МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

ПРАКТИКУМ

по теории языков программирования и методам трансляции

Учебно-методическое пособие

Часть 2. Лексический и синтаксический анализ языков программирования

2017 г.

УДК 681.3.06
П 56

Практикум по теории языков программирования и методам трансляции. Учебно-методическое пособие. Часть 2. Лексический и синтаксический анализ языков программирования / Автор Вл. Пономарев. Озерск: ОТИ НИЯУ МИФИ, 2017. — 100 с.

В пособии подробно излагается, как выполнять практические работы по дисциплине «Теория языков программирования и методы трансляции». Работы включают в себя проектирование лексического и синтаксического анализаторов, семантический анализ, проектирование трансляционной грамматики, формирование внутреннего представления программы в виде ПОЛИЗ.

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1.
2. Зав. кафедрой прикладной математики ОТИ НИЯУ МИФИ,
к.ф.-м.н. Акопян Р.Р.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий

Цель практических занятий — изучение лексического, синтаксического и семантического анализа языков программирования, а также общих принципов построения трансляторов. Практические занятия преследуют следующие конкретные цели:

- конструирование синтаксической грамматики;
- конструирование лексического анализатора;
- формирование потока токенов и запись его в файл;
- синтаксический анализ методом рекурсивного спуска;
- синтаксический анализ на основе грамматики класса LL(1);
- синтаксический анализ на основе грамматики класса LR(1);
- построение таблицы символов;
- семантический анализ;
- конструирование трансляционной грамматики;
- формирование внутреннего представления в виде ПОЛИЗ;
- запись ленты ПОЛИЗ во внешний файл;
- конструирование исполняющей машины ПОЛИЗ.

В работах должны быть разработаны модули, классы и алгоритмы:

- модуль констант;
- модуль синтаксической таблицы;
- класс лексического анализатора;
- класс синтаксического анализатора;
- синтаксический анализатор рекурсивного спуска;
- синтаксический анализатор LL(1);
- синтаксический анализатор LR(1);

Конечной целью является интерпретация входного текста на основе трансляционной грамматики класса LL(1).

1. Работа 1. «Грамматика»

Цель: разработка синтаксической грамматики для заданного языка.

Перед построением грамматики нужно определить спецификацию языка, который будет использоваться как входной язык.

Входной язык обычно строится на основе одного из существующих распространенных языков программирования, таких как С, Бейсик, Паскаль.

Чтобы разработать лексический анализатор, нужно знать, какие лексемы он должен распознавать и какие токены формировать. Поэтому первая задача заключается в том, чтобы разработать некоторый достаточно простой язык программирования и описать его синтаксис с помощью грамматики.

В качестве прообраза языка используется один из широко распространенных языков программирования. На основе синтаксиса одного из этих языков предлагается разработать свой собственный упрощенный язык, а на его основе — интерпретатор с этого языка, позволяющий выполнить простые вычисления.

На разрабатываемый язык накладываются жесткие ограничения с тем, чтобы реализация транслятора была достаточно простой.

Первое важное ограничение — языки не содержат функций (процедур). Это в значительной мере снижает трудоемкость разработки.

Второе ограничение снижает количество типов до двух, например, целый и вещественный типы, целый и массивы целых, строки и целые числа и т.п.

Третье ограничение снижает количество операций до 8-10.

Четвертое ограничение снижает количество операторов языка до 3-5.

Наконец, количество модулей программы на заданном языке — один.

Для разработки грамматики используется программа ReVoL SYNAX.

Разработку грамматики начинаем с описания общего вида модуля.

Программа на заданном языке - это один модуль, содержащий весь текст. Поэтому целевой символ грамматики — это «модуль» (module).

1.1. Разработка начальной части грамматики

Далее рассуждаем, из чего состоит модуль. Поскольку функций в языке нет, то модуль состоит из операторов. Следовательно, нам нужны понятия «оператор» (statement) и «множество операторов». Второе понятие будем называть «операторы» (statements).

Операторы можно поделить на две категории:

- оператор объявления (declaration statement)
- управляющий оператор (flow control statement)

Это деление может иметь смысл, а может и не иметь смысла. Деление имеет смысл тогда, когда синтаксис языка предусматривает размещение

всех операторов объявлений в начале модуля. В этом случае потребуются понятия «объявление» (declaration) и «объявления» (declarations).

Предположим, язык задает размещение операторов объявлений в начале модуля.

Тогда начинать разработку грамматики нужно следующим образом:

```
#0 <M>
#1 <M>=<DS><SS>
#2 <DS>=[d]
#3 <SS>=[s]
```

Здесь приведена грамматика в формате программы SYNAX. Нетерминалы заключены в угловые скобки, терминалы заключены в квадратные скобки, вместо стрелки используется знак равенства, пустая строка обозначается точкой. В программе SYNAX первая строка (помеченная номером 0) всегда должна содержать только целевой символ.

В целях ускорения разработки грамматики нетерминалы будем обозначать как можно более короткими названиями, происходящих от соответствующих английских слов. По завершении разработки мы сможет легко заменить названия более осмысленными. Сейчас, надеюсь, понятно, что M = module, DS = declarations, SS = statements.

При разработке грамматики в программе SYNAX следует придерживаться правила размещения правил грамматики: никогда не описывайте нетерминал раньше, чем он появился в правой части какого-либо предыдущего правила, иначе анализ грамматики может оказаться неуспешным.

Как видим, грамматика описывает целевой символ «модуль» как состоящий из объявлений и операторов управления, следующих в заданном порядке, сначала объявления, потом другие операторы.

Заметим, что символы DS и SS описываются как терминалы "d" и "s", не имеющие никакого смысла, и внедренные с целью сделать грамматику правильной. Эти терминалы являются «заглушками» нетерминалов, описать которые мы сейчас не хотим или не можем. Заглушки полезны при разработке отдельных частей грамматики.

Сейчас грамматика корректна и мы можем даже попытаться построить анализатор для нее, например, анализатор LL(1). Для этого сохраните грамматику под заданным именем, выберите в меню «Анализ», выберите вкладку «LL(1)», снова выберите в меню «Анализ».

Двигаемся дальше. Описываем символ «DS». Он представляет собой последовательность отдельных операторов объявления, в том числе ни одного. Поэтому можно придумать следующие правила:

```
<M>
<M>=<DS><SS>
<DS>=.
<DS>=<D>
<DS>=<DS><D>
<D>=[d]
<SS>=[s]
```

Здесь видно, что «объявления» могут быть пустыми (отсутствующими), одним объявлением, или уже имеющимися объявлениями, к которым приписано одно объявление. Само объявление "D" представлено заглушкой "s". Эта грамматика корректна. Например, можно построить следующие выводы:

```
M ⇒ DS SS ⇒ SS ⇒ b
M ⇒ DS SS ⇒ D SS ⇒ a b
M ⇒ DS SS ⇒ DS D SS ⇒ DS D SS ⇒* a a b
```

К сожалению, в этой грамматике есть левая рекурсия, поэтому она не годится для нисходящего разбора. Вместо того, чтобы устранять левую рекурсию, перепишем правила, заменив левую рекурсию правой:

```
<M>
<M>=<DS><SS>
<DS>=.
<DS>=<D><DS>
<D>=[d]
<SS>=[s]
```

Эта грамматика лучше, потому что она является LL(1), в чем легко убедиться с помощью программы SYNAX.

Аналогично можно описать и операторы управления, тогда получим, например:

```
<M>
<M>=<DS><SS>
<DS>=<D><DS>
<DS>=.
<D>=[d]
<SS>=<S><SS>
<SS>=.
<S>=[s]
```

Эта грамматика допускает пустые цепочки и является LL(1), что нам и нужно.

Разберем также вариант, когда операторы объявления и управления потоком могут следовать в произвольном порядке. В этом случае нет необходимости выделять понятие «объявления» и мы просто сокращаем предыдущую грамматику:

```
<M>
<M>=.
<M>=<S><M>
<S>=[s]
```

В первом случае мы описываем нетерминал D (объявление), а во втором случае описание этого нетерминала является частью описания нетерминала S (оператор).

1.2. Разбор операторов

Двигаясь дальше, мы должны описать операторы объявлений и операторы управления потоком вычислений. Как, например, описывается оператор объявления.

Сначала нужно понять, как он выглядит в заданном языке. Пусть объявление переменной выглядит следующим образом:

```
DIM A AS LONG
```

Так объявляются переменные в языке Basic. Чтобы объявить несколько переменных, используется следующий синтаксис:

```
DIM A AS LONG, B AS LONG
```

Заметим, что тип указывается для каждой переменной. Если мы хотим задать объявление только одной переменной, то можно предложить следующий вариант грамматики:

```
<D>=[Dim][id][As][Long]
```

Если мы хотим описать несколько переменных, нужно ввести символ для списка переменных, нетерминал DL (declaration list), например, так:

```
<D>=[Dim]<DL>
<DL>=[id][As][Long]<DL.>
<DL.>=[,][id][As][Long]<DL.>
<DL.>=.
```

Здесь символ "DL." обозначает продолжение цепочки "DL", которое либо добавляет еще одно объявление, либо завершается. Важно, что добавление этих правил не нарушает класс грамматики LL(1).

Теперь можно перейти к описанию операторов управления. Обязательным оператором является оператор присваивания. В нашем языке он будет обозначаться символом "=". Если не учитывать, что в языке могут быть заданы массивы, то оператор присваивания описывается так:

```
<S>=[id][=]<E>
<E>=[e]
```

Здесь нетерминал E — это обозначение для выражения, которое сейчас спрятано с помощью заглушки «e».

Если в языке заданы массивы, то грамматика может значительно усложниться. Мы не будем разрабатывать языки, в которых можно задавать многомерные массивы, иначе говоря, левой частью оператора присваивания в нашем языке будут только идентификатор переменной или элемент одномерного массива. Попытаемся описать оба этих случая:

```
<S>=[id][=]<E>
<S>=[id][(]<E>[)][=]<E>
<E>=[e]
```

При этом грамматика вышла из класса LL(1), что для нас нежелательно. Это произошло из-за того, что два правила начинаются с одного и того

же префикса id. Но мы помним, что для устранения префиксов есть так называемая левая факторизация.

```
<S>=[id]<SA>
<SA>=[=]<E>
<SA>=[(]<E>[)][=]<E>
<E>=[e]
```

Мы применили ее вручную. Можно было сделать это с помощью программы SYNAX, просто результат может оказаться неожиданным. Это же простое преобразование.

Перейдем к оператору IF, который задан в любом языке. Это оператор, который ведет к неоднозначности грамматики, однако программа SYNAX строит правильную синтаксическую управляющую таблицу, хотя и может сообщать об ошибке построения (зависит от версии программы).

Здесь мы сталкиваемся с особенностями разных языков.

В языке Си, например, оператор IF имеет следующий синтаксис:

```
if (выражение) оператор
if (выражение) оператор else оператор
```

при этом для того, чтобы написать несколько операторов вместо одного, используется разновидность оператора, которую мы назовем «блок».

Блок в Си — это фигурные скобки, внутри которых может располагаться произвольное количество операторов управления, которым предшествует произвольное количество операторов объявлений локальных (для данного блока) переменных. Примерно такая же ситуация наблюдается в языке Pascal.

В языке Basic последовательность операторов никак не обозначается, понятия блока нет, но последовательность при этом ограничивается ключевыми словами языка, такими, как THEN, ELSE, END и другими.

Кроме того, в этом языке имеет значение перенос строки, обозначающий следующий оператор блока операторов. Иначе говоря, каждый оператор в этом языке должен заканчиваться символом LF перевода строки (или аналогичным ему сочетанием символов), либо между операторами последовательности должен вставляться символ двоеточия. Поэтому в языке Basic несколько способов записи оператора IF, например, однострочный и многострочный.

Синтаксис однострочного оператора IF:

```
If Выражение Then Оператор [Else Оператор]
```

Синтаксис многострочного оператора IF:

```
If Выражение Then
    [Операторы]
[Else
    [Операторы]]
End If
```

Поэтому, если мы разрабатываем грамматику для языка, похожего на Basic, мы должны учитывать, что в конце каждого оператора должен быть символ LF:

```
<D>=[Dim]<DL>
<DL>=[id][As][Long]<DL.>
<DL.>=[,][id][As][Long]<DL.>
<DL.>=[LF]
<S>=[LF]
<S>=[id][=]<E>[LF]
<S>=[if]<E>[then][LF]<SS><SI>
<SI>=[end][if][LF]
<SI>=[else][LF]<SS>[end][if][LF]
<E>=[e]
```

Здесь правило `<S>=[LF]` нужно для того, чтобы можно было добавлять пустые строки в программный текст.

Еще один пример — как в Basic-подобном языке построить правило для оператора цикла WHILE:

```
<S>=[While]<E>[LF]<SS>[Wend][LF]
```

Правила для разбора других операторов разрабатываются аналогичным образом.

1.3. Разбор выражений

Перейдем к построению правил разбора выражений.

Традиционно эта часть грамматики вызывает трудности. Здесь важно понять, что грамматика для разбора выражения всегда строится на основе одной и той же классической грамматики:

```
E = E + T | E – T | T
T = T * P | T / P | P
P = id | const | (E)
```

Еще более важным является то, как выстраивается цепочка выражений. Чтобы обеспечить правильный приоритет операций, все операции одного приоритета вычисляются на одном уровне этой грамматики, результат вычисления операций одного уровня служит операндом для операций, уровень которых ниже и которые должны выполняться позднее. Если говорить конкретно о приведенной грамматике, то в ней есть три уровня приоритета, соответствующие трем строчкам, в которых записаны правила.

Самый высокий приоритет имеют операции, расположенные в последней строке. Это простейшие элементы данных, которые поставляют значения, такие как идентификатор или константа (непосредственная запись значения). Из этих значений в конечном итоге вычисляется результат выражения в целом. Мы говорим об операциях на этом уровне потому, что получению значений здесь может предшествовать его вычисление, например, правило P→(E) содержит операцию вычисления выражения.

