UNIVERSIDADE FEDERAL DO TOCANTINS - CAMPUS PALMAS
CIÊNCIA DA COMPUTAÇÃO - BACHARELADO
COMPILADORES - PROF: Marcos Ferreira

Lucas Farias
José Lucas
Ester Arraiz
Henrique Noronha

COMPILADORES

Palmas de de 2025

Semana 1

1.1 Grupo

Como proposto na disciplina a formação dos grupos de trabalho, nosso grupo é formado pelos integrantes: Lucas Farias, José Lucas, Ester Arraiz e Henrique Noronha. A formação do grupo foi feita com base em afinidades técnicas e na disposição para colaboração para o desenvolvimento do projeto ao longo da disciplina.

1.2 Escolha da Linguagem

Após discussões, decidimos criar uma linguagem simplificada chamada MiniLang, que será implementada e compilada utilizando Python. A escolha da linguagem foi guiada pela necessidade de manter um escopo gerenciável, considerando o tempo disponível e a complexidade do projeto. MiniLang é uma linguagem imperativa com foco em simplicidade, projetada para suportar operações básicas, como:

- Declaração de variáveis (inteiros e strings).
- Operações aritméticas (adição, subtração, multiplicação e divisão).
- Estruturas de controle (condicionais if-else e laços while).
- Entrada e saída básica (leitura e impressão de dados).

1.3 Descrição da Linguagem

A linguagem foi projetada para ser simples, mas funcional o suficiente para permitir a escrita de programas básicos. Abaixo, apresentamos a descrição da linguagem e exemplos de programas que ilustram suas capacidades, Sendo apenas para fins educacionais e para a construção de um compilador, permitindo a prática de conceitos como a análise léxica, sintática e semântica e geração de código.

Semana 2

2.1 Definição da gramática

A definição formal de uma gramática formal é um processo estruturado para descrever a sintaxe de linguagens formais, definindo como serão as regras de produção para símbolos terminais ou não-terminais. Para a estruturação deste projeto utilizaremos a BNF(Backus-Naur Form), que é uma metassintaxe usada para descrever sintaxes de

linguagens de programação, que usa regras de produção para terminais e não terminais. Para esta gramática as regras são estruturadas de forma hierárquica para refletir a procedência de alguns casos como na aritmética, onde a ocorrência das operações tem extrema importância. Foram também adicionados os principais operadores lógicos como while e o if.

Python

```
<program> ::= <stmt_list>

<stmt_list> ::= <stmt> | <stmt> <stmt_list>

<stmt> ::= <assign_stmt>
         | <if_stmt>
         | <while_stmt>
         | <print_stmt>

<assign_stmt> ::= <id> "=" <expr> ";"

<if_stmt> ::= "if" "(" <expr> ")" "{" <stmt_list> "}"
            | "if" "(" <expr> ")" "{" <stmt_list> "}"
"else" "{" <stmt_list> "}"

<while_stmt> ::= "while" "(" <expr> ")" "{" <stmt_list>
"}"

<print_stmt> ::= "print" "(" <expr> ")" ";"

<expr> ::= <term> | <expr> "+" <term> | <expr> "-" <term>
```

```
<term> ::= <factor> | <term> "*" <factor> | <term> "/"
<factor>

<factor> ::= <number> | <id> | "(" <expr> ")"

<id> ::= <letter> { <letter> | <digit> }

<number> ::= <digit> { <digit> }

<letter> ::= "a" | "b" | ... | "z" | "A" | "B" | ... |
"Z"
<digit> ::= "0" | "1" | ... | "9"
```

2.2 Construção do autômato léxico

A partir da gramática gerada para a linguagem foi construído o autômato léxico, criando os tokens para identificadores, números, operadores, atribuidores e etc, para o discernimento deste por meio do regex.

Estado Final	Token Gerado	Descrição
q1	TOKEN_NUMBER	[0-9]*
q2	TOKEN_ID ou TOKEN_KEYWORD	Identificador ou palavra-chave*
q3	TOKEN_ASSIGN	=
q4	TOKEN_PLUS	+
q5	TOKEN_SEMICOLON	;
q6	TOKEN_LPAREN	(
q7	TOKEN_RPAREN	)

