Язык внутреннего представления программ

Основные свойства языка внутреннего представления программ:

- он позволяет фиксировать синтаксическую структуру исходной программы;
- текст на нем можно автоматически генерировать во время синтаксического анализа;
- его конструкции должны относительно просто транслироваться в объектный код либо достаточно эффективно интерпретироваться.

Некоторые общепринятые способы внутреннего представления программ:

- постфиксная запись,
- префиксная запись,
- многоадресный код с явно именуемыми результатами,
- многоадресный код с неявно именуемыми результатами,
- связные списочные структуры, представляющие синтаксическое дерево.

ПОЛИЗ

ПОЛИЗ идеален для внутреннего представления интерпретируемых ЯП, которые, как правило, удобно переводятся в ПОЛИЗ и легко интерпретируются.

В ПОЛИЗе операнды выписаны слева направо в порядке их следования в исходном тексте.

Знаки операций стоят таким образом, что знаку операции непосредственно предшествуют ее операнды.

Более формально постфиксную запись выражений можно определить таким образом:

1) если E является единственным операндом, то ПОЛИЗ выражения E — это сам этот операнд;
2) ПОЛИЗом выражения **E1 θ E2**, где θ — знак бинарной операции, E1 и E2 операнды для θ, является запись **E1**' **E2**' **θ** , где E1' и E2' — ПОЛИЗ выражений E1 и E2 соответственно;
3) ПОЛИЗом выражения **θ E**, где θ — знак унарной операции, а E — операнд θ, запись **E**' **θ**, где E' — ПОЛИЗ выражения E;
4) ПОЛИЗом выражения (E) является ПОЛИЗ выражения E.

Что попадает в ПОЛИЗ при работе с модельным языком

Операции

- Все операции ЯП
  (у нас кроме тернарной ?:)
- ! – одноместная, безусловный
  переход на полиз-метку:
  p, !, …
- !F – двуместная, переход по false
  на полиз-метку: **E**$_{полиз}$**, p, !F, …**
- ; – 0-местная, удаление
  элемента из стека

Операнды

Константы
Переменные
Полиз-метки (p)
Адреса полиза (&x, например)

Например,

x = y => &x, y, =, ; , если у нас оператор-выражение
x++ => &x, #+, ; , +# − для префиксных ++, −−
−5 * 2 => 5, @, 2, *, \\ 5,2,*,@,…
a<b ? a : b => a, b, < , 9 , !F, a, 10, !, b, ….
1 2 3 4 5 6 7 8 9 10

Алгоритм вычисления выражений, записанных в ПОЛИЗе

- Выражение просматривается один раз слева направо, при этом:
  - Если очередной элемент ПОЛИЗа — это операнд, то его значение заносится в стек;
  - Если очередной элемент ПОЛИЗа — это операция, то на «верхушке» стека находятся ее операнды, они извлекаются из стека, и над ними выполняется операция, результат выполнения снова заносится в стек;

- Когда выражение, записанное в ПОЛИЗе, прочитано, в стеке останется один элемент — это значение всего выражения.

Для интерпретации, кроме ПОЛИЗа выражения, необходима дополнительная информация об операндах, хранящаяся в таблицах.

Неоднозначно интерпретируемые операции в ПОЛИЗе

Может оказаться так, что знак бинарной операции по написанию совпадает со знаком унарной; например, знак «−» в большинстве языков программирования означает и бинарную операцию вычитания, и унарную операцию изменения знака.

В этом случае во время интерпретации операции «−» возникнет неоднозначность: сколько операндов надо извлекать из стека и какую операцию выполнять. Устранить неоднозначность можно, по крайней мере, двумя способами:

- заменить унарную операцию бинарной, т. е. считать, что «−a» означает «0 − a»;
- ввести специальный знак для обозначения унарной операции; например, «−a» заменить на «@a». Такое изменение касается только внутреннего представления программы и не требует изменения входного языка.

