МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)

Кафедра “Моделирование систем и информационные технологии”

РАЗРАБОТКА ТРАНСЛЯТОРА ЯЗЫКА ПРОГРАММИРОВАНИЯ НА SVN, YACC и C++

Методические указания к курсовой работе по предмету
“Системное программное обеспечение”

Составитель В. В. Лидовский

Москва 2017

Методические указания к курсовой работе по предмету
“Системное программное обеспечение”
/ сост. В. В. Лидовский. — М.: МАИ, 2017. — с. 32.

ВВЕДЕНИЕ

Настоящие методические указания предназначены для обеспечения учебного процесса студентов четвертого курса дневной формы обучения специальности 09.03.01 “ИВТ” при выполнении курсовой работы по предмету “Системное программное обеспечение”.

Цели курсовой работы: научиться использовать на практике программу yacc (bison) и, опционально, программу lex (flex), изучить поэтапный процесс с использованием программы Subversion создания транслятора языка программирования и разработать действующий транслятор.

Для выполнения работы требуются знания языка программирования си++ и основ разбора снизу-вверх. При использовании программы lex или flex потребуется также знакомство с простейшими регулярными выражениями.

1. СИСТЕМНЫЕ ТРЕБОВАНИЯ

Для выполнения курсовой работы необходимы следующие системные компоненты:

а) генератор компиляторов bison или yacc;
б) компилятор языка си++ (предпочтительно GNU g++);
в) программа make;
г) система версионного контроля Subversion;
д) (опционально) генератор лексических сканеров flex/lex.

2. ПОСТАНОВКА ЗАДАЧИ

Разработать транслятор языка программирования, поддерживающего стандартную запись арифметических чисел, работу с переменными, основные операторы, процедуры и функции, а также дополнительные, определяемые вариантом задания, возможности.

Работу следует разбить на несколько этапов, начав с разработки простейших возможностей и постепенно переходя к более сложным.

3. ЭТАПЫ РАЗРАБОТКИ ТРАНСЛЯТОРА

3.1. Целочисленный калькулятор

Начнём работать с svn-репозиторием. Выписывем сначала рабочую копию файлов проекта — этот шаг всегда начинает работу при использовании нового компьютера.

```
svn co СЕРВЕР КАТАЛОГ
cd КАТАЛОГ
```

В локальный КАТАЛОГ следует записывать все файлы проекта. Адрес сервера можно узнать у администратора проекта.

Грамматику языка взаимодействия с калькулятором можно описать следующим образом — приоритеты операций установим затем явно.

list $\to \varepsilon$ | list nl | list expr nl

expr $\to$ number | expr + expr | expr − expr | expr $*$ expr | expr / expr | expr ^ expr | −expr | (expr)

В этой грамматике list — это список вводимых команд-строк, nl — маркер конца строки — клавиша Enter, expr — вычисляемое выражение. В yacc принято терминалы обозначать отдельными знаками в апострофах или заглавными буквами, а метасимволы — строчными. Создадим файл hoc.cpp.y, содержащий описание этой грамматики.

```
%{
#include <iostream>
#include <cctype>
#include <string>
using namespace std;
int yylex(), yyerror(const string&);
%}
%token NUMBER              //лексема, терминал
%left '+' '-'              //левая ассоциативность
%left '*' '/'              //приоритет повышается снизу-вверх
%right '^'
%left UNARYMINUS    //унарный минус, псевдотерминал
%%
list:  /* пусто */
| list '\n'
| list expr '\n' { cout << $2 << endl;}
;
expr: NUMBER //{$$ = $1;} //действие по-умолчанию
| expr '+' expr {$$ = $1 + $3;}
| expr '-' expr {$$ = $1 - $3;}
| expr '*' expr {$$ = $1 * $3;}
| expr '/' expr {$$ = $1 / $3;}
| expr '^' expr {$$ = $1;
   for (int i = 1; i < $3; i++) $$ *= $1;}
| '-' expr %prec UNARYMINUS {$$ = -$2;} //установка приоритета
| '(' expr ')' {$$ = $2;}
;
%%
int lineno = 1;  //for error messages

int yylex () {
```

```
  int c;
  while ((c = cin.get()) == ' ' || c == '\t');
  if (isdigit(c)) {
       cin.unget();
       cin >> yylval;
       return NUMBER;
  }
  if (c == '\n')
       lineno++;
  return c;
}
int yyerror(const string &s) {
  cerr << s << " in " << lineno << endl;
}
main () {
  yyparse();
}
```

Программа на yacc начинается с объявлений на языке си++, заключенных между `%{` и `%}`. Этот раздел может отсутствовать. Второй раздел — это правила трансляции: сначала категории лексем (токенов, tokens), затем приоритеты и ассоциативность, затем, между маркерами %%, грамматика и атрибуты. Последний раздел — это текст на си++. Тут должны присутствовать функции yylex (лексический анализатор, вызывается yyparse, может генерироваться программой lex), yyerror (вызывается в случае ошибки из yyparse) и main. Yacc генерирует текст функции yyparse и определения связанных с ней программных объектов: переменных, например, yylval, макросов, ... Переменная yylval типа YYSTYPE используется для передачи значения (атрибута) лексемы. Символы $$, $1, $2, ... — это соответственно атрибуты типа YYSTYPE (по умолчанию — int) нетерминала левой части и занумерованных с 1 компонент правой части продукции.

Приоритет терминала устанавливается вместе с ассоциативностью, порядком следования — описанные выше терминалы имеют меньший приоритет. Приоритет терминала присваивается также правилу, в котором он встречается последним. Если приоритет правила ниже, чем следующего за ним терминала, то выбирается сдвиг, а если не ниже, то свертка. Например, при разборе $5 + 3\_ * 2$ выбирается сдвиг, так как приоритет $*$ выше, чем правила expr $\rightarrow$ expr + expr. Команда %prec позволяет явно установить приоритет правила.

