TDDA69 Data and Program Structure
Macro

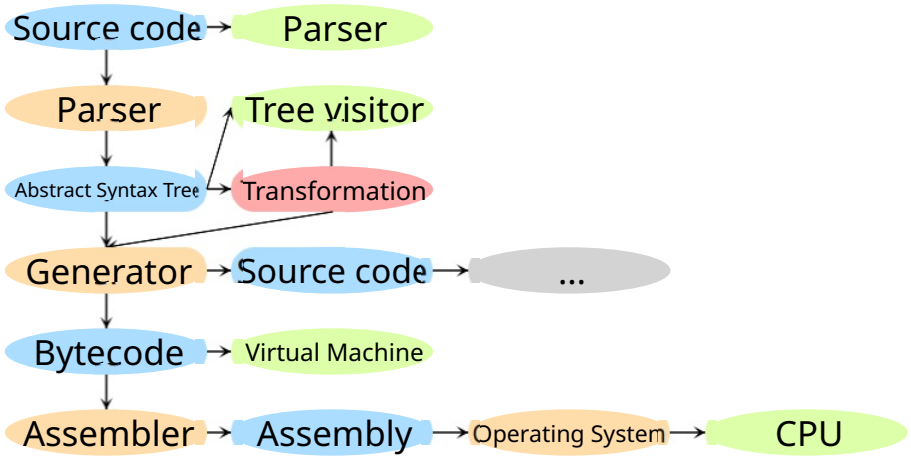
Cyrille Berger

LINKÖPING UNIVERSITY

Lectures

1 Introduction
2 Concepts and models of programming languages
3 Declarative Computation Model
4 Declarative Programming Techniques
5 Declarative Computation Implementation
6 Declarative Concurrency
7 Message Passing Concurrency
8 Explicit State and Imperative Model
9 Imperative Programming Techniques
10 Imperative Programming Implementation
11 Shared-State Concurrency
12 Relational Programming
13 Specialized Computation Models
14 **Macro**
15 Running natively and JIT
16 Garbage Collection
17 Summary

How is a program interpreted?



Lecture content

- Macros
  - Decorators
  - Syntactic Macros
  - Macros in KL
    - Implementation
    - Use cases

Macros

Macros

- A *macro* is a rule or function that map an input to an output.
- Example: C preprocessor, decorators in Python, sweetjs, lisp macro...

Why macro ?

- All you need for OO is state+closure:

```
function Person(self, protected, name)
{
  define self.name = name;
  cell priv = {
    define self.say = function say(msg)
    {
      std.print(priv.name + ' says: ' + msg);
    };
  };
  clear(priv);
}
cell bob = new(Person, 'Bob');
bob.say('state+closure are not nice syntax!');
```

- But a class-like syntax is nicer:

```
class Person {
private:
  define name;
public:
  Person(name) {
    self.name = name;
  }
  say(msg) {
    std.print(priv.name + ' says: ' + msg);
  }
}
cell bob = new(Person, 'Bob');
bob.say('Classes are sweet!');
```


C preprocessor (1/3)

- The first step of compiling a C program is:
  - to replace preprocessing keyword #include, #define, #ifdef...

```
#include <stdio.h>
#define MY_MACRO(x) (x)*(x)
...
```

  - and apply #defined macros

C preprocessor (2/3)

```c
#include <stdio.h>
#define PRINT(text)    printf(text)

int main(int _argc, const char** _argv)
{
  PRINT("Hello World!\n");
  return 0;
}
```

```
gcc -E test.c -o preprocessed.c
```


C preprocessor (3/3)

- If not careful, they lead to wrong or inconsistent behaviour:

```c
#define oldfunc(a, b) newfunc1(a);
newfunc2(b);
oldfunc(5, 6)
=> newfunc1(5); newfunc2(6);
for (i = 0; i < 5; i++) oldfunc(5, 6);
=> for (i = 0; i < 5; i++) newfunc1(5);
newfunc2(6);
#define max(a,b) (a) < (b) ? (b) : (a)
max(f(1),f(2))
=> (f(1)) < (f(2)) ? (f(2)) : (f(1))
```


Decorators

Decorators

- What about changing the behavior of a function without changing its code?

