Macros and Macro Hygiene

COMP6209

Julian Rathke
jr2@ecs.soton.ac.uk
Lecture Goals

• introduce C-style macros as “bare bones” code generation technique
  – similar techniques used in template engines

• show pitfalls of C-style macros
  – similar pitfalls occur in template engines
Macro Definitions

• macros associate names with code fragments
  
  #define PI 3.1415

• macro definitions can have parameters
  
  #define RADTODEG(x) ((x)*57.2958)

• macro definitions can be layered
  
  #define DEGTORAD(x) ((x)*PI/180)
  – even recursive

• common in
  – scientific computation
  – embedded programming

• often encode small programming idioms
Lexical macro expansion textually replaces macros by their definitions.

- Pre-processor scans the program text
- On `#define`, macro definition is extracted and stored
- If a name is macro-defined, it is expanded
  - Macro parameters replaced by expanded arguments
  - Result is re-expanded to expand lower macro layers
- If a name is not macro-defined, it is copied to output

```
#define SWAP(T,x,y) do {T tmp=x; x=y; y=tmp;} while(0)
...
void main() {
...
  int lo, hi;
  double x, y;
...
  SWAP(int, lo, hi);
...
  SWAP(double, x, y);
...
}
```

“Macro expansion is a tricky operation, fraught with nasty corner cases […]”

http://gcc.gnu.org
Macros vs. Functions

• macros look like functions…
  
  \[
  \text{#define DEGTORAD(x) ((x)\times\pi/180)}
  \]

  \[
  \text{vs.}
  \]

  \[
  \text{double degtorad(x) \{return(x\times\pi/180)\}}
  \]

• and macro expansion looks like function inlining:
  \[
  \text{DEGTORAD(5)-PI} \rightarrow ((5)\times3.1415/180)-3.1415
  \]

• but macro arguments and bodies can be arbitrary code fragments:
  \[
  \text{#define SWAP(T,x,y) \{T t=x;x=y;y=t;\}}
  \]

  \[
  \text{#define INITIALIZE = 0}
  \]

  \[
  \text{SWAP(int,lo,hi); } \rightarrow \{\text{int t=lo;lo=hi;hi=t;}\}
  \]

  \[
  \text{lo INITIALIZE; } \rightarrow \text{lo = 0;}
  \]

  not a legal function argument

  not a legal statement
Macros vs. Functions (cont’d)

• macros can create new names and definitions

```
#define ACCEPT(IF)                   \
public void accept(IF##Visitor v) {  \ 
  v.visit(this);                     \ 
}
```

```
interface CarPart{
  void accept(CarPartVisitor v);
}
```

```
class Wheel implements CarPart {
  ACCEPT(CarPart);
};
```

```
class Engine implements CarPart {
  ACCEPT(CarPart);
};
```

creates a single name
(“token pasting”)
Macros vs. Functions (cont’d)

- macros can create new names and definitions
  ```
  #define ACCEPT(IF)                   
  public void accept(IF##Visitor v) {  
    v.visit(this);
  }

  interface CarPart{
    void accept(CarPartVisitor v);
  }

  class Wheel implements CarPart {
    public void accept(CarPartVisitor v) {
      v.visit(this);
    }
  }

  ...
  ```

creates a single name ("token pasting")
Macros vs. Functions (cont’d)

- macros can create new names and definitions
  ```
  #define ACCEPT(IF)                   \ 
  public void accept(IF##Visitor v) { \ 
    v.visit(this);                     \ 
  }
  
  interface AST{
    void accept(ASTVisitor v);
  }
  
  class Stmt implements AST {
    ACCEPT(AST);                      \ 
  }
  
  class Expr implements AST {
    ACCEPT(AST);                      
  }
  ```

creates a single name ("token pasting")

can use same macro
Macros vs. Functions (cont’d)

