КОМПЬЮТЕРНЫЕ НАУКИ

УДК 681.3.06

РОЗРОБКА ТРАНСЛЯТОРА ПРОСТОЇ МОВИ ПРОГРАМУВАННЯ

ЦИМБАЛ О.М., ЦЕХМІСТРО Р.І.

Описується послідовність розробки транслятора простої мови програмування у складі лексичного та синтаксичного аналізаторів, блоків керування таблицями та інтерпретації. Надаються схеми синтаксичного аналізу та елементи програмного коду транслятора. Програмне забезпечення реалізовано у середовищі розробки Visual C++ 2005 (Beta).

Вступ

Мови програмування є засобами формалізації задумів фахівців у реалізації їх інженерних або наукових завдань. Від гнучкості мовних засобів залежить простота або складність опису проблеми та способів її програмного розв'язання.

Звичайно, в основному доводиться працювати з мовами програмування, що розроблені спеціалізованими фірмами - розробниками програмного забезпечення. Такі мови у поєднанні з середовищами розробника є надійним інструментом програміста. Під час розробки програми програміст має вкласти завдання у «прокрустове ложе» тієї мови програмування, якою він володіє, і відповідно до концепції, що реалізована мовою програмування. Але чи завжди ця мова достатньо відповідає опису проблеми або методам її розв'язання?

При вирішенні завдань практичного плану можуть знадобитися особливі, орієнтовані лише на вирішення конкретного класу завдань мовні засоби, за допомогою яких опис даних та методів їх обробки виглядатиме найбільш відповідним конкретній ситуації. Таким чином, може виникнути проблема опису власної мови програмування.

Метою дослідження є опис послідовності розробки транслятора простої мови програмування. Ставляться завдання визначення синтаксису та граматик простої мови програмування, реалізації лексичного та синтаксичного аналізаторів, інтерпретатора. Засобом створення транслятора є мова програмування Сі.

1. Загальні принципи побудови трансляторів

Транслятором називають програму перекладу (трансляції) початкової програми, записаної вхідною мовою, в еквівалентну їх об'єктну програму. Якщо мова високого рівня є вхідною, а мова Асемблер (або машинна мова) – вихідною, транслятор називають компілятором. [1] Саме за допомогою компіляторів створена більшість програмного забезпечення ЕОМ.

Іншим різновидом транслятора є інтерпретатор - програма, що забезпечує переклад початкової програми у програму, записану проміжною алгоритмічною мовою програмування. Найпростішим способом реалізації інтерпретатора вважається варіант, коли початкова програма спочатку повністю транслюється у машинні команди, а потім одразу ж і виконується.

У такій реалізації інтерпретатор не дуже відрізняється від компілятора, окрім того, що результуюча програма буде недоступною для користувача.

Більшість інтерпретаторів діє таким чином, що виконує початкову програму послідовно, по мірі її надходження на вхід інтерпретатора. При цьому користувач не має чекати на завершення компіляції усієї програми, він може послідовно вводити початкову програму і одразу ж спостерігати результат виконання по мірі вводу команд [1].

При цьому виявляється суттєва відмінна особливість інтерпретатора – якщо він виконує команди по мірі їх надходження, то не може виконувати оптимізацію початкової програми. Таким чином, стадія оптимізації у інтерпретаторі відсутня.

Не усі мови програмування можуть бути реалізовані за допомогою інтерпретатора. Мова не може бути інтерпретована по мірі надходження команд, якщо вона не припускає появу звернення до функцій і структур даних раніше їх безпосереднього опису.

Прикладами мов, що припускають інтерпретацію, є мови програмування Lisp, HTML, Jawa, JawaScript, також деякі версії мови Basic.

Узагальнена схема роботи інтерпретатора показана на рисунку.

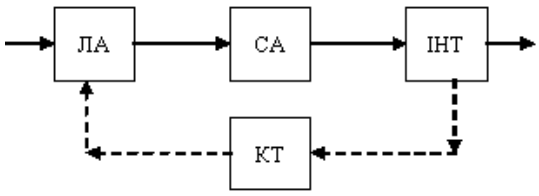

Схема інтерпретатора

На схемі ЛА означає лексичний аналізатор, СА - синтаксичний аналізатор, ІНТ – блок інтерпретації, КТ – блок керування таблицями.

Таким чином, робота інтерпретатора полягає у послідовному виконанні стадій лексичного, синтаксичного аналізу та інтерпретації програми засобами проміжної алгоритмічної мови програмування.

2. Опис узагальненої простої мови програмування

Реалізуємо інтерпретатор простої мови програмування (SPL – simple programming language). Така проста мова є подібною до спрощеного Алголу [2]. Лексичні особливості SPL описано у [3].

