UNIVERSIDADE FEDERAL DO CARIRI

Curso: Ciências da Computação
Disciplina: Compiladores **Código:** CC0032
Professor(a): Laszlon Rodrigues da Costa
Semestre: 2025.2 **Data:** 26/01/2026

Projeto Final - O Compilador "Mini-LANG"

Leia as Instruções:

- O projeto deverá ser desenvolvido em grupos de 4 a 5 integrantes.
- A formação dos grupos é de livre escolha dos alunos.
- A relação com os nomes dos integrantes deve ser enviada ao professor até o dia [02/03/2026].
- O projeto final deverá ser entregue até o dia [18/03/2026] via email: laszlon.costa@ufca.edu.br.
- É necessário o uso de ferramentas de versionamento de código (Git/GitHub) para facilitar a colaboração.
- A lista de commits com os nomes dos integrantes deve ser enviada junto com o projeto final.

Visão Geral

O objetivo deste trabalho é o desenvolvimento prático de um **Transpilador** (Compilador Fonte-para-Fonte) para a linguagem **Mini-Lang**. Este projeto visa consolidar os conhecimentos teóricos adquiridos na disciplina, permitindo a aplicação das etapas de análise e processamento de linguagens formais.

A *Mini-Lang* é uma linguagem imperativa, procedural e estaticamente tipada, projetada para fins didáticos. O produto final do projeto será um software capaz de ler códigos escritos em Mini-Lang e traduzi-los automaticamente para uma linguagem de alto nível de mercado (como C, Java ou Python), que poderá então ser executada nativamente.

Arquitetura do Transpilador

O projeto deve seguir o pipeline clássico de complicação, dividido em Front-End (análise) e Back-end (síntese). As etapas obrigatórias são:

1. **Análise léxica (scanner)**: Responsável por ler o arquivo de entrada e transformar a sequência de caracteres em um fluxo de tokens (identificadores, palavras reservadas, literais e operadores). Tokens irrelevantes, como espaços e comentários devem ser descartados.

2. **Análise sintática (parser)**: Verifica se a sequência de tokens obedece à gramática EBNF especificada. Esta etapa deve produzir uma **Árvore Sintática Abstrata** (AST) que represente a estrutura hierárquica do programa.

3. **Análise semântica**: Validação da lógica, o transpilador deve percorrer a AST para garantir regras que a gramática não captura, como:
   - Verificar se variáveis foram declaradas antes do uso.
   - Verificar compatibilidade de tipos em atribuições e operações (ex: impedir soma de um `bool` e um `int`).

- Garantir que funções sejam chamadas com o número de tipos corretos de argumentos.

4. **Geração de Código**: A etapa final, consiste em traduzir a AST validada para uma linguagem destino de executável.
   - Linguagens recomendadas: Python, C, C++ ou Java.
   - O código gerado deve ser funcional e executável, produzindo a saída correta.

A Lingugagem Mini-Lang

Para este projeto, a gramática apresentada é descrita utilizando a notação EBNF (*Extended Backus-Naur Form*). Ela é uma extensão do BNF clássica que permite descrever regras gramaticais de forma mais concisa e legível.

Como traduzir ENBF para BNF (Gramática Convencional)

1. Traduzindo Repetições: `{ A }` A notação `{ X }`, significa uma lista de X's (podendo ser vazia). Em EBNF, isso vira uma recurssão $S \rightarrow A\ \{B\}\ C$, já em BNF isso seria escrito como:

$$\begin{aligned} S &\rightarrow AXC \\ X &\rightarrow BX \mid \epsilon \end{aligned}$$

2. Traduzindo Opcionalidade: `[ A ]`: A notação `[ X ]`, indica que X pode aparecer ou não. Em EBNF, isso seria representado como:

$$S \rightarrow \text{if}(E)S\,[\text{else } S]$$

.

Já em BNF escrevemos:

$$\begin{aligned} S &\rightarrow \text{if}(E)SX \\ X &\rightarrow \text{else } S \mid \epsilon \end{aligned}$$

3. Traduzindo Agrupamento: (A | B): Os parênteses servem para controlar a prioridade das escolhas. Em EBNF

$$\text{Factor} \rightarrow (+|-)\ \text{Term},$$

em BFN:

$$\begin{aligned} \text{Factor} &\rightarrow \text{Op Term} \\ \text{OP} &\rightarrow + \mid - \end{aligned}$$

Exemplo de código

```
var x : int = 5;
var resultado : int = 1;

def calcular(n : int) : int {
  if (n > 0) {
    return n * calcular(n - 1);
```

```
    }
    return 1;
}

print "Calculando Fatorial de 5:";
set resultado = calcular(x);
print resultado;