• macros can create new names and definitions

```java
#define ACCEPT(IF)                   
public void accept(IF##Visitor v) {  
v.visit(this);
}
```

creates a single name ("token pasting")

```java
interface AST{
    void accept(ASTVisitor v);
}
```

```java
class Stmt implements AST {
    public void accept(ASTVisitor v) {
        v.visit(this);
    }
};
```

but get different type
• macros can be used to hide coding idioms in syntax:
  
  ```
  vector<AST*>& n = tree.getChildren();
  for(vector<AST*>::iterator it = n.begin();
      it != n.end(); ++it )
  {
    (*it)->accept(*this);
  }
  ```

• need to find out what can be generalised
• ... and replace by macro parameters

```
#define foreach(iv, BType, c, CType)       
for(CType<BType>::iterator iv = c.begin(); 
  iv != c.end(); ++iv )
```
Macros as Syntax Extensions (cont’d)

• can now write the example as

```cpp
vector<AST*>& n = tree.getChildren();
foreach(it, AST*, n, vector)
    { (*it)->accept(*this); }
```

• can use `foreach-“statement”` for other types:

```cpp
foreach(c, char, text, List)
    { printf(stdout, "\%c", *c); }
```

```cpp
for(List<char>::iterator c = text.begin();
     c != text.end(); ++c )
    { printf(stdout, "\%c", *c); }
```

• note: `foreach` is not a true statement!
Lexical macro expansion can cause hard-to-spot errors.

- operator priorities can interfere with complex macro arguments and bodies

\[
\text{#define DEGTORAD(x) } x \times \pi / 180
\]

\[
\text{DEGTORAD(a-b)} \rightarrow a - b \times 3.1415 / 180
\]

- solution: proper bracketing of macro arguments

\[
\text{#define DEGTORAD(x) } ((x) \times \pi / 180)
\]

NB: outer brackets required to make macro work in all operator contexts

\[
1 / \text{DEGTORAD(a)}
\]
Lexical macro expansion can cause hard-to-spot errors. (cont’d)

• macro body can interfere with surrounding syntax:

```c
#define SWAP(x,y) t=x; x=y; y=t
if(a>b) SWAP(a,b); else SWAP(b,a);
```

- solution: proper “packaging” of macro body

```c
#define SWAP(x,y) \
  do {t=x; x=y; y=t} while(0)
```

causes syntax error??

ends if
Lexical macro expansion can cause hard-to-spot errors. (cont’d)

• side effects in macro arguments can be duplicated:
  
  ```c
  #define MIN(x,y) (((x)<(y)) ? (x) : (y))
  /* a==1, b==2 */
  x=MIN(a++,b++);
  /* a==3, b==3, x==3 */
  - solution: use local temporary variables
  #define MIN(x,y) \ 
  ({ typeof (x) _x = (x); \ 
    typeof (y) _y = (y); \ 
    _x < _y ? _x : _y; })
  - problem: not ANSI-C
  ```
Lexical macro expansion can cause hard-to-spot errors. (cont’d)

- variable declarations in body can capture arguments:

```c
#define SWAP(T,x,y) { T t=x; x=y; y=t; }
int t=0; int s=1; SWAP(int,t,s);
```

- what really happens:
  ```bash
  bash-3.00$ gcc test.c; a.out
  s: 1  t: 0
  after first assign  - x (t): 3649524  y (s): 1
  first last assign  - x (t): 1  y (s): 1
  s: 1  t: 0
  ```

- solution: “obfuscation” of local variables

```c
#define SWAP(T,x,y) \\  { T _t_SWAP=x; x=y; y=_t_SWAP; }
```
Macro Hygiene

Definition:

A macro is called **hygienic** if its expansion cannot cause any collisions with any context.

- **syntactic hygiene** (= no capture of argument parts)
  - guarantees **structural integrity** of expanded code
  - can be achieved by using trees instead of text

- **variable hygiene** (= no capture of identifiers)
  - guarantees **referential integrity** of expanded code
  - can be achieved by controlled renaming

- **type hygiene**
  - guarantees **type integrity** of expanded code
  - difficult to achieve
Syntax macros use syntax trees to guarantee structural integrity.

