INSTITUTO MAUÁ DE TECNOLOGIA
MAUÁ

Curso de Engenharia de Computação
ECM253 – Linguagens Formais, Autômatos e Compiladores

Conceitos de Análise Semântica – II

Slides da disciplina ECM253 – Linguagens Formais, Autômatos e Compiladores
Curso de Engenharia de Computação
Instituto Mauá de Tecnologia – Escola de Engenharia Mauá
Prof. Marco Antonio Furlan de Souza
<marco.furlan@maua.br>

Algoritmos para a computação de atributos

- **Computação dos atributos durante a análise sintática**
  - **Analisadores** do tipo **LL** (por exemplo, recursivo descendentes) são mais simples para calcular **atributos herdados**, pois propagam diretamente os valores durante a análise para os nós-filho;
  - **Analisadores** do tipo **LR** e **LALR** (por exemplo, Bison) são mais simples para calcular **atributos sintetizados**, pois propagam diretamente os valores durante a análise para os nós-pai;
  - No caso de analisadores LR, os valores dos atributos sintetizados podem ser calculados por ações semânticas associadas às reduções.

Algoritmos para a computação de atributos

- **Computação dos atributos durante a análise sintática**
  - No caso particular do JFlex (Java) e Bison (C/C++), atributos herdados podem ser calculados com o auxílio de **ações embutidas**, que permitem salvar valores de atributos para serem utilizados por atributos herdados;
  - **Exemplo**. Considerar novamente a gramática a seguir:

$$
\begin{aligned}
\mathit{decl} &\rightarrow \mathit{type}\ \mathit{var\text{-}list} \\
\mathit{type} &\rightarrow \texttt{int} \mid \texttt{float} \\
\mathit{var\text{-}list} &\rightarrow \texttt{id}, \mathit{var\text{-}list} \mid \texttt{id}
\end{aligned}
$$

Algoritmos para a computação de atributos

- **Computação dos atributos durante a análise sintática**
  - O arquivo (Bison) para esta gramática poderia ser assim definido:

```
decl   : type { current_type = $1; }
         var_list
       ;
type   : INT { $$ = INT_TYPE; }
       | FLOAT {   $$ = FLOAT_TYPE; }
       ;
var_list : var_list ',' ID
             { setType(tokenString,current_type);}
         | ID
             { setType(tokenString,current_type);}
         ;
```

Análise semântica

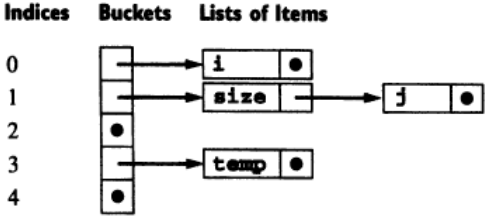
- **Tabela de símbolos**
  - É vista como um atributo **herdado** durante a compilação e armazena nomes e definições de variáveis, constantes, funções etc;
  - **Operações**
    - **Inserir**: armazenar informações relacionadas à declaração de um nome quando do processamento desse nome;
    - **Buscar**: recuperar informações de um nome quando ele for associado a algum código;
    - **Eliminar**: remover a informação de uma declaração quando ela não for mais necessária.
  - **Técnicas para tabelas de símbolos**
    - **Listas lineares**: simples, rápidas para inserir (se no início da lista), mais lentas para buscar($O(n)$) e eliminar;
    - **Árvores**: rápidas para inserir e buscar, mas complexas para eliminar;
    - **Tabelas de *hash***: técnica preferida em compiladores: as três operações podem ser executadas quase em tempo constante ($O(k)$).

Análise semântica

- **Tabelas de *hash***
  - É um vetor contendo entradas denominadas de *buckets*, indexada por um inteiro dentro do intervalo zero até o tamanho da tabela menos um;
  - Uma **função de *hash*** mapeia um chave que está sendo buscada para esse intervalo. A chave é normalmente uma cadeia de caracteres e o inteiro resultante representa a posição da chave na tabela;
  - O principal **problema** das funções de *hash* são as **colisões**: duas **chaves diferentes** são mapeadas no **mesmo inteiro**;
  - **Duas estratégias para resolver colisões**
    - **Endereçamento aberto**: quando ocorrer uma colisão, seguir no vetor (módulo de seu tamanho) até encontrar uma posição livre;
    - **Encadeamento separado**: nesse caso a tabela é um vetor de listas ligadas de elementos. Quando ocorrer uma colisão, cria-se e insere-se um novo nó na frente da lista associada àquela posição (preferido).

