Автоматизация программирования в СССР: заключительная часть

Пётр Советов, РТУ МИРЭА

C++ Russia 2024

Программирующая программа ПП-2.
Схемы Канторовича.
Нумерация значений.
Число Ершова.

Альфа-транслятор.
Переименование переменных.
Граф несовместимости.
Раскраска графа несовместимости.
Экономия памяти для массивов.

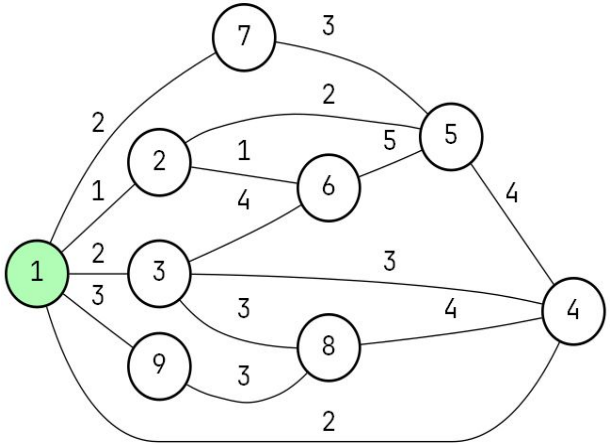
Смешанные вычисления.
Рефал и методы его компиляции.
Структурный синтез программ.

1. **Смешанные вычисления.**
2. Рефал и методы его компиляции.
3. Структурный синтез программ.

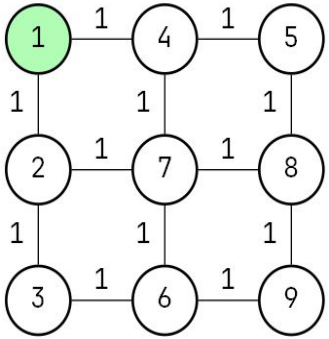
Специализация алгоритмов: от общего к частному

Возможно ли **синтезировать** специальные версии алгоритмов?

Алгоритм Дейкстры



Алгоритм Ли



Задача: обойтись без аппаратного умножения

В древних Z80 и 6502 не было аппаратного умножителя.

Во некоторых современных микроконтроллерах нет умножителя, например, на архитектуре RISC-V.

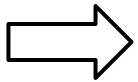
Как быть? Можно применить алгоритм двоичного умножения в столбик.

```
uint16_t mul(uint16_t x, uint16_t y) {
  uint16_t acc = 0;
  while (y) {
    if (y & 1) {
      acc += x;
    }
    x <<= 1;
    y >>= 1;
  }
  return acc;
}
```

Для умножения на **константу** можно вывести **специализированные варианты** этого алгоритма.

Пример специализации двоичного умножения (y=10)


```
uint16_t mul(uint16_t x, uint16_t y) {
  uint16_t acc = 0;
  while (y) {
    if (y & 1) {
      acc += x;
    }
    x <<= 1;
    y >>= 1;
  }
  return acc;
}
```



```
uint16_t mul_10(uint16_t x) {
  uint16_t acc = 0;
  x <<= 1;
  acc += x;
  x <<= 1;
  x <<= 1;
  acc += x;
  return acc;
}
```

Методика ручной специализации кода (1)


```
uint16_t mul(uint16_t x, uint16_t y) {
  uint16_t acc = 0;
  while (y) {
    if (y & 1) {
      acc += x;
    }
    x <<= 1;
    y >>= 1;
  }
  return acc;
}
```

1. **Выбрать доступный аргумент с известным значением.**
2. Отметить код, как доступный, если он зависит только от иного доступного кода.
3. Остальной код (задержанный) заменить на printf'ы.

Методика ручной специализации кода (2)


```
uint16_t mul(uint16_t x, uint16_t y) {
  uint16_t acc = 0;
  while (y) {
    if (y & 1) {
      acc += x;
    }
    x <<= 1;
    y >>= 1;
  }
  return acc;
}
```

1. Выбрать доступный аргумент с известным значением.
2. **Отметить код, как доступный, если он зависит только от иного доступного кода.**
3. Остальной код (задержанный) заменить на printf'ы.

Методика ручной специализации кода (3)

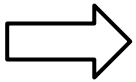

```
void mul_gen(uint16_t y) {
  printf("uint16_t mul_%d(uint16_t x) {\n", y);
  printf("  uint16_t acc = 0;\n");
  while (y) {
    if (y & 1) {
      printf("  acc += x;\n");
    }
    printf("  x <<= 1;\n");
    y >>= 1;
  }
  printf("  return acc;\n}\n");
}
```

1. Выбрать доступный аргумент с известным значением.
2. Отметить код, как доступный, если он зависит только от иного доступного кода.
3. **Остальной код (задержанный) заменить на printf'ы.**

Результат специализации (y=42)

Генерирующее расширение

```
void mul_gen(uint16_t y) {
  printf("uint16_t mul_%d(uint16_t x) {\n", y);
  printf("  uint16_t acc = 0;\n");
  while (y) {
    if (y & 1) {
      printf("  acc += x;\n");
    }
    printf("  x <<= 1;\n");
    y >>= 1;
  }
  printf("  return acc;\n}\n");
}
```



Остаточная программа

```
uint16_t mul_42(uint16_t x) {
  uint16_t acc = 0;
  x <<= 1;
  acc += x;
  x <<= 1;
  x <<= 1;
  acc += x;
  x <<= 1;
  x <<= 1;
  acc += x;
  x <<= 1;
  return acc;
}
```

Как **автоматически** построить генерирующее расширение?

Смешанные вычисления (А.П. Ершов, 1977 г.)

Интерпретатор: `int(p, d) → d'` `int('x + 1', x=2) → 3`

Смешанные вычисления (А.П. Ершов, 1977 г.)

Интерпретатор: `int(p, d) → d'` `int('x + 1', x=2) → 3`

Компилятор: `comp(p) → p'` `comp('x + 1') → 'push x push 1 add'`

Смешанные вычисления (А.П. Ершов, 1977 г.)

Интерпретатор:	`int(p, d) → d'`	`int('x + 1', x=2) → 3`
Компилятор:	`comp(p) → p'`	`comp('x + 1') → 'push x push 1 add'`
Генерирующее расширение:	`gen(d) → p`	`mul_gen(y=42) → mul_42`

Смешанные вычисления (А.П. Ершов, 1977 г.)

Интерпретатор:	`int(p, d) → d'`	`int('x + 1', x=2) → 3`
Компилятор:	`comp(p) → p'`	`comp('x + 1') → 'push x push 1 add'`
Генерирующее расширение:	`gen(d) → p`	`mul_gen(y=42) → mul_42`
Специализатор:	`spec(p, d) → p'`	`spec(mul, y=42) → mul_42`

Смешанные вычисления (А.П. Ершов, 1977 г.)

Интерпретатор:	`int(p, d) → d'`	`int('x + 1', x=2) → 3`
Компилятор:	`comp(p) → p'`	`comp('x + 1') → 'push x push 1 add'`
Генерирующее расширение:	`gen(d) → p`	`mul_gen(y=42) → mul_42`
Специализатор:	`spec(p, d) → p'`	`spec(mul, y=42) → mul_42`
Смешанный вычислитель:	`mix(p, d) → (p', d')`	Все примеры выше — это частные случаи mix

Ershov A. P. Mixed computation: Potential applications and problems for study //Theoretical Computer Science. – 1982. – Т. 18. – №. 1. – С. 41-67.

Поливариантные смешанные вычисления (Михаил Алексеевич Бульонков, 1984 г.)

Poly в процессе работы:

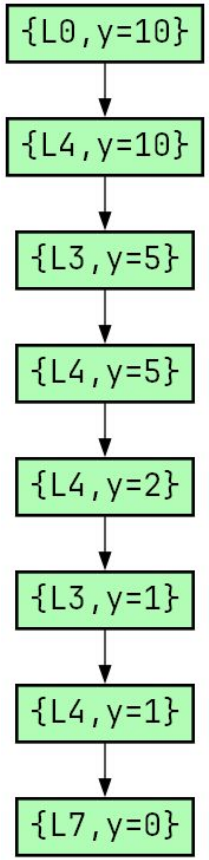
1. **выполняет** доступный код и **упрощает** задержанный код (для задержанного if обрабатывается обе ветви);
2. строит **граф состояний доступных данных.**

Poly позволяет добиться **нетривиальной** специализации кода!

Bulyonkov M. A. Polyvariant mixed computation for analyzer programs //Acta Informatica. – 1984.

