Внутреннее представление программы

Основные свойства языка внутреннего представления программ:

1) внутреннее представление фиксирует синтаксическую структуру исходной программы;

2) генерация внутреннего представления происходит в процессе синтаксического анализа;

3) конструкции языка внутреннего представления должны относительно просто транслироваться в объектный код либо достаточно эффективно интерпретироваться.

Некоторые способы внутреннего представления программ:

(а) постфиксная запись

(б) префиксная запись

(в) многоадресный код с явно именуемыми результатами

(г) многоадресный код с неявно именуемыми результатами

(д) связные списочные структуры, представляющие синтаксическое дерево.

В основе каждого из этих способов лежит некоторый метод представления синтаксического дерева.

ПОЛИЗ – польская инверсная запись **(постфиксная запись)**

Пример. Обычной (инфиксной) записи выражения

a*(b+c)-(d-e)/f

соответствует такая постфиксная запись:

abc+*de-f/-

- порядок операндов остался таким же, как и в инфиксной записи,
- учтено старшинство операций,
- нет скобок.

Простым будем называть выражение, состоящее из одной константы или имени переменной.

Приоритет и ассоциативность операций в инфиксных выражениях позволяют четко установить границы операндов:

a — простое выражение ;

$a + b * c \sim a + (b * c)$;

$a - b + c - d \sim ((a - b) + c) - d$.

ПОЛИЗ выражений

(1) если E является простым выражением, то ПОЛИЗ выражения E — это само выражение E;

(2) ПОЛИЗом выражения $\boldsymbol{E_1}\ \theta\ \boldsymbol{E_2}$,
где θ — знак бинарной операции, E_1 и E_2 операнды для θ,
является запись $\boldsymbol{E_1'}\ \boldsymbol{E_2'}\ \theta$,
где E_1' и E_2' — ПОЛИЗ выражений E_1 и E_2 соответственно;

(3) ПОЛИЗом выражения $\theta\ \boldsymbol{E}$, где θ — знак унарной операции, а E — операнд θ,
является запись $\boldsymbol{E'}\ \theta$,
где E' — ПОЛИЗ выражения E;

(4) ПОЛИЗом выражения (E) является ПОЛИЗ выражения E.

Алгоритм интерпретации с помощью стека

ПОЛИЗ просматривается поэлементно слева направо. В стеке хранятся значения промежуточных вычислений и результат.

(1) если очередной элемент — операнд, то его значение заносится в стек;

(2) если очередной элемент — операция, то на "вершине" стека сейчас находятся ее операнды (это следует из определения ПОЛИЗа и предшествующих действий алгоритма); они извлекаются из стека, над ними выполняется операция, результат снова заносится в стек;

(3) когда выражение, записанное в ПОЛИЗе, прочитано, в стеке останется один элемент — это значение всего выражения.

Замечание: для интерпретации, кроме ПОЛИЗа выражения, необходима дополнительная информация об операндах, хранящаяся в таблицах.

Алгоритм Дейкстры перевода в ПОЛИЗ выражений

Будем считать, что ПОЛИЗ выражения будет формироваться в массиве, содержащем лексемы — элементы ПОЛИЗа, и при переводе в ПОЛИЗ будет использоваться вспомогательный стек, также содержащий элементы ПОЛИЗа — операции, имена функций и круглые скобки.

1. Выражение просматривается один раз слева направо.

2. Пока есть непрочитанные лексемы входного выражения, выполняем действия:

а) Читаем очередную лексему.

б) Если лексема является числом или переменной, добавляем ее в ПОЛИЗ-массив.

в) Если лексема является символом функции, помещаем ее в стек.

г) Если лексема является разделителем аргументов функции (например, запятая):

до тех пор, пока верхним элементом стека не станет открывающаяся скобка, выталкиваем элементы из стека в ПОЛИЗ-массив. Если открывающаяся скобка не встретилась, это означает, что в выражении либо неверно поставлен разделитель, либо несогласованы скобки.

