

Análise Semântica e Tratamento de Erros Dependentes de Contexto

O componente Semântico de uma LP

Tarefas da Análise Semântica

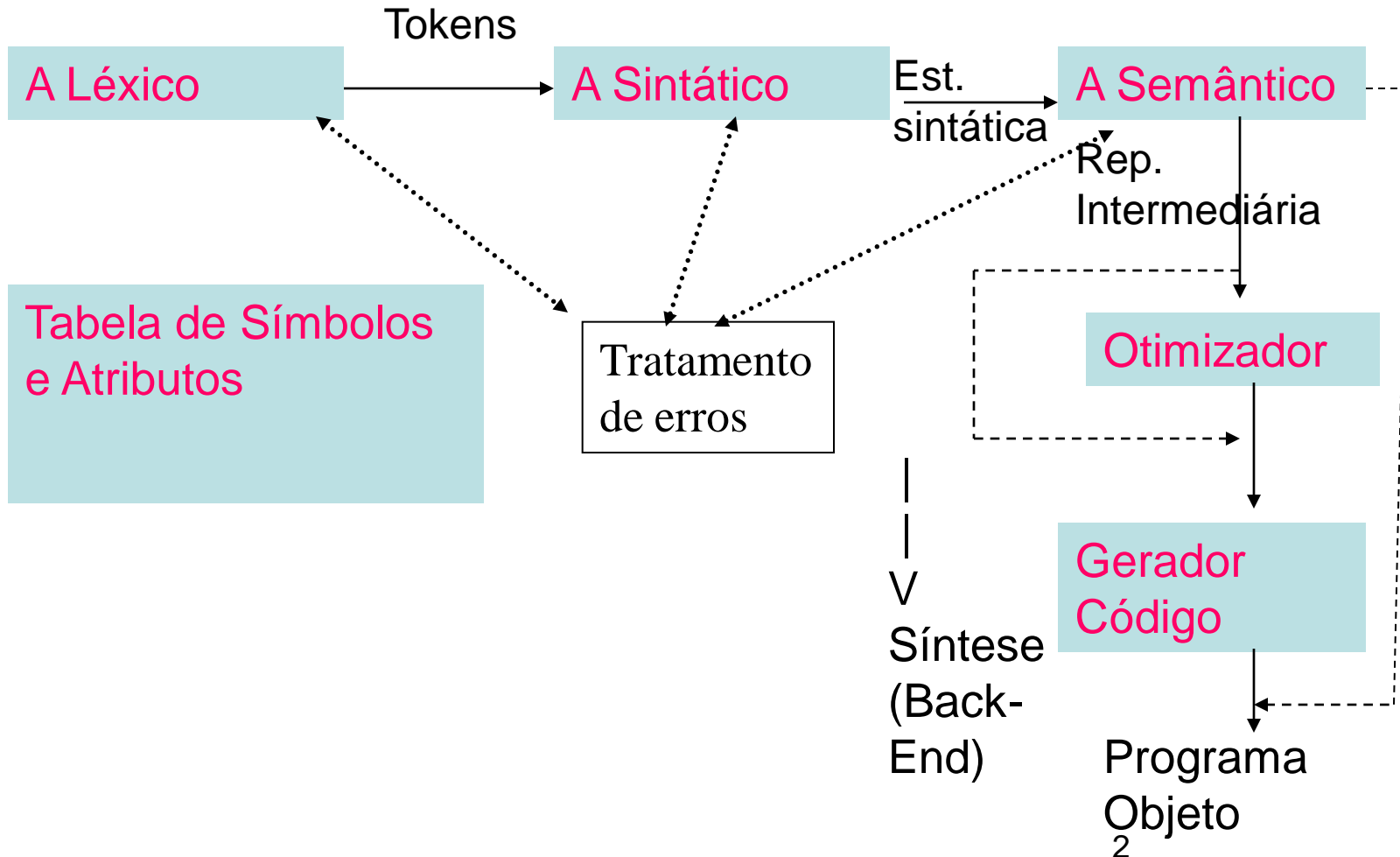
Implementação da Tabelas de Símbolos

Ações Semânticas em Compiladores
Dirigidos por Sintaxe e Erros da Análise
Semântica

Analizador Semântico

→ Análise (Front-End)

Programa
Fonte



O componente Semântico de uma LP

- A definição da sintaxe de uma LP utiliza, geralmente, uma GLC, mas nem todas as LP podem ser descritas por GLC's,
 - pois estas não checam a **compatibilidade de tipos** nem **regras de escopo de identificadores**, isto é, onde eles podem ser usados.
- Por exemplo.

$A := B + C;$

- É ilegal em Pascal se algumas das variáveis não foram declaradas ou se B ou C são booleanos.

