



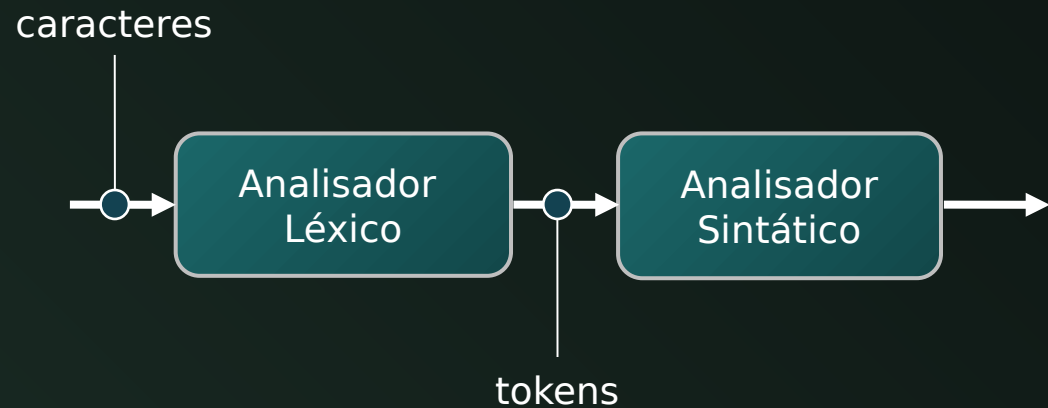
Judson Santos Santiago

Tabela de Símbolos

Compiladores

Introdução

- O tradutor implementado até agora suporta
 - Análise léxica:
 - Ignora espaços em branco
 - Reconhece números, identificadores e palavras-chave
 - Análise sintática
 - Verifica se a sequência de tokens obedece a gramática da linguagem



Introdução

- Os identificadores recebidos foram considerados sempre válidos

`<num,25> <+> <id,"val"> <-> <num,7>`

- Em uma linguagem mais complexa, os identificadores são:

- Declarados, possivelmente dentro de blocos
- Possuem escopo

```
int x;  
char y;  
{  
    bool y;  
    x;  
    y;  
}  
y;
```

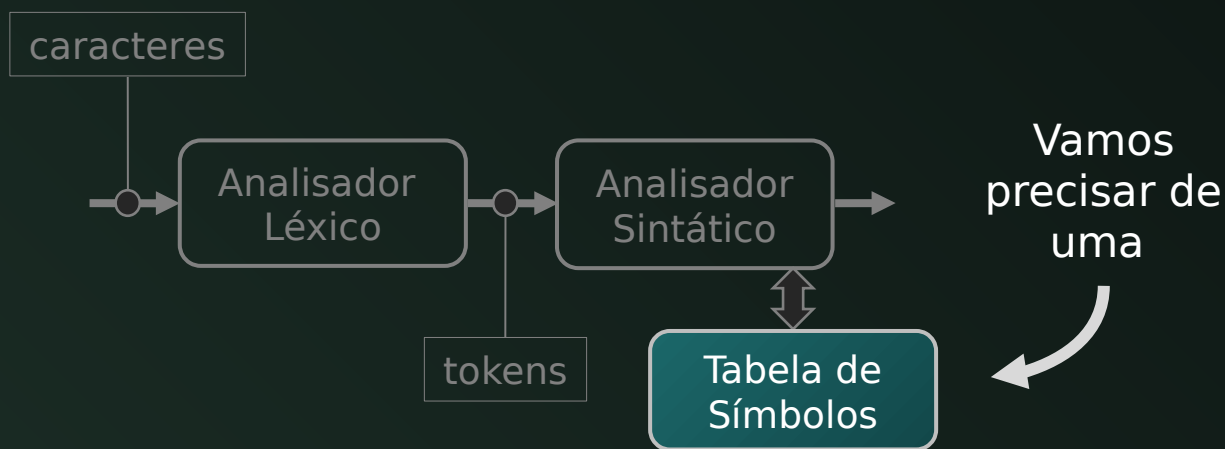


Tabela de Símbolos

- As tabelas de símbolos são **estruturas de dados** usadas para **guardar informações** sobre o programa fonte
 - São coletadas de forma incremental nas fases de análise
 - Usadas nas fases de síntese para gerar o código objeto



