

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,
- ❖ связанные 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 $E_1 \theta E_2$, где θ - знак бинарной операции, E_1 и E_2 операнды для θ , является запись $E_1' E_2' \theta$, где E_1' и E_2' – ПОЛИЗ выражений E_1 и E_2 соответственно;
- 3) ПОЛИЗом выражения θE , где θ - знак унарной операции, а E - операнд θ , является запись $E' \theta$, где E' - ПОЛИЗ выражения E ;
- 4) ПОЛИЗом выражения (E) является ПОЛИЗ выражения E .

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

$\&I, E, :=, \dots$

где "!=" - двухместная операция, а $\&I$ и E - ее операнды;

$\&I$ означает, что операндом операции "!=" является **адрес** переменной I , а не ее значение.

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

$p, !, \dots$

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

Введем вспомогательную операцию - условный переход "по лжи" с семантикой

if (!B) goto L

Это двухместная операция в операндами B и L. Обозначим ее !F, тогда в ПОЛИЗе она будет записана как

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 S1 else S2

с использованием введенной операции может быть описана так:

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

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

E, p2, !F, S1, p3, !, S2, ...

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** :

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

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

E, p1, !F, S, p0, !, ...

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 - одноместные операции.

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

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.

$$\begin{aligned} G_{expr}: \quad & E \rightarrow T \{ + T \} \\ & T \rightarrow F \{ * F \} \\ & F \rightarrow a \mid b \mid (E) \end{aligned}$$

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

$$\begin{aligned} G_{expr_polish}: \quad & E \rightarrow T \{ + T < cout << '+'; > \} \\ & T \rightarrow F \{ * F < cout << '*'; > \} \\ & F \rightarrow a < cout << 'a'; > \mid b < cout << 'b'; > \mid (E) \end{aligned}$$

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:

$G: \quad A \rightarrow I = A \mid E$

...

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

$G: \quad A \rightarrow I < \text{cout} << \text{"\&I"} ; > = A < \text{cout} << \text{'='} ; > \mid E$

...

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 $L_{\text{вх}}(\tau) = \{ \alpha \mid \exists \beta: (\alpha, \beta) \in \tau \}$.

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

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:

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

Пример.

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

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

$$\begin{aligned} G(L1): \quad S &\rightarrow aA \perp \mid bA \perp \\ A &\rightarrow aA \mid bA \mid \varepsilon \end{aligned}$$

Грамматика с действиями по переводу L1 в L2:

$$\begin{aligned} S &\rightarrow a \langle n = 1; m = 0; \rangle A \perp \mid b \langle n = 0; m = 1; \rangle A \perp \\ A &\rightarrow a \langle \text{if } (n) \{ \text{cout} \ll 'a'; n = 0; \} \text{ else } n = 1; \rangle A \mid \\ &\quad bA \langle \text{if } (m) \{ \text{cout} \ll 'b'; m = 0; \} \text{ else } m = 1; \rangle \mid \varepsilon \end{aligned}$$

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

Poliz prog (1000); класса **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.

Класс 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 ];
    }
};
```

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

$E \rightarrow E1 \mid E1 \left[= \mid < \mid > \right] \left\langle st_lex.push (TD [c_val]) \right\rangle E1 \left\langle check_op () \right\rangle$

$E1 \rightarrow T \left\{ \left[+ \mid - \mid or \right] \left\langle st_lex.push (TD [c_val]) \right\rangle T \left\langle check_op () \right\rangle \right\}$

$T \rightarrow F \left\{ \left[* \mid / \mid and \right] \left\langle st_lex.push (TD [c_val]) \right\rangle F \left\langle check_op () \right\rangle \right\}$

$F \rightarrow I \left\langle check_id () \right\rangle ; prog.put_lex (curr_lex); \right\rangle \mid$

$N \left\langle st_lex.push (LEX_INT); prog.put_lex (curr_lex); \right\rangle \mid$

$\left[true \mid false \right] \left\langle st_lex.push (LEX_BOOL); prog.put_lex (curr_lex); \right\rangle \mid$

$not F \left\langle check_not(); \right\rangle \mid (E)$

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

$S \rightarrow$

$read \left(I \left\langle check_id_in_read(); prog.put_lex (Lex (POLIZ_ADDRESS, c_val)); \right\rangle ; \right)$
 $\left\langle prog.put_lex (Lex (LEX_READ)); \right\rangle$

$S \rightarrow write (E) \left\langle prog.put_lex (Lex (LEX_WRITE)); \right\rangle$

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 = prog.get_free(); prog.blank();
 prog.put_lex (Lex (POLIZ_FGO)); >
then S1 < pl3 = prog.get_free(); prog.blank();
 prog.put_lex (Lex (POLIZ_GO));
 prog.put_lex (Lex (POLIZ_LABEL, prog.get_free()), pl2); >
else S2 < prog.put_lex (Lex (POLIZ_LABEL, prog.get_free()), pl3); >

while E do S

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

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); >

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 объекте класса Poliz - prog.

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 {
    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;  
    }  
}
```