Tema 4
Síntese
Geração de código

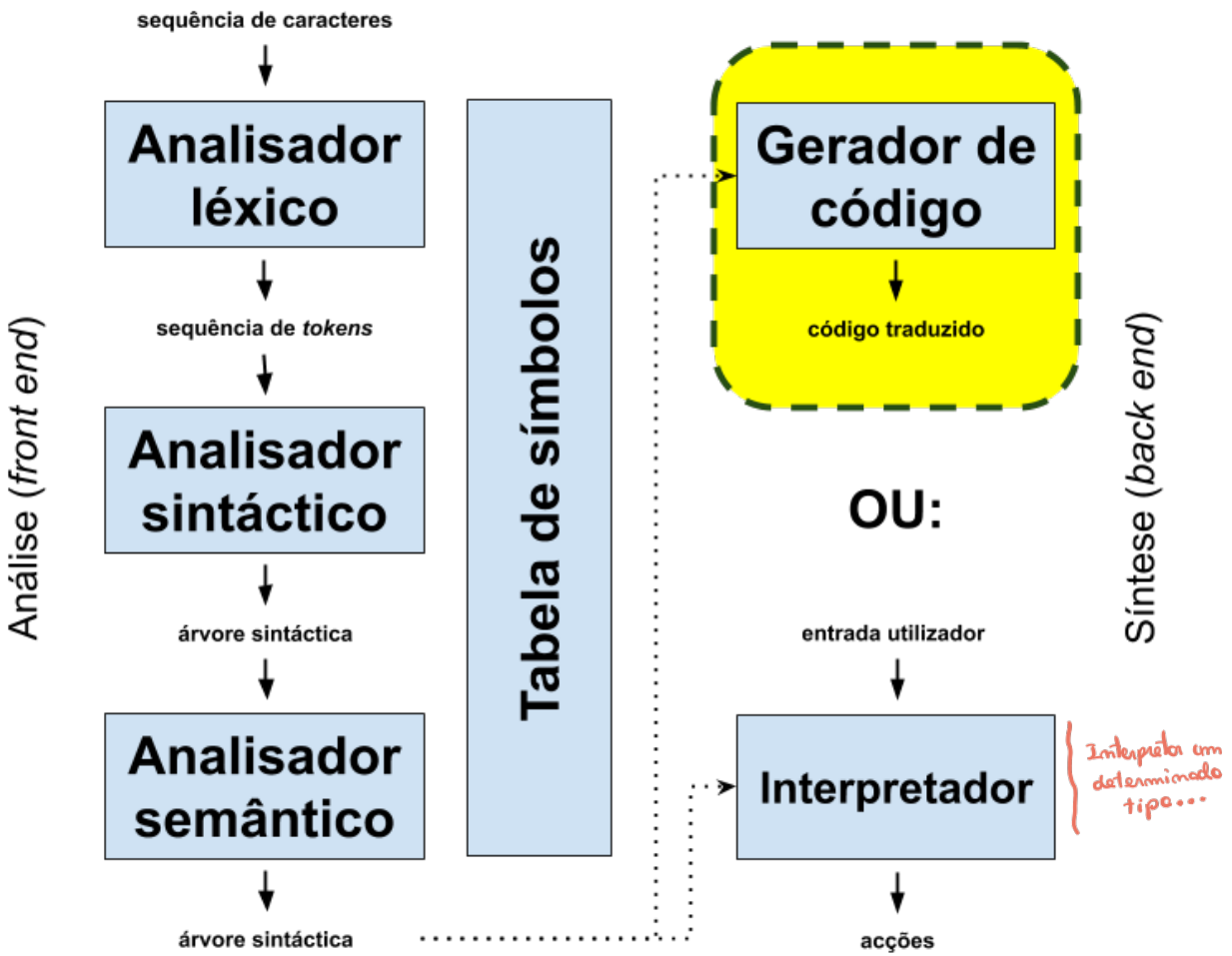
Compiladores, 2º semestre 2023-2024

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

Síntese: geração de código

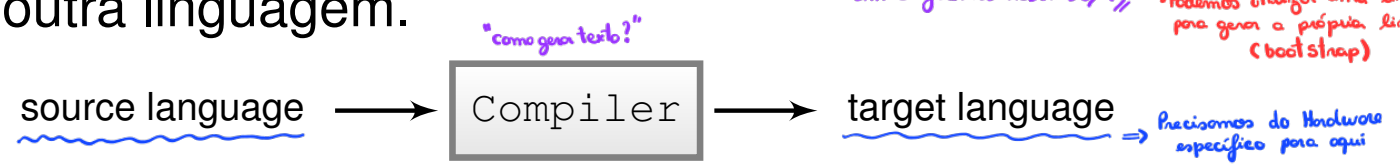
Síntese: geração de código

Informação estruturada pode ser sempre interpretada assim ...



