Вообще-то, теория вычислимости и теория алгоритмов - это не то, что должен обязательно знать каждый программист. С другой стороны, например, машина Тьюринга настолько на слуху, что иметь хотя бы общее представление о том, что это такое, весьма желательно.

Entscheidungsproblem

Началось всё около ста лет назад, в 1928 г., когда немецкий математик Давид Гильберт сформулировал т.н. “проблему разрешения” (Entscheidungsproblem): нахождение последовательности и приёмов действий (алгоритма), который на основании формальной системы правил и некоторого утверждения сделал бы вывод, ложно это утверждение или истинно. Происходило это в рамках проводимой Гильбертом аксиоматизации математики, для чего не хватало доказательства непротиворечивости и полноты арифметики натуральных чисел.

В 1930 г. австрийский математик Курт Гёдель доказал (и опубликовал в 1931) теоремы о неполноте. В первой из них утверждается, что любая формальная система или неполна или противоречива. Во второй теореме речь идёт об арифметике: e*сли формальная арифметика* S *непротиворечива, то в ней невыводима формула, содержательно утверждающая непротиворечивость* S (отсюда).

Теоремы Гёделя имеют не только математический смысл, но и философский. Они имеют прямое отношение к великому множеству логических парадоксов, вот, например, как формулируется парадокс Пиноккио:
“Когда Пиноккио лжет, его нос тут же заметно увеличивается. Что будет, если Пиноккио скажет: «Сейчас у меня удлинится нос»?
Если нос не увеличится — значит, мальчик соврал, и нос будет обязан тут же вырасти. А если нос вырастет — значит, мальчик сказал правду, но тогда почему вырос нос?”.

Суть подобных парадоксов - рекурсия, ссылка утверждения на себя. Для доказательства теорем Гёдель предложил некоторым образом пронумеровать все доказуемые утверждения формальной системы, а затем предъявить номер, который вычислить невозможно. Этот приём - нумерация выражений, с тех пор называется “гёделизация”. Для получения невозможных номеров как раз и используют рекурсию.

Парадоксы, связанные с логической рекурсией весьма разнообразны.
Вот, например:

Пусть у нас есть возможность написать текст фиксированной длины, он может быть длинным, но ограниченным. В этом тексте требуется описать число, используя общепринятые термины, запрещено использовать математические неопределённости, как, например, деление на 0. Вопрос, существует ли максимальное число, которое можно описать подобным образом?

С одной стороны, да. Ведь количество терминов ограничено, количество комбинаций терминов ограничено размером текста. Значит и количество вариантов описаний чисел ограничено, следовательно, среди них можно выбрать максимум.

С другой стороны, а что если мы опишем число так:
“Пусть X - максимальное число, которое можно описать в тексте ограниченной длины, используя общепринятые термины и не используя математические неопределённости.
Так вот, результат равен X+1”.

Аналогичным образом устроены числа Ришара.

- вещественные числа описываются свободным текстом, например, “соотношение длины окружности к радиусу”
- описания чисел сортируются в лексикографическом порядке
- если есть несколько описаний числа, из них оставляют лексикографически первое
- числам присваивают порядковые номера (Ришара) в отсортированном массиве описаний

Что, если мы опишем число, используя значение его номера Ришара? Получим противоречие. Ведь чтобы получить число Ришара надо отсортировать массив описаний, а мы ссылаемся на число так, что это может повлиять на сортировку.

Здесь же стоит упомянуть теорему польского математика Тарского “о невыразимости истины” (1936 г.). Она говорит о том, что понятие арифметической истины не может быть выражено средствами самой арифметики.

В 1936 г. Алонзо Чёрч опубликовал статью (An unsolvable problem of elementary number theory), в которой предложил формализм, названный им “λ-исчисление” (которое теперь называется λ-исчисление Чёрча). С помощью λ-исчисления Чёрч доказал, что Entscheidungsproblem в общем случае неразрешима. Т.е. алгоритм, который для любого утверждения некоторой формальной системы может вывести суждение об его истинности невозможен.