Кроме того, элемент массива, который также является поставщиком значения, или функция, также требуют вычислительных действий. На этом уровне, например, может выполняться такая операция, как унарный минус. Эти действия при вычислении выражения в целом выполняются в первую очередь. Название нетерминала P происходит от primitive, то есть примитивный элемент данных.

Что происходит на втором уровне приоритета (если считать снизу). Нетерминал T, название которого происходит от term, обозначает последовательность операций умножения и деления, выполняемых над примитивными операндами, которые поставляет самый нижний уровень.

Если перейти на уровень выше (по тексту грамматики, а не по приоритету операций), то на самом верхнем уровне выполняются операции сложения и вычитания, операндами которых являются термы, поставляемые нижележащим уровнем. Иначе говоря, выражение в этой грамматике — это последовательность сложений и вычитаний термов.

Если в вычислении выражений участвуют другие операции, например, операции отношений, логические, побитовые, то они формируют собственный уровень в данной грамматике и располагаются в соответствии с приоритетом операций либо выше (по тексту) либо ниже первой строки грамматики.

Например, в языке Pascal операция AND имеет приоритет выше, чем сложение, но ниже, чем умножение, а операция OR имеет такой же приоритет, что и сложение. Поэтому для этого языка эти две операции внедряются в классическую грамматику следующим образом:

```
E = E + A | E - A | E OR A | A
A = A AND T | T
T = T * P | T / P | P
P = id | const | (E)
```

В соответствии с этим выражение есть последовательность сложений, вычитаний и операций OR над последовательностями операций AND, а операции AND выполняются над термами.

Для применения этой грамматики в нашей работе требуется привести ее к нелеворекурсивному виду, например:

```
<E>
<E>=<T><T.>
<T.>=[+]<T><T.>
<T.>=[-]<T><T.>
<T.>=.
<T>=<P><P.>
<P.>=[*]<P><P.>
<P.>=[/]<P><P.>
<P.>=.
<P>=[id]
<P>=[I4]
<P>=[(]<E>[)]
```

Смысл символов вида "X." здесь тот же, что и раньше — это символ, который является продолжением цепочки (ее наращиванием) и может вырождаться в пустую цепочку.

Заметим, что это отдельная грамматика, поэтому в начале стоит целевой символ. По окончании разработки эту грамматику следует добавить к основной грамматике (без первой строки), заменив правило с заглушкой.

Рассмотрим сначала символ "P", поставляющий элементы данных. Как видно, элементами данных в этой грамматике могут быть идентификаторы (представляющие переменные программы), константы целого типа, обозначенные "I4" и выражения в скобках, которые должны вычисляться сначала.

В этом месте грамматики могут быть и другие элементы данных. Например, если в языке заданы массивы (как говорилось, только одномерные), то элемент массива также является примитивным элементом данных, поэтому его нужно добавить в грамматику:

```
<P>=[id]
<P>=[id][(]<E>[)]
<P>=[I4]
<P>=[(]<E>[)]
```

Как только мы сделаем это, грамматика выйдет из класса LL(1) из-за одинаковых префиксов для символа "P", поэтому нужно применить левую факторизацию:

```
<P>=[id]<PP>
<PP>=.
<PP>=[(]<E>[)]
<P>=[I4]
<P>=[(]<E>[)]
```

Это не единственное изменение в этой части разбора выражения.

Другими элементами данных являются строковые литералы, вызовы функций или обращения к методам объектов. Обращений к методам объектов по всей вероятности у нас не будет, но вызовы функций допустимы. Как было сказано, язык не предполагает описание функций, но встроенные в язык функции вполне могут быть заданы.

Встроенные в язык функции заранее заносятся в таблицу символов. Нам нужно решить, как эти функции будут анализироваться. Можно предложить два варианта.

Первый вариант — если при разборе символа "P" встречается идентификатор, то по таблице символов определяется, что это. Второй вариант — название функции является ключевым словом и тогда оно прямо указывается в грамматике, отличая его от переменных.

Разберем, как задать правила для первого варианта. Вызов функции в языке Basic формируется по-разному в зависимости от числа параметров. Если у функции параметров нет, то вызов — это просто идентификатор (без скобок, как в других языках).

Если параметры есть, то они перечисляются в круглых скобках через запятую. Очевидно, вызов функции по синтаксису в точности совпадает с вычислением элемента массива. Если мы договорились, что массивы в языке только одномерные, то количество выражений в скобках для вычисления элемента массива должно быть равно одному и это легко описывается так, как показано выше. Однако для функций количество параметров может быть разным, поэтому грамматика должна быть изменена с тем, чтобы вместо одного выражения в скобках можно было описать несколько:

```
<P>=[id]<PP>
<PP>=.
<PP>=[(]<E><E.>[)]
<E.>=[,]<E><E.>
<E.>=.
<P>=[I4]
<P>=[(]<E>[)]
```

Теперь в скобках может следовать произвольное количество выражений, разделенных запятыми, но не менее одного. Если у функции нет параметров, она анализируется просто как идентификатор.

При втором варианте, когда название функции является ключевым словом, каждая функция должна быть записана в правилах. Для примера введем в язык две функции для вычисления максимума и минимума, "`_max`" и "`_min`" соответственно, у каждой из функций два параметра. Тогда мы можем записать их в грамматику следующим образом:

```
<P>=[id]<PP.>
<PP.>=.
<PP.>=[(]<E>[)]
<P>=[I4]
<P>=[(]<E>[)]
<P>=[_max][(]<E>[,]<E>[)]
<P>=[_min][(]<E>[,]<E>[)]
```

Грамматика при этом остается в классе LL(1). Второй вариант лучше тем, что на долю семантического анализа здесь выпадает меньше работы (в будущем). Однако в этом случае никакой другой идентификатор не может иметь название, совпадающее с названием встроенной функции. Именно поэтому для функций взяты такие необычные названия.

На этом, самом высоком уровне приоритета, могут быть также правила для вычисления унарных операций, например, унарного минуса. Первый вариант может быть таким:

```
<P>=[-]<P>
<P>=[id]
<P>=[I4]
<P>=[(]<E>[)]
```

Второй вариант вводит еще один символ, обозначенный "U":

```
<T>=<U><U.>
<U.>=[*]<U><U.>
<U.>=[/]<U><U.>
<U.>=.
<U>=<P>
<U>=[-]<P>
<U>=[not]<P>
<P>=[id]
<P>=...
```

Здесь также включена операция NOT, которая для языка Pascal имеет высокий приоритет и должна находится на уровне символа "P".

Рассмотрим теперь другие уровни. Нас интересует, как включить в грамматику вычисление операций отношений и логических операций.

Для определенности будем считать, что в языке заданы операции:

- отношений: "**=**", "**<>**", "**<**", "**>**"
- логические: "**NOT**", "**AND**", "**OR**"

Если язык подобен языку Basic, то приоритеты операций следующие (от самого низкого к самому высокому, информация взята из MSDN):

```
OR
AND
NOT
= <> < >
+ -
* /
```

В соответствии с этим можно построить следующую грамматику для выражения:

```
<E>
<E>=<A><A.>
<A.>=[or]<A><A.>
<A.>=.
<A>=<N><N.>
<N.>=[and]<N><N.>
<N.>=.
<N>=[not]<N>
<N>=<R><R.>
<R.>=[=]<R><R.>
<R.>=[<>]<R><R.>
<R.>=[<]<R><R.>
<R.>=[>]<R><R.>
<R.>=.
<R>=<T><T.>
<T.>=[+]<T><T.>
<T.>=[-]<T><T.>
<T.>=.
<T>=<P><P.>
<P.>=[*]<P><P.>
<P.>=[/]<P><P.>
<P.>=.
<P>=[id]
<P>=[I4]
<P>=[(]<E>[)]
```

Как видим, выражение есть последовательность операций OR над операциями AND, операции AND есть последовательность отрицаний NOT и операций отношений, операции отношений есть последовательность отношений между мультипликативными операциями и так далее.

Остается соединить эту грамматику с грамматикой, описывающей модуль, и убедиться, что грамматика не вышла из класса LL(1).

1.4. Грамматика для языка вида Pascal

Если базовый язык похож на язык Pascal, то грамматика будет отличаться от грамматик для других языков, поскольку в языке Pascal явно задается начало и конец программного текста, и объявления переменных явно отделены от текста. Чтобы лучше понять, что мы должны описать, посмотрим на фрагмент текста на языке Pascal.

```
var i,
  j: integer;
var
  m: array[1..10] of integer;
  k: integer;
begin
  for i:=1 to 9 do begin
     j:= j + 1;
  end
end.
```

Программа на языке Pascal начинается с раздела объявлений, в котором у нас могут быть только объявления переменных (и массивов, если они заданы), после чего следует обязательный блок BEGIN…END и обязательная «точка» в конце (на самом деле первой может быть необязательная строка "program идентификатор;").

Объявления переменных следуют после ключевого слова VAR, при этом объявлений может быть сколько угодно, каждое объявление задает переменные одного типа, и заканчивается точкой с запятой. Ключевых слов VAR может быть сколь угодно много. В соответствии с этим можно составить примерно следующую начальную грамматику.

```
<M>
<M>=<DS><B>[.]
<DS>=[var]<DL><DL.>
<DS>=.
<DL>=[id]<IL>[:][integer][;]
<IL>=[,][id]<IL>
<IL>=.
<DL.>=<DL><DL.>
<DL.>=.
<B>=[begin]<SS>[end]
<SS>=<S><SS>
<SS>=.
<S>=[;]
<S>=<B>
```

Данная грамматика предписывает использовать ключевое слово VAR только один раз, после него обязательно следует минимум один список объявлений, помеченный символом "DL". Список объявлений содержит минимум один идентификатор, за которым может следовать список идентификаторов "IL", затем двоеточие, тип и точка с запятой.

За объявлениями, если они есть, следует «блок» "B", начинающийся ключевым словом BEGIN и завершающийся ключевым словом END. Внутри блока размещаются операторы управления, в том числе ни одного.

По счастью, для символа "S", описывающего оператор, нет необходимости в заглушке, потому что, во-первых, точка с запятой является допустимым оператором, а во-вторых, «блок» "B" также является допустимым оператором, что и было использовано в данной грамматике.

1.5. Встроенные процедуры

В языке могут быть заданы встроенные процедуры. Тогда вызов этих процедур — это вариант символа «оператор». Пример грамматики:

```
<S>=[id][(]<E><E.>[)]
<E.>=[,]<E><E.>
<E.>=.
```

При этом грамматика выйдет из класса LL и придется применить левую факторизацию, чтобы вернуть ее обратно в класс LL(1). Другой вариант — явно указать вызываемые процедуры.

1.6. Другие типы данных

Как было сказано, в задании должно быть указано минимум два типа данных, которые ваш язык должен поддерживать. Как правило, это разные типы данных, то есть не принадлежащие одной категории типа «целочисленные типы» или «вещественные типы».

Самое простое разделение типов данных — это целочисленные и вещественные типы. При этом типы данных не принадлежат одной категории, но значительно различаются в смысле различных видов литералов, описывающих константы данных типов. Это ведет к формированию в языке различных токенов для описания различных по типу констант.

Нужно договориться об обозначении токенов, описывающих различные литералы. Здесь уже встречалось обозначение [I4] для литерала, описывающего целочисленную константу. Если вы не собираетесь описывать константы, в которых явно указан тип, то для целочисленных констант это единственное обозначение. Если же язык будет определять размер константы, то другими вариантами являются [I1] и [I2].

Если задан язык с вещественными типами данных, то должны быть определены вещественные константы. Для простоты будем полагать, что вещественные константы имеют один из трех видов: ".0", "0." или "0.0". Здесь ноль обозначает целое число. Если язык не задает размер веществен

ной константы, то ее обозначение должно быть [R8], вещественная константа двойной точности. Другое возможное обозначение [R4], вещественная константа одинарной точности.

Если язык определяет строковые литералы (строковые константы), то для ее обозначения следует использовать токен [quote]. Заметим, что символьные константы, если они заданы, не образуют отдельных токенов, поскольку они являются целочисленными константами и ведут к токену [I4] или похожему.

Если язык определяет несколько типов данных, возникает вопрос, как их интегрировать в грамматику. Есть два варианта. Первый — каждый тип вписать соответствующим токеном во всех местах, где он встречается. Например, если есть грамматика для объявления переменной

```
<D>=[Dim]<DL>
<DL>=[id][As][Long]<DL.>
<DL.>=[,][id][As][Long]<DL.>
<DL.>=[LF]
```

то токен `[Long]` нужно продублировать токеном `[Double]` или другим

```
<D>=[Dim]<DL>
<DL>=[id][As][Long]<DL.>
<DL>=[id][As][Double]<DL.>
<DL.>=[,][id][As][Long]<DL.>
<DL.>=[,][id][As][Double]<DL.>
<DL.>=[LF]
```

Грамматика при этом выйдет из класса LL и нужно будет применить левую факторизацию для возвращения ее обратно в класс LL(1).

Второй вариант — использовать символ для типа, например

```
<D>=[Dim]<DL>
<DL>=[id][As]<TY><DL.>
<DL.>=[,][id][As]<TY><DL.>
<DL.>=[LF]
<TY>=[Long]
<TY>=[Double]
```

1.7. Результат

Результат данной работы заключается в формировании списка токенов в виде констант перечисления, которое затем вставляется в проект транслятора и определяет распознаваемые лексемы.

Чтобы получить этот список, по завершении формирования грамматики в программе SYNAX нужно перейти на вкладку «SYMBOLS» и выбрать в меню «Экспорт». Результат будет сохранен в текстовый файл.

1.8. Общие требования к проектам

Для всех языков должен быть задан оператор печати результата print.

Он рассматривается как дополнительное ключевое слово языка и выводит на терминал значение выражения, то есть описывается в грамматике следующим образом:

```
<S>=[print]<E>
```

Для языка типа Basic в конце правила должен быть токен [LF].

Кроме того, в каждом проекте комментарий соответствует комментарию, принятому в языке Си, например

```
/* 123 */
```

Это необходимо для того, чтобы построить конечный автомат для этого вида токена в соответствии с изученным материалом.

1.9. Язык, используемый в данном описании

Для демонстрации процесса разработки в данном пособии используется следующий язык, основанный на языке Basic.

