Министерство образования Республики Беларусь
Учреждение образования ««Полоцкий государственный университет имени Евфросинии Полоцкой»

Кафедра технологий программирования

Компиляторные технологии
Отчет по лабораторной работе № 2
Вариант 12

Выполнил — Ланцев Е.Н., студент гр. 21-ИТ-1,ФИТ

Проверил — Сыцевич Д.Н., Преподаватель-стажер кафедры ТП.

Полоцк
2022г.

Лабораторная работа № 2
«Организация таблиц идентификаторов»

Цель работы: изучить основные методы организации таблиц идентификаторов, получить представление о преимуществах и недостатках, присущих различным методам организации таблиц идентификаторов. Для выполнения лабораторной работы требуется написать программу, которая получает на входе набор идентификаторов, организует таблицы идентификаторов с помощью заданных методов, позволяет осуществить многократный поиск произвольного идентификатора в таблицах и сравнить эффективность методов организации таблиц. Список идентификаторов считать заданным в виде текстового файла. Длина идентификаторов ограничена 32 символами.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ (ответы на контрольные вопросы):

1. **Что такое таблица символов и для чего она предназначена? Какая информация может храниться в таблице символов?**

Таблица символов(идентификаторов)-специальное хранилище данных, организуемое компилятором для хранения некоторую информацию, связанную с каждым элементом исходной программы, и получения доступа к этой информации по имени элемента. Таблица идентификаторов состоит из набора полей данных (записей), каждое из которых может соответствовать одному элементу исходной программы.

Запись содержит всю необходимую компилятору информацию о данном элементе и может пополняться по мере работы компилятора. Количество записей зависит от способа организации таблицы идентификаторов, но в любом случае их не может быть меньше, чем элементов в исходной программе.

2. **Какие цели преследуются при организации таблицы символов?**

Цель: оперирование компилятором характеристиками основных элементов исходной программы – переменных, констант, функций и других лексических единиц входного языка.

3. **Какими характеристиками могут обладать лексические элементы исходной программы? Какие характеристики являются обязательными?**

Набор характеристик, соответствующий каждому элементу исходной программы, зависит от типа этого элемента, от его смысла (семантики) и, соответственно, от той роли, которую он исполняет в исходной и результирующей программах. В каждом конкретном случае этот набор характеристик может быть свой в зависимости от синтаксиса и семантики входного языка, от архитектуры целевой вычислительной системы и от структуры компилятора. Но есть типовые характеристики, которые чаще всего присущи тем или иным элементам исходной программы. Например для переменной – это ее тип и адрес ячейки памяти, для константы – ее значение, для функции – количество и типы формальных аргументов, тип возвращаемого результата, адрес вызова кода функции.

Главной характеристикой любого элемента исходной программы является его имя.

4. **Какие существуют способы организации таблиц символов?**

Основные способы организации таблиц идентификаторов:

- простые и упорядоченные списки;
- бинарное дерево;
- хэш-адресация с рехэшированием;
- хэш-адресация по методу цепочек;
- комбинация хэш-адресации со списком или бинарным деревом.

5. В чем заключается алгоритм логарифмического поиска? Какие преимущества он дает по сравнению с простым перебором и какие он имеет недостатки?

Алгоритм логарифмического поиска заключается в следующем: искомый символ сравнивается с элементом (N + 1)/2 в середине таблицы; если этот элемент не является искомым, то мы должны просмотреть только блок элементов, пронумерованных от 1 до (N + 1)/2 – 1, или блок элементов от (N + 1)/2 + 1 до N в зависимости от того, меньше или больше искомый элемент того, с которым его сравнили. Затем процесс повторяется над нужным блоком в два раза меньшего размера. Так продолжается до тех пор, пока либо искомый элемент не будет найден, либо алгоритм не дойдет до очередного блока, содержащего один или два элемента (с которыми можно выполнить прямое сравнение искомого элемента).

По сравнению с простым перебором число элементов, которые могут содержать искомый элемент, сокращается в два раза, максимальное число сравнений равно 1 + log2 N. Тогда время поиска элемента в таблице идентификаторов можно оценить как Тп = O(log2 N). Для сравнения: при N = 128 поиск в неупорядоченной таблице имеет в среднем 64 сравнения, а логарифмический метод требует не более 8 сравнений Метод имееет большое преимущество, поскольку время, затрачиваемое на поиск нужного элемента в массиве, имеет логарифмическую зависимость от общего количества элементов в нем.