Аналогично разрешаются неоднозначности операций ++ и −−.

ПОЛИЗ для операторов

Оператор присваивания I := E

в ПОЛИЗе будет записан как **&I , E, :=** , ; ,...

где «:=» — двухместная операция, а &I и E — ее операнды;

&I означает, что операндом операции «:=» является **адрес** переменной I, а не ее значение.

Если в ЯП присваивание является операцией, а используется как оператор (не в выражении, а наряду с другими операторами), то добавляем операцию ПОЛИЗа «;», извлекающую результат выполнения операции из стека.

Оператор перехода в терминах ПОЛИЗа означает, что процесс интерпретации надо продолжить с того элемента ПОЛИЗа, который указан как операнд операции перехода.

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

Пусть ПОЛИЗ оператора, помеченного меткой L, начинается с номера p, тогда оператор перехода **goto** L в ПОЛИЗе можно записать как

p , !, ...

где «!» — операция выбора элемента ПОЛИЗа, номер которого равен p.

Семантика **условного оператора** **if (E)** S1 **else** S2

с использованием операций ПОЛИЗа может быть описана так:

if (! E) **goto** L2; S1; **goto** L3; L2: S2; L3: ...

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

где pi — 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 :

L0: **if** (! E) **goto** L1; S; **goto** L0; L1: ... .

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

$E_{полиз}$, p1, !F, $S_{полиз}$, p0, !, ...

где pi - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 0, 1.

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

&I, read ;

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

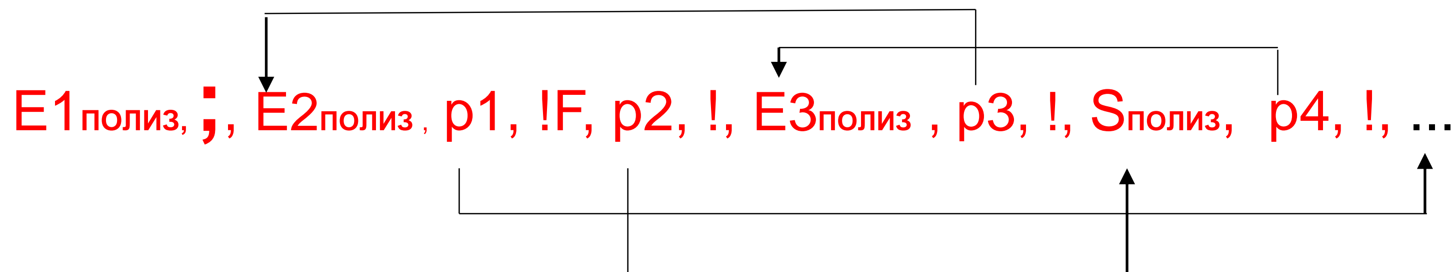
$E_{полиз}$, write ,

Рассмотрим **оператора цикла** **FOR (E1; E2; E3) S** :

Его семантика:

E1; L3: **if** (! E2) **goto** L1; **goto** L2; L4: E3; **goto** L3; L2: S; **goto** L4; L1: ... .

Тогда ПОЛИЗ оператора цикла **for** будет таким (порядок операндов — прежний):



где pi — номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 1, 2, 3, 4.

ПОЛИЗ для ленивых вычислений выражений с операцией «&&»:

E1 && E2 — его семантика:

if (!E1) **goto** L1; **if** (!E2) **goto** L1; 1; **goto** L2; L1: 0; L2: ... .

Тогда ПОЛИЗ такого фрагмента выражения будет следующим (порядок операндов — прежний):

E1полиз, p1, !F, E2полиз , p1, !F, 1, p2, !, 0, ...

