Theme 4

Synthesis

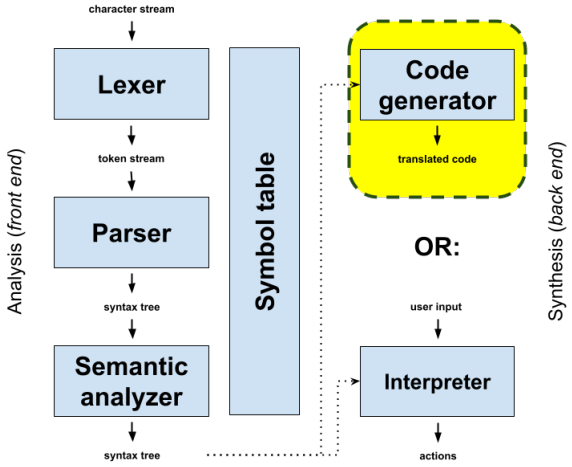
Code Generation

Compiladores, 2º semestre 2022-2023

Miguel Oliveira e Silva, Artur Pereira, DETI, University of Aveiro

Contents

1 Synthesis: code generation



- We can define the purpose of a compiler as being *translate* the source code of one language into another language.

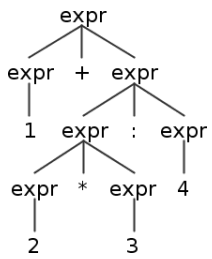
source language → `Compiler` → target language

- The generation of code for the target language can be done in different phases (which may include optimization phases), but we will cover only a single phase.
- The general strategy is to identify *code generation patterns*, and after semantic analysis go through the syntactic tree again (but already with the very important guarantee of the absence of syntactic and semantic errors) generating the target code at the appropriate points.

Example: Calculator

- Source code:

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



- A possible compilation for 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);
   }
}
```

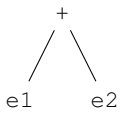
1.1 Machine code generation

- Traditionally, teaching language processors has tended to give primacy to code generation low-level (machine language, or *assembly*).

- The vast majority of the literature maintains this focus.
- However, from a practical point of view there will be few programmers who, making use of tools to generate language processors, need or desire this type of code generation.
- In this course we will, alternatively, discuss code generation in a broader perspective, including code generation in high-level languages.
- With regard to code generation in low-level languages, it is necessary to have a robust knowledge of computer architecture and deal with the following aspects:
  - Information representation and format (format for integers, real numbers, structures, *array*, etc.);
  - Memory management and addressing;
  - Function implementation (argument and result passing, support for call stack recursion and *frame pointers*);
  - Processor register allocation.
- (Consult the recommended bibliography to study this type of code generation.)

1.2 Code generation

- Regardless of the target language level, a possible strategy to solve this problem is to identify unambiguous *code generation patterns* associated with each *grammatical element of the language*.
- For this purpose, it is necessary to define the code generation context for each element (for example, generating instructions in the target language, or assigning a value to a variable), and then ensuring that it is compatible with all uses of the element.



$$\begin{array}{l} \cdots(e_1) \\ v_1 = e_1 \\ \cdots(e_2) \\ v_2 = e_2 \\ \\ v_+ = v_1 + v_2 \end{array}$$

- Since the vast majority of target languages are textual, these code generation patterns consist of text generation patterns.
- Therefore, in `Java`, we could delegate this problem to the data type `String`, `StringBuilder`, or even writing text directly to a file (or to *standard output*).
- However, there too the `ANTLR4` environment provides a more structured, systematic and modular help to deal with this problem.

2 *String Template*

- The *String Template* library provides a structured solution for generating textual code.
- Software and documentation can be found at `http://www.stringtemplate.org`
- Only needs the `ST-4.?.jar` package to be used (the installation of antlr4 already included this package).
- Let's look at a simple example:

```
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());
```

- Even though this is a very simple example, we can already see that the definition of the text pattern is separate from filling in the "holes" (attributes or expressions) defined, and generating the final text.

- We can thus delegate to different parts of the code generator, the definition of the patterns (which now belong to the context of the code element to be generated), the filling of defined “holes”, and the generation of the final code text .
- Patterns are blocks of text and expressions.
- The text corresponds to literal target code, and the expressions are “holes” that can be filled in with whatever text you want.
- Syntactically, expressions are identifiers delimited by `<expr>` (or by `$`).

```
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

- We can also group the patterns into a kind of functions (module `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);
```

- We can also put each function in a file:

```
// 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);
```

- A better option is to opt for modular files containing groups of functions/patterns:

```
// 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

- In this module we can also define dictionaries (associative arrays).

