Лекция 26
БГТУ, ФИТ, ПОИТ, 3 семестр, Языки программирования

Генерация кода в ассемблер

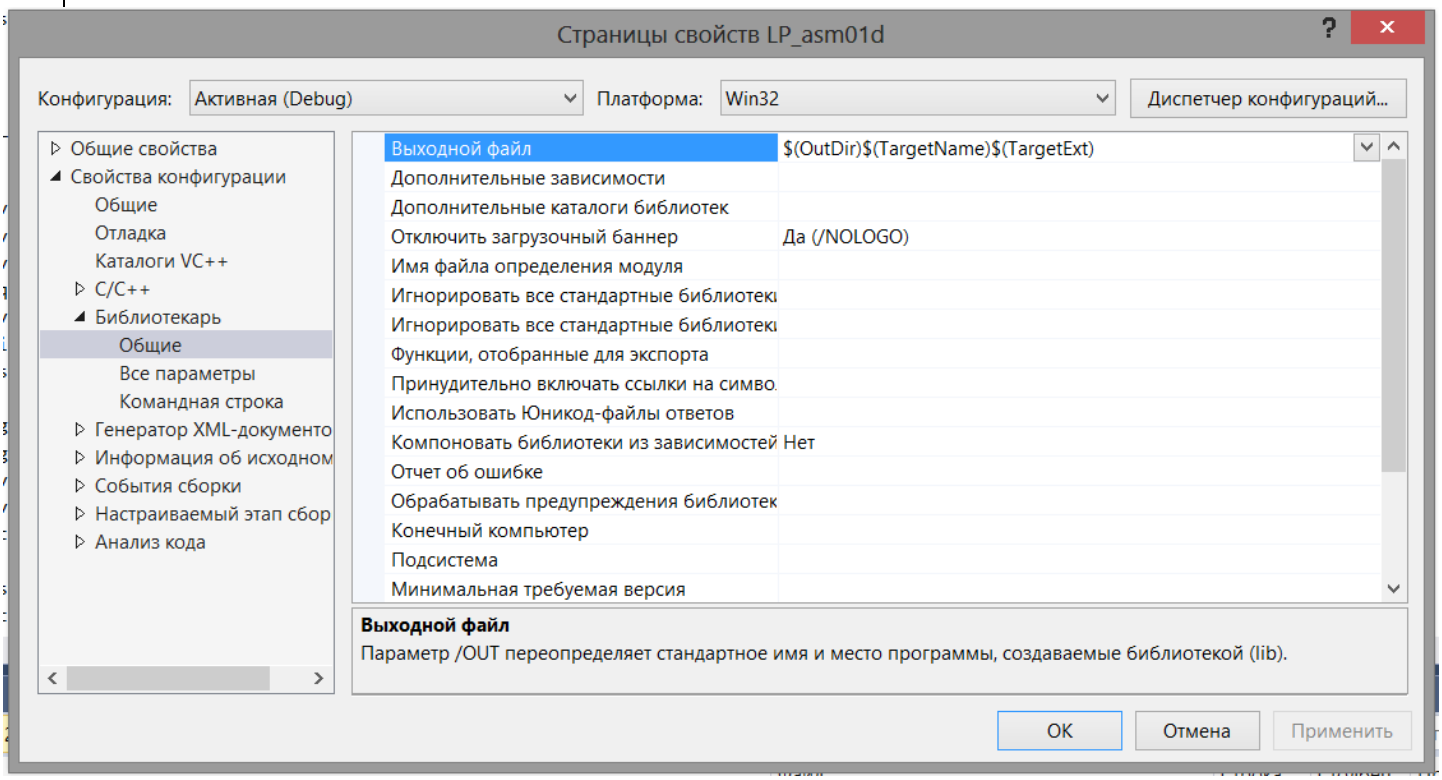
1. Объединение процедур, расположенных в разных модулях

1.1 Вызов функции на ассемблере из ассемблера

Создание статической библиотеки в Visual Studio на языке ассемблер.

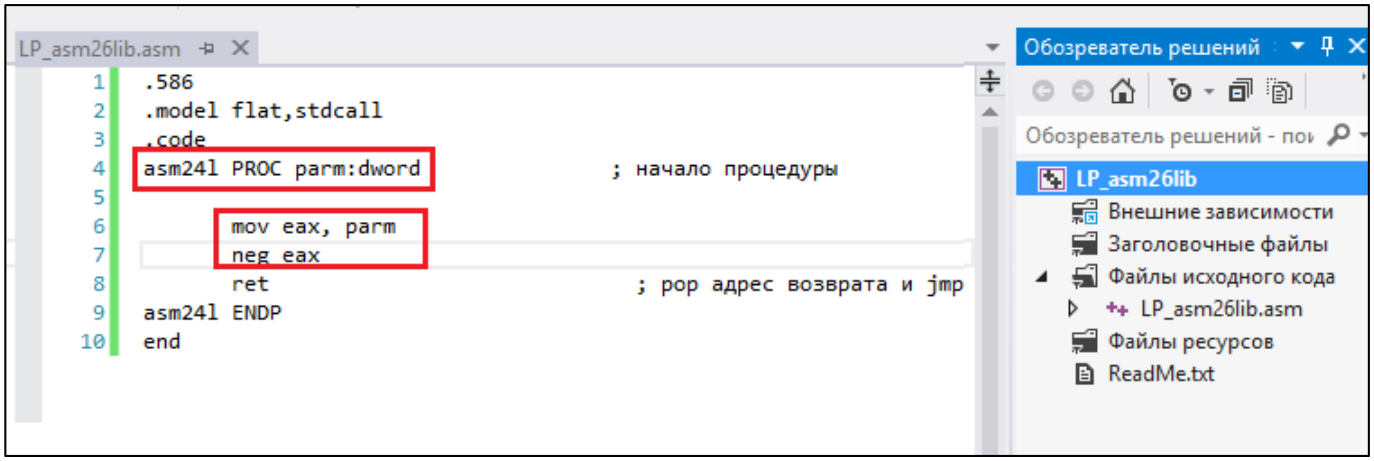
На странице ***Свойств проекта*** определяем ***тип конфигурации*** - **Статическая библиотека**.