д) Если лексема является операцией θ, тогда:

1) пока приоритет θ меньше либо равен приоритету операции, находящейся на вершине стека (для лево-ассоциативных операций), или приоритет θ строго меньше приоритета операции, находящейся на вершине стека (для право-ассоциативных операций) выталкиваем верхние элементы стека в ПОЛИЗ-массив;
2) помещаем операцию θ в стек.

е) Если лексема является открывающей скобкой, помещаем ее в стек.

ж) Если лексема является закрывающей скобкой, выталкиваем элементы из стека в ПОЛИЗ-массив до тех пор, пока на вершине стека не окажется открывающая скобка. При этом открывающая скобка удаляется из стека, но в ПОЛИЗ-массив не добавляется. Если после этого шага на вершине стека оказывается символ функции, выталкиваем его в ПОЛИЗ-массив. Если в процессе выталкивания открывающей скобки не нашлось и стек пуст, это означает, что в выражении не согласованы скобки.

3. Когда просмотр входного выражения завершен, выталкиваем все оставшиеся в стеке символы в ПОЛИЗ-массив. (В стеке должны были оставаться только символы операций; если это не так, значит в выражении не согласованы скобки.)

Представление операторов

Оператор присваивания

I := E

в ПОЛИЗе будет записан как

I E :=

где ":=" - это двухместная операция, а I и E - ее операнды; I означает, что операндом операции ":=" является адрес переменной I, а не ее значение.

Расширение набора операций ПОЛИЗА

Операция перехода (обозначается «!») в терминах ПОЛИЗа означает, что процесс интерпретации надо продолжить с того элемента ПОЛИЗа, который указан как операнд этой операции.

Чтобы можно было ссылаться на элементы ПОЛИЗа, будем считать, что все они перенумерованы, начиная с 1 (например, занесены в последовательные элементы одномерного массива).

Пусть ПОЛИЗ оператора, помеченного меткой L, начинается с номера p, тогда оператор перехода

goto L в ПОЛИЗе записывается так:

p !

где ! – операция выбора элемента ПОЛИЗа, номер которого равен p.

Операция условный переход "по лжи" с семантикой

if (!B) goto L

Это двухместная операция с операндами B и L. Обозначим ее !F, тогда в ПОЛИЗе переход «по лжи» записывается так:

B’ p !F

где p — номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L, B’ — ПОЛИЗ логического выражения B.

Семантика *условного оператора*

if E then S_1 **else** S_2

с использованием введенной операции может быть описана так:

if (! E) goto L_2; S_1; goto L_3; L_2: S_2; L_3: ...

Тогда ПОЛИЗ условного оператора будет таким (порядок операндов — прежний):

E’ p_2 !F S_1’ p_3 ! S_2’ ... ,

где p_i – номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 2,3, E’ – ПОЛИЗ логического выражения E.

Семантика оператора *цикла* **while E do** S может быть описана так:

L_0: if (! E) goto L_1; S; goto L_0; L_1: ... .

Тогда ПОЛИЗ оператора цикла while будет таким (порядок операндов – прежний!):

E’ p_1 !F S’ p_0 ! ... ,

где p_i - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 0,1, E’ – ПОЛИЗ логического выражения E.

Операторы ввода и вывода М-языка являются одноместными операциями.

Оператор ввода **read** (I) в ПОЛИЗе будет записан как **I read**

Оператор вывода **write** (E) в ПОЛИЗе будет записан как **E’ write**,

где E’ – ПОЛИЗ выражения E.

Синтаксически управляемый перевод

В основе синтаксически управляемого перевода лежит уже известная нам грамматика с действиями.

$G_{\texttt{expr}}$ – грамматика, описывающая простейшее арифметическое выражение:

$$E \to T \{+T\}$$
$$T \to F \{*F\}$$
$$F \to a \mid b \mid (E)$$

$G_{\texttt{expr\_polish}}$ – грамматика с действиями по переводу выражения в ПОЛИЗ :

$$E \to T \{+T \langle cout << '+'; \rangle\}$$
$$T \to F \{*F \langle cout << '*'; \rangle\}$$
$$F \to a \langle cout << 'a'; \rangle \mid b \langle cout << 'b'; \rangle \mid (E)$$

В процессе анализа методом рекурсивного спуска входной цепочки $a + b * c$ по грамматике $G_{\texttt{expr\_polish}}$ в выходной поток будет выведена цепочка

$a\ \ b\ \ c * +$

Определение: Пусть T_1 и T_2 — алфавиты. *Формальный перевод* τ — это подмножество множества всевозможных пар цепочек в алфавитах T_1 и T_2: $\tau \subseteq (T_1^* \times T_2^*)$.

