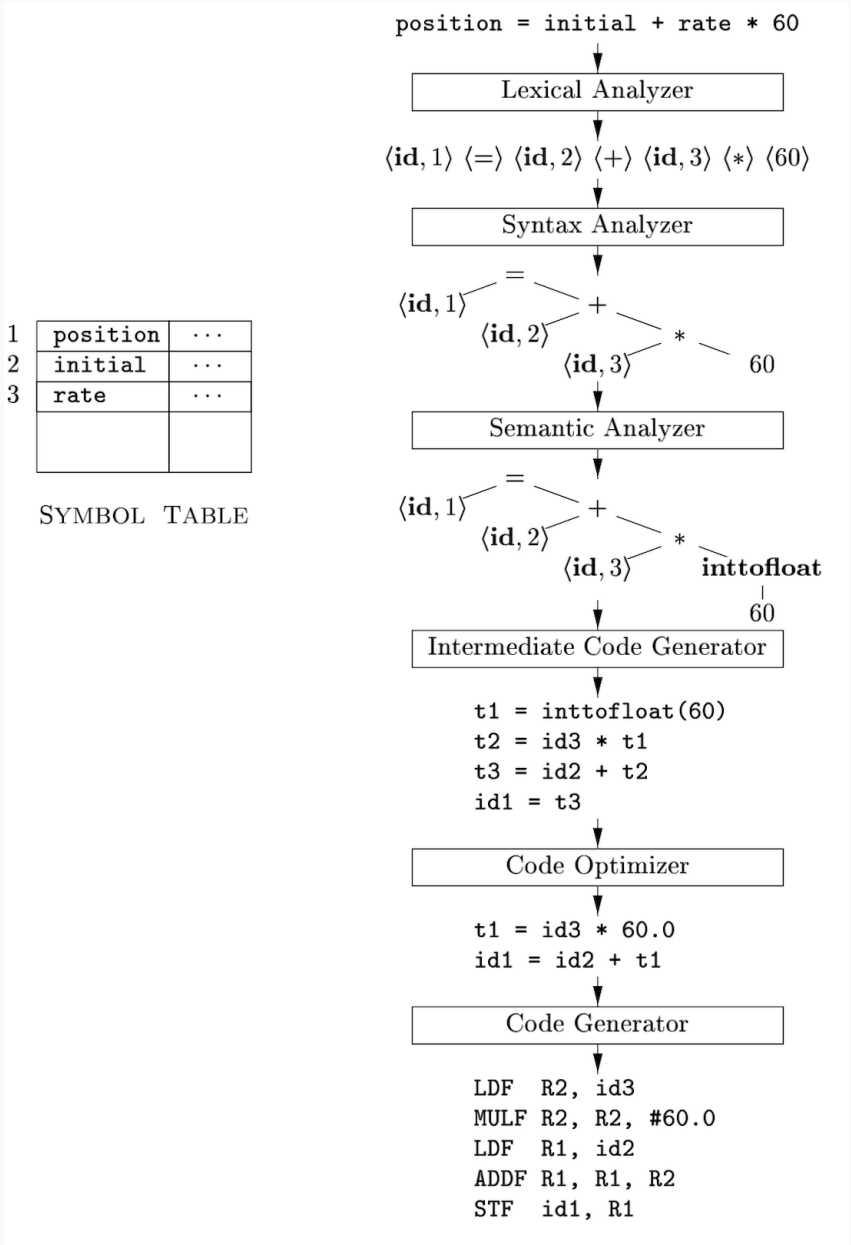
Compiladores: Introdução

Christiano Braga

Universidade Federal Fluminense

Janeiro 2021

A estrutura de um compilador



Aplicações da tecnologia de compilação

- Tradução de programas
  - Tradução binária
  - Síntese de hardware
  - Interpretadores de consultas à bancos de dados
- Ferramentas de produtividade
  - Análise estática de código
  - Análise de fluxo de dados
  - Verificação de tipos

- Escopo estático (ou léxico) e escopo dinâmico

```
x <- 1
f <- function(a) x + a
g <- function() {
  x <- 2
  f(0)
}
g()
```