Алфавіт SPL складається з символів латинської мови, десяткових цифр, множини службових символів: {“+”, “-“, “*“, “/“, “%“, “=“, “(“, “)“, “,“, “;“, “\t“, ““} та пробілу. З символів алфавіту складаються лексеми – мінімальні мовні ланцюжки із самостійним змістом.

Ідентифікатором у SPL є послідовність літер та цифр, яка починається з літери. Беззнаковим числом є послідовність цифр. З точки зору синтаксису SPL усі ідентифікатори є однаковими і утворюють лексему IDEN.

Числа без знаку утворюють лексему NUMB.

Окрім IDEN та NUMB у SPL визначаються спеціальні одинадцять ідентифікаторів – ключових слів мови. До них належать {“if”, “while”, “return”, “print”, “read”, “begin”, “end”, “int”, “const”, “then”, “do”}.

Символи пробілу “\n” та “\t” припустимі у кожному місці програми, за винятком ідентифікаторів та беззнакових чисел. Інші службові символи утворюють окремі лексеми мови.

SPL як проста мова має лише цілі скалярні дані – константи (ідентифікатори, числа) та змінні. Змінні можуть бути глобальними та локальними.

Вирази SPL будуються з констант, змінних та покажчиків функцій (з круглими дужками для позначення списку параметрів). Знаками операцій є “+”, “-“, “*“, “/“, “%“. Вирази мають вигляд: v, c, (e_1), f(e_1, e_2,…,e_n), $e_1 + e_2$, $e_1 – e_2$, $e_1 * e_2$, e_1 / e_2, $e_1 \% e_2$, $+ e_1$, $–e_1$, де v – змінна, c – константа, f – ідентифікатор функції, e_1, e_2,…,e_n – вирази.

У простій мові програмування визначаються такі оператори:

присвоєння	v = c;
умовний	if e then sl end; // виконується, якщо e>0
циклу	while e do sl end; // виконується, якщо e>0
повернення	return e;
введення	read e;
друку	print e;

де v – змінна, e – вираз, c – константа, sl – список операторів.

Програма на SPL складається з функцій, означень глобальних функцій та констант, що мають бути описані до їх використання (у цьому сенсі приклади, наведені у [4], є неточними).

Означення записуються у такому вигляді:

int v_1, v_2 …, v_k ;

const $c_1=n_1$, $c_2=n_2$,…, $c_k=n_k$;

де {n_1, n_2 …, n_k}, {v_1, v_2 …, v_k}, {c_1, c_2 …, c_k} – множини чисел, ідентифікаторів змінних та констант, відповідно.

Функція function мови SPL із списком параметрів (p_1, p_2,…,p_n) має вигляд:

```
function (p1, p2,…,pn)
begin
    dcl //означення констант та змінних
sl //список операторів
end
```

Список параметрів може бути порожнім.

Одна із функцій програми має назву main. Саме з main починається виконання програми.

У функціях мови SPL допускається пряма та непряма форми рекурсії. Функція може повертати значення або оператором return в місце виклику, або за допомогою зміни значень глобальних змінних. Функція закінчує роботу, якщо досягнуто ключового слова end, або коли значення повертається за допомогою return.

Нижче наводиться приклад SPL-програми, що здійснює обчислення функції x^y:

```
exp(a,b)
begin    int z;
    z=1;
    while b do
        if b%2 then z=z*a end;
        a=a*a; b=b/2;
    end;
    return z;
end


main()
begin    int x,y,z;
    read x; read y;
    print exp(x,y);
end
```

3. Опис лексичного та синтаксичного аналізаторів

Текст програми, що є реалізацією інтерпретатора мови SPL, розпочинається із директив підключення заголовочних файлів:

```
#include<iostream>      // за реалізацією VC++ 8.0 (beta)
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<ctype.h>
#include<string.h>
#include “pt01.h”                  // опис прототипів
```

Останній з них містить прототипи усіх використаних у тексті транслятора функцій (їх вигляд легко зрозуміти знаведених далі у тексті функцій) та визначення структур, що забезпечують режими роботи інтерпретатора:

```
typedef  struct        {int cod,opd;}cmd;
typedef  struct        {char *name; int what,val;}odc;
typedef  struct        {char *name; int isd,cpt,start;}fnd;
```