Вместо функции yylex, если она уже сгенерирована lex, можно было бы использовать директиву препроцессора `#include "lex.yy.c"` —

файл lex.yy.c получается обработкой файла hoc.cpp.l с описанием лексики программой lex. Для реализуемого калькулятора лексика может быть описана следующими строками.

```
%option   noyywrap  //системная установка
number   [0-9]+
%%
[ ]       /* пропуск пробелов */
{number}  {istringstream s(yytext); s >> yylval; return NUMBER;}
\n        ++lineno; return '\n';
.         return yytext[0];
```

В регулярных выражения lex знак точки — это любой символ, кроме конца строки. Для использования строковых потоков необходимо добавить заголовок `<sstream>`. Строка yytext — это входной текст для lex, соответствующий шаблону, регулярному выражению. Каждому шаблону сопоставляются действия. Если действия размещаются в нескольких строках, то их нужно брать в фигурные скобки. Шаблоны перебираются по-порядку — после встречи подходящего все остальные шаблоны игнорируются.

Для сборки используем makefile, пример которого приведён в приложение. В этом файле все отступы должны быть получены табуляцией. Команда make должна собрать исполнимый файл hoc.

Отметим теперь версионируемые файлы.

```
svn add makefile hoc.cpp.y hoc.cpp.l
```

Имя hoc.cpp.l необходимо указывать, только если используется программа lex. Зафиксируем теперь изменения в репозитории и обновим рабочую копию. Можно использовать любые подходящие комментарии вместо преведённых.

```
svn ci -m "Сделана версия 1"
svn up
```

3.2. **Поддержка вещественных чисел**

Для работы с вещественными числами достаточно сделать четыре изменения.

1. Внесем в начало, между `%{` и `%}`, следующую строку, определив тип YYSTYPE явно.

```
#define YYSTYPE double
```

2. Изменим правило для возведения в степень на

```
| expr '^' expr {$$ = pow($1, $3);}
```

— воспользуемся функцией стандартной математической библиотеки.

3. Добавим заголовок `<cmath>`.

4. В yylex (для десятичной точки) заменим строку с isdigit на

```
if (c == '.' || isdigit(c)) {
```

или в файле-лексике, hoc1.cpp.l, переопределим множество number.

```
number   [0-9]+\.?|[0-9]*\.[0-9]+
```

Опять зафиксируем изменения в репозитории и обновим рабочую копию.

3.3. **Работа с простейшими переменными**

Добавим простейшую поддержку работы с переменными, именами переменных будут маленькие английские буквы от a до z, а также механизм восстановления после ошибок. Для хранения переменных введем массив — добавим между `%{` и `%}` следующую строку.

```
double vars[26];
```

Имена переменных получаются целыми числами, поэтому при синтаксическом разборе будут возникать величины типов int и double, которые передаются через yylval. Для реализации такого совмещения естественно использовать объединение — для этого случая предоставляется специальное описание YYSTYPE. Добавляем вместо %token NUMBER следующий текст.

```
%union {
  double val; //значения величин
  int index;  //индекс переменной в vars
}
%token <val> NUMBER  //лексема с типом
%token <index> VAR
%type <val> expr  //устанавливает тип для выражений
%right '='        //присваивание, приоритет наименьший
```

Информация в %type позволяет автоматически выбирать нужный тип для значения-атрибута символа грамматики.

Добавим к грамматике следующие правила.

```
| VAR {$$ = vars[$1];}
| VAR '=' expr {$$ = vars[$1] = $3;}
```

Допустимы многократные присваивания типа `a = b = c = 5`.

К лексическому анализатору нужно добавить

```
if (islower(c)) {
  yylval.index = c - 'a';
  return VAR;
}
```

и изменить ввод, прибавив обращение к соответствующему полю yylval, `cin >> yylval.val;`,

или изменить lex-программу так, чтобы в ней появились следующие строки.

```
variable   [a-z]
{number}   {istringstream s(yytext);
            s >> yylval.val; return NUMBER;}
{variable} yylval.index = yytext[0] - 'a'; return VAR;
```

Слово error зарезервировано — оно дает возможность анализатору "поймать" ошибку синтаксиса и восстановиться после нее. Добавим к грамматике следующее правило.

```
| list error {yyerrok;}
```

Действие yyerrok позволяет анализатору вернуться назад к возможности правильного разбора, пропустив ошибочную конструкцию. Для восстановления после ошибки, определяемой пользователем, можно использовать средства try, catch и throw с передачей управления на стандартную функцию yyparse после перехвата ошибки. Работа с ошибками — одна из наиболее сложных частей анализатора и рассматриваются только простейшие возможности.

Таким образом модифицируем файл hoc.cpp.y и, опционально, файл описания лексики (hoc.cpp.l). Как обычно, фиксируем изменения и обновляем рабочую копию.

3.4. **Переменные, стандартные функции и константы**

Добавим к калькулятору поддержку переменных, задаваемых идентификаторами, а также стандартных функций и констант. Кроме того, печатать значение выражения будем не всегда, а только когда нужно, например, не будет распечатки после присваивания. Редактируем hoc.cpp.y и, опционально, hoc.cpp.l.

Заменим текст после %} на следующий.

```
%union {
  double val;
  Symbol *sym;  //указатель записи с полем-именем
}
%token <val> NUMBER
%token <sym> VAR BLTIN UNDEF //переменные, встроенные функции
                             //и константы, неопределенность
%type <val> expr assign   //устанавливает тип для выражений и
                                                  //присваиваний
```

Добавим в правило для list следующую строку, чтобы не печатать отдельные присваивания.

```
| list assign '\n'
```

Введем правило для assign

```
assign: VAR '=' expr {$$ = $1->val = $3; $1->type = VAR;}
;
```

и удалим соответствующую строку из правила для expr.

К expr-продукции добавим следующие строки вместо соответствующей строки.

```
| VAR {
    if ($1->type == UNDEF)
        throw "undefined variable: " + *$1->name;
    $$ = $1->val;
  }
| assign //для конструкций вида (a=5)*6
| BLTIN '(' expr ')' {$$ = (*($1->fp))($3);}
```

Использование assign в expr приводит к конфликту сдвиг/свертка, но приоритет у сдвига выше, поэтому отдельные присваивания не сворачиваются к выражению.

