ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
МИНИСТЕРСТВО ОБРАЗОВАНИЕ НАУКИ РФ
НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. Н.И. ЛОБАЧЕВСКОГО
ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНЫЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ

Автоматизация проектирования языковых средств для платформы .NET

Выполнил: студент группы 82-04
Морозов А. С.
Проверил: ассистент
Курылев А. Л.

Нижний Новгород, 2007

Содержание

Введение

В настоящее время искусственные языки, использующие для описания предметной области текстовое представление, широко применяются не только в программировании, но и в других областях. С их помощью описывается структура всевозможных документов, трехмерных виртуальных миров, графических интерфейсов пользователя и многих других объектов, используемых в моделях и в реальном мире. Для того чтобы эти текстовые описания были корректно составлены, а затем правильно распознаны и интерпретированы, используются специальные методы их анализа и преобразования. В основе методов лежит теория языков и формальных грамматик, а также теория автоматов. Программные системы, предназначенные для анализа и интерпретации текстов, называются трансляторами.

Несмотря на то, что к настоящему времени разработаны тысячи различных языков и их трансляторов, процесс создания новых приложений в этой области не прекращается. Это связно как с развитием технологии производства вычислительных систем, так и с необходимостью решения все более сложных прикладных задач. Кроме того, элементы теории языков и формальных грамматик применимы и в других разнообразных областях, например, при описании структур данных, файлов, изображений, представленных не в текстовом, а двоичном формате. Эти методы полезны и при разработке своих трансляторов даже там, где уже имеются соответствующие аналоги. Такая разработка может быть обусловлена различными причинами, в частности, функциональными ограничениями, отсутствием локализации, низкой эффективностью.

В данной работе мы сможем рассмотреть теорию создания трансляторов и компиляторов, найдем между ними отличие. Пройдем путь с самого начала и получим начальные знания в этой области, которых будет вполне достаточно для того что бы самому научится писать программы такого уровня. В частности мы реализуем компилятор языка Pascal, а точнее его подмножества.

Архитектура

1.1 Основные определения

Для начала, разберемся с определениями, которые будут часто встречается в данной лабораторной работе.

Транслятор – обслуживающая программа, преобразующая исходную программу, предоставленную на входном языке программирования, в рабочую программу, представленную на объектном языке.

Приведенное определение относится ко всем разновидностям транслирующих программ. Однако у каждой из таких программ могут иметься свои особенности по организации процесса трансляции. В настоящее время трансляторы разделяются на три основные группы: ассемблеры, компиляторы и интерпретаторы.

Ассемблер – системная обслуживающая программа, которая преобразует символические конструкции в команды машинного языка. Специфической чертой ассемблеров является то, что они осуществляют дословную трансляцию одной символической команды в одну машинную. Таким образом, язык ассемблера (еще называется автокодом) предназначен для облегчения восприятия системы команд компьютера и ускорения программирования в этой системе команд. Программисту гораздо легче запомнить мнемоническое обозначение машинных команд, чем их двоичный код. Поэтому, основной выигрыш достигается не за счет увеличения мощности отдельных команд, а за счет повышения эффективности их восприятия.

Вместе с тем, язык ассемблера, кроме аналогов машинных команд, содержит множество дополнительных директив, облегчающих, в частности, управление ресурсами компьютера, написание повторяющихся фрагментов, построение многомодульных программ. Поэтому выразительность языка, намного богаче, чем просто языка символического кодирования, что значительно повышает эффективность программирования.

Компилятор – это обслуживающая программа, выполняющая трансляцию на машинный язык программы, записанной на исходном языке программирования. Также как и ассемблер, компилятор обеспечивает преобразование программы с одного языка на другой (чаще всего, в язык конкретного компьютера). Вместе с тем, команды исходного языка значительно отличаются по организации и мощности от команд машинного языка. Существуют языки, в которых одна команда исходного языка транслируется в 7-10 машинных команд. Однако есть и такие языки, в которых каждой команде может соответствовать 100 и более машинных команд (например, Пролог). Т.к. современные процессоры достигли таких высот, что можно встретить и такую ситуацию, когда несколько команд исходного языка умещается в 1 машинную. Кроме того, в исходных языках достаточно часто используется строгая типизация данных, осуществляемая через их предварительное описание. Программирование может опираться не на кодирование алгоритма, а на тщательное обдумывание структур данных или классов. Процесс трансляции с таких языков обычно называется компиляцией, а исходные языки обычно относятся к языкам программирования высокого уровня (или высокоуровневым языкам).

Интерпретатор – программа или устройство, осуществляющее пооператорную трансляцию и выполнение исходной программы. В отличие от компилятора, интерпретатор не порождает на выходе программу на машинном языке. Распознав команду исходного языка, он тут же выполняет ее. Как в компиляторах, так и в интерпретаторах используются одинаковые методы анализа исходного текста программы. Но интерпретатор позволяет начать обработку данных после написания даже одной команды. Это делает процесс разработки и отладки

программ более гибким. Кроме того, отсутствие выходного машинного кода позволяет не "захламлять" внешние устройства дополнительными файлами, а сам интерпретатор можно достаточно легко адаптировать к любым машинным архитектурам, разработав его только один раз на широко распространенном языке программирования. Поэтому, интерпретируемые языки, типа Java Script, VB Script, получили широкое распространение. Недостатком интерпретаторов является низкая скорость выполнения программ. Обычно интерпретируемые программы выполняются в 50-100 раз медленнее программ, написанных в машинных кодах.

Синтаксис – совокупность правил некоторого языка, определяющих формирование его элементов. Иначе говоря, это совокупность правил образования семантически значимых последовательностей символов в данном языке. Синтаксис задается с помощью правил, которые описывают понятия некоторого языка. Примерами понятий являются: переменная, выражение, оператор, процедура. Последовательность понятий и их допустимое использование в правилах определяет синтаксически правильные структуры, образующие программы. Именно иерархия объектов, а не то, как они взаимодействуют между собой, определяются через синтаксис. Например, оператор может встречаться только в процедуре, а выражение в операторе, переменная может состоять из имени и необязательных индексов и т.д. Синтаксис не связан с такими явлениями в программе как "переход на несуществующую метку" или "переменная с данным именем не определена". Этим занимается семантика.

Семантика – правила и условия, определяющие соотношения между элементами языка и их смысловыми значениями, а также интерпретацию содержательного значения синтаксических конструкций языка. Объекты языка программирования не только размещаются в тексте в соответствии с некоторой иерархией, но и дополнительно связаны между собой посредством других понятий, образующих разнообразные ассоциации. Например, переменная, для которой синтаксис определяет допустимое местоположение только в описаниях и некоторых операторах, обладает определенным типом, может использоваться с ограниченным множеством операций, имеет адрес, размер и должна быть описана до того, как будет использоваться в программе.

Синтаксический анализатор – компонента компилятора, осуществляющая проверку исходных операторов на соответствие синтаксическим правилам и семантике данного языка программирования. Несмотря на название, анализатор занимается проверкой и синтаксиса, и семантики. Он состоит из нескольких блоков, каждый из которых решает свои задачи. Более подробно будет рассмотрен при описании структуры транслятора.

Стек – динамический линейный список, все записи в котором выбираются, вставляются и удаляются с одного конца, называемого вершиной стека. Стек можно сравнить со стопкой книг на полу. Вы можете добавлять книги на вершину стопки и убирать их с вершину, но добавить или убрать книгу из середины стопки вы не можете. Стеки часто называют списками типа последний вошел - первый вышел (Last-in-First-out list – LIFO)

Очередь – это упорядоченный список, где элементы добавляются в один конец, а удаляются с другого. Группа людей у кассы магазина образует очередь. Вновь прибывшие люди становятся в конец очереди. Когда клиент доходит до начала очереди, кассир обслуживает его, и человек уходит из очереди. Поэтому очереди иногда называют списками типа первый вошел – первый вышел (First – In – First - Out – FIFO).

Лексема – одна из смысловых единиц во входном тексте для компилятора.

Инфиксная запись – форма записи, в которой операторы указываются между соответствующими операндами, Например: (a+b)*c. Чтобы указать порядок вычислений при инфиксной записи приходится использовать скобки, которые не нужны в префиксной и постфиксной.

Префиксная запись – вид записи, предложенный польским математиком Яном Лукасевичем, в которой каждый оператор предшествует операнду, Например: $a + b$ имеет вид

$+ab$. Если все операторы содержат ровно два операнда или если каждый оператор включает в себя определенное число операндов, скобки не требуются, так как порядок вычисления всегда определен единственным образом. Такую запись можно определить как бесскобочную.

Постфиксная запись – вид записи, предложенный польским математиком Яном Лукасевичем, в которой каждый оператор записывается вслед операнду. Например: $a + b$ имеет вид $abs +$, а $a + b * c$ записывается так $abs * +$. Если каждому оператору соответствует определенное число операндов, например, точно два, то скобочки при такой записи не нужны, поскольку порядок вычисления всегда определяется однозначно; поэтому данная запись может быть названа бесскобочной.

1.2 Архитектура компилятора

Языки программирования достаточно сильно отличаются друг от друга по назначению, структуре, семантической сложности, методам реализации. Это накладывает свои специфические особенности на разработку конкретных трансляторов. Языки программирования являются инструментами для решения задач в разных предметных областях, что определяет специфику их организации и различия по назначению. В качестве примера можно привести такие языки как Фортран, ориентированный на научные расчеты, С, предназначенный для системного программирования, Пролог, эффективно описывающий задачи логического вывода, Лисп, используемый для рекурсивной обработки списков. Эти примеры можно продолжить. Каждая из предметных областей предъявляет свои требования к организации самого языка. Поэтому, можно отметить разнообразие форм представления операторов и выражений, различие в наборе базовых операций, а так же снижение эффективности программирования при решении задач, не связанных с предметной областью. Языковые различия отражаются и в структуре трансляторов.

Вместе с тем, все языки программирования обладают рядом общих характеристик и параметров. Эта общность определяет и схожие для всех языков принципы организации трансляторов:

1. Языки программирования предназначены для облегчения программирования. Поэтому их операторы и структуры данных более мощные, чем в машинных языках.
2. Для повышения наглядности программ вместо числовых кодов используются символические или графические представления конструкций языка, более удобные для их восприятия человеком.
3. Для любого языка определяется:
4. Множество символов, которые можно использовать для записи правильных программ (алфавит), основные элементы.
5. Множество правильных программ (синтаксис).
6. "Смысл" каждой правильной программы (семантика).

Независимо от специфики языка любой транслятор можно считать функциональным преобразователем F, обеспечивающим однозначное отображение X в Y, где X - программа на исходном языке, Y - программа на выходном языке. Поэтому сам процесс трансляции формально можно представить достаточно просто и понятно:

$$Y = F(X)$$

Формально каждая правильная программа X - это цепочка символов из некоторого алфавита A, преобразуемая в соответствующую ей цепочку Y, составленную из символов алфавита B.

Язык программирования, как и любая сложная система, определяется через иерархию понятий, задающую взаимосвязи между его элементами. Эти понятия связаны между собой в соответствии с синтаксическими правилами. Каждая из программ, построенная по этим правилам, имеет соответствующую иерархическую структуру. В связи с этим для всех языков и их программ можно дополнительно выделить следующие общие черты: каждый язык должен содержать правила, позволяющие порождать программы, соответствующие этому языку или распознавать соответствие между написанными программами и заданным языком.

Процесс нахождения синтаксической структуры заданной программы называется синтаксическим разбором.

Другой характерной особенностью всех языков является их семантика. Она определяет смысл операций языка, корректность операндов. Цепочки, имеющие одинаковую

синтаксическую структуру в различных языках программирования, могут различаться по семантике (что, например, наблюдается в C++, Pascal, Basic для приведенного выше фрагмента арифметического выражения).

Общие свойства и закономерности присущи как различным языкам программирования, так и трансляторам с этих языков. В них протекают схожие процессы преобразования исходного текста. Не смотря на то, что взаимодействие этих процессов может быть организовано различным путем, можно выделить функции, выполнение которых приводит к одинаковым результатам. Назовем такие функции фазами процесса трансляции.

Кроме этого можно выделить единый для всех фаз процесс анализа и исправление ошибок, существующих в обрабатываемом исходном тексте программы:

1.2.1 Лексический анализатор

Основная задача лексического анализа - разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов, или лексем, т.е. выделить эти слова из непрерывной последовательности символов. Все символы входной последовательности с этой точки зрения разделяются на символы, принадлежащие каким-либо лексемам, и символы, разделяющие лексемы (разделители). В некоторых случаях между лексемами может и не быть разделителей. С другой стороны, в некоторых языках лексемы могут содержать незначащие символы (пробел в Фортране).

С точки зрения дальнейших фаз анализа лексический анализатор выдает информацию двух сортов: для синтаксического анализатора, работающего вслед за лексическим, существенна информация о последовательности классов лексем, ограничителей и ключевых слов, а для контексного анализа, работающего вслед за синтаксическим, важна информация о конкретных значениях отдельных лексем (идентификаторов, чисел и т.д.). Поэтому общая схема работы лексического анализатора такова. Сначала выделяем отдельную лексему (возможно, используя символы- разделители). Если выделенная лексема - ограничитель, то он (точнее, некоторый его признак) выдается как результат лексического анализа. Ключевые слова распознаются либо явным выделением непосредственно из текста, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов. Если да, то выдается признак соответствующего ключевого слова, если нет - выдается признак идентификатора, а сам идентификатор сохраняется отдельно. Если выделенная лексема принадлежит какому-либо из других классов лексем (число, строка и т.д.), то выдается признак класса лексемы, а значение лексемы сохраняется.

Лексический анализатор может работать или как самостоятельная фаза трансляции, или как подпрограмма, работающая по принципу "дай лексему". В первом случае выходом лексического анализатора является файл лексем, во втором - лексема выдается при каждом обращении к лексическому анализатору (при этом, как правило, тип лексемы возвращается как значение функции "лексический анализатор", а значение передается через глобальную переменную). С точки зрения формирования значений лексем, принадлежащих классам лексем, лексический анализатор может либо просто выдавать значение каждой лексемы и в этом случае построение таблиц переносится на более поздние фазы, либо он может самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.). В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу.