Також слід визначити глобальні змінні програми:

```
cmd TCD[300], cmd *pcmd=TCD;
char TNM[400], *ptn=TNM, char *lstr;
static char nch='N';

odc TOB[100];
odc *pto=TOB; odc *ptol=TOB;
fnd TFN[30], fnd *ptf=TFN;

int adrnm,cpnm,out,lval,lex;
int cgv=0,clv=0,tc=0,nst=0;
int  st[500],sp,t,p;

using namespace std;
FILE *PF; errno_t err;
```

Під час проведення лексичного аналізу необхідно визначити лексеми, з яких складається програма, зокрема службові слова, якими є "if", "while", "return", "print", "read", "begin", "end", "int", "const", "then", "do". Вказані службові лексеми кодуються відповідними константами IFL, WHILEL, RETRL, PRITL, READL, BEGINL, ENDL, INTL, CONSTL, THENL, DOL. Також до лексем належать знаки операцій та роздільники, беззнакові цілі дані з кодом NUMB та значенням lval, ідентифікатори із кодом IDEN (входять до таблиці ідентифікаторів TNM), ідентифікатор кінця файла із кодом EOF.

Інформацію про коди лексем можна зберегти за допомогою перелічення:

```
enum {IFL=257, WHILEL, RETRL, PRITL, READL, BEGINL,
ENDL, INTL, REALL, CONSTL, THENL, DOL, NUMB, IDEN};
```

Запуск програми транслятора здійснюється у звичайному стилі консольної програми:

```
void main(int argc,char *argv[])
{printf("\n SPL-translator 02 \n");
 if (argc>1 && argv[1])
 {analys(argv[1]);
 cout << "\n Analys is finished \n \n";
 interp();
 cout << "\n Interpreting is finished \n \n";
 }
 else printf("\n no program file \n");
}
```

Для виконання SPL-програми необхідно викликати її з командного рядка операційної системи, вказавши як параметр ім'я файла з текстом SPL-програми, наприклад:

```
D:\ My Projects\PT01.exe 2.spl
```

Якщо вхідний SPL-файл задано, аналіз програми забезпечується викликом функції analys():

```
void analys (char *nf)
{err=fopen_s(&PF,nf,"r"); // відкриття SPL-файлу
 if(!err)
 {get (); // введення нової лексеми
 prog (); // аналіз блоку програми
 }
 else printf("Error %s file isn't opened",nf);
}
```

За введення лексеми відповідає функція get():

```
void get ()
{if(nch= ='N')nch=getc(PF);
 if(nch= =EOF) {lex=EOF;return;}
 while(isspace(nch))
 {if(nch= ='N')nst++;nch=getc(PF);}
 if(isdigit(nch))number();
 else if(islower(nch)) word();
 else if(index(nch)) {lex=nch;
 nch=getc(PF);}
     else if(nch= =EOF)lex=EOF;
 else printf("\n error %c illegal simbol",nch);
}
```

Введення лексем типу "слово" та "ціле число" забезпечується функціями number() та word():

```
void number ()
{for(lval=0;isdigit(nch);nch=getc(PF))  lval=lval*10+nch-'0';
 lex=NUMB; }

void word ()
{static char *serv[]={"if","while","return","print","read",
"begin","end","int","real","const","then","do"};
 static int cdl[]={IFL,WHILEL,RETRL,PRITL,READL, BEGINL,
ENDL,  INTL,REALL,CONSTL,THENL,DOL};
 char tx[40]={""},*p;
 int i;
 bool keyword=false;
 for(p=tx;(islower(nch) || isdigit(nch)) && !isspace(nch)
&& !index(nch) && nch!=10;nch=getc(PF))
 p="\0";
 for(i=0;i<12;i++)
```

```
if(strcmp(serv[i],tx)==0){lex=cdl[i];keyword=true;break;}
if(!keyword){lex=IDEN; lstr=add(tx); lval=(int)lstr; }
}
```

Пошук лексем-роздільників забезпечує index():

```
bool index(char b)
{bool ft=false;
 char a[11]={' ( ' , ' ) ' , ' , ' , ' : ' , ' ; ' , ' = ' , ' + ' , ' - ' , ' * ' , ' / ' , ' % '};
 for(int i=0;i<11;i++)
    if(a[i]==b){ft=true;break;}
 return ft;
}
```

Ідентифікатори, що не знайдені серед списку службових лексем та не введені раніше, мають бути додані до таблиці ідентифікаторів TNM[40], що й забезпечує функція add():

```
char *add (char * nm)
{char *p;
  for(p=TNM;p<ptn;p+=strlen(p)+1)
     if(strcmp(nm,p)==0)return (p);
 if(atoi(ptn+=strlen(nm)+1)>atoi(TNM+400))
    printf("\n overflow TNM");
 strcpy (p,nm);
 return p;
}
```

Зазначимо, що введення лексем проводиться послідовно, при цьому застосовується непряма рекурсія функцій get(), word(), number().