Граф состояний доступных данных (y=10)


```
L0:
  r = 0; goto L1
L1:
  if y goto L2 else goto L7
L2:
  if y & 1 goto L3 else goto L4
L3:
  r = r + x; goto L4
L4:
  x = x << 1; goto L5
L5:
  y = y >> 1; goto L6
L6:
  skip; goto L1
L7:
  return r;
```

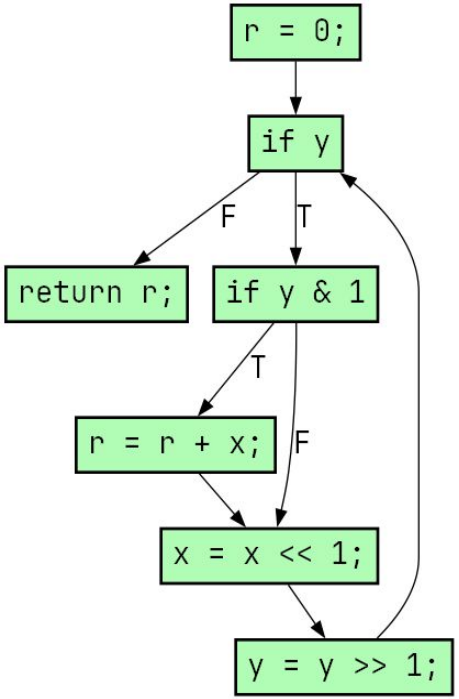


Узел графа: **метка** исходной программы (**L**) и **таблица** доступных данных **env**.

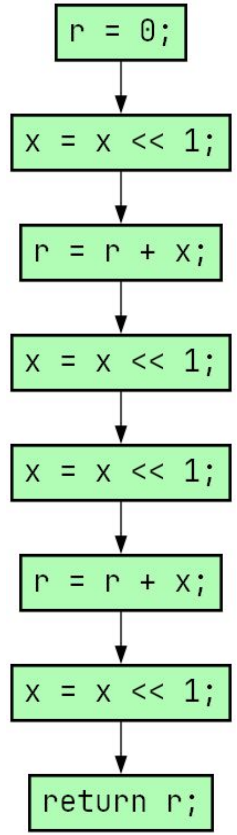
Поливариантность: исходной **метке L** может соответствовать **множество меток (L, env)** остаточной программы.

Граф конечен, если множество значений **env конечно**.

Остаточная программа (y=10)

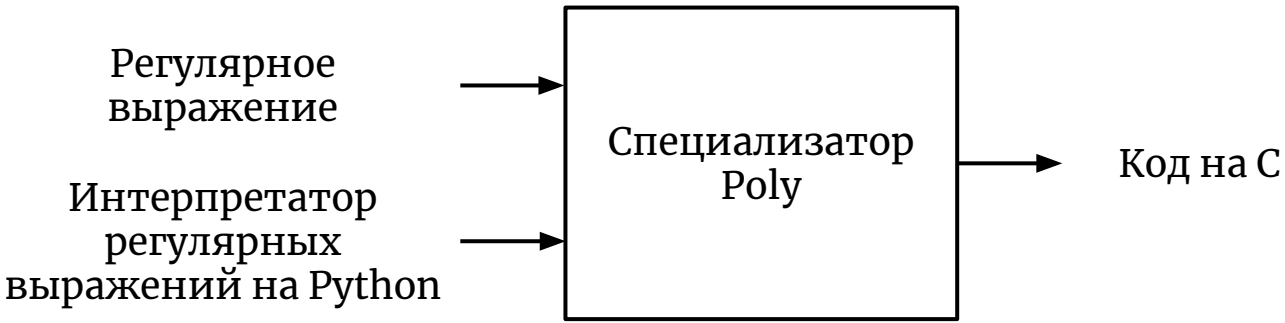



```
{L0,y=10}:
  r = 0; goto {L4,y=10}
{L4,y=10}:
  x = x << 1; goto {L3,y=5}
{L3,y=5}:
  r = r + x; goto {L4,y=5}
{L4,y=5}:
  x = x << 1; goto {L4,y=2}
{L4,y=2}:
  x = x << 1; goto {L3,y=1}
{L3,y=1}:
  r = r + x; goto {L4,y=1}
{L4,y=1}:
  x = x << 1; goto {L7,y=0}
{L7,y=0}:
  return r;
```



В остаточной программе
метками являются узлы графа.

Компилятор регулярных выражений




```
'[+-]?[0-9]+'
```

Интерпретатор регулярных выражений


```
def re_int(table, accept, text, size):
    state = 0
    i = 0
    while i < size:
        ch = text[i]
        state = table[state][ch]
        i += 1
    return accept[state]
```

table:

	+	-	0	1	2	3	4	5	6	7	8	9
S0	2	2	1	1	1	1	1	1	1	1	1	1
S1	3	3	1	1	1	1	1	1	1	1	1	1
S2	3	3	1	1	1	1	1	1	1	1	1	1
S3	3	3	3	3	3	3	3	3	3	3	3	3

accept:

S0	S1	S2	S3
False	True	False	False

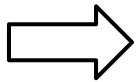
Проблема задержанного кода для таблиц


```
def re_int(table, accept, text, size):
    state = 0
    i = 0
    while i < size:
        ch = text[i]
        state = table[state][ch]
        i += 1
    return accept[state]
```

😕Переменная state зависит от **доступной** таблицы table, но вычисление **индекса** элемента таблицы **задержано**.

Трюк: выбор по неизвестному индексу в доступном массиве

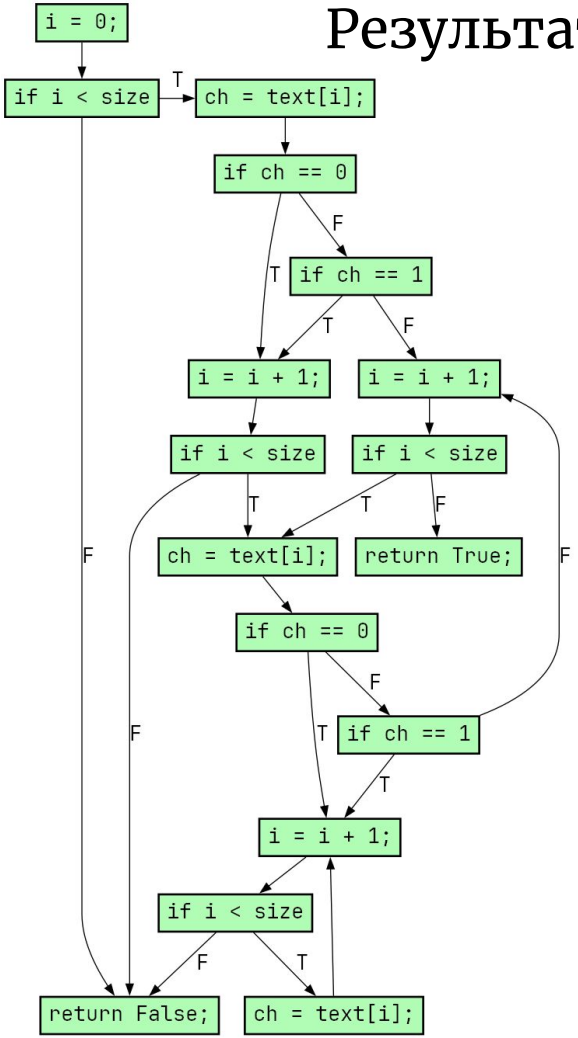

```
var = table[idx];
```



```
if (idx == 0) var = table[0];
else if (idx == 1) var = table[1];
else if (idx == 2) var = table[2];
else if ...
```

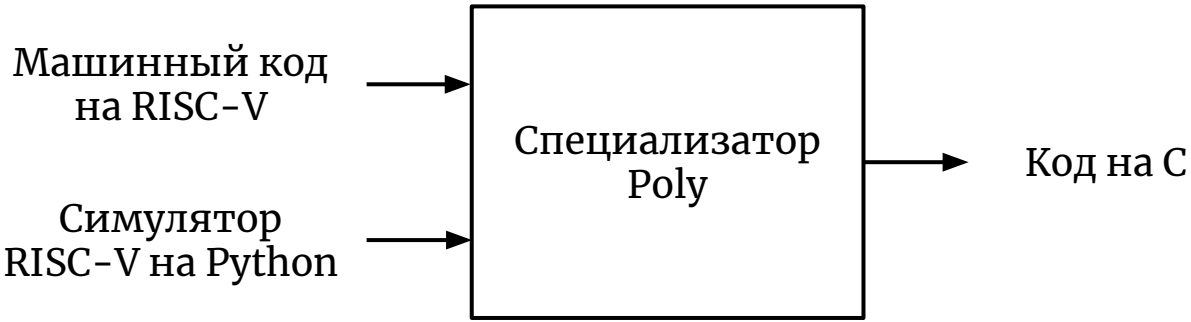
🙂Это преобразование можно добавить в Poly.