Недостатком логарифмического поиска является требование упорядочивания таблицы идентификаторов. Так как массив информации, в котором выполняется поиск, должен быть упорядочен, время его заполнения уже будет зависеть от числа элементов в массиве. Таблица идентификаторов зачастую просматривается компилятором еще до того, как она заполнена, поэтому требуется, чтобы условие упорядоченности выполнялось на всех этапах обращения к ней. Следовательно, для построения такой таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.

6. Расскажите о древовидной организации таблиц идентификаторов. В чем ее преимущества и недостатки?

При древовидной организации таблиц каждый узел дерева представляет собой элемент таблицы, причем корневым узлом становится первый элемент, встреченный компилятором при заполнении таблицы. Дерево называется бинарным, так как каждая вершина в нем может иметь не более двух ветвей. Для определенности будем называть две ветви «правая» и «левая».

Будем считать, что алгоритм работает с потоком входных данных, содержащим идентификаторы. Первый идентификатор, как уже было сказано, помещается в вершину дерева. Все дальнейшие идентификаторы попадают в дерево по следующему алгоритму:

1) Выбрать очередной идентификатор из входного потока данных. Если очередного идентификатора нет, то построение дерева закончено.
2) Сделать текущим узлом дерева корневую вершину.
3) Сравнить имя очередного идентификатора с именем идентификатора, содержащегося в текущем узле дерева.

4) Если имя очередного идентификатора меньше, то перейти к шагу 5, если равно – прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно!), иначе – перейти к шагу 7.

5) Если у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе – перейти к шагу 6.

6) Создать новую вершину, поместить в нее информацию об очередном идентификаторе, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

7) Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе – перейти к шагу 8.

8) Создать новую вершину, поместить в нее информацию об очередном идентификаторе, сделать эту новую вершину правой вершиной текущего узла и вернуться к шагу 1.

Недостатки и преимущества:

Для данного метода число требуемых сравнений и форма получившегося дерева зависят от того порядка, в котором поступают идентификаторы. Например, если в рассмотренном выше примере вместо последовательности идентификаторов Ga, D1, M22, E, A12, BC, F взять последовательность A12, BC, D1, E, F, Ga, M22, то дерево выродится в упорядоченный однонаправленный связный список. Эта особенность является недостатком данного метода организации таблиц идентификаторов. Другими недостатками метода являются: необходимость хранить две дополнительные ссылки на левую и правую ветви в каждом элементе дерева и работа с динамическим выделением памяти при построении дерева.

Если предположить, что последовательность идентификаторов в исходной программе является статистически неупорядоченной (что в целом соответствует действительности), то можно считать, что построенное бинарное дерево будет невырожденным.

Несмотря на указанные недостатки, метод бинарного дерева является довольно удачным механизмом для организации таблиц идентификаторов. Он нашел свое применение в ряде компиляторов. Иногда компиляторы строят несколько различных деревьев для идентификаторов разных типов и разной длины [1, 2, 3, 7].

7. Что такое хэш-функции и для чего они используются? В чем суть хэш-адресации?

В реальных исходных программах количество идентификаторов столь велико, что даже логарифмическую зависимость времени поиска от их числа нельзя признать удовлетворительной. Необходимы более эффективные методы поиска информации в таблице идентификаторов. Лучших результатов можно достичь, если применить методы, связанные с использованием хэш-функций и хэш-адресации.

Хэш-функцией F называется некоторое отображение множества входных элементов R на множество целых неотрицательных чисел Z:

$$F(r) = n, r \in \mathbf{R}, n \in \mathbf{Z}$$

Сам термин «хэш-функция» происходит от английского термина «hash function» (hash – «мешать», «смешивать», «путать»).

Хэш-адресация заключается в использовании значения, возвращаемого хэш-функцией, в качестве адреса ячейки из некоторого массива данных. Тогда размер массива данных должен соответствовать области значений используемой хэш-функции. Следовательно, в реальном компиляторе область значений хэш-функции никак не должна превышать размер доступного адресного пространства компьютера.