В 1936 г. также опубликована основополагающая статья на этот раз британского математика Алана Тьюринга “On computable numbers, with an application to the Entscheidungsproblem”. Он ввёл абстрактную машину, позднее названную “машиной Тьюринга”, которая как раз и занималась тем, что выносила суждения об утверждениях в рамках описанной для неё формальной системы. Тьюринг также показал, что есть утверждения, которые вычислить невозможно.

Позднее было установлено, что машина Тьюринга и λ-исчисление Чёрча эквивалентны, это разные способы описания одного и того же. Теперь данный факт называется тезис Чёрча-Тьюринга.

Итак, в 1936 г. точка была поставлена, Entscheidungsproblem неразрешима.

λ-исчисление

λ-исчисление - это не язык программирования. Это формальный аппарат для выражения алгоритмов. В нём нет констант, арифметических операторов, условных выражений, циклов … Одни только функции. Но с помощью них можно вывести всё, что вам требуется и оно будет работать как положено. Правда очень неэффективно, … но, еще раз, λ-исчисление - это не про программирование, а про формализацию идей.

- В основе λ-исчисления лежит анонимная функция (λ-функция), синтаксически это записывается как:
  `f = λx.t`
  где `x` — аргумент функции, `t` — её тело.
- Применение (аппликация) функции выглядит как
  `f x`
  где `f` — функция, `x` — подставляемое в неё значение аргумента
- Функция с двумя переменными выглядит так
  `f = λx.λy.t`
  её применение
  `f v w` или `(f v) w`
  скобки просто для обозначения порядка действий
- Вычисление `f v` заключается в том, что внутри тела функции `t` все включения `x` заменяются на `v`. Этот процесс называется β-редукция, а само применение `v` к `f` - редексом (от reducible expression - сокращаемое выражение)
- Если переменная `x` использована внутри тела функции `t`, то она считается связанной, иначе - свободной.
- Если в вычисляемом выражении (терме) не осталось свободных переменных, он считается вычисленным или, говорят, что он находится в нормальной форме. Некоторые выражения не имеют нормальной формы, а скатываются в бесконечную рекурсию, например, выражение
  `λx.xx λx.xx`
  на каждом шаге β-редукции порождает само себя, в нём первый терм - λ-функция, второй - подставляемый вместо `x` аргумент.

Что же со всем этим делать, спросит озадаченный читатель.
Уже говорилось выше, что в λ-исчислении нет ни констант, ни операторов, одни только функции, поэтому существуют функции специального вида, которые ведут себя как константы, операторы, конструкции языка, … Называется это кодирование Чёрча, посмотрим на несколько примеров:

- т.н. булеаны Чёрча
  ```
  true = λt.λf.t
  false = λt.λf.f
  ```
  двухаргументные функции, первая всегда возвращает первый аргумент, вторая - второй

- оператор ветвления выглядит так:

  ```
  if = λb. λx. λy.b x y
  ```

  где `b` принимает только значения `true` или `false` например,

  ```
  if false then else
  ```

  редуцируется следующим образом:

  ```
  (λb. λx. λy. b x y) false then else   =>
  (λx. λy. false x y) then else         =>
  (λy. false then y) else               =>
  (false then else)                     =>
  (λt.λf.f) then else                   =>
  (λf. then) else                       =>
  else
  ```

- логические операторы

  ```
  and = λp. λq. p q p
  or  = λp. λq. p p q
  ```

  пример

  ```
  and true false                        =>
  (λp. λq. p q p) true false            =>
  true false true                       =>
  (λt. λf. t) false true                =>
  false
  ```

- числа

  ```
  0 = λs.λz. z
  ```

  здесь `s` - функция, которую нужно применить к `z` в данном случае 0 раз

  ```
  1 = λs.λz. s z
  2 = λs.λz. s (s z)
  3 = λs.λz. s (s (s z))
  …
  6 = λs.λz. s (s (s (s (s z))))
  ```

