Tema 6

Síntese

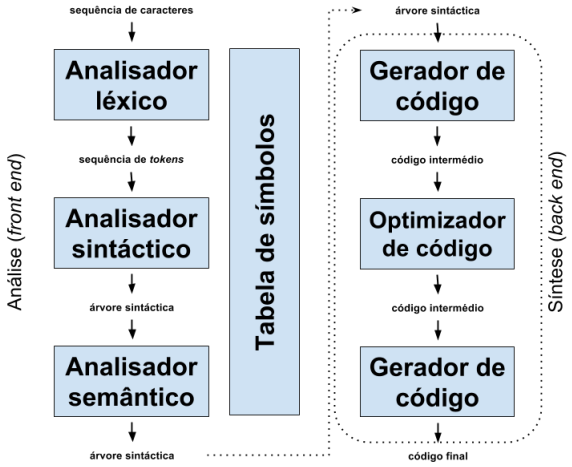
Geração de código e gestão de erros

Linguagens Formais e Autómatos, 2º semestre 2017-2018

Miguel Oliveira e Silva, DETI, Universidade de Aveiro

Conteúdo

1 Síntese: geração de código



- Podemos definir o objectivo de um compilador como sendo *traduzir* o código fonte de uma linguagem para outra linguagem.
- A geração do código da na linguagem destino pode ser feita por diferentes fases (como expresso na figura), 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.

1.1 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 UC vamos, alternativamente, discutir a geração de código numa perspectiva mais abrangente, incluindo a geração de código em linguagens de alto nível.
- 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.)

1.2 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 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.
- Como a larguíssima maioria das linguagens destino são textuais, esses padrões de geração de código consistem em padrões de geração de texto.
- Assim sendo, poderíamos delegar esse problema no tipo de dados `String` (em `Java`), 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.

2 *String Template*

- A biblioteca *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 necessário descarregar o pacote `ST-4.0.8.jar` e colocá-lo no *CLASSPATH* (ou melhor ainda no directório onde foi colocado o `jar` do `antlr4`).
- 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 o padrão de geração de código, está separado do preenchimento dos "buracos" definidos, e da geração de código final.

- 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 em grupos de uma 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);
```

- Podemos também colocar estes padrões em ficheiros:

```
// 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);
```

- Uma melhor opção é optar por ficheiros com grupos de padrões:

```
// file templates.stg

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

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

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");
...
```

2.1 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.

- 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)>
}
>>
```

2.2 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` com as instruções que atribuem o valor adequado à variável.
- Como habitual, para fazer estas associações podemos utilizar a classe `ParseTreeProperty`, definir atributos na gramática ou ainda fazer uso do resultados das funções de um *Visitor*.
- Desta forma, podemos fácil e de uma forma modular, gerar código para qualquer tipo de expressão.
- Padrões para expressões (para `Java`) podem ser:

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

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

operators ::= [
   "*":"*", "/":"/", "//":"/", "\\\\":"%", "+":"+",
   "-":"-", "=":"==", "/=":"!=", default:"null"
]

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

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

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

3 Síntese: geração de código intermédio

3.1 Código de triplo endereço

- Uma representação muito utilizada para geração de código (em geral, intermédio, e não final), é a 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).

3.2 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.

3.3 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)

3.4 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:
```

- De forma similar podemos gerar código para ciclos:

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

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

3.5 Funções

- A geração de código para funções pode ser feita recorrendo a uma estratégia tipo “macro” (i.e. na invocação da funções é colocado o código que implementa a função), ou implementando mesmo esses módulos algorítmicos.
- Neste último caso (que é vantajoso, já que por exemplo permite a recursividade) é necessário poder definir uma bloco algorítmico separado, e permitir a passagem de argumentos/resultado para/de a função.
- A passagem de argumentos pode seguir diferentes políticas (que tê de ser escolhidas): 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).

- Geralmente as arquitecturas de linguagens de baixo nível (CPU’s) têm instruções específicas para lidar com esta estrutura de dados.
- Vamos exemplificar esse procedimento:Este código apenas ilustra a ideia. Para uma análise mais detalhada devem consultar a temática de arquitectura de computadores *frame-pointer*.