Результат компиляции [+-]?[0-9]+




```
int re_code(char *text, int size) {
  int i = 0; char ch;
  if (i < size) goto L4; else goto L3;
L3: return 0;
L4: ch = text[i]; if (ch == 0) goto L18; else
goto L7;
L7: if (ch == 1) goto L18; else goto L19;
L18: i = i + 1; goto L21;
L19: i = i + 1; goto L26;
L21: if (i < size) goto L24; else goto L3;
L24: ch = text[i]; goto L27;
L26: if (i < size) goto L24; else goto L45;
L27: if (ch == 0) goto L40; else goto L28;
L28: if (ch == 1) goto L40; else goto L19;
L40: i = i + 1; goto L47;
L45: return 1;
L47: if (i < size) goto L48; else goto L3;
L48: ch = text[i]; goto L40;
}
```

Декомпиляция с помощью Poly



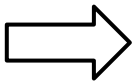

```
63 0e 05 00 93 05 05 00
13 05 10 00 33 05 b5 02
93 85 f5 ff e3 9c 05 fe
67 80 00 00 13 05 10 00
67 80 00 00
```

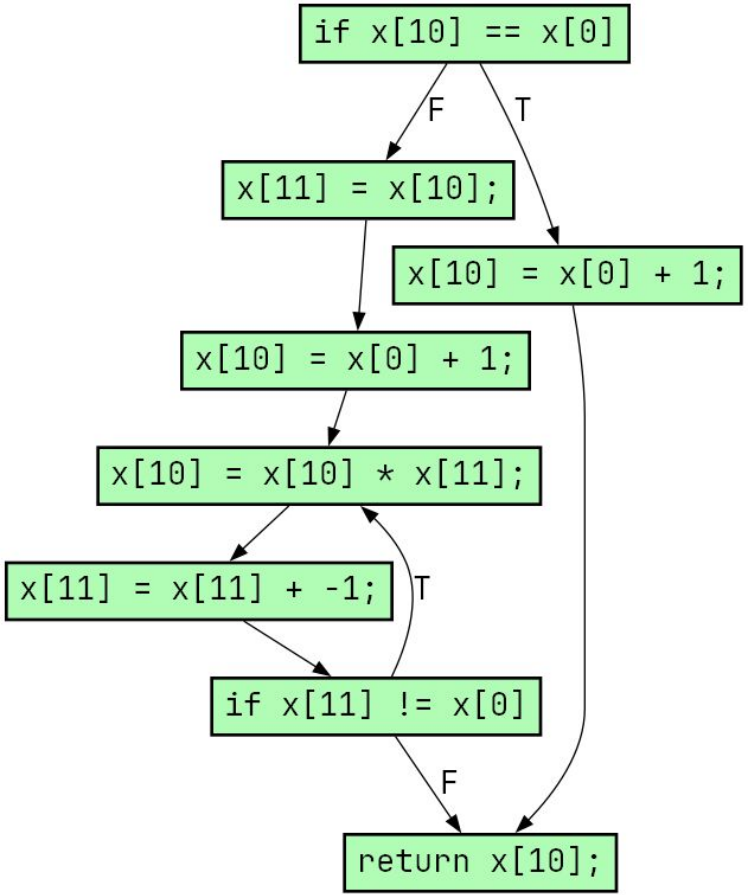
Интерпретатор упрощенного RISC-V


```
def risc_int(code, x, pc, result):
    while True:
        cmd = code[pc] | code[pc + 1] << 8
        cmd |= code[pc + 2] << 16 | code[pc + 3] << 24
        pc += 4
        op = cmd & 127
        rd = (cmd >> 7) & 31
        rs1, rs2 = (cmd >> 15) & 31, (cmd >> 20) & 31
        i_imm = cmd >> 20
        i_imm = (i_imm & 2047) - (i_imm & 2048)
        b_imm = (cmd >> 8) & 15 | (cmd >> 21) & 2032
        b_imm = (b_imm & 1023) - (b_imm & 1024)
        if op == 19: # ADDI
            x[rd] = x[rs1] + i_imm
        elif op == 51: # MUL
            x[rd] = x[rs1] * x[rs2]
        elif op == 99 and cmd & 4096: # BNE
            if x[rs1] != x[rs2]: pc += 2 * (b_imm - 2)
        elif op == 99: # BEQ
            if x[rs1] == x[rs2]: pc += 2 * (b_imm - 2)
        else: return x[result]
```

Результат (де)компиляции для ...?


```
63 0e 05 00 93 05 05 00
13 05 10 00 33 05 b5 02
93 85 f5 ff e3 9c 05 fe
67 80 00 00 13 05 10 00
67 80 00 00
```

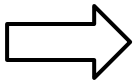




… для факториала


```
63 0e 05 00 93 05 05 00
13 05 10 00 33 05 b5 02
93 85 f5 ff e3 9c 05 fe
67 80 00 00 13 05 10 00
67 80 00 00
```

```
fact:
  beqz a0, fact_2
  mv a1, a0
  li a0, 1
fact_1:
  mul a0, a0, a1
  addi a1, a1, -1
  bnez a1, fact_1
  ret
fact_2:
  li a0, 1
  ret
```



```
int risc_code(int *x) {
  if (x[10] == x[0]) goto L50;
    else goto L41;
L41: x[11] = x[10]; goto L72;
L50: x[10] = x[0] + 1; goto L90;
L72: x[10] = x[0] + 1; goto L96;
L90: return x[10];
L96: x[10] = x[10] * x[11]; goto L111;
L111: x[11] = x[11] + -1; goto L129;
L129: if (x[11] ≠ x[0]) goto L96;
        else goto L90;
}
```

`x[0]` можно было упростить!

Первая проекция Футамуры

Специализация интерпретатора по входной программе дает **скомпилированный вариант** этой программы:

```
1.  spec(int, p) → p' и p'(d) == int(p, d)
```

В примерах выше **накладные расходы** на интерпретацию **исчезли**.

Специализация интерпретатора рег. выражений конкретным выражением дала **код на C** для этого выражения.

Специализация симулятора RISC-V машинным кодом дала **код на C** для этого кода.

Вторая проекция Футамуры

Некоторые специализаторы обладают полезным свойством — **самоприменимостью**:

1. `spec(int, p) → p' и p'(d) == int(p, d)`
2. `spec(spec, int) → comp и comp(p) → p'`

Самоприменимость позволяет избежать накладных расходов на специализацию для генерации кода, получить **компилятор**.

Это и есть **генерирующее расширение**:

```
spec(spec, p) → gen и gen(d) → p'
```

Для примера с умножением:

```
spec(spec, mul) → mul_gen и mul_gen(10) → mul_10
```

Три проекции Футамуры

Еще одно самоприменение **дает компилятор компиляторов**:

```
1.  spec(int, p) → p' и p'(d) == int(p, d)
2.  spec(spec, int) → comp и comp(p) → p'
3.  spec(spec, spec) → cocomp и cocomp(int) == comp
```

Йошихико Футамура опубликовал статью в 1971-м году, где описал первые **две проекции**.

Термин “проекции Футамуры” ввел в научный оборот А.П. Ершов. Ершов независимо открыл **первую проекцию**.

Все **три проекции** Футамуры были описаны В.Ф. Турчиным в 1977-м году.

Futamura Y. Partial evaluation of computation process-an approach to a compilercompiler //Systems, computers, controls. – 1971. – Т. 2. – №. 5. – С. 45-50.
Рефал Б. его реализация на вычислительных машинах //М.: ЦНИИПИАС. – 1977.

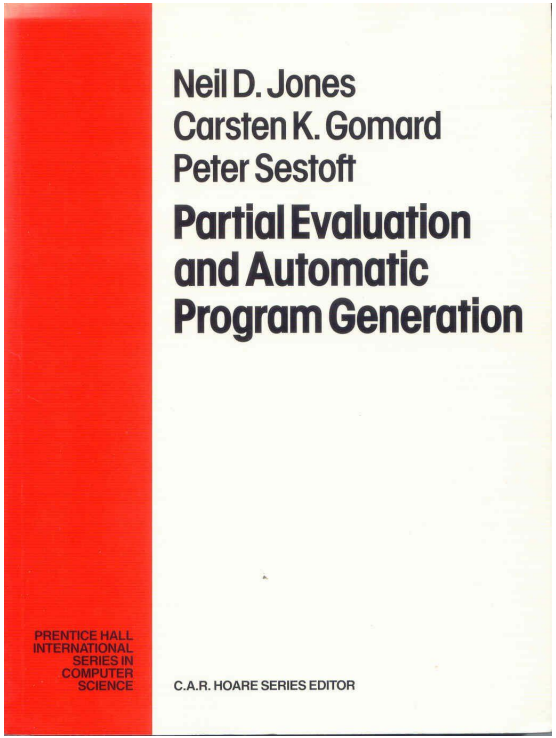
Смешанные вычисления и сегодня не получили широкого распространения

Принятый термин — **частичные вычисления** (partial evaluation).

- **Оффлайн-специализатор** использует предварительный **анализ периода связывания** (binding time analysis, BTA). Он нужен для установления доступных и задержанных участков программы.
- **Онлайн-специализатор** анализирует доступность объектов в процессе работы, без BTA (Truffle/Graal, AnyDSL).
- **Мета-трассировка** (meta-tracing) — специализация интерпретатора во время выполнения программы (PyPy).

Литература по частичным вычислениям

Классический учебник (1993 г.), где можно найти многочисленные ссылки на работы советских исследователей:



1. Смешанные вычисления.
2. **Рефал и методы его компиляции.**
3. Структурный синтез программ.