Синтаксис SPL (відповідно до [3]) описується розширеними граматиками із використанням символів [,] в регулярних виразах у правих частинах правил.

Якщо r – регулярний вираз, що задає множину R, то [r] – регулярний вираз для множини R И {e}, де - e порожній ланцюжок. Термінальними символами розширеної граматики є лексеми, записані малими літерами. Нетерміналами є послідовності літер, що позначають синтактичні конструкції програми:

PROG – програма; DCONST – означення констант; DVARB – означення змінних; DFUNC – означення функції; CONS – константа; PARAM – параметри; BODY – тіло функції; STML – список операторів; STAT – оператор; EXPR – вираз, TERM - доданок, FACT - множник, FCTL – список виразів.

Повний синтаксис мови SPL має вигляд:

```
PROG → ( DCONST | DVARB | DFUNC ) eof
DCONST → constl CONS ( ',' CONS ) * ';'
CONS → iden '=' [ '+' | '–' ] numb
DVARB → intl iden ( ',' iden ) * ';'
DFUNC → iden PARAM BODY
PARAM → '(' [ iden {',' iden}] ')'
BODY → beginl (DCONST | DVARB ) * STML endl
STML → STAT (';' STAT ) *
STAT → ifl EXPR thenl STML endl |
     whilel EXPR dol STML endl |
     retrl EXPR | pritl EXPR | readl iden |
     iden '=' EXPR
EXPR → [ '+' | '-' ] TERM ( ( [ '+' | '-' ] ) TERM)*
TERM → FACT ( ( '*' | '/' | '%' ) FACT )*
FACT → '(' EXPR ')' | numb | iden [ '(' [FCTL] ')' ]
FCTL → EXPR ( ',' EXPR ) *
```

Конструкція PROG забезпечує обробку тексту файла програми і відповідає однойменній функції:

```
void prog()
{fnd *p;
 lex;
 while(lex!=EOF)
  switch(lex)
    {case CONSTL: dconst ( );break;
     case INTL:   dvarb ( ); break;
     case IDEN:   dfunc ( ); break;
 default:         printf("\n %d syntax error (prog)",nst);
    }
 p=fmain();
  if(p){adrnm=p->start;cpnm=p->cpt;}
}
```

Обробку оголошень констант програми забезпечує конструкція DCONST і функції dconst () та cons( ):

```
void dconst()
{do    {get();cons();
    }while(lex==',');
     exam(';');
}

void cons()
{char *nm=(char*)lval;int s;
 exam(IDEN);exam('=');
 s=(lex=='-'?-1:1);
 if(lex=='+' || lex=='-')get();
 newob(nm,1,s*lval);exam(NUMB);
}
```

Синтаксичний аналіз використовує функцію exam(), яка перевіряє, чи має наступна лексема код lx. Якщо відповідність коду не знайдена, виводиться повідомлення про синтаксичну помилку:

```
void exam(int lx)
{if(lx!=lex) printf("\n %d syntax error (exam)",nst);
 get ( );
}
```

Функція dvarb() відповідає конструкції DVARB, що забезпечує обробку оголошень змінних:

```
void dvarb()
{do{get();
     newob((char*)lval,(out?2:3),(out?cgv++:++clv));
     exam(IDEN);
     }while(lex==',');
     exam(';');
}
```

Обробка функцій мови SPL має забезпечуватися викликом dfunc() відповідно до конструкції DFUNC:

```
void dfunc()
{char* nm=(char*)lval;int cp,st;
 get();out=0;
 cp=param();
  st=body();
  out=1;pto=ptol;
 TOB;
 defin(nm,cp,st);
}
```

У своєму складі dfunc () містить послідовні виклики функцій: get() – введення чергової лексеми, param () – обробки параметрів функції та body () – обробки тіла функції:

```
int param()
{odc *p;int cp=0;
exam('(');
if(lex!=')')
 {newob((char*)lval,3,++cp); exam(IDEN);
while(lex==',')
{get();newob((char*)lval,3,++cp);exam(IDEN);}
}exam(')');
  for(p=ptol;p<pto;p++)  p->val-=cp+3;
 return cp;
}


int body()
{int st;
exam(BEGINL);clv=0;
while(lex==CONSTL || lex==INTL || lex==REALL)
    if(lex==INTL)dvarb();
    else if(lex==REALL)dvarb();
    else dconst();
    st=gen(INI,clv);
    stml();exam(ENDL);
    gen(OPR,10);
 return st;
}
```

Слід зауважити, що наведені вище param () та body () функціонують у відповідності до синтаксичних конструкцій PARAM та BODY.

