МОСКВА
11-12 мая 2012

Application Developer Days
/*Программисты всех платформ, общайтесь!*/

Как написать JIT компилятор

Андрей Аксенов

Sphinx

Начнем… с конца

- Парсеры, bison, flex – это просто
- *Все* варианты рантайма – это нетяжело
  - Байткод, дерево, и даже и машкод
- Свои DSL – это выгодно
- *Ничего* про Java C2 не будет ;)
  - Времени на килодиссер маловато

А теперь чуть подробнее

- Два слова про парсеры в целом
- Два слова про flex, bison
- Два слова про варианты рантайма
- Два слова про DSL как *потребность*
- Тупой пример имени Красной Нити
  - Строчный калькулятор!

Делай-раз, про парсеры в целом (flex+bison 101+ftw)

Калькулятор

- Eval (“2+2”) == ?
- Eval (“2+a*3”) == ?
- Eval (“a = 2; b = 3; print 2+a*b;”) == ?

Калькулятор

- Eval (“2+2”) == ?
- Eval (“2+a*3”) == ?
- Eval (“a = 2; b = 3; print 2+a*b;”) == ?

- Для простоты ограничимся 1 формулой
- “Сложный” случай тупо… нуднее

Мой первый калькулятор!

```
const char * sExpr = "2+a*3";

void Parse ( const char * p )
{
    while ( isspace(*p) ) p++;
    const char * sTok = p;
    if ( isdigit(*p) )
        ...
```

Problem?

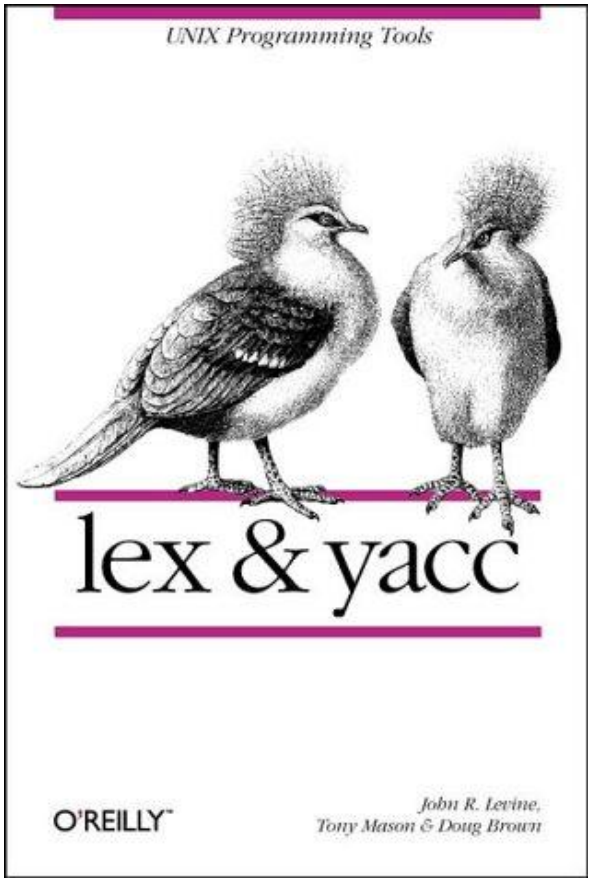
- Работает отлично, между прочим!
- Но – для *очень* простых парсеров
  - **option = value1 [, value2 [, value3 […]]]**
  - 2 + a*3 - sin(b^4<<5)
  - SELECT a, b*2 FROM myindex …