Удалим включения заголовков и перепишем первый раздел программы.

```
%{
#include <iostream>
#include <cctype>
#include <string>
#include <sstream>
#include <map>
#include <cmath>
using namespace std;

int yylex(), yyerror(const string&);

struct Symbol {
  string *name;
  short type; //VAR, BLTIN, UNDEF
  union {
    double val;  //VAR
    double (*fp)(double); //BLTIN
  };
};

struct Flist {
  string name;
  double (*fp)(double);
} flist [] = {{"sin", sin}, {"ln", log}, {"sqrt", sqrt},
   {"arctg", atan}};

struct Clist {
  string name;
  double val;
} clist [] = {{"pi", 3.1415926536}, {"e", 2.7182818284},
```

```
   {"phi", 1.6180339887}};

map<string, Symbol> names; //общая таблица имен
%}
```

В третьем разделе появится определение функции-инициализатора.

```
void init_names(Flist *p1, Clist *p2) {
  for (; p1 < flist + sizeof(flist)/sizeof(Flist); ++p1) {
     names[p1->name].name = &p1->name;
     names[p1->name].type = BLTIN;
     names[p1->name].fp = p1->fp;
  }
  for (; p2 < clist + sizeof(clist)/sizeof(Clist); ++p2) {
     names[p2->name].name = &p2->name;
     names[p2->name].type = VAR;
     names[p2->name].val = p2->val;
  }
}
```

В yylex вместо раздела с islower появится следующий раздел.

```
 if (isalpha(c)) {
      char sbuf[100], *p = sbuf;
      do {
        *p++ = c;
        c = cin.get();
      }
      while (!cin.eof() && isalnum(c));
      cin.unget();
      *p = 0;
      if (names.find(sbuf) == names.end()) {
        names[sbuf].type = UNDEF;
        names[sbuf].name = &(string&)names.find(sbuf)->first;
      }
      Symbol *s = &names[sbuf];
      yylval.sym = s;
      return s->type == UNDEF ? VAR : s->type;
  }
```

Можно вместо этого убрать строки с variable и добавить строку

```
symbol      [a-zA-Z][a-zA-Z0-9]*
```

и текст

```
{symbol}  {
     string sbuf(yytext);
     if (names.find(sbuf) == names.end()) {
        names[sbuf].type = UNDEF;
        names[sbuf].name = &(string&)names.find(sbuf)->first;
     }
```

```
    Symbol *s = &names[sbuf];
    yylval.sym = s;
    return s->type == UNDEF ? VAR : s->type;
}
```

к файлу с лексикой.

Функция main с поддержкой генерируемых исключений примет следующий вид.

```
main () {
  init_names(flist, clist);
  while (1) {
    try {
      yyparse();
      break;
    }
    catch (string s) {
      cerr << s << endl;
    }
  }
}
```

Фиксируем изменения в svn.

3.5. **Генерация кода**

Чтобы ввести поддержку работы с операторами и определяемыми пользователем подпрограммами можно использовать два подхода.

1. Реализовать макроподстановки, т. е. сделать язык, похожий на препроцессор си/си++. Такой подход используется также в языках TEX, METAFONT и некоторых других. Он требует значительного усложнения лексического сканера.

2. Реализовать компилятор в простой стековый код и интерпретатор этого кода. В пи-системах, яве, платформе .NET подобный подход приводит к генерации отдельных файлов с независимыми от аппаратуры кодами, которые потом исполняют специальные, относительно простые интерпретаторы. Во многих языках сценариев (оук, перл, питон, рубин и др.) подобный код генерируется по ходу исполнения программы и тут же выполняется интерпретатором.

Трансформируем программу-калькулятор так, чтобы введенные пользователем строки сначала переводились в стековый код, который затем исполняется. Например, выражение x=4+y трансформируется в код

```
  constpush     записать в стек константу
  4
  varpush       записать в стек переменную
  y             символ y
```

```
  eval          вычислить значение символа
  add           сложить числа на вершине стека
  varpush
  x
  assign_code   присвоить символу на вершине стека следующее
                значение за ним в стеке значение
  pop_code      убрать элемент с вершины стека
  STOP          маркер конца программы, останавливает
                интерпретатор
```

Программа становится большой и ее поэтому лучше разделить на части. Сначала изменяем файл hoc.cpp.y — существенно меняются лишь атрибуты грамматики, а также обработка лексем-чисел. Числа теперь заносятся в таблицу символов, что позволяет хранить одинаковые числа-константы в одном месте памяти. Вместо стандартной обработки ошибок через error–errok, изменив yyerror, перейдем к полностью контролируемой пользователем обработке ошибок. В main теперь будут друг за другом запускаться синтаксический анализатор, управляющий компиляцией, и интерпретатор.

```
%{
#include <cctype>
#include <sstream>
#include <map>
#include "hoc.h"
#define code2(c1,c2) code(c1);code(c2)
#define code3(c1,c2,c3) code(c1);code(c2);code(c3)
%}
%union {
  Symbol *sym;  //указатель записи
  Inst *inst;   //указатель команды стекового кода
}
%token <sym> VAR BLTIN UNDEF NUMBER
%right '='
%left '+' '-'
%left '*' '/'
%right '^'
%left UNARYMINUS
%%
list:
| list '\n'
| list assign '\n' {code2(pop_code, STOP); return 1;}
| list expr '\n' {code2(print, STOP); return 1;}
;
assign: VAR '=' expr {code3(varpush, (Inst) $1, assign_code);}
;
```

```
expr: NUMBER {code2(constpush, (Inst) $1);}
| VAR {code3(varpush, (Inst) $1, eval);}
| assign
| BLTIN '(' expr ')' {code2(bltin, (Inst)$1->fp);}
| expr '+' expr {code(add);}
| expr '-' expr {code(sub);}
| expr '*' expr {code(mul);}
| expr '/' expr {code(div);}
| expr '^' expr {code(power);}
| '-' expr %prec UNARYMINUS {code(negate_code);}
| '(' expr ')'
;
%%
```

Определения lineno, Flist, Clist, init_names остаются такими же.