- сложение

  ```
  plus = λx. λy. λs. λz. x s (y s z)
  ```

  1 + 2, `s'` и `z'` здесь - свободные переменные, `'`для того, чтобы не путать их с внутренними переменными

  ```
  plus one two s' z'                          =>
  (λx. λy. λs. λz. x s (y s z)) one two s' z' =>
  one s' (two s' z')                          =>
  (λs.λz. s z) s' (two s' z')                 =>
  s' (two s' z')                              =>
  s' (( λs.λz. s (s z) s' z')                 =>
  s' (s' (s' z'))                             =>
  three s' z'
  ```

- умножение, два варианта

```
times = λx. λy. x (plus y) z
times' = λx. λy. λs. λz. x (y s) z
```

2 * 3 =

```
times' two three s' z'                        =>
(λx.λy.λs.λz. x (y s) z) two three s' z'      =>
two (three s') z'                             =>
(λs.λz. s (s z)) (three s') z'                =>
three s' ((three s') z')                      =>
(λs.λz. s (s (s z))) s' ((three s') z')       =>
s' (s' (s' ((three s') z')))                  =>
s' (s' (s' (((λs.λz. s (s (s z))) s') z')))   =>
s' (s' (s' (((λz. s' (s' (s' z))) z')))       =>
s' (s' (s' (s' (s' (s' z')))))                =>
six s' z'
```

Машина Тьюринга

Машина Тьюринга довольно просто устроена:

- в состав машины входит неограниченная в обе стороны лента, разделённая на ячейки
- в ячейку ленты входит один символ некоторого алфавита A (специфичного для каждой конкретной машины)
- алфавит машины включает специальный “пустой” символ (nil), которым заполнены все неиспользуемые ячейки ленты
- считывающее устройство всегда находится над какой-то ячейкой и умеет передвигаться по ленте в обе стороны, считывать и записывать по одному символу
- конкретная машина Тьюринга имеет набор состояний Q
- а также набор переходов из состояния в состояние под воздействием прочитанных символов.
- переход описывается как $q_i(a_j) \rightarrow q_{i1}(a_{j1})d_k$ что означает: если головка находится в состоянии q_i, а в обозреваемой ячейке записана буква a_j, то головка переходит в состояние q_{i1}, в ячейку вместо a_j записывается a_{j1}, головка делает движение d_k, которое имеет три варианта: сдвиг на ячейку влево L, сдвиг на ячейку вправо R, остаться на месте N.
- Для остановки работы используется переход -> stop

Несмотря на свою кажущуюся примитивность, машина Тьюринга также неисчерпаема, как и λ-исчисление. Попробуем спроектировать специфическую машину для суммирования 8-разрядных целых чисел, записанных в бинарном виде.

- алфавит: {0, 1, nil},

- содержимое ленты: лента пуста (заполнена nil) за исключением 16 разрядов - по восемь на оба суммируемых аргументов, результат запишем во второй из

них, числа записаны в бинарном виде как little-endian - младшие разряды вперёд, один разряд - одна ячейка.
Т.е. с 0 по 7 ячейки - первое число, с 8 по 15 - второе

- состояние чтения разряда из левого (первого) аргумента
  $Q\_read\_left_i(0) \rightarrow Q\_shift\_right\_0\_7(0)R$ или
  $Q\_read\_left_i(1) \rightarrow Q\_shift\_right\_1\_7(1)R$
  т.е. после чтения разряда левого аргумента записываем его же обратно и двигаемся направо для чтения второго аргумента, приходится поддерживать две ветки переходов чтобы сохранить информацию о том, какой разряд был прочитан. Q_shift_right_0 - ветка для передачи прочитанного 0, Q_shift_right_1 - единицы. Последняя цифра в названии состояния (правее названия ветки) - номер состояния, в данном случае - на сколько еще разрядов следует сдвинуться.

- Q_shift_right_1_7(*) → Q_shift_right_1_6(*)R
  Q_shift_right_1_6(*) → Q_shift_right_1_5(*)R
  Q_shift_right_1_5(*) → Q_shift_right_1_4(*)R
  Q_shift_right_1_4(*) → Q_shift_right_1_3(*)R
  Q_shift_right_1_3(*) → Q_shift_right_1_2(*)R
  Q_shift_right_1_2(*) → Q_shift_right_1_1(*)R
  попав в ветку Q_shift_right мы безусловно переходим по её состояниям до тех пор, пока не сдвинемся суммарно на 8 разрядов, всё что читаем по дороге в неизменном виде записываем обратно. Теперь считывающая головка машины стоит над первым разрядом правого (второго) аргумента суммирования. Ветка Q_shift_right_0 аналогична Q_shift_right_1.

