7 – Tabela de Símbolos e Verificação de Tipos

Linguagens e Programação

Ana Madureira

Engenharia Informática
Ano Letivo: 2021/2022

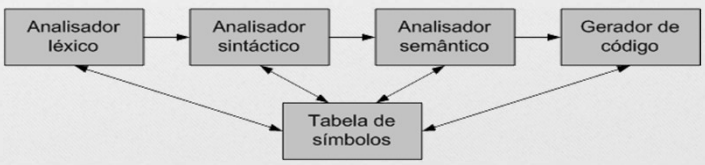
Fontes:

1. Compiladores Princípios e práticas, Keneth C.Louden, Thomson, 2004.
   Cap. 6 Análise Semântica
2. Compiladores Princípios,Técnicas e Ferramentas AlfredV.Aho,R.Sethi e Jeffrey D.Ullman,2007.
   Cap. 6 Verificação de Tipos

1

Tabela de Símbolos e Tipos: Função

- As tabelas de símbolos são extensamente utilizadas nos compiladores para armazenarem os nomes de variáveis, tipos, classes, e outras estruturas de programação
- Além do nome, cada entrada contém outras informações acerca do objecto nomeado, como seja o seu tipo, endereço ("offset" no segmento de dados estático, ou no registo de activação), número de referências, etc.
- Geralmente todos os módulos de um compilador necessitam de aceder à tabela de símbolos, para criar novas entradas, **adicionar informação** quando esta é descoberta (p. ex. o tipo de uma variável), ou simplesmente **consultar**
- Além disso, nas linguagens com estrutura de blocos, as tabelas de símbolos terão de reflectir o ***scope*** (âmbito) dos identificadores que contêm



2

Tabela de Símbolos e Tipos: Tipos de ocorrências

- **Ocorrências definidores de identificadores:**

  - às suas possíveis presenças em declarações

    - **Exemplo**:

      VAR

      A:REAL;

      I:ARRAY[1...10] OF INTEGER

- **Ocorrências aplicadas:**

  - às presenças em instruções

    - **Exemplo**:

      A:=10.17

      A:=A+ I[5]

3

Tabela de Símbolos e Tipos

- **Ocorrências definidores de identificadores e Ocorrências aplicadas:**

  - **A acção num compilador num caso e no outro é diferente**

    1. Inserção na tabela de símbolos no caso das **ocorrências definidores de identificadores**
    2. Pesquisa na tabela de símbolos (obtenção do tipo.....) no caso das **ocorrências aplicadas**

4

Tabelas de símbolos e tipos com um único nível

- Nas linguagens onde existem apenas variáveis globais não é necessário tomar nota do ***scope*** das mesmas (há apenas um nível de acesso às variáveis)
- Pode ser o próprio analisador léxico a criar as entradas na tabela de símbolos
- A tabela de símbolos é então uma estrutura de dados do tipo dicionário simples, devendo suportar operações como:
  - **add_symbol(x)** - cria uma nova entrada na tabela com nome x
  - **lookup_symbol(x)** - retorna a posição (apontador ou índice) na tabela da entrada contendo o nome ou a indicação de que não existe
  - **delete_symbol(x)** - retira o nome x da tabela.
- Estas tabelas podem ser implementadas como tabelas de *hashing*, árvores de pesquisa, listas ligadas, *arr* *stacks*, etc, com eficiência variada
- Nas linguagens com vários espaços de nomes (para variáveis, nomes de procedimentos, nomes de tipos, nomes de campos de estruturas, etc), poderá ser necessário dispor de tabelas separadas para cada espaç de nomes

5

Tabelas Multinível

- Nas linguagens estruturadas por blocos é necessário tomar nota de alguma forma do *scope* dos nomes
- Geralmente a própria estrutura da tabela reflecte o embutimento dos blocos
- As operações para este tipo de tabela incluem:
  - **create_scope(parent_scope)** - cria um novo nível, embutido no "parent_scope", retornando um seu apontador
  - **insert(scope, x)** - coloca o nome x no nível "scope" (ocorrências definidoras)
  - **lookup(x)** - procura o nome x, começando no scope mais próximo, retornando a sua posição na tabela (ocorrências aplicadas)
  - **lookup_last(x)** - procura apenas no nível mais recente
  - **delete_scope(scope)** - retira da tabela todo o nível correspondente ao scope "scope"

6

Verificação de tipos (análise semântica)