Síntese: geração de código (2)

- Podemos definir o objectivo de um compilador como sendo *traduzir* o código fonte de uma linguagem para outra linguagem.

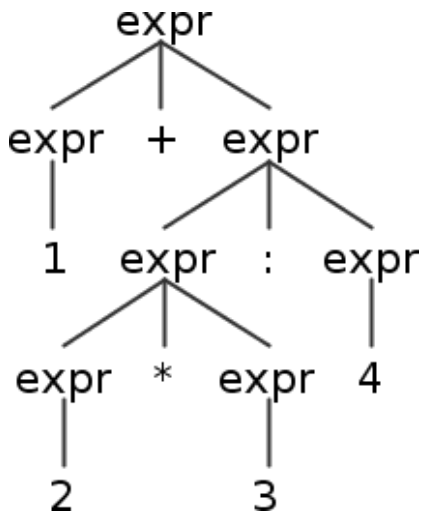


- A geração do código para a linguagem destino pode ser feita por diferentes fases (podendo incluir fases de optimização), mas nós iremos abordar apenas uma única fase.
- A estratégia geral consiste em identificar padrões de geração de código, e após a análise semântica percorrer novamente a árvore sintáctica (mas já com a garantia muito importante de inexistência de erros sintácticos e semânticos) gerando o código destino nos pontos apropriados.

Exemplo: Calculadora

- Código fonte:

```
1+2*3:4
```



COMPILAR!

- Uma possível compilação para Java:

```java
public class CodeGen {
   public static void main(String[] args) {
      int v2 = 1;
      int v5 = 2;
      int v6 = 3;
      int v4 = v5 * v6;
      int v7 = 4;
      int v3 = v4 / v7;
      int v1 = v2 + v3;
      System.out.println(v1);
   }
}
```

Geração de código máquina

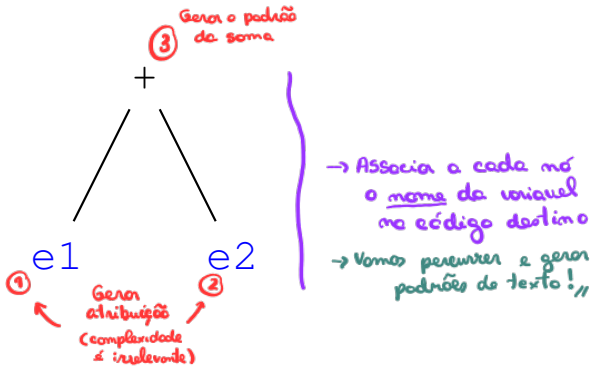
- Tradicionalmente, o ensino de processadores de linguagens tende a dar primazia à geração de código baixo nível (linguagem máquina, ou *assembly*).
- A larga maioria da bibliografia mantém esse enfoque.
- No entanto, do ponto de vista prático serão poucos os programadores que, fazendo uso de ferramentas para gerar processadores de linguagens, necessitam ou ambicionam este tipo de geração de código.
- Nesta disciplina vamos, alternativamente, discutir a geração de código numa perspectiva mais abrangente, incluindo a geração de código em linguagens de alto nível.

Geração de código máquina (2)

- No que diz respeito à geração de código em linguagens de baixo nível, é necessário um conhecimento robusto em arquitectura de computadores e lidar com os seguintes aspectos:
  - Representação e formato da informação (formato para números inteiros, reais, estruturas, *array*, etc.);
  - Gestão e endereçamento de memória;
  - Implementação de funções (passagem de argumentos e resultado, suporte para recursividade com pilha de chamadas e *frame pointers*);
  - Alocação de registos do processador.
- (Consultar a bibliografia recomendada para estudar este tipo de geração de código.)