Добавляем к yylex новый раздел для работы с числами вместо прежнего.

```
  if (c == '.' || isdigit(c)) {
      cin.unget();
      double w;
      cin >> w;
      ostringstream oss;
      oss << w;
      string s(oss.str());
      if (names.find(s) == names.end()) {
        names[s].type = NUMBER;
        names[s].name = &(string&)names.find(s)->first;
        names[s].val = w;
      }
      yylval.sym = &names[s];
      return NUMBER;
  }
```

Можно вместо этого подправить описание лексики.

```
{number}   {
              istringstream iss(yytext);
              double w;
              iss >> w;
              string s(yytext);
```

Функции yyerror и main изменяются незначительно.

```
int yyerror(const string &s) {
   ostringstream oss;
   oss << s << " in " << lineno << endl;
   throw oss.str();
}
```

```
main () {
  init_names(flist, clist);
  while (1) {
    try {
      for (initcode();yyparse();initcode())
        execute(prog);
      break;
    }
    catch (string s) {
      cerr << s << endl;
    }
  }
}
```

Затем создадим файл code.cpp с реализацией машины-интерпретатора. Программа записывается в массив prog и при исполнении использует стек на основе массива stack.

```
#include "hoc.h"
#include "y.tab.h"

#define NSTACK 5000 //размер стека
static Datum stack[NSTACK];            //защита абстракции -
                                       //модульной видимостью
static Datum *stackp; //указатель вершины стека

#define NPROG 2000 //макс. размер программы
Inst prog[NPROG]; //память
Inst *progp; //первое свободное место для создаваемого кода
Inst *pc; //program counter, счетчик команд при выполнении

void initcode() {
  stackp = stack;
  progp = prog;
}

Inst* code(Inst f) { //занести одну команду в память
  Inst *saved_progp = progp;
  if (progp >= prog + NPROG)
    throw (string) "program too big";
  *progp++ = f;
  return saved_progp;
}

void execute(Inst *p) { //выполнить команды
  for (pc = p; *pc != STOP;)
    (*(*pc++))();   //это весь интерпретатор!
```

```
}

//работа со стеком
void push(Datum d) {
   if (stackp >= stack + NSTACK)
     throw (string) "stack overflow";
   *stackp++ = d;
}

Datum pop() {
  if (stackp <= stack)
    throw (string) "stack underflow";
  return *--stackp;
}

//команды для интерпретатора
void pop_code() {
  pop();
}
void constpush() { //записать константу в стек
  Datum d;
  d.val = ((Symbol*) *pc++)->val;
  push(d);
}

void varpush() { //записать переменную в стек
  Datum d;
  d.sym = (Symbol*) *pc++;
  push(d);
}

void sub() { //вычитание
  Datum d1 = pop(), d2 = pop();
  d2.val -= d1.val;
  push(d2);
}
```

Аналогично реализуются add, mul, div, power и negate_code.

```
void bltin() {
  Datum d = pop();
  d.val = (*(double(*)(double))(*pc++))(d.val);
  push(d);
}

void assign_code() {
  Datum d1 = pop(), d2 = pop();
```

```
  d1.sym->val = d2.val;
  d1.sym->type = VAR;
  push(d2);
}

void eval() {
  Datum d = pop();
  if (d.sym->type == UNDEF)
    throw "undefined variable: " + *d.sym->name;
  d.val = d.sym->val;
  push(d);
}

void print() {
  Datum d = pop();
  cout << d.val << endl;
}
```

Значение функции code в данной версии программы не используется. Для согласования файлов создадим файл-заголовок hoc.h с общими и предварительными объявлениями.

```
#include <iostream>
#include <string>
#include <cmath>
using namespace std;

struct Symbol {
  string *name;
  short type; //VAR, BLTIN, UNDEF, NUMBER
  union {
    double val;  //VAR, NUMBER
    double (*fp)(double); //BLTIN
  };
};
int yylex(), yyparse(), yyerror(const string &);

union Datum { //тип стека интерпретатора
  double val;
  Symbol *sym;
};

typedef void (*Inst)();
#define STOP 0
extern Inst prog[];

Datum pop();
```

```
Inst* code(Inst);
void initcode(), push(Datum), pop_code(), execute(Inst*),
  constpush(), varpush(), add(), sub(), mul(), div(), power(),
  negate_code(), bltin(), assign_code(), eval(), print();
```

Более сложный процесс сборки программы нужно отразить в файле makefile. Цель clean позволяет, вызвав make clean, очистить каталог от временных файлов.

```
hoc: hoc.cpp.y hoc.h code.cpp makefile
        -lex hoc.cpp.l
        yacc -d hoc.cpp.y
        g++ -c code.cpp
        g++ -c y.tab.c
        g++ y.tab.o code.o -o hoc

clean:
        rm -f y.tab.[ch] lex.yy.c hoc *.o
```

Ключ -d приводит yacc к генерации файла y.tab.h, содержащего определения генерируемых yacc символов. Ключ -v можно использовать для распечатки LALR(1)-множеств пунктов с действиями, переходами и отметками конфликтов, а также суммарной справочной информации. Вместо yacc можно вызывать bison с ключом -y.

Необходимо добавить файлы code.cpp и hoc.h к версионируемым файлам проекта, зафиксировать изменения в репозитории и обновить рабочую копию.

3.6. **Операторы**

Добавим к языку поддержку операторов if, while, печати и составного, а также 6 операций сравнения и 3 логических.

```
%token <sym> VAR BLTIN UNDEF NUMBER PRINT WHILE IF ELSE
%type <inst> assign expr oper operlist while if cond endop
%right '='
%left OR
%left AND
%left GT GE LT LE EQ NE  //> >= < <= == !=
%left '+' '-'
%left '*' '/'
%right '^'
%left NOT
```

Добавим к list-продукции информацию об операторах.

```
| list oper '\n' {code(STOP); return 1;}
```

Оператор и связанные с ним понятия определяются правилами.

```
oper: expr {code(pop_code);} //$$=$1 происходит автоматически
| PRINT expr {code(prexpr); $$ = $2;}
```

```
| while cond oper endop {
    ($1)[1] = (Inst) $3; //переход на тело цикла
    ($1)[2] = (Inst) $4;} //переход на выход из цикла
| if cond oper endop { //if без else
    ($1)[1] = (Inst) $3; //переход на часть после then
    ($1)[3] = (Inst) $4;} //переход на следующий оператор
| if cond oper endop ELSE oper endop { //if с else
    ($1)[1] = (Inst) $3;
    ($1)[2] = (Inst) $6; //переход на else-часть
    ($1)[3] = (Inst) $7;}
| '{' operlist '}' {$$ = $2;}
;
cond: '(' expr ')' {code(STOP); $$ = $2;}
;
while: WHILE {$$ = code3(while_code, STOP, STOP);}
            //STOP - резервирование места для перехода
;
if: IF {$$ = code(if_code); code3(STOP, STOP, STOP);}
;
endop: {code(STOP); $$ = progp;}
;
operlist: {$$ = progp;}
| operlist '\n'
| operlist oper
;
```

