- Das Lambda-Kalkül geht auf Alonzo Church und Stephen Kleene zurück, die in den 30er-Jahren damit ein formales System für berechenbare Funktionen entwickelten.
- Zu den wichtigsten Arbeiten aus dieser Zeit gehört der Aufsatz von Alonzo Church: *An Unsolvable Problem of Elementary Number Theory*, *Americal Journal of Mathematics*, Band 58, Nr. 2 (April 1936), S. 345–363.
- Diese Arbeit zeigt, dass es keine berechenbare Funktion gibt, die die Äquivalenz zweier Ausdrücke des Lambda-Kalküls feststellen kann.
- Die Turing-Maschine und das Lambda-Kalkül sind in Bezug auf die Berechenbarkeit äquivalent.
- Das Lambda-Kalkül wurde von funktionalen Programmiersprachen übernommen (etwa von Lisp und Scheme) und wird auch gerne zur formalen Beschreibung der Semantik einer Programmiersprache verwendet (denotationelle Semantik).

Es gibt zwei wesentliche Punkte, weswegen Lambda-Ausdrücke auch in nicht-funktionalen Programmiersprachen (wie etwa C++) interessant sind:

- Anonyme Funktionen können lokal konstruiert und übergeben werden. Ein Beispiel dafür wäre das Sortierkriterium bei *sort*. Die lokal definierte anonyme Funktion kann dabei auch die Variablen der sie umgebenden Funktion sehen (*closure*).
- Funktionen können aus anderen Funktionen abgeleitet werden. Beispielsweise kann eine Funktion mit zwei Argumenten in eine Funktion abgebildet werden, bei der der eine Parameter fest vorgegeben und nur noch der andere variabel ist (*currying*).

Grundsätzlich kann das alles auch konventionell formuliert werden durch explizite Klassendefinitionen. Aber dann wird der Code umfangreicher, umständlicher (etwa durch die explizite Übergabe der lokal sichtbaren Variablen) und schwerer lesbarer (zusammenhängender Code wird auseinandergerissen). Allerdings können Lambda-Ausdrücke auch zur Unlesbarkeit beitragen.

Da in C++ Funktionsobjekte wegen der entsprechenden STL-Algorithmen recht beliebt sind, gab es mehrere Ansätze, Lambda-Ausdrücke in C++ einzuführen:

- *boost*::*lambda* von Jaakko Järvi, entwickelt von 1999 bis 2004
- *boost*::*phoenix* von Joel de Guzman und Dan Marsden, entwickelt von 2002 bis 2005
- Integration von Lambda-Ausdrücken in den C++-Standard ISO-14882-2012.

transform2.cpp

```
int main() {
   list<int> ints;
   for (int i = 1; i <= 10; ++i) {
      ints.push_back(i);
   }

   list<int> squares;
   transform(ints.begin(), ints.end(),
      back_inserter(squares), [](int val) { return val*val; });

   for (int val: squares) {
      cout << val << endl;
   }
}
```

- Mit Lambda-Ausdrücken werden implizit unbenannte Klassen erzeugt und temporäre Objekte instantiiert.
- In diesem Beispiel ist [](**int** *val*){ **return** *val*∗*val*; } der Lambda-Ausdruck, der ein temporäres unäres Funktionsobjekt erzeugt, das sein Argument quadriert.

⟨lambda-expression⟩ ⟶ ⟨lambda-introducer⟩ [⟨lambda-declarator⟩]
⟨compound-statement⟩

⟨lambda-introducer⟩ ⟶ „[“ [⟨lambda-capture⟩] „]“

⟨lambda-capture⟩ ⟶ ⟨capture-default⟩

⟶ ⟨capture-list⟩

⟶ ⟨capture-default⟩ „,“ ⟨capture-list⟩

⟨capture-default⟩ ⟶ „&“ | „=“

⟨capture-list⟩ ⟶ ⟨capture⟩ [„...“]

⟶ ⟨capture-list⟩ „,“ ⟨capture⟩ [„...“]

⟨capture⟩ ⟶ ⟨simple-capture⟩
⟶ ⟨init-capture⟩
⟨simple-capture⟩ ⟶ ⟨identifier⟩
⟶ „&“ ⟨identifier⟩
⟶ **this**
⟨init-capture⟩ ⟶ ⟨identifier⟩ ⟨initializer⟩
⟶ „&“ ⟨identifier⟩ ⟨initializer⟩
⟨lambda-declarator⟩ ⟶ „(“ ⟨parameter-declaration-clause⟩ „)“
[**mutable**] [⟨exception-specification⟩]
[⟨attribute-specifier-seq⟩]
[⟨trailing-return-type⟩]

- Die ⟨init-capture⟩ kommt mit dem C++14-Standard hinzu.

