МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Курсовая работа

по дисциплине «Системы программирования»
на тему: «Транслирующая грамматика, определяющая перевод логических выражений из инфиксной записи в постфиксную»

Выполнил:
Студент группы М8О-201Б-21
Голов Александр Юрьевич

Принял:
Доцент кафедры 806
Киндинова Виктория Валерьевна

Оценка:_____ **Дата:**_____

Москва, 2023

Содержание

1 Введение

Современный мир наполнен, если не переполнен, производными информационных технологий. В некотором смысле можно сказать, что современный мир полостью зависим от информационных технологий и, как следствие, программирования. А от чего же зависит само программирование? Ответ лежит на поверхности: программирование - процесс, зависимый от того или иного средства разработки, которые, в свою очередь, создавались при помощи других средств разработки. Продолжив углубляться в тему зависимостей программирования, мы с вами придём к единицам и нулям - машинным кодам, послужившим средством разработки в семидесятые годы прошлого века посредством перфокарт. Однако, современному «разработчику» такое программирование покажется пыткой, как и средство разработки «уровнем выше» в модели абстракций программирования - «assembly» - неминуемый этап в создании средств разработки, используемый повсеместно прямо или косвенно.

Программирование на «assembly» на сегодняшний день весьма неактуально ввиду трудоёмкости написания программ - всё же мы обращаемся напрямую к процессору. Проблеме трудоёмкости программирования человечество нашло решение ещё в прошлом веке, задавашисб вопросом: Что если «шагнуть» от процессора и создать более «дружелюбный» для разработчиков язык? Таким образом повились первые компиляторы, а вместе с ними и высокоуровненвые языки программирования, в том числе и мощнейший и актульный по сей день C.

2 Определение инструментов разработки

Чтобы перейти к выполнению поставленной задачи - разработке транслирующей грамматики, переводящей логическое выраджение из инфиксной записи в постфиксную, определим инструменты, которыми мы будем пользоваться.

2.1 Обратная польская запись

Инфиксная форма - самая распространённая форма, так как для человека она проще для представления. Она представляет из себя выражение, в котором

операторы располагаются между операндами. Отсюда исходит название данной формы. Пример инфиксной формы:

$$a \cup b \cap c$$

Префиксная же форма представляет из себя выражение, в котором операторы находятся перед операндами:

$$\cap c \cup ab$$

Соответственно, постфиксная форма представляет из себя выражение, в котором оператор находится после операндов:

$$ab \cup c \cap$$

2.2 Контекстно-свободная грамматика

Поскольку транслирующая грамматика - подмножество множества контекстно-свободных грамматик, определим для начала КС-грамматику. В классификации Хомского КС-грамматика - грамматика, распознаваемая распознавателем - МП-автоматом (автоматом с магазинно-стековой памятью). Более строго грамматика $G = (T, V, P, S_0)$ называется контекстно-свободной, если каждое правило $p \in P$ имеет вид $A \to \alpha : A \subset V, \alpha \subset (T \cup V)^*$.

2.3 Транслирующая грамматика

Дадим сразу формальное определение. Пятёрка объектов $G^T = (V, \Sigma_i, \Sigma_a, P, S)$, характеризующая КС-грамматику, где Σ_i - множество входных символов, Σ_a - множество операционных символов, V - множество нетерминальных символов, $S \in V$ - начальный символ грамматики, P - конечное множество правил вывода вида $A \to \alpha : A \in V, \alpha \in (\Sigma_i \cup \Sigma_a \cup V)^*$, называется транслирующей грамматикой, если её множество терминалов разбито на множества входных и операционных символов - Σ_i и Σ_a соответственно.

3 Первичный анализ

Итак. задача поставлена: на вход подаётся логическое выражение в инфиксной записи, состоящее из трёх логических операций: отрицание, объединения и пересечения, на выходе требуется получить эквивалентное выражение в польской нотации.