К assign-продукции добавим оператор `$$ = $3;` и внесём следующие изменения в expr-продукцию.

```
NUMBER {$$ = code2(constpush, (Inst) $1);}
| VAR {$$ = code3(varpush, (Inst) $1, eval);}
| BLTIN '(' expr ')' {$$ = $3; code2(bltin, (Inst)$1->fp);}
| '-' expr %prec NOT {$$ = $2; code(negate_code);}
| '(' expr ')' {$$ = $2;}
```

Эти изменения вызваны необходимостью отслеживать адрес генерируемых выражений. Сюда также добавляются строки с определениями новых операций.

```
| expr GT expr {code(gt);}
| expr GE expr {code(ge);}
| expr LT expr {code(lt);}
| expr LE expr {code(le);}
| expr EQ expr {code(eq);}
| expr NE expr {code(ne);}
| expr AND expr {code(and_code);}
| expr OR expr {code(or_code);}
| NOT expr {$$ = $2; code(not_code);}
```

Лексический анализатор потребует вспомогательной функции.

```
int follow(int expect, int ifyes, int ifno) {
  int c = cin.get();
  if (c == expect)
    return ifyes;
  cin.unget();
  return ifno;
}
```

Добавим к yylex следующий новый раздел.

```
  switch (c) {
    case '>': return follow('=', GE, GT);
    case '<': return follow('=', LE, LT);
    case '!': return follow('=', NE, NOT);
    case '=': return follow(c, EQ, c);
    case '|': return follow(c, OR, c);
    case '&': return follow(c, AND, c);
  }
```

Вместо этого можно добавить к hoc.cpp.l следующие строки.

```
>=    return GE;
\<=   return LE;
==    return EQ;
!=    return NE;
>     return GT;
\<    return LT;
\|\|  return OR;
&&    return AND;
!     return NOT;
```

Нужно добавить служебные слова к списку символов. Добавим массив данных

```
struct Klist {
  string name;
  int kval;
} klist [] = {{"if", IF}, {"else", ELSE}, {"while", WHILE},
   {"print", PRINT}};
```

и соответствующий цикл к initnames, у которой появится третий аргумент (p3).

```
   for (; p3 < klist + sizeof(klist)/sizeof(Klist); ++p3) {
      names[p3->name].name = &p3->name;
      names[p3->name].type = p3->kval;
    }
```

В main вызов этой функции должен быть дополнен третьим аргументом — klist.

Добавим к hoc.h объявления следующих новых функций.

```
void gt(), ge(), lt(), le(), eq(), ne(), and_code(), or_code(),
   not_code(), while_code(), if_code(), prexpr();
```

Также изменим следующую строку.

```
extern Inst prog[], *progp;
```

Добавим в конец code.cpp определения новых функций.

```
void gt() { //greater than
  Datum d1 = pop(), d2 = pop();
  d2.val = (d2.val > d1.val);
  push(d2);
}

//Функции ge, lt, le, eq и ne

void and_code() {
  Datum d1 = pop(), d2 = pop();
  d2.val = (d2.val && d1.val);
  push(d2);
}

//Функции or_code и not_code

void prexpr() {
  Datum d = pop();
  cout << d.val << endl;
}
void while_code() {
  Datum d;
  Inst *savepc = pc;
  execute(pc + 2); //условие
  d = pop();
  while (d.val) {
    execute(*(Inst**)savepc); //тело
    execute(savepc + 2); //условие
    d = pop();
  }
  pc = *(Inst**)(savepc + 1); //следующий оператор
}

void if_code() {
  Inst *savepc = pc;
  execute(pc + 3); //условие
  Datum d = pop();
  if (d.val)
```

```
    execute(*(Inst**)savepc); //then-часть
  else if (*(savepc + 1)) //есть ли else
    execute(*(Inst**)(savepc + 1)); //else-часть
  pc = (*(Inst**)(savepc + 2)); //следующий оператор
}
```

Разберем исполнение while_code (if_code — аналогичен). Код для этого оператора имеет следующий вид.

```
        НАЧАЛО - КОД while_code
pc      ПЕРЕХОД НА ТЕЛО ЦИКЛА
pc + 1  ПЕРЕХОД НА ВЫХОД
pc + 2  УСЛОВИЕ
        ТЕЛО
```

Для генерации кода этого оператора в память записываются три слова: собственно код и два приведенных перехода. Используется рекурсивный вызов execute — каждая часть оператора (условие и тело) заканчиваются STOP.

Рассмотрим генерацию кода для выражения if(a>3)print a else a=7.

```
    if_code
    адрес позиции L1
    адрес позиции L2
    адрес позиции L3
    varpush
    a
    eval
    constpush
    3
    gt
    STOP
L1  varpush
    a
    eval
    prexpr
    STOP
L2  constpush
    7
    varpush
    a
    assign_code
    pop_code
    STOP
L3
```

Скорректируем makefile и проведём сборку hoc. Затем, например, можно вычислить факториал 7.

```
f=i=1
while(i<7){i=i+1 f=f*i}
f
```

Цикл можно было бы записать более кратко, например, `while(i<7) f=f*(i=i+1)`.

В заключении этапа работаем с Subversion.

3.7. **Определяемые процедуры и функции**

Добавим поддержку подпрограмм, чтобы можно было, например, определить

```
func fac() {if($1<2)return 1 else return $1*fac($1-1)}
proc hello() while($1>0){print $1," Hello\n" $1=$1-1}
```

и вызвать hello(fac(7)). Добавим также поддержку строк, расширим возможности оператора print аргументом-списком и добавим оператор ввода данных.