O componente Semântico de uma LP

- Função: verificação do uso adequado
 - **Análise contextual**: declarações prévias de variáveis, procedimentos, etc.
 - Checagem de **tipos**
 - Coisas que vão além do domínio da sintaxe (GLC)
 - **Sensibilidade ao contexto!**

Componente Semântico

- Semântica Estática
- Semântica de Tempo de Execução

Categorias do Componente Semântico

- Semântica Estática

- conjunto de restrições que **determinam** se **programas** sintaticamente corretos **são válidos**.
 - Compreende checagem de tipos, análise de escopo de declarações, checagem de número e tipo de parâmetros de funções/procedimentos.
- A. Semântica aumenta a GLC e completa a definição do que são programas válidos.
 - Pode ser especificada informalmente (e geralmente é artesanal)
 - descrições em manuais de cada linguagem
 - ou formalmente
 - por exemplo, com uma **Gramática de Atributos**

Gramáticas de Atributos

- É uma gramática livre de contexto estendida para fornecer sensibilidade ao contexto através de atributos ligados a terminais e não terminais.
- Os atributos podem ser determinados durante a **compilação** ou **execução**
- Um atributo é qualquer propriedade de uma construção de LP.
Exemplos:
 - Tipo de dado de uma variável (**Compilação**)
 - Valor de uma expressão (**Tempo de Execução, a não ser que a expressão trate de constantes**)
 - Localização de uma variável na memória (**Depende da variável**)
 - Endereço do início do código objeto de um procedimento (**Compilação**)
 - Declaração de objeto no contexto (**Compilação, para linguagens que exigem declaração prévia como Pascal**)

- Semântica de Tempo de Execução

- Usada para especificar o que o programa faz, isto é, a relação do programa-fonte (objeto estático) com a sua execução dinâmica. **Importante para a geração de código.**
- É geralmente especificada informalmente nos manuais mas também existem modelos formais:
 - Vienna Definition Language (VDL)
 - Definições Axiomáticas
 - Modelos Denotacionais
 - Modelos Operacionais, como por exemplo, os Diagramas de Execução do livro do Kowaltowsky, cap. 7, pg 81.
 - **Gramática de Atributos**

- Uma especificação semântica precisa é motivada pela necessidade do compilador ser correto.
- Mas na maioria dos casos, temos definições informais que tornam certos pontos ambíguos e incompletos. Por ex:
 - `L : goto L;` é correto??? Alguns compiladores proíbem, outros não.
 - `If (I<>0) and (K div I > 10) then...`

Haverá divisão por zero se $I=0$ e se o compilador testar todas as expressões para se obter o resultado de um AND; mas isto não é necessário quando a primeira expressão é falsa. Porém, a definição do Pascal não diz nada sobre como tratar o AND.
- Muitas vezes é o compilador que serve de definição da linguagem quando ela não está totalmente especificada.

Gramática de atributos

- Gramática de atributos
 - Método usualmente utilizado
 - Conjunto de **atributos** e **regras semânticas** para uma gramática
 - Cada regra sintática tem uma regra semântica associada
 - Atributos associados aos símbolos gramaticais
 - Por exemplo, valor e escopo
 - `x.valor`
 - Regras semânticas que manipulam os atributos
 - Por exemplo, regra para somar os atributos valores de duas variáveis
 - `x:=a+b`, cuja regra é `x.valor:=a.valor+b.valor`

Gramática de atributos

- Atenção
 - Nem todo símbolo gramatical tem atributos
 - Podem haver manipulação de mais de um atributo em uma mesma regra e para um mesmo símbolo
- Em geral, a gramática de atributos de uma gramática específica:
 - o comportamento semântico das operações
 - a checagem de tipos
 - a manipulação de erros
 - a tradução do programa

Cômputo de atributos

- Com base na árvore sintática explícita
 - Grafos de dependência
 - Compilador de **mais de um passo**
- *Ad hoc*
 - Análise semântica “comandada” pela análise sintática
 - Compilador de **um único passo**

Cômputo de atributos

- Estruturas de dados externas
 - Em vez de se armazenar os atributos na árvore sintática ou de manipulá-los via parâmetros e valores de retornos, os atributos podem ser armazenados em estruturas separadas
 - Em compilação, a **tabela de símbolos** é utilizada,
 - junto com retorno de parâmetros/variáveis para checagem de tipos, que vai trabalhar com a gramática de <expressão>

Tabela de símbolos

- Estrutura principal da compilação
- Captura a sensibilidade ao contexto e as ações executadas no decorrer do programa
- Atrelada a todas as etapas da compilação
- Permite a realização da **análise semântica**
- Fundamental na **geração de código**

Como inserir as rotinas da Tabela de Símbolos no compilador?

- 2 formas:

Tabela de símbolos

- 1) As sub-rotinas de **inserção, busca e remoção** podem ser inseridas diretamente na **gramática de atributos**
 - Explicitamente, via chamadas de sub-rotinas de manipulação da tabela (inserção, busca e remoção de atributos)

Tabela de símbolos

- Exemplo: decl \rightarrow tipo var-lista
tipo \rightarrow int | float
var-lista \rightarrow id, var-lista | id

int x,y,z;