> В разделе ***Библиотекарь -> общие*** можно переопределить местоположение, имя и расширение создаваемой библиотеки.
>
> В разделе ***Библиотекарь -> командная строка*** отображается текущее значение параметра **/OUT**.

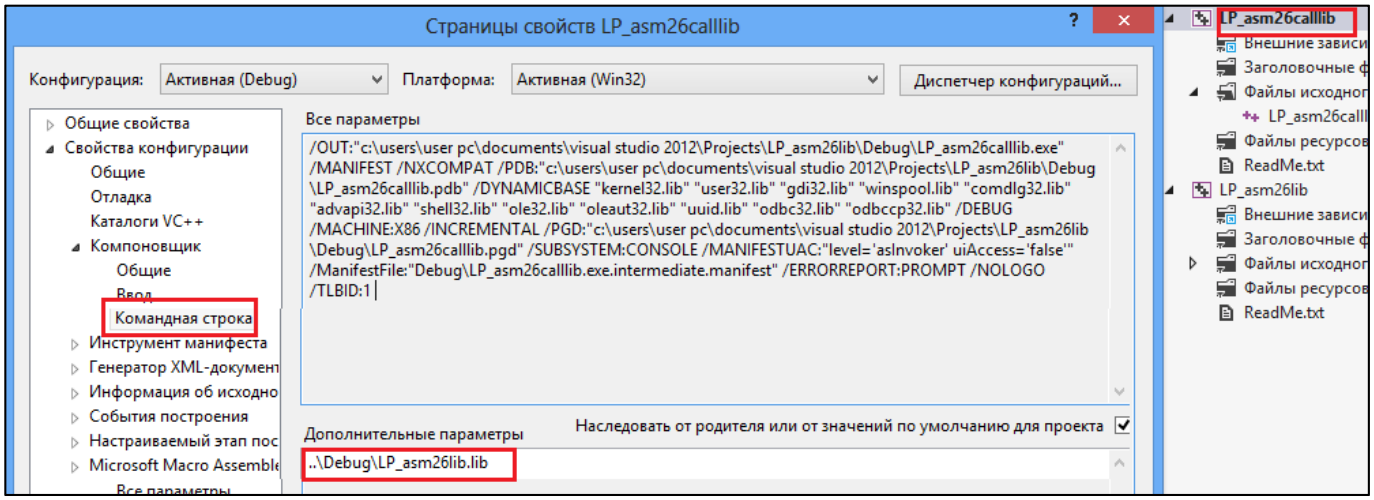


После построения проекта в папке проекта **Debug** будет размещен файл статической библиотеки (.lib).
В журнале (<имя>.log) проекта фиксируется ход выполнения сборки проекта.
Файл статической библиотеки создается утилитой **LIB**.

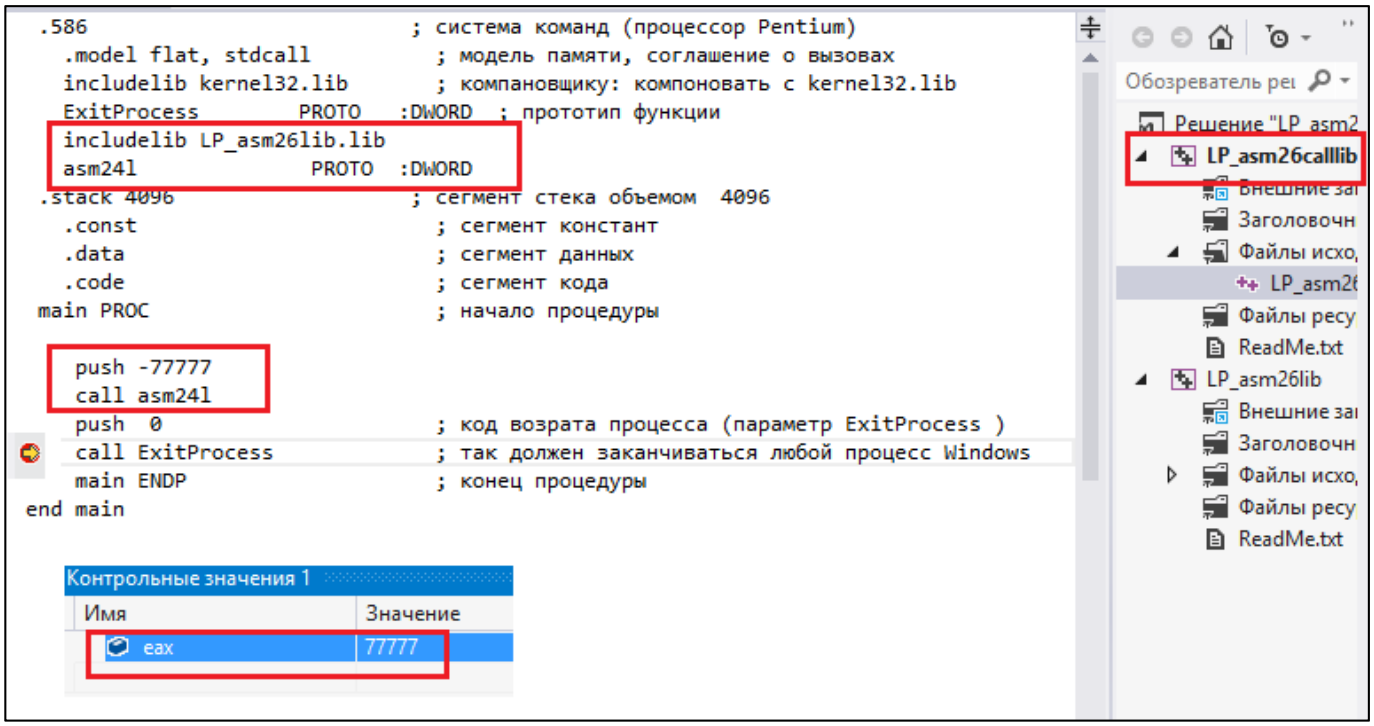
Процедура статической библиотеки:



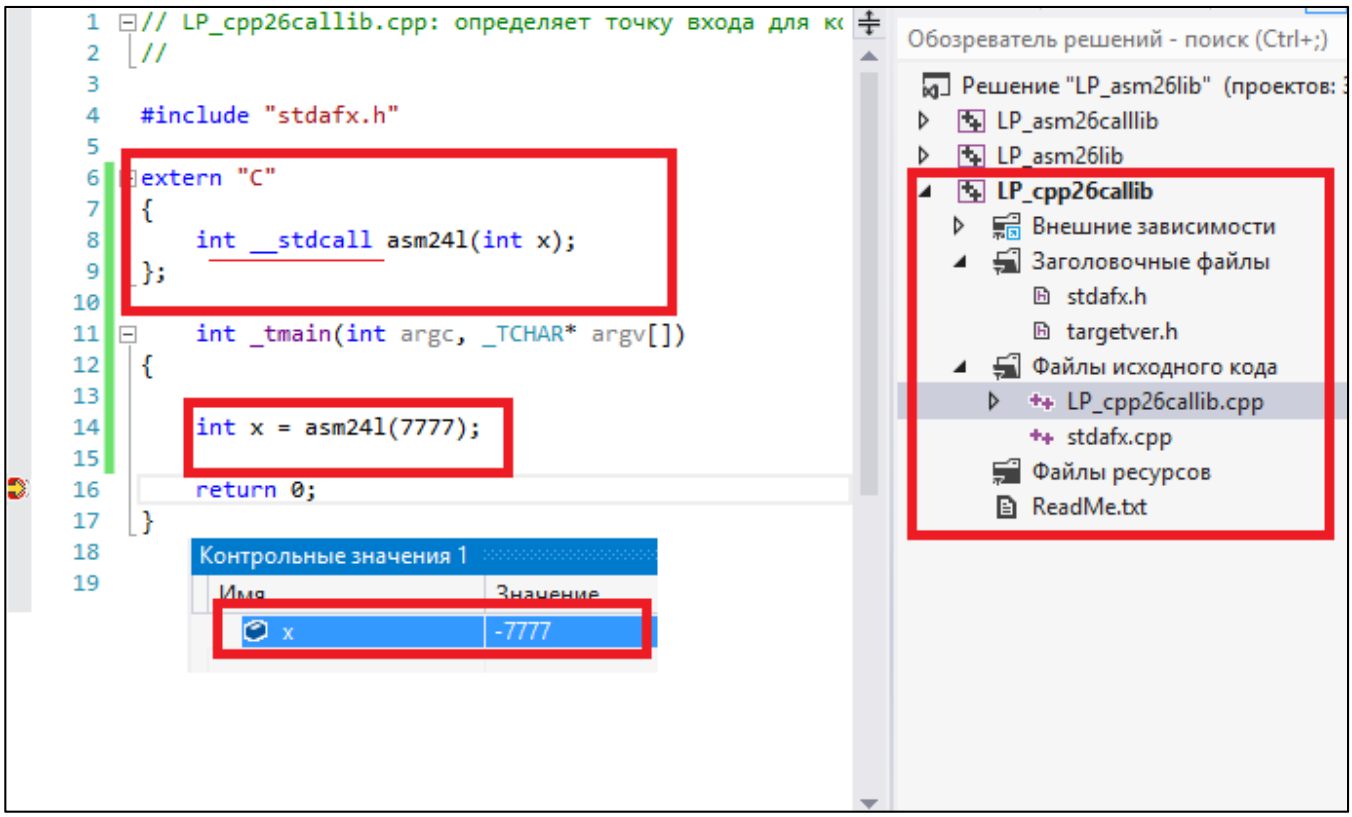
Подключение статической библиотеки:



Программа на языке ассемблер вызывает функцию из статической библиотеки:



1.2Вызов функции на ассемблере из C++



Сигнатура процедуры (функции) — это имя функции, тип возвращаемого значения и список аргументов с указанием порядка их следования и типов.

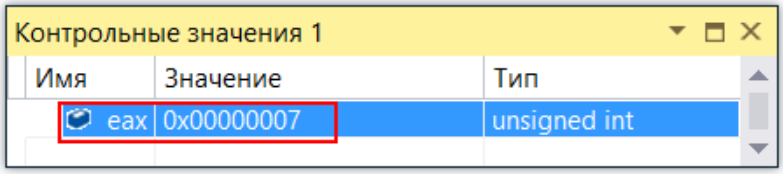
Символы, произведённые C++ компилятором, декорированы.
Символы, произведённые C компилятором, — без изменения, как в исходном коде.

Чтобы обойти декорирование в языке C++, при объявлении и определении функций используется спецификатор **extern "C"**.

1.3Вызов функции на C++ из ассемблера:

```
extern "C"
{
    int cneg(int parm)
    {
        return -parm;
    }
}
```

```
.586; система команд(процессор Pentium)
.model flat, stdcall; модель памяти, соглашение о вызовах

includelib kernel32.lib; компановщику: компоновать с kernel32
ExitProcess PROTO : DWORD; прототип функции

EXTRN cneg: proc

.stack 4096; выделение стека объёмом 4 мегабайта

.const; константы
.data
.code
main PROC; точка входа main
push -7
call cneg
push 0
call ExitProcess; завершение процесса Windows
main ENDP; конец процедуры
end main; конец модуля main
```



Процедура — именованная, правильным образом оформленная группа команд, которая объявляется один раз и может многократно вызываться по имени в любом месте программы.

MASM:

EXTRN <имя> — объявление внешнего имени по отношению к данному модулю.