Язык регистро-зависимый: нет. Инициализация переменных в объявлении: нет. Объявления только в начале модуля: нет. Типы данных: long. Константы: I4. Операции: `+`, `-`, `*`, `/`, `=`, `<>` . Операторы: `=`, `IF-THEN-END`, `WHILE-END`, `PRINT` . Встроенные функции: нет.

Для этого языка разработана следующая грамматика:

```
#0 <M>
#1 <M>=<S><M>
#2 <M>=.
#3 <S>=[LF]
#4 <S>=[dim][long][id]<D.>[LF]
#5 <D.>=[,][id]<D.>
#6 <D.>=.
#7 <S>=[id][=]<E>[LF]
#8 <S>=[if]<E>[then][LF]<M><A>
#9 <A>=[end][LF]
#10 <A>=[else]<M>[end][LF]
#11 <S>=[while]<E>[LF]<M>[end][LF]
#12 <S>=[print]<E>[LF]
#13 <E>=<R><R.>
#14 <R.>=[=]<R><R.>
#15 <R.>=[<>]<R><R.>
#16 <R.>=.
#17 <R>=<T><T.>
#18 <T.>=[+]<T><T.>
#19 <T.>=[-]<T><T.>
#20 <T.>=.
#21 <T>=<P><P.>
#22 <P.>=[*]<P><P.>
#23 <P.>=[/]<P><P.>
#24 <P.>=.
#25 <P>=[id]
#26 <P>=[I4]
#27 <P>=[(]<E>[)]
```

2. Работа 2. «Лексический анализатор»

Цели:
- консольное приложение и класс лексического анализа.
Задачи:
- чтение текстового файла программы в буфер приложения;
- формирование таблицы перекодировки входных символов;
- разработка функции для продвижения по тексту;
- разработка автомата предварительного лексического разбора.

2.1. Начальный проект

Создаем консольное приложение Win32, язык C/C++.
Название приложения SIMPLET, папка приложения C:\SIMPLET.
Модуль SIMPLET.cpp содержит основную функцию:

```
// SIMPLET.cpp
#include "stdafx.h"
void main(int argc, char * argv[]) {
}
```

Модуль stdafx.h включает следующие библиотеки:

```
// stdafx.h
#pragma once
#define _CRT_SECURE_NO_WARNINGS
#include "targetver.h"
#include <stdio.h>
#include <tchar.h>
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
```

Недостающие библиотеки следует добавить самостоятельно.

Предполагается, что консольная программа, запущенная из командной строки, принимает один параметр — спецификацию файла, который подлежит обработке.

При этом могут возникнуть следующие непредвиденные обстоятельства, которые должны быть обработаны:

- не задан параметр командной строки;
- параметр командной строки указывает на несуществующий файл.

Для простоты будем выводить сообщения обо всех непредвиденных ситуациях на консоль.

В папку проекта SIMPLET следует скопировать файлы, предоставляемые преподавателем, sttot.h, sttypes.h, и включить их в проект при помощи меню «Project — Add Existing Item».

2.2. Проверка непредвиденных ситуаций

Для проверки отсутствия (присутствия) параметра командной строки нужно сравнить количество аргументов главной функции с двойкой или единицей. Если задан один аргумент, то он является спецификацией файла самой консольной программы, например:

```
void main(int argc, char * argv[]) {
    if (argc < 2) {
        printf("Source file not specified\n");
        return;
    }
}
```

После запуска программы это сообщение должно выводиться.

В свойствах проекта установим аргумент командной строки, который необходим для отладки программы: 1.st. Кроме того, установим рабочую папку C:\SIMPLET\Debug.

Снова запускаем программу и убеждаемся, что теперь сообщение не выводится.

Далее проверяем существование файла. Для этого нам потребуется функция, которая открывает заданный файл. Перед главной функцией размещаем прототип этой функции:

```
// открывает файл программы для чтения
HANDLE SourceOpen(char * file_path);
```

После главной функции main описываем функцию SourceOpen:

```
// открывает файл программы для чтения
HANDLE SourceOpen(char * file_path) {
    return CreateFile(
        file_path,
        GENERIC_READ,           // открыть для чтения
        FILE_SHARE_READ,        // совмместный доступ на чтение
        NULL,                   // не учитывать безопасность
        OPEN_EXISTING,          // только существующий файл
        FILE_ATTRIBUTE_NORMAL,  // нормальные атрибуты файла
        0);                     // нет шаблона
}
```

Возвращаемся в главную функцию и пробуем открыть файл, заданный аргументом командной строки, после чего проверяем результат:

```
    // пробуем открыть файл
    HANDLE hFile = SourceOpen(argv[1]);
    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Source file does not exist.\n");
        return;
    }
```

После запуска программы это сообщение должно выводиться.

2.3. Подготовка текста программы на заданном языке

Создадим в папке C:\SIMPLET\Debug текстовый файл 1.st. Обратим внимание, что расширение содержит только два символа.

Файл содержит исходный транслируемый текст:

```
1 F +
```

Строка начинается с пробела, между всеми символами пробел, в конце строки следует нажать Enter, чтобы сформировать перевод строки.

2.4. Чтение файла в буфер

Существует много различных способов чтения входного файла. Поскольку нашей целью является построение простого транслятора, мы поступим самым простым способом, — прочитаем весь входной файл во внутренний буфер программы.

Нам понадобится буфер buff, переменная для размера файла dwBytes, переменная для количества прочитанных байт dwRead.

Объявим эти переменные перед прототипом функции SourceOpen:

```
// буфер текста
char buff = 0;
// чтение текста
DWORD dwBytes, dwRead;
```

Возвращаемся в главную функцию и определяем длину файла:

```
    // получим длину файла
    dwBytes = GetFileSize(hFile, 0);
```

Длину файла нужно проверить на ноль:

```
    // проверяем нулевую длину
    if (dwBytes == 0) {
        printf("Source file is empty.\n");
        return;
    }
```

Убедимся также, что длина файла не превышает 8190 байт, чтобы не выделять слишком много памяти.

```
    // проверяем максимальную длину
    if (dwBytes > 8190) {
        printf("Source file is too big.\n");
        return;
    }
```

Если длина файла не нулевая, то нужно выделить память для буфера на два символа больше, чем длина файла. Один символ нужен для признака конца текста, другой для признака конца строки:

```
    // выделим буфер
    buff = new char[2 + dwBytes];
```

Нужно проверить, удалось выделить буфер или нет. Самостоятельно. В случае ошибки выводим сообщение «Error allocate buffer».

Далее считываем файл целиком:

```
    // считываем файл в буфер
    if (ReadFile(hFile, buff, dwBytes, & dwRead, 0) == 0) {
        printf("Error reading source.\n");
        return;
    }
```

Если файл прочитать не удалось, то не удалось. Этот факт нам не важен для выполнения работ.

Признак конца текста определен в модуле sttypes.h, который сейчас нужно подключить:

```
#include "stdafx.h"
#include "sttypes.h"
```

После чтения файла в буфер добавляем в конец буфера признак конца текста CH_EOT и нулевой байт:

```
    // окончание буфера
    buff[dwRead++] = CH_EOT;
    buff[dwRead] = 0;
```

На данный момент это все, в конце главной функции нужно вернуть динамически выделенную память, например:

```
    delete [] buff;
```

2.5. Класс лексического анализатора

Добавляем в проект модуль для класса лексического анализатора.

Название модуля lexan.h.

Описываем класс лексического анализатора:

```
// lexan.h
#pragma once
// лексический анализатор
struct lexan {
};
```

Включаем модуль класса в основной модуль SIMPLET.cpp:

```
#include "stdafx.h"
#include "sttypes.h"
#include "lexan.h"
```

Описываем указатель на буфер входного текста и конструктор класса:

```
struct lexan {
    // конструктор
    lexan(char * buff) {
        m_buff = buff;
    }
private:
    // буфер текста
    char * m_buff;
};
```

Определяем метод класса, который будет выполнять разбор текста:

```
struct lexan {
    // конструктор
    lexan(char * buff) {
        m_buff = buff;
    }
    // разбор текста
    int parse() {
        return 1;
    } // parse
private:
    // буфер текста
    char * m_buff;
};
```

В главной функции создаем лексический анализатор и вызываем этот метод:

```
void main(int argc, char * argv[]) {
    . . .
    . . .
    . . .
    . . .
    // лексический анализатор
    lexan lex(buff);
    // разбор текста
    int res = lex.parse();
    delete [] buff;
}
```

2.6. Таблица перекодировки

Дальше каждый из нас пойдет своим путем, зависящим от входного языка. Сейчас нам нужна таблица перекодировки символов. Она находится в модуле sttot.h, который нужно подключить к модулю класса лексического анализатора:

```
// lexan.h
#pragma once
#include "sttot.h"
// лексический анализатор
struct lexan {
```

Откроем модуль sttot.h. Он содержит таблицу перекодировки. С использованием этой таблицы можно быстро определить, чем является очередной символ текста, например, буквой, цифрой, знаком операции и т.п.

Для каждого из нас задан свой набор операций. Поэтому некоторые символы из возможных символов таблицы ASCII, которой мы пользуемся для описания текста программы, допустимы, а некоторые недопустимы.

Для заданного входного языка определены следующие операции (смотрю описание своего языка):

«Операции: +, -, *, /, =, <>»

Это означает, что в тексте программы могут встретиться соответствующие знаки операций (операция типа AND здесь не имеет значения, как и любая другая, задаваемая с помощью идентификатора).

Смотрю на таблицу TOT и для всех символов, которые являются началами лексем, в таблице ставлю значение OPERA, а для всех остальных символов ставлю значение INVAL (недопустимое значение).

Здесь также важно понимать, что не все символы, которые встречаются в символах операций, являются началами лексем. Например, если перечень операций является следующим:

```
+,  -,  *,  /,  =,  <>
```

то символ "**>**" не является допустимым, поскольку он ни в какой операции не является первым.

В число допустимых знаков операций должны быть также включены знаки пунктуации (пунктуаторы), которые зависят от языка.

Примерами пунктуаторов являются знаки: ( ) , . ; : { }

Чтобы не ошибиться, откройте текстовый файл, который был сгенерирован программой SYNAX, и ищите в нем знаки пунктуации. Для моей грамматики был сгенерирован следующий файл (часть файла):

```
//  5 [,]
//  6 [=]
// 13 [<>]
// 14 [+]
// 15 [-]
// 16 [*]
// 17 [/]
// 18 [I4]
// 19 [(]
// 20 [)]
```

Из этого файла видно, что пунктуаторами в моем языке являются только запятая и круглые скобки.

В соответствии с вышесказанным я получаю следующие допустимые и недопустимые знаки операций:

```
unsigned char TOT[256] = { /* (c) ReVoL */
    /* CONTROLS */
    /* 00     01     02     03     04     05     06     07 */
    ENDOT, ENDOT, INVAL, INVAL, INVAL, INVAL, INVAL, INVAL,
    /* 08    TAB     LF     0B     0C     CR     0E     0F */
    INVAL, WHITE, WH_LF, INVAL, INVAL, WH_CR, INVAL, INVAL,
    /* 10     11     12     13     14     15     16     17 */
    INVAL, INVAL, INVAL, INVAL, INVAL, INVAL, INVAL, INVAL,
    /* 18     19     1A     1B     1C     1D     1E     1F */
    INVAL, INVAL, INVAL, INVAL, INVAL, INVAL, INVAL, INVAL,
    /* SIGNS */
    /* 20      !      "      #      $      %      &      ' */
    WHITE, INVAL, INVAL, INVAL, INVAL, INVAL, INVAL, INVAL,
    /*  (      )      *      +      ,      -      .      / */
    OPERA, OPERA, OPERA, OPERA, OPERA, OPERA, INVAL, SLASH,
    /*  0      1      2      3      4      5      6      7 */
    DIGIT, DIGIT, DIGIT, DIGIT, DIGIT, DIGIT, DIGIT, DIGIT,
    /*  8      9      :      ;      <      =      >      ? */
    DIGIT, DIGIT, INVAL, INVAL, OPERA, OPERA, INVAL, INVAL,
    /* ALPHAS */
    /*  @      A      B      C      D      E      F      G */
    INVAL, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA,
    /*  H      I      J      K      L      M      N      O */
    ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA,
    /*  P      Q      R      S      T      U      V      W */
    ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA,
    /*  X      Y      Z      [      \      ]      ^      _ */
    ALPHA, ALPHA, ALPHA, INVAL, INVAL, INVAL, INVAL, UNDER,
    /*  `      a      b      c      d      e      f      g */
    INVAL, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA,
    /*  h      i      j      k      l      m      n      o */
    ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA,
    /*  p      q      r      s      t      u      v      w */
    ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA, ALPHA,
    /*  x      y      z      {      |      }      ~        */
    ALPHA, ALPHA, ALPHA, INVAL, INVAL, INVAL, INVAL, INVAL,
```

Красным цветом здесь помечены знаки, которые не являются началами лексем в заданном языке. Синим цветом отмечены знаки, которые являются началами лексем операций или знаками пунктуации.

Обратите внимание на знак "/" (слеш), который является началом лексемы комментария и одновременно знаком операции. Он помечен особо, поскольку по заданию комментарий в стиле языка Си является обязательным.

2.7. Перечисление токенов

Файл, сгенерированный программой SYNAX, содержит перечисление всех символов грамматики. Эти символы включают в себя токены, они начинаются с префикса TOK_, и нетерминалы, они начинаются с префикса SYM_.