Análise semântica

- **Tabelas de *hash***
  - **Exemplo de encadeamento separado**



Análise semântica

- **Funções de *hash***
  - Toda função de *hash* deve executar três passos
    (1) Cada caractere da cadeia deve ser convertido em um inteiro;
    (2) Esses inteiros são combinado de alguma forma para gerar um novo inteiro;
    (3) O inteiro resultante deve ser adaptado ao intervalo $[0..size)$.
  - Funções práticas de *hash* possuem a forma:

$$h = (\alpha^{n-1}c_1 + \alpha^{n-2}c_2 + \ldots + \alpha c_{n-1} + c_n) \mod size = (\sum_{i=1}^{n} \alpha^{n-i}c_i) \mod size$$

Onde α é uma constante escolhida adequadamente (uma potência de 2 como 16 ou 128 simplificam o cálculo) e cada c_i representa um caractere da chave.

Análise semântica

- **Funções de *hash***
  - Exemplo em C

```c
#define SIZE ...
#define SHIFT 4

int hash(char *key){
    int temp = 0;
    int i = 0;
    while(key[i] != '\0'){
        temp = ((temp << SHIFT) + key[i])%SIZE;
        ++i;
    }
    return temp;
}
```

Análise semântica

- **Declarações**
  - O **comportamento da tabela de símbolos** depende dos tipos de **declarações** existentes na linguagem de programação sendo compilada;
  - Existem **quatro tipos básicos de declarações** frequentes nas linguagens de programação:
    - Declarações de constantes;
    - Declarações de tipos;
    - Declarações de variáveis;
    - Declarações de funções.

Análise semântica

- **Regras de escopo**
  - Escopo representa o local onde foi realizado uma declaração e afeta como esta declaração pode ser utilizada;
  - Existem **dois tipos** básicos de **regras de escopo**
    - **Declaração explícita ou antes do uso**: requer que um nome seja declarado antes de ser utilizado. Permite que a tabela de símbolos seja criada durante a compilação – promove a compilação em um passo;
    - **Declaração implícita ou por uso**: as declarações são anexadas às instruções sem uma prévia declaração. O aparecimento de uma nova variável representa sua declaração e uso simultâneos.

Análise semântica

- **Estrutura de bloco**
  - É uma propriedade comum encontrada nas linguagens de programação modernas;
  - **Bloco** é qualquer construção que contém declarações;
  - **Linguagens estruturadas em blocos** são aquelas que permitem aninhamentos de blocos;
  - **Regra do escopo mais próximo**: em um bloco aninhado, se houver uma declaração como o mesmo nome de outra declaração existente em um bloco mais externo, aquela do bloco mais interno será utilizada;
  - **Algumas linguagens** de programação **permitem** a **declaração** de **funções e procedimentos dentro de** blocos de **funções e procedimentos**;
  - Um **bloco** introduz um **novo escopo** no programa.

Análise semântica

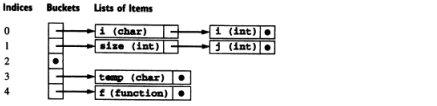
- **Tabela de símbolos e escopos aninhados**
  - Para implementar escopos aninhados, uma tabela de símbolos **não** deve **sobrescrever declarações** já existentes quando for **inserir** uma **nova declaração** de um **escopo aninhado**;
  - Do mesmo modo, não deve **eliminar** todas as declarações correspondentes a um determinado símbolo (podem haver símbolos iguais em escopos distintos);
  - Uma forma de **resolver esse problema** é **operar** nos **elementos da tabela** de símbolos **como** se fosse uma tabela de **pilhas** por símbolo: ao adentrar-se em um novo escopo, se houver um símbolo com o mesmo nome de um símbolo existente no escopo mais externo, esse será inserido na frente daquele mais externo;
  - Assim, **enquanto** se estiver **no escopo** mais interno, os **nomes acessados** serão aqueles que se encontram na **frente** das listas;
  - **Saindo do escopo**, basta **eliminar** esses **símbolos** que estão na **frente** das listas e retomar o escopo mais externo;
  - Uma **alternativa** é utilizar uma **nova tabela** de símbolos **para cada escopo** diferente.