PROC и ENDP — начало, конец процедуры.

call <ИмяПроцедуры> — команда вызова процедуры.

ret <число> — команда возврата управления вызывающей программе,

где <число> — количество байт, удаляемых из стека при возврате из процедуры (необязательный параметр).

Объединение процедур, расположенных в разных модулях

Каждый модуль должен сообщать транслятору, что некоторый объект (процедура, переменная) должен быть видимым вне этого модуля.
Транслятор также должен знать, что некоторые объекты определены вне данного модуля.
Все внешние ссылки в объединяемых модулях разрешаются на этапе ***компоновки***.

Организация интерфейса с процедурой

Для передачи аргументов в языке ассемблера существуют следующие способы:

- через регистры;
- через общую область памяти;
- через стек;
- с помощью директив extern и public.

Передача аргументов через стек

Вызывающая процедура заносит в стек параметры и передает управление вызываемой процедуре. При передаче управления процедуре в вершину стека поверх параметров автоматически записывается 4 байта с адресом возврата в вызывающую программу.

Стек обслуживается тремя регистрами:

- ESS - указатель дна стека (начала сегмента стека);
- ESP - указатель вершины стека;
- EBP - указатель базы.

Регистры ESS и ESP указывают на дно и вершину стека соответственно.
Регистр EBP используется для произвольного доступа к данным в стеке.

push ebp mov ebp, esp	Инициализация регистра EBP в процедуре (пролог процедуры)
Восстановление стека	Эпилог процедуры

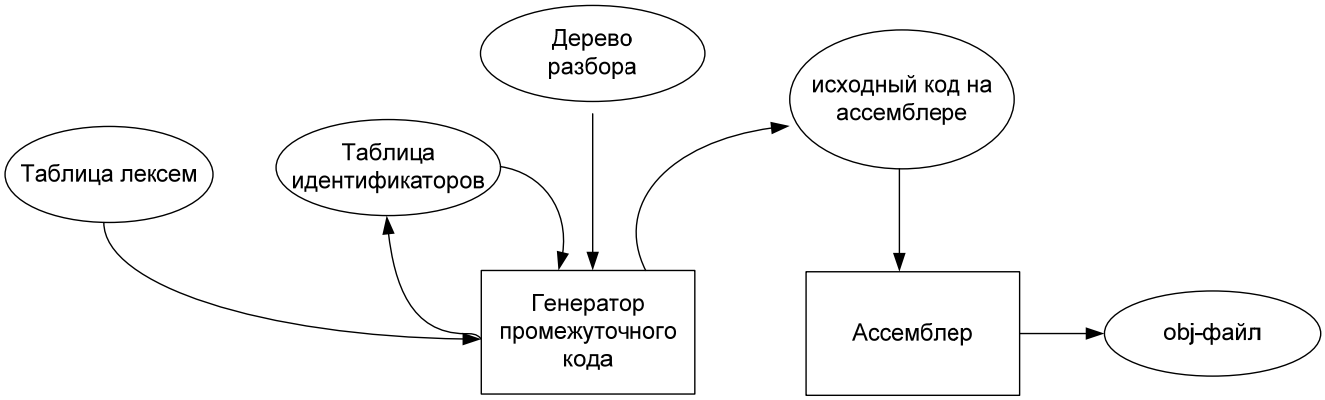
Восстановление стека:

```
add  esp, <длина_параметров_в_байтах>      // соглашение cdecl
ret       <длина_параметров_в_байтах>      // соглашение stdcall
```

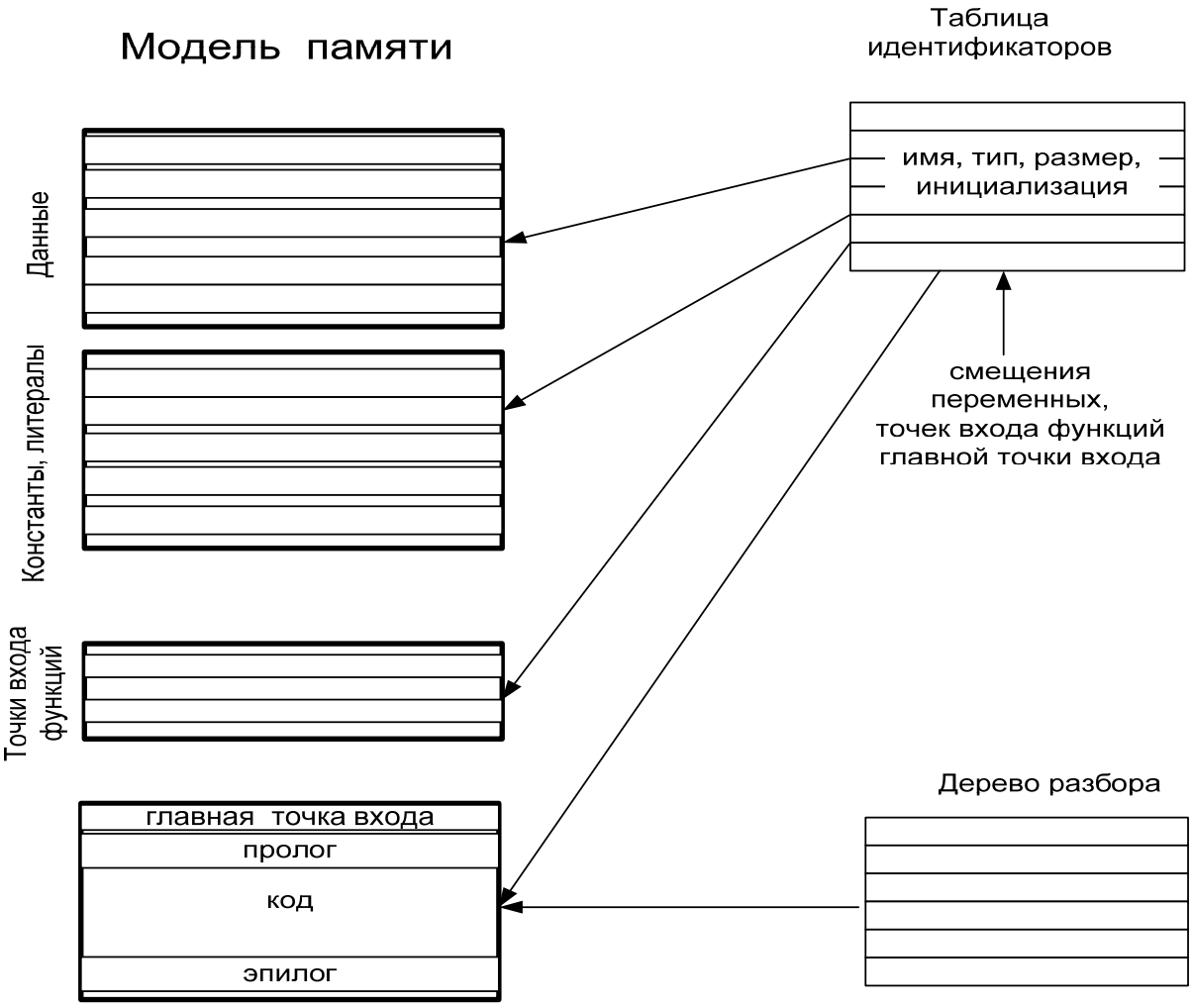
В программах, написанных на языке ассемблера, используется соглашение о вызовах stdcall.