Tabela de Símbolos

- Os **dados armazenados** na tabela são:
 - Os nomes de identificadores (variáveis, funções, classes, etc.)
 - Suas informações associadas:
 - Tipo, escopo, parâmetros, endereço, acesso, etc.
- Com o **objetivo** de:
 - Mantê-los organizados para acesso rápido
 - Verificar se o identificador foi declarado
 - Implementar verificação de tipos
 - Determinar o escopo de um nome

Tabela de Símbolos

Id	Tipo	Acesso
x	int	public
y	char	public
z	bool	private

Escopo

- O **escopo de uma declaração** é a parte do programa à qual a declaração se aplica

```
{  
  int x;  
  char y; B1  
  {  
    bool y; B2  
    x;  
    y;  
  }  
  x;  
  y;  
}
```

Declaração	Escopo
int x;	B1
char y;	B1 - B2
bool y;	B2

Escopo

- Os escopos são implementados definindo-se uma **tabela de símbolos separada** para cada um deles
 - Cada bloco de programa com declarações terá sua própria tabela
 - Permite identificar o tipo correto da variável **em cada uso**
 - Utiliza-se a **regra do aninhamento mais interno**:

A declaração de uma variável é

encontrada examinando-se os blocos do mais interno para o mais externo

```
{ { x:int; y:bool; } x:int; y:char; }
```

B1:	x	int
	y	char
B2:	y	bool

```
{  
  int x;          B1  
  char y;  
  {  
    bool y;      B2  
    x;  
    y;  
  }  
  x;  
  y;  
}
```


Gramáticas com Blocos

- A implementação do escopo é mais interessante em gramáticas que possuem blocos aninhados

- As produções a seguir resultam em blocos aninhados:

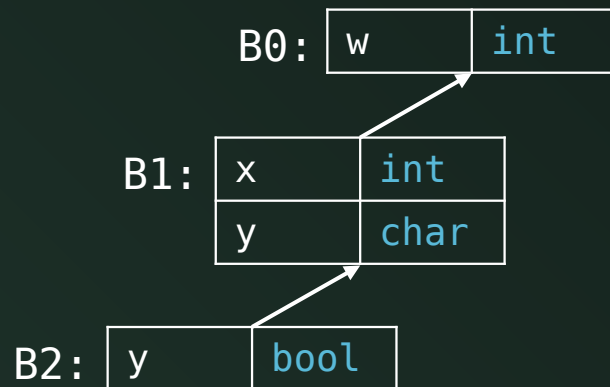
```
block { decls stmts }  
stmts stmts stmt  
stmt block
```

Exemplo de derivação:

```
block  
{ decls stmts }  
{ decls stmts stmt }  
{ decls stmts stmt stmt }  
{ decls stmts stmt block }  
{ decls stmts stmt { decls stmts } }  
...
```

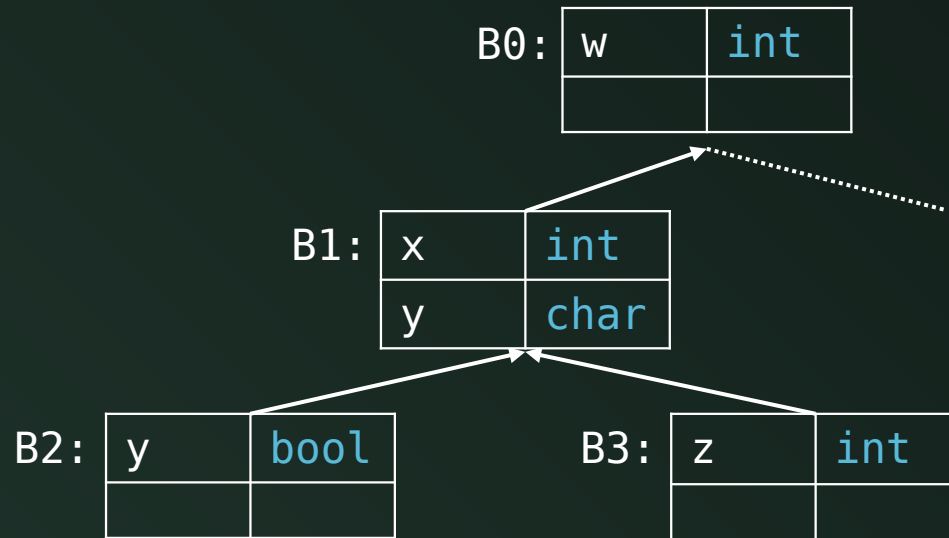

Encadeamento de Tabelas

- A regra de aninhamento mais interno pode ser implementada pelo encadeamento de tabelas de símbolos
 - A tabela do bloco interno aponta para a tabela do bloco mais externo
 - Mantém-se um ponteiro para a tabela atual:
 - Alterado na entrada e saída dos blocos