7

Verificação de tipos

- O compilador deve verificar:
  - Tipos de dados dos operandos, fornecidos a um operador, e a respectiva definição do operador são legais e compatíveis;
  - isto é deve, verificar se estão de acordo com o sistema de tipos definido na linguagem

- O **sistema de tipos** define para cada operador da linguagem e/ou construção gramatical, os tipos dos operandos e os tipos dos resultados possíveis

- As linguagens, para as quais todos os erros relacionados com o sistema de tipos podem ser detectados e identificados durante a compilação, dizem-se **estaticamente tipadas** (ou fortemente tipadas)

- As linguagens, para as quais pode ser necessário, durante a execução dos programas, fazer verificações de tipos, a fim de detectar erros, dizem-se **dinamicamente tipadas** (Prolog, Lisp, Scheme, p.ex.)

8

Sistemas de tipos

- Um sistema de tipos é uma colecção de regras para associar *expressões de tipos* às várias partes de um programa
- Os sistemas de tipos podem ser especificados através de definições dirigidas pela sintaxe, ou seja com atributos e ações semânticas
- As regras definidas num sistema de tipos, são regras que podem ser lidas em linguagem corrente como:
  - "Se ambos os operandos dos operadores aritméticos de adição, subtracção e multiplicação forem do tipo inteiro, então o resultado dessa operação é do tipo inteiro"
  - "O resultado do operador unário '&' é o endereço do seu operando; se o tipo do operando for " …", o tipo do resultado é "apontador para …" "
- A expressão "apontador para …" é uma *expressão de tipos* (operação sobre tipos)

9

Conversão entre tipos

- Muitas linguagens de programação definem regras para a conversão automática de um tipo de um operando, de modo a torná-lo compatível com o seu contexto

  Estas conversões automáticas são efectuadas pelo compilador, dizendo-se que teve lugar uma *coerção de tipos*
  - Geralmente numa expressão como "x + i", em que x é do tipo 'real' e i do tipo 'inteiro', a variável i sofre uma coerção para 'real' antes de se efectuar a adição

10

Conversão entre tipos

- Muitas linguagens de programação definem casos nos quais o programador pode explicitamente converter o tipo de dados de uma expressão num outro, utilizando operadores especiais de conversão (operadores de *cast*). Estes operadores são estaticamente tipados, uma vez que possuem tipos de entrada e saída bem definidos susceptíveis de serem verificados pelo compilador
  - Exemplo em C: p = (table *) calloc(n, sizeof(table));

O operador (table *) é um *cast*.

11

Expressões de tipos

- Uma expressão de tipos é qualquer expressão cujo resultado é um tipo: pode ser um tipo básico, ou o resultado da aplicação de um operador designado por *construtor de tipo*, a qualquer outra expressão de tipos. Os tipos básicos e construtores dependem da linguagem.
- Tipos básicos: são normais os seguintes - boolean, char, integer, real, void; um tipo básico especial é o tipo 'type_error', utilizado para marcar situações de erro durante a verificação de tipos
- Certas linguagens permitem associar um nome (identificador) a tipos definidos pelo utilizador; para essas linguagens os nomes que designam tipos são também expressões de tipos
- Um construtor de tipo aplicado a uma expressão de tipos é uma expressão de tipos

12

Construtores de tipos

- *Arrays* - Seja T uma expressão de tipos; então *array*(I, T) é uma expressão que representa o tipo de um array com elementos do tipo T e conjunto de índices I; geralmente I é um subconjunto (*subrange*) dos inteiros
- Subrange – Seja T um tipo simples ordenado; então subrange(min, max, T) é uma expressão que representa o conjunto dos valores t: {t ∈ T e min ≤ t ≤ max}
- Registos - Este construtor associa nomes e tipos dos campos de um registo ou estrutura
  - Exemplo: record((address × integer) × (lexeme × array(1..15, char))) define um tipo registo com 2 campos (address e lexeme) do tipo integer e array de char (com posições de 1 a 15)

13

Construtores de tipos

- Apontadores - A expressão (de tipos) apontador(T), em que T também é uma expressão, designa um tipo que é "um apontador para um objecto de tipo T"
- Funções - Associa-se também um tipo às funções. O respectivo construtor é normalmente notado por →. O tipo de função escreve-se D → C, em que D é o tipo do domínio da função e C o do contradomínio (ou resultado); quando os parâmetros de entrada são mais do que um, o tipo do domínio é representado por um produto cartesiano
  - Exemplo: A seguinte função em Pascal
    function f (a, b : char) : ↑integer;
    tem tipo: char × char → pointer(integer)
- Variáveis - Em certas situações pode haver necessidade de ter, em expressões de tipos, variáveis que representam tipos

