Análise semântica

Professor: Luciano Ferreira Silva, Dr.

Análise semântica

- **A análise sintática consegue verificar se uma expressão obedece às regras de formação de uma dada gramática;**
- **Mas seria muito difícil expressar por meio de gramáticas livres de contexto algumas regras usuais em linguagens de programação como:**
  - ✓ Todas as variáveis devem ser declaradas;
  - ✓ Situações em que o tipo da variável deve ser verificado;
- **A análise semântica vem trabalhar exatamente neste tipo de problemas.**

Análise semântica

- **A análise semântica é realizada por meio de heurísticas;**
  - ✓ Não possui o mesmo grau de formalismo associado às análises léxica e sintática;
  - ✓ Isto ocorre por que para representar tais regras seria necessário o uso de gramáticas sensíveis ao contexto;
    - Para as quais não há mecanismos adequados de processamento automático

Análise semântica

- **As tarefas básicas desempenhadas durante a análise semântica incluem:**
  - ✓ Verificação de tipos;
  - ✓ Verificação do fluxo de controle;
  - ✓ Verificação da unicidade da declaração de variáveis;
- **Dependendo da linguagem de programação, outros tipos de verificação podem ser necessários;**

Verificação de tipos

- **Exemplo C++:**

```
#include<iostream>
using namespace std;

int main(){
        a = 10;
        cout<<"Valor de a: "<< a <<endl;
}
```

- **Cada instrução isoladamente está correta;**
  - ✓ O compilador consegue construir, para cada comando, uma árvore sintática correta;
- **No entanto, a compilação desse programa apresenta a seguinte informação de erro:**

Verificação de tipos

```
...cpp: In function ‘int main()’:
...cpp:5: error: ‘a’ was not declared in this scope
```

- Na primeira utilização que se tenta fazer da variável a, o compilador reconhece que tal variável não foi declarada.
- Para poder detectar esse tipo de erro, ele precisa manter internamente a informação sobre quais variáveis já foram declaradas e podem ser utilizadas.
- Tal informação é mantida na tabela de símbolos (tipo de estrutura de dados).

Verificação de tipos

- **Exemplo C++:**

```
#include<iostream>
using namespace std;

int main(){
        int a = 9;
        float b = 5;
        cout<<"a%b: "<< a%b <<endl;
}
```

- **Cada instrução isoladamente está correta;**
- **As duas variáveis envolvidas foram declaradas;**
- **No entanto, a compilação desse programa apresenta a seguinte informação de erro:**

Verificação de tipos


```
...cpp: In function ‘int main()’:
...cpp:7: error: invalid operands of types ‘int’ and ‘float’
   to binary ‘operator%’
```

- O operador % admite apenas operandos inteiros, mas b é do tipo real (`float`);
- Para poder detectar este erro, a informação sobre o tipo de cada variável também é mantida na tabela de símbolos;

Verificação de tipos

- **Exemplo C++:**

```
#include<iostream>
using namespace std;

void mostra();
int main(){
        int a = 9;
        mostra();
}
void mostra(){
        cout<<"a:"<< a <<endl;
}
```

- **A compilação desse programa apresenta a seguinte informação de erro:**

Verificação de tipos

```
...cpp: In function ‘int main()’:
...cpp:5: error: ‘a’ was not declared in this scope
```

- Embora a mensagem seja parecida com aquela do primeiro exemplo, nesse caso existe uma variável `a` na tabela de símbolos, pois ela foi declarada na função `main`;
- No entanto, ao tentar usá-la na função `mostra`, o compilador reconheceu que a variável não era válida;
  - ✓ Pois `a` é uma variável local a `main` e portanto só pode ser utilizada no corpo dessa função;
- Esse tipo de informação sobre o escopo das variáveis declaradas também é mantido na tabela de símbolos.

Verificação de tipos

- **Exemplo C++ relativo ao escopo de identificadores:**

```
int a, b;
   ... //  escopo 1
void f(){
        float a, c;
   ... // escopo 2
}
   ... // escopo 1
void g(){
        float c, d;
   ... // escopo 3
}
```

- **Escopo 1: variáveis declaradas inteiras *a* e *b* e as funções *f* e *g*;**
- **Escopo 2 (dentro da função *f*): uma variável real *c*, e a variável *a* do escopo 1 é sobreposta pela variável real *a*, a variável *b* do escopo 1 ainda é válida;**
- **Escopo 3 (dentro da função *g*): duas variáveis reais locais *c* e *d*, e as duas variáveis do escopo 1 *a* e *b*, sendo que *c* deste escopo é diferente *c* do escopo 2;**

Verificação de tipos

- **Há duas formas básicas de manter a informação sobre o escopo:**

1. Trabalhar com múltiplas tabelas, uma para cada escopo.
   - Assim, quando o compilador realiza a análise semântica sobre o uso dos indicadores, ele utiliza como referência a tabela de símbolos como uma relativa àquele escopo;
2. Organizar a tabela de símbolos como uma pilha de tabelas;
   - a cada nova definição de escopo um conjunto de símbolos é agregado à pilha e ao final do escopo, esse conjunto é desempilhado.

Verificação de tipos

- **Todos os identificadores utilizados em um programa, e não apenas variáveis, devem estar presentes na tabela de símbolos;**
- **Outra informação que deve estar presente na tabela de símbolos é um atributo que indique o tipo de identificador ao qual aquele nome está associado:**
  - ✓ Uma variável;
  - ✓ Uma função;
  - ✓ Uma estrutura ou classe;
  - ✓ Uma palavra reservada da linguagem.