Назовем *входным* языком перевода τ язык $L_{вх}(\tau) = \{\alpha \mid \exists\, \beta : (\alpha, \beta) \in \tau \}$.

Назовем *целевым* (или *выходным*) языком перевода τ язык $L_{ц}(\tau) = \{\beta \mid \exists\, \alpha : (\alpha, \beta) \in \tau \}$.

Перевод τ *неоднозначен*, если для некоторых $\alpha \in T_1^*$, β, $\gamma \in T_2^*$, $\beta \neq \gamma$ справедливы соотношения: $(\alpha, \beta) \in \tau$ и $(\alpha, \gamma) \in \tau$.

Рассмотренная выше грамматика $G_{\texttt{expr\_polish}}$ задает однозначный перевод: каждому выражению ставится в соответствие единственная польская запись. Неоднозначные переводы могут быть интересны при изучении моделей естественных языков; для трансляции языков программирования используются однозначные переводы.

Генератор внутреннего представления программы на М-языке

Каждый элемент в ПОЛИЗе - это лексема, т.е. пара вида (тип_лексемы, значение_лексемы). При генерации ПОЛИЗа будем использовать дополнительные типы лексем:

POLIZ_GO - “!” ;
POLIZ_FGO - “!F” ;
POLIZ_LABEL - для ссылок на номера элементов ПОЛИЗа;
POLIZ_ADDRESS - для обозначения операндов-адресов (например, в ПОЛИЗе оператора присваивания).

Будем считать, что генерируемая программа размещается в объекте
Poliz prog (1000); класса Poliz:

```
class Poliz{
  lex *p;
  int size;
  int free;
public:
  Poliz(int max_size){p=new Lex [size =
max_size]; free = 0;};
  ~Poliz(){delete []p;};
  void put_lex(Lex l){p[free]=l; free++;};
  void put_lex(Lex l, int place){p[place]=l;};
  void blank(){free++;};
  int get_free(){return free;};
  lex& operator[](int index){
      if (index > size) throw "POLIZ:out of
array";else
  if(index > free) throw "POLIZ:indefinite element
                                    of array";
      else return p[index];
  };
  void print(){
  for (int i=0; i < free; i++) cout << p[i]; };
};
```

Добавим действия по генерации в некоторые функции семантического анализа: check_not( ) и c heck_op( ).

```
void Parser::check_not () {
    if (st_lex.pop() != LEX_BOOL)
        throw "wrong type is in not";
    else {
        st_lex.push (LEX_BOOL);
        prog.put_lex (Lex (LEX_NOT));
    }
}
```

```
    void Parser::check_op () {
        type_of_lex t1, t2, op, t = LEX_INT, r =
LEX_BOOL;
        t2 = st_lex.pop();
        op = st_lex.pop();
        t1 = st_lex.pop();
        if(op==LEX_PLUS ||op==LEX_MINUS
||op==LEX_TIMES ||op==LEX_SLASH)
            r = LEX_INT;
        if (op == LEX_OR || op == LEX_AND)
            t = LEX_BOOL;
        if (t1 == t2  &&  t1 == t) st_lex.push(r);
        else throw "wrong types are in operation";
        prog.put_lex (Lex (op) );
    }
```

Грамматика, содержащая действия по контролю контекстных условий и переводу выражений модельного языка в ПОЛИЗ

$E \rightarrow E1 \mid E1$ [= | < | >] ⟨ *st_lex.push* (*c_type*) ⟩ *E1* ⟨ *check_op* ( ) ⟩

$E1 \rightarrow T$ { [+ | - | *or*] ⟨ *st_lex.push* (*c_type*) ⟩ *T* ⟨ *check_op* ( ) ⟩ }