Работа лексического анализатора описывается формализмом конечных автоматов. Однако непосредственное описание конечного автомата неудобно практически. Поэтому для описания лексических анализаторов, как правило, используют либо формализм регулярных выражений, либо формализм контекстно-свободных грамматик, а именно подкласса

автоматных, или регулярных, грамматик. Все три формализма (конечных автоматов, регулярных выражений и автоматных грамматик) имеют одинаковую выразительную мощность. По описанию лексического анализатора в виде регулярного выражения или автоматной грамматики строится конечный автомат, распознающий соответствующий язык.

1.2.2 Синтаксический анализатор

Методов синтаксического разбора существует достаточно много, но в основном можно выделить два способа разбора:

- Нисходящий разбор. Суть разбора состоит в том, что текст некоей программы, который с самого начала представлен в виде очень большой строки, постепенно разбивается на лексемы, к лексемам применяется синтаксический анализ для построения внутреннего представления программы.
- Восходящий разбор. Суть этого типа разбора состоит в том, что программа представляется в виде последовательности лексем. которые далее "снизу вверх" склеиваются в более сложные предложения языка, после успешной склейки производится построение внутреннего представления программы.

На самом деле в реальных компиляторах редко используются те или иные методы разбора в чистом виде. Обычно используются комбинации известных методов, или же несколько изменённые методы разбора, в основном с целью повышения скорости разбора. Однако, последнее десятилетие из-за повышения при компиляции программ удельной массы проходов оптимизации, время синтаксического разбора занимает небольшой процент общего времени компиляции, потому нет особого смысла заниматься оптимизацией разбора.

1.2.3 Проходы оптимизации

Оптимизирующие преобразования обычно производятся над некоторым участком программы, и, в зависимости от типа преобразования, рассматриваются участки различной структуры и величины. Основные градации величины участков экономии такие:
оператор – в основном арифметические операторы являются участком экономии в рамках одного оператора;

- базовый блок – последовательность операторов с единственной точкой входа и без ветвлений, базовый блок (ББ) был объектом многих ранних исследований по оптимизации;
- самый вложенный цикл, в этом контексте выполняются многие оптимизации;
- идеально вложенное гнездо циклов (все циклы, кроме самого вложенного содержат в себе только один оператор – более глубоко вложенный цикл);
- вложенное гнездо циклов (любого формата);
- процедура (глобальная оптимизация);
- множество процедур, рассматриваемых вместе (межпроцедурная оптимизация).

Каждое оптимизирующее преобразование применяется с целью улучшения некоторой характеристики (характеристик) программы. Зачастую преобразование при улучшении одной из характеристик, ухудшает другую. Приведём примеры характеристик:

- количество занимаемых ресурсов процессора (например, функциональных устройств);
- минимизация количества выполняемых операций;

- минимизация количества обращений в оперативную память;
- минимизация размера программы и использования памяти для данных;
- минимизация энергопотребления.

Базовым понятием, используемым во всех оптимизирующих преобразованиях уровня базового блока и выше, является понятие зависимости по данным.

Примеры оптимизации

Уменьшение избыточных вычислений

Оригинальный код	Код после оптимизации

`int i,j;`
`j=8;`
`for (i=0; i<2+j; i++)`
`  a[i]=a[i]+(1+2+4*3-10);`
	`int i;`
	`for(i=0; i<10;i++)`
	`  a[i] = a[i]+5;`

Прямое преобразование

Оригинальный код	Код после оптимизации

`int x;`
`double y;`
`y = y * 2;`
`x = x * 4;`
`x = x / 2;`
	`int x;`
	`double y;`
	`y = y + y;`
	`x = x << 2;`
	`x = x >> 1;`

Удаление неиспользуемого кода

Оригинальный код	Код после оптимизации

`if (1>0)`
`  Function1();`
`int x=0;`
`if (x>0)`
`  Function2();`
`Function3();`
	`Function3();`

Подстановка операторов

Оригинальный код	Код после оптимизации

`int np1 = n + 1;`
`for( i = 0; i < n; i++ )`
`  a[np1] = a[np1] + a[i];`
	`for( i = 0; i < n; i++ )`
	`  a[n+1] = a[n+1] + a[i];`

Раскрутка циклов

Оригинальный код	Код после оптимизации

`for (i = 1; i < n-1; i++)`
`  a[i]=(a[i-1]+a[i+1])/2;`
	`for (i = 1; i < n-2; i+=2)`
	`{`
	`  a[i]=(a[i-1]+a[i+1])/2;`
	`  a[i+1]=(a[i]+a[i+2])/2;`
	`}`
	`if ((n–2) % 2 == 1)`
	`  a[n-2] =(a[n-3]+a[n-1])/2;`

Эта крошечная часть того, что происходит в мощном компиляторе. Здесь приведены пара примеров, которая увеличивает скорость выполнения программы на маленький процент. Но представим, что в нашей программе не сто строчек кода, а несколько сотен тысяч. Тогда процент будет существенен. Но тогда время на оптимизацию таких простых случаев необходимо больше. Именно поэтому компиляторы пишутся не 1 день, годами, для того что бы сделать код исполняемой программы как можно меньше и быстрее.

1.2.4 Генерация кода

Процесс генерации кода для базового блока (ББ) состоит из трёх основных этапов:

1. выбор инструкций целевого МП – сопоставление некоторой операции выражения соответствующей инструкции, перед сопоставлением инструкций производится определение класса регистров для имеющихся в программе переменных и промежуточных значений;
2. распределение регистров – размещение переменных в регистрах самым эффективным образом;
3. составление расписания инструкций – формирование машинного кода базового блока из выбранных инструкций с наименьшим временем исполнения.

Первый этап генерации кода — сопоставления графа зависимостей по данным и управлению программы инструкциям целевого МП. Для сопоставления инструкций базовый блок представляется в виде ациклического сильно связанного двудольного орграфа потока данных (орграф ББ). При таком представлении ББ автоматически выделяется суперскалярный параллелизм. В орграфе ББ выделяется множество начальных вершин (либо вводится фиктивная начальная вершина-команда) и проводится топологическая сортировка для подготовки этапа выбора инструкций.

Выбор инструкций для сопоставления с орграфом ББ происходит в процессе полного покрытия орграфа ББ необходимым количеством копий графов инструкций МП. Процесс сопоставления является итеративным и может происходить одновременно с окончательным распределением регистров и построением расписания инструкций (генерацией кода).

Практически важнейшей проблемой при генерации коду является распределение регистров, основанное на определении времени жизни переменных при выполнении кода

функции. Большинство арифметико-логических команд МП с RISC архитектурой не могут оперировать непосредственно со значениями в памяти, поэтому компилятор генерирует последовательность команд для загрузки необходимых в некоторый момент времени значений переменных в регистры. Для определения необходимых на текущий момент значений в регистровом файле необходимо частично разрешить задачу определения времени жизни переменных, копии которых находятся в регистровом файле (регистровых переменных).
В современных моделях МП с длинным командным словом, улучшенных RISC-МП, процессоров с архитектурой EPIC, нетрадиционных параллельных процессоров, ПЦОС ёмкость регистрового файла достигает 64 – 256 регистров. Алгоритм распределения регистров должен эффективно задействовать такое количество регистров.

В случае исполнения одной команды за такт (типичный RISC) особых проблем не возникает. Сначала строится расписание арифметико-логических команд, затем между командами могут быть вставлены (при необходимости) дополнительные команды загрузки или сохранения регистров в память. При более сложной архитектуре процессора, установка дополнительных команд проводилась итеративно до определения оптимального варианта, а затем все команды участвовали в составлении расписания.

При количестве одновременно исполняемых операций в длинной команде от 4 до 8 (или больше) проблема распределения регистров становится гораздо более сложной. Обычно это связано с тем, что возможности подачи данных из оперативной памяти в регистровый файл ограничены. Несколько легче эта проблема в случае, если мы обрабатываем регулярные большие куски данных – например, массивы, но, если доступ требуется к нескольким переменным, расположенным в памяти далеко друг от друга, проблема становится очевидной – вычислительные возможности процессора простаивают, так как скорость подачи данных относительно невелика. Решение этой проблемы путём увеличения количества интерфейсов к оперативной памяти частично решает проблему подачи данных (гарвардская и супергарвардская архитектуры), но компиляция становится ещё более сложной задачей. Естественно, существует некоторое количество неочевидных оптимизаций, связанных с группированием данных и загрузкой векторов, но такие средства – редкость.

Для старых моделей МП класса Intel 80486 проблема скорости подачи данных не поднималась в принципе, потому что ёмкость активной части регистрового файла не превышала 6 регистров, кроме того, инструкции МП обычно прямо оперировали значениями, расположенными в памяти. Однако, с появлением суперскалярных архитектур, проблема подачи данных компенсировалась в основном кэшем, и, в общем, анализ проводился с учётом того, что данные уже находятся в кэше 1-го уровня. Если рассмотреть процессор Itanium, то становится понятным, что необходимы серьёзные меры для оптимального использования 128 целочисленных регистров и 128 регистров для чисел с плавающей точкой.

Среди алгоритмов распределения регистров рассмотрим два метода, широко использующихся на практике: распределение регистров с помощью раскраски графов и распределением регистров с помощью линейного сканирования потока данных.

Оба метода используют т.н. информацию о времени жизни переменных для нахождения распределения переменных-кандидатов на загрузку в регистры на машинные регистры.

Для достижения цели метод, связанный с раскраской графов, представляет информацию о времени жизни в виде графа зависимостей, в котором вершины представляют собой переменные-кандидаты и между двумя вершинами существует дуга, если их времена жизни пересекаются – то есть эти переменные не могут содержаться в одном регистре. Для целевого процессора, содержащего *N* регистров, нахождение раскраски этого графа зависимостей времени жизни в *N* цветов эквивалентно распределению переменных-кандидатов по регистрам без конфликтов. Алгоритм итеративно строит граф зависимостей и пытается его раскрасить. Если раскраска не удаётся, некоторые кандидаты перемещаются в память,

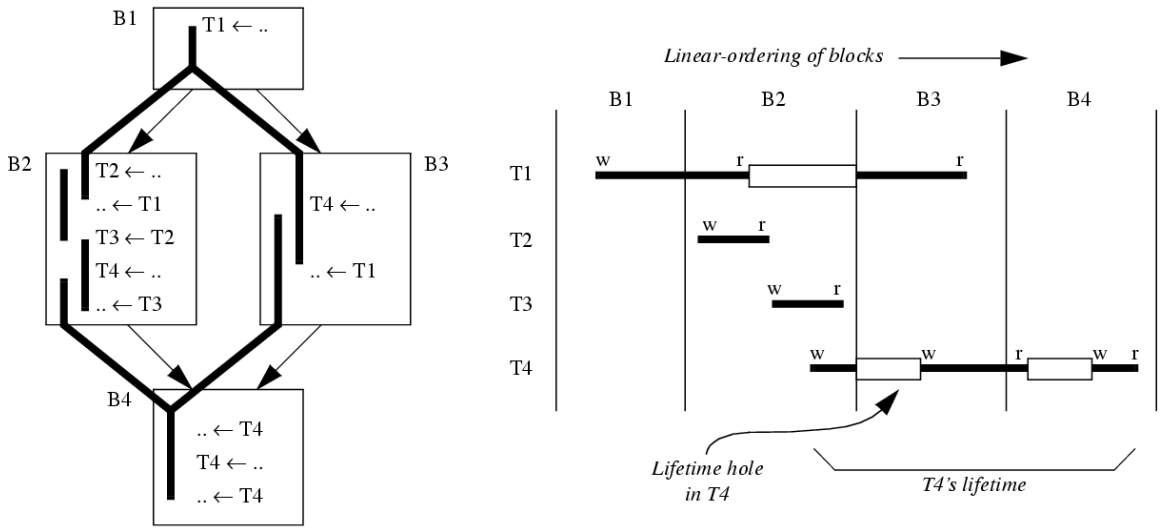


Рис. 1. Распределение вычислений в процессоре

убирается часть связей в графе, вставляется код для выгрузки/загрузки переменных (*spill code*), и процесс повторяется. На практике основное время уходит на процесс построения графа, размер которого является $O(n^2)$ от количества переменных-кандидатов. Так как модуль программы может иметь сотни переменных, и компилятор сам может генерировать достаточное количество временных переменных в случае агрессивных оптимизаций, раскраска графа может занять длительное время.

Распределение регистров с помощью сканирования потока данных оперирует с понятием «времени жизни», который начинается в момент присвоения переменной этого значения и заканчивается в момент последнего использования этого значения в некоторой ветви программы. При сканировании просматривается в линейном порядке код программы, с целью определения количества «активных» переменных. Количество «активных» интервалов определяет количество необходимых регистров в данной точке программы. В случае если их слишком много, ряд переменных временно сохраняется из регистров в память. В принципе, время работы такого распределителя линейно.

Так как последний алгоритм весьма прямолинеен, используется ряд улучшений, рассмотрим их.

При анализе времени жизни полезно выделить места, где переменная не имеет полезного значения. Например, на рисунке **Ошибка! Источник ссылки не найден.** показан базовый блок и обозначено время жизни переменных с указанием *дыр* во времени жизни.

В частности, если мы распределяем регистр *r* для переменной *t*, но в её времени жизни есть дыра, в которую умещается время жизни другой переменной *u*, то мы можем распределить переменной *u* тот же регистр *r*. В частности, на рисунке переменная *T3* целиком помещается в дыру во времени жизни *T1*.

Для подсчета времён жизни и «дыр» в них необходим реверсный проход по программному коду.

Таким образом, регистры представляются в виде ячеек, каждой из которых в некоторый момент времени может соответствовать только одно значение. Для того, чтобы несколько переменных могли быть размещены в ячейке необходимо, чтобы их времена жизни не пересекались. Если говорить о дырах во времени жизни, то две переменные также можно разместить в одном регистре, если время жизни одной из них целиком умещается в дыре во времени жизни другой. Эта трактовка несколько неоднозначна – фактически дыра во времени жизни обозначает, что образована новая переменная, но здесь понятие «дыры» вводится для удобства представления.