Geração de código

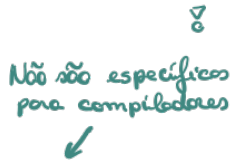
- Seja qual for o nível da linguagem destino, uma possível estratégia para resolver este problema consiste em identificar sem ambiguidade padrões de geração de código associados a cada elemento gramatical da linguagem.
- Para esse fim, é necessário definir o contexto de geração de código para cada elemento (por exemplo, geração de instruções na linguagem destino, ou atribuir a valor a uma variável), e depois garantir que o mesmo é compatível com todas as utilizações do elemento.



$$
\begin{cases} \cdots (e_1) \\ v_1 = e_1 \end{cases}_{1}
\quad
\begin{cases} \cdots (e_2) \\ v_2 = e_2 \end{cases}_{2}
\qquad
\{ v_+ = v_1 + v_2 \}_{3}
$$

Geração de código (2)

- Como a larguíssima maioria das linguagens destino são textuais, esses padrões de geração de código consistem em padrões de geração de texto.
- Assim sendo, em `Java`, poderíamos delegar esse problema no tipo de dados `String`, `StringBuilder`, ou mesmo na escrita directa de texto em em ficheiro (ou no *standard output*).
- No entanto, também aí o ambiente `ANTLR4` fornece uma ajuda mais estruturada, sistemática e modular para lidar com esse problema.



Podíamos usar o String Builder ou o String ...

↳

String Template

↳ Classe que vamos usar para fazer um Compilador em sentido estrito

String Template

- A biblioteca (`Java`) *String Template* fornece uma solução estruturada para a geração de código textual.
- O software e documentação podem ser encontrados em `http://www.stringtemplate.org`
- Para ser utilizada é apenas necessário o pacote `ST-4.?.jar` (a instalação feita do antlr4 já incluiu este pacote).
- Vejamos um exemplo simples:

```
import org.stringtemplate.v4.*;
...
// code gen. pattern definition with <name> hole:
ST hello = new ST("Hello ,␣<name>");
// hole pattern definition:
hello.add("name", "World");
// code generation (to standard output):
System.out.println(hello.render());
```

- Mesmo sendo um exemplo muito simples, podemos já verificar que a definição do padrão de texto, está separada do preenchimento dos "buracos" (atributos ou expressões) definidos, e da geração do texto final.

String Template (2)

- Podemos assim delegar em partes diferentes do gerador de código, a definição dos padrões (que passam a pertencer ao contexto do elemento de código a gerar), o preenchimento dos "buracos" definidos, e a geração do texto final de código.
- Os padrões são blocos de texto e expressões.
- O texto corresponde a código destino literal, e as expressões são em "buracos" que podem ser preenchidos com o texto que se quiser.
- Sintaticamente, as expressões são identificadores delimitados por `<expr>` (ou por `$`).

```
import org.stringtemplate.v4.*;
...
ST assign = new ST("<var> = <expr>;\n");
assign.add("var", "i");
assign.add("expr", "10");
String output = assign.render();
System.out.println(output);
```

String Template Group

- Podemos também agrupar os padrões numa espécie de funções (módulo `STGroup`):

```
import org.stringtemplate.v4.*;
...
STGroup group = new STGroupString(
        "assign(var,expr) ::= \"<var> = <expr>;\""
    );
ST assign = group.getInstanceOf("assign");
assign.add("var", "i");
assign.add("expr", "10");
String output = assign.render();
System.out.println(output);
```

String Template Group (2)

- Podemos também colocar cada função num ficheiro:

```
// file assign.st
assign(var,expr) ::= "<var> = <expr>;"
```

```
import org.stringtemplate.v4.*;
...
// assuming that assign.st is in current directory:
STGroup group = new STGroupDir(".");
ST assign = group.getInstanceOf("assign");
assign.add("var", "i");
assign.add("expr", "10");
String output = assign.render();
System.out.println(output);
```

String Template Group (3)

- Uma melhor opção é optar por ficheiros modulares contendo grupos de funções/padrões:

```
// file templates.stg

templateName(arg1, arg2, ..., argN) ::= "single-line template"

templateName(arg1, arg2, ..., argN) ::= <<
multi-line template preserving indentation and newlines
>>

templateName(arg1, arg2, ..., argN) ::= <%
multi-line template that ignores indentation and newlines
%>
```

```
import org.stringtemplate.v4.*;
...
// assuming that templates.stg is in current directory:
STGroup allTemplates = new STGroupFile("templates.stg");
ST st = group.getInstanceOf("templateName");
...
```

String Template: dicionários e condicionais

- Neste módulos podemos ainda definir dicionários (arrays associativos).

```
typeValue ::= [
    "integer":"int",
    "real":"double",
    "boolean":"boolean",
    default:"void"
]
```

- Na definição de padrões podemos utilizar uma instrução condicional que só aplica o padrão caso o atributo seja adicionado:

```
stats(stat) ::= <<
<if(stat)><stat; separator="\n"><endif>
>>
```

- O campo `separator` indica que em em cada operação `add` em `stat`, se irá utilizar esse separador (no caso, uma mudança de linha).

