Programming Languages

Metaprogramming

Dr. Michael Petter
Winter 2017/18
“Let’s write a program, which writes a program“

Learning outcomes

1. Compilers and Compiler Tools
2. Preprocessors for syntax rewriting
3. Reflection and Metaclasses
4. Metaobject Protocol
5. Macros
Motivation

- Aspect Oriented Programming establishes programmatic refinement of program code
- How about establishing support for program refinement in the language concept itself?
- Treat program *code as data*

⇒ Metaprogramming
Motivation

- Aspect Oriented Programming establishes programmatic refinement of program code
- How about establishing support for program refinement in the language concept itself?
- Treat program $\textit{code as data}$

Metaprogramming

- Treat programs as data
- Read, analyse or transform (other) programs
- Program modifies itself during runtime
Codegeneration Tools
Codegeneration Tools

Compiler Construction

In Compiler Construction, there are a lot of codegeneration tools, that compile DSLs to target source code. Common examples are lex and bison.

Example: lex:

lex generates a table lookup based implementation of a finite automaton corresponding to the specified disjunction of regular expressions.

```c
#include <stdio.h>

%{
    #include <stdio.h>
%
%
    /* Lexical Patterns */
    [0-9]+ { printf("integer: %s\n", yytext); }
    .|\n    { /* ignore */ }
%
    int main(void) {
        yylex();
        return 0;
    }
```
Codegeneration via Preprocessor
String Rewriting Systems

A Text Rewriting System provides a set of grammar-like rules (→ Macros) which are meant to be applied to the target text.

Example: C Preprocessor (CPP)

```c
#define min(X,Y) (( X < Y )? (X) : (Y))
x = min(5,x); // (( 5 < x )? (5) : (x))
x = min(++x,y+5); // (( ++x < y+5 )? (++x) : (y+5))
```
### String Rewriting Systems

A Text Rewriting System provides a set of grammar-like rules (→ *Macros*) which are meant to be applied to the target text.

**Example:** *C Preprocessor (CPP)*

```
#define min(X,Y) (( X < Y )? (X) : (Y))
x = min(5,x);  // (( 5 < x )? (5) : (x))
x = min(++x,y+5); // (( ++x < y+5 )? (++x) : (y+5))
```

⚠️ **Nesting, Precedence, Binding, Side effects, Recursion, ...**

- Parts of Macro parameters can bind to context operators depending on the precedence and binding behaviour
- Side effects are recomputed for every occurrence of the Macro parameter
- Any (indirect) recursive replacement stops the rewriting process
- Name spaces are not separated, identifiers duplicated
Example application: Language constructs [3]:

```c
ATOMIC {
    i--; 
    i++; 
}
```

```c
#define ATOMIC  \
acquire(&globallock);\ 
{ /* user code */ } \ 
release(&globallock);
```

⚠️ How can we bind the block, following the ATOMIC to the user code fragment? Particularly in a situation like this?

```c
if (i>0)  
    ATOMIC {
        i--; 
        i++; 
    }
```
**Compiletime-Codegeneration**

Prepend code to usercode

```c
if (1)
    /* prepended code */
    goto body;
el
    body:
    { /* block following the expanded macro */ }
```

Append code to usercode

```c
if (1)
    goto body;
else
    while (1)
        if (1) {
            /* appended code */
            break;
        }
    else body:
    { /* block following the expanded macro */ }
```
All in one

```c
if (1) {
    /* prepended code */
    goto body;
} else
    while (1)
        if (1) {
            /* appended code */
            break;
        }
    else body:
    { /* block following the expanded macro */ }
```
#define concat_(a, b) a##b
#define label(prefix, lnum) concat_(prefix, lnum)
#define ATOMIC
if (1) {
    acquire(&globallock);
    goto label(body, __LINE__);  
} else
    while (1)
    if (1) {
        release(&globallock);
        break;
    }
else
    label(body, __LINE__);

⚠️ **Reusability**

labels have to be created dynamically in order for the macro to be reusable (→ __LINE__)
Homoiconic Metaprogramming
**Homoiconicity**

In a homoiconic language, the primary representation of programs is also a data structure in a primitive type of the language itself.

- data is code
- code is data

- Metaclasses and Metaobject Protocol
- (Hygienic) Macros
Reflection
Reflective Metaprogramming

Type introspection

A language with *Type introspection* enables to examine the type of an object at runtime.