8. Что такое коллизия? Почему она происходит? Можно ли полностью избежать коллизий?

Коллизия – это ситуация, когда двум или более идентификаторам соответствует одно и то же значение хэш-функции.

Она происходит, когда двум различным идентификаторам, начинающимся с одной и той же буквы, будет соответствовать одно и то же значение хэш-функции. Тогда при хэш-адресации в одну и ту же ячейку таблицы идентификаторов должны быть помещены два различных идентификатора, что явно невозможно.

Для полного исключения коллизий хэш-функция должна быть взаимно однозначной: каждому элементу из области определения хэш-функции должно соответствовать одно значение из ее множества значений, и наоборот – каждому значению из множества значений этой функции должен соответствовать только один элемент из ее области определения. Тогда любым двум произвольным элементам из области определения хэш-функции будут всегда соответствовать два различных ее значения. Теоретически для идентификаторов такую хэш-функцию построить можно, так как и область определения хэш-функции (все возможные имена идентификаторов), и область ее значений (целые неотрицательные числа) являются бесконечными счетными множествами, поэтому можно организовать взаимно однозначное отображение одного множества на другое.

Но на практике существует ограничение, делающее создание взаимно однозначной хэш-функции для идентификаторов невозможным.

9. Что такое рехэширование? Какие методы рехэширования существуют?

Одним из методов решения проблемы коллизии является метод рехэширования (или расстановки). Согласно этому методу, если для элемента A адрес n0 = h(A), вычисленный с помощью хэш-функции h, указывает на уже занятую ячейку, то необходимо вычислить значение функции n1 = h1(A) и проверить занятость ячейки по адресу n1. Если и она занята, то вычисляется значение h2(A), и так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение hi(A) не совпадет с h(A). В последнем случае считается, что таблица идентификаторов заполнена и места в ней больше нет – выдается информация об ошибке размещения идентификатора в таблице.

Методы рехэширования:

- Линейное рехеширование(определение всех хэш-функций hi для всех i)
- Случайное рехеширование(Использование в качестве pi для функции hi(A) = (h(A) + pi) mod Nm последовательности псевдослучайных целых чисел)
- Рехеширование сложением (Рехэширования основанные на квадратичных вычислениях или на вычислении произведения)
- Метод цепочек

10. Расскажите о преимуществах и недостатках организации таблиц идентификаторов с помощью хэш-адресации и рехэширования.

Преимущество - метод весьма эффективен, поскольку как время размещения элемента в таблице, так и время его поиска определяются только временем, затрачиваемым на вычисление хэш-функции, которое в общем случае несопоставимо меньше времени, необходимого для многократных сравнений элементов таблицы.

Недостатки:

- Неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать всей

области значений хэш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше.

- Необходимость соответствующего разумного выбора хэш-функции. Этот недостаток является настолько существенным, что не позволяет непосредственно использовать хэш-адресацию для организации таблиц идентификаторов.

11. В чем заключается метод цепочек?

При реализации метода цепочек в таблицу идентификаторов для каждого элемента добавляется еще одно поле, в котором может содержаться ссылка на любой элемент таблицы. Первоначально это поле всегда пустое (никуда не указывает). Также необходимо иметь одну специальную переменную, которая всегда указывает на первую свободную ячейку основной таблицы идентификаторов (первоначально она указывает на начало таблицы).

Метод цепочек работает по следующему алгоритму:

1. Во все ячейки хэш-таблицы поместить пустое значение, таблица идентификаторов пуста, переменная FreePtr (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов.
2. Вычислить значение хэш-функции n для нового элемента A. Если ячейка хэш-таблицы по адресу n пустая, то поместить в нее значение переменной FreePtr и перейти к шагу 5; иначе перейти к шагу 3.
3. Выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m и перейти к шагу 4.
4. Для ячейки таблицы идентификаторов по адресу m проверить значение поля ссылки. Если оно пустое, то записать в него адрес из переменной FreePtr и перейти к шагу 5; иначе выбрать из поля ссылки новый адрес m и повторить шаг 4.
5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A (поле ссылки должно быть пустым), в переменную FreePtr поместить адрес за концом добавленной ячейки. Если больше нет идентификаторов, которые надо поместить в таблицу, то выполнение алгоритма закончено, иначе перейти к шагу 2.