Memoization (1/2)

- Fibonacci:

```
def fib(n):
  print(n)
  if n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    return fib(n-2) + fib(n-1)
```


Memoization (2/2)

- Manually rewritten to:

```
__fib_cache = {0: 0, 1: 1}
def fib_cached(n):
    if n in __fib_cache:
        return __fib_cache[n]
    else:
        __fib_cache[n] = fib_cached(n-2)
                       + fib_cached(n-1)
        return __fib_cache[n]
```


Memoization decorator (1/2)

```
def memoization(func):
  cache = {}
  def memoized(*args):
    if args in cache:
      return cache[args]
    else:
      val = func(*args)
      cache[args] = val
      return cache[args]
  return memoized
fib = memoization(fib)
```


Memoization decorator (2/2)

The following is

```
@memoization
def fib(n):
  return n if n < 2 else fib(n-2) +
fib(n-1)
```

equivalent to

```
def fib(n):
  return n if n < 2 else fib(n-2) +
fib(n-1)
fib = memoization(fib)
```

No constants in Python

- Python does not have constants

```
print(math.pi) -> 3.141592653589793
math.pi = 2
print(math.pi) -> 2
```

- You can fake them with properties:

```
class _math:
  def PI(self):
    return 3.141592653589793
  pi = property(PI, None)

math = _math()

print(math.pi)
math.pi = 3.141592653589793
```


Constants in Python with a decorator

```
def constant(f):
    def fget(self):
      return f()
  return property(fget, None)
class _Math(object):
  @constant
  def pi():
    return 3.141592653589793
math = _Math()

print(math.pi) -> 3.141592653589793
math.pi = 2 -> AttributeError: can't set
attribute
```


Adjustable decorators

- Decorators can take arguments:

```
@decorator(args...)
def func(fargs...):
    pass
```

is equivalent to

```
decorators(args)(func)
```


Bound-checking

```
def bound_checking_decorator(min, max):
  def make_decorator(func):
    def decorator(x):
      if(x < min or x > max):
        raise ValueError()
      return func(x)
    return decorator
  return make_decorator
@bound_checking_decorator(0, float('inf'))
def fib(n):
  return n if n < 2 else fib(n-2) + fib(n-1)
```

Chain decorators

```
@bound_checking_decorator(0,
float('inf'))
@memoization
def fib(n):
  return n if n < 2 else fib(n-2)
+ fib(n-1)
```


Syntactic Macros

Syntactic macros

- C macro works at lexical level
- Syntactic macros works on AST
  - Mostly common in Lisp-like languages

Hygienic macros

- Hygienic macros are syntactic macros, where macros and user environment are seperate, so that variable cannot be captured by macros
  - Example of capture:

```
#include<stdio.h>
#define ADD(res,a,b,c)  \
{  \
  int value = a;   \
  value += b;      \
  value += c;      \
  res = value;     \
}
int main(int argc, char** argv)
{
  int value = 0;
  ADD(value, 1, 2, 3);
  printf("%i
", value);
  return 0;
}
```

  - gcc -E test2.c -o preprocessed.c

AST Manipulations

- Give the possibility to write functions that transform an expression into an other before it is evaluated
- Macro expansion can take place:
  - During
  - During

Building AST

```
import ast

node = ast.Expression(ast.BinOp(
                ast.Num(2), ast.Mult(), ast.Num(3)))

fixed = ast.fix_missing_locations(node)

codeobj = compile(fixed, '<string>', 'eval')
print(eval(codeobj))
```


Get AST of a function

```
import inspect
import ast

def func(x):
  return x+1

print(func(1))

source = inspect.getsource(func)
ast_tree = ast.parse(source)
ast_tree.body[0].body[0].value.right.n = 2

exec(compile(ast_tree, __file__, mode='exec'))

print(func(1))
```