Таким образом, распределитель просматривает программу в прямом направлении, собирая информацию о времени жизни. При необходимости назначения временной переменной *t* регистра выбирается либо один из свободных регистров, либо же выбирается регистр, содержащий переменную, у которой сейчас существует «дыра» и время жизни *t* целиком умещается в «дыре». После сопоставления регистра все обращения к *t* в программе заменяются обращениями к *r*.

Если свободных регистров нет, необходимо подыскать «занятый» регистр, освобождение которого будет наименее болезненно. Должна быть выбрана переменная, которая будет позже остальных востребована, при этом необходимо учитывать глубину вложенности цикла, где она востребуется. Фактически, все переменные сохраняются в регистрах как можно дольше и записываются в память только в том случае, если регистр нужен для «более приоритетной» переменной. Ещё одна оптимизация заключается в том, что регистры, значение переменной в которых не отличается от значения в памяти, не будут записываться обратно в память.

При распределении регистров необходимо решить ещё одну проблему. В случае, если переменная вытесняется из памяти в одном или обоих базовых блоках B2 и/или B3, к моменту перехода потока управления в блок B4 фактически две копии переменной могут находиться в разных регистрах, либо же в одном потоке переменная может находиться в регистре, а в другом – в памяти. В этом случае производятся следующие действия: 1) если копии переменной находятся в разных регистрах, то в один из потоков управления вставляется команда копирования из регистра в регистр. В случае, если переменная в одном потоке управления изменена, а в другом её нет в регистрах, делается сохранение переменной в памяти. Вообще, поведение распределителя в данном случае может зависеть от обстоятельств – если эта переменная будет использована практически сразу после слияния потоков управления, есть смысл загрузить её в регистр в той ветви, гдё её нет в регистре до слияния потоков.

Более гибкое решение предполагает, что регистры у нас равноправны, следовательно, привязка регистров может быть нежёсткой, чтобы затем можно было подкорректировать номера при слиянии потоков, что позволит сэкономить несколько команд, но с другой стороны, усложнит алгоритм.

Ещё несколько улучшений алгоритма касаются поддержки современных микропроцессоров. Для процессоров с длинным командным словом алгоритмы усложнены – так как у нас несколько времён жизни переменных могут начинаться и завершаться одновременно, кроме того, команда загрузки/сохранения обычно может быть только одна на несколько параллельно исполняющихся арифметических команд. В этом случае переменные должны загружаться в память значительно раньше места их затребования, так как одна длинная команда может потребовать более трёх-четырёх переменных.

Подытоживая, отметим, что в принципе алгоритмы распределения регистров достаточно просты и понятны. С другой стороны архитектура процессора оказывает определяющее влияние, и алгоритм становится достаточно сложным.

1.3 Платформа .NET

Далее приведена краткая архитектура платформы .NET. Это было сделано для того, что бы читатель имел представление о том, что такое .NET в целом и как он работает. Более подробную информацию вы можете прочитать в книжке [6].

1.3.1 Архитектура .NET. Компиляция в управляемые модули

Итак, вы решили использовать .NET Framework как платформу разработки. Отлично! Ваш первый шаг заключается в том, чтобы определить вид приложения или компонента, которые вы собираетесь построить. Пусть вы уже решили этот второстепенный вопрос, все спроектировано, спецификации написаны, и вы готовы начать.

Теперь вы должны выбрать язык программирования. Обычно это непростая задача — ведь у разных языков разные возможности. Так, в неуправляемом C/C++ вы имеете доступ к системе на довольно низком уровне. Вы можете распоряжаться памятью по своему усмотрению, создавать потоки и т. д. А вот Visual Basic 6 позволяет очень быстро строить пользовательские интерфейсы и легко управлять COM-объектами и базами данных.

Название исполняющей среды — «общеязыковая исполняющая среда» (common language runtime, CLR) — говорит само за себя: это исполняющая среда, которая подходит для разных языков программирования. Возможности CLR доступны любым языкам. Если исполняющая среда использует исключения для обработки ошибок, то во всех языках можно получать сообщения об ошибках посредством исключений. Если исполняющая среда позволяет создавать поток, во всех языках могут создаваться потоки.

Фактически во время выполнения CLR не знает, на каком языке разработчик написал исходный код. А значит, вам следует выбрать тот язык, который позволяет решить вашу задачу простейшим способом. Вы можете писать свой код на любом языке, если используемый компилятор предназначен для CLR.

Если это так, каковы преимущества одного языка перед другим? Под компиляцией я подразумеваю контроль синтаксиса и анализ «корректного кода». Компиляторы проверяют ваш исходный код, убеждаются, что все написанное имеет какой-то смысл, и затем генерируют код, описывающий ваши намерения. Различные языки позволяют создавать программы, используя различный синтаксис. Не стоит недооценивать значение этого выбора. Для математических или финансовых приложений выражение ваших мыслей на языке APL может сохранить много дней работы по сравнению с применением синтаксиса языка Perl, например.
Microsoft создает компиляторы для нескольких языков, предназначенных для этой платформы: C++ с управляемыми расширениями, C# (произносится «си шарп»[1]), Visual Basic, JScript, J# (компилятор языка Java) и ассемблер Intermediate Language (IL). Кроме Microsoft, еще несколько компаний работают над компиляторами, которые генерируют код, работающий в CLR. Мне известны компиляторы Alice, APL, COBOL, Component Pascal, Eiffel, Fortran, Haskell, Mercury, ML, Mondrian, Oberon, Perl, Python, RPG, Scheme и Smalltalk.

На Рис. 2 показан процесс компиляции файлов с исходным кодом. Как видите, вы можете создавать файлы с исходным кодом на любом языке, поддерживающем CLR. Затем вы используете соответствующий компилятор для проверки синтаксиса и анализа исходного кода. Независимо от компилятора результатом является *управляемый модуль* (managed module). Управляемый модуль — это стандартный переносимый исполняемый (portable executable, PE) файл Windows, который требует для своего выполнения CLR. В будущем формат файла PE смогут использовать и другие ОС.

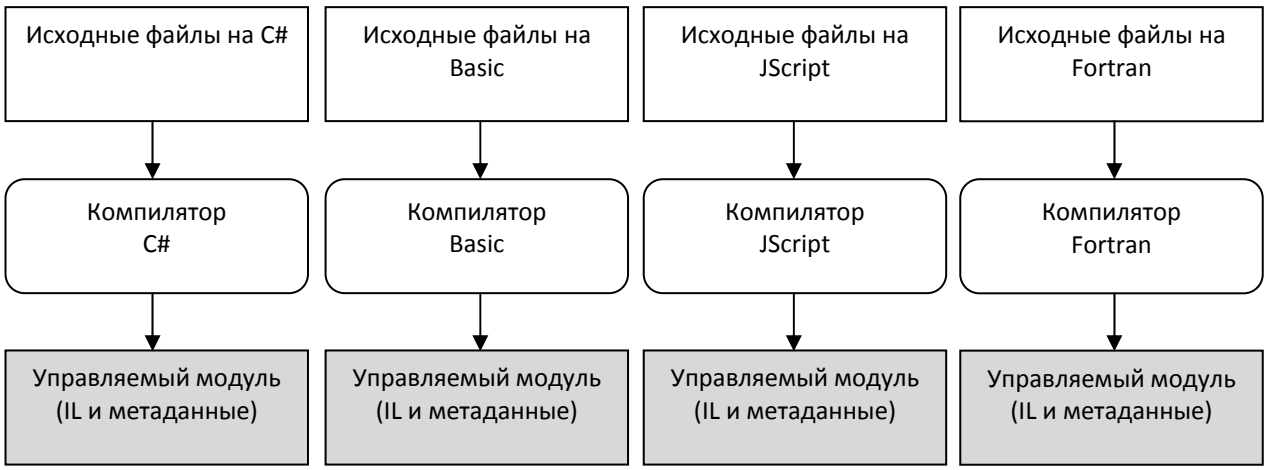


Рис. 2. Компиляция исходного кода в управляемые модули

В Табл. 1 писаны составные части управляемого модуля.

Часть	**Описание**
Заголовок PE	Стандартный заголовок PE – файла Windows, аналогичный заголовку Common Object File Format (COFF). Заголовок показывает тип файла: CUI, CUI или DLL, он также имеет временную метку, показывающую, когда файл был собран. Для модулей, содержащих только IL-код, основной объем информации в PE-заголовке игнорируется. Для модулей содержащих процессорный код, этот заголовок содержит сведения о процессорном коде.
Заголовок CLR	Содержит информацию (интерпретируемую CLR и утилитами), которая превращает этот модуль в управляемый. Заголовок включает нужную версию CLR, некоторые флаги, метку метаданных MethodDef точки входа в управляемый модуль (метод *Main*), а также месторасположение/размер метаданных модуля, ресурсов, строгого имени, некоторых флагов и др. информацию.
Метаданные	Каждый управляемый модуль содержит таблицы метаданных. Есть два основных вида таблиц: описывающие типы и члены, определенные в вашем исходном коде, описывающие типы и члены, на которые имеются ссылки в вашем исходном коде.
Код Intermediate Language (IL)	Код, создаваемый компилятором при компиляции исходного кода. Впоследствии CLR скомпилирует IL в команды процессора.

Табл. 1. Части управляемого модуля

В прошлом почти все компиляторы генерировали код для конкретных процессорных архитектур, таких как x86, IA64, Alpha или PowerPC. Все CLR-совместимые компиляторы вместо этого генерируют IL-код. (Мы рассмотрим IL-код подробнее ниже.) IL-код иногда называют *управляемым* (managed code), потому что CLR управляет его жизненным циклом и выполнением.

Каждый компилятор, предназначенный для CLR, кроме генерации IL-кода, также должен создавать полные *метаданные* (metadata) для каждого управляемого модуля. Коротко говоря, метаданные — это просто набор таблиц данных, описывающих то, что определено в модуле, например, типы и их члены. Метаданные имеют также таблицы, указывающие, на что ссылается управляемый модуль, например, на импортируемые типы и их члены. Метаданные расширяют возможности таких старых технологий, как библиотеки типов и файлы языка описания интерфейсов (Interface Definition Language, IDL). Важно заметить, что метаданные CLR гораздо полнее. И в отличие от библиотек типов и IDL они всегда связаны с файлом, содержащим IL-код. Фактически метаданные всегда встроены в тот же EXE/DLL, что и код, так что их нельзя разделить. Так как компилятор генерирует метаданные и код одновременно и

привязывает их к конечному управляемому модулю, метаданные и IL-код, описываемый ими, никогда не бывают рассинхронизированы.

Метаданные используются для разных целей.

- Метаданные устраняют необходимость в заголовочных и библиотечных файлах при компиляции, так как все сведения о типах/членах, на которые есть ссылки, содержатся в файле с IL-кодом, в котором они реализованы. Компиляторы могут читать метаданные прямо из управляемых модулей.
- Visual Studio .NET использует метаданные для того, чтобы помочь вам писать код. Ее функция IntelliSense анализирует метаданные и сообщает, какие методы предлагает тип и какие параметры требуются этим методам.
- В процессе верификации кода CLR использует метаданные, чтобы убедиться, что ваш код совершает только «безопасные» операции.
- Метаданные позволяют сериализовать поля объекта в блок памяти на удаленной машине и затем десериализовать, восстановив объект и его состояние на удаленной машине.
- Метаданные позволяют сборщику мусора отслеживать жизненный цикл объектов. Сборщик мусора может определить тип любого объекта и благодаря метаданным знает, какие поля в объекте ссылаются на другие объекты.

Microsoft C#, Visual Basic, JScript, J# и IL-ассемблер всегда создают управляемые модули, которые требуют для своего выполнения CLR. Для выполнения любого управляемого модуля на машине конечного пользователя должен быть установлен CLR, так же как для выполнения приложений MFC или Visual Basic 6 должны быть установлены библиотека классов Microsoft Foundation Class (MFC) или динамически подключаемые библиотеки Visual Basic.

По умолчанию компилятор Microsoft C++ создает неуправляемые модули — файлы EXE или DLL, с которыми мы все хорошо знакомы. Они не требуют CLR для своего выполнения. Однако если вызвать компилятор C++ с новым ключом в командной строке, он может создать управляемые модули, требующие CLR для своего выполнения. Компилятор C++ — уникальный среди упомянутых компиляторов Microsoft, так как это единственный язык, позволяющий писать как управляемый, так и неуправляемый код и встраивать их в один модуль. Это очень важное свойство, поскольку оно позволяет писать основной объем приложения в управляемом коде (в целях безопасности типов и совместимости компонентов) и в то же время иметь доступ к существующему неуправляемому коду C++.

1.3.2 Объединение управляемых модулей в сборку

На самом деле среда CLR работает не с модулями, а со *сборками.* Сборка (assembly) — это абстрактная концепция, понимание которой поначалу может вызвать затруднения. Во-первых, это логическая группировка одного или нескольких управляемых модулей или файлов ресурсов. Во-вторых, это самая маленькая единица, с точки зрения повторного использования, безопасности и отслеживания версий. В зависимости от того, какие средства и компиляторы вы выбрали, вы можете сделать однофайловую или многофайловую сборку.

По умолчанию компиляторы сами выполняют работу по превращению созданного управляемого модуля в сборку, т. е. компилятор C# создает управляемый модуль, который содержит декларацию, указывающую, что сборка состоит только из одного файла. Итак, в проектах, которые имеют только один управляемый модуль и не содержат файлов ресурсов

(или данных), сборка и будет управляемым модулем, и вам не нужно прилагать дополнительных усилий при компоновке приложения. Если вы хотите сгруппировать набор файлов в сборку, вам нужно знать о дополнительных инструментах (вроде компоновщика сборок AL.exe) и их опциях командной строки.