Elementos básicos de LP > *Ambientes e estados* (ou memória) I

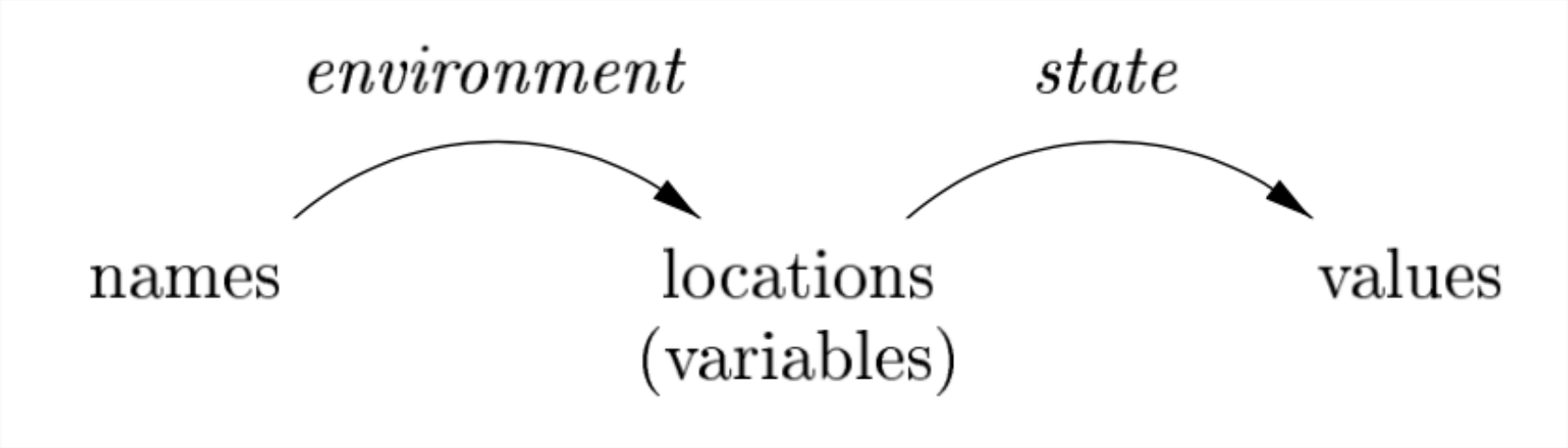


Figure 2: Ambiente e estado

Elementos básicos de LP > *Ambientes e estados* (ou memória) II

```
...
int i;                          /* global i        */
...
void f(···) {
      int i;                    /* local i         */
      ...
      i = 3;                    /* use of local i  */
      ...
}
...
      x = i + 1;                /* use of global i */
```

```
main() {
    int a = 1;                          B1
    int b = 1;
    {
        int b = 2;                      B2
        {
            int a = 3;                  B3
            cout << a << b;
        }
        {
            int b = 4;                  B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

- Parâmetros formais e atuais
- Chamada por valor
- Chamada por referência

