



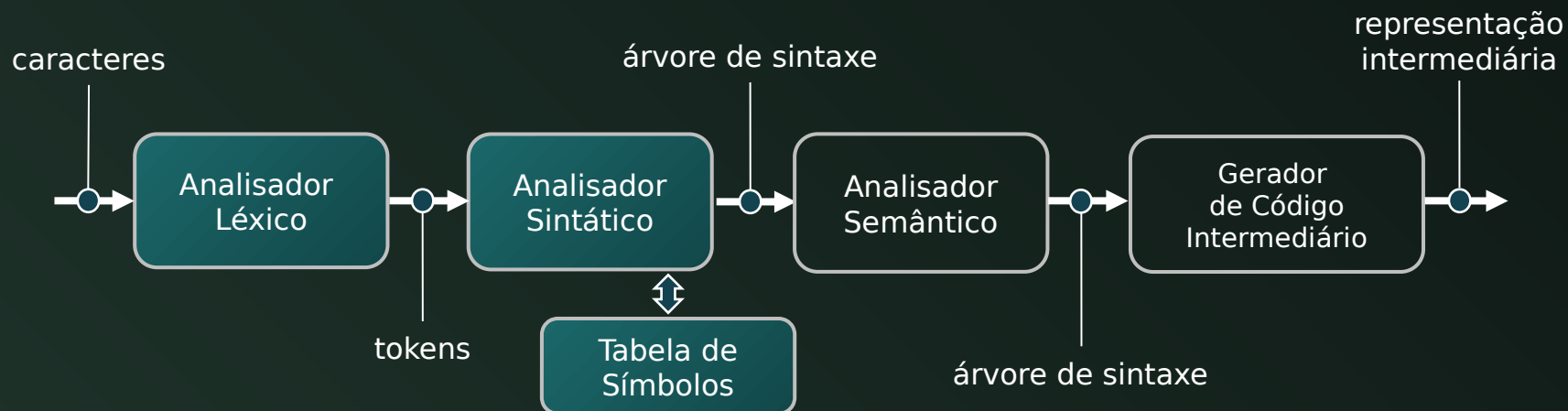
Judson Santos Santiago

Análise Semântica

Compiladores

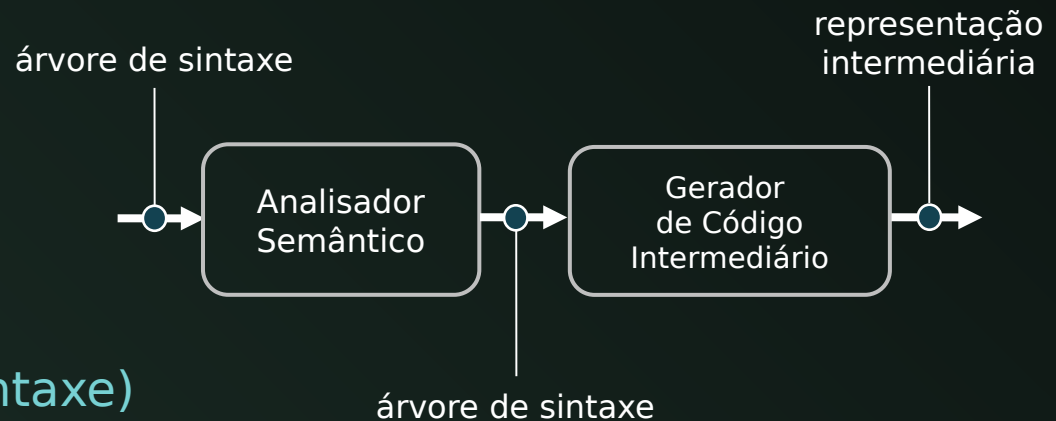
Introdução

- O **tradutor** implementado até agora suporta:
 - **Análise léxica**: identifica lexemas e gera tokens
 - **Análise sintática**: verifica se os tokens obedecem a gramática
 - **Tabela de símbolos**: implementa o escopo de variáveis em blocos



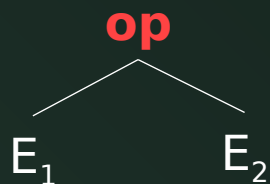
Introdução

- O *front-end* de um compilador verifica se um programa segue **as regras sintáticas e semânticas** da linguagem
 - Para isso ele constrói uma **representação** do código fonte
- As **mais importantes** são:
 - Árvores
Árvores sintáticas (ou árvores de sintaxe)
 - Representações lineares
Código de três endereços