- Lambda-Ausdrücke, die in eine Funktion eingebettet sind, „sehen“ die lokalen Variablen aus den umgebenden Blöcken.
- In vielen funktionalen Programmiersprachen überleben die lokalen Variablen selbst dann, wenn der sie umgebende Block verlassen wird, weil es noch überlebende Funktionsobjekte gibt, die darauf verweisen. Dies benötigt zur Implementierung sogenannte *cactus stacks*.
- Da für C++ der Aufwand für diese Implementierung zu hoch ist und auch die Übersetzung normalen Programmtexts ohne Lamda-Ausdrücke verteuern würde, fiel die Entscheidung, einen alternativen Mechanismus zu entwickeln, der über *lambda-capture* spezifiziert wird.

Kopien der Variablen aus der Umgebung


```
template<typename T>
function<T(T)> create_multiplier(T factor) {
   return function<T(T)>([=](T val) { return factor*val; });
}

int main() {
   auto multiplier = create_multiplier(7);
   for (int i = 1; i < 10; ++i) {
      cout << multiplier(i) << endl;
   }
}
```

- *create_multiplier* ist eine Template-Funktion, die ein mit einem vorgegebenen Faktor multiplizierendes Funktionsobjekt erzeugt und zurückliefert.
- Die Standard-Template-Klasse *function* wird hier genutzt, um das Funktionsobjekt in einen bekannten Typ zu verpacken.
- Die *lambda-capture* [=] legt fest, dass die aus der Umgebung referenzierten Variablen beim Erzeugen des Funktionsobjekts kopiert werden.

Erzeugung einer unbenannten Klasse 83

```
template<typename T>
class Anonymous {
   public:
      Anonymous(const T& factor_) : factor(factor_) {}
      T operator()(T val) const {
         return factor*val;
      }
   private:
      T factor;
};

template<typename T>
function<T(T)> create_multiplier(T factor) {
   return function<T(T)>(Anonymous<T>(factor));
}
```

- Der Lambda-Ausdruck führt implizit zu einer Erzeugung einer unbenannten Klasse (hier einfach *Anonymous* genannt).
- Jeder aus der Umgebung referenzierte Variable, die kopiert wird, findet sich als gleichnamige Variable der Klasse wieder, die bei der Konstruktion übergeben wird.

```
template<typename T>
tuple<function<T()>, function<T()>, function<T()>>
create_counter(T val) {
   shared_ptr<T> p(new T(val));
   auto incr = [=]() { return ++*p; };
   auto decr = [=]() { return --*p; };
   auto getval = [=]() { return *p; };
   return make_tuple(function<T()>(incr),
      function<T()>(decr), function<T()>(getval));
}
```

- In funktionsorientierten Sprachen werden gerne die gemeinsamen Variablen aus der Hülle benutzt, um private Variablen für eine Reihe von Funktionsobjekten zu haben, die wie objekt-orientierte Methoden arbeiten.
- Das ist auch in C++ möglich mit Hilfe von *shared_ptr*.
- Aber normalerweise ist es einfacher, eine entsprechende Klasse zu schreiben.

```
int main() {
   function<int()> incr, decr, getval;
   tie(incr, decr, getval) = create_counter(0);
   char ch;
   while (cin >> ch) {
      switch (ch) {
         case '+': incr(); break;
         case '-': decr(); break;
         default: break;
      }
   }
   cout << getval() << endl;
}
```

- *create_counter* erzeugt ein Tupel (Datenstruktur aus **#include** <tuple>) und *tie* erlaubt es, gleich mehrere Variablen aus einem Tupel zuzuweisen.
- Danach bleibt die gemeinsame private Variable solange bestehen, bis diese von *shared_ptr* freigegeben wird, d.h. sobald die letzte Referenz darauf verschwindet.

```
vector<int> values(10);
int count = 0;
generate(values.begin(), values.end(), [&]() { return ++count; });
```

- Alternativ können Variablen nicht kopiert, sondern per impliziter Referenz benutzt werden.
- Dann darf das Funktionsobjekt aber nicht länger leben bzw. benutzt werden, als die entsprechenden Variablen noch leben. Das liegt in der Verantwortung des Programmierers.
- *generate* steht über **#include** <algorithm> zur Verfügung und weist die von dem Funktionsobjekt erzeugten Werte sukzessiv allen referenzierten Werten zwischen dem ersten Iterator (inklusive) und dem zweiten Iterator (exklusive) zu.