Elementos básicos de LP > *Aliasing* (sinônimos)

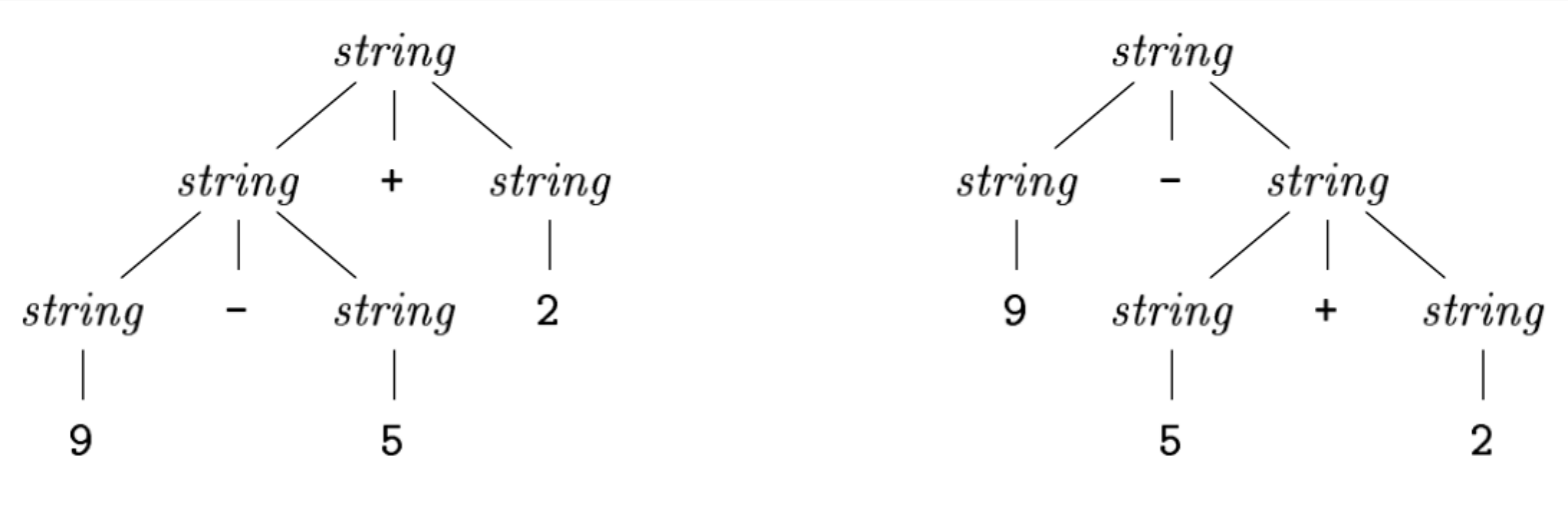
- Em C, o nome de um vetor é um ponteiro.
- Seja uma função `f(x, y)`.
  - Uma chamada `f(v, v)`, com `v` um vetor, faz com que `x` e `y` sejam sinônimos.
- Certas otimizações só podem ser feitas sob a garantia de inexistência de sinônimos.

- Gramáticas Livres de Contexto

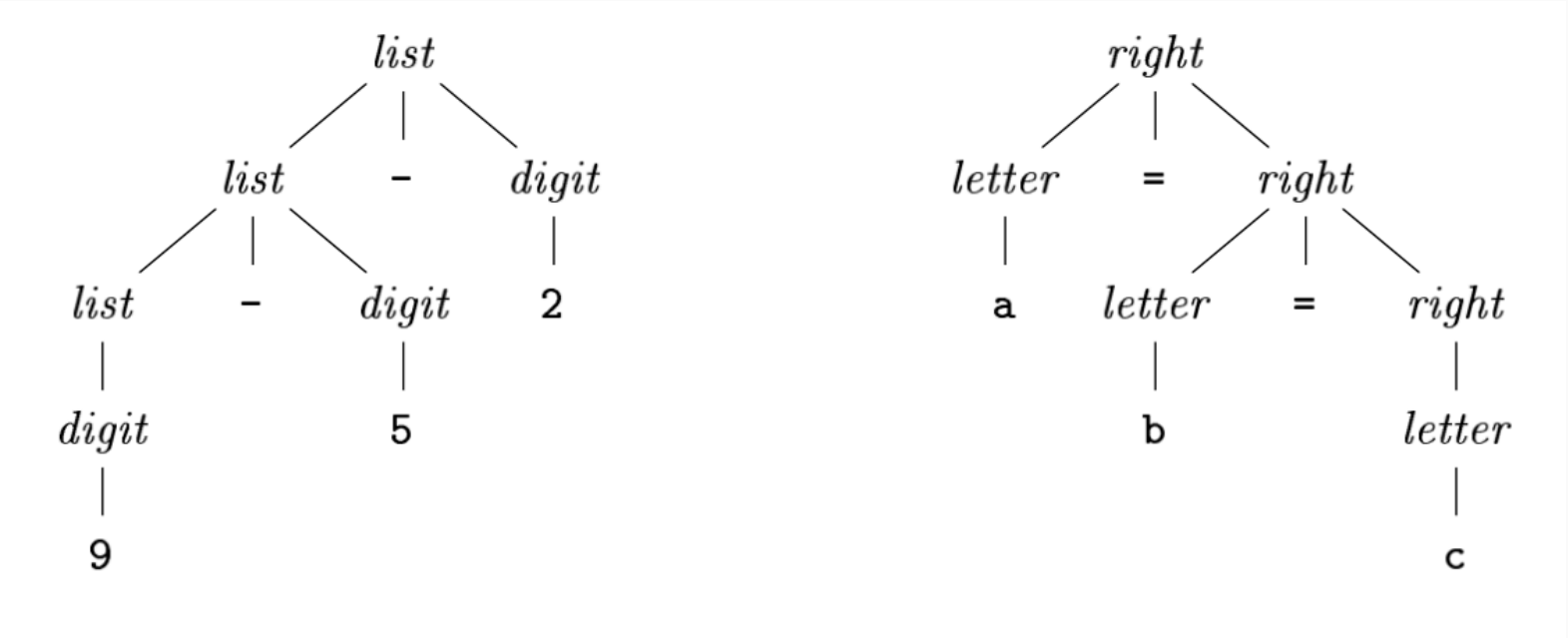
  list := list + list | list - list | digit

  digit := 0 | 1 | ... | 9

- Ambiguidade



- Associatividade de operadores



- Precedência de operadores

$$
\begin{array}{rcl}
expr & \rightarrow & expr + term \mid expr - term \mid term \\
term & \rightarrow & term * factor \mid term \,/\, factor \mid factor \\
factor & \rightarrow & \mathbf{digit} \mid (\ expr\ )
\end{array}
$$

  - Essa gramática é *recursiva à esquerda*, o que pode ser um problema para algoritmos de *parsing* preditivo.

