Tema 3

Análise Semântica

Gramáticas de atributos, tabela de símbolos

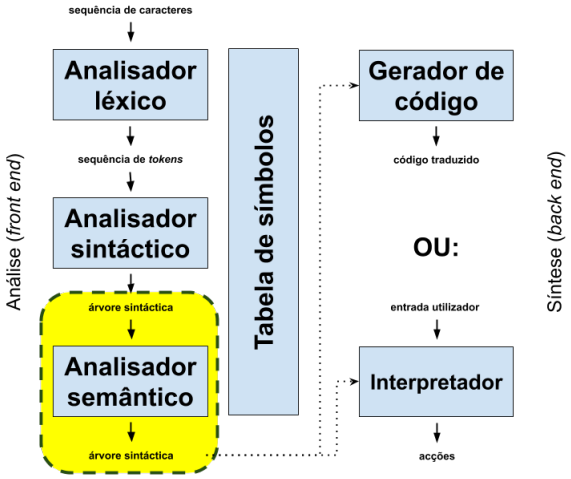
Compiladores, 2º semestre 2023-2024

Miguel Oliveira e Silva, Artur Pereira, DETI, Universidade de Aveiro

Conteúdo

1 Análise semântica: Estrutura de um Compilador

- Vamos agora analisar com mais detalhe a fase de análise semântica:



- No processamento de uma linguagem a análise semântica deve garantir, tanto quanto possível, que o programa fonte faz sentido (mediante as regras definidas na linguagem).
- Erros semânticos comuns:
  - Variável/função não definida;
  - Tipos incompatíveis (e.g. atribuir número real a uma variável inteira, ou utilizar uma expressão não booleana na condições de uma instrução condicional);
  - Definir instrução num contexto errado (e.g. utilizar em `Java` a instrução `break` fora de um ciclo ou `switch`).
  - Aplicação sem sentido de instrução (e.g. importar uma *package* inexistente em `Java`).
- Em alguns casos, estes erros podem ser avaliados ainda durante a análise sintáctica; noutros casos, só é possível fazer essa verificação após uma análise sintáctica bem sucedida, fazendo uso da informação retirada dessa análise.

1.1 Avaliação dirigida pela sintaxe

- No processamento de linguagens, a avaliação semântica pode ser feita associando informação e acções às regras sintácticas da gramática (i.e. aos nós da *árvore sintáctica*).
- Este procedimento designa-se por *avaliação dirigida pela sintaxe*.
- Por exemplo, numa gramática para expressões aritméticas podemos associar aos nós da árvore uma variável com o tipo da expressão, e acções que permitam verificar a sua correcção (e não permitir, por exemplo, que se tente somar um booleano com um inteiro).
- Em ANTLR4, a associação de atributos e acções à árvore sintáctica, pode ser feita durante a própria análise sintáctica, e/ou posteriormente recorrendo a *visitors* e/ou *listeners*.

1.2 Detecção estática ou dinâmica

- A verificação de cada propriedade semântica de uma linguagem pode ser feita em dois tempos distintos:
  - Em *tempo dinâmico*: isto é, durante o *tempo de execução*;
  - Em *tempo estático*: isto é, durante o *tempo de compilação*.
- Só em compiladores fazem sentido verificações estáticas de propriedades semânticas.

- Em interpretadores as fases de análise e síntese da linguagem são ambas feitas em tempo de execução, pelo que as verificações são sempre dinâmicas.
- A verificação estática tem a vantagem de garantir, em tempo de execução, que certos erros nunca vão ocorrer (dispensando a necessidade de proceder à sua depuração e teste).

2 Sistema de tipos

- O sistema de tipos de uma linguagem de programação é um sistema lógico formal, com um conjunto de regras semânticas, que por associação de uma propriedade (tipo) a entidades da linguagem (expressões, variáveis, métodos, etc.) permite a detecção de uma classe importante de erros semânticos: *erros de tipos*.
- A verificação de erros de tipo, é aplicável nas seguintes operações:
  - Atribuição de valor: $v = e$
  - Aplicação de operadores: $e_1 + e_2$ (por exemplo)
  - Invocação de funções: $f(a)$
  - Utilização de classes/estruturas: $o.m(a)$ ou $data.field$
- Outras operações, como por exemplo a utilização arrays, podem também envolver verificações de tipo. No entanto, podemos considerar que as operações sobre arrays são atribuições de valor e aplicação de métodos especiais.
- Diz-se que qualquer uma destas operações é válida quando existe *conformidade* entre as propriedades de tipo das entidades envolvidas.
- A conformidade indica se um tipo T_2 pode ser usado onde se espera um tipo T_1. É o que acontece quando $T_1 = T_2$.
  - Atribuição de valor ($v = e$).
    O tipo de e tem de ser conforme com o tipo de v
  - Aplicação de operadores ($e_1 + e_2$).
    Existe um operador $+$ aplicável aos tipos de e_1 e e_2
  - Invocação de funções ($f(a)$).
    Existe uma função global f que aceita argumentos a conformes com os argumentos formais declarados dessa função.
  - Utilização de classes/estruturas ($o.m(a)$ ou $data.field$).
    Existe um método m na classe correspondente ao objecto o, que aceita argumentos a conformes com os argumentos formais declarados desse método; e existe um campo $field$ na estrutura/classe de $data$.