В настоящий момент перечисление символов, расположенное в модуле sttypes.h, имеет следующий вид:

```
// Символы
typedef enum _STTokenType {
    TOK_COMMENT = -2,
    TOK_UNKNOWN = -1,
    // Токены
    TOK_EOT = 0,
    // Символы генератора кода
    OUT_END
} STTokenType;
```

Сгенерированный мне файл содержит следующее перечисление:

```
    // Токены
    TOK_EOT = 0,
    TOK_LF,
    TOK_DIM,
    TOK_LONG,
    TOK_ID,
    TOK_COMMA,
    TOK_EQ,
    TOK_IF,
    TOK_THEN,
    TOK_END,
    TOK_ELSE,
    TOK_WHILE,
    TOK_PRINT,
    TOK_NE,
    TOK_ADD,
    TOK_SUB,
    TOK_AST,
    TOK_FDIV,
    TOK_I4,
    TOK_LP,
    TOK_RP,
    TOK_LAST = TOK_RP,
    // Нетерминалы
    SYM_M,
    SYM_S,
    SYM_D_,
    SYM_E,
    SYM_A,
    SYM_R,
    SYM_R_,
    SYM_T,
    SYM_T_,
    SYM_P,
    SYM_P_,
    SYM_LAST = SYM_P_,
    // Конец символов грамматики
```

Сгенерированные здесь символы TOK_LAST и SYM_LAST являются служебными и впоследствии будут использованы в МП-автомате.

Этот текст нужно скопировать от строки

```
// Токены
```

до строки

```
// Конец символов грамматики
```

включительно и вставить в перечисление STTokenType вместо строк

```
// Токены
TOK_EOT = 0,
```

Изменять порядок следования констант перечисления недопустимо.

2.8. Предварительный лексический разбор

Как известно из теоретического курса, лексемы во входном тексте выделяются с помощью конечных автоматов. Так, для выделения идентификатора используется конечный автомат для разбора идентификатора, а для выделения константы используется конечный автомат для разбора константы. Возникает вопрос, кто и когда эти автоматы вызывает (конечный автомат в программе — это функция, в нашем случае функция класса, то есть метод).

Существует большое количество способов построения лексического анализатора. Мы используем следующий способ. Будем анализировать очередной символ входного текста и определять, началом лексемы какого типа он является. В зависимости от этого будем вызывать тот или иной конечный автомат.

Таким образом, предварительный лексический разбор заключается в определении типа лексемы по ее первому символу. Для этого необходимо, чтобы синтаксис языка допускал возможность такого разбора. В наших и вообще в современных языках это возможно.

Сначала нужно составить для себя перечень всех возможных первых символов лексем. Я сделаю это в виде следующей таблицы.

Первый символ	Лексема
TOT[символ] == WHITE	пробельный символ
TOT[символ] == WH_LF	перевод строки
TOT[символ] == WH_CR	возврат каретки
TOT[символ] == ALPHA	идентификатор
TOT[символ] == DIGIT	целочисленная константа
TOT[символ] == SLASH	либо комментарий, либо операция
TOT[символ] == OPERA	операция или пунктуатор

Порядок следования первых символов должен быть следующим: сначала пробельные символы и символы, связанные с концами строк, то есть

символы перевода строки и возврата каретки. Однако не для всех языков нужно анализировать символы перевода строки и возврата каретки.

Так, в языках Pascal и Си эти символы не имеют никакого значения и поэтому должны быть отмечены как пробельные в таблице перекодировки TOT с помощью константы WHITE (вместо WH_LF и WH_CR). Соответственно в таблице первых символов эти символы не указываются. Однако если в дальнейшем нам нужно будет подсчитывать строки текста, чтобы указывать текущую позицию, то символ WH_LF следует анализировать.

За пробельными символами будем указывать идентификаторы. Заметим, что идентификаторы в разных языках также могут задаваться разными способами. Наиболее распространены идентификаторы, которые начинаются либо с буквы, либо со знака подчеркивания, за которыми далее могут следовать буквы, цифры и знаки подчеркивания. В моем языке идентификатор не может начинаться со знака подчеркивания. Если бы я хотел сделать, чтобы идентификатор начинался также со знака подчеркивания, то я мог бы, например, отметить знак подчеркивания в таблице перекодировки TOT константой ALPHA вместо константы UNDER.

За идентификаторами будем разбирать константы. Константы могут быть целочисленными, вещественными, строковыми и символьными. Сначала разбираем целочисленные, затем вещественные, затем символьные, затем строковые. В соответствии с этим порядок первых символов лексем может быть таким:

TOT[символ] == DIGIT	целочисленная константа
TOT[символ] == CHDOT	вещественная константа
TOT[символ] == SINQU	символьная константа
TOT[символ] == QUOTE	строковая константа

Заметим, что если в языке заданы вещественные константы, то символ точки должен быть отмечен в таблице перекодировки TOT константой CHDOT. Если к тому же точка является пунктуатором, то это вызовет проблему разбора. Решать эту проблему будем следующим образом: если точка является пунктуатором, то вещественная константа не может начинаться с точки (начинается с целого числа).

Если в языке заданы символьные константы, то символ одиночной кавычки в таблице перекодировки должен быть отмечен константой SINQU. Если в языке заданы строковые константы, то символ двойной кавычки в таблице перекодировки должен быть отмечен константой QUOTE.

Последними в таблице первых символов должны быть символы операций и пунктуаторов, которые в таблице перекодировки отмеченные константой OPERA.

Если первый символ лексемы не является ни одним из указанных в таблице, он является недопустимым символом в программе.

На самом деле есть еще один символ, который нужно разбирать в каком-то месте. Это символ конца текста CH_EOT, который отмечен в таблице перекодировки константой ENDOT.

После того, как будет понятно, какие символы могут появляться в начале лексем, мы может приступить к проектированию предварительного лексического анализатора.

На самом деле мы уже все спроектировали, осталось это записать в виде соответствующего оператора, однако нужно сделать несколько дополнительных действий.

Вернемся к классу лексического анализатора. В каждый момент разбора мы анализируем либо текущий символ входного текста, либо его заменитель, который получается с помощью таблицы перекодировки. Однако самого символа у нас пока нет, в смысле переменной, которая его содержит. Поэтому добавляем следующие элементы данных в наш класс:

```
// рабочий (текущий) символ
char m_s;
// индекс текущего символа
int m_curr;
// индекс последнего символа
int m_last;
```

Кроме этого, нам понадобится также знать, в каком месте входного текста мы находимся, а для этого нужны еще два элемента данных:

```
// текущая строка
int m_blin;
// текущая позиция
int m_bpos;
```

Разбор текста нужно начать с получения первого символа текста. По ходу разбора мы должны перемещаться от первого символа к последующим символам. Нам нужен метод для продвижения по тексту, который мы назовем move_next и разместим после метода parse:

```
// переходит к следующему символу
void move_next() {

} // move_next
```

Этот метод должен устанавливать переменную m_s в значение следующего символа, если следующий символ есть. Поэтому если текущий символ в буфере равен CH_EOT, то метод должен просто завершаться, оставляя текущий символ неизменным (то есть равным CH_EOT).

В противном случае нужно переместить указатель буфера на следующий символ и прочитать следующий символ в переменную m_s. После этого нужно увеличить текущую позицию в строке m_bpos. Программируем указанные действия.

Нам нужен также метод, с помощью которого мы будем выводить сообщения анализатора. Назовем этот метод message_out и разместим после метода move_next:

```
// выводит сообщение
void message_out(char * message) {
    printf("lexan[%d:%d] %s.\n", m_blin, m_bpos, message);
} // message_out
```

Мы выводим сообщения анализатора в следующем формате:

```
lexan[m:n] message.
```

Здесь m — номер строки, n — позиция в строке в текущий момент.

Наконец, можно перейти к формированию метода разбора parse.

Сначала нужно вычислить количество символов в буфере, задать текущую строку и текущую позицию, установить текущее положение в тексте, получить первый символ:

```
void parse() {
    // индекс последнего символа
    m_last = strlen(m_buff);
    // текущая строка в тексте
    m_blin = 1;
    // текущая позиция в тексте
    m_bpos = 0;
    // текущая позиция в буфере
    m_curr = 0;
    // получим первый символ текста
    m_s = m_buff[m_curr];
} // parse
```

Далее следует бесконечный цикл предварительного разбора:

```
    // счетчик итераций
    int it_count = 0;
    // цикл предварительного разбора
    while (1) {
    } // while (1)
```

Переменная it_count нужна для подсчета итераций и выхода из цикла в случае слишком большого количества итераций (зацикливания). Для этого мы проверяем число итераций, которое не может быть больше числа символов в буфере.

```
    // цикл предварительного разбора
    while (1) {
        // проверка на зацикливание алгоритма
        if (it_count++ > m_last) {
            message_out("Internal deadlock");
            return 0;
        }
    } // while (1)
```

Эта проверка на зацикливание пусть всегда находится в конце цикла предварительного разбора.

Собственно разбор выполняем с помощью оператора switch.

Мы разбираем преобразованный с помощью таблицы перекодировки текущий символ текста и принимаем решение о том или ином действии:

```
while (1) {
    // разбираем преобразованный символ
    switch (TOT[m_s]) {
    case WHITE: // пробельный символ
        move_next();
        break;
    default:
        // недопустимый символ в строке
        message_out("Extra character");
        return;
    }
    // проверка на зацикливание алгоритма
    . . .
} // while (1)
```

Пока мы разобрали только пробельный и недопустимый символы. Постепенно добавим другие варианты разбора, которые получатся разными для разных проектов.

Добавим окончание разбора, то есть момент обнаружения признака конца текста. Место размещения, как было сказано, произвольное. Для определенности пусть будет первое место, то есть перед разбором пробельного символа:

```
switch (TOT[m_s]) {
case ENDOT: // успешное завершение
    message_out("Success");
    return 1;
```

Далее идем строго по порядку. Если нужно подсчитывать строки (например, в языке типа BASIC), то разбираем символ WH_LF:

```
case WH_LF: // перевод строки
    move_next();
    m_blin++;
    m_bpos = 0;
    break;
```

Далее разбираем идентификатор (это для всех):

```
case ALPHA: // идентификатор
    move_next();
    break;
```

Сейчас мы просто переходим к следующему символу, но в дальнейшем мы должны заменить этот переход вызовом соответствующего конечного автомата и разбором полученного результата. Это касается также и

всех других последующих вариантов разбора, в которых вызывается конечный автомат. Далее разбираем числовую константу:

```
case DIGIT: // числовая константа
    move_next();
    break;
```

Далее следуют разбор вещественной константы, строковой константы и операций. У каждого получится свой набор. Только разбор случая, когда символ является знаком "/", обозначаемым константой SLASH, должен быть у всех. Первоначально это может выглядеть примерно так:

```
case CHDOT: // вещественная константа
    move_next();
    break;
case QUOTE: // строковая константа
    move_next();
    break;
case SLASH: // комментарий или операция
    move_next();
    break;
case OPERA: // операция, пунктуатор
    move_next();
    break;
```

Если все сделано правильно и входной текст не содержит недопустимых символов, то программа должна выдавать «Success», то есть успешное окончание разбора.

3. Работа 3. «Конечный автомат»

Цель: разработка конечного автомата для разбора комментария.

3.1. Общие замечания

На разработку конечных автоматов лексического анализа отводится два занятия (4 часа). По окончании разработки конечного автомата для разбора комментария сразу переходите к разработке других автоматов, так как времени должно остаться достаточно для этого.

Как уже говорилось, конечный автомат — это функция. С практической точки зрения нам важно знать результат, поэтому функция должна возвращать его в виде константы перечисления STTokenType. Однако не практике возможны различные варианты.

Все лексемы условно можно поделить на две группы.

Лексемы первой группы ограничиваются с правой стороны первым недопустимым для данной лексемы символом. Примерами таких лексем служат идентификатор или целочисленная константа. Пусть нам нужно выделить идентификатор в тексте "abc+def". Мы принимаем все символы идентификатора до тех пор, пока очередным символом не станет символ "+", недопустимый для идентификатора. Автомат, разбирающий идентификатор, не может выдать ошибочное состояние.

Лексемы второй группы справа ограничены определенным символом или последовательностью символов. Примерами таких лексем являются комментарий в стиле языка Си, строковые и символьные литералы. Автомат, который разбирает лексему второй группы, может войти в ошибочное состояние в случае, если не может обнаружить завершающую последовательность символов.

На практике могут встречаться и промежуточные варианты. Например, автомат, который разбирает целочисленную константу с суффиксом в виде символа или символов, указывающих на размер константы. В этом случае после последовательности цифр, которая в общем случае не приводит к ошибочному состоянию, может следовать буква, которая не принадлежит ни к какому допустимому суффиксу. Можно построить автомат так, что он примет только число без суффикса, и тогда суффикс будет проанализирован как идентификатор. В этом случае ошибка будет обнаружена на стадии синтаксического анализа. Однако если учитывать, что за константой не может следовать идентификатор, автомат можно построить и так, что он выдаст ошибочное состояние и ошибка будет обнаружена раньше.

Поскольку существуют различные группы лексем, будем по-разному называть конечные автоматы, их распознающие. Автоматы, принимающие лексему в любом случае, будем называть с префиксом get_, а автоматы, которые могут обнаруживать ошибки, будем называть с префиксом is_.

При этом автоматы get могут ничего не возвращать, если их результат очевиден, а автоматы второй группы должны возвращать результат разбора. Условимся, что автоматы возвращают константу TOK_UNKNOWN в случае обнаружения ошибки. Заметим, что на практике может сложиться и иная ситуация.

Есть еще один важный момент, касающийся всех автоматов. После завершения работы любого автомата текущим символом должен быть тот, который непосредственно следует за лексемой, и неважно, какой это символ. Это накладывает на нас ответственность за правильное применение функции move_next. Лишний ее вызов может привести к пропуску символа входного текста, а пропущенный вызов может привести к формированию лишней лексемы, которой в тексте на самом деле нет.

3.2. Предварительное моделирование ситуации

Переходим в модуль lexan.h и описываем метод is_comment_c.

Размещаем его после метода parse:

```
// разбирает комментарий Си
STTokenType is_comment_c() {
    return TOK_UNKNOWN;
} // is_comment_c
```

Поскольку эта функция должна возвращать результат, возвращаем из функции константу TOK_UNKNOWN, и программа не содержит ошибок.

По завершении разработки оператор возврата может оказаться лишним, но сейчас он нам нужен как заглушка.

Далее нужно понять, где вызывается эта функция и как она должна работать. Первый символ лексемы комментария, — это слеш, который может оказаться и операцией деления.

У нас есть два варианта решения проблемы выбора.