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

- In defining patterns we can use a conditional statement that only applies the pattern if attribute is added:

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

- The `separator` field indicates that in each `add` in `stat`, whether this separator will be used (in this case, a line break).

String Template: Functions

- We can also define patterns using other patterns (as if they were functions).

```
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: lists

- There is also the possibility to use lists to concatenate text and pattern arguments:

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

- or:

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

- For more information about the possibilities of this library, consult the documentation exists at: `http://www.stringtemplate.org`.

2.1 Code generation: common patterns

- Modular code generation requires a uniform context that allows the inclusion of any combination of code to be generated.
- In its simplest form, the common pattern can simply be a sequence of instructions.

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

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

- With this pattern, we can insert whatever sequence of instructions we want in the "hole" `stat`.
- Of course, for more complex code generation we might consider including holes for class members, multiple classes, or even multiple files.

- For the `C` language, we would have the following pattern for a compilation module:

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

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

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

- If code generation is parse tree driven (as it usually is), then the code patterns to be generated must take into account the grammatical definitions of each symbol, allowing its modular application in each context.

2.2 Code generation for expressions

- To illustrate the simplicity and power of abstraction of *String Template* let's study the generation problem code for expressions.
- To solve this problem in a modular way, we can use the following strategy:
  1. consider that any expression has a variable (in the target language) associated with its value;
  2. in addition to this association, we can also associate to each expression a `ST` (`stats`) with the instructions that assign the proper value to the variable.
- As usual, to make these associations we can define attributes in the grammar, make use of the results of a *Visitor* function or use the `ParseTreeProperty` class
- In this way, we can easily and in a modular way generate code for any type of expression.

- Patterns for expressions (for `Java`) can be:

```
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])>"
```

- For C it would only be necessary to change the default `typeValue`:

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

3 Synthesis: intermediate code generation

3.1 Triple address code

- The pattern for expressions is an example of a representation often used for code generation low level (generally intermediate, not final), called *triple address encoding* (TAC).
- This designation comes from instructions of the form: $x = y$ **op** z
- However, in addition to this operation typical of binary expressions, this encoding contains other instructions (eg unary operations and flow control).
- At most, each instruction has three operands (i.e. three variables or memory addresses).
- Typically, each TAC instruction performs an elementary operation (and already with some proximity to the low-level languages of computer systems).

3.2 TAC: Example of binary expressions

- For example the expression $a + b * (c + d)$ can be transformed into the TAC sequence:

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

- This sequence – although making unruly use of the number of registers (which, in a machine code generator compiler, is resolved in a later optimization phase) – is codable in low-level languages.

3.3 TAC: Addresses and directions

- In this encoding, an address can be:
  - A source code name (variable, or memory address);
  - A constant (i.e. a literal value);
  - A temporary name (variable, or memory address), created in TAC decomposition.
- Typical TAC instructions are:
  1. Binary operation value assignments: $x = y$ **op** z
  2. Unary operation value assignments: $x =$ **op** y
  3. Copy instructions: $x = y$
  4. Unconditional jumps and labels: **goto** L and **label** L :
  5. Conditional jumps: **if** x **goto** L or **ifFalse** x **goto** L
  6. Conditional jumps with relational operator: **if** x **relop** y **goto** L (operator can be equality or order)
  7. Procedure invocations (**param** x_1 $\cdots$ **param** x_n ; **call** p,n ; $y =$ **call** p,n ; **return** y)
  8. Array instructions (i.e. operator is the square brackets, and one of the operands is the integer index).
  9. Instructions with pointers to memory (as in C)

3.4 Flow control

- The flow control instructions are the conditional instructions and the loops.
- In low-level languages these instructions often do not exist.
- What exists alternatively is the possibility of making "jumps" inside the code using addresses (*labels*) and jump instructions (*goto*, ...).

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

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

- In a similar way we can generate code for loops:

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

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

3.5 Functions

- Code generation for functions can be done using a "macro"-like strategy (i.e. in invocation the code that implements the function is placed inside the functions), or by implementing separate algorithmic modules.
- In the latter case (which, among other things, allows recursion), it is necessary to define a separate algorithmic block, as well as implementing arguments/result passing to/from the function.
- Passing arguments can follow different strategies: passing by value, passing by reference of variables, passage by reference of objects/records.
- To have recursive implementations it is necessary to define new variables in each function invocation.
- The data structure that allows us to do this in a very efficient and simple way is the execution stack.
- This stack stores the arguments, variables local to the function, and the result of the function (allowing code to that invokes the function not only passing the arguments to the function but fetching its result).
- Often low-level language architectures (CPU's) have specific instructions to deal with this data structure.
- Let's exemplify this procedure:This code just illustrates the idea. For a more detailed analysis, consult the theme of computer architecture *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
```