Estado	dígito	letra	_	=	+	;	(	)	espaço	outro
q0	q1	q2	q2	q3	q4	q5	q6	q7	q0	qE
q1	q1	qE	qE	q0	q0	q0	q0	q0	q0	qE
q2	q2	q2	q2	q0	q0	q0	q0	q0	q0	qE
q3	qE	qE	qE	qE	qE	qE	qE	qE	q0	qE
q4	qE	qE	qE	qE	qE	qE	qE	qE	q0	qE
q5	qE	qE	qE	qE	qE	qE	qE	qE	q0	qE
q6	qE	qE	qE	qE	qE	qE	qE	qE	q0	qE
q7	qE	qE	qE	qE	qE	qE	qE	qE	q0	qE
qE	qE	qE	qE	qE	qE	qE	qE	qE	qE	qE

Tokens Definidos:

- Identificadores (ID);
- Números inteiros (NUMBER);
- Operadores Aritméticos (PLUS, MINUS, TIMES, DIVIDE);
- Atribuição (EQUALS);
- Delimitadores (LPAREN, RPAREN, SEMICOLON);
- Palavras reservadas (IF, ELSE, WHILE, PRINT).

2.3 Paradigma de Programação

O projeto está sendo desenvolvido em Python para a compilação de uma linguagem similar a ele. Portanto escolhemos o paradigma **imperativo/estruturado**, sendo imperativo porque o código descreve de forma direta as transformações a serem realizadas sobre a entrada e sendo estruturado pelo fato do compilador está dividido em módulos (lexer, parser, semantic, codegen e main).

Desta forma, teremos maior facilidade na manutenção e evolução do projeto visto que este é feito em equipe. Outro benefício é a legibilidade e clareza do código para os demais membro e futuros utilizadores do compilador, além de ajudar na implementação modular de diferentes features na etapa de desenvolvimento.

Semana 3

3.1 implementação do Analisador léxico

Analisador léxico implementado, usando gerador de analisadores PLY (python lex-yacc). As regras foram definidas por meio de expressões regulares, associando padrões de símbolos da linguagem aos respectivos tokens citados em

2.2.

```python
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_EQUALS  = r'='
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
t_SEMICOLON = r';'
t_PRINT = r'print'
```

Desta forma o analisador é capaz de transformar um programa fonte em uma sequência de tokens estruturados, que posteriormente alimentam o parser.

3.2 Teste e reconhecimento de tokens

Foram realizados testes com programas simples para validar a correta identificação dos tokens definidos. Exemplo de um programa de entrada:

```python
x = 10;
y = 20;
print(x+y);
```

Como esta saída do analisador léxico:

Como observado, o analisador reconheceu corretamente os identificadores, operadores, números, delimitadores e a palavra-chave PRINT.

Semana 4

4.1 Parser

O parser foi implementado utilizando a biblioteca PLY, que fornece ferramentas para análise sintática baseada em gramáticas definidas por regras de produção. Sua função é transformar a sequência de tokens gerada pelo analisador léxico em uma estrutura de árvore sintática abstrata (AST), representando a lógica do programa fonte. O parser reconhece instruções de atribuição, comandos de impressão e expressões aritméticas, respeitando precedência de operadores e agrupamento por parênteses. Ao identificar erros de sintaxe, o parser exibe mensagens informativas para facilitar a correção. A finalidade principal é permitir que o código fonte seja compreendido estruturalmente, servindo de base para etapas posteriores como análise semântica ou geração de código.

4.2 Técnica descendente ou ascendente.

Para o desenvolvimento deste compilador, optamos pelo uso de técnicas de análise descendente LL(1) por ser mais simples que análises ascendentes LR(1) e por outros motivos como:

- A gramática de uma linguagem mínima é tipicamente compatível com LL(1), exigindo menos esforço para projetar e implementar.
- A análise preditiva recursiva é intuitiva, fácil de codificar e alinha-se bem com projetos educacionais ou experimentais.

- A manutenção e depuração são mais simples, adequadas para um projeto de pequena escala.

Por isso, LL(1) é geralmente a melhor escolha para a implementação de um mini compilador para uma linguagem mínima. Caso a gramática apresente limitações (como recursão à esquerda), técnicas como eliminação de recursão ou fatoração podem ser aplicadas para torná-la LL(1). Para projetos mais complexos ou com gramáticas mais elaboradas, LR(1) poderia ser considerado, mas isso é improvável no contexto de uma linguagem mínima.

4.3 Representar Árvore sintática

Cada regra do parser está associada a uma ação semântica, responsável por construir a árvore sintática abstrata (AST) durante o processo de reconhecimento das estruturas. A AST representa hierarquicamente a lógica do programa, permitindo identificar operações, variáveis e valores de forma estruturada.