В первом случае мы можем, обнаружив первый символ лексемы "/" в цикле предварительного разбора, сразу проанализировать и последующий символ и на основании этого анализа сделать выбор в пользу вызова автомата или в пользу формирования токена операции деления. В этом случае автомат может возвращать только TOK_COMMENT и TOK_UNKNOWN.

Во втором случае мы можем принять решение в автомате, если после приема первого символа "/" мы обнаружим, что следующий символ не является символом "*". В этом случае автомат может возвращать три варианта результата: TOK_FDIV, TOK_COMMENT и TOK_UNKNOWN.

Автомат может обнаружить также конец текста в какой-то фазе разбора. В этом случае лексический анализатор завершает работу с сообщением «Неожиданный конец текста». Мы должны решить, в каком месте и какая часть лексического анализатора должна сформировать это сообщение. Решение принимается следующим образом.

Если ошибочное состояние автомата единственное, то сообщение может формировать цикл предварительного разбора, в противном случае сообщение формирует функция конечного автомата. В нашем случае ошибочное состояние единственное, поскольку, если комментарий не закрыт, автомат дойдет до конца текста.

Переходим в метод parse и описываем переменную для результата:

```
// счетчик итераций
int it_count = 0;
// результат работы конечного автомата
STTokenType fa_result = TOK_UNKNOWN;
// цикл предварительного разбора
```

Анализируем результат разбора автомата is_comment_c:

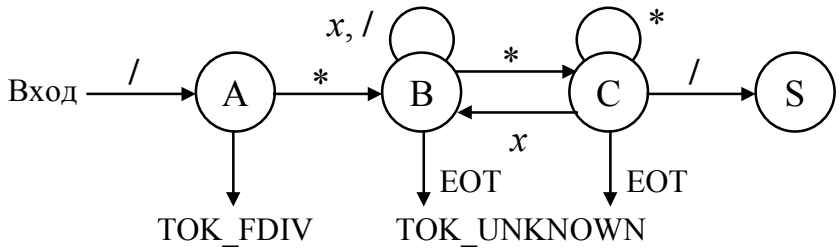
```
case SLASH: // комментарий или операция
    fa_result = is_comment_c();
    if (fa_result == TOK_FDIV) {
        // операция деления
    } else if (fa_result == TOK_UNKNOWN) {
        // ошибка комментария
        message_out("Unexpected end of file in comment");
        return 0;
    }
    break;
```

Заметим, что вызов функции move_next удален.

Мы не анализируем третий результат, так как нам он практически не важен, — мы удаляем комментарии из текста и не формируем токенов.

3.3. Конструирование автомата

Чтобы сконструировать конечный автомат, нужно иметь подходящее его описание. Мне проще конструировать, имея перед глазами граф переходов, поэтому я его рисую.



Работа автомата основана на понятии состояния. Поэтому определим состояния автомата в виде перечисления и зададим начальное состояние:

```
STTokenType is_comment_c() {
    // состояния
    enum { A, B, C, S } state = A;
    return TOK_UNKNOWN;
} // is_comment_c
```

Далее следует бесконечный цикл, в котором осуществляется переход к следующему символу и анализ состояний:

```
enum { A, B, C, S } state = A;
// рабочий цикл автомата
while (1) {
    move_next();
    switch (state) {
    case A:
        break;
    case B:
        break;
    case C:
        break;
    case S:
        break;
    }
}
```

Обратим внимание, что в автомате отсутствует начальное состояние. Мы входим в автомат при условии, что текущий символ соответствует начальному состоянию и нет необходимости его анализировать. Первое действие автомата перемещает текущий символ, что также соответствует состоянию «A». В этом состоянии, если текущий символ звездочка, мы переходим к состоянию «B», в противном случае возвращаемся с результатом в виде константы TOK_FDIV:

```
case A:
    if (m_s == '*') {
        state = B;
    } else {
        return TOK_FDIV;
    }
    break;
```

Нужно проверить, правильно ли мы запрограммировали эти действия. Открываем файл 1.st, редактируем его следующим образом:

```
/1•
```

Здесь символ "•" обозначает конец файла. Сохраняем текст и начинаем отладку. Устанавливаем точку остановки, например, на строке while автомата и запускаем выполнение.

Убеждаемся, что текущий символ m_s имеет значение '/' при входе в автомат. Делаем два шага, при этом вызывается функция move_next, текущим символом становится '1'.

Последующие шаги должны убедить нас, что автомат завершил работу, возвращаемое значение равно TOK_FDIV, текущий символ не изменился, и цикл предварительного разбора переходит к разбору целочисленной константы.

Переходим к состоянию «B». Здесь нам встречается символ "*x*", обозначающий «любой другой символ» и нужно понять, как это запрограммировать. Запомним: «любой другой символ» разбирается последним, он соответствует понятию «иначе». Если для разбора символа используется оператор switch, то понятию «иначе» соответствует default. Поэтому в этом состоянии мы анализируем звездочку и конец текста, а иначе не делаем ничего, поскольку «иначе» мы остаемся в текущем состоянии:

```
case B:
    if (m_s == CH_EOT) {
        return TOK_UNKNOWN;
    } else if (m_s == '*') {
        state = C;
    }
    break;
```

Для отладки этой части используем следующий текст файла 1.st:

```
/*1•
```

Пошаговой отладкой проверяем, как автомат переходит в состояние «B», принимает в этом состоянии символ '1', обнаруживает символ конца текста, далее автомат завершает работу, цикл разбора формирует сообщение, и работа лексического анализатора в целом также завершается.

Переходим к разбору символов в состоянии «C». Здесь также встречается «любой другой символ», и, поскольку он переводит автомат в другое состояние, разбор этого «любого другого символа» должен присутствовать явным образом. Примерный код для этого состояния следующий:

```
case C:
    if (m_s == CH_EOT) {
        return TOK_UNKNOWN;
    } else if (m_s == '*') {
    } else if (m_s == '/') {
        state = S;
    } else {
        state = B;
    }
    break;
```

Для отладки этой части используем следующий текст файла 1.st:

```
/**1•
```

При этом автомат перейдет в состояние «C» при обнаружении второй звездочки и вернется в состояние «B», обнаружив символ '1'. Далее автомат и лексический анализатор в целом завершают работу так, как это происходило в предыдущем случае.

Нам остается запрограммировать состояние «S». В этом состоянии автомат просто возвращает константу TOK_COMMENT, сообщая об успешном завершении разбора.

Для отладки этой части используем следующий текст файла 1.st:

```
/**/1•
```

Нам важно, чтобы после завершения работы автомата цикл предварительного разбора обнаружил целочисленную константу. Если это так, автомат правильно функционирует. Убеждаемся в этом самостоятельно.

4. Работа 4. «Разбор лексем»

Цель: конечные автоматы лексического разбора.

В этом занятии, которое является продолжением предыдущего, мы должны реализовать все необходимые конечные автоматы для заданного входного языка.

4.1. Конечный автомат для приема идентификатора

Это один из самых простых конечных автоматов. Теоретически он имеет два состояния, но поскольку начальное состояние практически не требуется, автомат имеет одно состояние, поэтому нет необходимости его как-то обозначать. Иначе говоря, этот автомат представляет собой просто бесконечный цикл, который принимает символы идентификатора до тех пор, пока не обнаружит недопустимый символ.

Автомат принимает идентификатор в любом случае, поэтому его логично назвать get_ident. Разместим функцию этого автомата перед функцией автомата для разбора комментария:

```
// принимает идентификатор
STTokenType get_ident() {
}
```

Переходим в цикл предварительного разбора и вызываем этот конечный автомат:

```
        case ALPHA: // идентификатор
            get_ident();
            break;
```

Собственно автомат, как было сказано, очень простой, мы убеждаемся, что текущий символ является буквой, цифрой или знаком подчеркивания, а иначе возвращаемся из автомата в цикл предварительного разбора:

```
STTokenType get_ident() {
    while (1) {
        switch (TOT[m_s]) {
        case ALPHA: case DIGIT: case UNDER:
            move_next();
            break;
        default:
            return TOK_ID;
        }
    }
}
```

Для отладки нам потребуется другой текст 1.st, например:

```
ab+•
```

Легко убедиться, что автомат работает правильно, и после выхода из автомата цикл предварительного разбора обнаруживает литерал оператора, что и требуется.

4.2. Разбор числовых констант

Если бы язык определял числовые константы только одного типа, все было бы достаточно просто. Автомат для приема целочисленной константы по своей конструкции ничем не отличается от автомата для приема идентификатора. Однако в языках такая ситуация никогда не встречается и на практике лексический анализатор должен распознавать множество различных числовых констант.

Проблема заключается в том, что разные числовые константы могут начинаться с разных символов. Например, целочисленная константа начинается с цифры, а символьная со знака одиночной кавычки.

При этом возникает вопрос, нужно ли конструировать один автомат для разбора всех типов числовых констант, или каждую константу разбирать своим собственным автоматом. Мое личное предпочтение заключается в выборе нескольких автоматов, потому что в этом случае они получатся более простыми, чем один невообразимой сложности монстр.

Другая проблема связана с разбором целочисленных и вещественных констант. И те, и другие могут начинаться с цифры, однако вещественные константы могут начинаться с точки. Это не является сложной задачей, если точка не является пунктуатором. Однако во всех современных языках, поддерживающих парадигму объектно-ориентированного программирования, точка пунктуатором является и соответственно нужно как-то выяснять, какое значение она имеет, то есть знать контекст. Например, если точка следует за идентификатором, то она является частью идентификатора или отдельным пунктуатором, а если за цифрой, то она является частью вещественной константы.

Рассмотрим следующий пример программы на языке Pascal.

```
var a : double;
begin
  a := .1;
end.1;
```

Если в первом случае точка начинает вещественную константу ".1", то почему во втором случае она должна быть выделена в пунктуатор, завершающий программный текст? Я попробовал скомпилировать этот текст и убедился, что программа синтаксически некорректна. В языке Pascal вещественная константа не может начинаться с точки по причинам, которые были изложены. Поэтому в первом случае перед точкой следует поставить знак "0" и программа становится корректной.

Автомат для приема целочисленной константы самый простой:

```
// принимает целое число
STTokenType get_number() {
    while (1) {
        switch (TOT[m_s]) {
        case DIGIT:
            move_next();
            break;
        default:
            return TOK_I4;
        }
    }
} // get_number
```

Его можно вызвать в цикле предварительного разбора так:

```
        case DIGIT: // числовая константа
            get_number();
            break;
```

Если вещественные константы не заданы языком, то это все.

Если вещественные константы заданы, то тогда следует смотреть, могут ли они начинаться с точки, как в языке Си и многих других.

Есть два варианта разбора числовых констант в этом случае. Если константы могут начинаться только с цифры, то следует построить один автомат, принимающий целые и вещественные константы, например:

```
// разбирает числовую константу
STTokenType is_number() {
    get_number();
    if (m_s == '.') {
        move_next();
        if (TOT[m_s] == DIGIT) {
            get_number();
        }
        return TOK_R8;
    } else {
        return TOK_I4;
    }
} // is_number
```

Этот автомат использует автомат get_number чтобы принять последовательность цифр. Он вызывается в цикле предварительного разбора в случае, если начало лексемы цифра, например:

```
        case DIGIT: // числовая константа
            fa_result = is_number();
            if (fa_result == TOK_I4) {
                // целочисленная константа
            } else if (fa_result == TOK_R8) {
                // вещественная константа
            }
            break;
```

Для тестирования этого автомата используем следующие тексты:

```
1+•
1.+•
1.1+•
```

Цикл предварительного разбора должен обнаруживать оператор "+".

Если вещественная константа может начинаться с точки, то опять есть два варианта. Можно построить один автомат, который будет принимать константы, начинающиеся либо с цифры, либо с точки, а можно построить дополнительный автомат, который принимает только вещественные константы, начинающиеся с точки. В любом случае автомат может возвращать ошибку, если за начальной точкой не следует ни одной цифры.

Мое предпочтение заключается в том, чтобы построить дополнительный автомат для разбора вещественной константы, начинающейся с точки.

Этот автомат может иметь следующий вид:

```
// разбирает вещественную константу
STTokenType is_real() {
    move_next();
    if (TOT[m_s] == DIGIT) {
        get_number();
        return TOK_R8;
    } else {
        message_out("Invalid real number");
        return TOK_UNKNOWN;
    }
} // is_real
```

Этот автомат также использует функцию get_number. Он вызывается в цикле предварительного разбора в случае, если начало лексемы точка:

```
        case CHDOT: // вещественная константа
            fa_result = is_real();
            if (fa_result == TOK_R8) {
                // вещественная константа
            } else {
                // ошибочная вещественная константа
                return;
            }
            break;
```

Для тестирования этого автомата используем следующие тексты:

```
.+•
.1+•
```

В первом случае должна фиксироваться ошибка, во втором случае цикл предварительного разбора должен обнаруживать оператор "+".

Соединение автоматов is_number и is_real в один дает автомат, который будет принимать числовые константы, начинающиеся как с цифры, так и с точки. Тогда этот автомат вызывается один раз в случаях, когда первый знак лексемы цифра или точка.

4.3. Разбор строковых литералов

Строковый литерал начинается со знака двойной или одинарной кавычки, содержит любые печатаемые символы, некоторые управляющие символы и последовательности, и завершается знаком такой же кавычки.

Для простоты будем полагать, что язык не распознает никаких управляющих последовательностей (escape-последовательностей) и не может содержать управляющие символы.

Автомат для разбора строкового литерала будет возвращать ошибку в случаях обнаружения символов, код которых меньше кода пробела, иначе он принимает литерал, *приняв* второй символ кавычки.

Возникает вопрос, как тогда внутри литерала записать кавычку? Поскольку мы разработчики языка, то можем принять любое решение. Например, — никак. Никаких кавычек внутри кавычек.

В этом случае все достаточно просто и если литерал записывается в двойных кавычках, то конечный автомат может иметь следующий вид:

```
// разбирает строковый литерал
STTokenType is_quote() {
    while (1) {
        move_next();
        if (TOT[m_s] == QUOTE) {
            move_next();
            return TOK_QUOTE;
        } else if (m_s < ' ') {
            message_out("Unexpected end of file in quote");
            return TOK_UNKNOWN;
        }
    }
}
```

Обратим внимание, что если текущий символ кавычка, то перед выходом из автомата ее нужно принять, иначе она останется текущим символом и будет распознана как начало, а не как завершение литерала.