Regras gramaticais	Regras semânticas
decl \rightarrow tipo var-lista	var-lista.tipo_dado = tipo.tipo_dado
tipo \rightarrow int	tipo.tipo_dado = integer
tipo \rightarrow float	tipo.tipo_dado = real
var-lista ₁ \rightarrow id, var-lista ₂	id.tipo_dado = var-lista ₁ .tipo_dado var-lista ₂ .tipo_dado = var-lista ₁ .tipo_dado If busca(id)=FALSE then inserir(id,id.tipo_dado) else ERRO("identificador já declarado")
var-lista \rightarrow id	id.tipo_dado=var-lista.tipo_dado If busca(id)=FALSE then inserir(id,id.tipo_dado) else ERRO("identificador já declarado")

Tabela de símbolos

2) As sub-rotinas de **inserção, busca e remoção** podem ser inseridas diretamente na **análise sintática/parser**

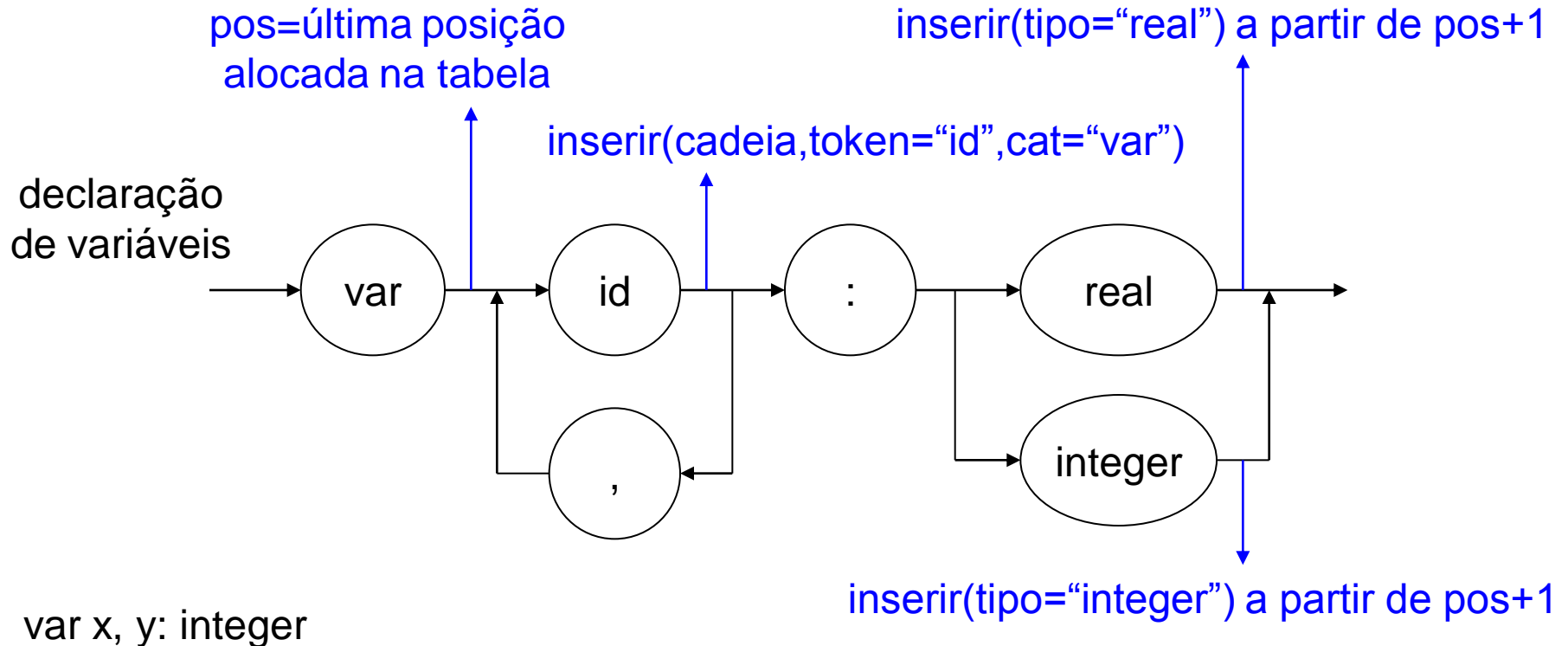
– Solução ad hoc

- Adequada para compilação de um único passo
- Usada em geradores de *parsers* como o YACC, JavaCC

Tabela de símbolos

- Inserção de elementos na tabela
 - Na declaração, principalmente
- Exemplo para Pascal
 - Vamos seguir a abordagem *ad hoc* no Trabalho 3

Tabela de símbolos



Cadeia	Nível/Escopo	Categoria	Tipo	Valor	...
...					
x	0	var	integer		...
y	0	var	integer		...

Tarefas da Análise Semântica

A tarefa do compilador a respeito da Análise Semântica é tripla:

1. Construir a descrição interna dos **tipos** e **estruturas de dados** definidos no programa do usuário;
2. Os **identificadores** (de constante, tipos, variáveis, procedimentos, parâmetros e funções) que são usados no programa têm que ser **guardados junto com seus atributos na TS**;
3. As informações obtidas deste modo tem que ser usadas para **checar o programa quanto a erros semânticos** (erros dependentes de contexto) e **checagens de tipos**.