2. Генерация исходного ассемблерного кода

2.1 Подход к генерации кода по дереву разбора с использованием стека.



2.2 Модель памяти SVV-2015



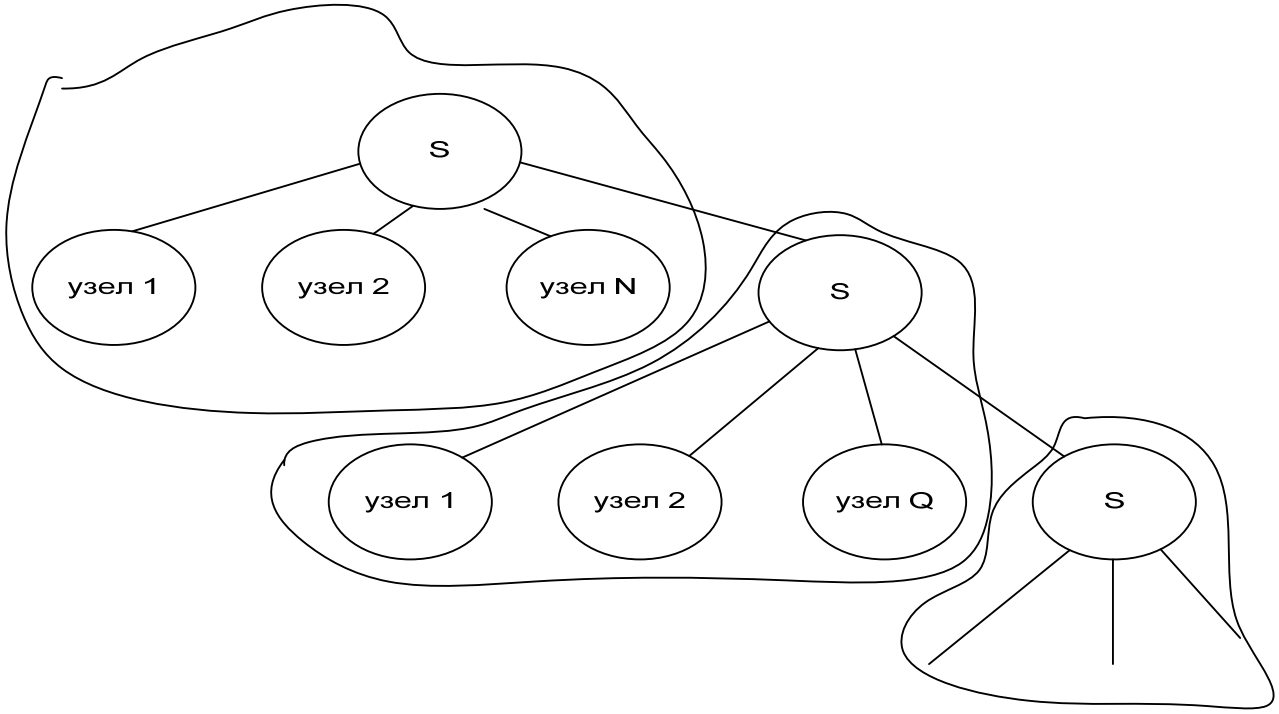
2.3 Дерево разбора

Дерево разбора:

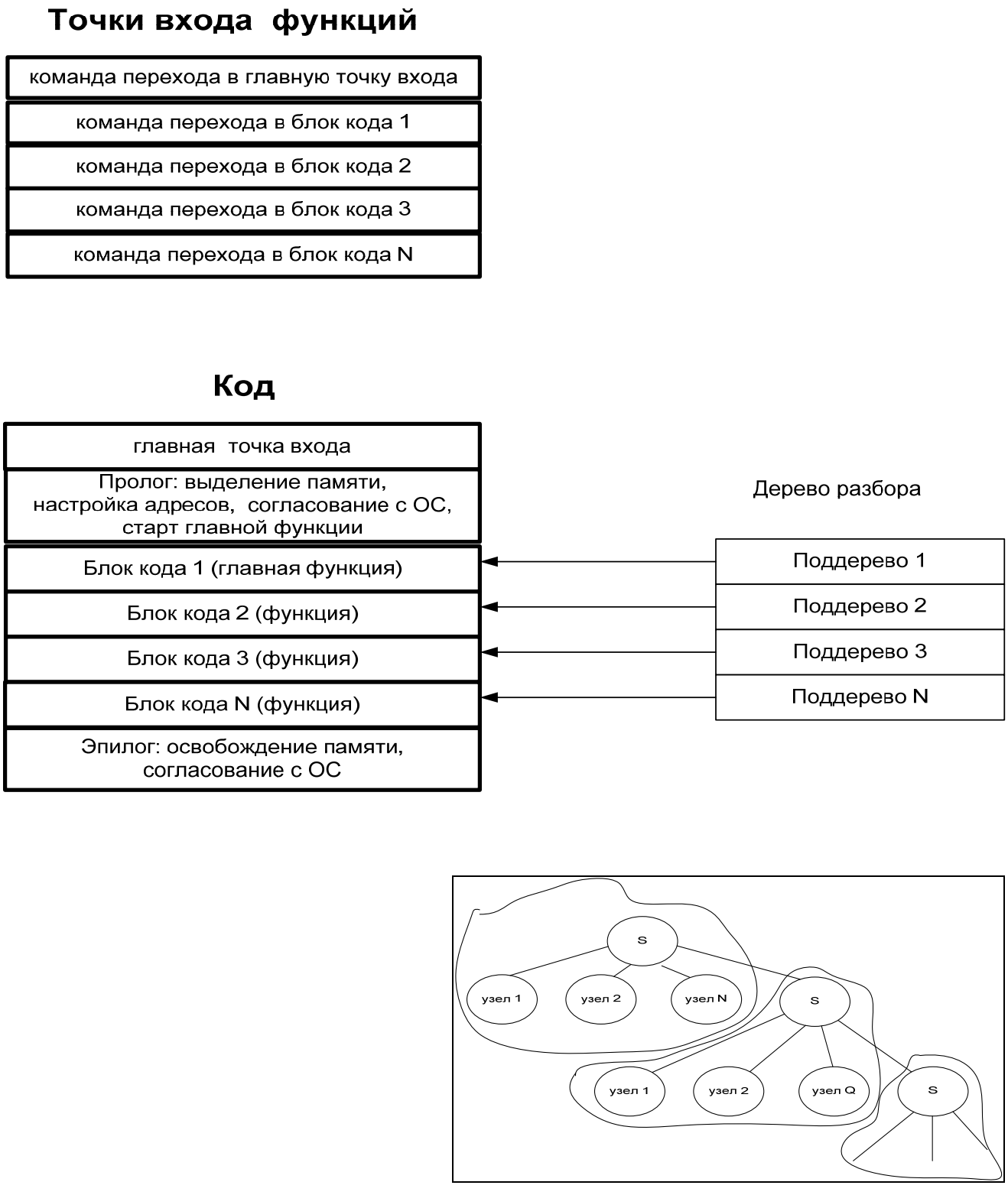
S – стартовый символ.

Каждый узел описывает функцию.

Каждой функции соответствует блок кода.