- в зависимости от того, какое значение мы прочитали во втором аргументе, записываем сумму и начинаем двигаться на этот раз на 7 разрядов влево.
  Q_shift_right_0_1(0) → Q_shift_left_0_6(0)L
  Q_shift_right_0_1(1) → Q_shift_left_0_6(1)L
  Q_shift_right_1_1(0) → Q_shift_left_0_6(1)L
  Q_shift_right_1_1(1) → Q_shift_left_1_6(0)L
  ветка Q_shift_left_1 предназначена для передачи переноса разряда т.к. прибыли по ветке с 1 в левом аргументе и в правом прочитали тоже единицу. Записываем 0 и переходим на ветку Q_shift_left_1

- Q_shift_left_0_6(*) → …
  … → Q_shift_left_0_1(*)L
  попав в ветку Q_shift_left мы безусловно переходим по её состояниям до тех пор, пока не сдвинемся суммарно на 7 разрядов, всё что читаем по дороге в неизменном виде записываем обратно. Теперь считывающая головка машины стоит над вторым разрядом левого аргумента суммирования.
  Ветка Q_shift_left_0 аналогична Q_shift_left_1.

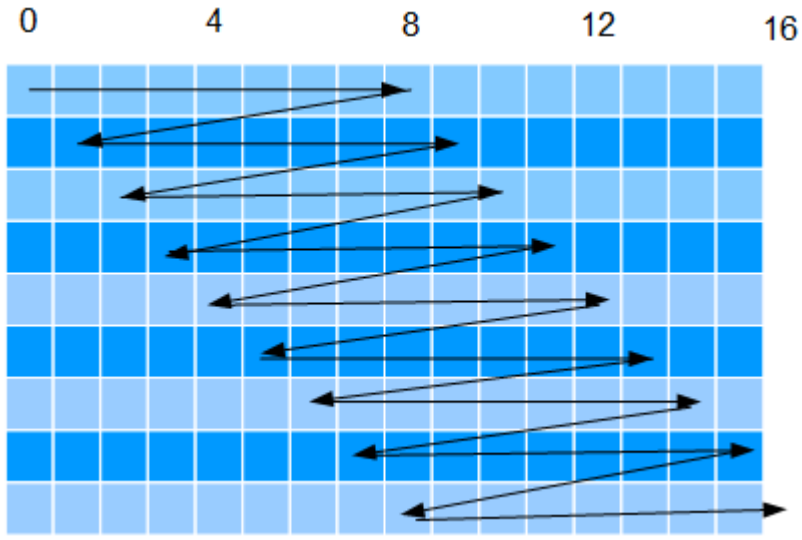
- чтение разряда из левого аргумента
  $Q\_shift\_left\_0\_1(0) \rightarrow Q\_shift\_right\_0\_7(0)R$

Q_shift_left_0_1(1) → Q_shift_right_1_7(1)R
если не было переноса разряда, действуем аналогично Q_read_left, т.е.после чтения разряда левого аргумента записываем его же обратно и двигаемся направо для чтения второго аргумента по одной из двух веток

- в случае же, если мы прибыли по ветке с переносом знака, всё усложняется
  Q_shift_left_1_1(0) → Q_shift_right_1_7(0)R
  Q_shift_left_1_1(1) → Q_shift_right_cr_7(1)R
  т.е. если прочитали 0, двигаемся по Q_shift_right_1, как будто прочитали 1, а если прочитали 1, придётся завести новую ветку Q_shift_right_cr

- ветка Q_shift_right_cr аналогична Q_shift_right_1, обработка чтения правого аргумента отличается
  Q_shift_right_cr_1(0) → Q_shift_left_1_6(0)L
  Q_shift_right_cr_1(1) → Q_shift_left_1_6(1)L

- нужны состояния для остановки работы алгоритма
  Q_shift_right_0_1(nil) → stop
  Q_shift_right_1_1(nil) → stop
  Q_shift_right_cr_1(nil) → stop
  т.е. если прочитано пустое значение, а это может быть только если мы вышли за пределы правого аргумента, следует остановиться.