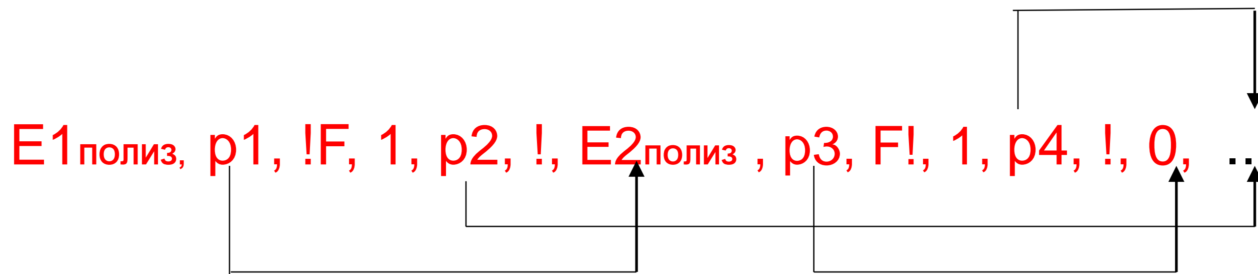
где pi — номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 1, 2

ПОЛИЗ для ленивых вычислений выражений с операцией «||»:

E1 || E2 — его семантика:

if(!E1) goto L1;1; goto L2; L1: if(!E2) goto L3; L2: 1; goto L4; L3: 0; L4: ...

Тогда ПОЛИЗ такого фрагмента выражения будет следующим (порядок операндов — прежний):



где pi - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L_i, i = 1, 2, 3, 4.

Синтаксически управляемый перевод

Синтаксический, семантический анализ и генерация внутреннего представления программы часто осуществляются одновременно.

Один из способов построения промежуточной программы — синтаксически управляемый перевод.

В основе синтаксически управляемого перевода лежит **грамматика с действиями**, которые параллельно с анализом исходной цепочки лексем позволяют генерировать внутреннее представление программы.

Пример:

Пусть есть грамматика, описывающая простейшее арифметическое выражение.

G_{expr}:

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

Тогда грамматика с действиями по переводу этого выражения в ПОЛИЗ будет такой:

Gexpr_polish:

$$E \rightarrow T \{ + T < cout << \text{'+'}; > \}$$
$$T \rightarrow F \{ * F < cout << \text{'*'}; > \}$$
$$F \rightarrow a < cout << \text{'a'}; > \mid b < cout << \text{'b'}; > \mid (E)$$

Синтаксически управляемый перевод

Если необходимо переводить в ПОЛИЗ в процессе синтаксического анализа методом рекурсивного спуска выражение, содержащее правоассоциативные операции, то для таких операций соответствующие правила вывода следует писать, например, таким образом:

G: A → I = A | E

...

А грамматика с действиями по переводу выражения в ПОЛИЗ для этих правил вывода будет такой:

G: A → I < cout << "&I" ; > = A < cout << '=' ; > | E

...

Определение формального перевода

Пусть $T1$ и $T2$ — алфавиты.

***Формальный** перевод* τ — это подмножество множества всевозможных пар цепочек в алфавитах $T1$ и $T2$: $\tau \subseteq (T1^* \times T2^*)$.

Входной язык перевода τ - язык L*вх*$(\tau) = \{ \alpha \mid \exists \beta: (\alpha, \beta) \in \tau \}$.

Целевой (или *выходной*) ***язык*** перевода τ - язык L*ц*$(\tau) = \{\beta \mid \exists \alpha: (\alpha, \beta) \in \tau\}$.

Перевод τ *неоднозначен*, если для некоторых $\alpha \in T1^*$, $\beta, \gamma \in T2^*$, $\beta \neq \gamma$, $(\alpha, \beta) \in \tau$ и $(\alpha, \gamma) \in \tau$.

Чтобы задать перевод из $L1$ в $L2$, важно точно указать *закон соответствия* между цепочками $L1$ и $L2$.

Пример. Пусть $L1 = \{ 0^n 1^m \mid n \geq 0, m > 0\}$ — входной язык,

$L2 = \{ a^m b^n \mid n \geq 0, m > 0\}$ — выходной язык, и

перевод τ определяется так: для любых $n \geq 0$, $m > 0$

цепочке $0^n 1^m \in L1$ соответствует цепочка $a^m b^n \in L2$.