Метаязык Рефал (Валентин Федорович Турчин, 1966 г.)

Для **быстрого создания** реализаций **языков** программирования.

Язык функционального программирования, основан на **нормальных алгоритмах** Маркова.

Программа состоит из функций с **одним** аргументом.

Функции состоят из **упорядоченного** набора правил переписывания.

Используется мощный механизм **сопоставления с образцом**.

```
int fact(int n) {
  switch (n) {
    case 0: return 1;
    case 1: return 1;
    default: return n * fact(n - 1);
  }
}
```

```
<fact 0> = 1
<fact 1> = 1
<fact s.n> = <mul s.n <fact <sub s.n 1>>>
```

Турчин В. Ф. Метаязык для формального описания алгоритмических языков //Цифровая вычислительная техника и программирование: Сборник.– –М.: Сов. Радио. – 1966. – С. 116-124.

Транслятор с АЛГОЛа, написанный на языке РЕФАЛ


```
§10.1 К'ОТЩИД' (ЕI) ≤ А Е2 ≡ К'ОТЩИДУ'(ЕI)К'КЛАСС' ≤ А. Е2 .
§10.2 К'ОТЩИД' (ЕI)(ЕА)Е2 ≡ (К'СТ'ЕI .)
                              (К'ПОБВ'ЕА .)К'ПОБВ' Е2 . )
§10.3 К'ОТЩИД' (ЕI) ≡ ( К'СТ' ЕI . )

§11.1 К'ОТЩИДУ'(ЕI)Б ≤ А Е2 ≡ К'ОТЩИД'(ЕI ≤ А) Е2 .
§11.2 К'ОТЩИДУ'(ЕI)Ц ≤ А Е2 ≡ К'ОТЩИД'(ЕI ≤ А) Е2 .
§11.3 К'ОТЩИДУ'(ЕI) ≤ К ≤ А Е2 ≡
                    ( К'СТ' ЕI .) ≤ А К'ПОБВ' Е2 .

§12.1 К'ОТЩЧ' (ЕI) ≤ А Е2 ≡ К'ОТЩЧУ' (ЕI) К'КЛАСС' ≤ А . Е2 .
§12.2 К'ОТЩЧ' (ЕI) ≡ ( S = E | EI | )

§13.1 К'ОТЩЧУ'(ЕI)Ц ≤ А Е2 ≡ К'ОТЩЧ'(ЕI ≤ А ) Е2 .
§13.2 К'ОТЩЧУ'(ЕI)Ч ≤ А ≤ В Е2 ≡ К'ОТЩЧ'(ЕI ≤ А ≤ В) Е2 .
§13.3 К'ОТЩЧУ'(ЕI) ≤ К ≤ А Е2 ≡ ( S = E | EI | ) ≤ А К'ПОБВ' Е2 .
```

```
f(g(x))
```

записывалось в виде:

```
К'f' К'g' x . .
```

Турчин В. Ф. Транслятор с АЛГОЛа, написанный на языке РЕФАЛ //Сб. Труды ВКП-1. В. Процессоры с известных языков. К. – 1968. – С. 134-151.

Компилятор арифметического выражения в код стековой машины (Wasm, JVM, ...)

```
<comp (t.a '+' t.b)> = <comp t.a> <comp t.b> 'add'
<comp (t.a '-' t.b)> = <comp t.a> <comp t.b> 'sub'
<comp (t.a '*' t.b)> = <comp t.a> <comp t.b> 'mul'
<comp (t.a '/' t.b)> = <comp t.a> <comp t.b> 'div'
<comp s.x> = ('push' s.x)
```

```
<comp (2 '*' ('x' '-' 1))> →
('push' 2) ('push' 'x') ('push' 1) 'sub' 'mul'
```

Типы переменных в образцах:

```
s.символ    ::= одиночное значение
t.терм      ::= символ | (выражение)
e.выражение ::= терм*
```

Шаги компиляции в код стековой машины (1)


```
    <comp (t.a '+' t.b)> = <comp t.a> <comp t.b> 'add'
    <comp (t.a '-' t.b)> = <comp t.a> <comp t.b> 'sub'
✅  <comp (t.a '*' t.b)> = <comp t.a> <comp t.b> 'mul'
    <comp (t.a '/' t.b)> = <comp t.a> <comp t.b> 'div'
    <comp s.x> = ('push' s.x)
```

`<comp (2 '*' ('x' '-' 1))>` → `<comp 2> <comp ('x' '-' 1)> 'mul'`

`t.a=2 t.b=('x' '-' 1)`

```
s.символ    ::= одиночное значение
t.терм      ::= символ | (выражение)
e.выражение ::= терм*
```

Шаги компиляции в код стековой машины (2)


```
    <comp (t.a '+' t.b)> = <comp t.a> <comp t.b> 'add'
    <comp (t.a '-' t.b)> = <comp t.a> <comp t.b> 'sub'
    <comp (t.a '*' t.b)> = <comp t.a> <comp t.b> 'mul'
    <comp (t.a '/' t.b)> = <comp t.a> <comp t.b> 'div'
✅  <comp s.x> = ('push' s.x)
```

```
<comp 2> <comp ('x' '-' 1)> 'mul' → ('push' 2) <comp ('x' '-' 1)> 'mul'
```

```
s.x=2
```

```
s.символ    ::= одиночное значение
t.терм      ::= символ | (выражение)
e.выражение ::= терм*
```

Шаги компиляции в код стековой машины (3)


```
    <comp (t.a '+' t.b)> = <comp t.a> <comp t.b> 'add'
✅  <comp (t.a '-' t.b)> = <comp t.a> <comp t.b> 'sub'
    <comp (t.a '*' t.b)> = <comp t.a> <comp t.b> 'mul'
    <comp (t.a '/' t.b)> = <comp t.a> <comp t.b> 'div'
    <comp s.x> = ('push' s.x)
```

```
('push' 2) <comp ('x' '-' 1)> 'mul' → ('push' 2) <comp 'x'> <comp 1> 'sub' 'mul'
```

```
t.a='x' t.b=1
```

```
s.символ    ::= одиночное значение
t.терм      ::= символ | (выражение)
e.выражение ::= терм*
```

Шаги компиляции в код стековой машины (4)


```
    <comp (t.a '+' t.b)> = <comp t.a> <comp t.b> 'add'
    <comp (t.a '-' t.b)> = <comp t.a> <comp t.b> 'sub'
    <comp (t.a '*' t.b)> = <comp t.a> <comp t.b> 'mul'
    <comp (t.a '/' t.b)> = <comp t.a> <comp t.b> 'div'
✅  <comp s.x> = ('push' s.x)
```

```
('push' 2) <comp 'x'> <comp 1> 'sub' 'mul' →
('push' 2) ('push' 'x') <comp 1> 'sub' 'mul'
```

```
s.x='x'
```

```
s.символ    ::= одиночное значение
t.терм      ::= символ | (выражение)
e.выражение ::= терм*
```

Шаги компиляции в код стековой машины (5)


```
    <comp (t.a '+' t.b)> = <comp t.a> <comp t.b> 'add'
    <comp (t.a '-' t.b)> = <comp t.a> <comp t.b> 'sub'
    <comp (t.a '*' t.b)> = <comp t.a> <comp t.b> 'mul'
    <comp (t.a '/' t.b)> = <comp t.a> <comp t.b> 'div'
✅  <comp s.x> = ('push' s.x)
```

```
('push' 2) ('push' 'x') <comp 1> 'sub' 'mul' →
('push' 2) ('push' 'x') ('push' 1) 'sub' 'mul'
```

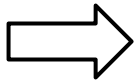
```
s.x=1
```

```
s.символ    ::= одиночное значение
t.терм      ::= символ | (выражение)
e.выражение ::= терм*
```

Компилятор линейного участка

Последовательность присваиваний (линейный участок) транслируется в код стековой машины.

```
(('n' '=' 10)
 ('s' '='
  (('n' '*' ('n' '+' 1)) '/' 2)))
```

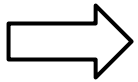


```
('push' 10)
('store' 'n')
('load' 'n')
('load' 'n')
('push' 1)
'add'
'mul'
('push' 2)
'div'
('store' 's')
```

Реализация компилятора линейного участка


```
<comp (t.a '+' t.b)> = <comp t.a> <comp t.b> 'add'
<comp (t.a '-' t.b)> = <comp t.a> <comp t.b> 'sub'
<comp (t.a '*' t.b)> = <comp t.a> <comp t.b> 'mul'
<comp (t.a '/' t.b)> = <comp t.a> <comp t.b> 'div'
<comp (s.var '=' t.exp) e.bb> = <comp t.exp> ('store' s.var) <comp e.bb>
<comp s.val> = (<op <type s.val>> s.val)
<comp _> = _
<op 'int'> = 'push'
<op 'str'> = 'load'
```

```
(('n' '=' 10)
 ('s' '='
  (('n' '*' ('n' '+' 1)) '/' 2)))
```



```
('push' 10)
('store' 'n')
('load' 'n')
('load' 'n')
('push' 1)
'add'
'mul'
('push' 2)
'div'
('store' 's')
```

```
s.символ    ::= одиночное значение
t.терм      ::= символ | (выражение)
e.выражение ::= терм*
```