Increment all numbers

```
def increment(f):
  source   = inspect.getsource(f)
  ast_tree = ast.parse(source)
  ast_tree.body[0].decorator_list = []
  class T(ast.NodeTransformer):
    def visit_Num(self, node):
      node.n += 1
      return node
  exec(compile(T().visit(ast_tree), __file__, mode='exec'))
  return locals()[ast_tree.body[0].name]
@increment
def func(n):
  return (2+n)*3

print(func(1))
```

Template (1/2)

```
def my_template():
  for x in range(1,10):
    __body__
  return v
@apply_template(my_template)
  def func(v):
    v = v * x
def func(v):
  for x in range(1,10):
    v = v * x
  return v
```


Template (2/2)

```
def apply_template(template):
  def t(f):
    f_ast            = ast.parse(inspect.getsource(f)).body[0]
    body_node    = f_ast.body[0]
    template_ast = ast.parse(inspect.getsource(template))
    template_ast.body[0].args = f_ast.args
    class T(ast.NodeTransformer):
      def visit_Expr(self, node):
        if(node.value.id == '__body__'):
          return body_node
        else:
          return node
    exec(compile(T().visit(template_ast), __file__, mode='exec'))
    return locals()[template_ast.body[0].name]

  return t
```


When are AST manipulations useful?

- In Python, AST manipulations are seldom used
- Functions decorators cover most of the need
- But some time macros will work better
  - If there is a need to control function return
  - To apply modifier on classes or expressions (decorators only work on functions)
  - They can be used for optimization, such as tail-call optimization
  - When calling a function, all arguments are executed
  - ...only once
  - They can save function call
- But AST manipulations are obfuscated
  - Python AST module API can change from version to version
  - No good syntax to define rules

Macros in KL

Macros in KL

- Inspired by sweet.js
  - sweet.js (http://sweetjs.org/) is a JavaScript library that brings support for hiegenic macro to JavaScript
  - The old version, not the new one
- A template matching system with generative rules

Defining rules (1/2)

- `rule <rule-definition> => <output>;`
  - Use $NAME to capture tokens
- Example:

```
rule identity($x) => $x;
identity (42);
identity ([1,2,3]);
```


Defining rules (2/2)

- `rule <rule-definition> => <output>;`
  - Use $NAME()... for a rule that repeat
- Example:

```
rule call($func $args(, $arg)...)
=> $func($args( $(,) $arg)...);
call (f1, 42);
call (f2, 1,2,3);
```


Hygiene

- Hygiene means that an invocation of a macro does not introduce or use existing bindings
- Two common problems to solve:
  - name clashes from a introducing a new
  - name clashes from using an existing

Example of hygienic syntax (1/4)

```
rule swap($x, $y)
  => [
       define tmp = $y;
       $y = $x;
       $x = tmp;
    ];

cell foo = 100;
cell bar = 100;
cell tmp = 'my other
temporary variable';

swap(foo, bar)
```

- It expands to:

```
cell foo = 100;
cell bar = 100;
cell tmp = 'my other
temporary variable';

[
  define tmp = bar;
  bar = foo;
  foo = tmp;
]
```


Example of hygienic syntax (2/4)

Lets change the macro to create a new environment:

```
rule swap($x, $y)
  => {
       define tmp = $y;
       $y = $x;
       $x = tmp;
  };

cell foo = 100;
cell bar = 100;
cell tmp = 'my other temporary
variable';

swap(foo, bar)
```

- It expands to:

```
{
  define tmp =
bar;
  bar = foo;
  foo = tmp;
}
```


Example of hygienic syntax (3/4)

But:

```
rule swap($x, $y)
  => {
       define tmp = $y;
       $y = $x;
       $x = tmp;
  };

cell foo = 100;
cell bar = 100;
cell tmp = 'my other
temporary variable';

swap(foo, tmp)
```

- It expands to:

```
{
  define tmp = tmp;
  tmp = foo;
  foo = tmp;
}
```

- Not what we wanted...
- tmp is not changed! And foo has wrong value

Example of hygienic syntax (4/4)