Сборка позволяет разделить логическое и физическое понятия повторно используемого, разворачиваемого компонента с управлением версиями. Как вы разделите код и ресурсы на разные файлы, исключительно ваше дело. Так, вы можете поместить редко используемые типы и ресурсы в отдельные файлы, которые являются частью сборки. Отдельные файлы могут загружаться из Web по мере надобности. Если файлы никогда не потребуются, они не будут скачаны, что сохранит место на жестком диске и ускорит установку. Сборки позволяют вам разбить на части процесс развертывания файлов и в то же время рассматривать все файлы как единый набор.

Модули сборки также содержат сведения о других сборках, на которые они ссылаются, включая номера версий. Эти сведения делают сборку *самоописываемой* (self-describing). Иначе говоря, CLR знает о сборке все, что нужно для ее выполнения. Дополнительной информации не требуется ни в реестре, ни в службе каталогов Active Directory. А раз так, развертывать сборки гораздо проще, чем неуправляемые компоненты.

1.3.3 Выполнение программ на платформе .NET

Каждая создаваемая сборка может быть либо исполняемым приложением, либо DLL, содержащими набор типов (компонентов) для использования в исполняемом приложении. За управление исполнением кода, содержащегося в этих сборках, отвечает, конечно же, CLR. Это значит, что на компьютере, выполняющем приложение, должна быть установлена платформа .NET Framework. В Microsoft создан дистрибутивный пакет .NET Framework для свободного распространения, который вы можете бесплатно поставлять своим заказчикам. В дальнейшем .NET Framework будет включена в новые версии Windows, и вам не придется ее поставлять со своими сборками.

Когда вы компонуете EXE-сборку, компилятор/компоновщик встраивает специальную информацию в заголовок и в раздел .text PE-файла результирующей сборки. При запуске EXE-файла эта специальная информация приводит к загрузке и инициализации CLR. Затем CLR находит метод, являющийся точкой входа приложения, и позволяет приложению начать выполнение.

Управляемые модули содержат метаданные и код на промежуточном языке (IL). IL — не зависящий от процессора машинный язык, разработанный Microsoft после консультаций с несколькими коммерческими и учебными институтами, специализирующимися на разработке языков и компиляторов. IL — язык более высокого уровня в сравнении с большинством других машинных языков. Он позволяет работать с объектами и имеет команды для создания и инициализации объектов, вызова виртуальных методов и непосредственного манипулирования элементами массивов. Он даже имеет команды генерации и захвата исключений для обработки ошибок, IL можно рассматривать как объектно-ориентированный машинный язык.

Обычно разработчики программируют на высокоуровневых языках, таких как C# или Visual Basic. Компиляторы этих языков создают IL-код. Между тем такой код может быть написан и на языке ассемблера, и Microsoft предоставляет ассемблер IL — ILAsm.exe. Кроме того, Microsoft поставляет и дизассемблер IL — ILDasm.exe.

Имейте в виду, что любой язык высокого уровня скорей всего использует лишь часть потенциала CLR. При этом язык ассемблера IL открывает доступ ко всем возможностям CLR. Так что, если выбранный вами язык программирования скрывает именно те функции CLR, которые

вам нужны, можно написать какой-то фрагмент на ассемблере или на другом языке программирования, позволяющем их задействовать.

Единственный способ узнать о возможностях CLR, доступных при использовании конкретного языка, — изучить соответствующую документацию. В этой книге я пытаюсь остановиться на возможностях CLR как таковой и на том, какие из них доступны при программировании на C#. Могу предположить, что в других книгах и статьях CLR будет рассмотрена с точки зрения других языков и разработчики будут иметь представление лишь о тех ее функциях, что доступны при использовании выбранного ими языка. Если считать, что выбранный язык позволяет достичь желаемых результатов, такой подход не так уж плох.

Современные процессоры не могут исполнять команды IL напрямую, но процессоры будущего могут реализовать такую возможность. Для выполнения какого-либо метода его IL-код должен быть преобразован в команды процессора. Этим занимается JIT-компилятор CLR.

Вот что происходит при первом обращение к методу (Рис. 3).

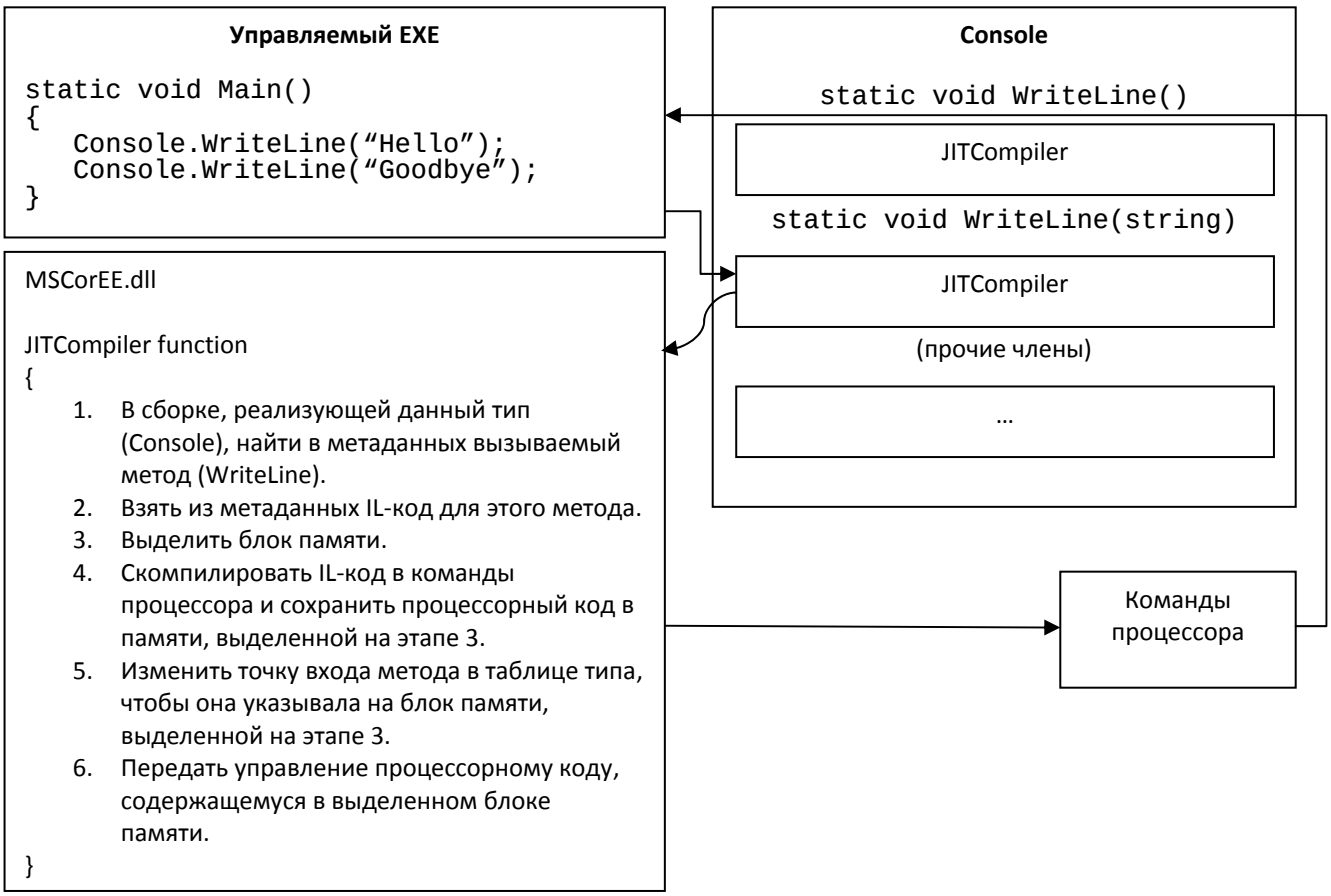


Рис. 3. Первый вызов метода

Непосредственно перед исполнением Main CLR находит все типы, на которые ссылается код Main. При этом CLR выделяет внутренние структуры данных, используемые для управления доступом к типам, на которые есть ссылки. На Рис. 3 метод Main ссылается на единственный тип — Console, и CLR выделяет единственную внутреннюю структуру. Эта внутренняя структура данных содержит по одной записи для каждого метода, определенного в типе. Каждая запись содержит адрес, по которому можно найти реализацию метода. При инициализации этой структуры CLR заносит в каждую запись адрес внутренней, недокументированной функции, содержащейся в самой CLR.

Когда Main первый раз обращается к WriteLine, вызывается функция JITCompiler. Она отвечает за компиляцию IL-кода вызываемого метода в собственные команды процессора.

Поскольку IL компилируется непосредственно перед исполнением (just in time), этот компонент CLR часто называют *JITter* или *JIT-компилятор* (JIT-compiler).

Функции JITCompiler известен вызываемый метод и тип, в котором он определен. JITCompiler ищет в метаданных соответствующей сборки IL-код вызываемого метода. Затем JITCompiler проверяет и компилирует IL-код в собственные команды процессора, которые сохраняются в динамически выделенном блоке памяти. После этого JITCompiler возвращается к внутренней структуре данных типа и заменяет адрес вызываемого метода адресом блока памяти, содержащего собственные команды процессора. В завершение JITCompiler передает управление коду в этом блоке памяти. Этот код — реализация метода WriteLine (той его версии, что принимает параметр String). Из этого метода управление возвращается в Main, который продолжает нормальную работу.

Затем Main обращается к WriteLine вторично. К этому моменту код WriteLine уже проверен и скомпилирован, так что производится обращение к блоку памяти, минуя вызов JITCompiler. Отработав, метод WriteLine возвращает управление Main.

Производительность теряется только при первом вызове метода. Все последующие обращения выполняются «на полной скорости»: повторная верификация и компиляция не производятся.

JIT-компилятор хранит команды процессора в динамической памяти. Это значит, что скомпилированный код уничтожается при завершении приложения. Так что, если потом вы снова вызовете приложение или если вы одновременно запускаете два его экземпляра (в двух разных процессах ОС), JIT-компилятор заново будет компилировать IL-код в команды процессора.

Для большинства приложений снижение производительности, связанное с работой JIT-компилятора, невелико. Большинство приложений раз за разом обращается к одним и тем же методам. На производительности это скажется только раз. К тому же скорей всего больше времени занимает выполнение самого метода, а не обращение к нему.

Полезно также знать, что JIT-компилятор CLR оптимизирует процессорный код аналогично компилятору неуправляемого кода C++. И опять же: создание оптимизированного кода занимает больше времени, но производительность его будет гораздо выше, чем неоптимизированного.

Разработчики с опытом программирования на неуправляемых C/C++, вероятно, задумаются над производительностью. В конце концов неуправляемый код компилируется для конкретного процессора и при вызове просто исполняется. В управляемой же среде компиляция производится в две фазы. Сначала компилятор проходит исходный код и переводит его в IL. Но для исполнения сам IL-код нужно перевести в команды процессора в период выполнения, что требует дополнительной памяти и процессорного времени.

Когда JIT-компилятор компилирует IL-код в команды процессора в период выполнения, он располагает более полными сведениями о среде выполнения в сравнении с компилятором неуправляемого кода. Вот некоторые способы, которые позволяют управляемому коду «опередить» неуправляемый.

- JIT-компилятор может обнаружить, что приложение запускается на процессоре Pentium 4 и сгенерировать процессорный код, полностью использующий все преимущества особых команд Pentium 4. Неуправляемые приложения обычно компилируются в расчете на процессор, являющийся «наименьшим общим знаменателем», избегая специфических команд, которые заметно повышают производительность приложения на новейших процессорах.

- JIT-компилятор может обнаружить, что некоторая проверка всегда приводит к отрицательному результату на конкретной машине. Например, рассмотрим метод с таким кодом:

```
if (numtterOfCPUs > 1)
{
        …
}
```

Этот код указывает JIT-компилятору, что для машины с одним процессором не нужно генерировать никакие команды процессора. В этом случае собственный код процессора оптимизирован для конкретной машины: он короче и выполняется быстрее.

- CLR может проанализировать выполнение кода и перекомпилировать IL-код в команды процессора при выполнении приложения. Перекомпилированный код может быть реорганизован с учетом обнаруженных некорректных прогнозов ветвления.

Это лишь малая часть аргументов в пользу того, что управляемый код будущего будет исполняться лучше сегодняшнего неуправляемого

Если ваши эксперименты покажут, что JIT-компилятор CLR не обеспечивает нужную производительность, можете использовать утилиту NGen.exe, поставляемую с .NET Framework SDK. NGen.exe компилирует весь IL-код некоторой сборки в процессорный и сохраняет результирующий код процессора в дисковом файле. При загрузке сборки в период выполнения, CLR автоматически проверяет наличие предварительно скомпилированной версии сборки и, если она есть, загружает скомпилированный код, так что компиляция в период выполнения не производится.

1.3.4 IL и верификация

IL ориентирован на работу со стеком, т. е. все его команды помещают операнды в стек исполнения и извлекают результаты из стека. Поскольку IL не поддерживает команды работы с регистрами, разработчики компиляторов могут расслабиться: не нужно думать об управлении регистрами, да и команд IL меньше (ведь команд работы с регистрами нет).

Команды IL не связаны и с типами. Так, команда IL *add* складывает два последних операнда, помещенных в стек; нет отдельной 32- и 64-разрядной команды *add*. При выполнении команда *add* определяет типы операндов в стеке и делает, что требуется.

По-моему, главное достоинство IL не в том, что он позволяет абстрагироваться от конкретного типа процессора. Главное — надежность приложений. При компиляции IL в команды процессора CLR выполняет *верификацию,* в процессе которой проверяется, все ли «безопасно» делает высокоуровневый IL-код: нет ли, например, чтения памяти, в которую ничего не записывалось, нужное ли число параметров передается методу и корректны ли их типы, правильно ли используются возвращаемые методами значения, имеют ли все методы операторы возврата и т. д.

Если выясняется, что IL-код «небезопасен», генерируется исключение, и соответствующий метод не выполняется.

