

# Aula 03 – Análise Semântica - Tabelas de Símbolos

Prof. Eduardo Zambon

Departamento de Informática (DI)  
Centro Tecnológico (CT)  
Universidade Federal do Espírito Santo (Ufes)

**Compiladores**  
***Compiler Construction (CC)***

- O terceiro módulo do compilador realiza a **análise semântica**.
- Esta análise pode ser subdividida em várias etapas.
- **Estes slides**: discussão sobre as principais estruturas de dados utilizadas por um compilador.
- **Objetivos**: apresentar os aspectos práticos que envolvem essas estruturas.

## Referências

### **Section 6.3 – The Symbol Table**

*K. C. Louden*

### **Section 5.5 – Symbol Tables**

*K. D. Cooper*

### **Section 7.4 – The Symbol Table**

*D. Thain*

- Após a análise léxica (*scanning*) e a análise sintática (*parsing*) vem a fase de **análise semântica** (ou **elaboração semântica**).
- **Propósito**: computar informações **adicionais necessárias** para o processo de compilação que **não podem ser obtidas** pelos algoritmos de *parsing*.
- A análise semântica realizada por um compilador é dita **estática**, pois precede a execução do programa.
- **Possíveis** ações desta etapa:
  - Construção de uma **tabela de símbolos/strings**.
  - **Verificação de tipos** de expressões/declarações.
  - **Inferência de tipos**.
  - Teste de **limites de vetores**, etc, etc.
- Algumas dessas ações podem ser **realizadas por (ou em conjunto com) outros módulos** do compilador.

- Quantidade de ações requeridas na fase de análise semântica **varia bastante** dependendo da LP.
- Em linguagens com **tipagem dinâmica**, como LISP e Python, pode não haver **nenhuma (ou muito pouca)** análise semântica estática.
- Por outro lado, linguagens com **tipagem estática**, como Ada e C, impõem um **conjunto maior de restrições** sobre o programa de entrada.
- A análise semântica estática requer:
  - Uma **descrição** das análises a serem realizadas.
  - Uma **implementação** destas análises usando algoritmos apropriados.
- **Analogia** com *parsing*: gramática em BNF seria a descrição e o algoritmo de *parsing* seria a implementação.
- Infelizmente não há um método consolidado como BNF para **descrição da semântica** de LPs.

# Tabela de Símbolos

- **Tabela de símbolos**: uma das estruturas mais importantes em um compilador.
- Típica estrutura de dados de **dicionário**: associa **uma chave (o símbolo)** a um conjunto de **informações**.
- Geralmente usada para armazenar informações sobre **identificadores**: nomes de **variáveis** e de **funções**.
- A tabela de símbolos serve como um **repositório central** de informações coletadas ao longo da compilação.
- Informações típicas associadas a uma **variável**: linha de declaração, tipo de dado, escopo, etc.
- Informações típicas associadas a uma **função**: linha de declaração, tipo de retorno, aridade, etc.
- Implementação do compilador pode usar somente **uma ou várias** tabelas de símbolos especializadas (para variáveis, para funções, etc).

Possíveis **estruturas** para a tabela de símbolos:

- **Lista linear:**

- Provêm uma implementação **simples** e direta das operações básicas.
- Complexidade das operações é **linear**.
- Usada somente quando não há preocupação com eficiência: **protótipo ou compilador experimental**.

- **Árvore:**

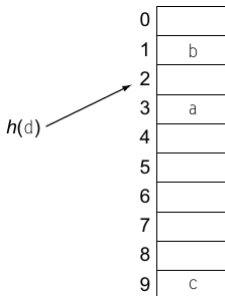
- Complexidade das operações é **sublinear**.
- **Pouco usada** na prática.

- **Hash table:**

- Complexidade das operações é **constante**.
- **Muito usada** na prática.