```
// 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
```

4 ANTLR4: gestão de erros

4.1 ANTLR4: relatar erros

- Por omissão o ANTLR4 faz uma gestão de erros automática que em geral responde bem às necessidades.
- No entanto, por vezes é necessário ter algum controlo sobre o processo.
- No que diz respeito à apresentação de erros, por omissão o ANTLR4 formata e envia essa informação para a saída *standard* da consola.

- Esse comportamento pode ser redefinido com a interface `ANTLRErrorListener`.
- Como o nome indica, o padrão de software utilizado é o de um *listener*, e tal como nos temos habituado em ANTLR existe uma classe base (com os métodos todos implementados sem código): `BaseErrorListener`
- O método `syntaxError` é invocado pelo ANTLR na presença de erros e aplica-se ao analisador sintáctico.

Relatar erros: exemplo 1

- Como exemplo podemos definir um *listener* que escreva também a pilha de regras do parser que estão activas.

```
import org.antlr.v4.runtime.*;
import java.util.List;
import java.util.Collections;

public class VerboseErrorListener extends BaseErrorListener {
   @Override public void syntaxError(Recognizer<?, ?> recognizer,
         Object offendingSymbol,
         int line, int charPositionInLine,
         String msg,
         RecognitionException e)
   {
      Parser p = ((Parser)recognizer);
      List<String> stack = p.getRuleInvocationStack();
      Collections.reverse(stack);
      System.err.println("rule stack: "+stack);
      System.err.println("line "+line+":"+charPositionInLine+
            " at "+offendingSymbol+": "+msg);
   }
}
```

- Podemos agora desactivar os *listeners* definidos por omissão e activar o novo *listener*:

```
...
AParser parser = new AParser(tokens);
parser.removeErrorListeners(); // remove ConsoleErrorListener
parser.addErrorListener(new VerboseErrorListener()); // add ours
parser.mainRule(); // parse as usual
...
```

- Note que podemos detectar a existência de erros após a análise sintáctica (já feito pelo `antlr4-main`):

```
...
parser.mainRule(); // parse as usual
if (parser.getNumberOfSyntaxErrors() > 0) {
   ...
}
```

- Podemos também passar todos os erros de reconhecimento de *tokens* para a análise sintáctica:

```
grammar AParser;
...
/**
Last rule in grammar to ensure all errors are passed to the parser
 */
ERROR: . ;
```

Relatar erros: exemplo 2

- Outro *listener* que escreva os erros numa janela gráfica:

```
import org.antlr.v4.runtime.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;

public class DialogErrorListener extends BaseErrorListener {
   @Override public void syntaxError(Recognizer<?, ?> recognizer,
         Object offendingSymbol, int line, int charPositionInLine,
         String msg, RecognitionException e)
   {
      Parser p = ((Parser)recognizer);
      List<String> stack = p.getRuleInvocationStack();
      Collections.reverse(stack);
      StringBuilder buf = new StringBuilder();
      buf.append("rule stack: "+stack+" ");
      buf.append("line "+line+":"+charPositionInLine+" at "+
            offendingSymbol+": "+msg);
      JDialog dialog = new JDialog();
      Container contentPane = dialog.getContentPane();
      contentPane.add(new JLabel(buf.toString()));
      contentPane.setBackground(Color.white);
      dialog.setTitle("Syntax error");
      dialog.pack();
      dialog.setLocationRelativeTo(null);
      dialog.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
      dialog.setVisible(true);
   }
}
```

4.2 ANTLR4: recuperar de erros

- A recuperação de erros é a operação que permite que o analisador sintáctico continue a processar a entrada depois de detectar um erro, por forma a se poder detectar mais do que um erro em cada compilação.
- Por omissão o ANTLR4 faz uma recuperação automática de erros que funciona razoavelmente bem.
- As estratégias seguidas pela ANTLR4 para esse fim são as seguintes:
  - inserção de *token*;
  - remoção de *token*;
  - ignorar *tokens* até sincronizar novamente a gramática com o fim da regra actual.
- (Não vamos detalhar mais este ponto.)

4.3 ANTLR4: alterar estratégia de gestão de erros

- Por omissão a estratégia de gestão de erros do ANTLR4 tenta recuperar a análise sintáctica utilizando uma combinação das estratégias atrás sumariamente apresentadas.
- A interface `ANTLRErrorStrategy` permite a definição de novas estratégias, existindo duas implementações na biblioteca de suporte: `DefaultErrorStrategy` e `BailErrorStrategy`.
- A estratégia definida em `BailErrorStrategy` assenta na terminação imediata da análise sintáctica quando surge o primeiro erro.
- A documentação sobre como lidar com este problema pode ser encontrada na classe `Parser`.
- Para definir uma nova estratégia de gestão de erros utiliza-se o seguinte código:

```
...
AParser parser = new AParser(tokens);
parser.setErrorHandler(new BailErrorStrategy());
...
```