14

Implementação da verificação de tipos

- A verificação de tipos faz-se normalmente através de regras semânticas, que devem implementar o sistema de tipos da linguagem, e de um atributo (*type*) associado aos não-terminais da gramática que contém o resultado de uma expressão de tipos
- Em muitos casos esse atributo pode ser sintetizado, o que torna muito fácil executar a verificação de tipos juntamente com a análise sintáctica
- Em muitas linguagens os identificadores devem ser declarados antes da sua utilização e nessas declarações indicam-se os respectivos tipos. Noutros casos os tipos são inferidos da forma como o identificador é escrito (Fortran, Basic)

15

Verificação de tipos: Exemplo

A verificação de tipos para uma linguagem simples com a seguinte gramática:

```
P → D ; S
D → D ; D |
    id : T
T → char |
    integer |
    boolean |
    array [ num ] of T |
    ↑ T
S → id = E |
    if E then S |
    while E do S |
    S ; S
E → char_const |
    num |
    bool_const |
    id |
    E mod E |
    E [ E ] |
    E < E |
    E and E |
    E ↑
```

Esta linguagem produz programas compostos por declarações (D), seguidas de instruções (S)

As declarações associam tipos (T) a identificadores; possui 3 tipos básicos (char, integer e boolean), além de apontadores e arrays. Os arrays são declarados com um único valor de índice, assumindo-se que este vai de 1 até esse valor; assim a declaração:

```
array[256] of char
```

corresponde a um array de 256 caracteres com índices que vão de 1 a 256.

No sistema de tipos a implementar usam-se ainda os nomes simples type_error, usado para assinalar erros e o valor void, associado às instruções sem erro.

Como na 1ª produção o D aparece antes de S, todas as declarações deverão estar feitas antes das instruções.

Um programa gerado por esta gramática poderá ser:

```
key : integer; result : boolean;
result = key < 1999
```

16

Acções semânticas para as declarações

- Durante as declarações vão-se tomando nota dos tipos dos identificadores que nelas aparecem.
- Associado a todos os compiladores existe sempre uma tabela de símbolos, onde é colocado o nome de todos os identificadores definidos pelo programador.
- Poderá caber ao analisador léxico a tarefa de ir preenchendo a tabela de símbolos com os identificadores que vai descobrindo, retornando como atributo um apontador ou índice do local onde o identificador ficou armazenado na tabela.
- Sempre que se descobre nova informação sobre um identificador, essa informação deve ser adicionada à respectiva entrada na tabela de símbolos(ocorrência definidora de identificadores).

Vamos associar a cada não-terminal um atributo type para tomar nota do respectivo tipo.

Como se disse os identificadores vêm do analisador léxico com um atributo (entry) que permite o acesso à sua entrada na tabela de símbolos.

A função addtype( ), usada nas acções semânticas, preenche um campo de uma entrada da tabela de símbolos onde deverá estar o respectivo tipo.

O terminal 'num' vem do analisador léxico com o atributo 'val', que contém o seu valor.

D → id : T	{ addtype(id.entry, T.type) }
T → char	{ T.type = char }
T → integer	{ T.type = integer }
T → boolean	{ T.type = bool }
T → array [num] of T1	{ T.type = array(1..num.val, T1.type) }
T → ↑ T1	{ T.type = pointer(T1.type) }

17

Verificação dos tipos de expressões

- Durante as declarações apenas identificámos os tipos das variáveis. Podiam--se aí já identificar alguns erros, como as declarações múltiplas. Por exemplo, se na tabela de símbolos já existir um tipo associado ao identificador, a função **addtype**( ) poderá identificar esse erro. A tabela de símbolos também guarda informação acerca do âmbito dos identificadores.
- É nas expressões que mais se faz sentir a necessidade da verificação de tipos.
- No esquema aqui apresentado faz-se uso de uma função **lookup**( ) que retorna o tipo associado a um identificador, presente na tabela de símbolos.
- Quando o atributo **type** ganha o valor **type_error**, poder-se-á aí emitir uma mensagem de erro.
- Os terminais (***tokens***) poderão trazer do analisador léxico informação acerca da sua posição no texto fonte, que poderá ser aproveitada para a emissão das mensagens de erro.