Фиг.1. Движение головки машины Тьюринга при суммировании 8-разрядных чисел

Вот так простой алгоритм переноса разряда при суммировании превратился в довольно громоздкую конструкцию. Неудивительно, что автоматы, подобные машине Тьюринга иногда называют Тьюринговской трясиной. Эта конструкция придумана математиком для математиков, она предназначена скорее для упражнения ума и уж точно не для реальных вычислений.

Марковские алгоритмы (нормальные алгоритмы)

Это еще один формальный способ определения понятия “алгоритм” (алгори**Ф**м, в терминах А.А.Маркова (младшего) , на тот момент, профессора Ленинградского Университета). Появился он позже машины Тьюринга и λ-исчисления, в конце 1940-х гг.

Нормальные алгоритмы оперируют с алфавитами и словами.
Для каждого алгоритма определяется его алфавит, из алфавита образуются слова, а также определяется т.н. схема - набор правил подстановки/замены.
Правила подстановки могут быть регулярными и завершающими. После применения завершающего правила работа алгоритма останавливается, после регулярного продолжается. Завершающего правила может и не быть, в таком случае алгоритм работает до тех пор, пока возможна хоть одна подстановка.

Если можно применить несколько подстановок, выбирается та, которой соответствует самый короткий префикс. Когда правило A → B применяется к строке PAS, получается строка PBS, подстроку P будем называть префиксом, а S - суффиксом. Так вот, из всех возможных подстановок выбирается та, которая даёт самый короткий префикс.

Рассмотрим на примере.
Алгоритм без завершающего правила, преобразующий число из двоичного вида в унарное представление (непозиционная система счисления с единственной цифрой, например, числу 3 соответствует запись |||).

- алфавит: { 0, 1, | }
- правила:
  1 → 0|
  |0 → 0||
  0 → ‘’ (пустая строка)
- исходная строка: 110
- Работа алгоритма:
  110 → 0|10
  0|10 → 0|0|0
  0|0|0 → 00|||0
  00|||0 → 00||0||
  00||0|| → 00|0||||
  00|0|||| → 000||||||
  000|||||| → 00||||||
  00|||||| → 0||||||
  0|||||| → ||||||

Машина Тьюринга может быть реализована в виде нормального алгоритма, в этом смысле они полные по Тьюрингу. Любому нормальному алгоритму соответствует какая-то машина Тьюринга и наоборот. Однако, нормальные алгоритмы могут быть весьма эффективны в определённых задачах, в отличие от машины Тьюринга. Неудивительно, что на их основе создан довольно популярный язык программирования Рефал (Валентин Турчин, 1966).

Языки программирования

Тезис Чёрча говорит об эквивалентности λ-исчисления и машины Тьюринга - всё, что может быть вычислено машиной Тьюринга, также можно получить с помощью λ-исчисления (частично рекурсивных функций). Был введён также термин полнота по Тьюрингу - способность “исполнителя” реализовать любую вычислимую функцию, включая себя самого. При том, что любая вычислимая функция реализуема машиной Тьюринга это означает, что для любого “исполнителя” достаточно доказать возможность реализации машины Тьюринга, чтобы стать “полным по Тьюрингу” и гарантировать вычислимость всего что в принципе можно вычислить.

Однако, на практике не нужен “любой исполнитель”, если под исполнителем понимать компьютер. Нужен тот, который умеет решать задачи максимально эффективно, и не сам а с помощью языка программирования, который по возможности облегчает жизнь программиста, а не усложняет её как, например, INTERCAL[1], который был создан как пародия на другие языки программирования. Ни машина Тьюринга ни λ-исчисление Чёрча в чистом виде для решения практических задач совершенно непригодны.