Árvores Sintáticas

- Uma **árvore sintática** representa construções da linguagem
 - As construções são representadas por nós da árvore
 - Os componentes significativos das construções formam nós filhos
 - **Exemplo:**
 - Expressões
 - O operador **op** forma **um nó** e é interligado aos seus operandos E_1 e E_2



Árvore sintática

9 - 5 + 2



Árvores Sintáticas

- As árvores sintáticas podem ser criadas para **qualquer construção**

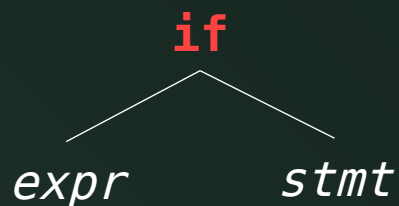
- **Exemplo:**

- Instruções

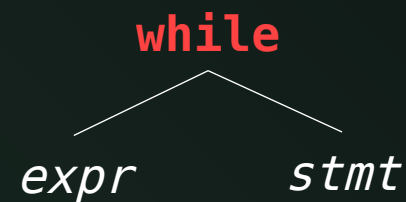
- A expressão condicional e as instruções são os **componentes significativos** do **if** e **while**

Ao contrário da **árvore de derivação**, a **árvore sintática** não precisa conter todos os símbolos da gramática.

if (*expr*) *stmt*








while (*expr*) *stmt*








Construção da Árvore Sintática

- O **esquema de tradução** abaixo constrói a árvore sintática para uma linguagem simples, com expressões e instruções

<i>program</i>	 <i>block</i>	{ return block.n; }
<i>block</i>	 { <i>stmts</i> }	{ block.n = stmts.n; }
<i>stmts</i>	 <i>stmts</i> ₁ <i>stmt</i>	{ stmts.n = new Seq(stmts ₁ .n, stmt.n); }
	ϵ	{ stmts.n = null; }
<i>stmt</i>	 <i>expr</i> ;	{ stmt.n = new Eval(expr.n); }
	if (<i>expr</i>) <i>stmt</i> ₁	{ stmt.n = new If(expr.n, stmt ₁ .n); }
	while (<i>expr</i>) <i>stmt</i> ₁	{ stmt.n = new While(expr.n, stmt ₁ .n); }
	 <i>stmt</i> ₁ while (<i>expr</i>);	{ stmt.n = new Do(stmt ₁ .n, expr.n); }