E → char_const	**{ E.type = char }**
E → num	**{ E.type = integer }**
E → bool_const	**{ E.type = bool }**
E → id	**{ E.type = lookup(id.entry) }**
E → E1 mod E2	**{ E.type = (E1.type == integer && E2.type == integer) ? integer : type_error }**
E → E1 [E2]	**{ E.type = (E2.type == integer && E1.type == array(s, t)) ? t : type_error }**
E → E1 < E2	**{ E.type = (E1.type == char && E2.type == char) ? bool : ((E1.type == integer && E2.type == integer) ? bool : type_error) }**
E → E1 and E2	**{ E.type = (E1.type == bool && E2.type == bool) ? bool : type_error }**
E → E1 ↑	**{ E.type = (E1.type == pointer(t)) ? t : type_error }**

18

Verificação das instruções

- Neste exemplo as instruções deverão ter o tipo **void** (não existente na linguagem) ou **type_error** se os tipos dos seus constituintes não estiverem correctos.
- A 1ª instrução apenas permite do lado esquerdo de uma atribuição um identificador; no entanto se a sua regra gramatical fosse S → E = E, permitindo do lado esquerdo expressões como a[10] ou p↑, teríamos de verificar, além da compatibilidade de tipos, a existência de um **l-value** para essa expressão.
- Por exemplo, poderíamos ter um outro atributo (has_**l-value**) que seria construído durante a análise sintáctica das expressões. **L-value** ou **left-value** representa directamente uma posição de memória. Algumas expressões que possuem **l-value** são: id, id[E], id↑ (podem ser usadas no lado esquerdo de uma atribuição).
- **R-value** ou **right-value** representa o valor de uma expressão (o conteúdo de uma posição de memória ou o resultado de uma operação); pode ser usado do lado direito de uma atribuição.

```
S → id = E              { S.type = (E.type == lookup(id.entry)) ? void : type_error }
S → if E then S1        { S.type = (E.type == bool) ? S1.type : type_error }
S → while E do S1       { S.type = (E.type == bool) ? S1.type : type_error }
S → S1 ; S2             { S.type = (S1.type == void && S2.type == void) ? void :
                  type_error }
```

19

Verificação de funções

- Embora a gramática que nos serviu de exemplo até ao momento não inclua funções, vamos acrescentá-las agora, considerando somente funções de um parâmetro.
- A sua declaração (associação de um nome ao tipo de função) pode ser feita acrescentando a seguinte regra para tipo:
  - **T → T1 '→' T2**
- A seta que aparece entre plicas é um terminal da linguagem; uma declaração de uma função, com aquela regra poderia ser:
  - **func : char → integer**
- que nos diz que **func** é uma função de argumento de tipo **char** com resultado de tipo inteiro.
- Para permitir a chamada de funções teríamos agora de acrescentar mais uma regra às expressões:
  - **E → E1 (E2)**
- As regras semânticas de verificação de tipos poderiam ser:

```
T → T1 '→' T2          { T.type = T1.type → T2.type }
E → E1 (E2)            { E.type = (E2.type == s && E1.type == s → t) ? t : type_error }
```

20

Overloading de operadores

- Diz-se que um operador está ***overloaded*** se tiver significados diferentes, dependentes do contexto
- Por exemplo, quando se usa um operador aritmético, tal como '+', normalmente este pode ser aplicado a inteiros, reais ou ainda a outros tipos, sendo assim necessário verificar em que contexto é aplicado para que se possa reconhecer o seu significado e, possivelmente, proceder à conversão de tipo de algum dos operandos :

A + B	com A e B inteiros com A e B reais com A e B complexos

- Um operador *overloaded*, em cada situação, só poderá ter um único significado; não são permitidas ambiguidades
- A resolução de uma situação de *overloading* designa-se por identificação de operadores

21

Resolução de situações de *overloading*

- Uma situação ambígua de *overloading* fica *resolvida* quando é possível determinar um único significado concreto da ocorrência de um símbolo ***overloaded***
- Muitas vezes a situação de *overloading* resolve-se apenas pelo exame dos tipos dos operandos
- Exemplo:

E → E1 op E2
O tipo de E1 e E2 pode ser suficiente para determinar o significado do operador op.