Когда возникла потребность в реальных вычислениях, первые компьютеры разрабатывались без опоры на математический аппарат и уже задним числом выяснилось, что, например, ENIAC обладает полнотой по Тьюрингу, а Mark I - нет.

Для работы с первыми компьютерами, конечно, никаких языков программирования не существовало, Это неудивительно, учитывая, что, например, для организации цикла в Mark I приходилось склеивать перфоленту в кольцо. Тем не менее, очень быстро выяснилось, особенно, учитывая с какой скоростью появлялись новые архитектуры, что переносимый (с архитектуры на архитектуру) способ записи программ жизненно необходим.

При большом количестве развивающихся архитектур естественным образом возникло и разнообразие проектов языков высокого уровня. Тех, что принято называть языками третьего поколения при том, что первым поколением считается программирование на автокоде, а вторым - использование ассемблера. Ассемблер позволяет писать код в мнемонически легко запоминаемом виде, но не избавляет от деталей архитектуры. А язык программирования абстрагирует программиста от этих особенностей и позволяет оперировать данными близко к тому, как это принято в математике. Собственно первые языки программирования и были скорее трансляторами формул, чем языками в современном понимании.

Исторически первым можно считать Plankalkül Конрада Цузе (Konrad Zuse), придуманный (но не реализованный) им для своей же машины Z4 (1945).

Из успешных попыток, получивших развитие, стоит упомянуть A-0, разработанный под руководством Грейс Хоппер (Grace Hopper). Язык был написан для проекта UNIVAC-I

[1] К той же категории можно отнести Befunge, Brainfuck, Thue, …

Преспера Эккерта и Джона Мокли, с которого, кстати, началась фон-Неймановская (см. главу 2.1.) ветка архитектур. Изначально язык представлял собой последовательность вызовов функций с аргументами. При этом сами функции были системными, написанными на ассемблере. По мере развития появились языки A-1, A-2, A-3, AT-3, B-0.

A-3 (1955) фактически был расширением A-2, библиотекой математических функций для него и получил (в маркетинговых целях) собственное имя ARITH-MATIC.

AT-3 (Algebraic Translator-3, 1955..1957), использовавшийся для математических расчетов, получил название MATH-MATIC, B-0 предназначался для коммерческого применения и обрёл маркетинговое имя FLOW-MATIC.

AT-3 оказал влияние на возникший чуть позже ALGOL-58, B-0 повлиял на COBOL (1959) настолько, что Грейс Хоппер называют бабушкой Кобола.

В Советском Союзе первый язык программирования (транслятор) был написан под руководством Алексея Ляпунова на математическом факультете МГУ в 1952..1955 гг. Назывался он ПП-1 (программирующая программа 1) и целевой архитектурой служил компьютер МЭСМ. Развитием послужил транслятор ПП-2 (1955, ЭВМ Стрела), для которого существовал, кстати, и отладчик, возможно, первый в истории.

А.А.Ляпунов развивал направление программирования, которое называл операторным. Суть его заключается в алгоритмической декомпозиции процесса решения задачи (разбиение на операторы), таким образом, чтобы средствами языка программирования сконструировать программу, выполняющую требуемое преобразование данных. В сущности, Ляпунов подвёл теоретическую базу под императивное программирование, которое до того момента развивалось стихийно. Также, как до этого подвели базу под функциональное программирование Чёрч, Клини и Тьюринг.

Здесь следует определиться с терминами.

- в императивных языках программист явно задаёт последовательность действий
- “Основной особенностью функционального программирования … является то, что в ней реализуется *модель вычислений без состояний*. Если императивная программа на любом этапе исполнения имеет состояние, то есть совокупность значений всех переменных, и производит побочные эффекты[2], то чисто функциональная программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. То, что в императивных языках делается путём присваивания значений переменным, в функциональных достигается путём передачи выражений в параметры функций. Непосредственным следствием становится то, что чисто функциональная программа не может изменять уже имеющиеся у неё данные, а может лишь порождать новые путём копирования и/или расширения старых. Следствием того же является отказ от циклов в пользу рекурсии” (отсюда)

[2] побочными эффектами функции (помимо результата её работы) могут быть изменение переменных, переданных по ссылке или изменение глобальных переменных

Можно сказать, что λ-исчисление Чёрча является прародителем всех функциональных языков так же как машина Тьюринга стала прообразом всех императивных языков.

