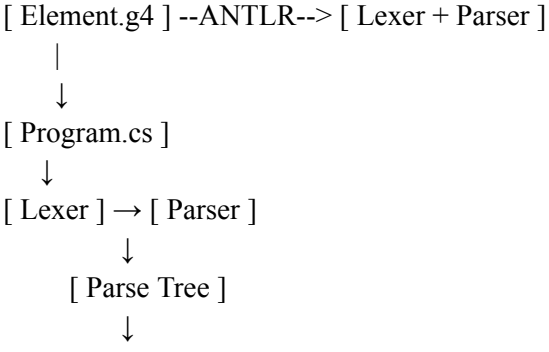
3. Executar o projeto:

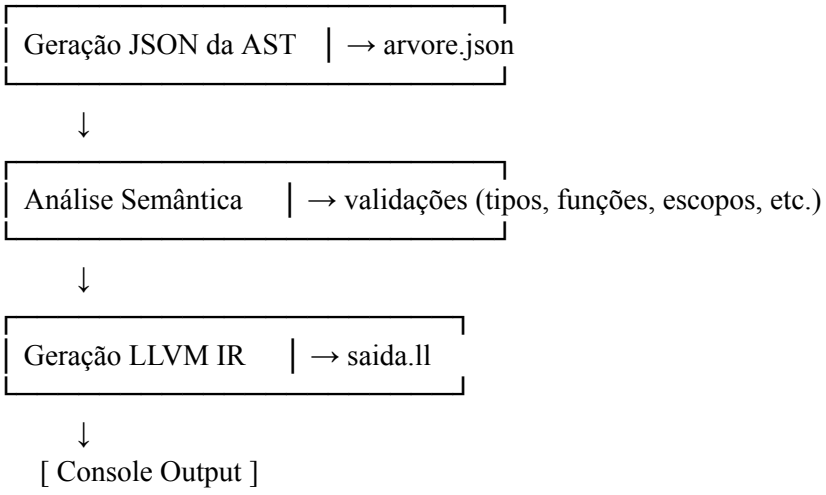
```
CSharp
dotnet run
```

4. Verificar:

- Saída no console com erros e mensagens
- Arquivo arvore.json com a árvore gerada

Resumo visual do Processo





8- Estrutura dos Arquivos

Arquivo	**Descrição**
Program.cs	Arquivo principal, executa lexer, parser, AST e análise semântica.
Element.g4	Gramática da linguagem Element.
SemanticAnalyzer.cs	Implementa a análise semântica.
arvore.json	Arquivo gerado contendo a AST.
LLVMCodeGenerator.cs	Traduz a AST para código LLVM IR. Suporta variáveis, funções, aritmética, if, while, chamadas de função e return

9- Facilidade na Implementação

- **Uso do ANTLR:**

O ANTLR facilitou muito a geração do lexer e parser, evitando implementar manualmente a análise léxica e sintática. O uso dele foi essencial para manter o código modular, especialmente com a adição de escopos aninhados e métodos.

- **Árvore Sintática em JSON:**

A geração em JSON permite visualizar claramente a estrutura da árvore, facilitando a depuração e compreensão.

- **C# com boa integração:**

A linguagem C# possui bibliotecas poderosas para manipulação de arquivos e JSON, acelerando o desenvolvimento.

10- Dificuldades Encontradas

- **Gerenciamento de Escopos:**

Inicialmente, controlar escopos para variáveis locais dentro de blocos, funções e estruturas condicionais foi um desafio.

- **Validação Semântica:**

A implementação de verificação de funções, especialmente parâmetros e chamadas, exigiu ajustes no rastreio do contexto correto.

- **ANTLR no C#:**

A configuração do ANTLR no ambiente .NET teve dificuldades, especialmente na geração automática de arquivos .cs a partir do .g4.

- **Falta de tratamento de tipos:**

Atualmente não há checagem de compatibilidade de tipos nas expressões, o que seria um próximo passo natural.

- **Adição do tipo void (V):**

Exigiu tratamento especial para funções sem retorno na análise semântica.

- **Implementação de objeto.metodo():**

Desafiadora devido à necessidade de expandir a gramática e adaptar a inferência de tipo com estrutura de classes.

- **Manipulação de vetores (Ar<T>):**

Foi implementado suporte a vetores genéricos com tipo fixo (Ar<T>). A geração de código considera corretamente o tipo interno e o vetor é tratado como uma estrutura de tamanho fixo em LLVM.

11- Referências Bibliográficas

PARR, Terence. **The definitive ANTLR 4 reference.** 1. ed. Dallas; Raleigh: The Pragmatic Bookshelf, 2013. 978-1-934356-99-9.
https://dl.icdst.org/pdfs/files3/a91ace57a8c4c8cdd9f1663e1051bf93.pdf#sec.matching-identifiers