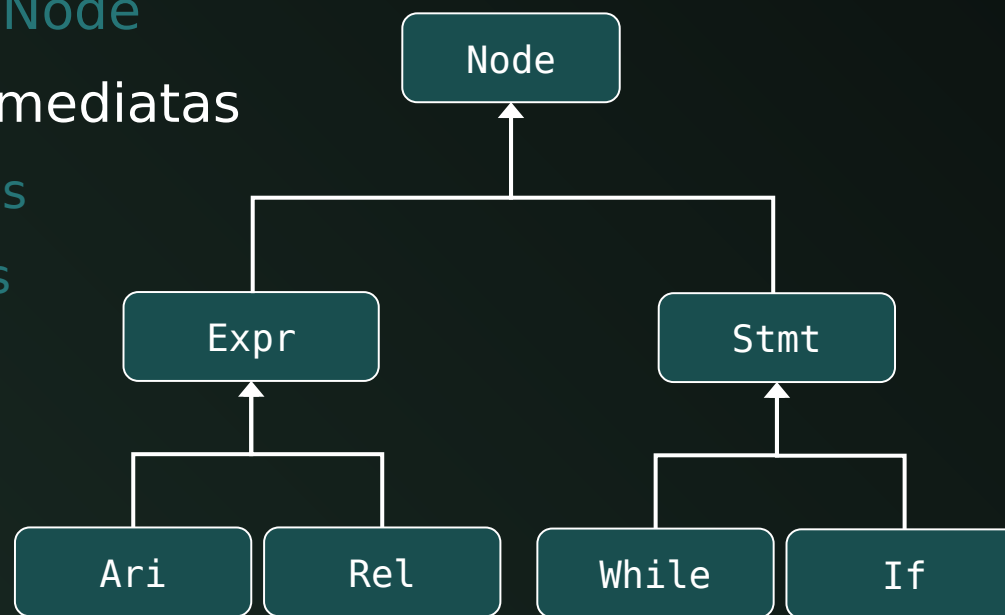
Construção da Árvore Sintática

continuação

<i>expr</i>	 <i>rel</i> = <i>expr</i> ₁ <i>rel</i>	{ <i>expr</i> .n = new Assign(<i>rel</i> .n, <i>expr</i> ₁ .n); } { <i>expr</i> .n = <i>rel</i> .n; }
<i>rel</i>	 <i>rel</i> ₁ < <i>ari</i> <i>rel</i> ₁ <= <i>ari</i> <i>ari</i>	{ <i>rel</i> .n = new Rel('<', <i>rel</i> ₁ .n, <i>ari</i> .n); } { <i>rel</i> .n = new Rel('<=', <i>rel</i> ₁ .n, <i>ari</i> .n); } { <i>rel</i> .n = <i>ari</i> .n; }
<i>ari</i>	 <i>ari</i> ₁ + <i>term</i> <i>term</i>	{ <i>ari</i> .n = new Ari('+', <i>ari</i> ₁ .n, <i>term</i> .n); } { <i>ari</i> .n = <i>term</i> .n; }
<i>term</i>	 <i>term</i> ₁ * <i>factor</i> <i>factor</i>	{ <i>term</i> .n = new Ari('*', <i>term</i> ₁ .n, <i>factor</i> .n); } { <i>term</i> .n = <i>factor</i> .n; }
<i>factor</i>	 (<i>expr</i>) <i>num</i>	{ <i>factor</i> .n = <i>expr</i> .n; } { <i>factor</i> .n = new Num(<i>num</i> .value); }

Construção da Árvore Sintática

- Todos os **não-terminais** do esquema de tradução possuem um **atributo n**, que é um nó da árvore sintática
 - Os nós são implementados pela **classe Node**
 - A classe Node possui duas subclasses imediatas
 - Expr – para todos os tipos de **expressões**
 - Stmt – para todos os tipos de **instruções**
 - Cada construção possui uma classe que é **subclasse** de **Expr** ou **Stmt**
 - **Expressões**: Ari, Rel, etc.
 - **Instruções**: While, If, etc.



Instruções

- Os **nomes das instruções** são usados como nomes das classes


```
if (expr) stmt1      { stmt.n = new If(expr.n, stmt1.n); }
```


```
while (expr) stmt1    { stmt.n = new While(expr.n, stmt1.n);  
                        }
```

```
do stmt1 while      { stmt.n = new Do(stmt1.n, expr.n); }  
(expr);
```

- Algumas instruções **não iniciam com palavras-chave**

- Neste caso é utilizada uma classe própria

```
stmt    expr;      { stmt.n = new Eval(expr.n); }
```

```
stmt    stmts1      { stmts.n = new Seq(stmts1.n, stmt.n); }  
s      stmt
```

Instruções

- As **declarações de variáveis** não precisam estar na árvore sintática
 - Elas geram entradas na tabela de símbolos

```
int w;           B0
{
  int x;         B1
  char y;
  {
    bool z;      B2
    x;
    y;
  }
  x;
  y;
}
```

Tabela de Símbolos

Id	Tipo	Acesso
w	int	global
x	int	local
y	char	local
z	bool	local

Instruções

- Um **bloco** é uma sequência de instruções

block  { *stmts* }

stmts  *stmts*₁ *stmt*

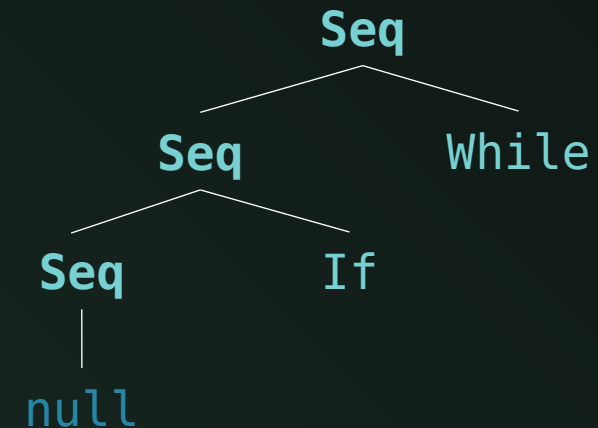
stmt   *block*

```
{ block.n = stmts.n; }
```

```
{ stmts.n = new Seq(stmts1.n,  
stmt.n); }
```

```
{ stmts.n = null; }
```

- Uma **sequência de instruções** sempre termina com vazio (nulo)