Заметим также, что если литерал заключается в одинарные кавычки, как в языке Pascal, то название автомата все равно должно быть is_quote.

Автомат вызывается, если первый символ лексемы соответствующая кавычка, например:

```
case QUOTE: // строковая константа
    fa_result = is_quote();
    if (fa_result == TOK_UNKNOWN) {
        // неправильный литерал
        return;
    } else {
        // правильный литерал
    }
```

Если же мы хотим записывать кавычки внутри кавычек, то можно предложить способ, используемый в языке Basic: две кавычки подряд — это кавычка внутри литерала. Тогда автомат немного усложнится:

```
// разбирает строковый литерал
STTokenType is_quote() {
    while (1) {
        move_next();
        if (TOT[m_s] == QUOTE) {
            move_next();
            if (TOT[m_s] != QUOTE) {
                return TOK_QUOTE;
            }
        } else if (m_s < ' ') {
            message_out("Unexpected end of file in quote");
            return TOK_UNKNOWN;
        }
    }
}
```

Для тестирования автомата используются тексты, содержащие четное и нечетное количество кавычек, например:

```
"+•
""+•
"""+•
""""+•
```

При нечетном количестве кавычек автомат возвращает ошибку.

4.4. Разбор операций

Последний конечный автомат разбирает операции и пунктуаторы. Он не сложный, но немножко нудный. Принцип работы его такой.

Если операция описывается одним знаком, то если этот знак не является началом другой операции, то возвращается константа, соответствующая операции. Если знак операции является началом двух и более операций, то после того, как принят первый знак, анализируется второй, при необходимости третий, и после этого принимается решение, какая операция принимается. Автомат не может давать ошибок, но он возвращает константу операции. Для моего языка автомат имеет вид:

```
// разбирает операции и пунктуаторы
STTokenType get_opera() {
    switch (m_s) {
    case '(': move_next(); return TOK_LP;
    case ')': move_next(); return TOK_RP;
    case ',': move_next(); return TOK_COMMA;
    case '+': move_next(); return TOK_ADD;
    case '-': move_next(); return TOK_SUB;
    case '*': move_next(); return TOK_AST;
    case '=': move_next(); return TOK_EQ;
    case '<':
```

```
            move_next();
            if (m_s == '>') {
                move_next();
                return TOK_NE;
            } else {
                return TOK_UNKNOWN;
            }
        default: return TOK_UNKNOWN;
        }
    }
```

Несмотря на то, что автомат разбирает все операции, на случай неправильного конструирования анализатора он возвращает ошибку.

Вызов автомата для разбора операций и пунктуаторов мало отличается от вызовов предыдущих автоматов:

```
            case OPERA: // операция, пунктуатор
                fa_result = get_opera();
                if (fa_result == TOK_UNKNOWN) {
                    // ошибка разработчика
                    message_out("Operator parse error");
                    return;
                }
                break;
```

4.5. Разбор ключевых слов

Таким образом, лексический анализатор сейчас должен разбирать все токены грамматики за исключением ключевых слов, которые пока относятся к идентификаторам. Чтобы определить ключевые слова, нужно анализировать идентификаторы. Здесь также есть варианты.

Первый вариант — построить такой автомат для разбора идентификатора, который сразу распознает ключевые слова. В качестве примера приведу автомат, который распознает только одно ключевое слово LONG:

```
// разбирает идентификаторы
STTokenType is_ident() {
    int state = 0;
    if ((m_s | 32) == 'l') state = 1;
    while (1) {
        move_next();
        switch (state) {
        case 0:
            return get_ident();
            break;
        case 1:
            if ((m_s | 32) == 'o') state = 2; else state = 0;
            break;
        case 2:
            if ((m_s | 32) == 'n') state = 3; else state = 0;
            break;
        case 3:
            if ((m_s | 32) == 'g') state = 4; else state = 0;
            break;
```

```
            case 4:
                switch (TOT[m_s]) {
                case ALPHA: case DIGIT: case UNDER: state = 0; break;
                default: return TOK_LONG;
                }
                break;
            }
        }
    }
```

Автомат либо разбирает слово буква за буквой, либо переходит в состояние 0, в котором принимается обычный идентификатор.

Для разбора множества ключевых слов этот автомат будет иметь много состояний и получится чрезвычайно сложным. Но самое главное заключается в том, что его сложно будет изменить для того, чтобы добавить или удалить разбор какого-то ключевого слова. Тем не менее, это один из вариантов, который используется в реальных компиляторах.

Вызов этого автомата может иметь следующий вид:

```
            case ALPHA: // идентификатор
                fa_result = is_ident();
                break;
```

Дополнительно замечу, что операция (m_s | 32) переводит символ m_s в нижний регистр, так как мой язык является регистро-независимым. Для регистро-зависимого языка эта операция не нужна.

Еще замечу, что если основательно потрудиться, автомат можно как-то минимизировать и оптимизировать, но это действительно адский труд, который допустим, если разрабатывается что-то стоящее.

Второй вариант простой. Сначала получим идентификатор, а затем вызовем автомат, который даст ответ в виде константы ключевого слова или константы TOK_ID. Это возможно, если сам идентификатор выделен в отдельный буфер. Буфера у нас пока нет, поэтому этот способ сейчас недоступен. Этот способ позволяет легко изменять набор ключевых слов, и мы применим его немного позднее.

5. Работа 5. «Классы токена»

Цель: разработка классов для формирования потока токенов.

Мы подошли к моменту, когда цель лексического анализа должна получить конкретное воплощение в виде потока токенов.

Токены и лексемы. Лексема — неделимая часть текста программы на заданном языке. Токен — это терминал синтаксической грамматики. Синтаксический анализатор строит вывод с использованием токенов, которые в нашем трансляторе заданы перечислением STTokenType. Однако синтаксический анализатор не только строит вывод, удостоверяя таким образом правильность входного текста, но и генерирует текст на выходном языке или производит вычисления, поэтому ему недостаточно знать, какие токены образуют текст программы. Синтаксический анализатор должен иметь некоторую дополнительную информацию о токенах.

Для примера, если мы строим интерпретатор, то тот факт, что в выражении встречается токен [I4], мало что дает для вычисления выражения, равно как и токен [quote], обозначающий строковый литерал.

Поэтому токены должны быть снабжены набором атрибутов, которые содержат всю необходимую информацию, нужную разработчику для генерации выходного текста или непосредственных вычислений.

Сложно предсказать, какая информация потребуется в дальнейшем, однако можно сделать некоторые предположения.

Мы полагаем, что текст токена не будет лишним, особенно если токен является идентификатором или строковым литералом. Если токен является числовой константой, нам нужно знать ее числовое значение. Чтобы формировать сообщения об ошибках, нам требуется знать положение токена в тексте программы, то есть строку и позицию в строке.

5.1. Класс токена

Добавим в проект новый модуль sttoken.h. Описываем в нем класс токена следующим образом:

```
// sttoken.h
#pragma once
#include "sttypes.h"
// класс токена
struct sttoken {
    // целочисленное значение
    int int_val;
    // вещественное значение
    int dbl_val;
    // тип токена
    STTokenType tt;
    // текстовое значение
    char str_val[2 + MAX_TOKEN_BUFF];
    // положение
    int tlin, tpos;
};
```

Заметим, что строковое значение токена ограничено константой, определенной в модуле sttypes.h. Для нас это означает, что строковый литерал не может быть больше по размеру, чем буфер str_val для ее хранения.

Добавим в класс метод clear и вызовем его в конструкторе:

```
    // метод для очистки
    void clear() {
        tt = TOK_UNKNOWN;
        tlin = 0;
        tpos = 0;
        int_val = 0;
        dbl_val = 0.0;
        for (int i = 0; i < (2 + MAX_TOKEN_BUFF); i++) str_val[i] = 0;
    }
    // конструктор
    sttoken() {
        clear();
    }
```

Больше ничего в этом классе не требуется, поэтому переходим к разработке класса, который будет представлять поток токенов.

5.2. Класс потока токенов

Для простоты будем полагать, что это массив элементов sttoken фиксированного размера. Это не очень правильно, но совсем не хочется возиться с динамическими массивами из-за того, что количество токенов может быть произвольным. Мы просто ограничим количество токенов константой MAX_TOKEN, определенной в модуле sttypes.h.

Добавляем в модуль sttoken.h новый класс ниже класса sttoken:

```
// класс потока токенов
class sttokens {
    // массив токенов
    sttoken ST[1 + MAX_TOKEN];
public:
    // счетчик токенов
    int count;
    // конструктор
    sttokens() {
        count = 0;
    }
};
```

Элементы данных этого класса — это массив токенов и счетчик, показывающий текущее количество токенов. Конструктор класса устанавливает счетчик токенов в начальное нулевое значение.

Описываем методы класса.

Метод, добавляющий новый токен, возвращает ноль, если массив заполнен, или номер токена, который был «добавлен».

Счет токенов при этом ведется от единицы:

```
// добавляет токен и возвращает его индекс
int add(STTokenType tt) {
    if (index < 1 || index > count) assert(0);
    ST[++count].tt = tt;
    return count;
}
```

Метод, возвращающий токен — это операция индексирования:

```
// возвращает указанный токен
sttoken & operator [] (int index) {
    if (index > MAX_TOKEN) index = 0;
    return ST[index];
}
```

Добавим также метод для записи токенов в текстовый файл для визуального контроля результата лексического анализа:

```
// запись в текстовый файл
void print(FILE * f) {
    for (int i = 1; i <= count; i++) {
        fprintf(f, "%d ", i);
        switch (ST[i].tt) {
        case TOK_UNKNOWN: fprintf(f, "UNKNOWN"); break;
        case TOK_EOT: fprintf(f, "EOT"); break;
        . . .
        . . .
        . . .
        . . .
        . . .
        . . .
        . . .
        . . .
        . . .
        case TOK_ID:      fprintf(f, "ID"); break;
        case TOK_EQ:      fprintf(f, "EQ"); break;
        case TOK_I4:      fprintf(f, "I4"); break;
        case TOK_LP:      fprintf(f, "LP"); break;
        case TOK_RP:      fprintf(f, "RP"); break;
        default: fprintf(f, "UNEXPECTED");
        }
        fprintf(f, " %d", ST[i].tlin);
        fprintf(f, " %d", ST[i].tpos);
        fprintf(f, " %d", ST[i].int_val);
        fprintf(f, " %.10f", ST[i].dbl_val);
        fprintf(f, " \"%s\"\n", ST[i].str_val);
    }
}
```

Здесь нужно будет добавить строки case для всех токенов языка. Это немного нудная часть работы, при выполнении которой нужно быть внимательным, чтобы не пропустить ни одного токена. Если в дальнейшем изменится набор токенов, метод нужно будет скорректировать. На всякий случай метод записывает UNEXPECTED, если токен не задан.

5.3. Тестирование классов токена

Включаем классы токена в модуль SIMPLET.cpp:

```
// SIMPLET.cpp
#include "stdafx.h"
#include "sttypes.h"
#include "sttoken.h"
#include "lexan.h"
```

Перейдем в модуль lexan.h.

Описываем переменную для таблицы токенов в классе lexan:

```
    // токены
    sttokens toks;
private:
    // буфер текста
    char * m_buff;
    // рабочий (текущий) символ
    char m_s;
```

Нам нужен метод для добавления токена, описываем его в классе после метода parse:

```
    // добавляет токен
    int tok_add(STTokenType tt) {
        int tok = toks.add(tt);
        if (tok == 0) {
            // токен не добавлен
            message_out("Tokens overflow");
            return 0;
        }
        return tok;
    } // tok_add
```

На данном этапе мы можем установить для токена только тип, других атрибутов у нас пока нет, они будут сформированы позднее.

Мы можем добавить сейчас заключительный токен, указывающий на конец потока токенов, а также токен перевода строки, если такой задан языком. При добавлении токенов нужно проверять, был токен добавлен или нет, и в случае неудачи завершать работу анализатора:

```
            case ENDOT: // успешное завершение
                // добавляем завершающий токен
                if (!tok_add(TOK_EOT)) return 0;
                message_out("Success");
                return 1;
            case WH_LF: // перевод строки
                if (!tok_add(TOK_LF)) return 0;
                move_next();
                m_blin++;
                m_bpos = 0;
                break;
```

В класс lexan нужно добавить метод для записи файла токенов:

```
    // выводит поток токенов в текстовый файл
    void print(char * path) {
        FILE * f = fopen(path, "w");
        if (!f) return;
        toks.print(f);
        fclose(f);
    }
```

В основной функции нужно добавить вызов этого метода:

```
    // лексический анализатор
    lexan lex(buff);
    // разбор текста
    int res = lex.parse();
    // вывод потока токенов в файл
    lex.print("toks.txt");
    delete [] buff;
}
```

После запуска проекта нужно убедиться, что формируется файл потока токенов в виде текста, имеющий следующий примерный вид:

```
1 EOT 0 0 0 0.0000000000 ""
```

6. Работа 6. «Поток токенов»

Цель: Формирование атрибутов токена.

Токены гораздо легче обнаружить, чем сформировать их атрибуты. Здесь придется выполнить много вычислительной работы, чтобы получить как строковое, так и числовое значение токена. К сожалению, без этой работы не обойтись, иначе в дальнейшем мы ничего практического сделать не сможем.

6.1. Подготовка к формированию токенов

Для формирования атрибутов токена нужны дополнительные элементы данных: буфер для текста лексемы, указатель на положение в нем, переменные для числового значения лексемы и положение лексемы.