$T \rightarrow F$ { [* | / | *and*] ⟨ *st_lex.push* (*c_type*) ⟩ *F* ⟨ *check_op* ( ) ⟩ }

$F \rightarrow I$ ⟨ *check_id* ( ); *prog.put_lex* (*curr_lex*); ⟩ |

N ⟨ *st_lex.push* (*LEX_INT*); *prog.put_lex* (*curr_lex*); ⟩ |

[*true* | *false*] ⟨ *st_lex.push* (*LEX_BOOL*); *prog.put_lex* (*curr_lex*); ⟩ |

not F ⟨ *check_not*( ); ⟩ | (*E*)

Пример реализации процедуры анализа и перевода для нетерминала F :


```
void Parser::F ()
{
  if ( c_type == LEX_ID )
  {
    check_id();
    prog.put_lex (Lex (LEX_ID, c_val));
    gl();
  }
  else if ( c_type == LEX_NUM )
  {
    st_lex.push ( LEX_INT );
    prog.put_lex ( curr_lex );
    gl();
  }
  else if ( c_type == LEX_TRUE )
  {
    st_lex.push ( LEX_BOOL );
    prog.put_lex (Lex (LEX_TRUE, 1) );
    gl();
  }
```

```
  else if ( c_type == LEX_FALSE)
  {
    st_lex.push ( LEX_BOOL );
    prog.put_lex (Lex (LEX_FALSE, 0) );
    gl();
  }
  else if (c_type == LEX_NOT)
  {
    gl();
    F();
    check_not();
  }
  else if ( c_type == LEX_LPAREN )
  {
    gl();
    E();
    if ( c_type == LEX_RPAREN)
      gl();
    else
      throw curr_lex;
  }
  else
    throw curr_lex;
}
```

Действия для оператора присваивания

$S \to I \langle$ *check_id* ( ); *prog.put_lex* (*Lex* (*POLIZ_ADDRESS, c_val*)); $\rangle$:= $E \langle$*eqtype* ( ); *prog.put_lex* (*Lex* (*LEX_ASSIGN*)); $\rangle$

Для условного

if (!E) goto l2; S1; goto l3; l2: S2; l3:…

$S \to$ ***if*** $E \langle$ *eqbool*( ); *pl2 = prog.get_free*( ); *prog.blank*( );
prog.put_lex (*Lex* (*POLIZ_FGO*)); $\rangle$
then *S1* $\langle$ *pl3 = prog.get_free*( ); *prog.blank*( );
prog.put_lex (*Lex* (*POLIZ_GO*));
prog.put_lex (*Lex* (*POLIZ_LABEL, prog.get_free*( )), *pl2*); $\rangle$
else *S2* $\langle$ *prog.put_lex* (*Lex* (*POLIZ_LABEL, prog.get_free*( )), *pl3*); $\rangle$

Оператор цикла **while E do** S описывается так:

L_0: if (!E) goto l_1; S; goto l_0; l_1: ... .

а грамматика с действиями по контролю контекстных условий и переводу оператора цикла в ПОЛИЗ будет такой:

S → ***while*** ⟨ *pl0 = prog.get_free* ( )*;* ⟩ *E* ⟨ *eqbool* ( )*;*
pl1 = prog.get_free ( )*; prog.blank* ( )*;*
prog.put_lex (*Lex* (*POLIZ_FGO*))*;* ⟩
do *S* ⟨*prog.put_lex* (*Lex* (*POLIZ_LABEL, pl0*)*;*
prog.put_lex (*Lex* (*POLIZ_GO*))*;*
prog.put_lex (*Lex* (*POLIZ_LABEL, prog.get_free*( ))*, pl1*)*;* ⟩

Замечание: переменные pli (i=0,1,2,3) должны быть локализованы в процедуре S, иначе возникнет ошибка при обработке вложенных условных операторов.