**Example:** Java `instanceof`

```java
public boolean equals(Object o){
    if (!(o instanceof Natural)) return false;
    return ((Natural)o).value == this.value;
}
```
Reflective Metaprogramming

Metaclasses (→ code is data)

Example: Java Reflection / Metaclass `java.lang.Class`

```java
static void fun(String param){
    Object incognito = Class.forName(param).newInstance();
    Class meta = incognito.getClass(); // obtain Metaobject
    Field[] fields = meta.getDeclaredFields();
    for(Field f : fields){
        Class t = f.getType();
        Object v = f.get(o);
        if(t == boolean.class && Boolean.FALSE.equals(v))
            // found default value
        else if(t.isPrimitive() && ((Number) v).doubleValue() == 0)
            // found default value
        else if(!t.isPrimitive() && v == null)
            // found default value
    }
}
```
Metaobject Protocol
Metaobject Protocol

Metaobject Protocol (MOP \[^{[1]}\])

Example: Lisp’s CLOS metaobject protocol

... offers an interface to manipulate the underlying implementation of CLOS to adapt the system to the programmer’s liking in aspects of

- creation of classes and objects
- creation of new properties and methods
- causing inheritance relations between classes
- creation generic method definitions
- creation of method implementations
- creation of specializers (→ overwriting, multimethods)
- configuration of standard method combination (→ before, after, around, call-next-method)
- simple or custom method combinators (→ +, append, max,...)
- addition of documentation
Hygienic Macros
Clojure programs are represented after parsing in form of symbolic expressions (*S-Expressions*), consisting of nested trees:

**S-Expressions**

S-Expressions are either

- an atom
- an expression of the form \((x \cdot y)\) with \(x, y\) being S-Expressions

**Remark:** Established shortcut notation for lists:

\[
(x_1 \ x_2 \ x_3) \equiv (x_1 \cdot (x_2 \cdot (x_3 \cdot ())))
\]
Special Forms

Special forms differ in the way that they are interpreted by the clojure runtime from the standard evaluation rules.

Language Implementation Idea: reduce every expression to special forms:

(def symbol doc? init?)
(do expr*)
(if test then else?)
(let [binding*] expr*)
(eval form) ; evaluates the datastructure form
(quote form) ; yields the unevaluated form
(var symbol)
(fn name? ([params*] expr*)+)
(loop [binding*] expr*)
(recur expr*) ; rebinds and jumps to loop or fn
;...
Macros are configurable syntax/parse tree transformations.

Language Implementation Idea: define advanced language features in macros, based very few *special forms* or other macros.

Example: While loop:

```
(macroexpand '(while a b))
; => (loop* [] (clojure.core/when a b (recur)))
```

```
(macroexpand '(when a b))
; => (if a (do b))
```
Macros can be written by the programmer in form of S-Expressions:

```lisp
(defmacro infix
  "converting infix to prefix"
  [infixed]
  (list (second infixed) (first infixed) (last infixed)))
```

...producing

```lisp
(infix (1 + 1))
; => 2
(macroexpand '(infix (a + b)))
; => (+ a b)
```

⚠️ **Quoting**

Macros and functions are directly interpreted, if not *quoted* via

```lisp
(quote keyword) ; or equivalently:
'keyword
; => keyword
```
Homoiconic Runtime-Metaprogramming

Why bother?

Metaprogramming

Introduction

```
(defmacro fac1 [n]
  (if (= n 0)
    1
    (list ' * n (list 'fac1 (- n 1)))))
```

```
(defn fac2 [n]
  (if (= n 0)
    1
    (* n (fac2 (- n 1)))))
```

```
(fac1 4)
; => 24
```

```
(fac2 4)
; => 24
```

...produces

```
(macroexpand '(fac1 4))
; => (* 4 (fac1 3))
```

```
(macroexpand-all '(fac1 4))
; => (* 4 (* 3 (* 2 (* 1 1)))))
```
Macros vs. Functions

- Macros as static AST Transformations, vs. Functions as runtime control flow manipulations
- Macros replicate parameter forms, vs. Functions evaluate parameters once
- Macro parameters are uninterpreted, not necessarily valid expressions, vs. Functions parameters need to be valid expressions
Homoiconic Runtime-Metaprogramming

⚠ Macro Hygiene

*Shadowing* of variables may be an issue in macros, and can be avoided by generated symbols!

```lisp
(def variable 42)
(macro mac [&stufftodo] ~(let [variable 4711] ~@stufftodo))
(mac (println variable))
; => can't let qualified name: variable
```

```lisp
(macro mac [&stufftodo] ~(let [variable# 4711] ~@stufftodo))
```

⇝ Symbol generation to avoid namespace collisions!
Richard P. Gabriel. Gregor kiczales, jim des rivières, and daniel g. bobrow, the art of the metaobject protocol. 
URL: https://doi.org/10.1016/0004-3702(93)90073-K, 
doi:10.1016/0004-3702(93)90073-K.

Daniel Higginbotham. 
*Clojure for the Brave and True: Learn the Ultimate Language and Become a Better Programmer.* 
URL: https://www.braveclojure.com/clojure-for-the-brave-and-true/.

Simon Tatham. 
Metaprogramming custom control structures in C. 
[Online; accessed 07-Feb-2018].