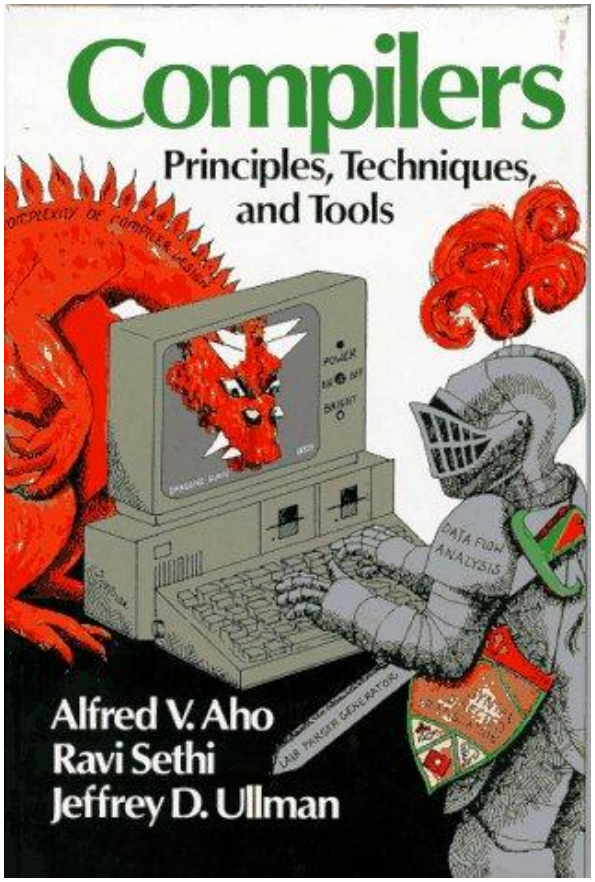
Problem?

- Для *просто* простых – уже сложновато
  - option = value1 [, value2 [, value3 […]]]
  - **2 + a*3 - sin(b^4<<5)**
  - **SELECT a, b*2 FROM myindex …**
- Получается много глупого кода
- (Очень) тяжело поддерживать

Solution!

- Решение, понятно, давно придумано





Разделяем и властвуем!

- Было – все перепутано
- Стало – три отдельные фазы
  - Лексический разбор
  - Синтаксический разбор
  - Действия

Лексический разбор

- Вход – “2+a*3”
- Выход – отдельные токены
  - 2, +, a, *, 3
- Токены – типизированные полиморфы!
  - type = T_INT, value = 2
  - type = T_OP, op = +

Синтаксический разбор

- Вход – поток *токенов*
- Выход –
  - Или успешное натягивание потока на заданные правила синтаксиса
  - Или ошибка разбора
- Заодно – исполнение действий!

Действия

- Просто код (вашей) программы
- Вход – аргументы совпавшего правила
- Выход – желанные сайд-эффекты

Пример, даешь калькулятор!

- Лексер, смертельно упрощенный:

```
%%
“SIN”          { return T_SIN; }
[0-9]+         { return T_NUMBER; }
[a-zA-Z]+      { return T_VARIABLE; }
%%
```

Пример, даешь калькулятор!

- Лексер, добавляем операторы:

```
"+"        { return '+'; }
"-"        { return '-'; }
"/"        { return '/'; }
"*"        { return '*'; }
```

Пример, даешь калькулятор!

- Лексер, исправляем переменные:

~~[a-zA-Z]+ { return T_VARIABLE; }~~

```
[a-zA-Z][a-zA-Z0-9]+  { return
    T_VARIABLE; }
```

Пример, даешь калькулятор!

- Парсер, объявляем токены:

```
%token T_NUMBER
%token T_VARIABLE
```

Пример, даешь калькулятор!

- Парсер, объявляем приоритеты:

```
%left '+' '-'
%left '*' '/'
%%
```

Пример, даешь калькулятор!

- Парсер, самое главное: **правила**!

```
expr: item
    | expr '+' expr | expr '-' expr
    | expr '*' expr | expr '/' expr
    | T_SIN '(' expr ')'
    | '(' expr ')' ;
```

Пример, даешь калькулятор!

- Парсер, top-down уточнение:

```
item:
    T_NUMBER
    | T_VARIABLE;
%%
```

И вот *весь* наш калькулятор

```
%%
[0-9]+ { return T_NUMBER; }
[a-zA-Z]+ { return T_VARIABLE; }
“+” { return ‘+’; }
“-” { return ‘-’; }
“/” { return ‘/’; }
“*” { return ‘*’; }
%%
```

```
%token TOKEN_NUMBER
%token TOKEN_VARIABLE
%left '+' '-‘
%left '*' '/‘
%%
expr:
    item
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '(' expr ')‘
    ;
item:
    TOKEN_NUMBER
    | TOKEN_VARIABLE;
%%
```

Что мы пропустили? ;)

Сделать что-нибудь!!!