Независимо от Ляпунова, Екатериной Ющенко и Владимиром Королюком на основе идей А.Н.Колмогорова создаётся т.н. адресный язык программирования (1955..1958), в котором впервые появляется концепция указателя. Указателем называется адрес в памяти, который можно как обычное число сохранить в переменной, производить над ним манипуляции, разыменовать (получать значение). Повторно “впервые” концепция указателя появилась в языке PL/1 от IBM (1965).

Из языков программирования, доживших до наших дней, нельзя не упомянуть FORTRAN и LISP.

FORTRAN (Formula Translator, автор Джон Бэкус (John Backus) 1957..1958) изначально был аналогом АТ-3 разработанным IBM примерно в то же время для мэйнфрейма IBM-701. Язык оказался весьма эффективен для научных и инженерных расчетов (и до сих пор таков), плюс его продвигала IBM на всех своих платформах. Так что язык получил широкое распространение, в том числе в СССР, на нём написано огромное количество кода, переписывать который на более современные языки программирования слишком накладно. Существовавший какое-то время транслятор фортрана в C (f2c) не смог его заменить т.к. производительность деградировала после трансляции. Неудивительно, если язык доживет до своего столетнего юбилея.

LISP (LISt Processing language, 1958) создан Джоном Маккарти, бывшим в тот момент профессором Массачусетского технологического института (MIT). Занимался он искусственным интеллектом, но не в нынешнем его понимании, а скорее в области автоматического доказательства теорем. Имея опыт работы с системой работы со списками (IPL) он захотел привнести эти возможности в FORTRAN (IBM-704), с которым тоже имел дело, но не нашел понимания у разработчиков.
После этого Маккарти пытался найти общий язык с разработчиками ALGOL-а, но тоже был встречен без энтузиазма. Так что пришлось писать свой собственный язык.

Все языки, которые создавались до этого момента, были императивными (в нынешних терминах), LISP оказался первым мульти-парадигмальным, программа на этом языке может быть и императивной и функциональной.

Функциональность LISP достигается за счет использования λ-функций, неотъемлемой части языка. Императивность - поддержкой цепочек команд со строго последовательным исполнением, присваивания значений переменным, циклов.
С помощью LISP также можно создавать самомодифицирующийся код посредством вызова интерпретатора LISP во время работы интерпретатора LISP.
Позднее в языке появились объектно-ориентированные возможности.
Добавьте к этому автоматическое управление памятью - “сборку мусора”[3] - что ещё надо, программисту?

[3] вариант автоматического управления памятью, когда специальный процесс подбирает “брошенные” области памяти

Возвращаясь к тезису Чёрча. Насколько эквивалентны λ-исчисление и машина Тьюринга, настолько же эквивалентны функциональные и императивные языки программирования. Если с помощью какого-либо языка программирования возможно написать машину Тьюринга, он считается полным по Тьюрингу, большинство современных языков таковы. Любая программа на полном по Тьюрингу языке имеет (минимум один) эквивалент на любом из других полных по Тьюрингу языков.

По разным причинам существуют (существовали) десятки разных языков программирования. Речь об императивных языках, т.к. именно на них ведётся львиная доля разработки, к тому же большинство императивных языков при необходимости допускают вполне себе функциональный стиль программирования. Существует ли какой-то минимальный набор особенностей, который гарантирует полноту по Тьюрингу?
Да, этот вопрос давно изучен и существует теорема Бёма-Якопини (Böhm–Jacopini, 1966), которая гласит, что любой исполняемый алгоритм может быть реализован с помощью трёх структур управления

- участков последовательного исполнения инструкций
- ветвлений - исполнения одной из двух веток кода в зависимости от значения логического выражения
- циклов - выполнения участка кода до тех пор, пока истинно логическое выражение

Что же, о парадигмах программирования можно много спорить, но надо помнить несомненное - можно использовать язык любого типа, но *в конечном счете всё будет исполнять вполне себе императивный процессор*.