Verificação de tipos

- **A estratégia usada pelo compilador para tratar estes nomes (*name mangling*) é não padronizada e cada projetista adota a sua maneira:**
  - ✓ Pode-se usar nomes iguais para representar identificadores diferentes ou não;
  - ✓ Pode-se criar um nome interno para representar cada identificador,
  - ✓ O modo com que será feito este processo é variável, etc;
- **As linguagens C e C++ apesar de serem parecidas, possuem diferentes compiladores;**
- **Existem diferentes compiladores para linguagens iguais;**

Verificação de tipos

- **Com a tabela de símbolos , por meio da análise semântica, o compilador pode realizar:**
  - ✓ A verificação de que o uso dos identificadores está de acordo com sua definição;
- **Existe dois tipos de verificação:**
  - ✓ A estática:
    - Detecta erros da análise apenas do código-fonte;
    - Existem situações de erro que só podem ser detectadas durante a execução do programa;
      - Como a atribuição de um valor a uma variável além do limite representável por seu tipo;
  - ✓ A dinâmica:
    - Detecta situações de erro durante a execução do programa.

Verificação de tipos

- Outro exemplo em C++ de verificação de tipo:

```
#include<iostream>
using namespace std;

int main(){
        int a = 0xFF0;
        int* b;

        b = a;
}
```

- A compilação desse programa apresenta a seguinte informação de erro:

```
...cpp: In function ‘int main()’:

...cpp:8: error: invalid conversion from ‘int’ to ‘int*’
```

Verificação de tipos

- Ocorreu uma tentativa inválida de conversão de uma variável escalar, do tipo `int`, para uma variável ponteiro, do tipo `int*`;
- Em alguns casos, o compilador realiza a conversão automática de um tipo para outro que seja adequado à aplicação do operador;

Verificação de tipos

- Exemplo em C:

```
a = x – ‘0’;
```

- A constante do tipo caractere *‘0’* é automaticamente convertida para inteiro para compor corretamente a expressão aritmética na qual ele toma parte;
- Todo *char* em uma expressão é convertido pelo compilador para um *int*;
- Esse procedimento de conversão de tipo é denominado coerção (*cast*).

Verificação de tipos

- **Em C e C++ a seguinte seqüência determina a realização automática de coerção em expressões aritméticas com dois operandos:**

1. `char` e `short` são convertidos para `int`, `float`, para `double`;
2. Se um dos operandos é `double`, o outro é convertido para `double` e o resultado é `double`;
3. Se um dos operandos é `long`, o outro é convertido para `long` e o resultado é `long`;
4. Se um dos operandos é `unsigned`, o outro é convertido para `unsigned` e o resultado é `unsigned`;
5. Senão, todos o operandos são `int` e o resultado é `int`;

Verificação de tipos

- Quando uma conversão imprevista ocorre, o compilador envia uma mensagem de erro;
- Porém, o programador pode indicar para o compilador uma conversão explicita e forçar uma coerção;

```
int main(){
        int a = 0xFF0;
        int* b;

        b = (int*)a;
}
```

- Neste caso, nenhuma mensagem é gerada pelo compilador;

Verificação de fluxo

- **Na verificação de fluxo o objetivo é detectar erros nas estruturas de controle de fluxo de execução, como:**
  - ✓ em repetições (`for`, `do`, `while`)
  - ✓ em alternativas (`if else`, `switch case`);
- **Exemplo:**

```
void f2 (int j, int k){
   if (j==k)
        break;
   else
        continue;
}
```

Verificação de fluxo

- **A compilação desse programa apresenta a seguinte informação de erro:**

```
In function ‘f2’:
...: break statement not within loop or switch
...: continue statement not within loop
```

- **Ele reconhece que o comando `break` só pode ser usado para:**
  - ✓ Quebrar a seqüência de um comando de interação (*within loop*);
  - ✓ Indicar o fim de um bloco associado à execução de um `case` (*within switch*);
- **Um comando `continue` só pode ser usado em um comando de iteração para:**
  - ✓ Indicar que a iteração corrente já está encerrada e que a execução deve prosseguir com a reavaliação da condição de repetição;

Verificação de unicidade

- **Na verificação de unicidade detecta situações tais como duplicação em:**
  - ✓ declarações de variáveis;
  - ✓ componentes de estruturas;
  - ✓ rótulos do programa;
- **Exemplo:**

```
void f3 (int k){
   struct{
   int a;
   float a;
   }x;
   float x;
```

Verificação de unicidade

```
    switch(k){
    case 0x31: x.a = k;
    case '1': x = x.a;
    }
}
```

- A compilação desse programa apresenta a seguinte informação de erro:

```
In function 'f3':
...: duplicate member 'a'
...: previous declaration of 'x'
...: duplicate case value
```

Verificação de unicidade

1. **A primeira mensagem detecta a definição de dois membros na mesma estrutura com o mesmo nome, `a`, o que não é permitido;**
2. **A segunda mensagem refere-se à situação de que há duas variáveis de mesmo nome, `x`;**
3. **A terceira mensagem indica que dois `case`s em uma expressão `switch` receberam o mesmo rótulo, o que não é permitido;**

- ✓ Observe que, embora a forma de expressar o valor nas duas opções do comando `case` tenha sido diferente;