- Это был скелет

- Лексер: типа мало, надо значение
- Парсер: правила мало, надо действие

Что-нибудь в лексере

- Тип токена YYSTYPE, по умолчанию int
  - Не путать с *номером* токена, всегда int!
  - Либо %union, %token <field> etc в парсере
  - Либо руками typedef снаружи
- Совсем рабочий кусок лексера:

```
[0-9]+        { lvalp->m_iValue = atoi ( yytext );
                return TOKEN_NUMBER; }
```

Что-нибудь в парсере

- Правила как бы трансформируют “токены” в другие “токены”
  - На самом деле, "термы"
  - Правила как в BNF
  - Каждая селедка - рыба!
- На выходе – последний мега-терм

Что-нибудь в парсере

```
%token <m_iValue> T_INT
%token <m_sName> T_VARIABLE
%token <m_pExpr> expr
...
    | expr '+' expr {
        $$ = CreateOpNode ( OP_ADD, $1, $2 ); }
    | T_INT { $$ = CreateConstNode ( $1 ); }
```

Боевой вариант, *столько же* LOC

```
%%
[0-9]+ { …; return T_NUMBER; }
[a-zA-Z]+ { …; return T_VARIABLE; }
“+” { return ‘+’; }
“-” { return ‘-’; }
“/” { return ‘/’; }
“*” { return ‘*’; }
%%
```

```
%token …
%left '+' '-‘
%left '*' '/‘
%%
expr:
      item { … }
      | expr '+' expr { … }
      | expr '-' expr { … }
      | expr '*' expr { … }
      | expr '/' expr { … }
      | '(' expr ')' { …}
      ;
item:
      TOKEN_NUMBER { …}
      | TOKEN_VARIABLE { …}
      ;
%%
```

Итого-раз

Не так страшен уасс

- Парсер вручную – да, адово сложно (*)
- Парсер автоматом – просто!!!
  - Автоматизируем 2 из 3 штук
  - Нетяжело сделать лексер на flex
  - Нетяжело сделать парсер на bison
  - И останется сделать только **мясо**

Почему именно flex/bison?

- Классика (lex/yacc * GNU = flex/bison)
- “Любу знают все”
- **Все еще** покрывает **кучу** задач
  - *Если* начнет жать, antlr, lemon, accent…
- Учитесь, оно того стоит
  - Лучше день потерять!

Делай-два, про рантайм

Варианты рантайма

- Определяют “как будем считать”
- Определяют “во что будем разбирать”
- Иногда рантайма совсем ***нет***
- Иногда рантайм таки есть
  - Как сделать?
  - Байткод (оно же VM), дерево, машкод

Когда рантайма нет?

- Типично, при *разовых* акциях
- Например, разобрать конфиг
- Например, один раз (!) вычислить

- Делаем дела **прямо в action-ах**
- Результат в момент окончания разбора

Когда рантайма нет

```
// parser
expr:
  expr '+' expr { $$ = $1 + $3; }
  | expr '-' expr { $$ = $1 - $3; }
...
// application
float Result = yyparse();
```

Байткод, общее

- В чем отличие от машкода?
- Тоже набор инструкций, однако
  - Упрощенный
  - Портабельный
  - Чаще стековый (JVM, NET)
- Привет, обратная Польская нотация!

Байткод, пример данных

- Вход
  - 2+3*a
- (Вариант) разбора
  - 3 a * 2 +
- Байткод
  - load_const 3; load_var a; **op_multiply**; …

Байткод, пример парсера

```
// parser
expr:
  expr '+' expr {
     dCodes.Append ( $1 );
     dCodes.Append ( $3 );
     dCodes.Append ( OP_ADD );
  }
  | ...
```

Байткод, пример VM

```
Vector<float> dStack;
for ( int i=0; i<dCodes.Length(); i++ )
    switch ( dCodes[i].m_Opcode )
{
    case LOAD_CONST:
        dStack.Push ( dCodes[i].m_fValue ); break;
    case LOAD_VAR:
        dStack.Push ( GetVar ( dCodes[i].m_sName ) );
        break;
   case OP_MUL:
        dStack.Push ( dStack.Pop() * dStack.Pop() );
        break;
   ...
```