1) Representação de Tipos e Estruturas de Dados

- As linguagens modernas oferecem um grande repertório de tipos e também permitem que o programador especifique seus próprios tipos de dados.
- Ao compilador cabe:
 - representar as especificações dos tipos
 - e usar tais informações para **a previsão do uso de memória** em tempo de execução, pelos objetos que forem declarados como sendo de um tipo especificado.

- Em tempo de compilação:
- Cria-se um **descriptor** (estrutura de dados que contém informações a respeito do objeto a que se referem);
- Preenche-se o descriptor com informações acerca da parte estática da estrutura/tipo;
- E se reservam áreas para o preenchimento posterior em tempo de execução das informações dinâmicas.
- No caso de vetores e matrizes, por exemplo:
 - muitos compiladores oferecem ao programador recursos automáticos de detecção do uso de índices fora dos limites especificados.
 - Neste caso, e nos casos de uso de tipos dinâmicos, as ações semânticas geram código (**testes**) no programa-objeto para a verificação de limites e de proteção contra endereçamentos incorretos.

Exemplo de Pascal

- Com 4 tipos simples e 4 possibilidades de estruturação, 8 possibilidades de dados são conseguidos:
 - primitivos/simples (integer, char, real e boolean),
 - enumerados
 - intervalo,
 - ponteiro,
 - set,
 - array e string,
 - record e
 - file.
- Os descritores têm campos comuns (**rótulo** e **nbytes**) e 8 partes variantes.

Possível implementação com record variantes

- Combinações de estruturas podem ser facilmente representadas por listas ligadas

```
Type tipos_possiveis = (escalares, enumerado, intervalo, ponteiro, tipo_set,  
vetor, tipo_record, tipo_file);
```

```
Descritor = Record
```

```
  Rotulo: string;
```

```
  Nbytes: integer;
```

```
  Case tipo: tipos_possiveis of
```

```
    Intervalo: (tipo_elem1: Pont_type; inf, sup: Valor);
```

```
    Enumerado: (Lista: Pont_type1);
```

```
    Ponteiro: (tipo_elem2: Pont_type);
```

```
    Tipo_set: (tipo_elem3: Pont_type);
```

```
    Vetor: (tipo_elem4: Pont_type; tipo_indice: Pont_type);
```

```
    Tipo_Record: (Prim_campo: Pont_type2);
```

```
    Tipo_file: (tipo_elem5: Pont_type)
```

```
  End;
```

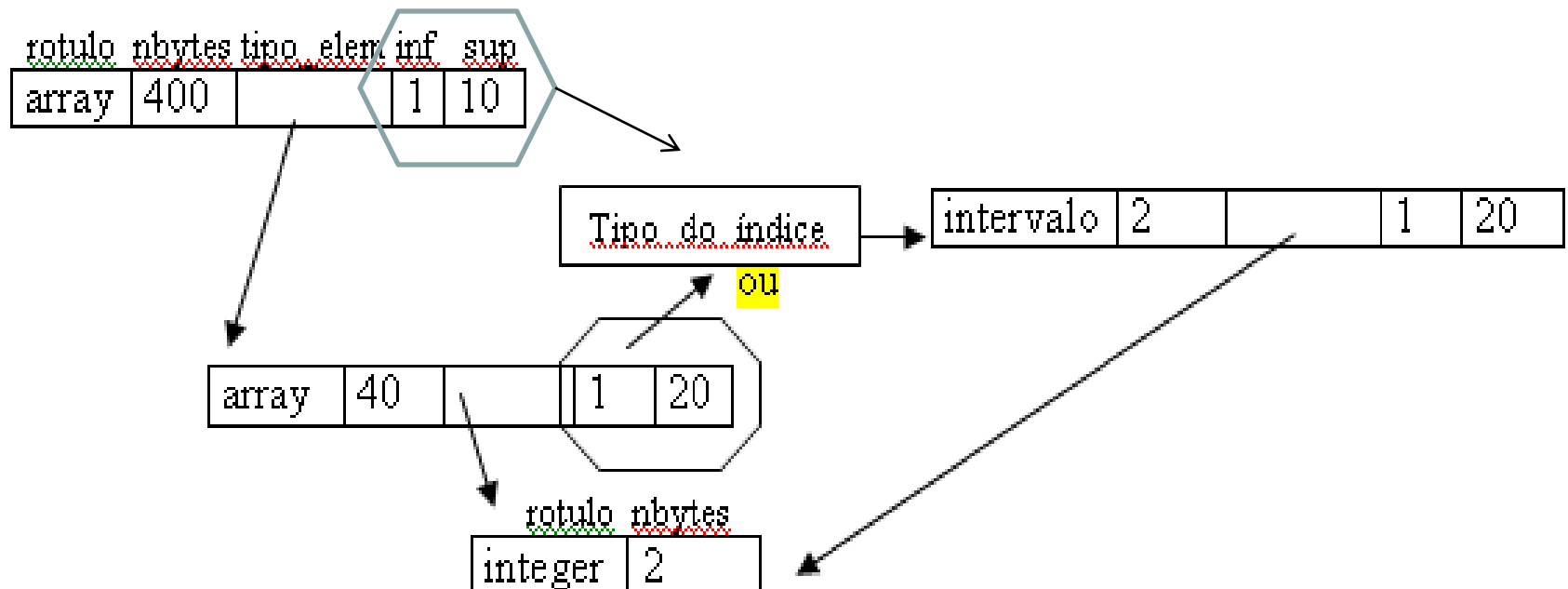
```
End;
```

Exemplo de Descritores para

- 1) array [1..10] of array [1..20] of integer;
- 2) file of array [1..10, 'A'..'Z'] of set of 0..31;
- 3) Record
 - a: integer;
 - b,c: real;
 - d: booleanEnd;
- 4) (azul, verde, vermelho);
- 5) string[10];