Интерпретатор стекового кода (на основе ранее созданного компилятора)

```
<step 'add' (e.stk s.x s.y)> = (e.stk <add s.x s.y>)
<step 'sub' (e.stk s.x s.y)> = (e.stk <sub s.x s.y>)
<step 'mul' (e.stk s.x s.y)> = (e.stk <mul s.x s.y>)
<step 'div' (e.stk s.x s.y)> = (e.stk <div s.x s.y>)
<step ('push' s.x) (e.stk)> = (e.stk s.x)

<int (t.cmd e.bb) t.stk> = <int (e.bb) <step t.cmd t.stk>>
<int () (e.stk s.result)> = s.result

<sem e.bb> = <int (<comp e.bb>) ()>
```

```
<sem ((1 '+' 2) '*' 3)> → 9
```

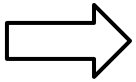
Интерпретатор линейного участка


```
<step 'add' (e.stk s.x s.y) t.env> = (e.stk <add s.x s.y>) t.env
<step 'sub' (e.stk s.x s.y) t.env> = (e.stk <sub s.x s.y>) t.env
<step 'mul' (e.stk s.x s.y) t.env> = (e.stk <mul s.x s.y>) t.env
<step 'div' (e.stk s.x s.y) t.env> = (e.stk <div s.x s.y>) t.env
<step ('push' s.x) (e.stk) t.env> = (e.stk s.x) t.env
<step ('load' s.var) (e.stk) t.env> = (e.stk <get t.env s.var>) t.env
<step ('store' s.var) (e.stk s.val) t.env> = (e.stk) <set t.env s.var s.val>

<int (t.cmd e.bb) t.stk t.env> = <int (e.bb) <step t.cmd t.stk t.env>>
<int () () t.env> = t.env

<sem e.bb> = <int (<comp e.bb>) ()>
```

```
(('n' '=' 10)
 ('s' '='
  (('n' '*' ('n' '+' 1)) '/' 2)))
```



```
(('n' 10) ('s' 55))
```

e-переменные являются “ленивыми”

Получить по ключу значение в ассоциативном списке:

```
<get (e.x (s.key t.val) e.y) s.key> = t.val
```

```
<get (('a' 1) ('b' 2) ('c' 3) ('d' 4)) 'e'>
```

```
1.  e.x= s.key=a s.val=1 e.y=(b 2) (c 3) (d 4)
2.  e.x=(a 1) s.key=b s.val=2 e.y=(c 3) (d 4)
3.  e.x=(a 1) (b 2) s.key=c s.val=3 e.y=(d 4)
4.  e.x=(a 1) (b 2) (c 3) s.key=d s.val=4 e.y=
5.  неуспех
```

В рег. выражении `e.x` примерно соответствует `(?P<x>.*?)`

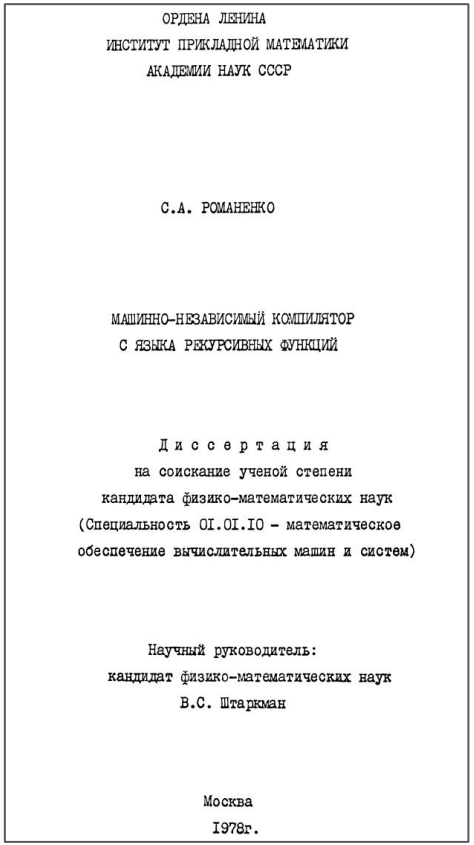
Оптимизирующий компилятор Рефала (Сергей Анатольевич Романенко, 1972 г.)

Один из **лучших** оптимизирующих **компиляторов функциональных языков** своего времени.

Написан на Рефале.

Компилирует код для **виртуальной Рефал-машины**.

На основе виртуальной Рефал-машины позже был разработан **аппаратный** Рефал-процессор (1985 г.).



ОРДЕНА ЛЕНИНА
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
АКАДЕМИИ НАУК СССР

С.А. РОМАНЕНКО

МАШИННО-НЕЗАВИСИМЫЙ КОМПИЛЯТОР
С ЯЗЫКА РЕКУРСИВНЫХ ФУНКЦИЙ

Диссертация
на соискание ученой степени
кандидата физико-математических наук
(Специальность 01.01.10 – математическое
обеспечение вычислительных машин и систем)

Научный руководитель:
кандидат физико-математических наук
В.С. Штаркман

Москва
1978г.

Смирнов В. К. и др. Аппаратная реализация языка Рефал в ИПМ им. МВ Келдыша //Препринты ИПМ им. МВ Келдыша. – 2003. – №. 99. – С. 1-21.

Компиляция левой части правила (1)


```
<f s.x e.y>
```

```
e.y, s.x = left_sym(h0)
```

Команда `left_sym` отщепляет слева от `h0` один элемент-символ. Остаток `h0` сохраняется в `e.y`.

h-переменная (hole) хранит пару (начало, конец) для фрагмента сопоставляемого выражения.

Компиляция левой части правила (2)


```
<f s.x e.y>
```

```
e.x, s.x = left_sym(h0)
```

```
<f e.y s.x>
```

```
e.y, s.x = right_sym(h0)
```

Команда `right_sym` отщепляет справа от `h0` один элемент-символ. Остаток `h0` сохраняется в `e.y`.

h-переменная (hole) хранит пару (начало, конец) для фрагмента сопоставляемого выражения.

Компиляция левой части правила (3)


```
<f s.x e.y>
```

```
e.x, s.x = left_sym(h0)
```

```
<f e.y s.x>
```

```
e.y, s.x = right_sym(h0)
```

```
<f s.x e.y s.x>
```

```
h1, s.x = left_sym(h0)
e.y = right_same(h1, s.x)
```

Команда `right_same` отщепляет справа от `h1` один элемент, совпадающий с `s.x`.

h-переменная (hole) хранит пару (начало, конец) для фрагмента сопоставляемого выражения.

Компиляция левой части правила (4)


```
<f s.x e.y>
```

```
e.x, s.x = left_sym(h0)
```

```
<f e.y s.x>
```

```
e.y, s.x = right_sym(h0)
```

```
<f s.x e.y s.x>
```

```
h1, s.x = left_sym(h0)
e.y = right_same(h1, s.x)
```

```
<comp (t.a '+' t.b)>
```

```
h1, h2 = left_list(h0)
h3, t.a = left_term(h2)
h5 = left_val(h3, '+')
h6, t.b = left_term(h5)
empty(h6)
empty(h1)
```

`left_list` отщепляет элемент-список, сохраняет в `h2` список, а в `h1` — остаток `h0`. `empty` проверяет фрагмент на пустоту.

Компиляция с открытыми e-переменными


```
<get (e.x (s.key t.val) e.y) s.key>
```

```
h1, h2 = left_list(h0)
h3, s.key = left_sym(h1)
empty(h3)
h5, e.x = open_exp(h2)
h5, e.x = extend_exp(h5, e.x)    ← Откат!
e.y, h8 = left_list(h5)
h9 = left_same(h8, s.key)
h10, t.val = left_term(h9)
empty(h10)
```

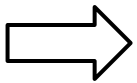
```
<uniq e.p1 s.x e.p2 s.x e.p3>
```

```
h1, e.p1 = open_exp(h0)
h1, e.p1 = extend_exp(h1, e.p1) ← Откат!
h3, s.x = left_sym(h1)
h5, e.p2 = open_exp(h3)
h5, e.p2 = extend_exp(h5, e.p2) ← Откат!
e.p3 = left_same(h5, s.x)
```

При неуспехе сопоставления происходит откат до предыдущей открытой e-переменной. Делается попытка ее удлинить.

Компиляция функции


```
<comp t.a '+' t.b> = ...
<comp t.a '-' t.b> = ...
```



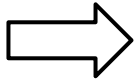
```
    alt(L8)
    h1, h2 = left_term(h0)
    h3 = left_val(h1, '+')
    h4, h5 = left_term(h3)
    empty(h4)
    ... правая часть ...
    return
L8:
    h1, h2 = left_term(h0)
    h3 = left_val(h1, '-')
    h4, h5 = left_term(h3)
    empty(h4)
    ... правая часть ...
    return
```

Команда alt сохраняет адрес следующего правила для перехода по неуспеху.