Байткод, снова общее

- Радость: очень понятная модель (RPN)
- Радость: упрощен и портабелен
- Радость: простые вирт-машинки
- Плохо: не влобно ложится на регистры
- Плохо: лучше отдельно оптимизить

Дерево, общее

- **Abstract Syntax Tree**
- Почти оно, с “небольшим” довеском
  - AST – только представление
  - Нужно – еще и исполнение
- Итого, деревьев по факту может оказаться и **два**

Дерево, пример парсера

```
// parser
expr:
  expr '+' expr {
     $$ = new NodeAdd ( $1, $3 );
  }
  | ...
```

Дерево, пример “VM”

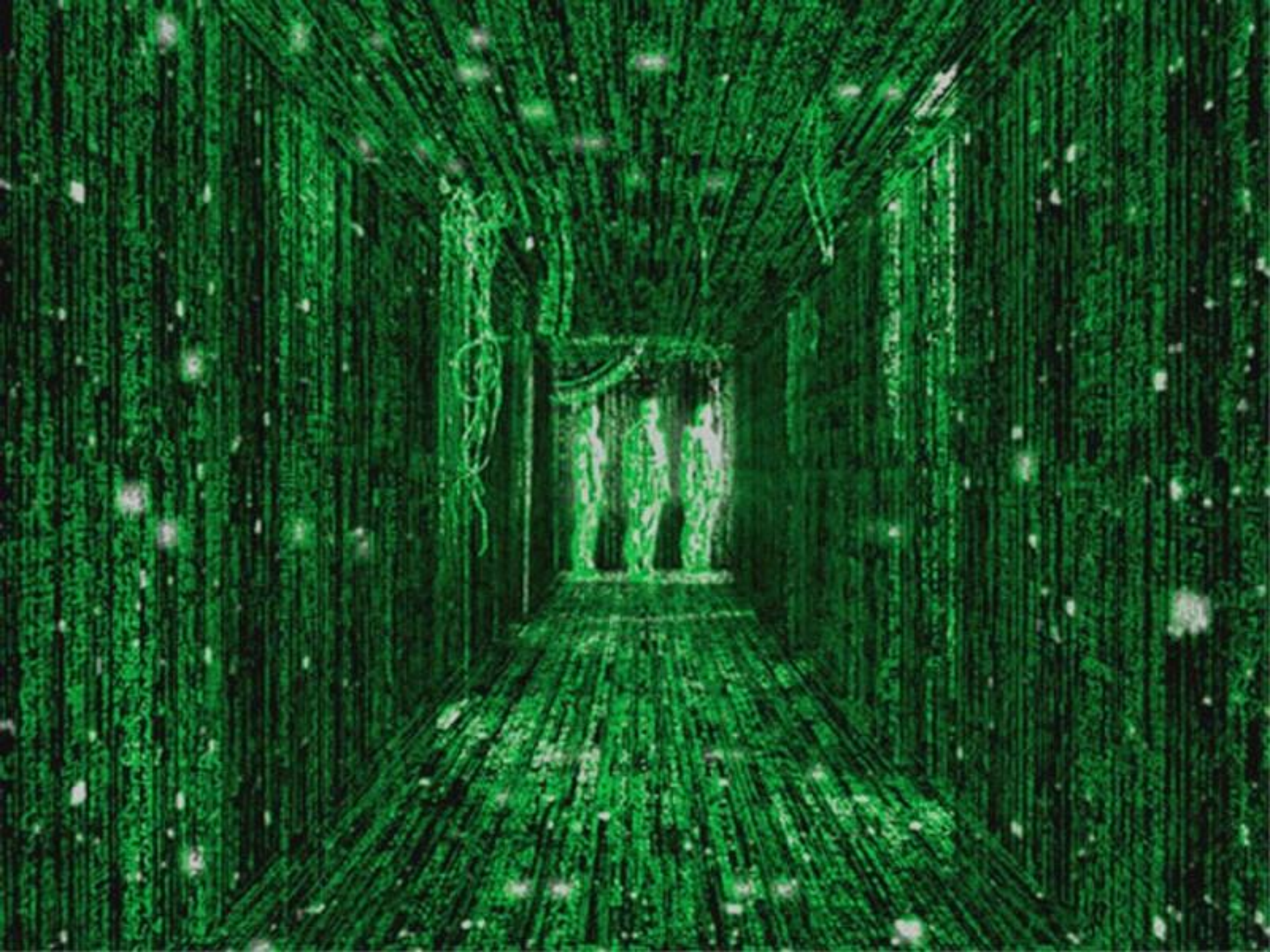
```
virtual float NodeConst::Eval()
{
    return m_Value;
}

virtual float iNodeAdd::Eval()
{
    return m_pArg1->Eval() + m_pArg2->Eval();
}

// ...
```