12. Расскажите о преимуществах и недостатках организации таблиц идентификаторов с помощью хэш-адресации и метода цепочек.

Преимущество - метод весьма эффективен, поскольку как время размещения элемента в таблице, так и время его поиска определяются только временем, затрачиваемым на вычисление хэш-функции, которое в общем случае несопоставимо меньше времени, необходимого для многократных сравнений элементов таблицы.

Недостатки:

- Неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать всей области значений хэш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше.
- Необходимость соответствующего разумного выбора хэш-функции. Этот недостаток является настолько существенным, что не позволяет непосредственно использовать хэш-адресацию для организации таблиц идентификаторов.

Метод цепочек является очень эффективным средством организации таблиц идентификаторов. Среднее время на размещение одного элемента и на поиск элемента в таблице для него зависит только от среднего числа коллизий, возникающих при вычислении хэш-функции. Накладные расходы памяти, связанные с необходимостью иметь одно дополнительное поле указателя в таблице идентификаторов на каждый ее элемент, можно признать вполне оправданными, так как возникает экономия

используемой памяти за счет промежуточной хэш-таблицы. Этот метод позволяет более экономно использовать память, но требует организации работы с динамическими массивами данных.

13. Как могут быть скомбинированы различные методы организации хеш-таблиц?

Алгоритм:

1. Как и для метода цепочек, в таблице идентификаторов организуется специальное дополнительное поле ссылки. Но в отличие от метода цепочек оно имеет несколько иное значение: при отсутствии коллизий для выборки информации из таблицы используется хэш-функция, поле ссылки остается пустым. Если же возникает коллизия, то через поле ссылки организуется поиск идентификаторов, для которых значения хэш-функции совпадают – это поле должно указывать на структуру данных для дополнительного метода: начало списка, первый элемент динамического массива или корневой элемент дерева.
2. Используется хэш-таблица, аналогичная хэш-таблице для метода цепочек. Если по данному адресу хэш-функции идентификатор отсутствует, то ячейка хэш-таблицы пустая. Когда появляется идентификатор с данным значением хэш-функции, то создается соответствующая структура для дополнительного метода, в хэш-таблицу записывается ссылка на эту структуру, а идентификатор помещается в созданную структуру по правилам выбранного дополнительного метода.

14. Расскажите о преимуществах и недостатках организации таблиц идентификаторов с помощью комбинированных методов.

В первом варианте при отсутствии коллизий поиск выполняется быстрее, но второй вариант предпочтительнее, так как за счет использования промежуточной хэш-таблицы обеспечивается более эффективное использование памяти.

Как и для метода цепочек, для комбинированных методов время размещения и время поиска элемента в таблице идентификаторов зависит только от среднего числа коллизий, возникающих при вычислении хэш-функции. Накладные расходы памяти при использовании промежуточной хэш-таблицы минимальны.

Недостатком комбинированных методов является более сложная организация алгоритмов поиска и размещения идентификаторов, необходимость работы с динамически распределяемыми областями памяти, а также бóльшие затраты времени на размещение нового элемента в таблице идентификаторов по сравнению с методом цепочек.

