Die Essenz von Closures

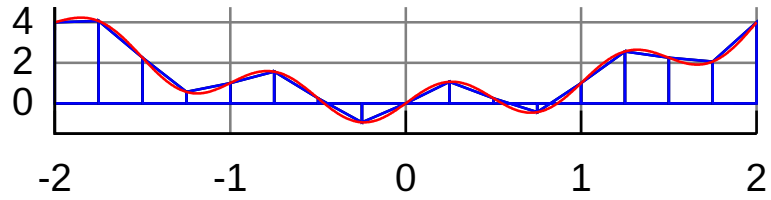
aus Sicht des Sprachentwurfs

M. Anton Ertl, TU Wien

Verschachtelte Funktionen, äußere Variablen, Closures

```
double numint(double (*f)(double),
              double l, double h);
```

$$\int_l^h f(x)dx$$




```
void foo()
{
  double y;
  double g(double x)
  {
    return pow(x,-y);
  }
  y=2.0;
  printf("%f\n",numint(g,2.0,5.0));
  y=3.0;
  printf("%f\n",numint(g,2.0,5.0));
}
```

- GCC-Erweiterung von C
- Das übergebene g braucht Speicher
- Wie wird der freigegeben?
  Scheme: Garbage Collection
  Pascal, C++, GCC: Stack
- Stack schränkt Benutzbarkeit ein
  z.B. Callbacks für GUIs
- Jenseits von Stacks ohne GC?
  $\Rightarrow$ Explizite Freigabe
- Ziel: Erweiterung für Sprachen ohne GC

Genese

- "'Closures – the Forth way"' Ertl, Paysan 2018
- Ausgangspunkt: Verschachtelte Funktionen mit äußeren Variablen
- Implementationsidee: Flat Closures
  passen gut zu expliziter Freigabe
- Erste Idee: explizites Einfangen (*capture*) äusserer locals
- Besser: Zweistufige Übergabe von Parametern
  Einfacher zu implementieren
  Oft besser zu benutzen
- auch für andere Sprachen interessant

Zweistufige Übergabe, Stack-Allokation

```
double numint(double (*f)(double), double l, double h);

double g(double y)(double x)
{
  return pow(x,-y);
}

void foo()
{
  printf("%f\n",numint(g(2.0),2.0,5.0));
  printf("%f\n",numint(g(3.0),2.0,5.0));
}
```

Zweistufige Übergabe, Heap-Allokation

```
double numint(double (*f)(double), double l, double h);

double g(double y)(double x)
{
  return pow(x,-y);
}

void foo()
{
  double (*g1)(double) = g(2.0 : mallocx);
  printf("%f\n",numint(g1,2.0,5.0));
  freex(g1);
}
```

Implementierung: Flat Closures

g(2.0) erzeugt zur Laufzeit:

```
double g_dynamic_2_0(double x)
{
  return g_static(x,2.0);
}
```

```
        movsd two(%rip), %xmm1
        mov   $g_static, %r11
        jmp   %r11
two:    .double 2.0
```

Alle Closures von g rufen diesen statischen Code auf:

```
double g_static(double x,double y)
{
  return pow(x,-y);
}
```

```
        xorpd nmask(%rip), %xmm1
        jmp   pow
nmask:  .quad 0x8000000000000000
```

Anonyme Funktion (λ) mit Heap-Allokation

```
double numint(double (*f)(double), double l, double h);

void foo()
{
  double (*g1)(double) = [double y=2.0 : mallocx](double x) {
    return pow(x,-y);
  }
  printf("%f\n",numint(g1,2.0,5.0));
  free(g1);
}
```

Und wenn ich Daten ändern will?

```
int counter(int *p)(void) {
  return (*p)++;
}

void foo()
{
  int c1v = 5;
  int *c2p = malloc(sizeof(int));  *c2p = 0;
  int (*c1)() = counter(&c1v);
  int (*c2)() = counter(c2p : mallocx);
  int (*c1a)() = counter(&c1v);
  printf("%d %d\n",(*c1)(),(*c2)()); // 5 0
  printf("%d\n",(*c1a)());           // 6
  printf("%d %d\n",(*c1)(),(*c2)()); // 7 1
}
```

- *assignment conversion*, explizit
- Je nach Sprache
  in C Zeiger
  in C++ Referenzen
  in Pascal `VAR`-Parameter

Und ganz ohne Namen?

Nicht in C $\Rightarrow$ Gforth

```
/* C (mit Variablennamen) */
[double y=2.0 : mallocx](double x) {
    return pow(x,-y);
}

\ Gforth, x und y als Namen
2e [{: f: y :}h {: f: x :} x y fnegate f** ;]

\ Gforth 2018, x bleibt auf dem Stack
2e [{: f: y :}h  y fnegate f** ;]

\ Gforth ganz ohne Namen
2e [f:h fnegate f** ;]
```

Zusammenfassung: Die Essenz von Closures

- Daten übergeben, aber nicht als Parameter erster Stufe
- Lexical Scoping ist nicht nötig
- Namen sind nicht nötig (je nach Sprache)
- Alternative: zweistufige Parameterübergabe
- Implementation: Flat Closures
  explizit deallokierbar
- Für änderbare Variablen: explizite *assignment conversion*