Дерево, снова общее

- Радость: тоже понятная модель (AST)
- Радость: изоморфно (!) байт-коду
- Радость: удобно трансформировать
- Непонятно: операции размазаны
- Плохо: обходы дерева…
- Радость для калькулятора: **БЫСТРЕЕ**



```
(gdb) disas /m main
     Dump of assembler code for function main:
     5       {
        0x08048330 <+0>:     push   %ebp
        0x08048331 <+1>:     mov    %esp,%ebp
        0x08048333 <+3>:     sub    $0x8,%esp
        0x08048336 <+6>:     and    $0xfffffff0,%esp
        0x08048339 <+9>:     sub    $0x10,%esp

     6         printf ("Hello.\n");
     => 0x0804833c <+12>:    movl   $0x8048440,(%esp)
        0x08048343 <+19>:    call   0x8048284 <puts@plt>

     7         return 0;
     8       }
        0x08048348 <+24>:    mov    $0x0,%eax
        0x0804834d <+29>:    leave
        0x0804834e <+30>:    ret

     End of assembler dump.
```

Машкод, общее

- Радость: родной для железа
- Радость: предельная скорость
- Плохо: сложнее (?) генерировать
- Плохо: регистровая машина
- Плохо: нужны (?) оптимизации

Машкод, фокусы

- Про калькулятор и не только
- Раз. FPU и есть стековая машина ;)
- Два. Конечный стек легко и явно мапится на регистры
- Три. Бесконечный стек чуть сложнее мапится на регистры и память

Машкод, пример генератора

```
// assemble mul
if ( tNode.m_iToken=='*' )
{
    CreateNative ( tNode.m_iLeft, uAttrType, pRes );
    CreateNative ( tNode.m_iRight, uAttrType, pRes );
    *pRes->m_pCurCode++ = 0xDE; // fmulp st(1),st(0)
    *pRes->m_pCurCode++ = 0xC9;
    return;
}
```

Машкод, а исполнять-то как?!

- В три строки!
- Неизбежное, указатель на функцию

```
typedef float ( * NativeEval_fn )
    ( void * pArg );
NativeEval_fn pFn =
    ( NativeEval_fn ) m_pCode;
return pFn ( pArg );
```

Машкод, а исполнять-то как-2?!

- Еще бывает DEP и его братья ;)
- Linux

```
mprotect ( addr, len, PROT_EXEC );
```

- Windows

```
VirtualProtect ( …, PAGE_EXECUTE );
```

- До кучи: еще бывает dlopen, dlclose…

Итого про машкод

- Разрушаем мифы!
- Прототип про калькулятор поверх AST
  - **150 строк, 1 вечер**
  - Без учета AST (ну, второй вечер)
- **95% производительности** натива (?!)
- Портабельность? x86, SSE2+ победили

Итого про рантайм

- Разрушаем мифы!
- Вот, три классических варианта
- Вот, каждый вполне применим
- Вот, каждый вполне нетяжел
- Никому не верьте, особенно мне ;)

Делай-три, про радость DSL

…радость в DSL есть.

Я НИ ФИГА НЕ ПОНЯЛ ТОЧКА ЖПГ!

Вопросы? Ответы? Резюме? ;)

shodan@sphinxsearch.com