Объявляем их в классе lexan:

```
// буфер лексемы
char buff[2 + MAX_TOKEN_BUFF];
// указатель положения в буфере лексемы
int buff_pos;
// буфер для приема целого числа
__int64 number;
// буфер для приема вещественного числа
double fraction;
// начало лексемы - строка
int tlin;
// начало лексемы - положение
int tpos;
```

Для подготовки к приему очередной лексемы атрибуты должны быть очищены, поэтому после метода parse объявляем метод для подготовки:

```
// подготавливает анализатор к приему токена
void prepare_tok() {
} // prepare_tok
```

Подготовка заключается в очистке переменных. Кроме этого, мы запишем также текущие значения положения во входном тексте (иначе говоря, положение первого символа токена):

```
void prepare_tok() {
    for (int i = 0; i < (2 + MAX_TOKEN_BUFF); i++) buff[i] = 0;
    buff_pos = 0;
    number = 0;
    fraction = 0;
    tlin = m_blin;
    tpos = m_bpos;
} // prepare_tok
```

Переходим в начало цикла предварительного разбора и вызываем метод для подготовки к приему токена:

```
    // цикл предварительного разбора
    while (1) {
        // подготавливаемся к приему токена
        prepare_tok();
        // разбираем преобразованный символ
        switch (TOT[m_s]) {
```

Атрибуты токена должны быть записаны в токен, поэтому нам нужно теперь изменить функцию для добавления токена tok_add:

```
// добавляет токен
int tok_add(STTokenType tt) {
    int tok = toks.add(tt);
    if (tok == 0) {
        // токен не добавлен
        message_out("Tokens overflow");
        return 0;
    }
    // запись атрибутов токена
    strcpy(toks[tok].str_val, buff);
    toks[tok].int_val = (int)number;
    toks[tok].dbl_val = fraction;
    toks[tok].tlin = tlin;
    toks[tok].tpos = tpos;
    return tok;
} // tok_add
```

6.2. Формирование токена «идентификатор»

Начнем формирование токенов с приема токена «идентификатор», как самого простого. Тем не менее, не все так просто, как мы сейчас увидим. Для этого токена достаточно иметь только строковое значение.

Мы принимаем символы идентификатора один за другим и должны записывать их в буфер лексемы. Однако есть два ограничения, которые мы должны учитывать.

Первое ограничение связано с размером буфера в классе токена. Оно ограничивает не только длину идентификатора, но и размер строкового литерала. В связи с этим при записи символа нужно проверять текущее значение указателя buff_pos.

Второе ограничение заключается в ограничении длины идентификатора явным образом (помимо неявного ограничения размером буфера).

Программист может придумать для своих целей идентификатор произвольной длины, например, 100 символов. Мы можем увеличить размер буфера, и это благотворно скажется на строковых литералах, но нельзя задавать длину идентификатора непомерно большой, потому что идентификаторы записываются в таблицу символов, в которой в дальнейшем выполняется поиск, и чем длиннее идентификаторы, тем больше время их поиска и, как следствие, меньше производительность транслятора.

Поэтому для всех без исключения трансляторов устанавливается так называемая «распознаваемая длина идентификатора». Это та длина, которая может быть записана в таблицу символов. При этом некоторые идентификаторы могут стать одинаковыми, если программист не знает об ограничении. Например, если распознаваемая длина идентификатора равна 8 символам, то два идентификатора «_12345678» и «_12345670» окажутся одинаковыми с точки зрения транслятора, хотя лексический анализатор примет их целиком, потому что с точки зрения формальных языков длина идентификатора не имеет значения.

Для сравнения, для языка Borland C++ распознаваемая длина идентификатора равна 32 символам, для Microsoft Visual C++ 6.0 — 247 символам, для языка пакетных файлов MS-DOS 6 — 8 символам, для нашего транслятора она равна константе MAX_ID (8 символов), которая определена в модуле sttypes.h.

Еще одно замечание, касающееся идентификаторов. Некоторые языки программирования являются регистро-зависимыми и с ними все просто.

Для регистро-независимых языков нужно выполнять перевод всех символов идентификаторов в какой-то один регистр, либо в нижний, либо в верхний. Мы будем выполнять перевод символов в нижний регистр. Это нужно для того, чтобы потом было проще выполнять сравнение.

Метод get_ident, принимающий идентификатор.

Во-первых, разделим разбор букв и цифр со знаками подчеркивания, потому что только буквы можно переводить в другой регистр методом, который мы используем, а мы будем использовать операцию m_s | 32.

Во-вторых, чтобы быстрее сравнить текущую длину с допустимой длиной, вычислим допустимую длину в начале метода:

```
STTokenType get_ident() {
    // максимальная длина идентификатора
    int max = (MAX_ID < MAX_TOKEN_BUFF) ? MAX_ID : MAX_TOKEN_BUFF;
    while (1) {
        switch (TOT[m_s]) {
        case ALPHA:
            move_next();
            break;
        case DIGIT: case UNDER:
            move_next();
            break;
        default:
            return TOK_ID;
        }
    }
}
```

Теперь перед вызовами метода move_next нужно добавить проверку записанной длины и запись символов. Для случая ALPHA это может выглядеть, например, так:

```
case ALPHA:
    if (buff_pos < max) {
        buff[buff_pos++] = (m_s | 32);
    }
    move_next();
```

Для случаев DIGIT и UNDER не нужно выполнять перевод регистра:

```
case DIGIT: case UNDER:
    if (buff_pos < max) {
        buff[buff_pos++] = m_s;
    }
    move_next();
```

Перейдем в цикл предварительного разбора и добавим токен идентификатора следующим образом:

```
case ALPHA: // идентификатор
    get_ident();
    if (!tok_add(TOK_ID)) return;
    break;
```

Для проверки редактируем текст 1.st, например, так:

```
LONG_67890 123456•
```

Запускаем программу, смотрим файл toks.txt. У меня он имеет вид:

```
1 ID 1 0 0 0.0000000000 "long_6789"
2 EOT 1 17 0 0.0000000000 ""
```

Видим, что перевод в нижний регистр осуществлен, длина идентификатора ограничена восемью символами, что и нужно было получить. Положение токена в тексте также задано (первая строка, нулевая позиция).

6.3. Формирование токенов ключевых слов

Мы используем для определения токенов ключевых слов специальный метод is_keyword, который возвращает либо константу ключевого слова, либо константу TOK_ID:

```
// разбирает ключевые слова
STTokenType is_keyword() {
    return TOK_ID;
} // is_keyword
```

Собственно разбор заключается в последовательных сравнениях буфера лексемы с ключевыми словами. Нужно учитывать, что в буфере находится идентификатор, который либо переведен в нижний регистр, либо записан как есть (для языка Си). Поэтому сравнение идет бинарное, то есть точное. Пример разбора для моего языка:

```
if (!strcmp(buff, "dim")) return TOK_DIM;
else if (!strcmp(buff, "long")) return TOK_LONG;
else if (!strcmp(buff, "if")) return TOK_IF;
else if (!strcmp(buff, "then")) return TOK_THEN;
else if (!strcmp(buff, "else")) return TOK_ELSE;
else if (!strcmp(buff, "end")) return TOK_END;
else if (!strcmp(buff, "while")) return TOK_WHILE;
else if (!strcmp(buff, "print")) return TOK_PRINT;
return TOK_ID;
```

Ни одно ключевое слово не должно быть пропущено, в том числе нужно не забыть слова, являющиеся операциями, такие, как AND.

Теперь этот метод нужно вызвать в цикле предварительного разбора после того, как идентификатор принят, и записать соответствующий токен, например:

```
case ALPHA: // идентификатор
    get_ident();
    fa_result = is_keyword();
    if (!tok_add(fa_result)) return 0;
    break;
```

Остается протестировать, как анализатор распознает ключевые слова и идентификаторы. Тестирующий текст должен содержать все ключевые слова входного языка, а также несколько идентификаторов, например:

```
DI DiM LON LonG if IF THE THEn end EnD enD END
WHIL WHILe printa print PRinT AND AN
```

Как уже говорилось раньше, такой способ выявления ключевых слов проще реализовать и, самое главное, проще редактировать, так как добавление или удаление ключевого слова заключается в добавлении или удалении строки.

6.4. Формирование токена «целочисленная константа»

С точки зрения формальной грамматики, целое число есть просто бесконечная последовательность цифр, и программист может придумать какое угодно число, например, 4294967297.

Рассмотрим два примера:

```
int x = 4294967297;
double y = 4294967297;
```

Эти строчки можно вставить в начало основной функции и проверить, каков будет результат. Число x получит значение единицы, а число y получит значение целочисленной константы. В чем здесь фокус?

Рассмотрим целое число. Число типа int в нашем случае имеет размер в 32 бита, а число 4294967297 в двоичном представлении имеет 33 бита:

```
1 00000000 00000000 00000000 00000001
```

Иначе говоря, это число состоит из 2^{32} и единицы. Старший бит, значение которого равно 2^{32}, не уместился в разрядную сетку, осталась единица. Произошло так называемое переполнение разрядной сетки.

Во втором случае, когда переменная вещественная, переполнения произойти не может, — в вещественных числах переполняется порядок, а не мантисса, и имеет значение только количество значащих цифр. При преобразовании константы в вещественное значение можно потерять только последние цифры мантиссы, потому что они могут не уместиться в разрядную сетку мантиссы, но если порядок не переполнился, число будет записано правильно, но с определенной точностью. Именно поэтому вещественные числа называют неточными.

Мы разбираем целочисленную константу, которая должна давать разные значения в разных случаях. Поэтому при формировании токена важно иметь как целочисленное значение константы, так и вещественное.

Другая сторона проблемы — как должен поступить транслятор, если обнаружит переполнение разрядной сетки? Ответ достаточно сложный. В языке Си переполнение не фиксируется, и константа просто усекается с получением непредсказуемого результата. В языках Pascal и Basic попытка присвоить целому числу значение, выходящее за диапазон числа приведет к формированию ошибки. Однако чтобы знать диапазон числа, нужно иметь информацию о том, каков размер нужен в данном контексте. Во время лексического анализа контекст недоступен, поэтому обычно константы принимаются максимально допустимого размера, а проверка переполнения при присвоении выполняется при присвоении, то есть в синтаксическом анализаторе.

Лексический анализатор может выдавать ошибку при переполнении разрядной сетки целочисленной константы максимального размера.

Тогда, чтобы записать числовую константу, например, такую:

```
12345678901234567890
```

нужно указывать, что она имеет вещественный тип, например:

```
int y = 12345678901234567890.;
```

Зафиксировать ошибку или предупреждение в этом случае можно в синтаксическом анализаторе.

В случае десятичных чисел мы не можем просто ограничить число десятичных цифр и при этом получить правильный результат. В случае шестнадцатеричных записей наоборот, мы можем ограничивать количество цифр, поскольку в шестнадцатеричной системе счисления количество цифр кратно размеру физического представления числа.

Лексический анализатор должен принять все цифры десятичного числа, сформировать числовое значение и определить, допустимо оно или нет, или не определять этого.

Метод get_number, принимающий последовательность цифр.

Будем принимать цифры следующим образом.

Получив цифру в m_s, вычтем из нее 48 (код цифры ноль), и получим собственно число. Умножим число number на 10 и прибавим полученное число. И так до тех пор, пока цифры не закончатся. Чтобы лучше понять, что это за метод, рассмотрим конкретный пример.

Пусть числовой литерал "123". Изначально number равно нулю. Получаем первый знак "1", код которого равен 49. Вычитаем 48 и получаем 1. Умножаем number на 10 и получаем ноль. Прибавляем 1 и получаем 1.

Получаем второй знак "2". Код равен 50, после вычитания 48 остается 2, умножаем number на 10, получаем 10, прибавляем 2, получаем 12. Очевидно, выполнив эти действия для третьего знака, получим число 123.

Записываем этот алгоритм в метод get_numer:

```
case DIGIT:
    (number *= 10) += m_s - 48;
    (fraction *= 10) += double(m_s - 48);
    move_next();
    break;
```

Для тестирования используем следующий текст 1.st:

```
4294967297•
```

Точка остановки на операторе return метода get_ident. Запускаем программу, после остановки проверяем значение переменной number и убеждаемся, что получилась единица. Проверяем значение переменной fraction и убеждаемся, что оно равно значению константы.

Дальнейшие действия зависят от входного языка.

Если язык допускает только целочисленные константы, то изменяем метод get_number:

```
default:
    number = (int)(number & 0xFFFFFFFF);
    return TOK_I4;
```

Далее переходим в цикл предварительного разбора, добавляем токен:

```
case DIGIT: // числовая константа
    get_number();
    if (!tok_add(TOK_I4)) return 0;
    break;
```

Если заданы также и вещественные константы, то изменяем метод is_number вместо метода get_number:

```
STTokenType is_number() {
    get_number();
    number = (int)(number & 0xFFFFFFFF);
    if (m_s == '.') {
```

Затем добавляем токен в цикле предварительного разбора:

```
case DIGIT: // числовая константа
    fa_result = is_number();
    if (fa_result == TOK_I4) {
        // целочисленная константа
        if (!tok_add(TOK_I4)) return 0;
    } else if (fa_result == TOK_R8) {
        // вещественная константа
        if (!tok_add(TOK_R4)) return 0;
```

Что нужно сделать, чтобы транслятор обнаруживал переполнение разрядной сетки? Это тоже не сложно. В том месте, где метод is_number возвращает константу TOK_I4, нужно поставить проверку:

```
if ((number & 0xFFFFFFFF00000000) == 0) {
    return TOK_I4;
} else {
    message_out("Integer number overflow");
    return TOK_UNKNOWN;
}
```

Смысл условия — убедиться, что принятое число целиком содержится в первых четырех байтах из восьми для 64-битного буфера number.

6.5. Разбор недесятичной целочисленной константы

Языки могут задавать также целочисленные константы в шестнадцатеричной или восьмеричной форме. Например, в языке Си константа

```
0xA0 // число 160
```

задает число в шестнадцатеричной форме, а константа

```
020 // число 16
```

задает число в восьмеричной форме.

Если ваш язык определяет одну из подобных форм задания целочисленной константы, нужно обеспечить правильное вычисление ее значения.