3 Gramáticas de atributos

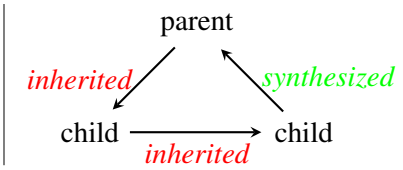
- Já vimos que atribuir sentido ao código fonte de uma linguagem requer, não só, correcção sintáctica (assegurada por gramáticas independentes de contexto) como também correcção semântica.
- Nesse sentido, é de toda a conveniência ter acesso a toda a informação gerada pela análise sintáctica, i.e. à árvore sintáctica, e poder associar nova informação aos respectivos nós.
- Este é o objectivo da *gramática de atributos*:
  - Cada símbolo da gramática da linguagem (terminal ou não terminal) pode ter a si associado um conjunto de zero ao mais *atributos*.
  - Um atributo pode ser um número, uma palavra, um tipo, ...
  - O cálculo de cada atributo tem de ser feito tendo em consideração a dependência da informação necessária para o seu valor.

- Entre os diferentes tipos de atributos, existem alguns cujo valor depende apenas da sua vizinhança sintáctica.
  - Um desses exemplos é o valor de uma expressão aritmética (que para além disso, depende apenas do próprio nó e, eventualmente, de nós descendentes).
- Existem também atributos que (podem) depender de informação remota.
  - É o caso, por exemplo, do tipo de dados de uma expressão que envolva a utilização de uma variável ou invocação de um método.

3.1 Dependência local: classificação de atributos

- Os atributos podem ser classificados duas formas, consoante as dependências que lhes são aplicáveis:
  1. Dizem-se *sintetizados*, se o seu valor depende apenas de nós descendentes (i.e. se o seu valor depende apenas dos símbolos existentes no respectivo corpo da produção).
  2. Dizem-se *herdados*, se depende de nós "irmãos"ou de nós ascendentes.

parent → child child



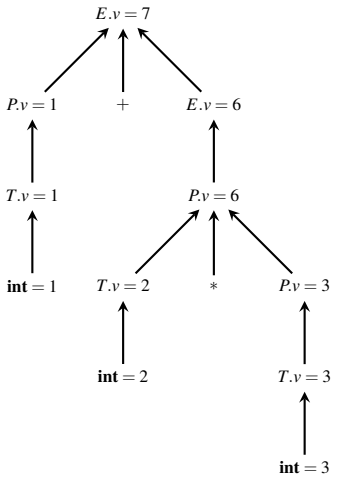
- Formalmente podem-se designar os atributos anotando com uma seta no sentido da dependência (para cima, nos atributos sintetizados; e para baixo nos herdados).

Exemplo dependência local: expressão aritmética

- Considere a seguinte gramática:

$$\begin{aligned} E &\rightarrow P + E \mid P \\ P &\rightarrow T * P \mid T \\ T &\rightarrow (E) \mid \textbf{int} \end{aligned}$$

- Se quisermos definir um atributo v para o valor da expressão, temos um exemplo de um atributo sintetizado.
- Por exemplo, para a entrada — $1+2*3$ — temos a seguinte árvore sintáctica anotada:



$$\begin{aligned} E &\rightarrow P + E \mid P \\ P &\rightarrow T * P \mid T \\ T &\rightarrow (E) \mid \textbf{int} \end{aligned}$$

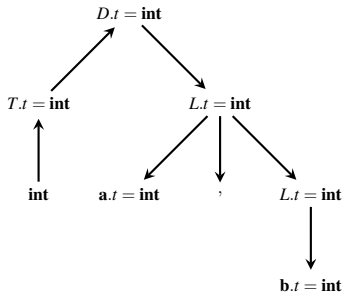
Produção	Regra semântica
$E_1 \rightarrow P + E_2$	$E_1.v = P.v + E_2.v$
$E \rightarrow P$	$E.v = P.v$
$P_1 \rightarrow T * P_2$	$P_1.v = T.v * P_2.v$
$P \rightarrow T$	$P.v = T.v$
$T \rightarrow (E)$	$T.v = E.v$
$T \rightarrow$ **int**	$T.v =$ **int**$.value$