Синтаксична конструкція STML забезпечує обробку списку операторів, хоча детальну обробку забезпечує найбільш складна конструкція мови SPL – STAT. Цим схемам оброки відповідають дві функції: stml () та stat (). Власне, перша з них здійснює послідовний (в разі потреби) виклик другої – із безпосереднім аналізом операторів if – then, while – do, return, print, read.

```
void stml()
{stat();
while(lex==';') {get(); stat();}}


void stat()
{odc *p;                int t1,t2;
switch(lex)
 {case IFL:             get();expr();exam(THENL);
                        t1=gen(JMC,0);
                        stml();exam(ENDL);
                        TCD[t1].opd=tc;break;
 case WHILEL:           get();t1=tc;
                        expr();exam(DOL);
                        t2=gen(JMC,0);stml();
                        gen(JMP,t1);TCD[t2].opd=tc;
                        exam(ENDL);break;
 case RETRL:            get();expr();
                        gen(OPR,9);break;
 case PRITL:            get();expr();
                        gen(OPR,2);break;
 case READL:            get();p=findob((char*)lval);
                        gen(OPR,1);
                        if(p->what==1)
                        printf("\n %s isn't variable",lval);
                        gen(p->what==2?STE:STI,p->val);
                        exam(IDEN);break;
 case IDEN:             p=findob((char*)lval);get();
                        exam('=');expr();
    if(p->what==1)printf("\n %s isn't variable",p->name);
    gen(p->what==2?STE:STI,p->val);break;
```

```
case ENDL: break;
default:                printf("\n %d syntax error (stat)",nst);

}}
```

Як видно зі схеми конструкції STAT, її складовими є вирази EXPR або списки операторів STML. EXPR є комбінацією операцій додавання-віднімання, множення-ділення виразів та констант і складається з комбінацій конструкцій TERM та опосередковано – FACT і FCTL. Вказаним вище конструкціям відповідають функції expr(), term(), fact(), fctl():

```
void expr()
{int neg=(lex==' - ');
 if(lex==' + ' || lex==' - ')get ();
 term ();
if(neg)gen(OPR,8);
 while(lex==' + ' || lex==' - ')
 {neg=(lex==' - ' ? 4 : 3); get();term();gen(OPR,neg);
}}

void term()
{int  op;
 fact();
 while(lex=='*' || lex=='/' || lex=='%')
 {op=(lex=='*' ? 5 : (lex=='/' ? 6:7));
get();fact();gen(OPR,op);}
}

int fctl ()
{int cf=1;
expr();
 while(lex==',' ) {get();expr();cf++;}
 return cf;
}

void fact()
{char *nm;
 int cp;
 odc *p;fnd *pl;
switch(lex)
 {case IDEN: nm=(char*)lval;get();
      if(lex=='(' )
      {get();cp=(lex==' ) ' ? 0 : fctl());
      exam(')');pl=eval(nm,cp);
       gen(LIT,cp);cp=gen(CAL,pl->start);
       if(!pl->isd)pl->start=cp;
      }
      else {                p=findob(nm);
 gen(p->what==1 ? LIT :
(p->what==2 ? LDE:LDI),p->val);
      }
      break;
 case '(': get();expr();exam(')');break;
 case NUMB: gen(LIT,lval);get();break;
 default:                printf("\n %d syntax error (fact)",nst);
 }
}
```

Слід одразу зазначити, що у текстах наведених функцій реалізації синтаксичного аналізатора включені додаткові функції контролю таблиць, пов'язані вже з інтерпретацією програми.

4. Програмна реалізація функцій інтерпретатора

Результатом роботи блоку синтаксичного аналізу є проміжний код, що надалі обробляється функціями інтерпретатора програми.

Набір функцій інтерпертації формує SPL-процесор [4], який складається з: пам'яті програми, що моделюється масивом TCD та змінними adrnm, cpnm, cgv; стеку st зі значеннями змінних та констант програми. Стек st характеризується покажчиками: вершини стека – t, початку області локальних даних функції – sp (запису активації). Покажчик p – вказує на поточну команду, обрану у STD.

Під час виклику функції SPL у стеку st виділяється запис активації виклику, до якого вносяться дані про значення фактичних параметрів функції $p_1, p_2, \ldots, p_n$; кількість параметрів n; адреса повернення у TCD; адреса попереднього запису активації у стеку; локальні та тимчасові дані, використані під час обчислення виклику функції.