Exemplo de Descritores em Pascal para:

`array [1..10] of array [1..20] of integer;`



String e SET

- String é implementada como um array of char;
- Set como um array of boolean (bitarray).

- Os tipos escalares (real, integer, boolean e char) não possuem outros campos além do **número de bytes**.
 - **Devem ser inseridos inicialmente na Tabela de Símbolos** para que as definições do programa do usuário possam pegar a informação sobre o número de bytes destes tipos pré-definidos.
- Outros identificadores pré-definidos,
 - como os dos procedimentos de leitura e escrita (read/write) e
 - das funções de biblioteca (sin, cos, abs, ln, ...)
 - devem também ser inseridos na parte mais baixa (início) da Tabela de Símbolos.
- A operacionalização do fato de serem **pré-definidos** é esta inserção na parte inicial da Tabela de Símbolos.

2) Tabela de Símbolos

Uma Tabela de Símbolos reflete a estrutura do programa, pois guarda informações sobre o escopo de identificadores.

- À medida que o compilador processa o programa-fonte são encontrados identificadores que aparecem, por exemplo, em declarações de **variáveis** e de **procedimentos** com seus parâmetros.
- Para cada referência, o compilador terá necessidade de conhecer os atributos correspondentes (que sejam de interesse para a geração de código, verificação de erros semânticos e checagem de tipos).
- Por exemplo, no caso de uma **variável** quais **seriam?**

Atributos de um identificador de variável

- Sua categoria
 - Seu tipo
 - Seu endereço na área de dados (usa-se endereços que são posteriormente somados ao endereço de início da área de dados na memória)
-
- **NOTEM: todos os objetos são IDENTIFICADORES; é o atributo CATEGORIA que os diferencia**

- Estas informações são normalmente associadas com o identificador quando é processada a declaração a declaração correspondente e
 - deverão ser guardadas para serem usadas enquanto for processado o trecho do programa que é o escopo (nível) dessa declaração.
- A criação das entradas é geralmente feita
 - pelo **Analizador Léxico**, porém uma maneira melhor de modularizar as funções dos módulos de um compilador é deixar esta tarefa para o **Analizador Semântico**.

A maneira de implementar a Tabela de Símbolos depende:

- Da linguagem a ser compilada: em compiladores de linguagens estruturadas por blocos há uma vantagem de se utilizar a TI como:
 - uma “pilha” (a pesquisa começa do topo)
 - ou um display de níveis com ponteiros para árvores binárias
 - ou tabelas hash para cada escopo (nível).
- Do sistema no qual o compilador será implementado: tabelas de tamanho fixo vs comprimento variável (depende da possibilidade de se usar variáveis dinâmicas na implementação do compilador,
 - embora a grande massa das linguagens atuais tenha o tipo de dados ponteiro).
- Das características de eficiências desejadas: busca seqüencial vs busca em árvores binárias de busca vs buscas em tabelas hash.
 - Embora o melhor desempenho da busca seja T HASH, há problemas com a remoção dos ident de mesmo escopo

3) Listas de alguns erros dependentes de contexto

- Identificador já declarado no escopo (nível) atual
- Tipo não definido
- Limite inferior > limite superior na declaração de vetores/matrizes
- Função não declarada (quando há parênteses)
- Função, variável, parâmetro, ou constante não definidas (checagem no lado direito de atribuições)
- Incompatibilidade no número de parâmetros
- Procedimento não declarado (quando há parênteses)
- Função, variável, procedimento ou parâmetros não definidos (lado esquerdo de atribuições/lado esquerdo)
- Identificador de tipo esperado

Estruturas de Dados usadas para implementar a Tabela de Símbolos

- 1) Lista Linear Desordenada
- 2) Lista Linear Ordenada
- 3) Árvore de Busca Binária Global
- 4) Tabelas Hash
- 5) Tabelas Estruturadas por Bloco
Árvores de busca binária para cada nível

Implementação para Tabelas de Símbolos

Lista Linear Desordenada

- (implementação como **lista estática seqüencial**) - Tabelas de tamanho fixo (**vetor**):
 - id são inseridos no topo; (tempo constante)
 - Verificar e remover de tempo linear no tamanho da lista
 - desvantagem de serem muito pequenas para certos programas e grandes para outros.
 - Justificado somente por razões de simplificação didática.
- (implementação como **lista dinâmica**) - Lista Encadeada:
 - inserção também de tempo constante (insere no começo ou final) e verificar e remover de tempo linear no tamanho da lista
 - conserta a desvantagem citada acima