1.4 Грамматики и формальные языки

Мы неформально определяем язык как подмножество множеств всех предложений из "слов" или символов некоторого основного словаря. Нас не интересует смысл этих предложений. Например, русский язык состоит из предложений, которые являются последовательностями, составленными из слов (Пусть, всегда, будет, солнце и т.д) и знаков пунктуации (например, запятые, точки, скобки). Язык программирования Паскаль состоит из программ, которые являются последовательностями, составленными из таких символов, как begin, end, знаков пунктуации, букв и цифр. Язык четных чисел состоит из последовательностей, составленных из цифр 1, ..., 9, в которых последней цифрой должны быть О, 2, 4, 6 или 8.

Алфавит - это непустое конечное множество элементов. Назовём элементы алфавита символами. Всякая конечная последовательность символов алфавита А называется цепочкой. Вот несколько цепочек "в алфавите" $A = \{a, b, c\}$: $a, b, c, ab,$. Мы также допускаем существование пустой цепочки , т. е. цепочки, не содержащей ни одного символа. Важен порядок символов в цепочке; так цепочка ab не то же самое, что ba, и $abca$ отличается от $aabc$. Длина цепочки x (записывается как $|x|$) равна числу символов в цепочке. Таким образом,

$$|| = 0, \qquad |a| = 1, \qquad |abb| = 3$$

Заглавные буквы M, N, S, T, U, используются как переменные или имена символов алфавита, в то время как строчные буквы u, v, w используются для обозначения цепочек символов. Таким образом, можно написать x=STV, и это означает, что x является цепочкой, состоящей из символов S, T и V именно в таком порядке.

(Вместо термина цепочка (в английском языке - string) некоторые авторы используют термины строка или строчка.)

Если x и y - цепочки, то их катенацией xy является цепочка y, полученная путем дописывания символов цепочки y вслед за символами цепочки x. Например, если $x = XY$, $y = YZ$, то $xy = XYYZ$ и $yx = YZXY$.

Поскольку A, не содержащая символов, то в соответствии с правилом катенации для любой цепочки x мы можем записать:

$$xA = Ax = x$$

Если $z = xy$ - цепочка, то x - голова, а y - хвост цепочки z. x - правильная голова, если y - не пустая цепочка, z - правильный хвост, если x - не пустая цепочка. Множества цепочек в алфавите обычно обозначаются заглавными буквами A, B. Произведение AB двух множеств цепочек А и В определяется как:

$$AB = \{xy \mid x \in A, y \in B\}$$

Например, если $A = \{a, b\}, B = \{c, d\}$, то множество $AB = \{ac, ad, bc, bd\}$.

Также необходимо определить понятие степени цепочек. Если x - цепочка, то x^0- пустая цепочка , $x^1 = x$, $x^2 = xx$, и в общем случае: $x^n = \underbrace{xxx \dots x}_{n}$

Иногда удобнее и, как правило, нагляднее писать x . .. вместо xy, если нас не интересует y - остальная часть цепочки. Таким образом, три точки "..." обозначают любую возможную цепочку, включая и пустую. Наиболее часто встречаются следующие обозначения:

Обозначение	Смысл
z = x...	x - голова цепочки z, нам безразличен хвост.
z = ...x	x - хвост цепочки z. нам безразлична голова.
z = ...x...	x встречается где-то в цепочке z.
z = S...	Символ S-первый символ цепочки z.

z = ...S Символ S - последний символ цепочки z.
z = ...S... Символ S встречается где-то в цепочке z.

Грамматикой $G[Z]$ называется конечное, непустое множество правил; Z - это символ, который должен встретиться в левой части, по крайней мере, одного правила. Он называется начальным символом. Все символы, которые встречаются в левых и правых частях правил, образуют словарь V.

Если из контекста ясно, какой символ является начальным символом Z, мы будем писать G вместо $G[Z]$. Пример. Грамматика G $[< число >]$ содержит следующие 13 правил:

(1) <число> : := <чс>
(2) <чс> : := <чс><цифра>
(3) <чс> : := <цифра>
(4) <цифра> : := 0
(5) <цифра> : := 1
(6) <цифра> : := 2
(7) <цифра> : := 3
(8) <цифра> : := 4
(9) <цифра> : := 5
(10) <цифра> : := 6
(11) <цифра> : := 7
(12) <цифра> : := 8
(13) <цифра> : := 9

V={0, I, 2, 3, 4, 5, 6, 7, 8, 9, <цифра>, <чс>, <число>}.

1.4.1 Способ записи языка. Метаязык Хомского - Щутценберже

Существуют различные способы записи синтаксических правил, что в основном определяется условными обозначениям и ограничениями на структуру правил, принятыми в используемых метаязыках. Метаязыки используются для задания грамматики языков программирования со времен Алгола 60. Еще раньше они начали использоваться при описании небольших языков в статьях, посвященных формальным грамматикам. Кратко рассмотрим основные вехи становления и развития метаязыков. Во всех случаях будем определять идентификатор.

Метаязык Хомского вышел из недр математической логики, но из-за своей громоздкости не имеет столь популярности, как его же продолжение. Метаязык Хомского используется только для описания небольших абстрактных языков. Более компактное описание возможно с применением метаязыка Хомского - Щутценберже, использующего следующие обозначения метасимволов:

- символ “=” отделяет левую часть правила от правой;
- нетерминалы обозначаются буквой A с индексом, указывающим на его номер;
- терминалы - это символы используемые в описываемом языке;
- каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом “+”, что позволяет, при желании, использовать в левой части только разные нетерминалы

Введение возможности альтернативного перечисления позволило сократить описание языков. Описание идентификатора будет выглядеть следующим образом:

1. $A_1 = A + B + C + D + E + F + G + H + I + J + K + L + M + N + O + P + Q + R + S + T + U + V + W + X + Y + Z + a + b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q + r + s + t + u + v + w + x + y + z$
2. $A_2 = 0 + 1 + 2 + 4 + 5 + 6 + 7 + 8 + 9$
3. $A_3 = A_1 + A_3A_1 + A_3A_2$

1.4.2 Способ записи языка. БНФ (Формы Бекуса Наура)

Метаязыки Хомского и Хомского - Щутценберже использовались в математической литературе при описании простых абстрактных языков. Метаязык, предложенный Бэкусом и Науром, впервые использовался для описания синтаксиса реального языка программирования Алгол 60. Наряду с новыми обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов. Это сделало описание языка нагляднее и позволило в дальнейшем широко использовать данную нотацию для описания реальных языков программирования. Были использованы следующие обозначения:

- символ "::=" отделяет левую часть правила от правой;
- нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки "<" и ">";
- терминалы - это символы, используемые в описываемом языке;
- каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты "|".

Пример описания идентификатора с использованием БНФ:

- $< \text{буква} >:: = \text{A}|\text{B}|\text{C}|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z$
- $< \text{цифра} >:: = 0|1|2|3|4|5|6|7|8|9$
- $< \text{идентификатор} >::= < \text{буква} > \mid < \text{идентификатор} >< \text{буква} > \mid < \text{идентификатор} >< \text{цифра} >$

Правила можно задавать и раздельно:

- $< \text{идентификатор} >:: = < \text{буква} >$
- $< \text{идентификатор} >:: = < \text{идентификатор} >< \text{буква} >$
- $< \text{идентификатор} >:: = < \text{идентификатор} >< \text{цифра} >$

Расширенные Бэкуса-Наура формы (РБНФ)

Метаязыки, представленные выше, позволяют описывать любой синтаксис. Однако, для повышения удобства и компактности описания, целесообразно вести в язык дополнительные конструкции. В частности, специальные метасимволы были разработаны для описания необязательных цепочек, повторяющихся цепочек, обязательных альтернативных цепочек. Существуют различные расширенные формы метаязыков, незначительно отличающиеся друг от друга. Их разнообразие зачастую объясняется желанием разработчиков языков программирования по-своему описать создаваемый язык. К примерам таких широко известных

метаязыков можно отнести: метаязык PL/I, метаязык Вирта, используемый при описании Модулы-2, метаязык Кернигана-Ритчи, описывающий Си. Зачастую такие языки называются расширенными формами Бэкуса-Наура (РБНФ).

В частности, РБНФ, используемые Виртом, имеют следующие особенности:

- Квадратные скобки "**[**" и "**]**" означают, что заключенная в них синтаксическая конструкция может отсутствовать;
- фигурные скобки "**{**" и "**}**" означают ее повторение (возможно, 0 раз);
- круглые скобки "**(**" и "**)**" используются для ограничения альтернативных конструкций;
- сочетание фигурных скобок и косой черты "**{/**" и "**/}**" используется для обозначения повторения один и более раз. Нетерминальные символы изображаются словами, выражающими их интуитивный смысл и написанными на русском языке.

Если нетерминал состоит из нескольких смысловых слов, то они должны быть написаны слитно. В этом случае для повышения удобства в восприятии фразы целесообразно каждое ее слово начинать с заглавной буквы или разделять слова во фразах символом подчеркивания. Терминальные символы изображаются словами, написанными буквами латинского алфавита (зарезервированные слова) или цепочками знаков, заключенными в кавычки. Синтаксическим правилам предшествует знак "$" в начале строки. Каждое правило оканчивается знаком "**.**" (точка). Левая часть правила отделяется от правой знаком "=" (равно), а альтернативы - вертикальной чертой "|". В соответствии с данными правилами синтаксис идентификатора будет выглядеть следующим образом:

$ буква = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|
"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|"a"|"b"|"c"|"d"|"e"|
"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z".
$ цифра = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
$ идентификатор = буква {буква | цифра}.

1.4.3 Способ записи языка. Диаграммы Вирта

Диаграммы Вирта представляют собой альтернативную форму записи грамматики, которая представляется графом, имеющим начальную и конечную вершину. Вершины графа представляют собой позиции разбора входного потока. Вершины соединяются направленными дугами, которые подписаны цепочками символов (или нетерминалами - именами "вложенных" диаграмм Вирта). В случае, если в некоторой позиции диаграммы при разборе языка далее по тексту входного языка находится некоторая цепочка символов, совпадающая с одной из подписей на дугах, исходящих из этой вершины, то будет совершён переход в позицию, в которую ведёт подписанная этой цепочкой символов дуга. Если возможности перехода нет, то разбор согласно диаграмме считается неуспешным. Впервые Вирт их применил для задания языка Pascal. Поэтому в большинстве книг по Pascal можно обнаружить Диаграммы Вирт, например в книжке [13].

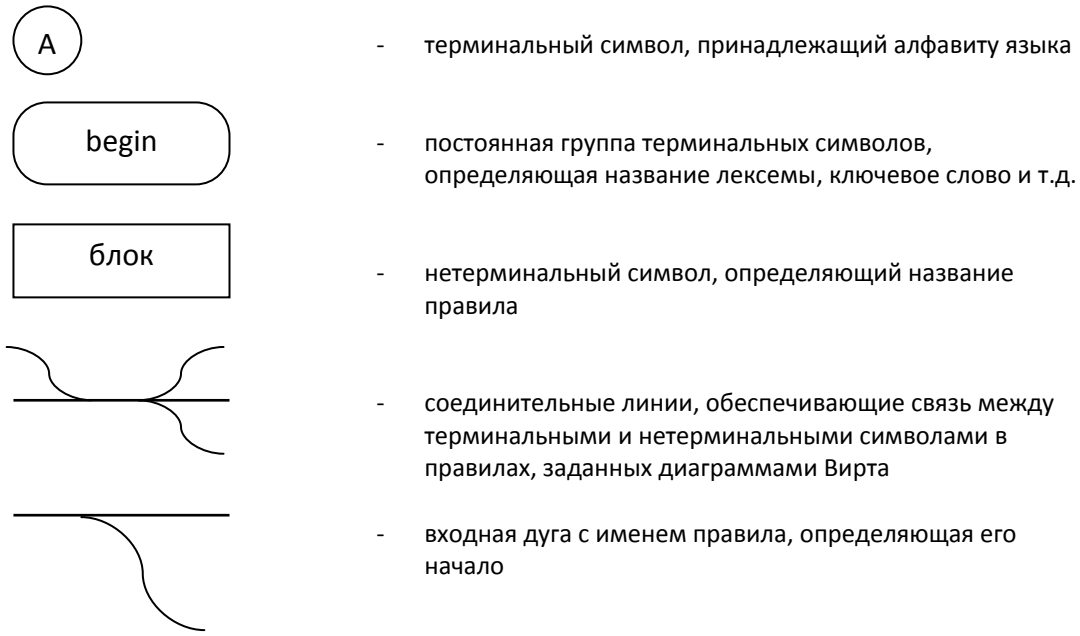


Рис. 4. Графические примитивы, используемые при построении

- терминальные символы и их постоянные группы располагаются в окружностях или прямоугольниках со скругленным вертикальными сторонами;
- нетерминальные символы заносятся внутрь прямоугольников;
- каждый графический элемент, соответствующий терминалу или нетерминалу, имеет по одному входу и выходу, которые обычно рисуются на противоположных сторонах;
- каждому правилу соответствует своя графическая диаграмма, на которой терминалы и нетерминалы соединяются посредством дуг;
- альтернативы в правилах задаются ветвлением дуг, а итерации - их слиянием;
- должна быть одна входная дуга (располагается обычно слева и сверху), задающая начало правила и помеченная именем определяемого нетерминала, и одна выходная, задающая его конец (обычно располагается справа и снизу).

Обычно стрелки на дугах диаграмм не ставятся, а направления связей отслеживаются движением от начальной дуги в соответствии с плавными изгибами промежуточных дуг и ветвлений. Таким же образом определяются входы и выходы терминалов и нетерминалов. Специальных стандартов на диаграммы Вирта нет, поэтому графические обозначения могут меняться в зависимости от средств рисования. Можно, например, использовать псевдографику или просто текстовые символы, связи со стрелками. Однако такой вид правил менее удобен для восприятия и поэтому применяется крайне редко.

Диаграммы Вирта позволяют задавать альтернативы, рекурсии, итерации и по изобразительной мощности эквивалентны РБНФ. Но графическое отображение правил более наглядно. Кроме этого допускается произвольное проведение дуг, что уменьшает количество элементов в правиле за счет его неструктурированности. Диаграммы Вирта являются удобным исходным документом для построения лексического и синтаксического анализаторов.