Análise semântica

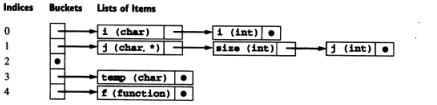
- **Tabela de símbolos e escopos aninhados**
  - **Exemplo – tabela de pilhas de símbolos**

```
int i,j;

int f(int size)
{ char i, temp;
  ...
  { double j;
    ...
  }
  ...
  { char * j;
    ...
  }
}
```



(a) After processing the declarations of the body of `f`

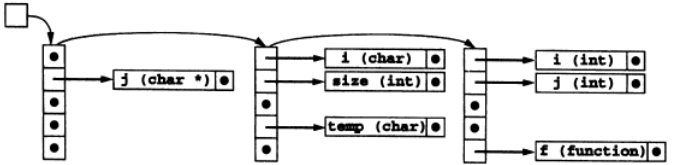


(b) After processing the declaration of the second nested compound statement within the body of `f`

(c) After exiting the body of `f` (and deleting its declarations)

Análise semântica

- **Tabela de símbolos e escopos aninhados**
  - **Exemplo – uma tabela para cada escopo**



Análise semântica

- **Escopo estático e escopo dinâmico**
  - **Escopo estático**: é o tipo de escopo discutido até aqui. As regras de escopo seguem basicamente o leiaute do programa;
  - **Escopo dinâmico**: as regras de escopo seguem o o caminho de execução;
  - O programa em C abaixo exibe o valor de `i` igual a `1`, pois C segue o princípio de escopo estático, mas se fosse com escopo dinâmico exibiria `2`:

```
#include <stdio.h>

int i = 1;

void f(void)
{ printf("%d\n",i);}

void main(void)
{ int i = 2;
  f();
  return 0;
}
```

Análise semântica

- **Interações em declarações de mesmo nível**
  - Representa a interação de declarações em um mesmo nível de aninhamento;
  - Ocorre quando uma nova declaração depende de uma outra declaração realizada imediatamente antes;
  - Podem ser processadas de **modo sequencial**, sendo imediatamente buscadas/inseridas na tabela de símbolos ou de modo **colateral**, onde as declarações são acumuladas em um tabela temporária e então adicionadas à tabela de símbolos;
  - Outra preocupação em declarações de mesmo nível é a **chamada recursiva de funções**– o nome da função já deve estar presente na tabela de símbolos para não ocorrerem erros;
  - A **dependência mútua** de **nomes** de funções também é outro problema – resolvido com protótipos em C e declarações **forward** em Pascal.

Análise semântica

- **Exemplo de gramática de atributos usando tabela de símbolos**
  - Considere a gramática a seguir:

$$
\begin{aligned}
&S \rightarrow exp \\
&exp \rightarrow \texttt{(}\ exp\ \texttt{)} \mid exp + exp \mid \texttt{id} \mid \texttt{num} \mid \texttt{let}\ dec\text{-}list\ \texttt{in}\ exp \\
&dec\text{-}list \rightarrow dec\text{-}list\ \texttt{,}\ decl \mid decl \\
&decl \rightarrow \texttt{id} = exp
\end{aligned}
$$

  - Ela introduz escopos no programa. Um exemplo de programa é:

```
let x = 2+1, y = 3+4 in x + y
```

  Outro exemplo, com escopo aninhado:

```
let x = 2, y = 3 in
  (let x = x+1, y=(let z=3 in x+y+z)
   in (x+y)
  )
```

Análise semântica

- **Exemplo** – gramática de atributos usando tabela de símbolos
  - Desenvolvimento da gramática de atributos