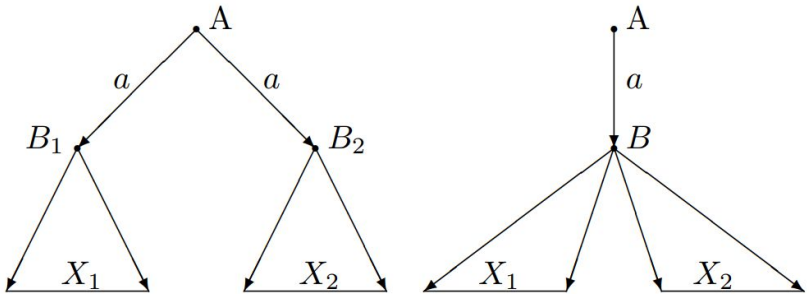
Романенко С. А. Машинно-независимый компилятор с языка рекурсивных функций : дис. – Диссертация на соискание уч. степени к. ф.-м. н., М, 1978.

Объединение совпадающих частей правил


```
  alt(L8)
  h1, h2 = left_term(h0)
  h3 = left_val(h1, '+')
  h4, h5 = left_term(h3)
  empty(h4)
  ... правая часть ...
  return
L8:
  h1, h2 = left_term(h0)
  h3 = left_val(h1, '-')
  h4, h5 = left_term(h3)
  empty(h4)
  ... правая часть ...
  return
```



```
  h1, h2 = left_term(h0)
  alt(L8)
  h3 = left_val(h1, '+')
  h4, h5 = left_term(h3)
  empty(h4)
  ... правая часть ...
  return
L8:
  h3 = left_val(h1, '-')
  h4, h5 = left_term(h3)
  empty(h4)
  ... правая часть ...
  return
```



Источник рис. на пред. слайде.

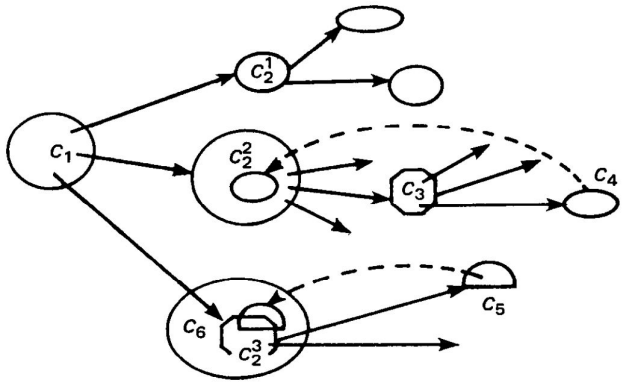
Специализация программ в Рефале: суперкомпиляция (В.Ф. Турчин, 1977 г.)

Преимущественно для **функциональных** языков (Рефал, экспериментальные языки).

Использует поливариантный **онлайн**-режим, без BTA.

Граф обобщенных состояний вычислений в **символическом** виде.

Новые состояния могут нетривиальным образом сводиться к уже созданным (**частные случаи, обобщения**).



😕Потенциально **мощнее** известных методов частичных вычислений, но используется только в **академических** проектах.

Базисный Рефал и его реализация на вычислительных машинах. М.: ЦНИПИАСС, 1977. - 258 с.
Turchin V. F. The concept of a supercompiler //ACM Transactions on Programming Languages and Systems (TOPLAS). – 1986. – Т. 8. – №. 3. – С. 292-325.

Современные системы для быстрого построения реализаций языков

- PLT Redex (Racket).
- Spoofax.
- oMeta.

Это наборы метаязыков. Сопоставление с образцом на уровне Рефала, но **без оптимизаций** компилятора Рефала.

В языках ML-семейства (SML, OCaml, Haskell) поддерживается компиляция сопоставления с образцом, но для **менее выразительных** форм образцов.

1. Смешанные вычисления.
2. Рефал и методы его компиляции.
3. **Структурный синтез программ.**

Дедуктивный синтез программ

Спецификация: в виде **теоремы**.

Способ синтеза: автоматическое **доказательство** теоремы с **извлечением** программы из доказательства.

😕**Сложность** создания **спецификации** может превосходить **сложность** написания самой **программы**.
😕Задача дедуктивного синтеза, в общем случае, является **NP-полной.**

Структурный синтез программ (Энн Харальдович Тыугу, 1970 г.)

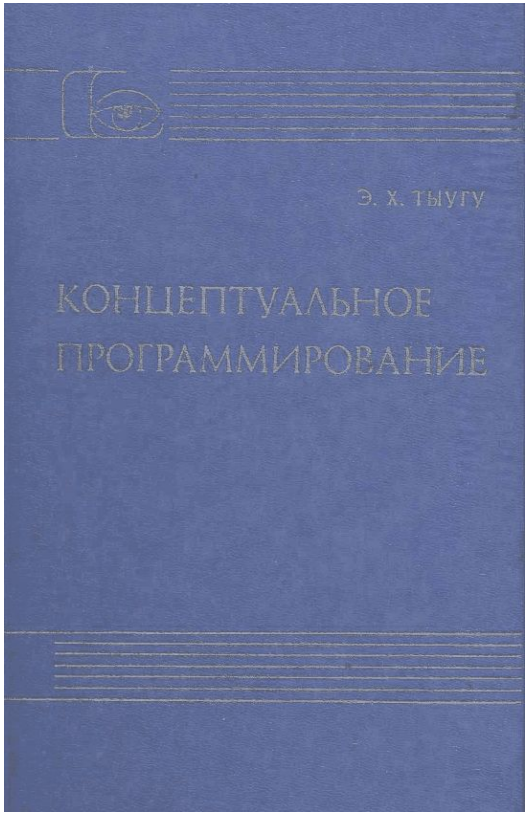
Один из старейших вариантов **дедуктивного синтеза** программ.

Это синтез из функций-**черных ящиков**. Корректность синтеза — с точностью до корректности функций.

Известны только **входные** и **выходные** связи функций. Эти связи **постоянны**:

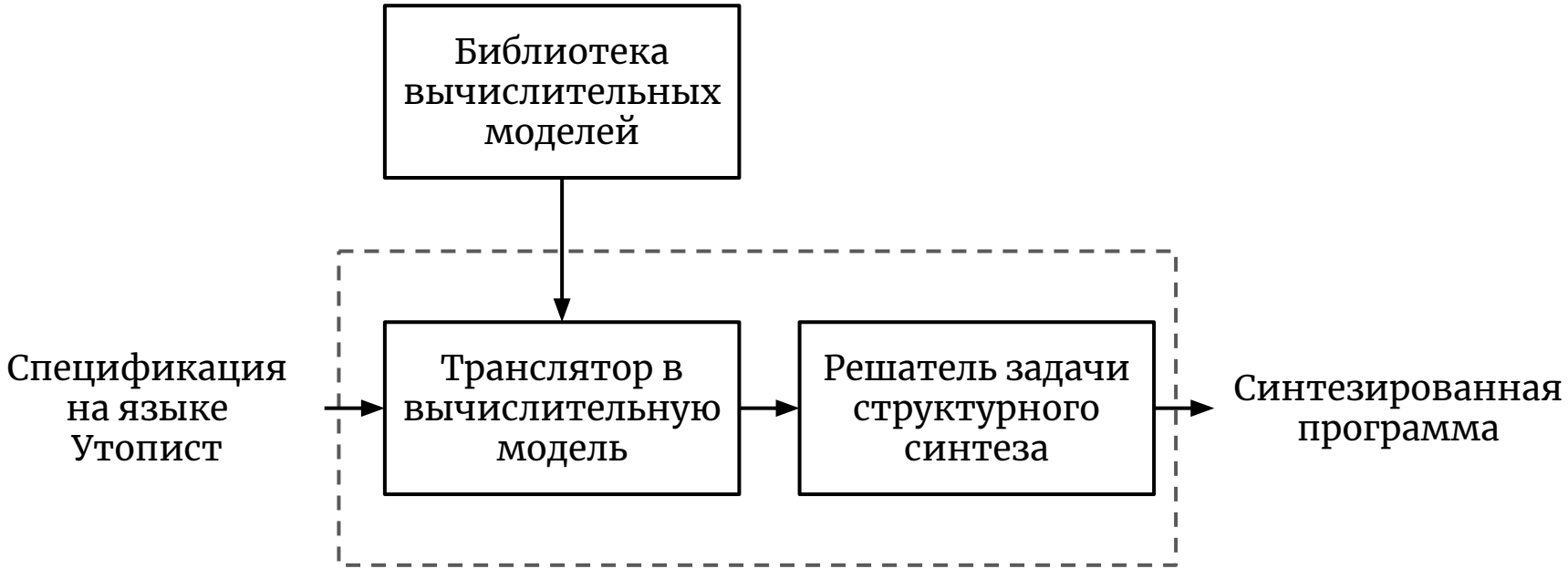
$$y_1, \ldots, y_n = f(x_1, \ldots, x_m)$$

Имея $x_1, \ldots, x_m$ можно получить $y_1, \ldots, y_n$.