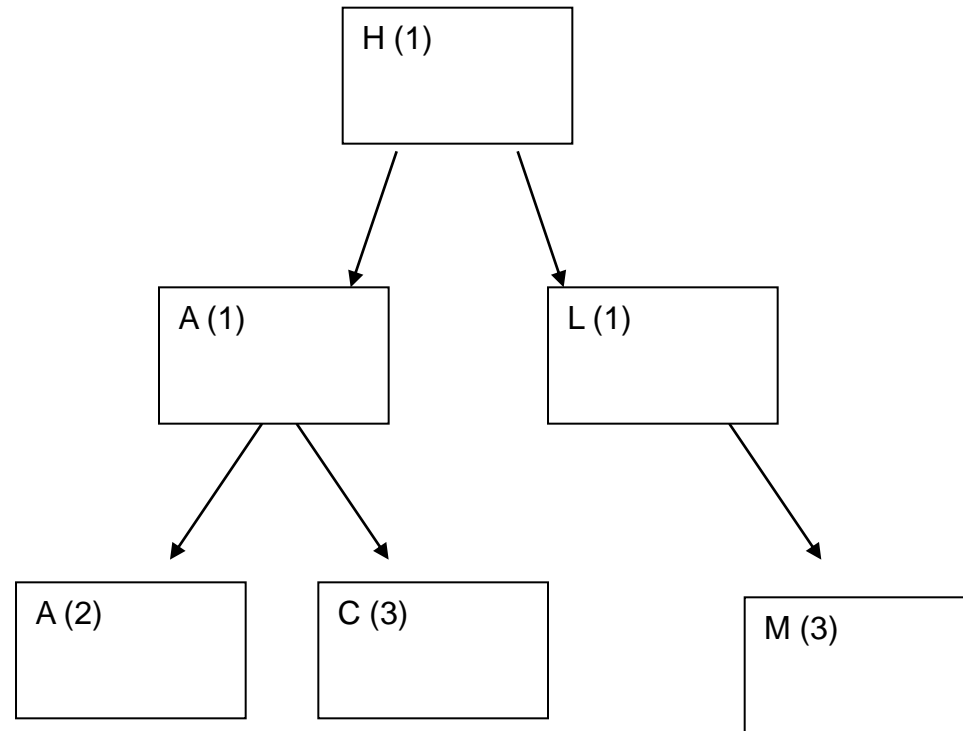
Lista Linear Ordenada

- Com inserção ordenada, um algoritmo de busca binária em um **vetor** requer tempo de $O(\log(n))$.
 - Entretanto, cada novo id deve ser inserido ordenadamente e esta operação é cara.
 - Listas ordenadas são úteis para tabelas de palavras reservadas e não para TS.

Árvore de Busca Binária Global

- Combinam a flexibilidade de tamanho e eficiência de inserção de uma estrutura de dados encadeada, com a rapidez da busca binária.
- Na média, buscar ou inserir id randômicos requer tempo $O(\log(n))$.
- Entretanto, o desempenho do caso médio não pode ser garantido desde que os id declarados num programa **não aparecem de forma randômica**.
- Vantagem: implementação simples, o overhead de espaço gasto com ponteiros é proporcional ao número de id inseridos.
- A operação remover é mais complexa.
- Em linguagens estruturadas por blocos podemos ter vários identificadores com mesmo nome se estiverem em escopos diferentes.
 - Isto causa problemas de **busca e desalocação de escopos nas** tabelas de busca binárias únicas/globais.
 - **Uma solução seria ter uma árvore para cada escopo**: variáveis globais, dentro de procedimentos de nível 1, de nível 2 (encaixados no de nível 1), como veremos.

Árvore de Busca Binária Global



- Eliminar escopos já usados é caro. Busca deve seguir até as folhas para pegar a última inserção

Tratamento de escopo: regra do aninhamento mais próximo

Sub-rotinas aninhadas (no caso do Pascal)
Variáveis globais e locais com mesmo nome

```
int i, j;
int f(int tamanho)
{
    char i, temp;
    // ...
}
void g()
{
    float j;
    // ...
}
void h()
{
    char * j;
    // ...
}
main()
{
    // ...
}
```


Escopos de arquivo (global) e de bloco em C: cada instância de i representa uma variável diferente

```
#include <stdio.h>
int i = 1;          /* i defined at file scope */

int main(int argc, char * argv[])
*----- {
1
1
1      printf("%d\n", i);          /* Prints 1 */
1
1  *--- {
1  2      int i = 2, j = 3;          /* i and j defined at
1  2                                  block scope */
1  2      printf("%d\n%d\n", i, j); /* Prints 2, 3 */
1  2
1  2 *-- {
1  2 3      int i = 0; /* i is redefined in a nested block */
1  2 3          /* previous definitions of i are hidden */
1  2 3      printf("%d\n%d\n", i, j); /* Prints 0, 3 */
1  2 *-- }
1  2
1  2      printf("%d\n", i);          /* Prints 2 */
1  2
1  *--- }
1
1      printf("%d\n", i);          /* Prints 1 */
1
1      return 0;
1
*----- }
```

Tabelas Hash Global

- São a forma mais comum de se implementar TS nos compiladores e outros sistemas relacionados.
- Com uma tabela bem grande, uma boa função hash, e um tratamento de colisão apropriado, o tempo de busca/inserção será constante.
- Para o tratamento de colisões:
 - a melhor saída é uma lista encadeada (open hashing), assim não limitamos o número de entradas que pode ser feito na tabela.
 - O tamanho médio das listas é m/n , com m = tamanho da tabela e n o número de entradas.
 - Vantagens:
 - 1) minimiza o overhead gasto com espaço para a tabela desde que cada entrada requer somente o espaço de um ponteiro;
 - 2) não falha como as outras técnicas de colisão que usam o espaço da tabela hash (array).

