EECS 490 – Lecture 22

Macros and Code Generation

11/26/17

Announcements

- HW5 due Tue 12/5 at 8pm
- Project 5 due Tue 12/12 at 8pm

Metaprogramming

- Technique of writing a computer program that operates on other computer programs
- Analogous to higher-order functions, which operate on other functions
- Compilers and program analyzers are examples of metaprograms
- Metaprogramming is also a useful technique for generating code to be included as part of a program
- Examples:
  - Macros
  - Code generators
  - Template metaprogramming

Macros

- A *macro* is a rule that translates an input sequence into some replacement output sequence
- The replacement process is called *macro expansion*
- Macro expansion may be implemented as a *preprocessing* step, prior to lexical and syntactic analysis
- Expansion may also be integrated with a later analysis step such as syntax analysis
- The most widely used macro system is the C preprocessor (CPP), integrated into C and C++

Example: Swap in CPP

- We can write a *function-like* swap macro as follows:

```
#define SWAP(a, b) do {    \
    auto tmp = b;          \
    b = a;                 \
    a = tmp;               \
  } while (false)
```

Backslash at end of line denotes line continuation

Do/while allows macro to be used in a context that requires a single statement

- Example:

```
int x = 3, y = 4;
if (x < y)
  SWAP(x, y);
cout << x << " " << y << endl;
```

- Expands to:

```
do { auto tmp = y; y = x; x = tmp; } while(false);
```

Problem with Swap

- We can get unexpected results if we use a complex expression as an argument:

```
int x = 3, y = 4, z = 5;
SWAP(x < y ? x : y, z);
cout << x << " " << y << " " << z << endl;
```

```
3 4 3
```

- This expands to (line breaks and spacing added):

```
do {
  auto tmp = z;
  z = x < y ? x : y;
  x < y ? x : y = tmp;
} while (false);
```

Equivalent to
`x < y ? x : (y = tmp)`
since ? : and = have the same precedence and are right associative

Parenthesization

- Parenthesizing uses of a macro argument ensures the correct associativity and precedence

```
#define SWAP(a, b) do {    \
    auto tmp = (b);        \
    (b) = (a);             \
    (a) = tmp;             \
  } while (false)
```

- Result of expansion:

```
do {
  auto tmp = (z);
  (z) = (x < y ? x : y);
  (x < y ? x : y) = tmp;
} while (false);
```

The Hygiene Problem

- Consider the following example:

```
int main() {
  int x = 3, tmp = 4;
  SWAP(tmp, x);
  cout << x << " " << tmp << endl;
}
```

3 4

- Expansion results in (modulo spacing):

```
do { auto tmp = (x); (x) = (tmp); (tmp) = tmp; }
while (false);
```

- The argument `tmp` is *captured* by the temporary variable introduced by the expansion

- The macro is *non-hygienic*, since it doesn't distinguish between the scope of the macro and that of the argument

Scheme Macros

- Scheme macros, as defined by the R5RS spec, are hygienic
- Introduced by a `define-syntax`, `let-syntax`, or `letrec-syntax` form
- Example to translate `let` into `lambda`:

```
(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))))
```

Literals that must match in both the pattern and use

Input pattern

Output pattern

Hygienic macro variable

Like Kleene star (zero or more occurrences of previous item)

Swap in Scheme

- The following defines a swap macro:

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b)
     (let ((tmp b))
       (set! b a)
       (set! a tmp)))))
```

Scheme macros are hygienic, so tmps do not conflict

```
> (define x 3)
> (define y 4)
> (swap x y)
> x
4
> y
3
> (define tmp 5)
> (swap x tmp)
> x
5
> tmp
4
```

Recursive Macros

- Evaluation procedure for macros:
  1. Evaluate first item in list as always
  2. If it is a macro, expand the rest of the list without evaluating it first
  3. Evaluate result of expansion
- This allows recursive macros:

Base case

Recursive case