Тыугу Э. Х. Решение задач на вычислительных моделях //Журнал вычислительной математики и математической физики. – 1970. – Т. 10. – №. 3. – С. 716-733.

Система ПРИЗ
(Программа Решения Инженерных Задач)

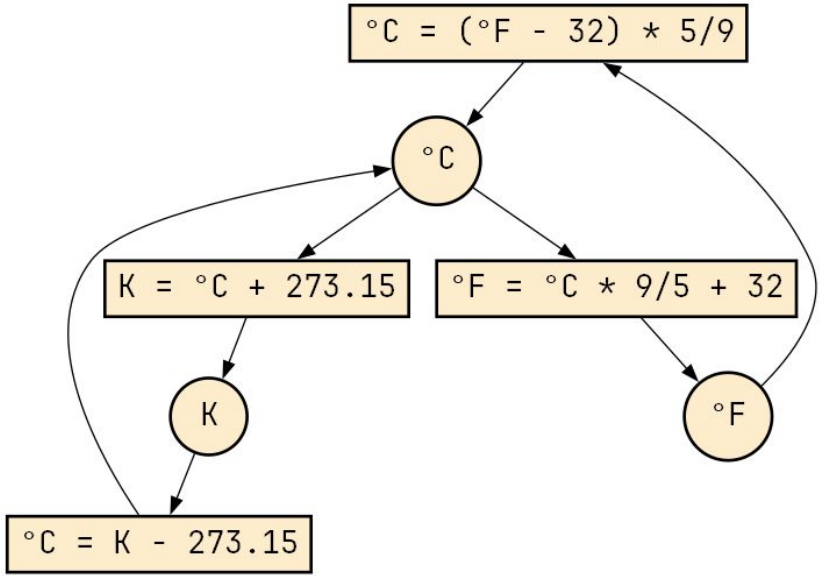


Вычислительная модель температурных шкал


```
°C = (°F - 32) * 5/9
°C = K - 273.15
```

вычислить °C **по** °F.
```
°C = (°F - 32) * 5/9
```

вычислить °F **по** K.
```
°C = K - 273.15
°F = °C * 9/5 + 32
```

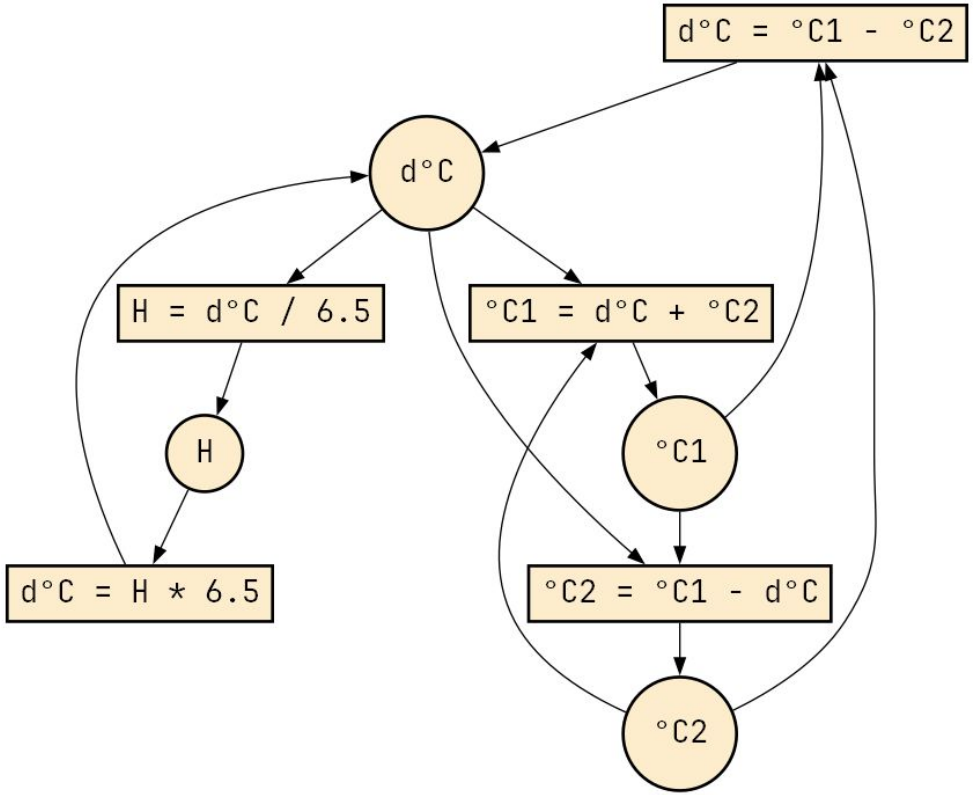


Вычислительная модель температуры и высоты


```
d°C = °C1 - °C2
H = d°C / 6.5
```

вычислить °C2 **по** °C1, H.
```
d°C = H * 6.5
°C2 = °C1 - d°C
```

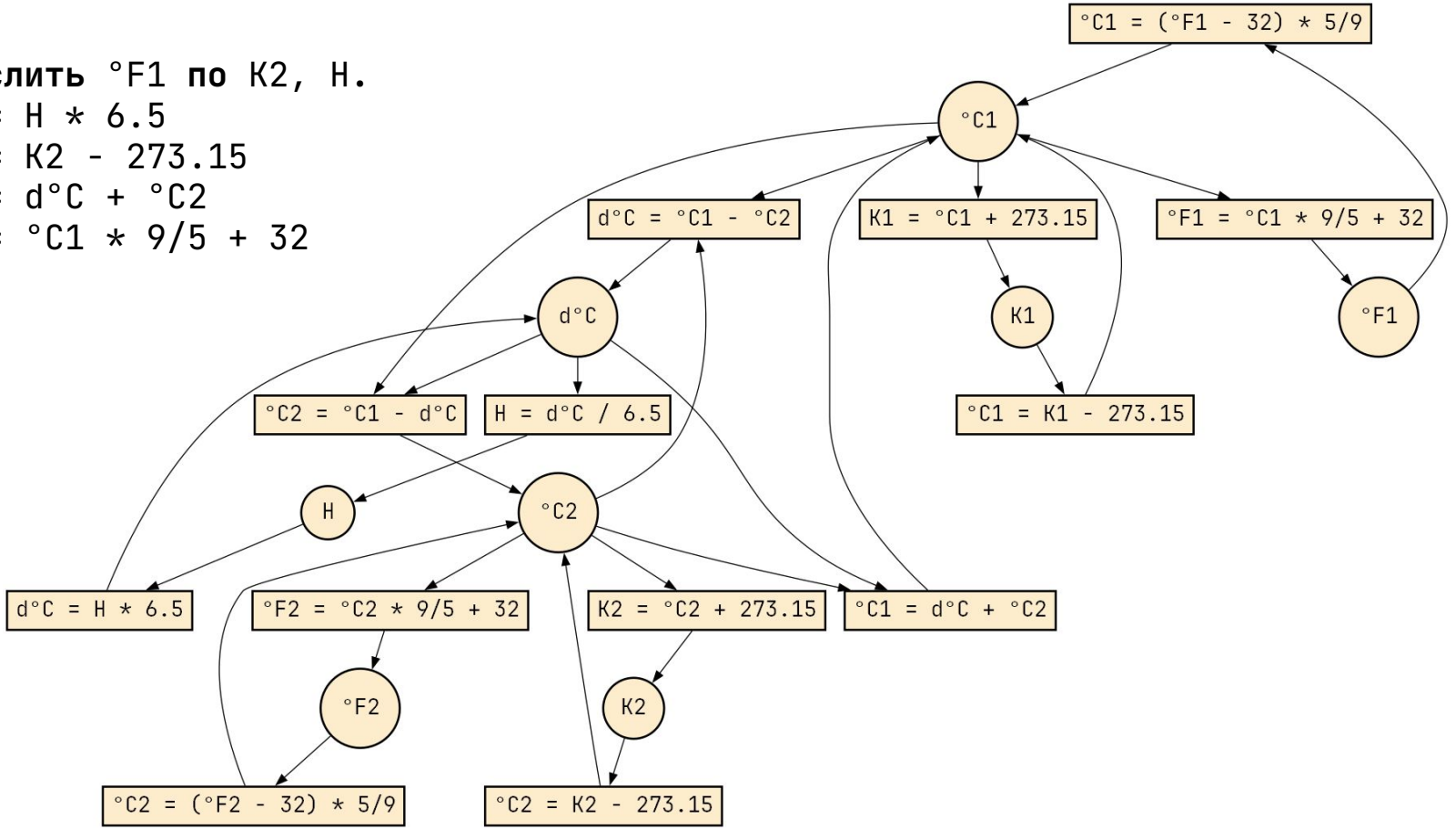
вычислить H **по** °C1, °C2.
```
d°C = °C1 - °C2
H = d°C / 6.5
```



Объединение вычислительных моделей

вычислить °F1 **по** K2, H.

```
d°C = H * 6.5
°C2 = K2 - 273.15
°C1 = d°C + °C2
°F1 = °C1 * 9/5 + 32
```



Алгоритм структурного синтеза


```
Синтезировать(Модель: граф, Дано: множество, Получить: множество) {
    Известно то, что в Дано.
    Пока Получить не содержится в Известно {
        Взять из Модели функцию ƒ.
            Если все аргументы ƒ Известны, а результаты — нет {
                Добавить к Известным новые результаты ƒ.
                Добавить ƒ к синтезируемой программе.
                Более ƒ не применять.
            }
    }

    // Чистка программы
    Живые — все переменные из Получить.
    Цикл по программе в обратном порядке {
        Если среди результатов функции ƒ есть Живые {
            Добавить ƒ в новую программу.
            Убрать из Живых результаты ƒ и добавить к Живым аргументы ƒ.
        }
    }
}
```

Алгоритм структурного синтеза: обсуждение

Структурный синтез **выразительнее** подходов, на основе топологической сортировки (dataflow-программирование, утилита make).