Грамматика с действиями по контролю контекстных условий и переводу в ПОЛИЗ операторов ввода и вывода:

$S \rightarrow$ *read* (*I* ⟨ *check_id_in_read*( );
prog.put_lex (*Lex* (*POLIZ_ADDRESS, c_val*)); ⟩)
⟨ *prog.put_lex* (*Lex* (*LEX_READ*)); ⟩

$S \rightarrow$ *write* (*E*) ⟨ *prog.put_lex* (*Lex* (*LEX_WRITE*)); ⟩

Интерпретатор ПОЛИЗа для модельного языка

```
class Executer {
    Lex pc_el;
    public:
        void execute (Poliz& prog);
};

void Executer::execute ( Poliz& prog ) {
    Stack < int, 100 > args;
    int i, j, index = 0, size = prog.get_free();
    while ( index < size ) {
        pc_el = prog [ index ];
        switch ( pc_el.get_type () ) {
            case LEX_TRUE: case LEX_FALSE: case LEX_NUM:
            case POLIZ_ADDRESS: case POLIZ_LABEL:
                args.push ( pc_el.get_value () ); break;
            case LEX_ID:
                i = pc_el.get_value ();
                if ( TID[i].get_assign () ){
                     args.push ( TID[i].get_value () );
                                                 break;}
                else throw "POLIZ: indefinite identifier";
```

```
case LEX_NOT:
    args.push( !args.pop() ); break;
case LEX_OR:
    i = args.pop();
    args.push ( args.pop() || i ); break;
case LEX_AND:
    i = args.pop();
    args.push ( args.pop() && i ); break;
case POLIZ_GO:
    index = args.pop() - 1; break;
case POLIZ_FGO:
    i = args.pop();
    if ( !args.pop() ) index = i-1; break;
case LEX_WRITE:
    cout << args.pop () << endl; break;
```

```
case LEX_READ:
    {int k;
    i = args.pop ();
    if ( TID[i].get_type () == LEX_INT ){
        cout << "Input int value for";
        cout << TID[i].get_name () << endl;
        cin >> k;
    }
    else {
        char j[20];
        rep:
        cout << "Input boolean value;
        cout << (true or false) for";
        cout << TID[i].get_name() << endl;
        cin >> j;
        if (!strcmp(j, "true")) k = 1;
        else
        if (!strcmp(j, "false")) k = 0;
        else {
            cout<< "Error in input:true/false";
            cout << endl;
```

```
                    goto rep;}
                }
                TID[i].put_value (k);
                TID[i].put_assign ();
                break;}
            case LEX_PLUS:
                args.push ( args.pop() + args.pop() ); break;
            case LEX_TIMES:
                args.push ( args.pop() * args.pop() ); break;
            case LEX_MINUS:
                i = args.pop();
                args.push ( args.pop() - i ); break;
            case LEX_SLASH:
                i = args.pop();
                if (!i) { args.push(args.pop() / i); break;}
                else throw "POLIZ:divide by zero";
            case LEX_EQ:
                args.push ( args.pop() == args.pop() );
break;
            case LEX_LSS:
                i = args.pop();
                args.push ( args.pop() < i); break;
            case LEX_GTR:
```

```
                    i = args.pop();
                    args.push ( args.pop() > i ); break;
                case LEX_LEQ:
                    i = args.pop();
                    args.push ( args.pop() <= i ); break;
                case LEX_GEQ:
                    i = args.pop();
                    args.push ( args.pop() >= i ); break;
                case LEX_NEQ:
                    i = args.pop();
                    args.push ( args.pop() != i ); break;

                case LEX_ASSIGN:
                    i = args.pop();
                    j = args.pop();
                    TID[j].put_value(i);
                    TID[j].put_assign(); break;
                default: throw "POLIZ: unexpected elem";
            }//end of switch
            index++;
            };//end of while
        cout << "Finish of executing!!!" << endl;
}
```

```
class Interpretator {
        Parser pars;
        Executer E;
    public:
        Interpretator (char* program): pars (program){};
        void interpretation ();
};


void Interpretator::interpretation (){
        pars.analyze ();
        E.execute ( pars.prog );
}
```

```
int main () {
    try {
            Interpretator I ("program.txt");
            I.interpretation ();
            return 0;
    }
    catch (char c) {
        cout << "unexpected symbol " << c << endl; return 1;
    }
    catch (Lex l) {
        cout << "unexpected lexeme"; cout << l; return 1;
    }
    catch(const char *source) {
        cout << source << endl; return 1;
    }
}
```