- Eliminação de recursão à esquerda

$A := A\alpha \mid A\beta \mid \gamma$

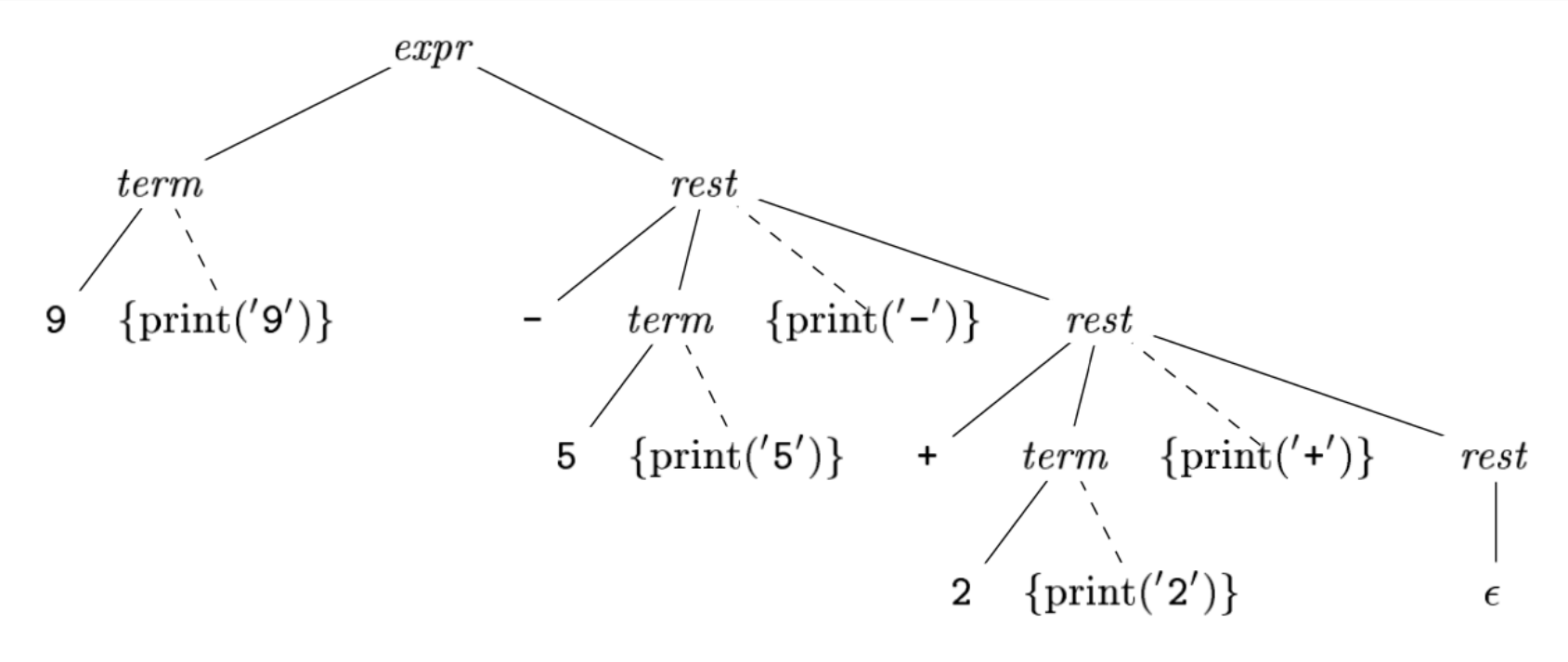
é transformada à

$A := \gamma R$

$R := \alpha R \mid \beta R \mid \epsilon$

Um tradutor de expressões > Esquema II

$$
\begin{array}{lcl}
\mathit{expr} & \rightarrow & \mathit{term}\ \mathit{rest} \\
\mathit{rest} & \rightarrow & \texttt{+}\ \mathit{term}\ \{\ \mathrm{print}(\texttt{'+'})\ \}\ \mathit{rest} \\
 & | & \texttt{-}\ \mathit{term}\ \{\ \mathrm{print}(\texttt{'-'})\ \}\ \mathit{rest} \\
 & | & \epsilon \\
\mathit{term} & \rightarrow & \texttt{0}\ \{\ \mathrm{print}(\texttt{'0'})\ \} \\
 & | & \texttt{1}\ \{\ \mathrm{print}(\texttt{'1'})\ \} \\
 & & \cdots \\
 & | & \texttt{9}\ \{\ \mathrm{print}(\texttt{'9'})\ \}
\end{array}
$$


expr
term
rest
9
{print('9')}
-
term
{print('-')}
rest
5
{print('5')}
+
term
{print('+')}
rest
2
{print('2')}
ϵ