Dada uma função hash uniforme, se nós temos n nomes e uma tabela com tamanho m , a média e buscas são proporcionais a $n(n + e)/m$. Se m for grande, maior que n , este método é mais eficiente que a lista linear.

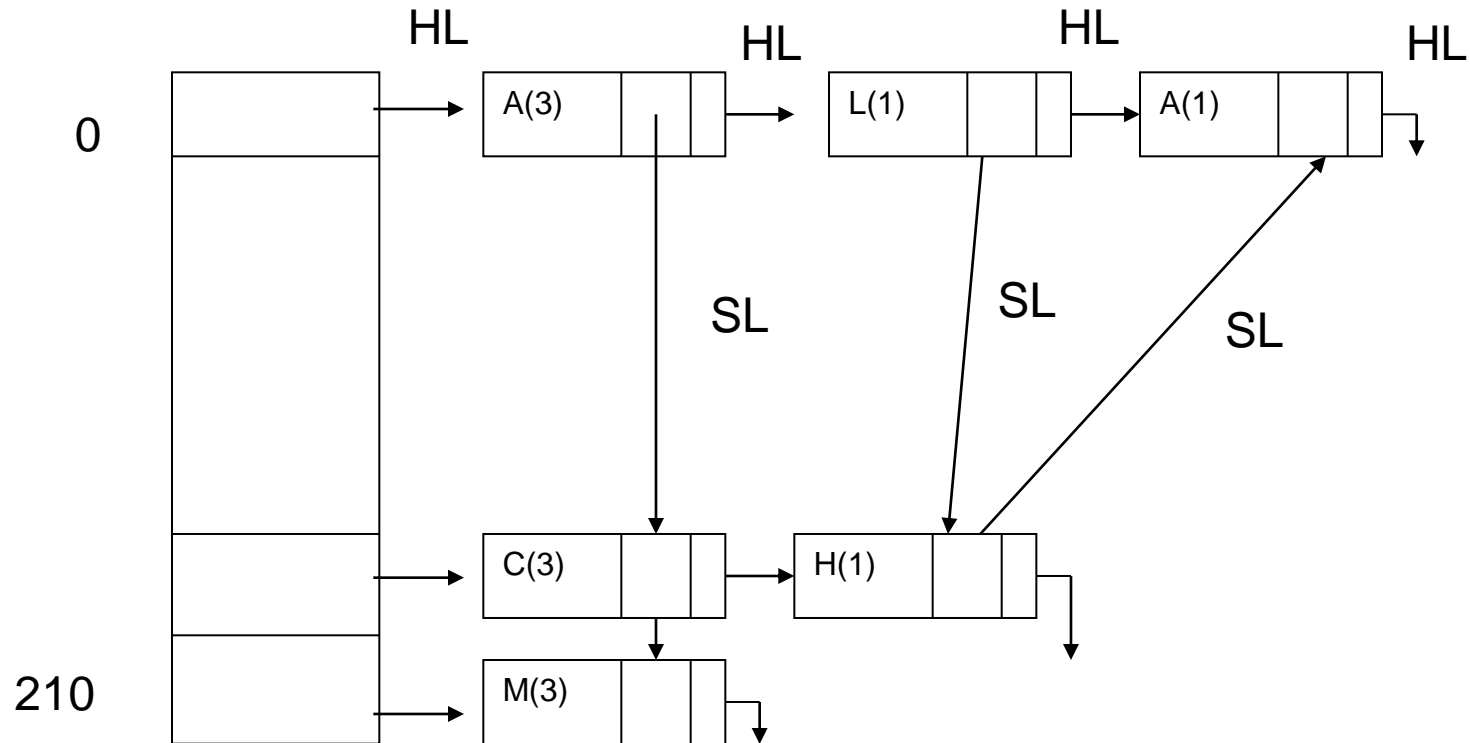
Busca:

- Para determinar se existe um id **s** na TS, aplicamos a função hash $h(s)$ que retorna um inteiro de $0..m-1$ e percorre-se a lista apontada pela entrada. Se id não está é criada uma entrada que é ligada na cabeça da lista indexada por $h(s)$.
- Deve-se tomar cuidado na hora de projetar a função hash de tal forma que ela seja fácil de se calcular e que distribua os nomes uniformemente entre m listas. Ver pág. 435 do livro (em inglês) do dragãozinho vermelho para exemplos de funções hash.