У будь-який момент виконання SPL-програми доступними є глобальні дані, записані в основі стека st зі зміщенням k для адресації st[k], та локальні дані у верхній його частині, на яку вказує sp. Для параметрів, розташованих нижче sp, зміщення становить $k<0$; для локальних даних, записаних вище sp, $k>0$ і адресація має вигляд st[sp+k].

SPL-програма формується в процесі синтаксичного аналізу у масиві TCD, кожен елемент якого має два поля: TDC[i].cod – із кодом операції, TDC[i].opd – операція над стеком.

Операції над стеком задають кодами, що містяться у спеціальному переліченні:

```
enum    {OPR,LIT,LDE,LDI,STE,STI,CAL,INI,JMP,JMC};
```

де OPR – виконати операцію a над вершиною стека; LIT – завантижити в стек константу a; LDE, LDI – записати в стек значення глобальної або локальної змінної зі зміщенням a;

STE, STI – запам'ятати значення вершини стека у глобальній (локальній) змінній зі зміщенням a; CAL – викликати функцію з точкою входу a; INI – збільшити t на a (виділення пам'яті); JMP – здійснити безумовний перехід на команду a; JMC – виконати умовний перехід на команду a, якщо верхній елемент стека менший або рівний 0 (вилучення зі стека).

У команді з кодом OPR допустимими є операції: ввести в стек число зі стандартної консолі stdin (a=1); вивести верхнє значення зі стека у консоль stdout (a=2); виконати над двома верхніми елементами операцію '+' або '-', '*', '/' , '%' (коди a=3 ё7); змінити знак верхнього елемента стека (a=8); повернутися з функції зі збереженням результату у верхньому елементі стека (a=9); зупинити виконання SPL-програми (a=10). Повернення з main призводить до друку результата, при цьому p= –1. Досягнення лексеми end функції (повернення зупинкою) дає p = –2.

Функція interp() починає та завершує виконання SPL-програми, забезпечує загальне керування процесом:

```
void interp()
{int i;
 t=-1;printf("SPL: interpreting \n");
 for(i=0;i<cgv;i++)push(0);
 if(cpnm)
 {printf("%d >",cpnm);
 fflush(stdout);
 for(i=0;i<cpnm;i++) push(read());
 }
 push(cpnm);push(-2);push(-1);
 sp=t;p=adrnm;
 do {comman(); p++;
 }while(p>=0);
 if(p==-1)printf("\n Result: %d \n ",st[t]);
}
```

Функція comman() виконує поточну команду TCD[p]:

```
void comman()
{int  a=TCD[p].opd;
 st;
 switch(TCD[p].cod)
 {case OPR: operate(a);break;
 case LIT: push(a);break;
 case LDE: push(st[a]);break;
 case LDI: push(st[sp+a]);break;
 case STE: st[a]=st[t—];break;
 case STI: st[sp+a]=st[t—];break;
 case CAL: push(p);push(sp);sp=t;p=a-1;break;
 case INI: {int i;
                    for(i=0;i<a;i++)push(0);}
                    break;
 case JMP: p=a-1;break;
 case JMC: if(st[t—]<=0) p=a-1;break;
 }
}
```

operate() забезпечує виконання операції OPR a:

```
void operate(int a)
{int j=t-1;
 switch(a)
 {case 1: printf("1>");fflush(stdout);push(read());break;
 case 2: printf("%d \n",st[t—]);break;
 case 3: st[j]+=st[t—];break;
 case 4: st[j]-=st[t—];break;
 case 5: st[j]*=st[t—];break;
 case 6: if(st[t]==0)printf("/: dilnik=0");st[j]/=st[t—];break;
 case 7: if(st[t]<=0)printf("%% dilnik=0");st[j]%=st[t—];break;
 case 8: st[t]=-st[t];break;
 case 9: j=st[sp-2];st[sp-j-2]=st[t];
t=sp-j-2;p=st[sp-1];sp=st[sp];break;
 case 10:p=-3; break;
}}
```

Функція push(a) заносить значення a у стек:

```
void push(int a) {if(t>=499)printf("\n Overflow st");
 st[++t]=a;}
```

Введення числа з потоку stdin зебезпечується read( ):

```
int read()
{int v=0,c=0;
 do {c=getchar();}
 while(isspace(c));
 if(!isdigit(c))printf("\n Input error ");
 for(v=0;isdigit(c);c=getchar()) v=v*10+c-48;
 ungetc(c,stdin);
 return v;
}
```

Інформація про константи, глобальні та локальні змінні аналізованої функції вміщується у таблицю даних: для глобальних даних out=1, для локальних out=0. При цьому використовується структура odc та масив TOB (типу odc, наведені у глобальних об'явах). Поле TOB[i].name задає покажчик імені, даного в TNM, TOB[i] .val – значення константи або зміщення змінної; TOB[i].what – вид даних. Якщо TOB[i].what рівне 1, розглядається константа, 2 – глобальна змінна, 3 – локальна змінна.