Expressões

- A precedência da multiplicação sobre a soma foi estabelecida pelo uso do símbolo não-terminal *term*
 - Uma vez derivada uma multiplicação, não é mais possível derivar uma soma
 - Isso garante que a multiplicação será feita primeiro
 - Ficará mais profunda na árvore de derivação

ari	 $ari_1 + term$
	$term$
term	 $term_1 * factor$
	$factor$

Expressões




- Para lidar com os operadores relacionais é preciso acrescentar mais um nível de precedência
 - Representada pelo não-terminal *ari*
 - Isso garante que as operações aritméticas serão feitas primeiro

<i>rel</i>	 $rel_1 < ari$
	$rel_1 \leq ari$
	ari
<i>ari</i>	 $ari_1 + term$
	$term$

Expressões

- A **sintaxe abstrata** nos permite **agrupar semelhantes**
 - Mesmas regras de verificação de tipos
 - Mesmas regras para geração de código

Ex.: operadores de soma (+) e multiplicação (*)

<i>rel</i>	 <i>rel</i> ₁ < <i>ari</i>	{ <i>rel.n</i> = new Rel('<', <i>rel</i> ₁ .n, <i>ari.n</i>); }
	<i>rel</i> ₁ <= <i>ari</i>	{ <i>rel.n</i> = new Rel('<=', <i>rel</i> ₁ .n, <i>ari.n</i>); }
	<i>ari</i>	{ <i>rel.n</i> = <i>ari.n</i> ; }
<i>ari</i>	 <i>ari</i> ₁ + <i>term</i>	{ <i>ari.n</i> = new Ari('+', <i>ari</i> ₁ .n, <i>term.n</i>); }
	<i>term</i>	{ <i>ari.n</i> = <i>term.n</i> ; }
<i>term</i>	 <i>term</i> ₁ * <i>factor</i>	{ <i>term.n</i> = new Ari('*', <i>term</i> ₁ .n, <i>factor.n</i>); }
	<i>factor</i>	{ <i>term.n</i> = <i>factor.n</i> ; }

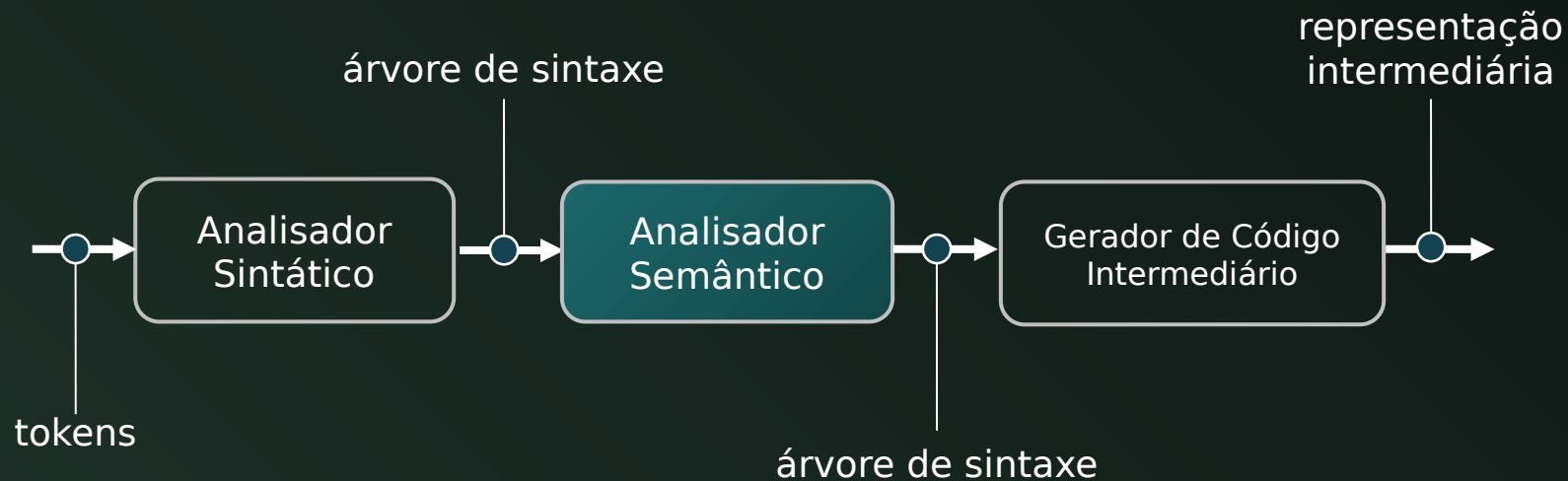
Expressões

- A tabela abaixo mostra **exemplos de agrupamentos** possíveis

Sintaxe Concreta	Sintaxe Abstrata
=	assign
	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	ari
* / %	ari
!	not
~unário	minus
[]	access