Remoção:

- Para a operação elimina, cada entrada deve ter 2 links: um “**hash link**” e outro “**scope link**” (que agrega todas as entradas de mesmo escopo para facilitar as remoções).

HL = Hash Link
SL = Scope link



Considerem implementar uma tabela **hash para cada escopo**, o que elimina a necessidade de se usar o scope link.

Tabelas Estruturadas por bloco (árvore de busca binária para cada nível)

- Usa um display de níveis sendo que em cada nível os identificadores são organizados em uma árvore de busca binária.
- Nesta estrutura, operações de inserção são eficientes.
- A eliminação de toda a árvore é feita de maneira trivial, pois a alocação é dinâmica.
- Quando os atributos dos nomes forem necessários para a verificação de erros semânticos, o display que aponta a **árvore de busca binária com os nomes locais** é analisada (mais alto nível), então em seguida a árvore que contém os nomes globais e finalmente a dos identificadores padrões.

...

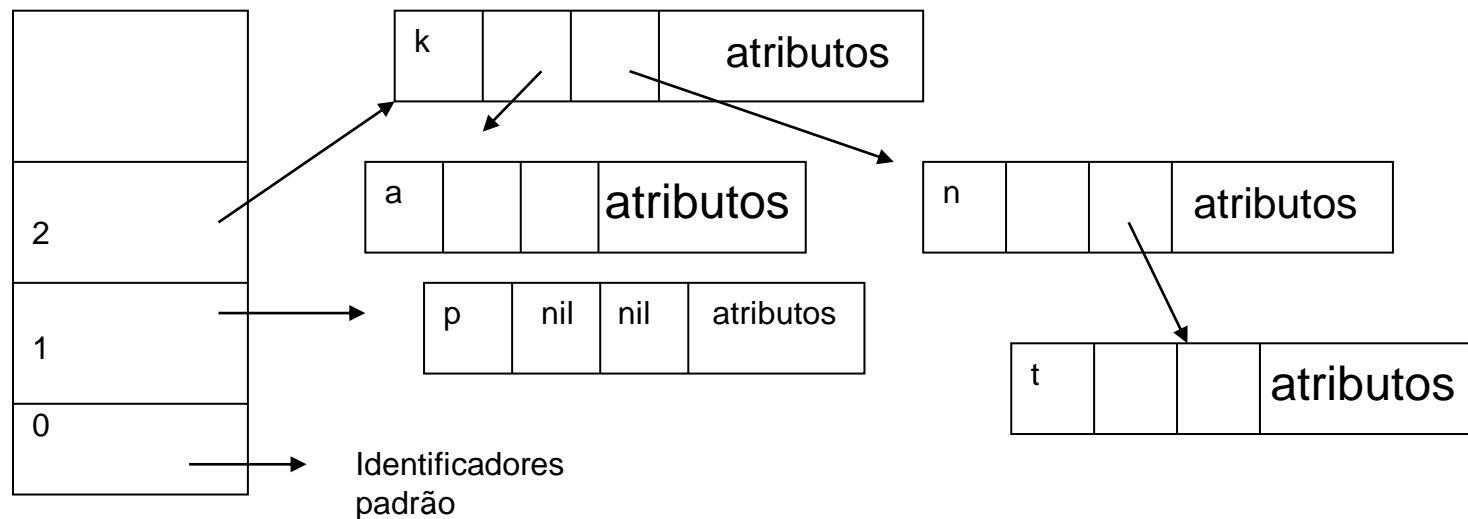
```
procedure p (K:integer);
```

```
  const n = 12;
```

```
  type t = set of 1..n;
```

```
  var a: t;
```

```
begin ...
```



Interface do TAD Tabela de Símbolos

Busca(Tab: TS; id: string; ref: Pont_entrada; declarado: boolean);

{busca id na Tab; retorna uma referência ref (ponteiro) para a entrada correspondente e um flag – declarado - para indicar se o nome já estava presente}

Elimina(Tab: TS; K:nível);

{elimina todos os id que estão num dado nível K (escopo) }

Insere(Tab: TS; id: string; ref: Pont_entrada; declarado: boolean);

{insere id na TS; retorna um ponteiro para a entrada e um flag para indicar se o nome já estava presente}

Declarado(Tab: TS; id: string; K: nivel): boolean;

{verifica se o id está declarado no nível K (corrente)}

Seta_atributos(Ref: Pont_entrada; AT: atributos);

Obtem_atributos(Ref: Pont_entrada; AT: atributos);

Fazer também uma rotina para **inserir todos os identificadores pré-definidos** (tipos escalares, read/write) na TS.

Estrutura das entradas da TS

ident, nível e categoria: campos para todas as categorias de identificadores. Os outros são dependentes de cada categoria.

Ident	Nível	Categoria
		Variável tipo_v end1
		Tipo nbytes tipo_elementar
		Constante tipo_c valor {valori,valorc, valorr, valors, valorb}
		Parâmetro tipo_p end2 classe_transf
		Procedimento npar1 end3
		Função npar2 end4 tipo_f

A implementação segue a mesma idéia dos descritores de tipos e estruturas de dados