# Hash Table

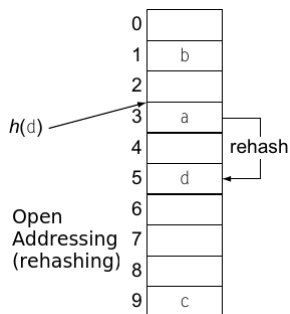
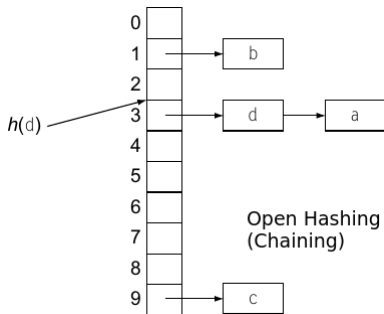
- Composta por um **vetor de entradas** (chamados *buckets*), indexado por **valores inteiros pequenos**.
- Uma **função hash  $h$**  mapeia a **chave** de busca (um nome) em um **valor hash** na faixa de índices.
- O item (informação) associado à chave  **$k$**  é armazenado no *bucket* de índice  **$h(k)$** .
- Exemplo de uma tabela de **tamanho 10**, com os nomes *a*, *b* e *c* já inseridos, e o nome  **$d$**  sendo inserido em  **$h(d) = 2$** .



0	
1	b
2	
3	a
4	
5	
6	
7	
8	
9	c

# Hash Table

- Se a função  $h$  mapeia **dois ou mais símbolos** para o mesmo  $hash$ , ocorre uma **colisão**.
- No exemplo anterior, aconteceria se  $h(d) = 3$ .
- Uma função  $h$  sem colisões é dita **função de hash perfeito**.
- Esse tipo de função pode ser obtida em algumas aplicações mas **não no cenário de compiladores**.
- Existem dois métodos principais para se **tratar colisão**:





- Compiladores geralmente implementam a tabela de símbolos como uma *hash table que usa chaining*.
- A *eficiência* da tabela é totalmente dependente do *espalhamento* provido pela *função  $h$* .
- Idealmente as chaves devem ser *distribuídas igualmente* ao longo de todos os *buckets*.
- Normalmente se utiliza um *número primo como tamanho da tabela* pois isso gera um melhor espalhamento.
- A *qualidade* de uma função *hash* pode ser testada de forma *experimental*, como mostrado a seguir.

# Hash Table

```
#define SZ 3571

uint32_t silly(char *buf, int len) {
    uint32_t h = 0;
    for (int i = 0; i < len; i++)
        h += buf[i];
    return h % SZ;
}

uint32_t adler(char *buf, int len) {// Mark Adler (1995) -> Zlib
    uint32_t s1 = 1;
    uint32_t s2 = 0;
    for (int i = 0; i < len; i++) {
        s1 = (s1 + buf[i]) % 65521; // Largest prime number < 2^16
        s2 = (s1 + s2) % 65521;
    }
    return ((s2 << 16) | s1) % SZ;
}
```

# Hash Table

```
uint32_t joaat(char *buf, int len) { // Jenkins-one-at-a-time
    uint32_t h = 0;                  // Bob Jenkins (1997)
    for (int i = 0; i < len; i++) {
        h += buf[i];
        h += (h << 10);
        h ^= (h >> 6);
    }
    h += (h << 3);
    h ^= (h >> 11);
    h += (h << 15);
    return h % SZ;
}

/*
Test with "/usr/share/dict/cracklib-small" (54.763 entries)
silly -- Min = 0, Max = 205, Collision rate = 0.293674
adler -- Min = 4, Max = 28, Collision rate = 0.007565
joaat -- Min = 4, Max = 31, Collision rate = 0.008700
*/
```

# Construindo a Tabela de Símbolos

A tabela de símbolos define **duas funções** principais.

- 1 **Insert** (*name*, *record*) : **armazena** a informação em *record* na posição ***h(name)*** da tabela.
  - 2 **Lookup** (*name*) : **retorna** a informação armazenada na posição ***h(name)***, **se ela existir**. Caso contrário, retorna um valor especial indicando **falha na busca**.
- Ao processar uma **declaração de variáveis**, o compilador computa um **conjunto de atributos** para cada variável.
  - **Ao reconhecer um identificador**, o insere na tabela de símbolos usando **Insert**.
  - Se um nome **só pode aparecer uma vez** na declaração, usa **Lookup** para detectar uma repetição.
  - Quando um nome aparece **fora de uma declaração**, usa **Lookup** para recuperar a informação da tabela.