- Nestes casos os esquemas de verificação de tipos já abordados anteriormente servem para a resolução da situação de *overloading* presente

22

Verificação de tipos com *overloading*

- Outras vezes não é suficiente examinar apenas os tipos dos operandos. O tipo de uma expressão, em vez de poder ter um único tipo, poderá ter vários possíveis. Na maior parte das linguagens, o contexto deve providenciar informação suficiente para restringir a um único tipo.
- **Exemplo**:
  Suponhamos que o operador * pode ter os seguintes três significados:
  1) integer × integer → integer
  2) integer × integer → complex
  3) complex × complex → complex

A:= 2 * (3 * 5) (3 * 5) * z onde z é complexo e 2, 3 e 5 são inteiros

(1),(2) (1) (2) (3)

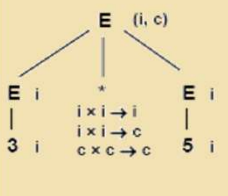
- O tipo do operador * na expressão com dois operandos inteiros não pode ser determinado conhecendo apenas os operandos

23

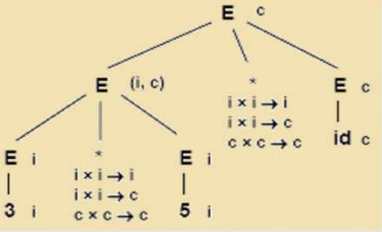
Exemplo (cont.)

- Após as declarações anteriores, a expressão **3 * 5** tanto pode ser do tipo *integer* como do tipo *complex*, dependendo do seu contexto. Se a expressão completa é **2 * (3 * 5)**, então **3 * 5** será do tipo *integer*. Mas se a expressão for **(3 * 5) * z** e se **z** for declarada como *complex*, então **3 * 5** será do tipo *complex*
- Solução possível: propagação dos tipos na árvore de parse, usando a notação pós-fixa e uma ordem de visita dos nós primeiro em profundidade

Expressão 3 * 5



Expressão (3 * 5) * z



24

Polimorfismo

- Funções polimórficas – funções em que as instruções do seu corpo podem ser executadas com argumentos de diferentes tipos
- Funções polimórficas – facilitam a implementação de algoritmos que manipulem estruturas de dados, independentemente do seu tipo
- A implementação de um módulo de verificação de tipos pode ser dificultado no caso de linguagens com funções polimórficas
- Exemplo:

```
int max(int, int) - extrai o máximo entre dois inteiros
int max(const int *, int) - extrai o máximo de um array de inteiros
```

25

Polimorfismo

- Consideremos o seguinte pseudo-código:

```
function deref(p);
begin
    return p↑
end;
```

A função deref tem o mesmo efeito que os operadores ↑ do Pascal e * do C.
p (do tipo β) é um apontador para um objecto de tipo desconhecido α.
β = pointer(α)

$$\forall \alpha, pointer(\alpha) \xrightarrow[deref]{} \alpha$$

- Vamos usar uma linguagem para verificar funções polimórficas, gerada a partir da seguinte gramática:

P → D; E

D → D; D | id : Q

Q → ∀ type_variable, Q | T

T → T '→' T | basic_type | type_variable | (T)

E → E(E) | E, E | id

26

Polimorfismo

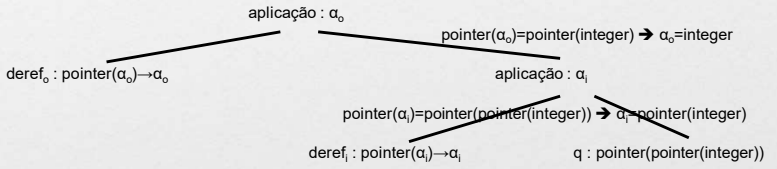
Lb1 : Lb2
Lb1 – subexpressão
Lb2 – tipo da subexpressão

- Os programas gerados por esta gramática são constituídos por uma sequência de declarações, seguido pela expressão E que se pretende verificar, por exemplo:

```
deref:
q: pointer(pointer(integer));
deref(deref(q))
```

$$\forall \alpha, pointer(\alpha) \xrightarrow[deref]{} \alpha$$

- A árvore de *parse* para a expressão $deref_o(deref_i(q))$, cujo tipo se pretende verificar é a seguinte:



- Cada aplicação da função **deref** retira um nível de referência

27