Разбор по диаграмме заканчивается в конечной вершине. Обычно каждой дуге приписывается определённая процедура, производящая полезные действия. Типичным случаем использования диаграммы Вирта является *сканер*.

1.5 Построение собственного транслятора/компилятора

1.5.1 Лексический и синтаксический анализ.

Как уже было оговорено ранее в главе «1.2.1 Лексический анализатор» основная задача лексического анализа - разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов, или лексем, т.е. выделить эти слова из непрерывной последовательности символов.

Разработать универсальный компилятор не получится, как не старайся. Поэтому будем считать, что мы ограничимся следующими типами данных:

- int – целый числа
- double – вещественные числа
- string – строка
- bool – булевой тип данных

Будем считать, что лексема принадлежит типу:

- int – если в её состав входят только следующий набор допустимых символов: {"0","1","2","3","4","5","6","7","8","9"}.
- double – если в её состав входят только следующий набор допустимых символов: {"0","1","2","3","4","5","6","7","8","9",".",","}
- string – если лексема окружена кавычками (одинарными или двойными).
- bool – если в её состав входят только следующий набор допустимых символов: {0,1}

Все же остальное будем разбивать просто на слова, где под словами будем понимать последовательность символов разделенных пробелами.

Для просто, мы организуем функцию, которая будет «выдирать» по кусочку – по символу, из нашего текста, на который мы смотрим как на последовательность символов. Также будем утверждать, что текст исходной программы является несвязанной последовательностью символов и если в ходе разбора окажется, что наш компилятор не выдал ошибок, значит, текст исходной задачи корректен.

Создадим один большой класс, который мы будет дополнять в процессе разбора компилятора, который и будет собственно компилятором. Обзовем его так:

```
class language
{
      …
}
```

Напишем вспомогательный класс, который поможет нам хранить текст. Дополним его нашей нужной функцией по считыванию одного символа.

Тем более такой класс можно использовать и в других программах. Назовем его так:

```
class source
{
…
}
```

Более полное описание вы сможете найти в файле «***class_source.hpp***» и «***class_source.cpp***»

Теперь, необходимо все-таки реализовать нашу функцию по преобразованию из последовательности символов в лексему.

Так как функция получилось довольно таки громоздкая то код приводить я думаю не следует его можно будет прочитать в файле «***lang.cpp***»

Самое простое позади теперь необходимо реализовать все функции синтаксического анализатора. Да, их будет не одна.

1.5.2 Промежуточное представление

1.5.3 Генерация кода

Программное обеспечение

2.1 Программа NET_Redactor

2.1.1 Внешний вид. Меню

После запуска программы перед вами появляется главное окно программы, которое выглядит следующим образом.

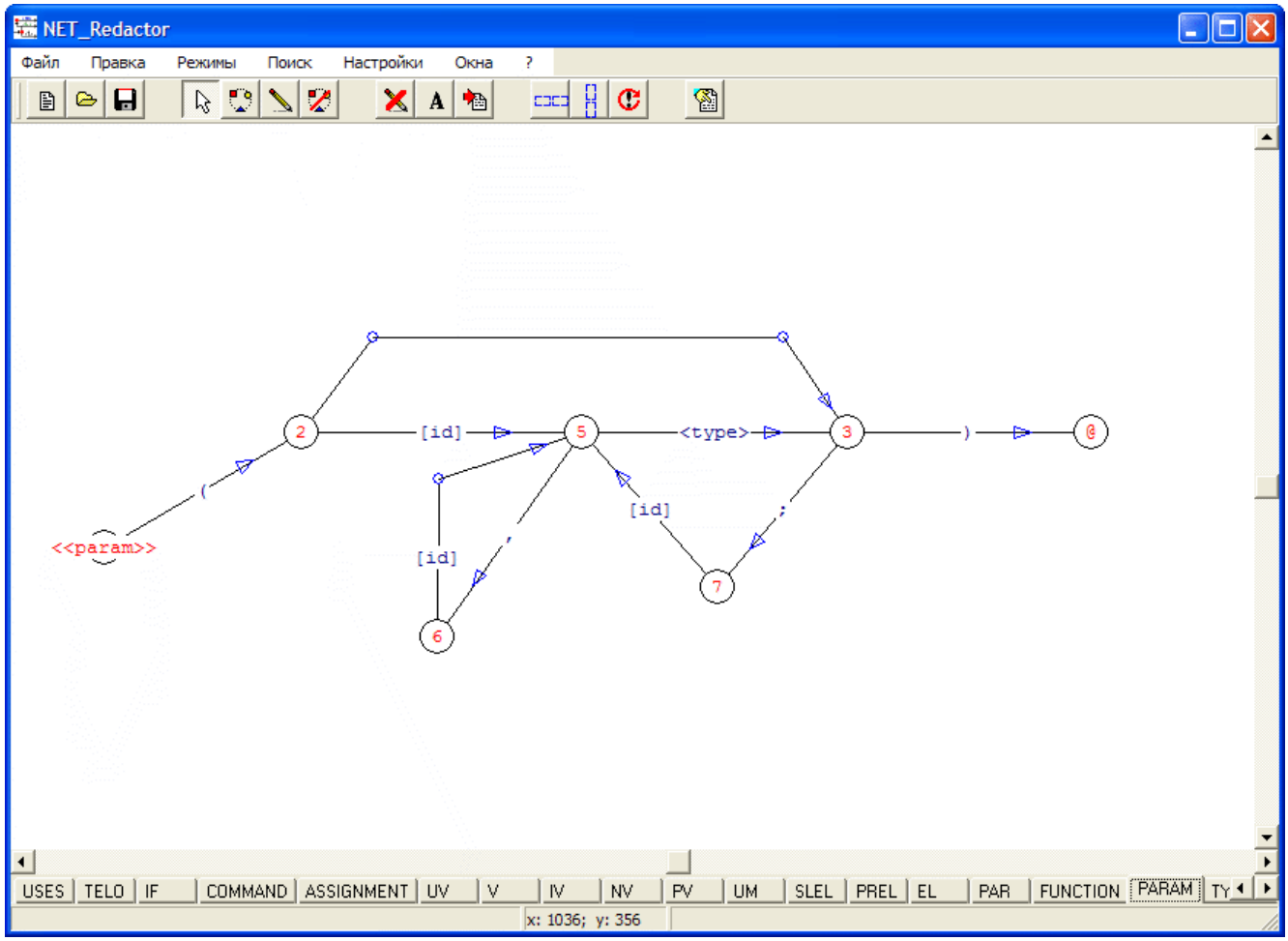


Рис. 5. Главное окно программы NET_Redactor

Скажу сразу, что все пункты меню не будут здесь представлены, так как они уже описаны в справочной системе к данной программе, мы рассмотрим только основные из них.

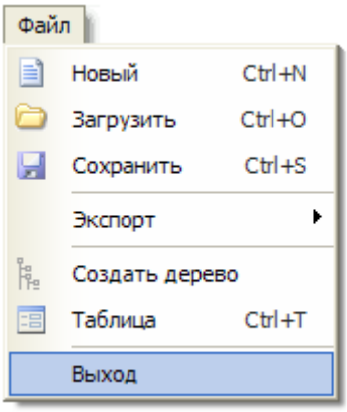


Рис. 6. Меню "Файл" программы NET_Redactor

Меню «Файл»

- «Новый» - Создать новую сеть
- «Открыть» - Открыть новую сеть
- «Сохранить» - Сохранить сеть. Если сеть не была ни разу сохранен, то откроется окно, в котором необходимо выбрать путь и имя сохраняемого файла. Иначе она будет сохранен с текущим именем.
- «Экспорт» - Эта возможность сохранить сеть в формат wmf и jpg, а также сохранить целый набор файлов в отдельную директорию.
- «Создать дерево» - Происходит создание дерева по имеющемуся набору сетей из текущей директории.
- «Таблица» - Открывается окно по созданию таблиц.
- «Выход» - завершение работы программы.

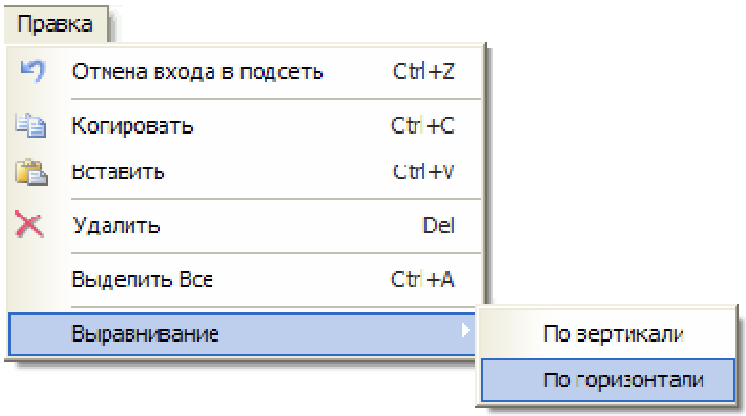


Рис. 7. Меню "Правка" программы NET_Redactor

Меню «Правка»

- «Отмена входа в подсеть» - В NET_Redactore существует возможность не загружать сети каждый раз как только вы хотите посмотреть одну из них. Достаточно стоит выбрать соответствующий режим и щелкнуть на дугу. Программа сама откроет нужную сеть и что бы вернуться обратно, не загружаю предыдущую достаточно нажать этот пункт меню.
- «Копировать» - Запоминается в выделенная группа объектов.
- «Вставить» - Вставляется ранее заполненная группа объектов.
- «Удалить» - Удаляется выбранная группа объектов.
- «Выделить все» - Выделяется все объекты
- «Выравнивание по вертикали/ по горизонтали» - Если выбрать несколько объектов и нажать на этот пункт то их координаты по x / y выровняются.

2.1.2 Основные возможности

Так же как и в предыдущем пункте, я не буду указывать абсолютно все возможности, а только перечислю основные из них:

Для начала посмотрим, какими свойствами обладает дуга:

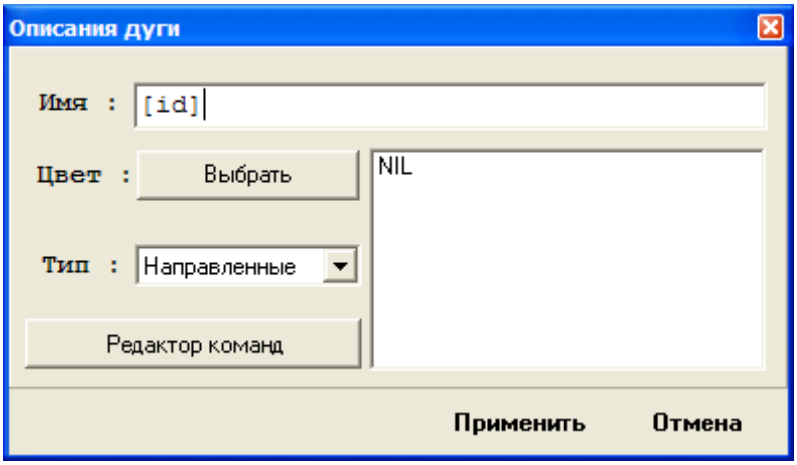


Рис. 8. Свойства дуг

Каждой дуге можно поставить ее имя – в данном случае это есть конструкция языка (ключевое слово). Для красоты можно выбрать её цвет. Также указывается и тип дуги, она может быть как направленная так и не направленная. Справа приведен список команд, которые будут исполняться, когда будет обработанная эта дуга.

Далее перейдем к вершинам:

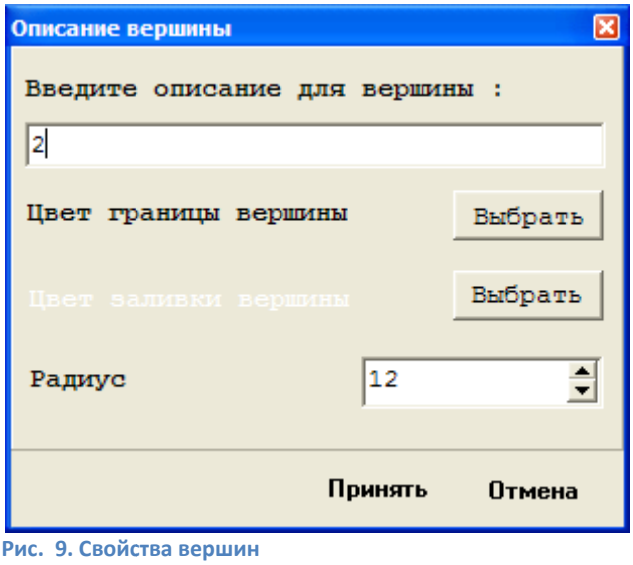


Рис. 9. Свойства вершин

Каждая вершина, практически не несет ни какой нагрузке в языке, поэтому тут задавать нечего, ну разве, что кроме красоты, а именно заданием размера вершины и её раскраски.