Изменим тип yylval.

```
%union {
  Symbol *sym;
  Inst *inst;
  int narg; //число аргументов подпрограммы, long для x86_64
}
```

Появятся новые категории лексем

```
%token <sym> STRING FUNCCALL PROCCALL RETURN FUNCDEF PROCDEF
%token <sym> READ
%token <narg> ARG
```

и новые данные.

```
%type <inst> prlist begin
%type <sym> subrname
%type <narg> arglist
```

В list-правиле появится строка определения новой подпрограммы.

```
| list deffn '\n'
```

В assign-продукции появится случай для работы с параметрами — они могут использоваться как локальные переменные и им можно присваивать значения. Параметры именуются $1, $2, ...

```
| ARG '=' expr {
    defonly("$"); code2(argassign, (Inst) $1); $$ = $3;}
```

В oper-продукции изменится правило с PRINT на

```
| PRINT prlist {$$ = $2;}
```

— теперь печатать можно не только отдельные выражения, но и списки из выражений и строк. Сюда также будут добавлены правила для возврата из подпрограмм и вызова процедур.

```
| RETURN {defonly("return"); code(procret);}
| RETURN expr {defonly("return"); code(funcret); $$ = $2;}
| PROCCALL begin '(' arglist ')' {
    $$ = $2;
    code3(call, (Inst) $1, (Inst) $4);}
```

Появятся продукции для begin, prlist, deffn, subrname (устанавливает, что именем подпрограммы может быть имя переменной или подпрограммы, т.е. эти объекты можно переопределять) и arglist (подсчет числа аргументов подпрограммы).

```
begin: {$$ = progp;}
;
prlist: expr {code(prexpr);}
| STRING {$$ = code2(prstr, (Inst) $1);}
| prlist ',' expr {code(prexpr);}
| prlist ',' STRING {code2(prstr, (Inst) $3);}
;
deffn: FUNCDEF subrname {$2->type = FUNCCALL; indef = 1;}
  '(' ')' oper {
        code3(constpush, (Inst)&names.begin()->second,
                                                    funcret);
        define($2);
        indef = 0;}
| PROCDEF subrname {$2->type = PROCCALL; indef = 1;}
   '(' ')' oper {code(procret); define($2); indef = 0;}
;
subrname: VAR
| FUNCCALL
| PROCCALL
;
arglist: {$$ = 0;}
| expr {$$ = 1;}
| arglist ',' expr {$$ = $1 + 1;}
;
```

В expr-продукции появятся варианты для аргумента подпрограммы (можно использовать только внутри подпрограммы), вызова функции и операции ввода данных в заданную переменную со стандартного потока ввода.

```
| ARG {defonly("$"); $$ = code2(arg, (Inst) $1);}
| FUNCCALL begin '(' arglist ')' {
    $$ = $2;
    code3(call, (Inst) $1, (Inst) $4);}
| READ '(' VAR ')' {$$ = code2(varread, (Inst) $3);}
```

В список (массив klist) ключевых слов нужно добавить следующие.

```
{"func", FUNCDEF}, {"proc", PROCDEF}, {"return", RETURN},
    {"read", READ}
```

Для работы со стандартными escape-последовательностями языка си добавим перед yylex функцию backslash.

```
int backslash(int c) {
  static string transtab = "b\bf\fn\nr\rt\t";
  if (c != '\\') return c;
  c = cin.get();
  if (transtab.find(c) != string::npos)
    return transtab[transtab.find(c) + 1];
  return c;
}
```

В yylex появится раздел для работы со строками, последовательностями знаков в кавычках, и раздел для работы с аргументами подпрограмм.

```
if (c == '$') {
    int n = 0;
    while (isdigit(c = cin.get()))
      n = n*10 + c - '0';
    cin.unget();
    if (n == 0)
      throw (string) "strange $...";
    yylval.narg = n;
    return ARG;
  }
  if (c == '"') { //строка в кавычках
    char sbuf[100], *p;
    for (p = sbuf; (c = cin.get()) != '"'; p++) {
      if (c == '\n')
        throw (string) "missing quote";
      if (p > sbuf + sizeof(sbuf)) {
        *p = 0;
        throw "string too long " + string(sbuf);
      }
      *p = backslash(c);
    }
    *p = 0;
    yylval.sym = new Symbol;
    yylval.sym->name = new string(sbuf);
    return yylval.sym->type = STRING;
  }
```

Если использовать lex/flex, то нужно добавить к файлу с описанием лексики аналогичные разделы.

```
\$[0-9]+   {
              istringstream iss(yytext + 1);
              int n;
              iss >> n;
              if (n == 0)
                throw (string) "strange $...";
              yylval.narg = n;
              return ARG;
           }
\"[^"]*\"  {
              string istr(yytext + 1), so("\b\n\r\t\f"),
                    si[] = {"\\b", "\\n", "\\r", "\\t", "\\f"};
              istr.erase(istr.length() - 1);
              int i, p;
              for (i = 0; i < so.length(); i++)
                 while ((p = istr.find(si[i])) != string::npos)
                    istr.replace(p, 2, so.substr(i, 1));
              yylval.sym = new Symbol;
              yylval.sym->name = new string(istr);
              return yylval.sym->type = STRING;
           }
```

В code.cpp появятся определения новых переменных. Возврат из подпрограмм организуем не синтаксически, а семантически, при помощи returning.

```
Inst *progbase = prog; //начало текущей подпрограммы
int returning; //1, если находимся в режиме возврата
static Datum *bp; //указатель базы локальных переменных
```

В initcode добавятся следующие строки.

```
fp = frame;
returning = 0;
progp = progbase; //вместо progp = prog;
```

Незначительно изменится цикл в execute.

```
for (pc = p; *pc != STOP && !returning;)
  (*(*pc++))();
```

Изменения, связанные с returning, произойдут в while_code.

```
while (d.val) {
  execute(*(Inst**)savepc); //тело
  if (returning)
    break;
  execute(savepc + 2); //условие
  d = pop();
}
if (!returning)
```

```
    pc = *(Inst**)(savepc + 1); //следующий оператор
```
и в if_code.
```
  if (!returning)
    pc = *(Inst**)(savepc + 2); //следующий оператор
```

Теперь при печати с помощью print не нужно всегда печатать переход на новую строку — убираем endl из prexpr. Для печати строк понадобится новая функция.

```
void prstr() { //печать строки
  cout << *(((Symbol*) *pc++)->name);
}
```