```
(define-syntax let*
  (syntax-rules ()
    ((let* () body1 body2 ...)
     (let () body1 body2 ...))
    ((let* ((name1 val1) (name2 val2) ...)
       body1 body2 ...)
     (let ((name1 val1))
       (let* ((name2 val2) ...)
         body1 body2 ...)))))
```

CPP Object-Like Macros

- *Object-like* macros are simple text replacement, replacing one sequence of text for another

- Commonly used in the past to define constants

```
#define PI 3.1415926535

int main() {
  cout << "pi = " << PI << endl;
  cout << "tau = " << PI * 2 << endl;
}
```

- Much better in modern programming to use `const` or `constexpr` variables

Break Time!

- We'll start again in five minutes.

Example: Complex Numbers

- Suppose we want to implement a complex number:

```
struct Complex {
  double real, imag;
};

ostream &operator<<(ostream &os, Complex c) {
  return os << "(" << c.real << "+" << c.imag
            << "i)";
}
```

- We would also like to provide overloaded arithmetic operators +, -, and *
- We would like to avoid repetition in writing these operators

Operator Structure

- Each operator has the following common structure:

```
Complex operator <op>(Complex a, Complex b) {
  return Complex{ <expression for real>,
                  <expression for imag> };
}
```

Uniform initialization syntax

- We can write a macro to abstract this structure:

```
#define COMPLEX_OP(op, real_part, imag_part)  \
  Complex operator op(Complex a, Complex b) { \
    return Complex{ real_part, imag_part };   \
  }
```

Using the Macro

- We can then define the operations as follows:

```
COMPLEX_OP(+, a.real+b.real, a.imag+b.imag);
COMPLEX_OP(-, a.real-b.real, a.imag-b.imag);
COMPLEX_OP(*, a.real*b.real - a.imag*b.imag,
              a.imag*b.real + a.real*b.imag);
```

- Result of expansion (line breaks and spacing added):

```
Complex operator +(Complex a, Complex b) {
  return Complex{ a.real+b.real, a.imag+b.imag };
};
Complex operator -(Complex a, Complex b) {
  return Complex{ a.real-b.real, a.imag-b.imag };
};
Complex operator *(Complex a, Complex b) {
  return Complex{ a.real*b.real - a.imag*b.imag,
                  a.imag*b.real + a.real*b.imag };
};
```

Complex/Real Operators

- We also want operations between `Complex` and `double` values

```
Complex operator <op>(<type1> a, <type2> b) {
  return <expr1> <op> <expr2>;
}
```

Operands converted to `Complex`

- Macros:

```
#define REAL_OP(op, typeA, typeB, argA, argB) \
  Complex operator op(typeA a, typeB b) {     \
    return argA op argB;                      \
  }

#define CONVERT(a) (Complex{ a, 0 })
```

Using the Macro

- Operations:

```
REAL_OP(+, Complex, double, a, CONVERT(b));
REAL_OP(+, double, Complex, CONVERT(a), b);
REAL_OP(-, Complex, double, a, CONVERT(b));
REAL_OP(-, double, Complex, CONVERT(a), b);
REAL_OP(*, Complex, double, a, CONVERT(b));
REAL_OP(*, double, Complex, CONVERT(a), b);
```

- Result:

```
Complex operator +(Complex a, double b) { return a + (Complex{ b, 0 }); };
Complex operator +(double a, Complex b) { return (Complex{ a, 0 }) + b; };
Complex operator -(Complex a, double b) { return a - (Complex{ b, 0 }); };
Complex operator -(double a, Complex b) { return (Complex{ a, 0 }) - b; };
Complex operator *(Complex a, double b) { return a * (Complex{ b, 0 }); };
Complex operator *(double a, Complex b) { return (Complex{ a, 0 }) * b; };
```

Using Complex Numbers

- We can now use complex numbers as follows:

```
int main() {
  Complex c1{ 3, 4 };
  Complex c2{ -1, 2 };
  double d = 0.5;
  cout << c1 + c2 << endl;
  cout << c1 - c2 << endl;
  cout << c1 * c2 << endl;
  cout << c1 + d << endl;
  cout << c1 - d << endl;
  cout << c1 * d << endl;
  cout << d + c1 << endl;
  cout << d - c1 << endl;
  cout << d * c1 << endl;
}
```

```
(2+6i)
(4+2i)
(-11+2i)
(3.5+4i)
(2.5+4i)
(1.5+2i)
(3.5+4i)
(-2.5+-4i)
(1.5+2i)
```

Functions on Complex Numbers

- Suppose we have an interactive application that asks the user what operation to perform and then dispatches to the appropriate function

- Supported operations:

```
Complex Complex_conjugate(Complex c) {
  return Complex{ c.real, -c.imag };
}

string Complex_polar(Complex c) {
  return "(" + to_string(sqrt(pow(c.real, 2) +
    pow(c.imag, 2))) + "," +
    to_string(atan(c.imag / c.real)) +
    ")";
}
```

Stringification and Concatenation

- Common structure of dispatch:

```
if (<input> == "<action>")
  cout << Complex_<action>(<value>) << endl;
```

- Macro definition:

```
#define ACTION(str, name, arg)                    \
  if (str == #name)                               \
    cout << Complex_ ## name(arg) << endl
```

***Stringification* operator converts to a string literal**

***Token pasting* operator concatenates tokens**

Interactive Loop

- Loop code:

```
while (cin >> s) {
  ACTION(s, conjugate, c1);
  ACTION(s, polar, c1);
}
```

- Result of expansion (line breaks and spacing added):

```
while (cin >> s) {
  if (s == "conjugate")
    cout << Complex_conjugate(c1) << endl;
  if (s == "polar")
    cout << Complex_polar(c1) << endl;
}
```

The Macro Namespace

- Macros do not have a distinct namespace
- A macro that is defined will replace all eligible tokens
- Defining a macro pollutes the global namespace
- Common strategies
  - Choose names that are unlikely to conflict with other names, e.g. by prepending the name of the library

    ```
    COMPLEXLIB_CONVERT
    COMPLEXLIB_ACTION
    ```

  - Undefine macros when they are no longer needed

    ```
    #undef CONVERT
    #undef ACTION
    ```

Code Generation

- Macros allow us to use the facilities of a language to generate code
- However, a macro system may be unavailable or otherwise unsuited for the task at hand
- We can write a *code generator* in a separate program, in the same language or a different one, in order to generate the required code
- This is also called *automatic programming*

Scheme c*r Combinations

- Scheme implementations are required to provide combinations of **car** and **cdr** up to 4 levels deep

```
(cadar x) -> (car (cdr (car x)))
```

- We can write a Python program to generate definitions for these combinations, which we can then include as a library file in an interpreter

```python
import itertools

for i in range(2, 5):
    for seq in itertools.product(('a', 'd'),
                                 repeat=i):
        print(defun(seq))
```

Levels 2 up to 4

Create sequences of 'a' and 'd' with length i

Define a combination for the sequence

Defining a Combination

- Function to construct body:

```
def cadrify(seq):
    if len(seq):
        body = "(c{0}r {1})"
        return body.format(seq[0],
                           cadrify(seq[1:]))
    return "x"
```

Call for first item in sequence

Base case

Recursive call

- Function to construct a definition:

```
def defun(seq):
    func = "(define (c{0}r x) {1})"
    return func.format(''.join(seq),
                       cadrify(seq))
```

Combinations

- Result of script:

```
(define (caar x) (car (car x)))
(define (cadr x) (car (cdr x)))
(define (cdar x) (cdr (car x)))
(define (cddr x) (cdr (cdr x)))
(define (caaar x) (car (car (car x))))
(define (caadr x) (car (car (cdr x))))
...
(define (cdddar x) (cdr (cdr (cdr (car x)))))
(define (cddddr x) (cdr (cdr (cdr (cdr x)))))
```