String Template: Funções

- Podemos ainda definir padrões utilizando outros padrões (como se fossem funções).

```
module(name,stat) ::= <<
public class <name> {
   public static void main(String[] args) {
      <stats(stat)>
   }
}
>>

conditional(stat,var,stat_true,stat_false) ::= <<
<stats(stat)>
if (<var>) {
   <stat_true>
}<if(stat_false)>
else {
   <stat_false>
}<endif>
>>
```

String Template: listas

- Também existe a possibilidade de utilizar listas para concatenar texto e argumentos de padrões:

```
binaryExpression(type,var,e1,op,e2) ::=
   "<decl(type,var,[e1,\" \",op,\" \",e2])>"
```

- ou:

```
binaryExpression(type,var,e1,op,e2) ::= <<
<decl(type,var,[e1," ",op," ",e2])>
>>
```

=> tem 3 buracos type, var, valor

queremos um op binário

Para colocar um espaço

- Para mais informação sobre as possibilidades desta biblioteca devem consultar a documentação existente em: `http://www.stringtemplate.org`.

Geração de código: padrões comuns

→ Queremos gerar código, pendurado na árvore sintática!

→ e tendo atenção apenas aos filhos

Geração de código: padrões comuns

- Uma geração de código modular requer um contexto uniforme que permita a inclusão de qualquer combinação de código a ser gerado.
- Na sua forma mais simples, o padrão comum pode ser simplesmente uma sequência de instruções.

```
stats(stat) ::= <<
<if(stat)><stat; separator="\n"><endif>
>>

module(name,stat) ::= <<
public class <name>
{
   public static void main(String[] args)
   {
      <stats(stat)>
   }
}
>>
```

- Com este padrão, podemos inserir no lugar do "buraco" `stat` a sequência de instruções que quisermos.
- Naturalmente, que para uma geração de código mais complexa podemos considerar a inclusão de buracos para membros de classe, múltiplas classes, ou mesmo vários ficheiros.

Geração de código: padrões comuns (2)

- Para a linguagem C, teríamos o seguinte padrão para um módulo de compilação:

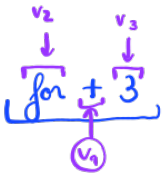
```
stats(stat) ::= <<
<if(stat)><stat; separator="\n"><endif>
>>

module(name,stat) ::= <<
#include <stdio.h>
#include <math.h>

int main()
{
   <stats(stat)>
}
>>
```

- Se a geração de código for guiada pela árvore sintáctica (como normalmente acontece), então os padrões de código a ser gerados devem ter em conta as definições gramaticais de cada símbolo, permitindo a sua aplicação modular em cada contexto.

Geração de código para expressões



Geração de código para expressões

- Para ilustrar a simplicidade e poder de abstração do *String Template* vamos estudar o problema de geração de código para expressões.
- Para resolver este problema de uma forma modular, podemos utilizar a seguinte estratégia:
  1. considerar que qualquer expressão tem a si associada uma variável (na linguagem destino) com o seu valor;
  2. para além dessa associação, podemos também associar a cada expressão um `ST` (`stats`) com as instruções que atribuem o valor adequado à variável.
- Como habitual, para fazer estas associações podemos definir atributos na gramática, fazer uso do resultados das funções de um *Visitor* ou utilizar a classe `ParseTreeProperty`
- Desta forma, podemos fácil e de uma forma modular, gerar código para qualquer tipo de expressão.