Определим функции для работы с подпрограммами.

```
void define(Symbol *sp) {
  sp->deffn = (Inst) progbase;
  progbase = progp;
}

void call() { //вызов подпрограммы
  Datum d;
  d.sym = (Symbol*)(pc + 2);  //адрес возврата
  push(d);
  d.sym = (Symbol*)bp;  //адрес базы локальных параметров
  push(d);
  bp = stackp;   //новое значение базы
  execute((Inst*)((Symbol*)*pc)->deffn); //обращение к
          //записи в таблице имен для вызова подпрограммы
  returning = 0;
}

void ret() { //общий возврат из подпрограммы
  Datum d = pop();
  bp = (Datum*) d.sym;
  d = pop();
  pc = (Inst*)d.sym;
  stackp -= (long)*(pc - 1); //удаление аргументов
  returning = 1;
}

void funcret() { //возврат из функции
  Datum d = pop(); //значение функции
  Symbol *sp = *(*(Symbol***)(stackp - 2) - 2);
        //обращение к таблице имён через сохранённый pc
  if (sp->type == PROCCALL)
    throw *sp->name + " (proc) returns value";
  ret();
```

```
  push(d);
}

void procret() { //возврат из процедуры
  Symbol *sp = *(*(Symbol***)(stackp - 2) - 2);
  if (sp->type == FUNCCALL)
    throw *sp->name + " (func) returns no value";
  ret();
}

double *getarg() { //возвращает указатель на аргумент
  int narg = (long)*pc++; //номер аргумента
  long nargs = (long)*(*(Symbol***)(bp - 2) - 1);
               //обращение к коду через bp и сохранённый pc
  if (narg > nargs)
    throw *(*(*(Symbol***)(bp - 2) - 2))->name
                                  + " not enough arguments";
  return &(bp - nargs + narg - 3)->val;
}

void arg() { //перенести аргумент в стек операций
  Datum d;
  d.val = *getarg();
  push(d);
}

void argassign() {   //записать значение с вершины стека
  Datum d;                                    //в аргумент
  push(d = pop());
  *getarg() = d.val;
}
```

Определим также функцию для ввода данных.

```
void varread() { //считывает значения для переменных
  Datum d;
  Symbol *var = (Symbol*) *pc++;
  if (!(cin >> var->val))
    throw "non-number read into " + *var->name;
  d.val = var->val;
  var->type = VAR;
  push(d);
}
```

Отразим в hoc.h изменения в текстах программы. В определении Symbol появится поле deffn.

```
void (*deffn)(); //FUNCDEF, PROCDEF
```

Декларация внешних переменных приобретет следующий вид.

```
extern Inst prog[], *progp, *progbase;
extern int indef;
```

К списку функций нужно добавить следующие.

```
defonly(const string&), define(Symbol*), call(), procret(),
  funcret(), arg(), argassign(), prstr(), varread();
```

Корректируем makefile и запускаем сборку. Проверяем, набрав в файл fib.hoc текст

```
func fib() {
  if ($1 < 3) return 1 else return fib($1 - 1) + fib($1 - 2)
}
fib(32)
```

и запустив `hoc<fib.hoc` с командой строки.

Подсчет числа аргументов у подпрограмм вводится только для контроля за стеком, например, для отслеживания случаев, когда внутри подпрограммы запрашивается аргумент с номером, большим общего числа аргументов.

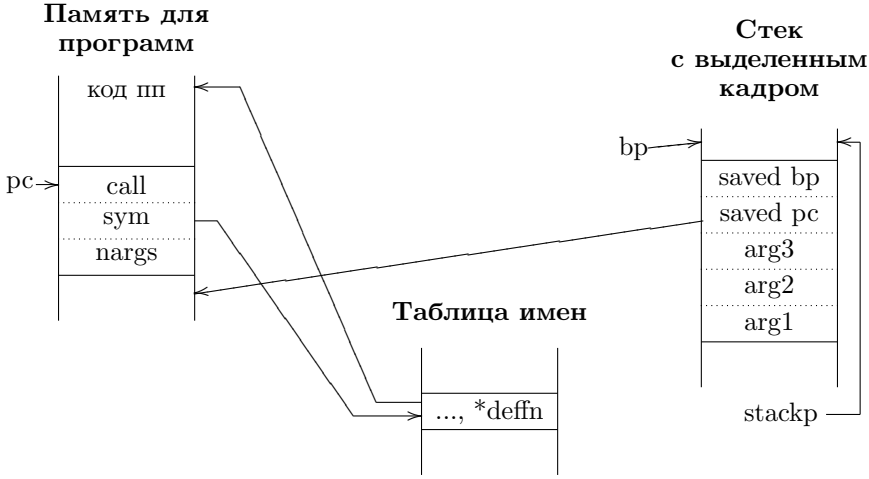
Для организации вызовов подпрограмм с параметрами в стеке создается специальный кадр подпрограммы. В кадре хранятся адрес возврата, адрес базы параметров, указатель на запись в таблице имен для подпрограммы (для лучших сообщений об ошибке) и число аргументов при вызове (для означенного ранее контроля). Аргументы и результат функции также сохраняются в стеке.

В deffn-продукции использовалась возможность помещать семантику в середину синтаксического правила. Это использовалось для проверки допустимости синтаксических конструкций, возможных только внутри подпрограмм: работа с параметрами и возврат. Функция defonly (только в определении) генерирует исключение при встрече таких конструкций вне тела подпрограмм. Работа defonly основана на переменной indef — она устанавливается в 1 вначале определения и в 0 в конце. Это случай, когда правильность конструкции проще проверить через семантику, а не синтаксис. Использование переменной returning также упрощает синтаксис — после ее установки все операторы до конца подпрограммы пропускаются без исполнения.

Подпрограммы заносятся в начало массива prog, вызывая смещение начала кода для текущих выражений (progbase), поэтому в вызове execute в main нужно заменить параметр. При переопределении подпрограмм память от старого кода не освобождается. При сбросе синтаксического анализатора, например, при ошибке, подпрограммы остаются в памяти.

Процесс вызова подпрограммы с тремя аргументами можно изобразить приведённой схемой.

Проверочные примеры можно добавить к проекту. Фиксируем изменения и обновляем рабочую копию. Отчет о проведённых фиксациях можно получить командой svn log.



4. ПОРЯДОК ВЫПОЛНЕНИЯ И ОТЧЕТ ПО КУРСОВОЙ РАБОТЕ

4.1. Порядок выполнения курсовой работы

После каждого этапа выполнения работы следует проводить фиксацию изменений. На сложных этапах (операторы, подпрограммы) можно проводить дополнительные фиксации промежуточных результатов.