Análise Semântica

- O analisador semântico aplica métodos de verificação estática[†] em cima da árvore de sintaxe



Tempo de Compilação

- Se a linguagem fornece informações suficientes para que o compilador tome uma decisão, então essa questão pode ser decidida **em tempo de compilação**

- O tipo e escopo de uma variável
- A localização de memória[†] de uma **variável estática** (ou membro de classe)
- O valor de uma **expressão constante**

```
class Timer
{
    private:
        static int freq;
    ...
}
```

```
int x = 10 +
5;
```

Tempo de Execução

- Se alguma decisão não pode ser tomada durante a compilação, sendo necessário executar o programa, ela é uma decisão **em tempo de execução**
 - A localização de memória de objetos alocados dinamicamente
 - O resultado de uma **expressão não constante**
 - Expressão que depende de um valor fornecido pelo usuário
 - Expressão que depende de uma chamada de uma função complexa

```
#include <cstdlib>
int main()
{
    int y = 10 + rand();
    ...
}
```

Verificação Estática

- As verificações estáticas são **verificações de consistência** feitas durante a compilação
 - **Verificação sintática**: a sintaxe é mais que apenas uma gramática
Ex.: só pode existir um identificador por escopo, um break deve estar dentro de um laço, etc.
 - **Verificação de tipos**: as regras de tipo garantem que um operador ou função seja aplicado ao número e ao tipo correto de operandos
Ex.: se uma conversão for necessária, o verificador de tipos pode inserir uma conversão explícita na árvore sintática
- Vamos analisar algumas **verificações estáticas simples**[†]

Verificação Estática

- Existe uma distinção entre o significado dos identificadores que aparecem no lado esquerdo e no lado direito de uma atribuição
 - O lado direito especifica um valor (*valor-r*)
 - O lado esquerdo especifica um local de armazenamento (*valor-l*)

```
i = 5;      // i é um valor-l  
j = i + 1   // i é um valor-r
```

- A verificação estática precisa garantir que o lado esquerdo é um *valor-l*
 - Um *identificador*, como *i* ou *vet[2]*, possui um *valor-l*
 - Uma *constante*, como o número 5, possui apenas um *valor-r*

Verificação Estática

- A linguagem C++ permite **diferenciar referências** para:
 - Valores-R: uma constante ou expressão
 - Valores-L: uma variável

```
void func(int && v)
{
    cout << "R-value: " << v << endl;
}
```

```
void func(int & v)
{
    cout << "L-value: " << v << endl;
}
```

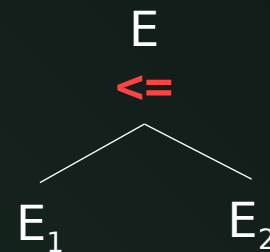
```
int main()
{
    int val = 5;
    func(val);
    func(5);
    ...
}
```

Verificação Estática

- A **verificação de tipo** garante o casamento entre o tipo esperado por uma construção e o tipo recebido
 - Na **instrução if** espera-se que *expr* tenha tipo booleano
- Em uma **operação relacional** ($E_1 \leq E_2$) espera-se que os dois **operandos** tenham o mesmo tipo e o **resultado** seja booleano

- O tipo pode ser verificado na construção do nó

```
if ( $E_1.type == E_2.type$ )  
     $E.type = bool$   
else  
    SemanticError();
```



Verificação Estática

- A ideia de **casar os tipos** se aplica mesmo nos casos:
 - **Coerção**: a linguagem especifica as coerções permitidas
Ex.: na expressão $2 * 3.14$, o inteiro é convertido em ponto flutuante
- No exemplo anterior, o operador \leq poderia aceitar tipos diferentes, se estes fossem convertíveis para o mesmo tipo
- **Sobrecarga**: o significado de um operador sobrecarregado é determinado considerando-se os tipos conhecidos de seus operando
Ex.: o operador $+$ pode ser adição de números ou concatenação de strings

Resumo

- A construção de **representações auxiliares** do código fonte permitem a realização da **análise semântica**
 - Árvores de sintaxe
 - Representam as construções das linguagens **através de uma árvore**
 - Cada construção vira um nó
 - Os componentes de cada construção são nós filhos
 - Ex.: **if** (*expr*) *inst*
 - Permitem a execução de **verificações estáticas** sobre o programa:
 - Verificações sintáticas
 - Verificações de tipo