- Better solution is for macro engine to rename variables during expansion:

```
rule swap($x, $y)
  => [
       define tmp = $y;
       $y = $x;
       $x = tmp;
    ]
}

var foo = 100;
var tmp = 'my other temporary
variable';

swap(foo, tmp)
```

- It expands to:

```
[
  define tmp$1 = tmp;
  tmp$1 = foo;
  foo = tmp$1;
  clear(tmp$1);
]
```

- The usage of '$' is reserved to the macro engine

Implementation

Compiler

- In scheme/lisp
  - lexer -(token)→ reader -(s-expression)→ expander/parser -(AST)→
- Commonly in JavaScript
  - lexer ←(feedback/token)→ parser -(AST)→
  - Feedback is needed to distinguish division (/) from regular expressions (/x/).
- In sweetjs
  - lexer -(token)→ reader -(token-tree)→ expander/parser -(AST)→
- In KL:
  - lexer -(token)→ reader -(token-tree)→ macro engine -(token-tree)→ flatening -(token)→ parser -(AST)→

Token tree / Reader

- Tokens for '{ 42 }' are ['{', '42', '}']
- Token tree match delimiters:
  - ['{}', ['42']]
- Rules for building the tree:
  - Identifier/value token are leaf nodes
  - Nodes are grouped in statements and expressions
  - ...
- The reader convert a stream of tokens into a token-tree

Example of token tree

```
cell acc = 1;

while(acc < 10)
{
  acc = acc * 2;
};

test.case("for macro")
    .check_equal(acc, 16);
```

Macro engine

- The macro engine *visit* the tree:
  - It maintain a list of rules under consideration
  - Tree matching algorithm
  - For each node of the tree

    For each rule under consideration, check if the current node satisfy the rule, if it doesn't, discard the rule

    If a rule matches, apply the transformation, clear the list of rules and reset the macro engine

Example of rule matching

```
rule
  while ($condition)
  $body
=> {
  define loop = function() {
    cond($condition, { $body; loop() });
  };
  loop();
}
```


Use cases

Class in KI (1/2)

- All you need for OO is state+closure:

```
function Person(self, protected,
name)
{
  define self.name = name;
  cell priv = {
    define self.say = function
say(msg)
    {
      std.print(priv.name + '
says: ' + msg);
    };
  };
  clear(priv);
}
cell bob = new(Person, 'Bob');
bob.say('state+closure are not
nice syntax!');
```

- But a class-like syntax is nicer:

```
class Person {
private:
  define name;
public:
  Person(name) {
    self.name = name;
  }
  say(msg) {
    std.print(priv.name + ' says:
' + msg);
  }
}
cell bob = new(Person, 'Bob');
bob.say('Classes are sweet!');
```

Class in KI (2/2)

```
class Person
{
  cell name;
  cell age;
  Person(name, age)
  {
    self.name = name;
    self.age  = age;
  }
  function says(text)
  {
    std.print(self.name +
" says " + text);
  }
};
```

```
rule
  class $name
  {
    $variables(cell $varname;)...
    $name $cargs $cbody;
    $functions($fname $fargs $fbody ;)...
  }
  => [
  define $name = function $cargs {
    define self = {
      $variables(cell $varname;)...
      $functions(define $fname = function $fargs
{
        $fbody
      };)...
    };
    $cbody;
    return (self);
  };
]
```


contracts

- Macro can be used to implement contract:

```
function dbl(n -> Number) -> Number {
    return n + n;
}

function avg_score(person -> { name:
String, age: Number }, scores -> Number)
-> Number or Boolean {
  ...
}
```


Maros Advantages / Inconvenients

- Advantages
  - Allow to extend a programming language without needs to change the interpreter
  - Can help code reusability
  - Introduce symbolic programming techniques
- Incovenients
  - Need to be well designed and well supported by the language
  - Introduce dialects, which might make the code harder to read

Conclusion

- Decorators
- Macros and AST transformations