Grammar Rule	Semantic Rules
$S \rightarrow exp$	$exp.symtab = emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$
$exp_1 \rightarrow exp_2 + exp_3$	$exp_2.symtab = exp_1.symtab$ $exp_3.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$ **or** $exp_3.err$
$exp_1 \rightarrow$ (exp_2)	$exp_2.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$
$exp \rightarrow$ **id**	$exp.err =$ **not** $isin(exp.symtab,$**id**$.name)$
$exp \rightarrow$ **num**	$exp.err =$ **false**
$exp_1 \rightarrow$ **let** *dec-list* **in** exp_2	$dec\text{-}list.intab = exp_1.symtab$ $dec\text{-}list.nestlevel = exp_1.nestlevel + 1$ $exp_2.symtab = dec\text{-}list.outtab$ $exp_2.nestlevel = dec\text{-}list.nestlevel$ $exp_1.err = (decl\text{-}list.outtab =$ *errtab*$)$ **or** $exp_2.err$

Análise semântica

- **Exemplo** – de gramática de atributos usando tabela de símbolos
  - Desenvolvimento da gramática de atributos

$dec\text{-}list_1 \rightarrow dec\text{-}list_2$, $decl$	$dec\text{-}list_2.intab = dec\text{-}list_1.intab$ $dec\text{-}list_2.nestlevel = dec\text{-}list_1.nestlevel$ $decl.intab = dec\text{-}list_2.outtab$ $decl.nestlevel = dec\text{-}list_2.nestlevel$ $dec\text{-}list_1.outtab = decl.outtab$
$dec\text{-}list \rightarrow decl$	$decl.intab = dec\text{-}list.intab$ $decl.nestlevel = dec\text{-}list.nestlevel$ $dec\text{-}list.outtab = decl.outtab$
$decl \rightarrow$ **id** = exp	$exp.symtab = decl.intab$ $exp.nestlevel = decl.nestlevel$ $decl.outtab =$ **if** ($decl.intab = errtab$) **or** $exp.err$ **then** $errtab$ **else if** ($lookup(decl.intab,$**id**$.name) =$ $decl.nestlevel$) **then** $errtab$ **else** $insert(decl.intab,$**id**$.name, decl.nestlevel)$

Análise semântica

- **Expressões de tipo e construtores de tipo**
  - Linguagens de programação sempre possuem tipos predefinidos – **tipos simples**;
  - Algumas linguagens como Pascal e C permitem a definição de novos tipos simples;
  - **Novos tipos** de dados podem ser criados com **construtores de tipos**:
    - Arranjos (vetores e matrizes);
    - Estruturas;
  - **Ponteiros**: algumas linguagens suportam o conceito de **ponteiros** – é necessário definir expressões de referência e deferência de valores;
  - **Tipos função**: linguagens como Pascal e C permitem definir tipos que permitem criar variáveis que representarão funções;
  - **Classes**: similar às estruturas, introduz um novo escopo no programa onde além dos dados, pode-se declarar funções – métodos. Acrescentam **características** que vão **além** do **sistema de tipos** como **herança** e **vinculação dinâmica**.

Análise semântica

- **Equivalência de tipos**
  - Um sistema verificador de tipos realiza a tarefa de verificar se **duas expressões** representam o **mesmo tipo**;
  - Isto é denominado de **equivalência de tipos**;
  - Uma forma de implementar este tipo de verificação é escrever uma função que a partir de duas expressões de tipo retorne verdadeiro se ambas forem do mesmo tipo ou falso, caso contrário:

$$\textbf{function}\ typeEqual\ (\ t1, t2 : TypeExp\ )\ :\ Boolean;$$

  - Existem dois métodos básicos para testar a equivalência de tipos:
    - **Equivalência estrutural**: o algoritmo percorre recursivamente a estrutura das expressões envolvidas e dois tipos são iguais se e somente se possuem a mesma estrutura – árvores sintáticas de mesma estrutura;
    - **Equivalência de nomes**: duas expressões são do mesmo tipo se e somente se são do **mesmo tipo simples ou** se possuem o **mesmo nome de tipo**.