Можно записать τ с помощью теоретико-множественной формулы:

$\tau = \{ (0^n 1^m, a^m b^n) \mid n \geq 0, m > 0 \}$, L*вх* $(\tau) = L1$, L*ц* $(\tau) = L2$.

Пример.

$L1 = \{ \omega\perp \mid \omega \subset \{ a, b \}^{+}, \sum a = n, \sum b = m \}$

$L2 = \{ a^{[n/2]}\ b^{[m/2]} \mid n >= 0, m >= 0\}$

G(L1): $S \rightarrow aA\perp \mid bA\perp$

$A \rightarrow aA \mid bA \mid \varepsilon$

Грамматика с действиями по переводу L1 в L2:

S → a < n = 1; m = 0; > A⊥ | b < n = 0; m = 1; > A ⊥

A → a < if (n) { cout << 'a'; n = o; } else n = 1; > A |

bA < if (m) { cout << 'b'; m = o; } else m = 1; } > | ε

Генератор внутреннего представления программы на М-языке

Каждый элемент в ПОЛИЗе — это лексема вида
(тип_лексемы, значение_лексемы).

При генерации ПОЛИЗа используются дополнительные типы лексем:

POLIZ_GO — "!" − ;
POLIZ_FGO — "!F" ;
POLIZ_LABEL — для ссылок на номера элементов ПОЛИЗа;
POLIZ_ADDRESS — для обозначения операндов-адресов

Генерируемая программа размещается в объекте

vector <Lex> poliz;

Генерация внутреннего представления программы проходит во время синтаксического анализа параллельно с контролем КУ.

Для генерации ПОЛИЗа используется информация, «собранная» синтаксическим и семантическим анализаторами, а производится она с помощью действий, вставленных в функции семантического анализа.

Грамматика с действиями по контролю КУ и переводу в ПОЛИЗ выражений и операторов присваивания, ввода и вывода М-языка

```
E → E1 | E1 [ = | < | > ] < st_lex.push (c_type) > E1 < check_op () >
E1 → T { [ + | - | or ] < st_lex.push (c_type) > T < check_op () > }
T → F { [ * | / | and ] < st_lex.push (c_type) > F < check_op () > }
F → I < check_id (); poliz.push_back ( curr_lex ); > |
        N < st_lex.push (LEX_INT); poliz.push_back( curr_lex ); > |
  [ true | false ] < st_lex.push (LEX_BOOL); poliz.push_back (curr_lex ); > |
                  not F < check_not(); > | (E)

S →  I <check_id (); poliz.push_back (Lex (POLIZ_ADDRESS, c_val)); >  :=
        E < eqtype (); poliz.push_back (Lex (LEX_ASSIGN)); >

S →  read ( I <check_id_in_read();
                      poliz.push_back (Lex(POLIZ_ADDRESS, c_val));>)
        < poliz.push_back (Lex (LEX_READ)); >

S →  write ( E ) < poliz.push_back (Lex (LEX_WRITE)); >
```

Грамматика с действиями по контролю КУ и переводу в ПОЛИЗ условного оператора и оператора цикла

if E then S1 else S2

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

```
S → if  E  < eqbool();  pl2 = poliz.size(); poliz.push_back (Lex());
                  poliz.push_back (Lex (POLIZ_FGO)); >
      then S1 < pl3 = poliz.size(); poliz.push_back (Lex());
                  poliz.push_back (Lex (POLIZ_GO));
                  poliz[pl2] = Lex (POLIZ_LABEL, poliz.size()); >
      else S2 < poiz[pl3] = Lex (POLIZ_LABEL, poliz.size()); >
```

while E do S

l0: if (!E) goto l1; S; goto l0; l1: ... .

```
S →  while < pl0 = poliz.size(); > E  <  eqbool ();
                              pl1 = poliz.size(); poliz.push_back (Lex());
                              poliz.push_back (Lex (POLIZ_FGO)); >
       do    S  < poliz.push_back (Lex (POLIZ_LABEL, pl0);
                  poliz.push_back (Lex (POLIZ_GO));
                  poliz[pl1] = Lex (POLIZ_LABEL, poliz.size()); >
```