После подготовки всех необходимых на текущем этапе текстов программ следует при помощи make собрать соответствующий этапу транслятор, а затем провести его тестирование.

4.2. Отчет по курсовой работе

Отчет по работе выполняется на отдельных бланках или листах и должен содержать:

1) цель исследования;
2) краткую характеристику реализованного языка;
3) листинг программ;
4) описание части программы, реализующей определяемое вариантом задания определение (изменения в базовой программе, новые типы, переменные, подпрограммы и т. п.);
5) выводы.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Назначение созданного транслятора.
2. Характеристики программы yacc/bison (lex/flex).
3. Способы реализации операторов и подпрограмм.
4. Как реализовать поддержку стандартной функции log2 — двоичного логарифма?
5. Как добавить поддержку новой стандартной константы?
6. Как добавить поддержку простейших, задаваемых цепочкой строчных английских букв переменных, в имени у которых значимыми являются только первые два символа, к hoc версии 2?
7. Сформулируйте как можно больше конкретных расширений для hoc.
8. Какие массивы проще для реализации на уровне поддержки синтаксиса обычные в стиле си или ассоциативные? Поясните примерами.
9. Чем текстовые файлы для make принципиально отличаются от большинства тестовых форматов на основе чистого текста (plain text)?
10. В чем недостатки стандартного способа контроля за ошибками?
11. Как зависит скорость поиска символа в таблице имен компилятора от размера этой таблицы: линейно, логарифмически, экспоненциально, никак? Эффективно ли таблица имен использует память?
12. Разобрать исполнение if_code.
13. Описать процесс возврата из подпрограммы.
14. Почему присваивание не всегда сворачивается к выражению?
15. Построить дерево разбора для заданной синтаксической конструкции hoc.
16. Написать код, генерируемый при разборе заданной синтаксической конструкции.
17. Какое значение возвращается при обращении к функции, невозвращающей значения?

6. ВАРИАНТЫ РАБОТ

Курсовой работа по теме “Разработка транслятора языка программирования на SVN, Yacc и C++” имеет 16 вариантов заданий, отличающихся друг от друга расширениями к описанному языку hoc:

1. Добавить поддержку си-оператора do и поддержку комментариев си++.
2. Реализовать поддержку си-оператора for, используя скобки вместо точек с запятой, например, for(i=1(i<10)i=i+1).

3. Добавить поддержку си-операций инкремента и декремента как постфиксной, так и в префиксной формах.
4. Реализовать составные присваивания в стиле си, а именно +=, -=, *=, /=, ^=.
5. Реализовать поддержку операторов си break и continue.
6. Исправить операции && и ||, сделав так, чтобы их второй аргумент вычислялся только в случае необходимости.

7-8 Добавить поддержку локальных переменных со статическим/автоматическим распределением памяти, используя служебное слово static/auto.

9. Реализовать поддержку многомерных ассоциативных массивов с возможностью удаления отдельных элементов, отдельных размерностей и всего массива в целом.
10. Реализовать операторы unless (отличается от if только инвертированием условия и отсутствием else-альтернативы) и until (отличается от while только инвертированием условия).
11. Добавить поддержку констант, реализовав конструкцию инициализации со служебным словом const. Встроенные переменные превратить в константы.
12. Добавить распечатку получаемого после компиляции кода в виде листинга мнемокодов.
13. Добавить распечатку выполняющихся инструкций мнемокодами.
14. Реализовать очистку памяти от ставших ненужными символьных строк.
15. Реализовать освобождения памяти от старого кода при переопределении подпрограмм.
16. Сделать размер стека выражений или массива для программ динамическим.

ЛИТЕРАТУРА

1. *Ахо А., Сети Р., Ульман Д.* Компиляторы / Ахо А., Сети Р., Ульман Д. — М., СПб., Киев: Вильямс, 2003.
2. *Керниган Б. В. , Пайк Р.* Unix — универсальная среда программирования / Керниган Б. В. , Пайк Р. — М.: Финансы и статистика, 1992.
3. *Страуструп Б.* Язык программирования C++ / Страуструп Б. — М.: Binom Publisher, СПб: Невский диалект, 1999.

ПРИЛОЖЕНИЕ 1

Пример файла makefile для начальной версии программы

```
hoc: hoc.cpp.y hoc.cpp.l makefile
        -lex hoc.cpp.l
        yacc hoc.cpp.y
        g++ -o hoc y.tab.c

hoc.cpp.l:
        echo >hoc.cpp.l

clean:
        rm -f y.tab.c lex.yy.c hoc
```

ПРИЛОЖЕНИЕ 2

Некоторые дополнительные команды системы Subversion

`svn st` — файлы, изменившиеся с последней фиксации (опция `-q` скрывает неверсионированные файлы)

`svn log` — журнал фиксаций

`svn diff ФАЙЛ` — показ результатов текущего редактирования по файлу

`svn diff` — показ результатов текущего редактирования по всем версионированным файлам

`svn revert ФАЙЛ` — отказ от всех изменений с момента последней фиксации в заданном файле

`svn cat ФАЙЛ -r ВЕРСИЯ` — печать содержимого файла заданной версии

`svn ps svn:keywords Id ФАЙЛ` — установка поддержки раскрытия значения Id в заданном файле

`svn ren ФАЙЛ1 ФАЙЛ2` — версионированное переименование файла

`svn list` — содержимое репозитория, соответствующего рабочей копии

`svn list СЕРВЕР` — содержимое репозитория на заданном сервере

`svn del ФАЙЛ` — версионированное удаление файла

`svn info ФАЙЛ` — информация о текущем репозитории

ОГЛАВЛЕНИЕ

Владимир Викторович Лидовский

РАЗРАБОТКА ТРАНСЛЯТОРА ЯЗЫКА ПРОГРАММИРОВАНИЯ НА SVN, YACC и C++

Методические указания к курсовой работе по предмету
“Системное программное обеспечение”

Ведущий редактор *О. В. Филиппова*

Технический редактор *О. В. Филиппова*

Оригинал-макет подготовлен в пакетах Plain-TEX и XY-pic

Подписано в печать ?.?.2017 Формат 60x84/16
Печать на ризографе. Усл. печ. л. 2. Уч.-изд. л. 2,19
Тираж ? экз. Заказ № ?

Издательско-типографский центр МАИ
Москва