Análise semântica

- **Algoritmo para equivalência estrutural de tipos**

```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
  if t1 and t2 are of simple type then return t1 = t2
  else if t1.kind = array and t2.kind = array then
      return t1.size = t2.size and typeEqual ( t1.child1, t2.child1)
  else if t1.kind = record and t2.kind = record
       or t1.kind = union and t2.kind = union then
  begin
    p1 := t1.child1 ;
    p2 := t2.child1 ;
    temp := true ;
    while temp and p1 ≠ nil and p2 ≠ nil do
        if p1.name ≠ p2.name then
          temp := false
        else if not typeEqual ( p1.child1 , p2.child1 )
        then temp := false
        else begin
          p1 := p1.sibling ;
          p2 := p2.sibling ;
        end;
    return temp and p1 = nil and p2 = nil ;
  end
```

Análise semântica

- **Algoritmo para equivalência estrutural de tipos**

```
else if t1.kind = pointer and t2.kind = pointer then
   return typeEqual ( t1.child1 , t2.child1 )
else if t1.kind = proc and t2.kind = proc then
begin
   p1 := t1.child1 ;
   p2 := t2.child1 ;
   temp := true ;
   while temp and p1 ≠ nil and p2 ≠ nil do
      if not typeEqual ( p1.child1 , p2.child1 )
      then temp := false
      else begin
        p1 := p1.sibling ;
        p2 := p2.sibling ;
      end;
   return temp and p1 = nil and p2 = nil
          and typeEqual ( t1.child2 , t2.child2 )
end
   else return false ;
end ; (* typeEqual *)
```

Análise semântica

- **Algoritmo para equivalência por nomes de tipos**

```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
  if t1 and t2 are of simple type then
     return t1 = t2
  else if t1 and t2 are type names then
     return t1 = t2
  else return false ;
end;
```

Análise semântica

- **Exemplo de inferência e verificação de tipo**

Grammar Rule	Semantic Rules
var-decl → **id** : *type-exp*	*insert*(**id** .*name*, *type-exp*.*type*)
type-exp → **int**	*type-exp*.*type* := *integer*
type-exp → **bool**	*type-exp*.*type* := *boolean*
$type\text{-}exp_1$ → **array** **[num] of** $type\text{-}exp_2$	$type\text{-}exp_1$.*type* := *makeTypeNode*(*array*, **num** .*size*, $type\text{-}exp_2$.*type*)
stmt → **if** *exp* **then** *stmt*	**if not** *typeEqual*(*exp*.*type*, *boolean*) **then** *type-error*(*stmt*)
stmt → **id** := *exp*	**if not** *typeEqual*(*lookup*(**id** .*name*), *exp*.*type*) **then** *type-error*(*stmt*)
exp_1 → exp_2 + exp_3	**if not** (*typeEqual*(exp_2 .*type*, *integer*) **and** *typeEqual*(exp_3 .*type*, *integer*)) **then** *type-error*(exp_1) ; exp_1 .*type* := *integer*
exp_1 → exp_2 **or** exp_3	**if not** (*typeEqual*(exp_2 .*type*, *boolean*) **and** *typeEqual*(exp_3 .*type*, *boolean*)) **then** *type-error*(exp_1) ; exp_1 .*type* := *boolean*
exp_1 → exp_2 **[** exp_3 **]**	**if** *isArrayType*(exp_2 .*type*) **and** *typeEqual*(exp_3 .*type*, *integer*) **then** exp_1 .*type* := exp_2 .*type.child1* **else** *type-error*(exp_1)
exp → **num**	*exp*.*type* := *integer*
exp → **true**	*exp*.*type* := *boolean*
exp → **false**	*exp*.*type* := *boolean*
exp → **id**	*exp*.*type* := *lookup*(**id** .*name*)

Referências bibliográficas

AHO, A. V.; SETHI, R.; LAM, M. S. **Compiladores: princípios, técnicas e ferramentas**. 2. ed. [s.l.] Pearson, 2007.

COOPER, K.; TORCZON, L. **Construindo compiladores**. 2. ed. Rio de Janeiro: Elsevier, 2014.

LOUDEN, K. C. **Compiladores: princípios e práticas**. [s.l.] Pioneira Thomson Learning, 2004.