

Software Language Engineering Code generation

Tijs van der Storm



Centrum Wiskunde & Informatica



university of
groningen

Recap

- Grammar -> Parser -> Parse Tree -> AST
- Name resolution: recover referential structure
- Checking: find errors not captured by syntax
- Today:
 - semantics
 - compilation/code generation

Transformation

- Translation
- Restructuring
- Generation
- Optimization
- ...

Compilation

- Translation from high-level to low-level
(= lowering the level of abstraction)
 - Java -> JVM byte code
 - C -> x86 machine code
 - JVM byte code -> x86 machine code
 - QL -> HTML + Javascript
- So not, e.g., Java to C# translation

Compiler pipe line

- Simplification:
 - desugar: $\text{unless } (x) \ S \rightarrow \text{if } (!x) \ S$
 - “lowering”: $\text{if } (x) \ S \rightarrow \text{if } (x) \ S \text{ else } ;$
- Source level optimization
 - e.g. $\text{if } (\text{true}) \ S \rightarrow S$, $0 * x \rightarrow 0$, etc.
- Intermediate representation
 - example: SSA

Source code generation

- AST based: transform trees, format at the end
- String-based: generate source code directly
 - e.g. using template frameworks

AST based generation

- + type-safe (“syntax correct”)
- + allows post-processing
- - cumbersome, big AST types
- - not WYSIWYG
- - need pretty printer

Template-based generation

- + quick and dirty, no grammar/AST/formatter required
- + wysiwyg
- - not syntax-safe, no IDE support
- - post-processing requires parsing

Byte-code generation

- + no need for target compiler, so fast compilation
- + very expressive
- - low-level: it's not source code after all
- - requires knowledge of VM infrastructures

Some challenges in code generation

- Origin tracking: how to trace errors and debug info back to original language?
- Modular source code generation is hardly possible.
- Name capture
 - (see, e.g., Erdweg et al., ECOOP'13)

```

state opened
  close => closed
end

```

```

state closed
  open => opened
  lock => current
end

```

```

state current
  unlock => closed
end

```

(a) Input

```

str controller2run(Controller ctl) =
  "void run(Scanner input, Writer output) {
    ' int current = <ctl.states[0]>;
    ' while (true) {
    '   String tk = input.nextLine();
    '   <for (s ← ctl.states) {>
    '     <state2if(s)>
    '   <}>
    ' }
  }";

```

```

str state2if(State s) =
  "if (current == <s.name>) {
    ' <for (transition(e, s2) ← s.transitions) {>
    '   if (<e>(tk)) current = <s2>;
    '   <}>
    '   continue;
  }";

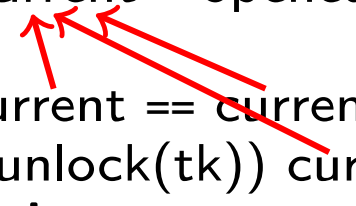
```

(b) Excerpt of state machine compiler

```

static final int current = 2;
void run(...) {
  int current = opened;
  ...
  if (current == current) {
    if (unlock(tk)) current = closed;
    continue;
  }
  ...
}

```

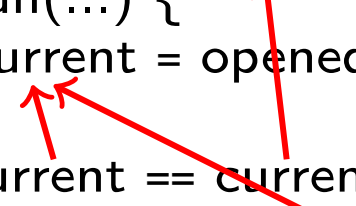


(c) Incorrect output

```

static final int current0 = 2;
void run(...) {
  int current = opened;
  ...
  if (current == current0) {
    if (unlock(tk)) current = closed;
    continue;
  }
  ...
}

```



(d) Repaired output

Code generation in Rascal

- Built-in string templates
- AST types for Java, Javascript, HTML, and others
- Flybytes: generate JVM bytecode directly
- Wasm: web assembly (BSc project)

State machines

