

# Técnicas de Programação 2

## Implementação da Linguagem Oberon-0

Rodrigo Bonifácio

March 16, 2021

*Oberon-0 is an imperative language designed by Niklaus Wirth (??)*

*Oberon-0 is an imperative language designed by Niklaus Wirth (??)*

## Features

- declarações: constantes e variáveis
- tipos: primitivos (bool, int, ...), arrays, records, sets
- expressões: relacionais, aritméticas, booleanas, ...
- comandos: assignment, condicionais, repetição
- procedimentos: passagem por valor e por referência

*Oberon-0 is an imperative language designed by Niklaus Wirth (??)*

## Features

- declarações: constantes e variáveis
- tipos: primitivos (bool, int, ...), arrays, records, sets
- expressões: relacionais, aritméticas, booleanas, ...
- comandos: assignment, condicionais, repetição
- procedimentos: passagem por valor e por referência

Explorada em um desafio sobre novas tecnologias para *meta programação* (?).

*“We wanted [to explore] a language with a reasonable level of complexity but not a large language that had many features that would not illustrate the power of the tools”*

(?)

*“We wanted [to explore] a language with a reasonable level of complexity but not a large language that had many features that would not illustrate the power of the tools”*

(?)

Vamos explorar as construções da linguagem Scala e algumas técnicas para o **desenho** e **evolução** de uma implementação de Oberon-0

*“We wanted [to explore] a language with a reasonable level of complexity but not a large language that had many features that would not illustrate the power of the tools”*

(?)

Vamos explorar as construções da linguagem Scala e algumas técnicas para o **desenho** e **evolução** de uma implementação de Oberon-0—cuja implementação inicial seguiu o desafio de programação proposto por ?.

```
MODULE Multiples;

CONST
  limit = 10;

VAR
  base, count : INTEGER;
  mult : INTEGER;

PROCEDURE calcmult (i : INTEGER; base : INTEGER;
                   VAR result : INTEGER);
BEGIN
  result := i * base
END calcmult;

BEGIN
  Read (base);
  FOR count := 1 TO limit DO
    calcmult (count, base, mult);
    Write (mult);
    WriteLn
  END
END Multiples.
```

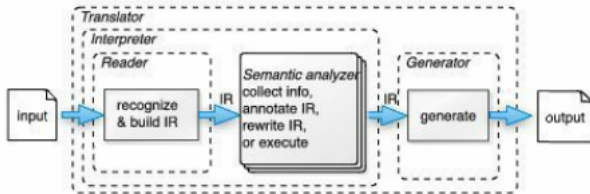


# Implementação de Linguagens

# Implementação de Linguagens

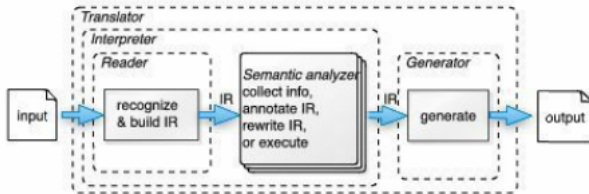


## Estilo Arquitetural



## Multistage Pipeline (?)

## Estilo Arquitetural



### Multistage Pipeline (?)

#### Componentes da Implementação Oberon-0

1. parser
2. análise semântica
3. (otimização de código)?
4. { interpretação | tradução }

# Parser

# Parser

Dois sub-componentes: **analisador léxico** e **analisador sintático**. O parser busca reconhecer um programa em uma determinada linguagem, e produzir como saída uma representação intermediária (tipicamente uma AST).

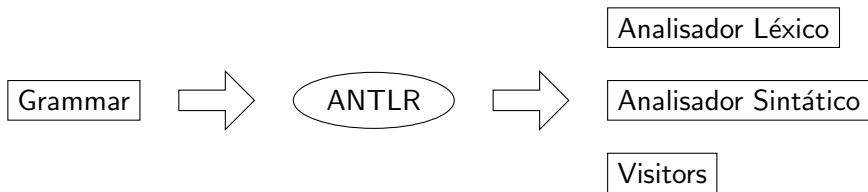
# Parser

Dois sub-componentes: **analisador léxico** e **analisador sintático**. O parser busca reconhecer um programa em uma determinada linguagem, e produzir como saída uma representação intermediária (tipicamente uma AST).

## Diferentes formas de implementação

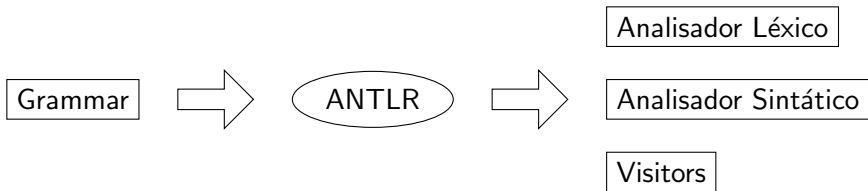
- *from scratch*
- usando uma biblioteca de combinadores
- usando um gerador de parser (Bison/YACC, BNFC, **ANTLR**, ...)