Крім того, у функціях надалі використаються покажчики (типу TOB) pto – першого вільного елемента у TOB, ptol – початку опису локальних даних.

Введення інформації про новий об'єкт з іменем nm у TOB забезпечує функція newob( ):

```
void newob(char* nm, int wt, int vl)
{odc *pe, *p;
 pe=(out ? TOB : ptol);
  for(p=pto-1;p>=pe;p—)
 {if(nm==p->name)printf("\n %s: described twicely",nm); }
 if(pto>=TOB+100)printf("\n TOB overflow");
  pto->name=nm;pto->what=wt;
  pto->val=vl;pto++;
 if(out)ptol++;
}
```

Пошук у TOB об'єкта з іменем nm забезпечує findob():

```
odc *findob(char *nm)
{odc *p;
 for(p=pto-1;p>=TOB;p—) if (nm==p->name)return p;
 printf("\n %s isn't described ",nm);
}
```

Ще одна таблиця – TFN містить дані про функції програми і відповідає типу структури fnd, також оголошеному раніше. Полями типу fnd є: TFN[i].name – ім'я у TFN, TFN[i].cpt – кількість параметрів функції, TFN[i].isd – ознака, чи описана функція (1 – описана, 0 – не описана). Поле TFN[i].start задає точку входу описаної функції або покажчик ланцюга викликів для зворотного заповнення для неописаної функції.

Функції, наведені далі, використовують покажчик ptf – першого вільного місця у TFN.

Додавання нової функції з іменем nm і значеннями компонентів у таблицю TFN забезпечується newfn(nm, df, cp, ps):.

```
fnd *newfn(char *nm, int df, int cp, int ps)
{if(ptf>=TFN+30)printf("\n TFN overflow");
 ptf->name=nm; ptf->start=ps;
 ptf->isd=df; ptf->cpt=cp;
 return ptf++; }
```

Пошук у TFN опису функції з іменем nm забезпечує:

```
fnd *findfn(char *nm)
{fnd *p;
  for(p=ptf-1;p>=TFN;p—)
 if(!strcmp(nm,p->name))return p;
 return NULL;
}
```

Виклик функції nm із cp-кількістю параметрів проваджується у eval( ):

```
fnd *eval(char *nm, int cp)
{fnd *p;
 if(!(p=findfn(nm)))
     return newfn(nm,0,cp,-1);
 if(p->cpt==cp)return p;
 printf("\n %s the number of parameters ",nm);
}
```

Обробка опису функції nm із кількістю параметрів cp та точкою входу ad виконується у defin():

```
void defin(char *nm, int cp, int ad)
{fnd *p;
 int c1=0,c2=0;
 if(p=findfn(nm))
 {if(p->isd)printf("\n %s desribed twicely",nm);
 if(p->cpt!=cp)printf("\n %s the number of parameters ",nm);
 p->isd=1;
 for(c1=p->start;c1!=ad;c1++) {c2=c1; }
 p->start=ad;
 }
 else newfn (nm,1,cp,ad);
}
```

Пошук неописаних функцій та функції main, завершення роботи з TFN забезпечує fmain():

```
fnd *fmain()
{static char nm[]="main";
 fnd *pm=NULL;
 fnd *p;//ptf=TFN;
  for(p=ptf-1;p>=TFN;p—)
 {if(!p->isd)
     printf("\n %s function isn't defined ",p->name);
 if(strcmp(nm,p->name)==0){pm=p;break;}
 }
 if(pm)return pm;
 else {printf("\n no main ");return NULL;}
}
```

Формування коду команди здійснює gen():

```
int gen(int co, int op)
{if(tc>=300)printf("\n TCD overflow");
 TCD[tc].cod=co;
  TCD[tc].opd=op;
 return tc++;
}
```

Зазначимо, що усі наведені вище ділянки програми є модифікованими варіантами інтерпретатора, наведеного у [3]. Модифікації піддані записи заголовків функцій, додано окремі локальні та глобальні змінні, замінено систему вводу-виводу.

Висновки

З практичної точки зору розглянута побудова простої Алгол-подібної мови програмування. Наведена програмна реалізація лексичного, синтаксичного аналізу, блоку інтерпретації.

Наукова цінність роботи полягає у модифікації транслятора простої мови програмування відповідно до вимог сучасних систем програмування і стандарту мови програмування Сі.