Exemplo dependência local: declaração

- Considere a seguinte gramática:

$$\begin{aligned} D &\rightarrow T\,L \\ T &\rightarrow \textbf{int} \mid \textbf{real} \\ L &\rightarrow \textbf{id}, L \mid \textbf{id} \end{aligned}$$

- Se quisermos definir um atributo `t` para indicar o tipo de cada variável **id**, temos um exemplo de um atributo herdado.
- Por exemplo, para a entrada — **int** a, b — temos a seguinte árvore sintáctica anotada:



$$\begin{aligned} D &\rightarrow T\,L \\ T &\rightarrow \textbf{int} \mid \textbf{real} \\ L &\rightarrow \textbf{id}, L \mid \textbf{id} \end{aligned}$$

Produção	Regra semântica
$D \rightarrow T\,L$	$D.t = T.t$
	$L.t = T.t$
$T \rightarrow$ **int**	$T.t =$ **int**
$T \rightarrow$ **real**	$T.t =$ **real**
$L_1 \rightarrow$ **id**$, L_2$	**id**$.t = L_1.t$
	$L_2.t = L_1.t$
$L \rightarrow$ **id**	**id**$.t = L.t$

3.2 ANTLR4: Declaração de atributos associados à árvore sintáctica

- Podemos declarar atributos de formas distintas:
  1. Directamente na gramática independente de contexto recorrendo a argumentos e resultados de regras sintácticas;

```
expr[String type] returns[int value]: // type not used
      e1=expr '+' e2=expr
      {$value = $e1.value + $e2.value;}        #ExprAdd
```

```
    |  INT
       {$value = Integer.parseInt($INT.text);} #ExprInt
    ;
```

2. Indirectamente fazendo uso do *array* associativo `ParseTreeProperty`:

```
protected ParseTreeProperty<Integer> value =
   new ParseTreeProperty<>();
...
@Override public void exitInt(ExprParser.IntContext ctx){
   value.put(ctx, Integer.parseInt(ctx.INT().getText()));
}
...
@Override public void exitAdd(ExprParser.AddContext ctx){
   int left = value.get(ctx.e1);
   int right = value.get(ctx.e2);
   value.put(ctx, left + right);
}
```

- Podemos ainda utilizar o resultado dos métodos `visit`.

ANTLR4: Declaração de atributos `ParseTreeProperty`

- Este *array* tem como chave nós da árvore sintáctica, e permite simular quer argumentos, quer resultados, de regras.
- A diferença está nos locais onde o seu valor é atribuído e acedido.
- Para simular a passagem de *argumentos* basta atribuir-lhe o valor *antes* de percorrer o respectivo nó (nos *listeners* usualmente nos métodos `enter..`), sendo o acesso feito no *próprio* nó.
- Para simular *resultados*, faz-se como no exemplo dado (i.e. atribui-se o valor no *próprio* nó, e acede-se nos nós *ascendentes*).

Gramáticas de atributos em ANTLR4: síntese

- Podemos associar três tipos de informação a regras sintácticas:
  1. Informação com origem em regras utilizadas no corpo da regra (atributos sintetizados);
  2. Informação com origem em regras que utilizam esta regra no seu corpo (atributos herdados);
  3. Informação local à regra.
- Em `ANTLR4` a utilização directa de todos estes tipos de atributos é muito simples e intuitiva:
  1. Atributos sintetizados: resultado de regras;
  2. Atributos herdados: argumentos de regras;
  3. Atributos locais.
- Alternativamente, podemos utilizar o *array* associativo `ParseTreeProperty` (que se justifica apenas para as duas primeiras, já que para a terceira podemos utilizar variáveis locais ao método respectivo); ou o resultado dos métodos `visit` (no caso de se utilizar *visitors*) para atributos sintetizados.

4 Tabela de símbolos

- A gramática de atributos é adequada para lidar com atributos com dependência local.
- No entanto, podemos ter informação cuja origem não tem dependência directa na árvore sintáctica (por exemplo, múltiplas aparições duma variável), ou que pode mesmo residir no processamento de outro código fonte (por exemplo, nomes de classes definidas noutro ficheiro).
- Assim, sempre que a linguagem utiliza símbolos para representar entidades do programa – como sejam: variáveis, funções, registos, classes, etc. – torna-se necessário associar à identificação do símbolo (geralmente um identificador) a sua definição (categoria do símbolo, tipo de dados associado).

- É para esse fim que existe a *tabela de símbolos*.
- A tabela de símbolos é um *array* associativo, em que a chave é o nome do símbolo, e o elemento um objecto que define o símbolo.
- As tabelas de símbolos podem ter um alcance global, ou local (por exemplo: a uma bloco de código ou a uma função).