• example: macros in Scheme (R4S)

```scheme
(define-syntax push
  (syntax-rules
    ((push ?v ?s) =>
      (set! ?s (cons ?v ?s))))
```

• macro body must be syntactically well structured
  – Scheme: properly bracketed s-expression
  – languages with richer syntax more difficult

• macro body **not** necessarily compilation unit
  – can have any non-terminal of the underlying syntax
Syntax macros use syntax trees to guarantee structural integrity. (cont’d)

- macro expansion inserts trees:

- macro processor needs to be *language-aware*:
  - must *parse* macro bodies
  - must check syntactic categories
  - requires tighter integration with compiler, less general
Hygienic macros use renaming to guarantee referential integrity.

- referential integrity: names refer to the “right” objects
- broken by capture of argument by binder in body:

```c
#define SWAP(T,x,y) {T t=x;x=y;y=t}
int t=0; int s=1; SWAP(int,t,s);
```

- usual solution: obfuscation
  - does not prevent “unlucky capture”
  - does not prevent repeated introduction of names through repeated macro invocations in same scope
- better solution: bound renaming of introduced names
  - must be done for every macro invocation
  - must be done before arguments are instantiated
Hygienic macros use renaming to guarantee referential integrity. (cont’d)

- referential integrity: names refer to the “right” objects
- broken by capture of free name in body:

```scheme
(define-syntax push
  (syntax-rules
   ((push ?v ?s) =>
    (set! ?s (cons ?v ?s)))

(let ((pros (list "cheap" "fast"))
      (cons (list)))
  (push "unreliable" cons))
```

¬ (let ((pros (list "cheap" "fast"))
        (cons (list)))
    (set! cons (cons "unreliable" cons)))
Hygienic macros use renaming to guarantee referential integrity. (cont’d)

- referential integrity: names refer to the “right” objects
- broken by capture of free name in body:

```scheme
(define-syntax push
  (syntax-rules ((push ?v ?s) =>
      (set! ?s (cons ?v ?s)))))
(let ((pros (list "cheap" "fast"))
    (cons (list)))
  (push "unreliable" cons))
```

```
_cons free in push (defined outside)
```

```
_cons captures occurrence in push
```

```
(cons (cons "unreliable" cons))
```

```
(cons (list))
```

```
(set! cons (cons "unreliable" cons)))
```

```
(set! cons (cons (cons "unreliable" cons)))
```
Hygienic macros use renaming to guarantee referential integrity. (cont’d)

• referential integrity: names refer to the “right” objects
• broken by capture of free name in body:

(define-syntax push
  (syntax-rules
   ((push ?v ?s) =>
    (set! ?s (cons ?v ?s))))))
(let ((pros (list “cheap” “fast”)))
  (cons (list)))
(push “unreliable” cons))
→ (let ((pros (list “cheap” “fast”)))
    (cons (list)))
  (set! cons (cons “unreliable” cons)))
Hygienic macros use renaming to guarantee referential integrity. (cont’d)

- referential integrity: names refer to the “right” objects
- broken by capture of free name in body:
- solution: hygienic macro expansion algorithm
  - “timestamp” names with nesting levels

```lisp
(define-syntax push
  (syntax-rules
    ((push ?v ?s) =>
      (set! ?s (cons:0 ?v ?s))))

(let ((pros:1 (list:0 "cheap" "fast"))
      (cons:1 (list:0)))
  (push "unreliable\nopt\ns:1")
⇒ (let ((pros:1 (list:0 "cheap" "fast"))
        (cons:1 (list:0)))
     (set! cons:1 (cons:0 "unreliable" cons:1)))
```
Suggested further reading

• http://gcc.gnu.org/onlinedocs/cpp/Macros.html#Macros
  – the “original” macro hygiene paper; exposition a bit mathematical and uses Lisp
• Shriram Krishnamurthi, Matthias Felleisen, Bruce F. Duba: *From Macros to Reusable Generative Programming*. GCSE 1999: 105-120.
  – shows how to use syntactic macros in Scheme
  – nice overview of different macro techniques