Интерпретатор ПОЛИЗа для М-языка

Польская инверсная запись выбрана в качестве языка внутреннего представления программы, в частности, потому, что записанная на нем программа может быть легко проинтерпретирована.

Идея алгоритма очень проста: просматриваем ПОЛИЗ слева направо; если встречаем операнд, то записываем его в стек; если встретили знак операции, то извлекаем из стека нужное количество операндов и выполняем операцию, результат (если он есть) заносим в стек.

Программа на ПОЛИЗе хранится в виде последовательности лексем в векторе лексем ***poliz***.

Лексемы могут быть следующие:

- лексемы-константы (числа, true, false),
- лексемы-метки ПОЛИЗа,
- лексемы-операции (включая введенные в ПОЛИЗе) и
- лексемы-переменные (их значения — номера строк в таблице TID).

```
class Executer {
public:
        void execute (vector <Lex> & poliz);
};
```

```
void Executer::execute (vector <Lex> & poliz ) {
    Lex  pc_el;
    stack < int > args;
    int  i,  j,  index = 0, size = poliz.size ( );

    while ( index < size ) {
        pc_el = poliz [ 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:
      from_st (args, i);
      args.push (!i);  break;
case LEX_OR:
      from_st (args, i);
      from_st (args, j);
      args.push (i || j);  break;
case LEX_AND:
      from_st (args, i);
      from_st (args, j);
      args.push (i && j);  break;
case POLIZ_GO:
      from_st (args, i);  index = i - 1;
      break;
case POLIZ_FGO:
      from_st (args, i);
      from_st (args, j);
      if ( ! j ) index = i-1;  break;
case LEX_WRITE:
      from_st (args, i);
      cout << j << endl;
      break;
```

```
case LEX_READ: {
        from_st (args, i);
        if ( TID [ i ].get_type () == LEX_INT ) {
              cout << "Input int value for";
              cout << TID[i].get_name () << endl;
              cin >> k;
        }
        else {  string  j;   int k;
                   while(1) {
                        cout << "Input boolean value (true or false) for";
                        cout << TID [ i ].get_name ( ) << endl;
                        cin >> j;
                        if (j != "true“  && j != “false”) {
                            cout << "Error in input: true/false" << endl;
                            continue;
                        }
                        k = (j == “true”) ? 1 : 0;
                        break;
                  }
                TID [ i ].put_value (k);
                TID [ i ].put_assign ();
                 break; }
```

```
case LEX_PLUS:
      from_st (args, i);
      from_st (args, j);
      args.push (i + j);  break;
case LEX_TIMES:
      from_st (args, i);
      from_st (args, j);
      args.push (i * j);  break;
case LEX_MINUS:
      from_st (args, i);
      from_st (args, j);
      args.push (j - i);  break;
case LEX_SLASH:
      from_st (args, i);
      from_st (args, j);
      if ( !i ) { args.push ( j / i);  break; }
      else   throw "POLIZ:divide by zero";
case LEX_EQ:
      from_st (args, i);
      from_st (args, j);
      args.push (i == j);  break;
```

```
        ....
      case LEX_GEQ:
            from_st (args, i);
            from_st (args, j);
             args.push (j >= i);  break;
      case LEX_NEQ:
            from_st (args, i);
            from_st (args, j);
             args.push (i != j);  break;
      case LEX_ASSIGN:
            from_st (args, i);
            from_st (args, j);
            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 (const char * program) : pars(program) { };
     void interpretation ( );
};

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

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