- A informação associada a cada símbolo depende do tipo de linguagem definida, assim como de estarmos na presença de um interpretador ou de um compilador.
- São exemplos dessas propriedades:
  - **Nome**: nome do símbolo (chave do *array* associativo);
  - **Categoria**: o que é que o símbolo representa, classe, método, variável de objecto, variável local, etc.;
  - **Tipo**: tipo de dados do símbolo;
  - **Valor**: valor associado ao símbolo (apenas no caso de interpretadores).
  - **Visibilidade**: restrição no acesso ao símbolo (para linguagens com encapsulamento).

Tabela de símbolos: implementação

- Numa aproximação orientada por objectos podemos definir a classe abstracta `Symbol`:

```
public abstract class Symbol {
   public Symbol(String name, Type type) { ... }
   public String name() { ... }
   public Type type() { ... }
}
```

- Podemos agora definir uma variável:

```
public class VariableSymbol extends Symbol {
   public VariableSymbol(String name, Type type) {
      super(name, type);
   }
}
```

- A classe `Type` permite a identificação e verificação da conformidade entre tipos:

```
public abstract class Type {
   protected Type(String name) { ... }
   public String name() { ... }
   public boolean subtype(Type other) {
      assert other != null;
      return name.equals(other.name());
   }
}
```

- Podemos agora implementar tipos específicos:

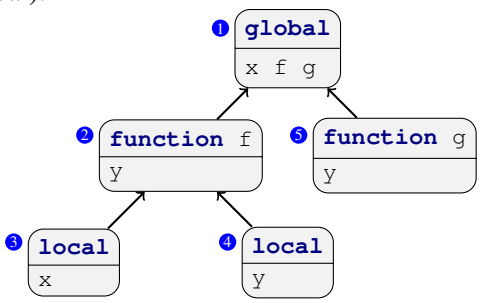
```
public class RealType extends Type {
   public RealType() { super("real"); }
}


public class IntegerType extends Type {
   public IntegerType() { super("integer"); }

   public boolean subtype(Type other) {
      return super.subtype(other) ||
             other.name().equals("real");
   }
}
```

4.1 Agrupando símbolos em contextos

- Se a linguagem é simples, contendo um único contexto de definição de símbolos, então o tempo de vida dos símbolos está ligado ao tempo de vida do programa, sendo suficiente uma única tabela de símbolos.
- No entanto, se tivermos a possibilidade de definir símbolos em contextos diferentes, então precisamos de resolver o problema dos símbolos terem tempos de vida (e/ou visibilidade) que dependem do contexto dentro do programa.
- Considere como exemplo o seguinte código (na linguagem `C`):

```
❶ // start of global scope
  int x;          // define variable x in global scope
❷ void f() {      // define function f in global scope
    int y;        // define variable y in local scope of f
❸   { int x; }   // define variable x in nested local scope
❹   { int y; }   // define variable y in another nested local scope
  }
❺ void g() {      // define function g in global scope
    int y;        // define variable y in local scope of g
  }
...
```

- A numeração identifica os diferentes contextos de símbolos.
- Um aspecto muito importante é o facto dos contextos poderem ser definidos dentro de outros contextos.
- Assim o contexto ❷ está definido dentro do contexto ❶; e, por sua vez, o contexto ❸ está definido dentro do ❷.
- Em ❹ o símbolo `x` está definido em ❶.
- Para representar adequadamente esta informação estrutura-se as diferentes tabelas de símbolos numa árvore onde cada nó representa uma pilha de tabelas de símbolos a começar nesse nó até à raiz (tabela de símbolos global).



- Consoante o ponto onde estamos no programa. temos uma pilha de tabelas de símbolos definida para resolver os símbolos.
- Pode haver repetição de nomes de símbolos, valendo o definido na tabela mais próxima (no ordem da pilha).
- Caso seja necessário percorrer a árvore sintáctica várias vezes, podemos registar numa lista ligada a sequência de pilhas de tabelas de símbolos que são aplicáveis em cada ponto do programa.

5 Instruções restringidas por contexto

- Algumas linguagens de programação restringem a utilização de certas instruções a determinados contexto.
- Por exemplo, em Java as instruções `break` e `continue` só podem ser utilizadas dentro de ciclos ou da instrução condicional `switch`.
- A verificação semântica desta condição é muito simples de implementar, podendo ser feita durante a análise sintáctica recorrendo a predicados semânticos e um contador (ou uma pilha) que registe o contexto.

```
@parser::members {
   int acceptBreak=0;
}
...
forLoop: 'for' '(' expr ';' expr ';' expr ')'
         {acceptBreak++;}
         instruction
         {acceptBreak--;}
       ;
break: {acceptBreak > 0}? 'break' ';'
     ;
instruction: forLoop | break | ...
           ;
```