ВАРИАНТ 12

```
using System.Text;
using Microsoft.VisualBasic;

namespace Zamai.Lexer;

public class Parser
{
    private string _input;
    private Token? _nextToken;
    private Tokenizer _tokenizer;

    public Parser(string input)
```

```
    {
        _input = input;
        _tokenizer = new Tokenizer(input);
        _nextToken = _tokenizer.GetNextToken();
    }

    public Dictionary<string,Token> ChainParse()
    {
        var tokens = new Dictionary<string, Token>();
        while (_nextToken != null)
        {
            var hash = Encoding.ASCII.GetBytes(_nextToken.Value.ToString());

            while (tokens.ContainsKey(ByteArrayToString(hash)))
                hash = Encoding.ASCII.GetBytes($"{new
Random().Next(1000,9999)}");

            tokens.Add(ByteArrayToString(hash),_nextToken);
            _nextToken = _tokenizer.GetNextToken();
        }

        return tokens;
    }

    internal NodeToken BinaryTreeParse()
    {
        var token = new NodeToken(_nextToken);
        _nextToken = _tokenizer.GetNextToken();
        while (_nextToken != null)
        {
            token.AddNeibour(new NodeToken(_nextToken));
            _nextToken = _tokenizer.GetNextToken();
        }

        return token;
    }

    public void PrintHashChain(Dictionary<string, Token> tokens)
    {
        foreach (var token in tokens)
        {
            Console.Write($"{token.Key}: {token.Value.Type} -
{token.Value.Value}\n");
        }
    }

    public void FindHashChain(string searchToken)
    {
        var tokens = ChainParse();
        PrintHashChain(tokens);

        if
(tokens.ContainsKey(ByteArrayToString(Encoding.ASCII.GetBytes(searchToken))))
        {
            Console.WriteLine($"Found,
{tokens[ByteArrayToString(Encoding.ASCII.GetBytes(searchToken))].Value},
HASH:{ByteArrayToString(Encoding.ASCII.GetBytes(searchToken))}");
            return;
        }

        Console.WriteLine("Not found");
    }

    public void FindBinaryTree(string searchToken)
```

```
    {
        var tokens = BinaryTreeParse();
        PrintBinaryTreeAndSearch(tokens,searchToken);

    }

    public void PrintBinaryTreeAndSearch(NodeToken tree, string search)
    {
        PrintNodeAndSearch(tree, search);
    }

    private void PrintNodeAndSearch(NodeToken nodeToken, string search)
    {
        Console.Write($"{nodeToken.Value} ");
        if (nodeToken.Value == search)
        {
            Console.WriteLine("FOUND");
            return;
        }

        if (nodeToken.Left is not null)
        {
            PrintNodeAndSearch(nodeToken.Left, search);
        }

        if (nodeToken.Right is not null)
        {
            PrintNodeAndSearch(nodeToken.Right , search);
        }
    }

    static string ByteArrayToString(byte[] arrInput)
    {
        int i;
        StringBuilder sOutput = new StringBuilder(arrInput.Length);
        for (i=0;i < arrInput.Length; i++)
        {
            sOutput.Append(arrInput[i].ToString("X2"));
        }
        return sOutput.ToString();
    }
}
```

Листинг программы – Parser.cs

```
namespace Zamai.Lexer;

internal class Tokenizer
{
    private int _cursor;
    private string _input;

    public Tokenizer(string input)
    {
        _input = input;
        _cursor = 0;
    }
    public Token? GetNextToken()
    {
        if (_cursor == _input.Length)
            return null;

        var str = _input[_cursor..];
        foreach(var expressionRule in Specification.RegularExpressionRules)
        {
            var matched = expressionRule.Key.Match(str);
```

```
            if (matched.Success)
            {
                _cursor += matched.Value.Length;
                if (expressionRule.Value == null)
                    return GetNextToken();

                return new
Token((Specification.TokenType)expressionRule.Value, matched.Value);
            }
        }

        return null;
    }
}
```

Листинг программы –Tokenizer.cs

```
using System.Text;
using Zamai.Lexer;

namespace Zamai.Intepretator;

static class Program
{
    static void Task2()
    {
        var content = File.ReadAllText("../../../sample.zm", Encoding.UTF8);
        var lexer = new Parser(content);

        var userInput = Console.ReadLine();
        lexer.FindHashChain(userInput);



        var content1 = File.ReadAllText("../../../sample.zm", Encoding.UTF8);
        var lexer1 = new Parser(content);

        var userInput1 = Console.ReadLine();
        lexer1.FindBinaryTree(userInput1);
    }

    static void Main()
    {
         // Task2();


        var content1 = File.ReadAllText("../../../sample.zm", Encoding.UTF8);
        var lexer = new Parser(content1);
        var userInput = Console.ReadLine();
        lexer.FindHashChain(userInput);

    }
}
```

Листинг программы – Program.cs

```
47: Identifier - G
4A: Identifier - J
44: Identifier - D
53: Identifier - S
33313538: Identifier - D
4B4F: Identifier - KO
4B4F4A494F4A: Identifier - KOJIOJ
```

Рисунок 1 - Результат работы программы