2.1.3 Построение таблицы и дерева

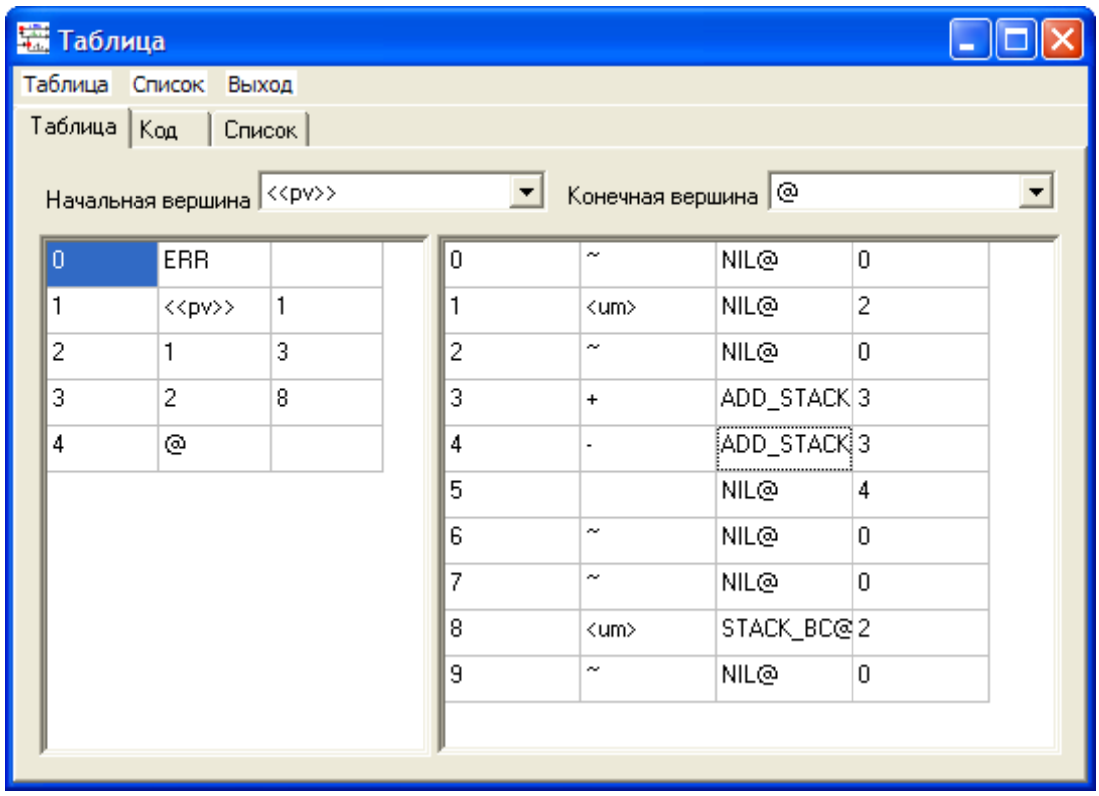


Рис. 10. Построение таблицы

Для того что бы построить таблицу по сети, необходимо выбрать пункт меню ***Файл -> Таблица*** . После чего вы увидите окно программы, где создается таблица (Рис. 10). Затем необходимо выбрать начальную и конечную вершину. И нажать ***Таблица -> Создать***.

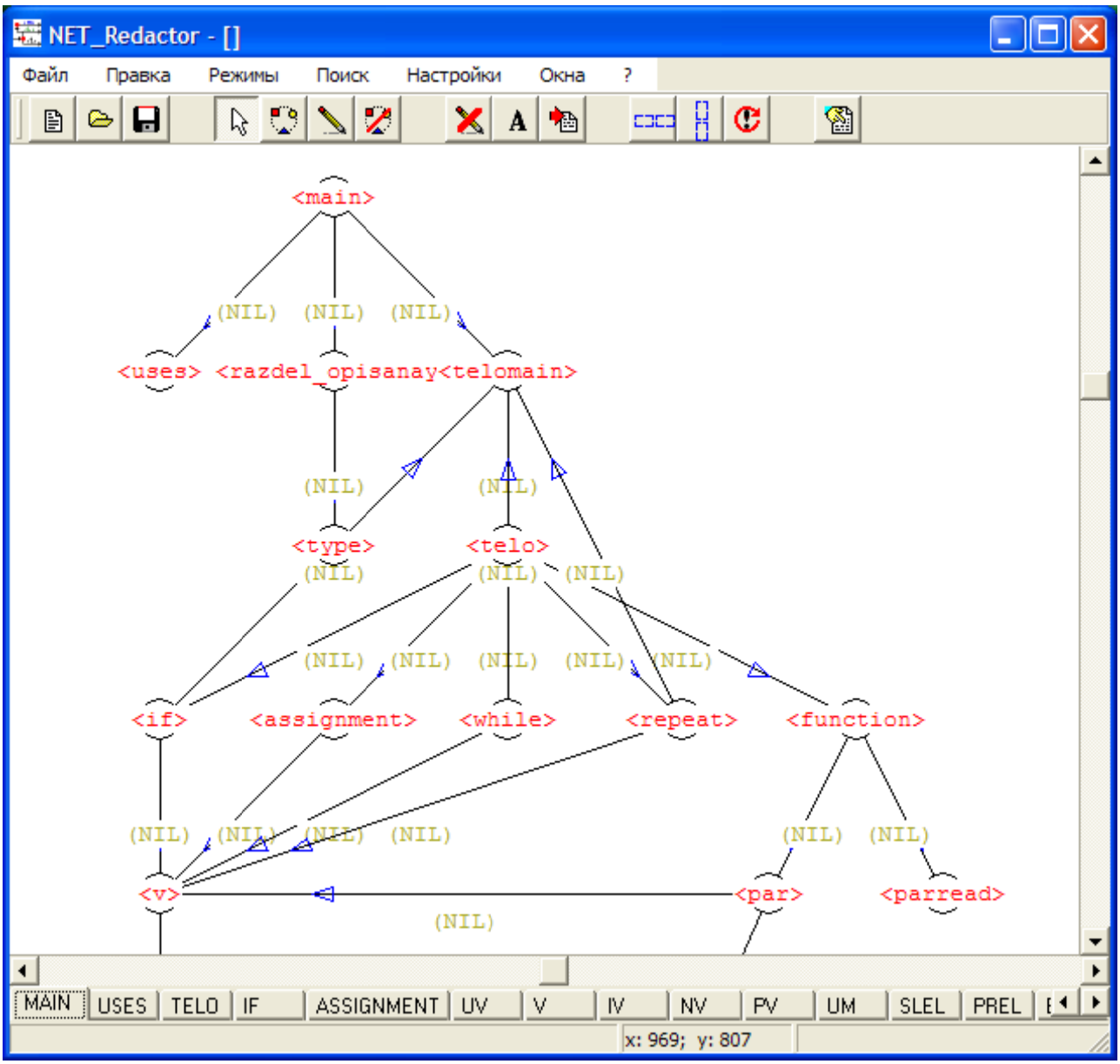


Рис. 11. Построение дерева

Для того что бы построить дерево по сетям, необходимо выбрать пункт меню ***Файл -> Создать дерево***. После чего вы увидите окно программы, где создается таблица (Рис. 11). И перед вами сразу создаться дерево отношений. Это дерево показывает, какая сеть какую вызывает. Иногда это бывает полезно, для того что бы посмотреть соотношения между сетями, а иногда это помогает найти ошибку.

2.1.4 Построение описания

Для построения описания необходимо, что бы как минимум все сети были определены и не содержали ошибок. Далее необходимо открыть ***Файл -> Таблица*** и затем перейти на вкладку «Список» (Рис. 12). Затем нажать кнопку добавить. В результате вас попросят выбрать набор файлов, из которых вы собираетесь описание. Ну вот мы и выбрали. Далее жмем кнопку выполнить, а когда счетчик дойдет до 100%, то сохраняем все файлы с помощью кнопки «Сохранить». Все, описание готово !

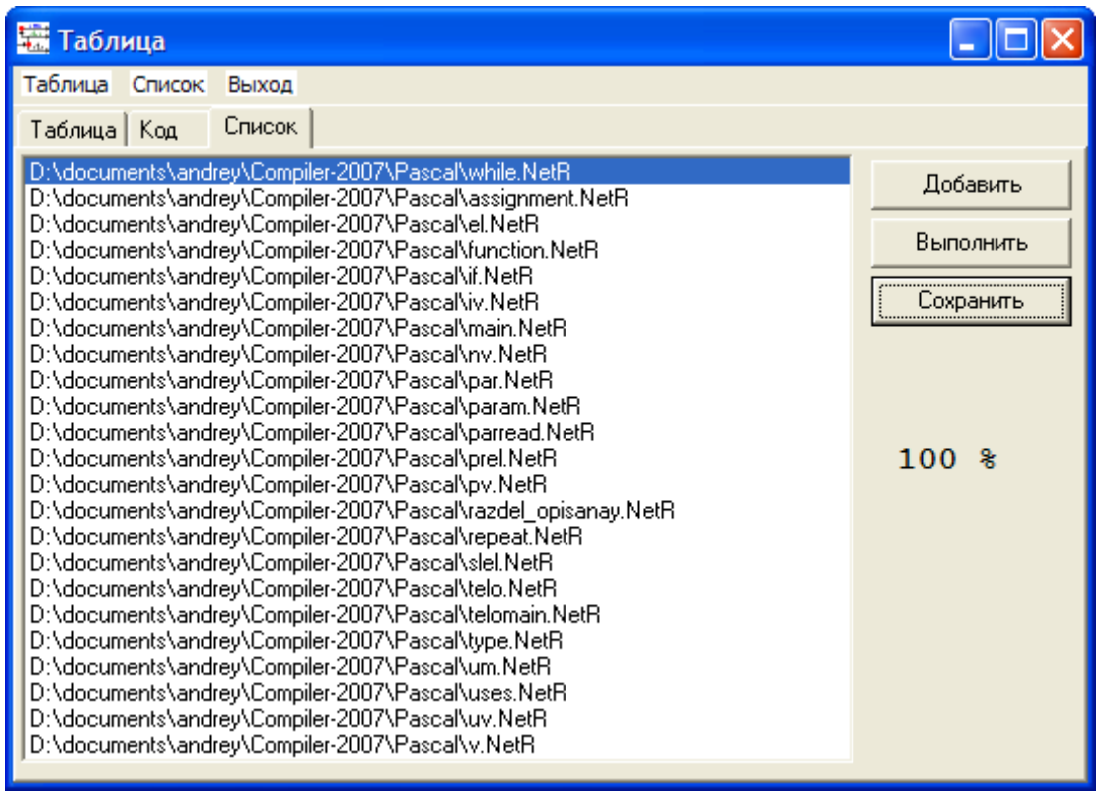


Рис. 12. Создание описания для компилятора

2.2 Программа .NET Compiler

2.2.1 Внешний вид. Меню

После запуска приложения вы сразу же уведите главное окно программы (Рис. 13).

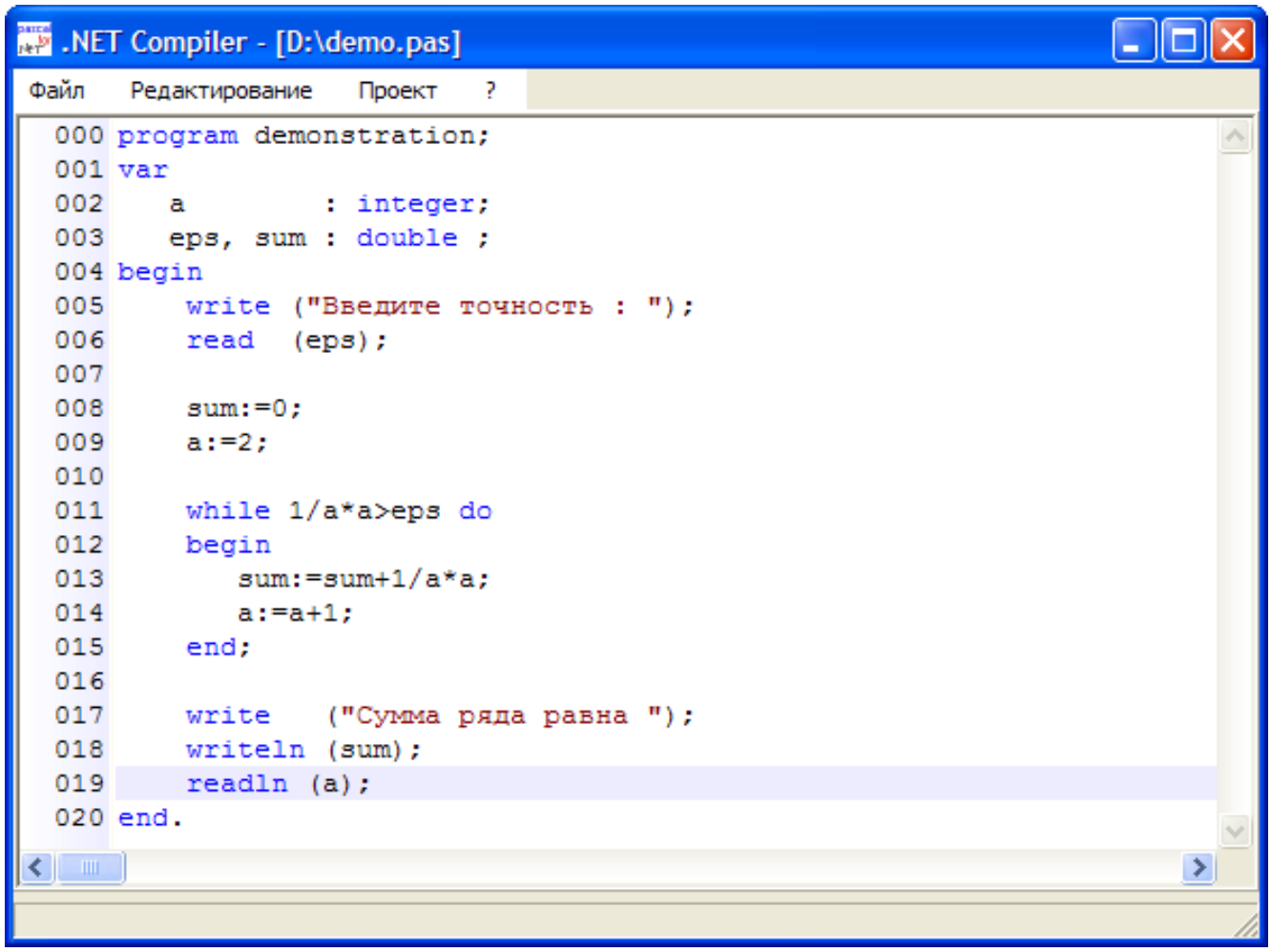


Рис. 13. Главное окно программы .NET Compiler

.NET Compiler – это простой текстовый редактор, который имеет свои преимущества. Преимущества на лицо, точнее на главном окне программы, а именно это подсветка синтаксиса. Эта динамическая. Что это значит? А значит это то что если вы сменили компилятор, например, создали свой язык, то и текстовый редактор будет подсвечивать ключевые слова вашего компилятора. Как это сделано? Очень просто. Когда генерируется описание, программа NET_Redactor генерирует еще один файл с ключевыми словами, называемый ***syntaxis.dll***. Именно его необходимо положить в папку, с названием вашего языка и в файле ***options.dat***, который лежит вместе с ***exe*** данного приложения, указать имя папки, где лежит файл ***syntaxis.dll.*** Вот и все хитрости! Мелочи, а приятно. Наверно никогда не было так просто писать свой текстовый редактор с подсветкой синтаксиса именно вашего языка.