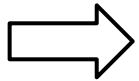
При этом структурный синтез в более серьезной реализации имеет **сложность O(n)**, в отличие, к примеру, от NP-полного SAT-решателя.

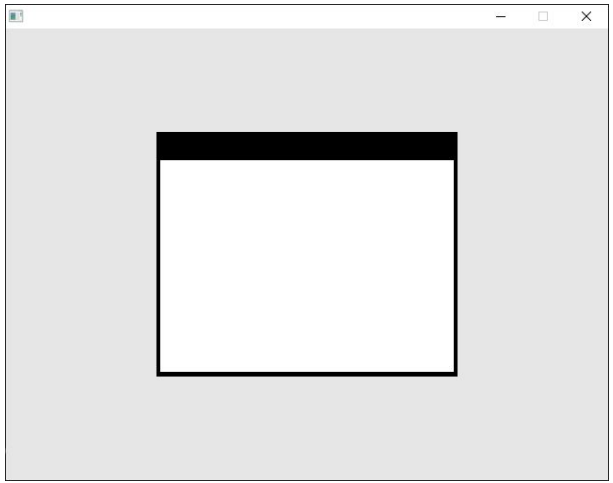
Возможен структурный синтез с использованием **функций высшего порядка** (подзадач). Этот вариант применяется, в том числе, для синтеза **ветвлений** и **циклов**.

Диковский «Решение в линейное время алгоритмических проблем связанных с синтезом ациклических программ», Программирование, 1985

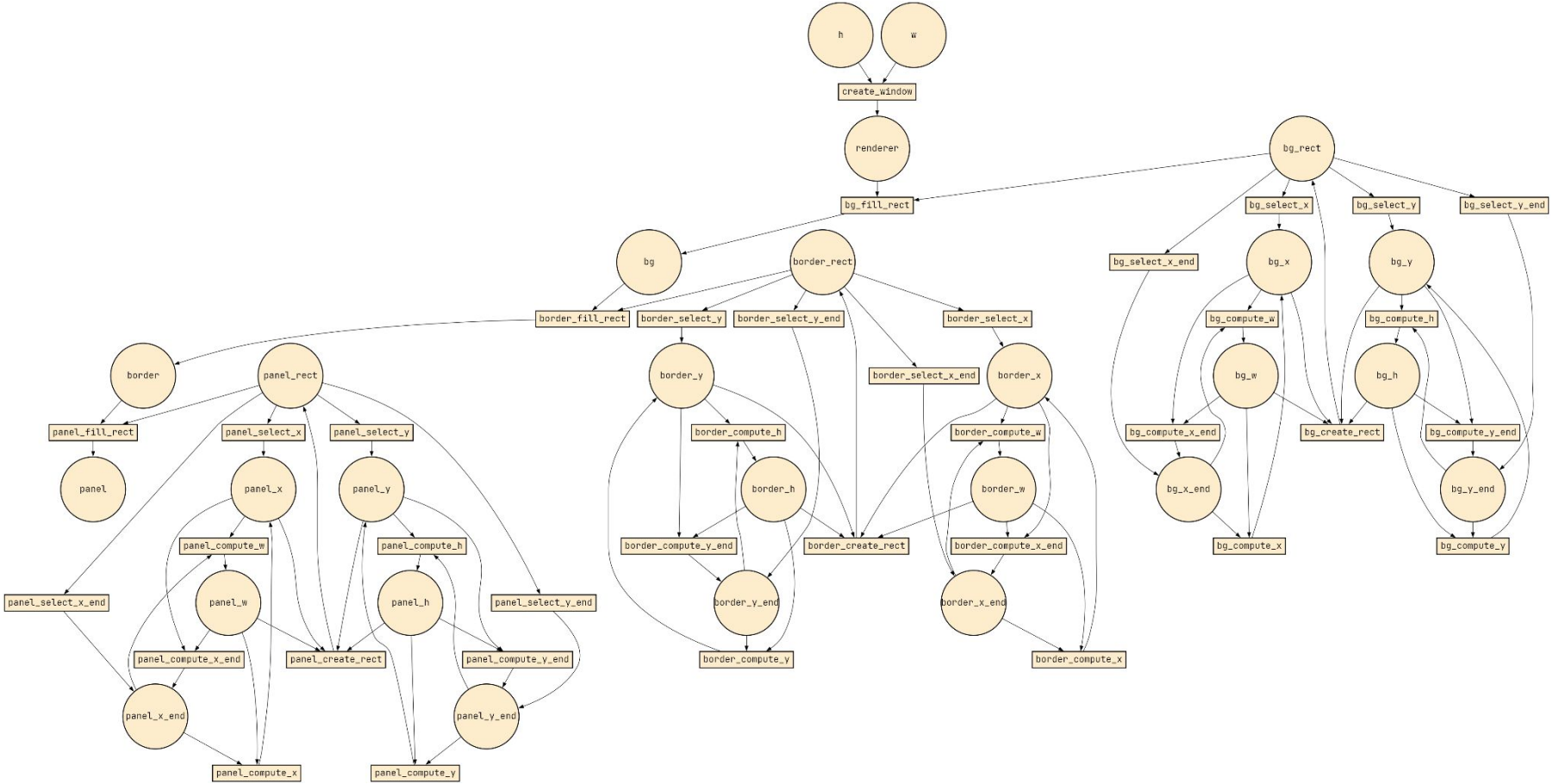
Синтез GUI-кода (SDL2) по визуальному описанию (набросок)


```
{
  "demo": {
    "type": "window",
    "w": 640,
    "h": 480
  },
  "bg": {
    "type": "rect",
    "x": 0,
    "y": 0,
    "w": 640,
    "h": 480,
    "color": "e6e6e6"
  },
  "border": {
    "type": "rect",
    "x": 160,
    "y": 110,
    "x_end": 480,
    "y_end": 370,
    "color": "000000"
  },
  "panel": {
    "type": "rect",
    "x": 164,
    "y": 140,
    "x_end": 476,
    "y_end": 365,
    "color": "ffffff"
  }
}
```





Вычислительная модель для GUI-кода



Фрагмент синтезированного кода API SDL2


```
int border_w = 480 - 160;
int border_h = 370 - 110;
SDL_Rect bg_rect = {0, 0, 640, 480};
int panel_w = 476 - 164;
SDL_Rect border_rect = {160, 110, border_w, border_h};
SDL_Init(SDL_INIT_VIDEO);
SDL_Window *window;
SDL_Renderer *renderer;
SDL_CreateWindowAndRenderer(640, 480, SDL_WINDOW_SHOWN, &window, &renderer);
SDL_SetRenderDrawColor(renderer, 230, 230, 230, 255);
SDL_RenderFillRect(renderer, &bg_rect);
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
SDL_RenderFillRect(renderer, &border_rect);
int panel_h = 365 - 140;
SDL_Rect panel_rect = {164, 140, panel_w, panel_h};
SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
SDL_RenderFillRect(renderer, &panel_rect);
```

Синтез программ сегодня

Дедуктивные системы синтеза наименее распространены. Пример: система SPIRAL для синтеза реализаций алгоритмов цифровой обработки сигналов.

Супероптимизаторы — начинают применяться в компиляторах (Souper для LLVM).

Индуктивный синтез — самый распространенный (Flash fill в Excel). Неполная спецификация — набор примеров (тестов). Корректность не гарантируется.

Структурный синтез сегодня, в основном, забыт. Найдена лишь одна система: CoCoViLa (https://cocovila.github.io/).

“Новое — хорошо забытое старое”

В **старых** работах содержатся забытые оригинальные идеи, новые взгляды на общепринятые вещи.

Я нашел много полезного в старых **англоязычных** текстах по информатике. Вдохновили на поиск: Bret Victor, Alan Kay, Joe Armstrong и другие.

Оказалось, что **советские** источники заслуживают не меньшего внимания.

В **трех докладах** невозможно описать все работы советской компиляторной школы. Они еще **ждут** своего исследователя!

Спасибо за внимание!