Практична значущість результатів роботи полягає у скороченні часу розробки нових мов програмування і можливості реалізації власних трансляторів вже існуючих мов.

Наступним завданням авторів є розробка інтерпретатора мови Пролог, який забезпечить подання знань системи керування інтелектуальним роботом для машинобудівного виробництва (на базі промислового робота РМ-01).

Література: 1. *Гордеев А.В., Молчанов А.Ю.* Системное программное обеспечение. СПб.: Питер, 2002. 736 с. **2.** *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. 536 с. **4.** *Проценко В.С., Чаленко П.Й., Ставровський А.Б.* Техніка програмування мовою Сі. К.: Либідь, 1993. 224 с.

Надійшла до редколегії 11.01.2006

Рецензент: д-р техн. наук, професор Кривуля Г.Ф.

Цимбал Олександр Михайлович, канд. техн. наук, докторант ХНУРЕ. Наукові інтереси: мови програмування, системи штучного інтелекту. Адреса: Україна, 61166, Харків, пр. Леніна, 14, тел. (057) 70-21-486.

Цехмістро Роман Іванович, канд. фіз.-мат. наук, старший викладач ХНУРЕ. Наукові інтереси: системи телекомунікацій, мови програмування. Адреса: Україна, 61166, Харків, пр. Леніна, 14, тел. (057) 70-21-486.

УДК 519.6:004.93

ОТБОР ИНФОРМАТИВНЫХ ПРИЗНАКОВ НА ОСНОВЕ МОДИФИЦИРОВАННОГО МЕТОДА МУРАВЬИНЫХ КОЛОНИЙ

СУББОТИН С.А., ОЛЕЙНИК А.А., ЯЦЕНКО В.К.

Впервые предлагается метод отбора признаков для построения распознающих и прогнозирующих моделей на основе модификации метода муравьиных колоний. С помощью разработанного метода проводится анализ информативности признаков для построения моделей коэффициента упрочнения деталей авиадвигателей.

Введение

Выбор информативной системы признаков является одной из наиболее важных задач теории распознавания образов. Однако удовлетворительного решения, определяющего порядок автоматического отыскания признаков посредством переработки информации, получаемой на уровне абсолютного описания объекта, пока не найдено [1].

Необходимость определения информативных признаков из имеющейся совокупности обусловлена также тем, что качество распознавания и прогноза не инвариантно к системе используемых признаков. В практических случаях характеристики классов и прогнозирующие правила определяются по экспериментальным данным ограниченного объема, поэтому добавление неинформативных признаков приводит к более сильному пересечению представителей классов в пространстве признаков, что может ухудшить качество прогноза [2, 3].

Задачу отбора информативных признаков можно сформулировать следующим образом.

Пусть имеется N результатов экспериментов, отображающих значения изучаемого отклика y в зависимости от изменения n признаков $x_1, x_2, \ldots, x_n$. Требуется определить те признаки, которые влияют на интересующий нас отклик y наиболее сильно – информативные признаки.

В настоящее время существуют различные подходы к построению информативной системы признаков. Среди них можно выделить методы полного перебора, эвристические, информационные, статистические, вероятностные и нейросетевые методы [2, 3].

Традиционные методы перебора [3] являются высоко-итеративными, что обусловлено необходимостью комбинаторного поиска приемлемого сочетания признаков, и характеризуются большими затратами машинного времени, а также для задач с большой размерностью оказываются малопригодными.

Одним из наиболее эффективных методов решения поисковых задач комбинаторной природы является метод муравьиных колоний [4], который позволяет находить оптимальное решение быстрее многих традиционных методов перебора, поскольку в его основе лежит вероятностный подход, и он не требует перебора всех возможных решений. Особенностью данного метода также является отсутствие необходимости вычисления производных. В то же время классический метод муравьиных колоний ориентирован преимущественно на решение задачи коммивояжёра.

Цель данной работы – создание метода отбора информативных признаков на основе модификации метода муравьиных колоний.

1. Метод муравьиных колоний

Метод муравьиных колоний основан на взаимодействии нескольких муравьёв (программных агентов, являющихся членами большой колонии) и используется для решения оптимизационной проблемы. Моделируемые агенты совместно решают проблему и помогают другим агентам в дальнейшей оптимизации решения.

Базовая идея метода муравьиных колоний состоит в решении оптимизационной задачи путем применения непрямой связи между автономными агентами [4, 5].

В методе муравьиных колоний предполагается, что окружающая среда представляет собой закрытую двумерную сеть – это группа узлов, соединенных посредством граней. Каждая грань имеет вес, кото-