# ANTLR (Gerador de Parser)





## ANTLR (Gerador de Parser)



- Precisamos apenas transformar o resultado do parser ANTLR em uma representação intermediária (e independente do ANTLR). Justificativa: podemos alterar a gramática / estratégia de implementação do parser sem quebrar as demais fases do pipeline.

# Árvore Sintática Abstrata

Representação de um módulo de um programa mais fácil de ser manipulada em algum estágio de um compilador ou interpretador.

## Cenários de uso de uma AST

- verificação de tipos
- cálculo de métricas
- refatoramento de código
- interpretação de programas
- geração direta de código

# Árvore Sintática Abstrata

Representação de um módulo de um programa mais fácil de ser manipulada em algum estágio de um compilador ou interpretador.

## Cenários de uso de uma AST

- verificação de tipos
- cálculo de métricas
- refatoramento de código
- interpretação de programas
- geração direta de código

Alguns tipos de análise / transformações de programas são viáveis apenas em representações mais baixo nível (exemplo: three-address code)

# Interpretador

- Oberon é uma linguagem **imperativa**

- Oberon é uma linguagem **imperativa**: modelo computacional corresponde a uma sequencia de comandos que atualizam o **estado do programa**.

- Oberon é uma linguagem **imperativa**: modelo computacional corresponde a uma sequencia de comandos que atualizam o **estado do programa**.
- O estado do programa corresponde aos **mapeamentos** entre variáveis e constantes (tanto globais quanto locais) em valores (expressões)

- Oberon é uma linguagem **imperativa**: modelo computacional corresponde a uma sequencia de comandos que atualizam o **estado do programa**.
- O estado do programa corresponde aos **mapeamentos** entre variáveis e constantes (tanto globais quanto locais) em valores (expressões)—em um determinado instante.



## Representação do Estado

Classe `Environment` contendo:

## Representação do Estado

Classe `Environment` contendo:

- `globals`: `Map[String, Exp]` com as variáveis globais
- `locals`: `Stack[Map[String, Exp]]` com as variáveis locais
- `procedures`: `List[Procedure]` com as declarações de procedimentos

## Representação do Estado

Classe `Environment` contendo:

- `globals`: `Map[String, Exp]` com as variáveis globais
- `locals`: `Stack[Map[String, Exp]]` com as variáveis locais
- `procedures`: `List[Procedure]` com as declarações de procedimentos

e operações para indicar que um procedimento foi chamado, que uma procedimento retornou (concluiu a execução) e que uma atribuição foi realizada.

## Interpretação de um Programa Oberon

- (a) mapear as constantes globais no ambiente
- (b) mapear as variáveis globais no ambiente
- (c) mapear os procedimentos no ambiente
- (d) interpretar o bloco de comandos principal

## Interpretação de um Programa Oberon

- (a) mapear as constantes globais no ambiente
- (b) mapear as variáveis globais no ambiente
- (c) mapear os procedimentos no ambiente
- (d) interpretar o bloco de comandos principal
  - solução atual: **visitor** + **pattern matching**

```
trait OberonVisitor {  
  type T  
  var result : T = _  
  def visit(module: OberonModule) : Unit  
  def visit(constant: Constant) : Unit  
  def visit(variable: VariableDeclaration) : Unit  
  def visit(procedure: Procedure) : Unit  
  def visit(arg: FormalArg) : Unit  
  def visit(exp: Expression) : Unit  
  def visit(stmt: Statement) : Unit  
  def visit(aType: Type) : Unit  
}
```

```
case class OberonModule(  
  name: String ,  
  constants: List[Constant],  
  variables: List[VariableDeclaration],  
  procedures: List[Procedure],  
  stmt: Option[Statement]  
)  
{  
  def accept(v: OberonVisitor): Unit = v.visit(this)  
}
```

## Observação

- Em cada chamada de procedimento, precisamos:
  - (i) associar os argumentos atuais com os argumentos formais ★
  - (ii) atualizar a pilha com as declarações locais do procedimento
  - (iii) executar o bloco de comandos do procedimento.



## Observação

- Em cada chamada de procedimento, precisamos:
  - (i) associar os argumentos atuais com os argumentos formais ★
  - (ii) atualizar a pilha com as declarações locais do procedimento
  - (iii) executar o bloco de comandos do procedimento.
- Quando retornamos de um procedimento, precisamos **liberar** essas variáveis locais.

```
def call(p: Procedure, args: List[Expression]): Unit = {  
  p.args.map(formal => formal.name)  
    .zip(args)  
    .foreach(pair => env.setLocalVariable(pair._1, pair._2))  
  
  p.constants.foreach(c => env.setLocalVariable(c.name, c.exp))  
  p.variables.foreach(v => env.setLocalVariable(v.name, Undef()))  
}
```

# Referências I

# Técnicas de Programação 2

## Implementação da Linguagem Oberon-0

Rodrigo Bonifácio

March 16, 2021