```

Listing 1: Exemplo de Código Fonte em Mini-Lang

Estrutura Gramatical EBNF

```
<program>           = { <statement> }

<block>             = "{" { <statement> } "}"

<statement>         = <variable-decl> ";"
                    | <assignment> ";"
                    | <print-statement> ";"
                    | <if-statement>
                    | <while-statement>
                    | <return-statement> ";"
                    | <function-decl>
                    | <block>

<function-decl>     = "def" <identifier> "(" [<formal-params>] ")" ":" <type> <block>

<formal-params>     = <formal-param> { "," <formal-param> }
<formal-param>      = <identifier> ":" <type>

<while-statement>   = "while" "(" <expression> ")" <block>
<if-statement>      = "if" "(" <expression> ")" <block> ["else" <block>]

<return-statement>  = "return" <expression>
<print-statement>   = "print" <expression>
<type>              = "int" | "real" | "bool" |  "void"

<variable-decl>     = "var" <identifier> ":" <type> "=" <expression>
<assignment>        = "set" <identifier> "=" <expression>

<expression>        = <simple-expression> { <relational-op> <simple-expression> }
<simple-expression> = <term> { <additive-op> <term> }
<term>              = <factor> { <multiplicative-op> <factor> }

<factor>            = <literal>
                    | <identifier>
                    | <function-call>
                    | <sub-expression>
                    | <unary>

<unary>             = ( "+" | "-" | "not" ) { <expression> }
<sub-expression>    = "(" <expression> ")"
<function-call>     = <identifier> "(" [<actual-params>] ")"
<actual-params>     = <expression> { "," <expression> }

<relational-op>     = "<" | ">" | "==" | "!=" | "<=" | ">="
<additive-op>       = "+" | "-" | "or"
<multiplicative-op> = "*" | "/" | "and"

<identifier>        = ( "_" | <letter> ) { "_" | <letter> | <digit> }
<digit>             = [0-9]
<letter>            = [a-zA-Z]
<literal>           = <integer-literal> | "true" | "false"
<integer-literal>   = [0-9]+

```

Entregas e pontuações

O projeto será analisado utilizando 4 critérios.

Denifição e reconhecimento linguagem (20% da nota)

- Objetivo: Definir a linguagem e identificar os Tokens.
- Entrega:

  1. Código do Analisador Léxico (Scanner).
  2. O programa deve ler um arquivo fonte e imprimir a lista de tokens (ex: `<ID, numero>`, `<Assing, => >`).
  3. Arquivo contendo a especificação formal da gramática.
  4. Não será aceito o uso de ferramentas automáticas de geração de analisadores léxicos (ex: JFlex, Lex, etc). O código do scanner deve ser implementado manualmente ou utilizando bibliotecas que não gerem código automaticamente.

A estrutura da linguagem (30% da nota)

- Objetivo: Reconhecer a estrutura hierárquica do código.
- Entrega:

  1. Código do Analisador Sintático (Parser), funcionando a partir dos tokens gerados pelo Analisador Léxico (Scanner).
  2. Implementação das classes da Árvore Sintática Abstrata (AST).
  3. O programa deve ler o código e exibir a árvore visuamente (ou em formato de texto indentado/JSON).
  4. Tratar de erros sintáticos básicos (ex: “Esperado um ‘;’ n linha 20”).
  5. Não será aceito o uso de ferramentas automáticas de geração de analisadores sintáticos (ex: Yacc, Bison, ANTLR, etc). O código do parser deve ser implementado manualmente ou utilizando bibliotecas que não gerem código automaticamente.

Tratamento de erros semânticos (25% da Nota)

- Objetivo: Garantir que o código fonte tenha sentido lógico.
- Entraga:

  1. Implementação da tabela de símbolos.
  2. O compilador deve barrar erros semânticos:
     - Uso de variável não declarada.
     - Declaração duplicada de variável no mesmo escopo.
     - Tipos incompatíveis (ex: somar `bool` com um `float`).

Funcionalidade prática do compilador (25% da Nota)

- Objetivo: Fazer o Código funcionar.
- Entrega:
  1. Geração do código final. O compilador deve traduzir da Mini-Lang para Python ou C.
     - Caso seja gerado um código em C ou C++, o mesmo deve ser compilado utilizando um compilador comercial e gerar um binário válido.
     - Caso seja gerado um código em python o mesmo deve ser executado sem erros.
  2. O arquivo gerado deve ser executável e produzir a saída correta para o usuário.

Considerações Finais

Recomenda-se o uso de Python no desenvolvido do projeto (pela facilidade com árvores e strings), mas C++ ou java serão aceitos.

- Nas últimas semanas, os grupos irão apresentar seu projeto em sala e ao final será feito um teste ao vivo:
  1. O professor fornecerá 3 códigos ”secretos”em mini-lang.
  2. Cada grupo irá compilar e executar esses códigos ao vivo.
  3. Critérios de sucesso:
     (a) Compilou sem cair (crash)?
     (b) Detectou erros semânticos (quando houver)?
     (c) O resultado da execução está correto?