Geração de código para expressões (2)

- Padrões para expressões (para `Java`) podem ser:

```
typeValue ::= [
   "integer":"int", "real":"double",
   "boolean":"boolean", default:"void"
]

init(value) ::= "<if(value)> = <value><endif>"
decl(type,var,value) ::=
   "<typeValue.(type)> <var><init(value)>;"

binaryExpression(type,var,e1,op,e2) ::=
   "<decl(type,var,[e1,\" \",op,\" \",e2])>"
```

- Para C apenas seria necessário mudar o padrão `typeValue`:

```
typeValue ::= [
   "integer":"int", "real":"double",
   "boolean":"int", default:"void"
]
```

Exemplo: compilador simples

tenho foto 09/04/24

Código de triplo endereço

Código de triplo endereço

- O padrão para expressões é um exemplo duma representação muito utilizada para geração de código baixo nível (em geral, intermédio, e não final), designada por codificação de triplo endereço (TAC).
- Esta designação tem origem nas instruções com a forma: $x = y$ **op** z
- No entanto, para além desta operação típica de expressões binárias, esta codificação contém outras instruções (ex: operações unárias e de controlo de fluxo).
- No máximo, cada instrução tem três operandos (i.e. três variáveis ou endereços de memória).
- Tipicamente, cada instrução TAC realiza uma operação elementar (e já com alguma proximidade com as linguagens de baixo nível dos sistemas computacionais).

TAC: Exemplo de expressões binárias

- Por exemplo a expressão $a + b * (c + d)$ pode ser transformada na sequência TAC:

```
t8 = d;
t7 = c;
t6 = t7+t8;
t5 = t6;
t4 = b;
t3 = t4*t5;
t2 = a;
t1 = t2+t3;
```

- Esta sequência – embora fazendo uso desregrado no número de registos (o que, num compilador gerador de código máquina, é resolvido numa fase posterior de optimização) – é codificável em linguagens de baixo nível.

TAC: Endereços e instruções

- Nesta codificação, um endereço pode ser:
  - Um nome do código fonte (variável, ou endereço de memória);
  - Uma constante (i.e. um valor literal);
  - Um nome temporário (variável , ou endereço de memória), criado na decomposição TAC.
- As instruções típicas do TAC são:
  1. Atribuições de valor de operação binária: $x = y$ **op** z
  2. Atribuições de valor de operação unária: $x =$ **op** y
  3. Instruções de cópia: $x = y$
  4. Saltos incondicionais e etiquetas: **goto** L e **label** L :
  5. Saltos condicionais: **if** x **goto** L ou **ifFalse** x **goto** L
  6. Saltos condicionais com operador relacional: **if** x **relop** y **goto** L (o operador pode ser de igualdade ou ordem)
  7. Invocações de procedimentos (**param** x_1 $\cdots$ **param** x_n ; **call** p, n ; $y =$ **call** p, n ; **return** y)
  8. Instruções com arrays (i.e. o operador é os parêntesis rectos, e um dos operandos é o índice inteiro).
  9. Instruções com ponteiros para memória (como em C)

Controlo de fluxo

- As instruções de controlo de fluxo são as instruções condicionais e os ciclos.
- Em linguagens de baixo nível muitas vezes estas instruções não existem.
- O que existe em alternativa é a possibilidade de dar "saltos" dentro do código recorrendo a endereços (*labels*) e a instruções de salto (*goto*, ...).

```
if (cond) {
    A;
}
else {
    B;
}
```

```
ifFalse cond goto l1
A
goto l2
label l1:
B
label l2:
```

Controlo de fluxo (2)

- De forma similar podemos gerar código para ciclos:

```
while(cond) {
  A;
}
```

```
label l1:
ifFalse cond goto l2
A
goto l1
label l2:
```

Funções

- A geração de código para funções pode ser feita recorrendo a uma estratégia tipo “macro”, ou implementando módulos algorítmicos separados.
- Neste último caso, é necessária a definição de um bloco algorítmico separado, assim como implementar a passagem de argumentos/resultado para/de a função.
- A passagem de argumentos pode seguir diferentes estratégias: passagem por valor, passagem por referência de variáveis, passagem por referência de objectos/registos.
- Para termos implementações recursivas é necessário que se definam novas variáveis em cada invocação da função.
- A estrutura de dados que nos permite fazer isso de uma forma muito eficiente e simples é a pilha de execução.
- Esta pilha armazena os argumentos, variáveis locais à função e o resultado da função (permitindo ao código que invoca a função não só passar os argumentos à função como ir buscar o seu resultado).

Funções (2)

- Geralmente as arquitecturas de linguagens de baixo nível (CPU's) têm instruções específicas para lidar com esta estrutura de dados.
- Vamos exemplificar esse procedimento:

```
// use:
... f(x,y);
...
// define:
int f(int a, int b) {
   A;
   return r;
}
```

```
// use:
push 0 // result
push x
push y
call f,2
pop r // result
...
// define:
label f:
pop b
pop a
pop r
store stack-position
A
// reset stack to stack-position
restore stack-position
push r
return
```