А теперь приступим к изучению меню.

Меню «Файл»

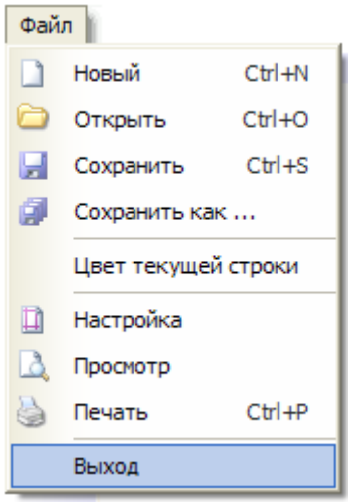


Рис. 14. Меню "Файл"

- «Новый» - Создать новый документ
- «Открыть» - Открыть новый документ
- «Сохранить» - Сохранить документ. Если документ не был ни разу не сохранен, то откроется окно, в котором необходимо выбрать путь и имя сохраняемого файла. Иначе он будет сохранен с текущим именем.
- «Сохранить как …» - Какое бы то ни было название у документа, вам предложат ввести новое имя для сохранения файла.
- «Цвет текущей строки» - Чтобы было удобно видеть, с какой строкой мы сейчас работаем, можно выбрать цвет подсветки. Чтобы этого цвета не было достаточно выбрать его белым.
- «Настройка» - Окно с настройками для предварительного просмотра и печати.
- «Просмотр» - Предварительный просмотр документа. Так он будет выглядеть после того как вы его напечатаете на принтере.
- «Печать» - печать документа.
- «Выход» - завершение работы программы.

Меню «Редактирование»

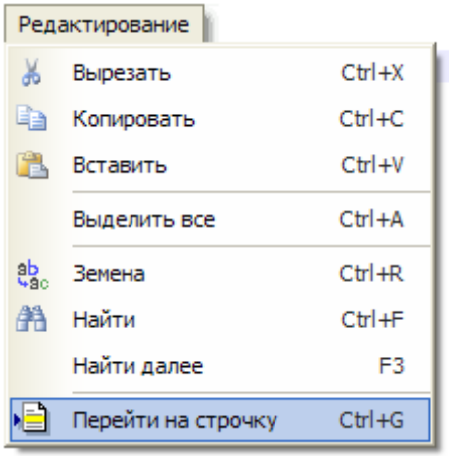


Рис. 15. Меню "Редактирование"

- «Вырезать» - удаляет текущий фрагмент текста и запоминает его
- «Копировать» - запоминает текущий фрагмент текста
- «Вставить» - вставляет фрагмент текста из памяти в документ
- «Выделить все» - выделяет все текст документа
- «Замена» - происходит замена одной сроки другой
- «Поиск» - возможен поиск по документу. Открываете это окно. Вводите слово и нажимаете найти. Документ переходи в то место, где этот текст встретился
- «Найти далее» - Для того что бы найти следующие вхождение теста, необходимо нажать на этот пункт меню
- «Перейти на строчку» - вводите номер строки, и вас переносит на ту строчку, на которую вы захотели.

Меню «Проект»

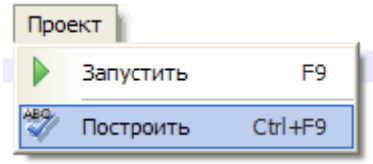


Рис. 16. Меню "Проект"

- «Запустить» - текущий документ будет запущен на выполнение
- «Построить» - текущий документ будет откомпилирован

Меню «?»

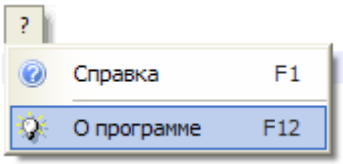


Рис. 17. Меню "?"

- «Справка» - вызов окна справочной системы
- «О программе» - в этом окне вы сможете прочитать информацию о версии данной программы и другую информацию.

2.2.2 Компилирование и запуск приложения

После того как вы открыли документ и написали код можно его скомпилировать. Для этого необходимо нажать соответствующий пункт меню. ***Проект -> Построить.*** Готово! Теперь приложение построено и готово к выполнению. Теперь достаточно нажать ***Проект -> Выполнить*** и приложение запуститься.

2.3 Шаблонный компилятор

2.3.1 Создание компилятора

Компилятор создается просто. Особенно когда у нас уже есть описание. Мы просто открывает проект компилятора в Microsoft Visual Studio 2005 и нажимает кнопку Rebuilt. Все новый компилятор готов. Не правда ли очень просто.

2.3.2 Компилирование файла

Компилятор поддерживает командную строку. Чтобы откомпилировать файл необходимо указать правильные параметры запуска.

```
compiler.exe    name.pas
```

И все! Мы получим приложение name.exe, которое готово к выполнению.

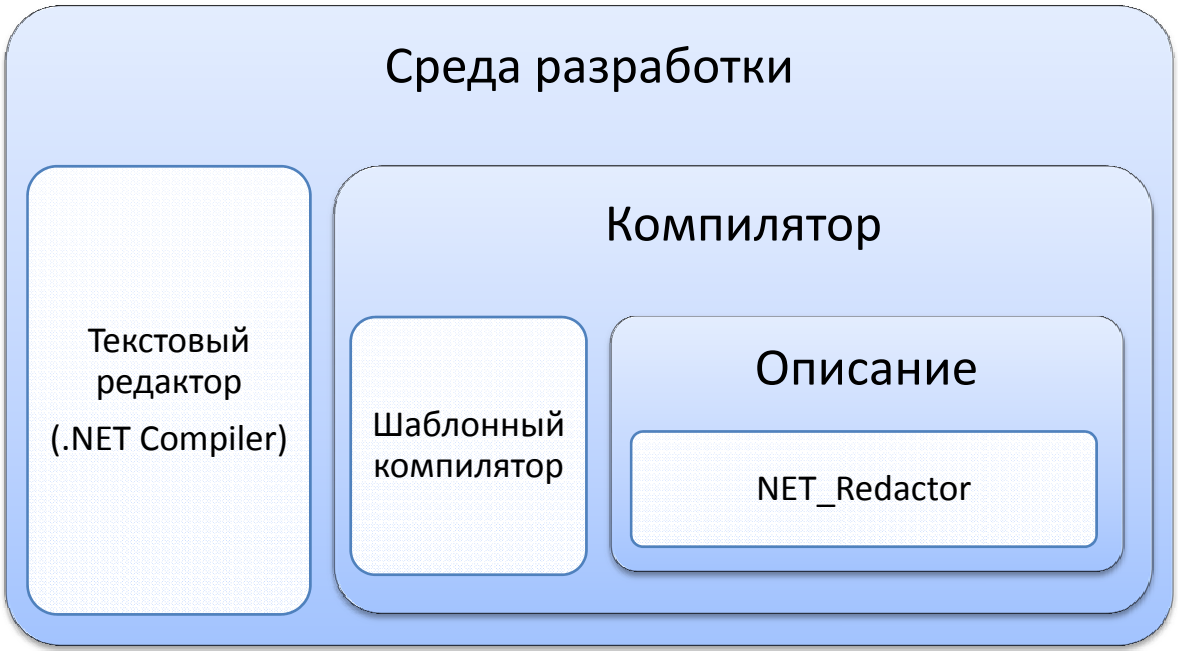
2.3.3 Параметры компилятора для отладки

Когда пишется компилятор, хотим мы этого или нет, возникают ошибки. Эти ошибки очень трудно найти. Поэтому в компилятор встроен один параметр который поможет отладить их. Достаточно написать так:

```
compiler.exe    name.pas    -debug
```

Тогда на выходе программы сформируется файл, который содержит полную информацию о том как компилятор обрабатывал этот документ. Информации очень много, но её полностью достаточно, что бы исключить практически все ошибки. Так, например, для кода из 14 строк сформируется отчет на 1162 строчки, что примерно равно 27кб. Многовато, но много не мало. Большинство ошибок возникает при рисование и проектирование сетей. Именно на локализацию этих ошибок и призвана данная опция.

2.4 Взаимодействие



Теперь подведем итог и посмотрим, как это все работает.

У нас есть программа NET_Redactor, с помощью которой мы рисуем сети, расставляя грамматику, синтаксис и семантику языка. Затем получаем описание. Комбинируя это описание с шаблонным компилятором, мы получаем полноценный компилятор. Вдобавок с текстовым редактором, который специально заточен для компилятора. У вас получается полноценная среда для разработок. Итого Visual Studio на колесах.

Заключение

В данной работе мы проанализировали и разобрали процесс компиляции. Вникли во все его тонкости и познали истину, а истина заключается вот в чем: процесс написание компилятора, очень и очень сложен по разработке алгоритмов и трудоемкости написания кода, я уже не говорю про стадию тестирования … ведь тестирование это главное. Если мы возьмем простую задачу на пример сложить два числа, то вроде бы просто. Но представим теперь, что мы допустили ошибку в программе и при тестировании мы её нашли. Все бы хорошо, если бы одно но. Как теперь показать, что после того как мы исправили эту ошибку, мне не привнесли другую или сказать, что после исправления этой ошибки мы исправили абсолютно все. Очень трудно. Но задача простая, и так как числа конечны (числа в компьютера конечны), то можем написать тест который проверяет, а правда ли наша программа выдает, то, что нужно. Правда, если мы не сделаем тех же ошибок второй раз … то в этом легко убедится. Да простая задача. Но вернемся к нашей. Как доказать, что наш компилятор который порождает бесконечное число программ работает правильно. Это, проблема. Именно поэтому компиляторы пишутся не 2-3 месяца и даже не год, а 5 и более. Но все таки отчаиваться не стоит. Компиляторы писали, пишут, и будут писать, так как без них никуда.

Что же касается проделанной работы, то можно сказать, что проделано очень много. И Все примеры разобранные и закодированные могут послужить достойным примером в учебном процессе для написание полноценного, качественного и многофункционального компилятора. Много сделано для задания лексики, синтаксиса, семантики. В частности разработана программа с помощью, которой можно легко задавать все эти качества языка. При помощи специализированного текстового редактора можно легко и просто писать новые программы. Все это есть учебное пособия типа «собери свой компилятор за 21 день». Конечно звучит смешно, но в скором будущем это возможно и будет реальностью, ведь уже сделано самое главное – сделан первый шаг.

Благодарности

Автор выражает благодарность Лялину С. С. и Митину Р. О. за консультацию и помощь во время выполнения данной работы, а также выражает благодарность своему научному руководителю Курылеву А. Л. за большой вклад в работу.

Список использованной литературы

1. Lidin Serge. *Inside Microsoft .NET IL Assembler*, 2002 - 407c.
2. *Standard ECMA - 335. 4^{th} Edition / June 2006. Common Language Infrastructure (CLI)*
3. Методическое пособие из курса «Исследовательский компилятор» - *Практикум «Оптимизирующие компиляторы» (на примере GCC)*. Авторы: Рагозин Д. В., Галкин С. В., Лялин С. С., Митин Р. О.
4. Рагозин Д. В., Курс «Введение в компиляцию» http://wl.unn.ru/~ragozin/compiler/compil/qwest.htm
5. В. Альфред Ахо, Рави Сети, Д. Джефри Ульман. *Компиляторы: принципы, технологии и инструменты.*: Пер. с англ. - М.: Издательский дом 'Вильямс', 2003. - 768с.: - Парал. тит. англ.
6. Дж. Рихтер. *Программирование на платформе Microsoft .NET Framework.* Мастер-класс. / Пер. с англ. - 3-е изд. - М.: Издательско-торговый дом 'Русская Редакция'; СПб.: Питер, 2005. 512с.: ил.
7. Т.Кормен, Ч.Лейзерсон, Р.Ривест, К.Штайн - *Алгоритмы. Построение и Анализ*, 2-е издание.: Пер. с англ.-М.: Издательский дом "Вильям", 2007.-1296с.: ил.-Парал. тит. англ.
8. Эквин Дональд Кнут. *Искусство программирование, том 1. Основные алгоритмы*, 3-е изд.: Пер. с англ. -М.: Издательский дом 'Вильямс', 2005. - 720 с.: - Парал. тит. англ.
9. Р.Лафоре - *Объектно-ориентированное программирование в C++.* Классика Computer Science. 4-е изд.-СПб.: Питер, 2005.-924с.: ил.
10. Б.Страуструп - *Язык программирования C++. Специальное издание* / Пер. с англ. - М.: ООО "Бином - Пресс":, 2006.-1104с.: ил.
11. Т.Арчер, Э.Уайтчепел - *Visual C++ .NET. Библиотека пользователя.*: Пер. с англ. - М.: Издательский дом "Вильямс", 2005. - 1216с.: ил.-Парал. тит. англ.
12. Microsoft Corporation *Основы Microsoft Visual Studio 2003* / пер. с англ. - М.: Издательско-торговый дом 'Русская Редакция', 2003, - 464с.: ил.
13. Б. Эллиот Коффман. *Turbo Pascal*, 5-е издание.: Пер. с. англ. - М.: Издательский дом 'Вильямс', 2002. - 896с.: ил. - Парал. тит. англ.
14. Р.Стивенс*. Delphi. Готовые алгоритмы*; Пер. с англ. Мерещука П.А. - 2-е изд., стер. - М.: ДМК Пресс; СПб.: Питер,2004. - 384с.: ил.
15. С.Бобровский. *Delphi 5 : Учебный курс.* - СПб.: Питер, 2002. - 640с.: ил.

Алфавитный указатель