Encadeamento de Tabelas

- O encadeamento das tabelas **resulta em uma árvore**
 - Mais de um bloco pode estar aninhado dentro de outro



Encadeamento de Tabelas

- Essa técnica funciona para **outras construções**:
 - Classes
 - Cada uma tem sua própria tabela
 - Contém entradas para **atributos e métodos**

```
class Point
{
private:
    int x;
    int y;
public:
    Point() {
        x = 0;
        y = 0;
    }
}
```

x	int	private
y	int	private
Point	void	public

A classe é definida dentro de um bloco: as mesmas regras de escopo se aplicam

Ambientes e Estados

- A **execução** de um programa gera mudanças:

- Nos **valores** dos dados na memória

```
// muda o valor na posição de memória designada para x  
x = y + 1;
```

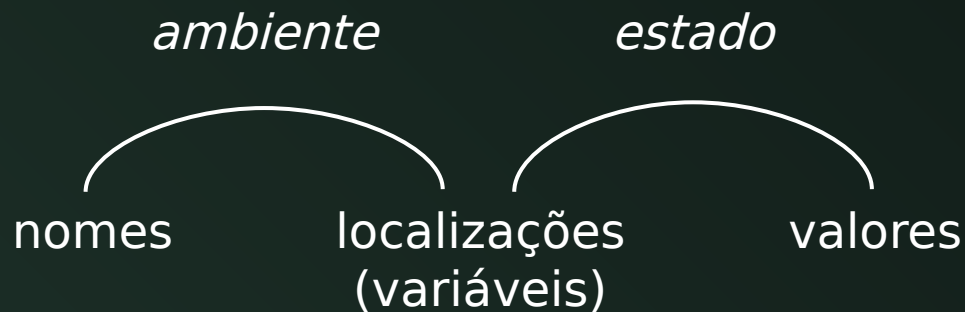
- Na **interpretação dos nomes** para os dados

- A localização associada ao nome “x” pode mudar durante a execução:

- Se x for um **atributo de uma classe**, cada objeto terá um endereço diferente para x
- Se x for uma **variável local**, cada chamada da função dará um endereço para x
- Se x estiver **presente em escopos diferentes**, terá um endereço em cada escopo

Ambientes e Estados

- A **associação de nomes e valores** à localizações de memória pode ser descrita por **dois mapeamentos**
 - **Ambiente**: mapeia um nome a uma posição de memória
 - **Estado**: mapeia um valor a uma posição de memória



Ambientes e Estados

- O **ambiente e estado mudam** com a execução do programa
 - **Ambiente:** muda de acordo com as **regras de escopo** da linguagem
 - **Estado:** muda conforme as variáveis **recebem novos valores**

```
int i; // variável global

void func()
{
    int i; // variável local
    i = 3; // uso do i local
}

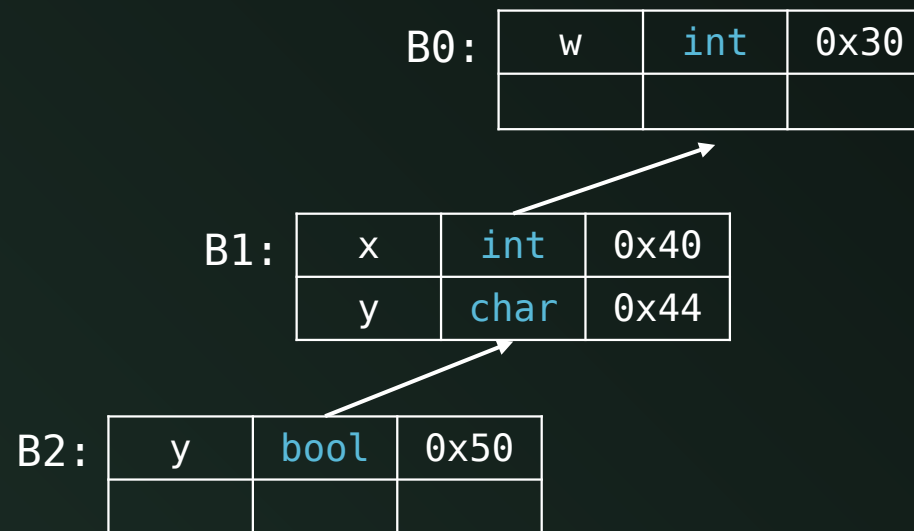
int main()
{
    func();
    int x = i + 1; // uso do i global
}
```