Определим приоритеты операций:
Таблица 1 - Приоритеты операций

Приоритет	Операция
1	$\neg$
2	$\cap$
3	$\cup$

Также обратим внимание на то, что мы имеем дело с двумя двуместными операторами «и», «или» - $\cap$, $\cup$ и одним одноместым - «не» - $\neg$. Таким образом задача сводится к проектирвоанию транслирующей грамматики, обрабатывающей всего три операнда.

4 Описание алгоритма

Правило $p_i \in P$ транслирующей грамматики имеет вид:

$$A \to \alpha, : \{\beta\} : \ A \in V, \ \beta \in \Sigma_a \, \alpha \in (\Sigma_i \cup \Sigma_a \cup V)^*.$$

Проще говоря, в фигурных скобках указывается операционный символ, подлежащий вствке в выходную цепочку. Принцип работы транслирующей грамматики схож с принципом работы синтаксически управляемой схемы, более того, это два различных способа описать по сути один и тот же процесс.

Опишем алгоритмы действий для всех возоможных случаев.

4.1 Отрицание

Как уже говорилось, отрицание - олноместная операция, следовательно для приведения логического выражения с отрицанием из инфиксной записи в «ПОЛИЗ» достаточно поменять местами оператор и опреранд.

$$\neg a \longrightarrow a \neg.$$

4.2 Конъюнкция

Конънкция же двуместная операция, алгоритм приведения выражения к польской нотации иной: поменять местами правый операнд и оператора.

$$a \cap b \longrightarrow ab \cap .$$

4.3 Дизъюнкция

Алгоритм преобразования двуместных операторов, коим, как уже говорилось, и является дизъюнкция, приведён в примере для конъюнкции.

$$a \cup b \longrightarrow ab \cup .$$

5 Построение транслирующей грамматики

Для наглядности и простоты понимания ограничим множество булевых переменных одним элементом - i, тем не менее, забегая вперёд, важно отметить, что правила для алфавита с одним элементом вида

- $B \to B \cap A$,
- $A \to i,\ A \in V,\ i \in \Sigma_i \cup \Sigma_a$

для алфавита мощности n преобразуются в правила вида

- $B_1 \to B_1 \cap A_1$,
- $A_1 \to \alpha_1,\ A_1 \in V,\ \alpha_1 \in \Sigma_i \cup \Sigma_a$

$$\vdots$$

- $B_n \to B_n \cap A_n$,
- $A_n \to \alpha_n,\ A_n \in V,\ \alpha_n \in \Sigma_i \cup \Sigma_a$

Рассмотрим возможные случаи, сразу продумывая правила транслирующей грамматики.

5.1 Одноместная операция

Если входной символ является оператором отрицания - $\neg$, это значит, что уже был прочитан его операнд и в выходную последовательность необходимо записать сам операнд и оператор.

5.2 Двуместные операции

Если входной символ является оператор конъюнкции - $\cap$ или оператор дизъюнкции - $\cup$, это значит, что уже был прочитан первый операнд и нам следует его записать в выходную последовательность. Следующим действием будет прочтение второго операнда и запись как второго операнда, так и оператора в выходную последовательность.

5.3 Определение транслирующей грамматики

Начнём с очевидного - определения алфавита грамматики.

$$
\begin{gathered}
G^T = (V, \Sigma_i, \Sigma_a, P, S), \\
V = \{A\}, \\
\Sigma_i = \{i,\ \neg,\ \cap,\ \cup\}, \\
\Sigma_a = \{i,\ \neg,\ \cap,\ \cup\}, \\
S = \{S_0\}, \\
P = \{ \\
p_1:\ S_0 \to S_0 \cup A\,\{\cup\}, \\
p_2:\ S_0 \to S_0 \cap A\,\{\cap\}, \\
p_3:\ S_0 \to \neg A S_0\,\{\neg\}, \\
p_4:\ S_0 \to A, \\
p_5:\ A \to i\{i\}, \\
\}
\end{gathered}
$$

Выполним перевод цепочки $\neg i \cup i \cap i$:

- $2.\cup$
- $4.i\cup$

- $4.ii\cup$
- $1.\cap ii\cup$
- $4.i\cap ii\cup$
- $3.\neg i\cap ii\cup$
- $4.i\neg i\cap ii\cup$

6 Написание программы

6.1 Алгоритм Дейкстры

Для преобразования в постфиксную форму будем использовать улучшенный Эдсгером Вибе Дейкстрой алгоритм. Принцип работы алгоритма Дейкстра:

- Проходим исходную строку;
- При нахождении числа, заносим его в выходную строку;
- При нахождении оператора, заносим его в стек;
- Выталкиваем в выходную строку из стека все операторы, имеющие приоритет выше рассматриваемого;
- При нахождении открывающейся скобки, заносим её в стек;
- При нахождении закрывающей скобки, выталкиваем из стека все операторы до открывающейся скобки, а открывающуюся скобку удаляем из стека.

6.2 Листинг программы

```
/*
 * C# Program to Convert Infix to Postfix
 */
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
namespace Infix
{
    class Program
    {
        static bool convert(ref string infix, out string postfix)
        {

            int prio = 0;
            postfix = "";
            Stack<Char> s1 = new Stack<char>();
            for (int i = 0; i < infix.Length; i++)
            {
                char ch = infix[i];
                if (ch == '+' || ch == '-' || ch == '*' || ch == '/')
                {
                    if (s1.Count <= 0)
                        s1.Push(ch);
                    else
                    {
                        if (s1.Peek() == '*' || s1.Peek() == '/')
                            prio = 1;
                        else
                            prio = 0;
                        if (prio == 1)
                        {
                            if (ch == '+' || ch == '-')
```

```
                {
                    postfix += s1.Pop();
                    i--;
                }
                else
                {
                    postfix += s1.Pop();
                    i--;
                }
            }
            else
            {
                if (ch == '+' || ch == '-')
                {
                    postfix += s1.Pop();
                    s1.Push(ch);

                }
                else
                    s1.Push(ch);
            }
        }
    }
    else
    {
        postfix += ch;
    }
}
int len = s1.Count;
```

```
    for (int j = 0; j < len; j++)
        postfix += s1.Pop();
    return true;
}
static void Main(string[] args)
{
    string infix = "";
    string postfix = "";
    if (args.Length == 0)
    {
        infix = args[0];
        convert(ref infix, out postfix);
        System.Console.WriteLine("InFix␣␣:\t" + infix);
        System.Console.WriteLine("PostFix:\t" + postfix);
    }
    else
    {
        infix = "a+b*c-d";
        convert(ref infix, out postfix);
        System.Console.WriteLine("InFix␣␣␣:\t" + infix);
        System.Console.WriteLine("PostFix␣:\t" + postfix);
        System.Console.WriteLine();
        infix = "a+b*c-d/e*f";
        convert(ref infix, out postfix);
        System.Console.WriteLine("InFix␣␣␣:\t" + infix);
        System.Console.WriteLine("PostFix␣:\t" + postfix);
        System.Console.WriteLine();
        infix = "a-b/c*d-e--f/h*i++j-/k";
```

```
                convert(ref infix, out postfix);
                System.Console.WriteLine("InFix␣␣␣:\t" + infix);
                System.Console.WriteLine("PostFix␣:\t" + postfix);
                System.Console.WriteLine();
                Console.ReadLine();
            }
        }
    }
}
```

6.3 Пример работы программы

```
InFix   : a+b*c-d
PostFix : abc*+d-

InFix   : a+b*c-d/e*f
PostFix : abc*+de/f*-

InFix   : a-b/c*d-e--f/h*i++j-/k
PostFix : abc/d*-e--fh/i*-+j+k/-
```

7 Вывод

По результатам проделанной работы стало понятно, что уже более-менее умеющим программировать людям бывает тяжело погрузиться в теорию компиляторов, заставляя себя отказываться от мощных утилит высокоуровнего программирования. Тем не менее «выжить в условиях assembly» можно, и эта работа в том числе подтвердила это уверждение.