2.4 Модель памяти SVV-2015: точки входа функций и код

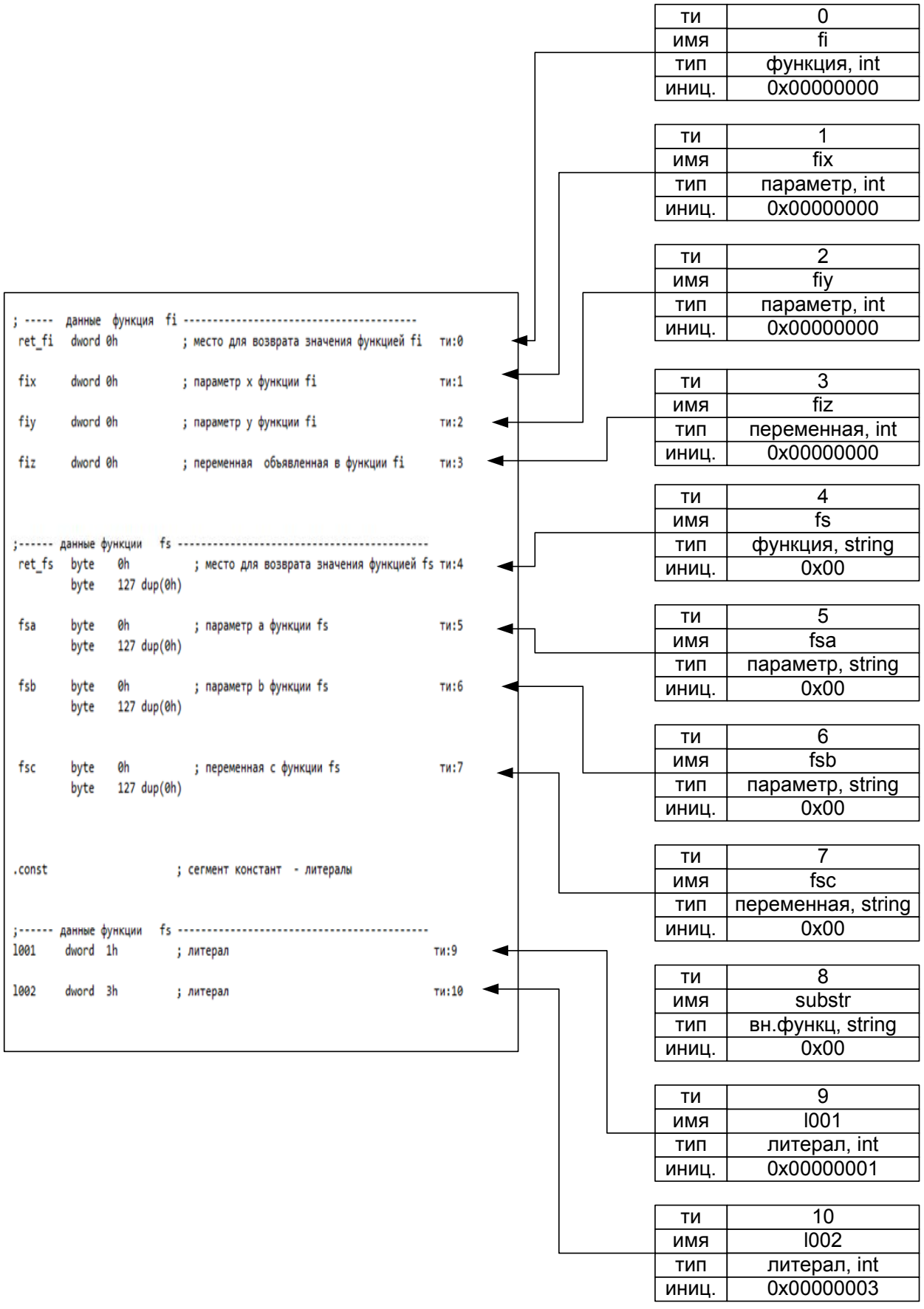


2.5 Простой вариант генерации кода на ассемблере

Скелет главной функции:

```
.586                        ; система команд (процессор Pentium)
.model flat,stdcall         ; модель памяти, соглашение о вызовах
 includelib kernel32.lib    ; компановщику: компоновать с kernel32.lib
                            ; можем компоновать со стандартной библиотекой

 ExitProcess PROTO   :DWORD  ; прототип функции

 .stack 4096                ; сегмент стека объемом  4096 - для вычислений



 .data                      ; сегмент данных  -  переменные и параметры

 .const                     ; сегмент констант  - литералы

 .code                       ; сегмент кода  - испоняемый код

  main PROC                  ; начало процедуры  - согласование с ОС


    push 0                   ; код возрата процесса (параметр ExitProcess )
    call ExitProcess         ; так должен заканчиваться любой процесс Windows
 main ENDP                   ; конец процедуры
end main
```

2.6 Модель памяти SVV-2015: данные



2.7 Заготовки-шаблоны для функций

```
 .code                        ; сегмент кода  -

;  ---- функция fi -----------------------------                    ти:0
fi_func  PROC uses eax ebx ecx edi esi

; перед вызовом функции должны быть заполнены   fix  и  fiy
; результат работы поместить в ret_fi

fi_func  ENDP

;  ---- функция fs-----------------------------                     ти:4
fs_func  PROC uses eax ebx ecx edi esi

; перед вызовом функции должны быть заполнены  fsa  и  fsb

; результат работы поместить в ret_fs

fs_func  ENDP

; ----- функция  main --------------------------                   ти:11
svv2016   PROC uses eax ebx ecx edi esi


; перед вызовом  fi_func параметры поместить в  fix и  fiy
; результат  fi_func взять из  ret_fi


; перед вызовом  fs_func параметры поместить в  fsa и  fsb
; результат  fi_func взять из  ret_fs


; код возврата поместить в EAX

svv2016   ENDP

 ;-----  согласование с операционной системой
 main PROC                   ; начало процедуры

   ; если необходимо, то можно передать параметры
   call  svv2016             ; вызов  функции main  сгенерированного кода svv2016


   push eax                  ; код возрата процесса (параметр ExitProcess )
   call ExitProcess          ; так должен заканчиваться любой процесс Windows
 main ENDP                   ; конец процедуры
end main
```

ти	0
имя	fi
тип	функция, int
иниц.	0x00000000

ти	4
имя	fs
тип	функция, string
иниц.	0x00

ти	11
имя	main
тип	точка входа, int
иниц.	0x00

Сгенерировать автоматически пролог и эпилог

2.8 Для внешних функций можно сгенерировать функцию-обертку для согласования стандарта вызова:

- функцию написать на C++ и поместить ее библиотеку;
- вызов функции выполнить из функции обертки;
- в генерируемом коде вызывать функцию обертку.

3. Генерация кода для выражений

```
.data                       ; сегмент данных - переменные и параметры
 svv2016x     sdword 0h     ; переменная x со знаком
 svv2016y     sdword 0h     ; переменная y со знаком
 svv2016z     sdword 0h     ; переменная z со знаком
 svv2016a     sdword 0h     ; переменная a со знаком
 svv2016b     sdword 0h     ; переменная b со знаком

.const                      ; сегмент констант - литералы
 l001  sdword 1h            ; литерал
 l002  sdword 3h            ; литерал
 l003  sdword 4h            ; литерал

.code                       ; сегмент кода - исполняемый код
```

```
;-------------------------------------------------------------------------------
; x = 1;
; y = x;
; z = x*y;
; z = z + x*y;
; a = 3;
; b = 7;
; z = x*y + b*(x+y);
```

```
; генерация кода: x = 1 --> svv2016x = l001

     ; генерация кода:   l001
     push l001

     ; генерация кода:  x=
     pop  svv2016x
```

```
; генерация кода: y = x --> svv2016y = svv2016x

    ; генерация кода:   svv2016x
    push  svv2016x

    ; генерация кода:  y=
     pop  svv2016y
```

```
; генерация кода: z = x*y   --> svv2016z = svv2016x*svv2016y -->svv2016z = svv2016x svv2016y *

      ; генерация кода:  svv2016x
      push  svv2016x

      ; генерация кода:  svv2016y
      push  svv2016y

      ; генерация кода: *
      pop   eax
      pop   ebx
      imul  ebx      ;         eax = eax*ebx
      push  eax

      ; генерация кода:  z =
       pop  svv2016z
```

```
; генерация кода: z = z+ x*y   --> svv2016z = svv2016z+ svv2016x*svv2016y -->svv2016z = svv2016z svv2016x svv2016y *+

; генерация кода:  svv2016z
       push  svv2016z

; генерация кода:  svv2016x
       push  svv2016x

; генерация кода:  svv2016y
       push  svv2016y

; генерация кода: *
       pop   eax
       pop   ebx
       imul  ebx          ;  eax = eax*ebx
       push  eax

; генерация кода: +
       pop   eax
       pop   ebx
       add   eax, ebx     ;   eax = eax+ebx
       push  eax

; генерация кода:  z =
       pop  svv2016z
```

```
; a = 3  --> a = l002
; b = 7 ---> b = l003
; z = x*y + b*(x+y) --> svv2016z = svv2016x*svv2016y + svv2016b * (svv2016x + svv2016y)
;                       svv2016z = svv2016x svv2016y * svv2016b svv2016x svv2016y +*+
; генерация кода: a = 3   --> svv2016a = l002

; генерация кода:  l002
       push l002
; генерация кода:  a =
      pop svv2016a
; генерация кода:  l003
       push l003
; генерация кода:  b =
      pop svv2016b
; генерация кода:  svv2016x
       push svv2016x
; генерация кода:  svv2016y
       push svv2016y
; генерация кода: *
        pop   eax
        pop   ebx
        imul  ebx           ;  eax = eax*ebx
        push  eax
; генерация кода:  svv2016b
       push svv2016b
; генерация кода:  svv2016x
       push svv2016x
; генерация кода:  svv2016y
       push svv2016y
; генерация кода: +
        pop   eax
        pop   ebx
        add   eax, ebx      ;   eax = eax+ebx
        push  eax
; генерация кода: *
        pop   eax
        pop   ebx
        imul  ebx           ;  eax = eax*ebx
        push  eax
; генерация кода: +
        pop   eax
        pop   ebx
        add   eax, ebx      ;   eax = eax+ebx
        push  eax
; генерация кода:  svv2016z =
      pop svv2016z
```