Quando `func()` está sendo executada, o **ambiente se ajusta** para que `i` se refira a localização reservada na pilha da função

Ambientes e Estados

- O **ambiente** é um apontador para uma **tabela de símbolos**
 - Ao analisar as linhas 5 a 9, o ambiente aponta para B2
 - Ao passar para a linha 10, o ambiente aponta para B1
 - B2 torna-se inacessível, mas pode-se alcançar a tabela global (B0)

```
01: int w;           B0
02: {
03:   int x;         B1
04:   char y;
05:   {
06:     bool y; B2
07:     x = 1;
08:     y = true;
09:   }
10:   x = 2;
11:   y = 'A';
12: }
```



Implementação

```
class SymTable
{
private:
    HashTable table;
    SymTable prev;
public:
    SymTable(SymTable p) { table = new HashTable(); prev = p; }
    void put(string s, Symbol sym) { table.put(s, sym); }
    Symbol get(string s)
    {
        for (SymTable st = this; st != null; st = st.prev)
        {
            Symbol found = st.table.get(s);
            if (found != null) return found;
        }
        return null;
    }
}
```


O pseudocódigo mostra
uma implementação
possível para a tabela
de símbolos

Usando a Tabela

- Como **incorporar a tabela de símbolos** ao analisador léxico e sintático já desenvolvidos?

- O analisador sintático preditivo é baseado em um esquema de tradução dirigido por sintaxe:


- **Ações semânticas** realizam as traduções

```
expr  expr + term { print('+') }  
    | expr - term { print('-') }  
    | term
```


- Elas podem também ser usadas para inserir e consultar símbolos na tabela

Usando a Tabela

- Uma ação semântica **entra com informações** sobre o identificador
 - Quando uma **declaração** desse identificador é analisada








```
decl  type id; { s = new Symbol;  
                    s.type = type.lexeme;  
                    symTable.put(id.lexeme, s); }
```

- Uma ação semântica **recupera informações** sobre o identificador
 - Quando o identificador for **usado** no programa

```
fact  id { s = symTable.get(id.lexeme);  
        print(id.lexeme);  
        print(':');  
        print(s.type); }
```

Usando a Tabela

- O esquema de tradução usa a tabela de símbolos

<i>program</i>		{ symTable = null; }
	<i>block</i>	
<i>block</i>	 {	{ saved = symTable; symTable = new SymTable(symTable); print('{'); }
	<i>decls</i>	
	<i>stmts</i> }	{ symTable = saved; print('}'); }
<i>decls</i>	 <i>decls decl</i>	
	ϵ	
<i>decl</i>	 <i>type id</i> ;	{ s = new Symbol; s.type = type.lexeme; symTable.put(id.lexeme, s); }
<i>stmts</i>	 <i>stmts stmt</i>	
	ϵ	
<i>stmt</i>	 <i>block</i>	
	<i>fact</i> ;	{ print(';') }
<i>fact</i>	 <i>id</i>	s = symTable.get(id.lexeme); print(id.lexeme); print(':'); print(s.type);

Resumo

- A tabela de símbolos é usada para **passar informações**
 - Declarações de identificadores são **armazenadas** na tabela
 - Usos de identificadores são **consultados** na tabela
- O **esquema de tradução** contém **ações semânticas**
 - Para guardar o estado atual do ambiente
 - Alterar o ambiente conforme o compilador navega pelos blocos
- O processamento dos blocos { } cria as tabelas
 - Elas são **conectadas em uma árvore**