Semana 5

5.1 Associações semânticas

A etapa de tradução dirigida por sintaxe, as ações semânticas implementadas nas regras do parser não apenas constroem a AST, mas também podem ser estendidas para incluir informações de tipos e escopos, tornando a árvore anotada e apta para análises semânticas posteriores. Isso facilita a verificação de tipos, o controle de variáveis e a detecção de possíveis erros semânticos, além de preparar a AST para a geração de código.

5.2 Gerar AST

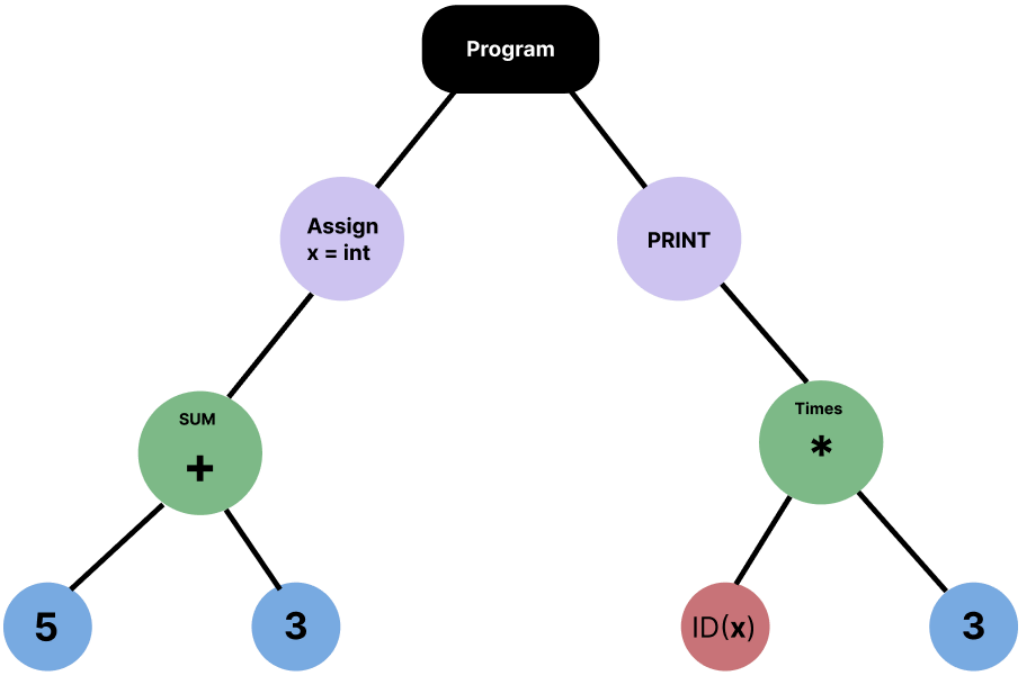
Para a implementação precisamos associar ações semânticas às regras da gramática definidas no parser e gerar uma Árvore de Sintaxe Abstrata (AST) anotada com informações de tipos e escopos. Abaixo, apresento um artefato com a implementação atualizada do parser, incluindo ações semânticas para construir a AST anotada.
Para o exemplo de código a seguir:

```
Python
x = 5 + 3;
print(x * 2);
```

Teremos a AST gerada:

```
None
program {'scope': 'global'}
  assign (x) {'type': 'int', 'scope': 'global'}
    + {'type': 'int'}
      num (5) {'type': 'int'}
      num (3) {'type': 'int'}
  print {'type': 'int', 'scope': 'global'}
    * {'type': 'int'}
      id (x) {'type': 'int', 'scope': 'global'}
      num (2) {'type': 'int'}
```

Formando (simplificadamente) a AST:



Semana 6

6.1 e 6.2 Transformar AST em código Intermediário

Para a geração de código intermediário utilizamos a AST denotada anteriormente em 5.2, mas somente para instruções que contém três operadores por enquanto. Dito isto, optamos por gerar um TAC, que é um Three-Access-Code, que processa no máximo três operadores por vez. Por Exemplo:

```python
x = 5 + 3;
print(x * 2);
```

Gera:

```
t1 = 5 + 3
x = t1
t2 = x * 2
print t2
```

Concluindo. No arquivo codegen.py são carregados os tokens e o parser gerado em parser.py, onde é construído as variáveis temporárias chamadas t1, t2 e t3 após construir a AST, montando claramente as operações realizadas.