3.1 Готовые шаблоны:

```
//шаблоны
#define EXPR_INT       "push    %s \n"             //  i
#define EXPR_INT_E     "pop     %s \n"             //  =
#define EXPR_INT_PLUS  "pop     eax\npop     ebx\nadd     eax, ebx\npush    eax\n"          //  +
#define EXPR_INT_MUL   "pop     eax\npop     ebx\nimul    eax, ebx\npush    eax\n"          //  *
#define GEN1(b, tmpl, var)  sprintf_s(b, 1024,tmpl, #var)
#define GEN0(b, tmpl)       sprintf_s(b, 1024,tmpl)
```

Пояснения:

- <имя>_s – это безопасные функции с указанием емкости приемника. sprintf_s - возвращает количество байт, записанных в буфер или -1
- Буфер – место, где сохраняется генерируемый код.
- Стрингификация – операция # (преобразование фрагмента кода в строку)

```
char buf[1024];

// x =1
int k = GEN1(buf,EXPR_INT,          1001);                  // 1
    k+= GEN1(buf+k,EXPR_INT_E,      svv2016x);              // =
```

```
// y = x
    k+= GEN1(buf+k,EXPR_INT,        svv2016x);              // x
    k+= GEN1(buf+k,EXPR_INT_E,      svv2016y);              // =
```

```
// z = xy*
    k+= GEN1(buf+k,EXPR_INT,        svv2016x);              // x
    k+= GEN1(buf+k,EXPR_INT,        svv2016y);              // y
    k+= GEN0(buf+k,EXPR_INT_MUL);                           // *
    k+= GEN1(buf+k,EXPR_INT_E,      svv2016z);              // =
```

```
// z = zxy*+
    k+= GEN1(buf+k,EXPR_INT,        svv2016z);              // z
    k+= GEN1(buf+k,EXPR_INT,        svv2016x);              // x
    k+= GEN1(buf+k,EXPR_INT,        svv2016y);              // y
    k+= GEN0(buf+k,EXPR_INT_MUL);                           // *
    k+= GEN0(buf+k,EXPR_INT_PLUS);                          // +
    k+= GEN1(buf+k,EXPR_INT_E,      svv2016z);              // =
```

```
// a = 3;
    k+= GEN1(buf+k,EXPR_INT,       l002);                  // 3
    k+= GEN1(buf+k,EXPR_INT_E,     svv2016a);              // =
```

```
// b = 7;
    k+= GEN1(buf+k,EXPR_INT,       l003);                   // 7
    k+= GEN1(buf+k,EXPR_INT_E,     svv2016b);               // =
```

```
// z = xy* bxy+*+;
     k+= GEN1(buf+k,EXPR_INT,       svv2016x);              // x
     k+= GEN1(buf+k,EXPR_INT,       svv2016y);              // y
     k+= GEN0(buf+k,EXPR_INT_MUL);                          // *
     k+= GEN1(buf+k,EXPR_INT,       svv2016b);              // b
     k+= GEN1(buf+k,EXPR_INT,       svv2016x);              // x
     k+= GEN1(buf+k,EXPR_INT,       svv2016y);              // y
     k+= GEN0(buf+k,EXPR_INT_PLUS);                         // +
     k+= GEN0(buf+k,EXPR_INT_MUL);                          // *
     k+= GEN0(buf+k,EXPR_INT_PLUS);                         // +
     k+= GEN1(buf+k,EXPR_INT_E,     svv2016z);              // =

     std::cout << buf;
```

3.2 Сгенерированный код

```
push    1001
pop     svv2016x
push    svv2016x
pop     svv2016y
push    svv2016x
push    svv2016y
pop     eax
pop     ebx
imul    eax, ebx
push    eax
pop     svv2016z
push    svv2016z
push    svv2016x
push    svv2016y
pop     eax
pop     ebx
imul    eax, ebx
push    eax
pop     eax
pop     ebx
add     eax, ebx
push    eax
pop     svv2016z
push    1002
pop     svv2016a
push    1003
pop     svv2016b
push    svv2016x
push    svv2016y
pop     eax
pop     ebx
imul    eax, ebx
push    eax
push    svv2016b
push    svv2016x
push    svv2016y
pop     eax
pop     ebx
add     eax, ebx
push    eax
pop     eax
pop     ebx
imul    eax, ebx
push    eax
pop     eax
pop     ebx
add     eax, ebx
push    eax
pop     svv2016z
```

1) План памяти строим по ТИ
2) Выполняем преобразование выражений в ПОЛИЗ
3) Заготавливаем шаблоны кода соответствующего правила грамматики
4) Генерируем код по дереву разбора