# Escopos Aninhados

- Todas as LPs atuais permitem declarações de variáveis em **múltiplos níveis**.
- Isso quer dizer que **não há um namespace unificado**.
- **Bloco**: qualquer construto que pode **conter declarações** de nomes.
- LPs permitem **aninhamento** de blocos.
- Cada bloco define o **escopo (nível de aninhamento)** e o **tempo de vida** do nome.
- **Tempo de vida** é o período **durante a execução** na qual o **valor** da variável é **preservado**.
- Em uma LP com blocos aninhados, o *front-end* precisa de um mecanismo para **traduzir uma referência** para o escopo apropriado.
- Isso é ilustrado no exemplo a seguir.

# Escopos Aninhados

```
static int w;      /* level 0 */
int x;

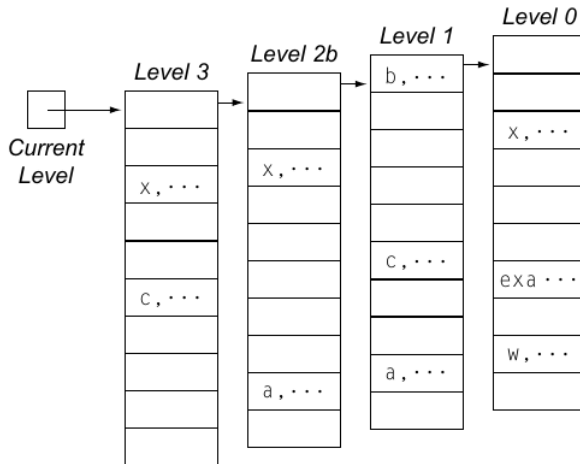
void example(int a, int b) {
    int c;         /* level 1 */
    {
        int b, z;   /* level 2a */
        ...
    }
    {
        int a, x;    /* level 2b */
        ...
        {
            int c, x; /* level 3 */
            b = a + b + c + w;
        }
    }
}
```

Level	Names
0	w, x, example
1	a, b, c
2a	b, z
2b	a, x
3	c, x

No programa C acima, níveis 2a e 2b são independentes, os demais são aninhados.

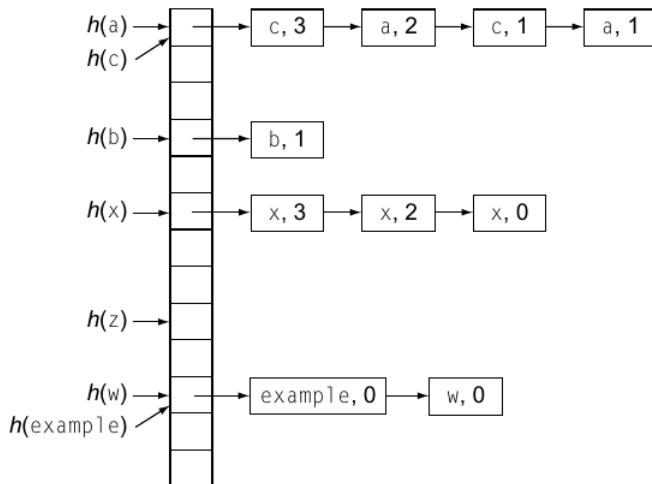
# Escopos Aninhados

Duas novas operações: `InitializeScope()` e `FinalizeScope()`, criam uma lista de tabelas.



# Escopos Aninhados

Alternativamente, pode-se incluir o escopo na **composição da chave** da tabela de símbolos.





# Tabela de Literais (*Strings*)

- O compilador deve evitar ao máximo operar com *strings*: muito ineficiente.
- Idealmente, o único componente do compilador que deve manipular *strings* diretamente é o *scanner*.
- Isso pode ser realizado utilizando-se uma **tabela de literais (*strings*)**.
- É uma estrutura similar à tabela de símbolos para armazenar as *strings* que aparecem no programa de entrada.
- Operações:
  - 1 **Put (*string*)**: **armazena sem repetições** a *string* passada na tabela e retorna um índice (ponteiro) para a entrada.
  - 2 **Get (*index*)**: **retorna** a *string* armazenada na posição *index*.

⇒ Ambas as tabelas serão usadas no Laboratório 03.

# Aula 03 – Análise Semântica - Tabelas de Símbolos

Prof. Eduardo Zambon

Departamento de Informática (DI)  
Centro Tecnológico (CT)  
Universidade Federal do Espírito Santo (Ufes)

**Compiladores**  
***Compiler Construction (CC)***