Пусть входной язык Си. Разделим прием констант в разных системах счисления, — добавим два новых метода, is_octal и is_hex для вычисления восьмеричной и шестнадцатеричной форм задания соответственно:

```
// разбирает восьмеричную константу
STTokenType is_octal() {
    return TOK_UNKNOWN;
} // is_octal
// разбирает шестнадцатеричную константу
STTokenType is_hex() {
    return TOK_UNKNOWN;
} // is_hex
```

Оба метода могут фиксировать ошибку в случае, если встречается недопустимая цифра. Эти методы должны вызываться в методе is_number:

```
// разбирает числовую константу
STTokenType is_number() {
    if (m_s == '0') {
        move_next();
        if (m_s == '.') {
            move_next();
            return get_fraction();
        } else if ((m_s | 32) == 'x') {
            move_next();
            return is_hex();
        } else {
            return is_octal();
        }
    } else {
        get_number();
    }
    . . .
```

Рассмотрим метод is_octal.

```
STTokenType is_octal() {
    while (1) {
        switch (m_s) {
        case '0': case '1': case '2': case '3':
        case '4': case '5': case '6': case '7':
            (number <<= 3) += m_s - 48;
            move_next();
            break;
        case '8': case '9':
            message_out("Invalid octal number");
            return TOK_UNKNOWN;
        default:
            number = (int)(number & 0xFFFFFFFF);
            fraction = (double)(number & 0xFFFFFFFF);
            return TOK_I4;
        }
    }
}
```

Оператор switch здесь построен иначе. Так, он разбирает не перекодированный символ m_s, а непосредственно сам символ m_s, поскольку не все цифры являются допустимыми. Далее, проверяется, не является ли символ десятичной цифрой 8 или 9. Вдобавок алгоритм вычисления значения умножает число number не на 10, а на 8, поэтому вместо операции умножения должна быть использована операция сдвига влево на три бита, потому что она работает намного быстрее.

Метод is_hex похож на метод is_octal, но есть особенности. Первое — нужна функция для определения того, что текущий символ являются допустимым, типа is_valid_hex. При этом перед тем, как ее вызвать, вероятно, нужно убедиться, что символ является буквой или цифрой. Здесь можно придумать множество вариантов, вплоть до табличного (то есть построить еще одну таблицу HEX типа TOT). Второе — вместо трех сдвигов числа

number нужно выполнить четыре, так как одна шестнадцатеричная цифра соответствует четырем битам.

Дополнительно заметим, что после восьмеричных и шестнадцатеричных констант могут следовать буквенные суффиксы.

6.6. Формирование токена «вещественная константа»

Будем вычислять вещественное значение константы следующим образом. До точки мы принимаем целое число и записываем его в атрибуты number и fraction. Далее, если мы принимаем вещественную часть числа, то принимаем целое число после точки и преобразуем его в вещественную часть, к которой затем прибавляем атрибут number.

Для приема дробной части числа нам потребуется новый метод, так как он должен выполнить преобразование целого числа в дробную часть.

Скопируем метод get_number и переименуем его в get_fraction:

```
// принимает дробную часть
STTokenType get_fraction() {
    while (1) {
        switch (TOT[m_s]) {
        case DIGIT:
            (fraction *= 10) += double(m_s - 48);
            move_next();
            break;
        default:
            return TOK_R8;
        }
    }
} // get_fraction
```

Добавим объявление переменных и очистку переменной fraction:

```
STTokenType get_fraction() {
    int i, j = 0;
    fraction = 0;
    while (1) {
```

При приеме цифры будем ее подсчитывать в переменной j:

```
        case DIGIT:
            (fraction *= 10) += double(m_s - 48);
            j++;
            move_next();
            break;
```

Перед завершением метода поделим fraction на 10 j раз и прибавим к результату атрибут number:

```
        default:
            for (i = 0; i < j; i++) fraction /= 10;
            fraction += number;
            return TOK_R8;
```

В методе is_number заменим вызов get_number на get_fraction:

```
if (m_s == '.') {
    move_next();
    if (TOT[m_s] == DIGIT) {
        get_fraction();
    }
    return TOK_R8;
```

Если у вас определен метод is_real, то в нем также нужно заменить вызов get_number на get_fraction.

В цикле предварительного разбора нужно добавить токен там, где вызывается метод is_number и метод is_real, если он есть:

```
case DIGIT: // числовая константа
        . . .
    } else if (fa_result == TOK_R8) {
        // вещественная константа
        if (!tok_add(TOK_R8)) return;
    } else {
        . . .
case CHDOT: // вещественная константа
    fa_result = is_real();
    if (fa_result == TOK_R8) {
        // вещественная константа
        if (!tok_add(TOK_R8)) return;
    } else {
        . . .
    }
```

Тестируем программу на различных значениях чисел самостоятельно.

6.7. Формирование токена «строковый литерал»

Формирование токена «строковый литерал» предполагает запись значения атрибута buff примерно так же, как это было сделано при формировании токена, но есть особенности. Если входной язык является регистронезависимым, то при приеме символов идентификатора мы переводим буквы в один регистр. При приеме строкового литерала делать это недопустимо. Другая особенность заключается в том, что внутри строкового литерала могут встречаться последовательности из двух кавычек, вместо которых нужно записывать одну.

Добавим в класс lexan вспомогательный метод, который будет записывать в буфер лексемы один символ строковой константы.

Метод проверяет, есть ли еще место в буфере и если есть, то записывает символ и возвращает единицу, иначе формирует сообщение об ошибке и возвращает ноль:

```
// записывает символ строкового литерала
int put_qoute(char s) {
    if (str_pos < MAX_TOKEN_BUFF) {
        buff[str_pos++] = s;
        return 1;
    } else {
        message_out("Quote is to big");
        return 0;
    }
} // put_qoute
```

Пример метода is_qoute, использующий метод put_qoute:

```
// разбирает строковый литерал
STTokenType is_quote() {
    while (1) {
        move_next();
        if (TOT[m_s] == QUOTE) {
            move_next();
            if (TOT[m_s] != QUOTE) {
                return TOK_QUOTE;
            } else if (!put_qoute()) return TOK_UNKNOWN;
        } else if (m_s < ' ') {
            message_out("Unexpected end of file in quote");
            return TOK_UNKNOWN;
        } else if (!put_qoute()) return TOK_UNKNOWN;
    }
} // is_quote
```

В цикле предварительного разбора добавим запись токена:

```
        case QUOTE: // строковая константа
            fa_result = is_quote();
                . . .
                // правильный литерал
                if (!tok_add(TOK_QUOTE)) return;
            }
```

Для тестирования можно использовать следующий текст 1.st:

```
"aBc"""de"
```

Разбор этого текста выдаст ошибку.

6.8. Формирование токенов «операция» и «пунктуатор»

Для операторов и пунктуаторов нужно только добавить токен:

```
        case OPERA: // операция, пунктуатор
            fa_result = get_opera();
            if (fa_result == TOK_UNKNOWN) {
                // ошибка разработчика
                message_out("Operator parse error");
                return 0;
            }
            if (!tok_add(fa_result)) return;
            break;
```

Таким образом, сейчас должны быть разобраны все токены, предусмотренные входным языком.

7. Работа 7. «Подготовка к синтаксическому анализу»

Цель: класс для синтаксического анализатора.
Задачи:
- подготовить классы синтаксического анализатора.

7.1. Класс синтаксического анализатора

Добавляем в проект заголовочный модуль syntaxan.h. Начальное описание класса синтаксического анализатора может иметь следующий вид:

```
// syntaxan.h
#pragma once
struct syntaxan {
    syntaxan() {
    }
private:
};
```

Включаем этот модуль в модуль SIMPLET.cpp:

```
// SIMPLET.cpp
#include "stdafx.h"
#include "sttypes.h"
#include "sttoken.h"
#include "lexan.h"
#include "syntaxan.h"
```

В конструктор класса syntaxan нужно передать поток токенов, который запоминается в локальной переменной:

```
struct syntaxan {
    syntaxan(sttokens toks) {
        m_toks = toks;
    }
private:
    // токены
    sttokens toks;
};
```

Для выполнения собственно синтаксического анализа в класс нужно добавить метод parse:

```
struct syntaxan {
    syntaxan(sttokens toks) {
        m_toks = toks;
    }
    // разбор потока токенов
    void parse() {
    }
private:
    // токены
    sttokens m_toks;
};
```

В модуле SIMPLET.cpp в функции main нужно сконструировать объект класса syntaxan, передать ему объект класса sttokens из класса lexan и вызвать метод parse:

```
    // вывод потока токенов в файл
    lex.print("toks.txt");
    // синтаксический анализатор
    syntaxan syn(lex.toks);
    // синтаксический разбор
    syn.parse();
    delete [] buff;
}
```

7.2. Класс рабочего стека

Анализ КС-грамматик выполняется с помощью МП-автомата, который требует наличия магазина, то есть стека. Нам нужен стек.

Добавляем в проект новый заголовочный модуль ststack.h.

Начальное содержание файла примерно следующее:

```
//ststack.h
#pragma once
template <class T, int N>
struct ststack {
private:
};
```

Мы будем конструировать стек с заданной глубиной.

Глубина задана константой MAX_STACK в модуле sttypes.h.

Элементы данных — это массив элементов T размером N и счетчик элементов:

```
template <class T, int N>
struct ststack {
    // счетчик элементов
    int count;
private:
    // массив элементов
    T ST[N];
};
```

Опишем функцию, которая очищает стек:

```
struct ststack {
    // очистка
    void reset() {
        count = 0;
        for (int i = 0; i < N; i++) ST[i] = (T)-1;
    }
private:
    . . .
};
```

Конструктор просто очищает элементы данных класса:

```
template <class T, int N>
struct ststack {
    // счетчик элементов
    int count;
    // конструктор
    ststack() {
        reset();
    }
private:
    // массив элементов
    T ST[N];
};
```

Описываем метод для проталкивания элемента в стек:

```
    // проталкивание
    int push(T tok) {
        if (count < N) {
            ST[count++] = tok;
            return count;
        } else {
            assert(0);
            return 0;
        }
    }
```

В случае, если достигнута максимальная глубина стека, программа остановится из-за оператора assert, и мы увидим некоторое сообщение.

Описываем метод, возвращающий элемент на вершине стека:

```
    // выталкивание
    T pop() {
        if (count < 1) {
            assert(0);
            count = 0;
            return (T)-1;
        } else {
            return ST[--count];
        }
    }
```

Описываем метод, выталкивающий из стека заданное количество элементов:

```
    // выталкивание n элементов
    T pop(int n) {
        count -= n;
        if (count < 1) {
            assert(0);
            count = 0;
            return (T)-1;
        }
        return ST[count - 1];
    }
```

Следующий метод возвращает элемент на вершине стека:

```
// вершина
T top() {
    if (count < 1) {
        assert(0);
        return (T)-1;
    } else {
        return ST[count - 1];
    }
}
```

Сейчас нужно подключить модуль стека в модуле SIMPLET.cpp:

```
// SIMPLET.cpp
#include "stdafx.h"
#include "sttypes.h"
#include "sttoken.h"
#include "lexan.h"
#include "ststack.h"
#include "syntaxan.h"
```

В классе синтаксического анализатора создадим рабочий стек:

```
struct syntaxan {
    syntaxan(sttokens toks) {
        m_toks = toks;
    }
    // синтаксический разбор
    void parse() {
    }
private:
    // токены
    sttokens m_toks;
    // рабочий стек
    ststack<STTokenType, MAX_STACK> sta;
};
```

На стеке будут находиться элементы перечисления STTokenType, которые с точки зрения теории соответствуют магазинному алфавиту, при этом символу ⊥ соответствует константа TOK_EOT.

8. Работа 8. «Анализатор LL(1)»

Цель: построение синтаксического анализатора LL(1)
Задачи:
- формирование управляющей синтаксической таблицы;
- проектирование вспомогательных методов класса syntaxan;
- моделирование МП-автомата с подбором альтернатив.

8.1. Управляющая синтаксическая таблица

Анализ методом LL(1) предполагает построение управляющей синтаксической таблицы. Для этого откройте программу ReVoL SYNAX, откройте разработанную грамматику, выполните ее анализ. Далее перейдите на вкладку LL(1) и выполните анализ. Если компилятор подтвердит, что грамматика является LL(1), то выполните экспорт этой вкладки.

Добавим в проект новый заголовочный модуль syntab.h:

```
// syntab.h

#pragma once
```

Скопируем сгенерированный программой SYNAX текст и вставим в новый модуль. Модуль syntab.h включим в модуль syntaxan.h:

```
// syntaxan.h
#pragma once
// синтаксическая таблица
#include "syntab.h"
struct syntaxan {
```

Убедимся, что проект не содержит ошибок.

Изучим модуль syntab.h.

Сначала в модуле определена константа, задающая максимальную длину правила. В случае необходимости ее значение можно исправлять.

Далее расположена таблица RULE, содержащая правила грамматики. В ней используются константы перечисления STTokenType. Таблица содержит только правые части правил. Нумерация правил начинается с единицы, значение «0» в синтаксической таблице зарезервировано для ячеек, указывающих на ошибку.

Далее сформирована константа MAX_RULE, указывающая на максимальный номер правила (соответствует количеству правил).

Далее сформирован массив RLEN, содержащий длины всех правил. При построении трансляционной грамматики в правила будут добавляться так называемые операционные символы, и длина правил будет изменяться.

Поэтому для вычисления фактической длины правил далее размещена функция get_rule_len. Эта функция должна быть вызвана в начале работы синтаксического анализатора.

Заметим, что в таблице правил любое правило заканчивается токеном TOK_EOT, указывающим на конец правила. Фактическое значение этого токена равно нулю, и этот токен всегда должен оставаться в конце.

Далее в модуле определены константы ACC и ESC, задающие коды таблицы для действий ACCEPT и ESCAPE соответственно. Первое действие обозначает допуск входной цепочки, второе обозначает шаг «выброс» алгоритма с подбором альтернатив.

Третья константа, START, задает значение целевого символа синтаксической грамматики.

Наконец, в конце модуля располагается управляющая синтаксическая таблица SYNTA. Она достаточно хорошо документирована, чтобы можно было легко ориентироваться в ней. Если грамматика не является LL(1), синтаксическая таблица формируется компилятором SYNAX таким образом, чтобы ее нельзя было скомпилировать.

Заметим, что нулевые значения в синтаксической таблице обозначают ошибки разбора входной цепочки. В эти ячейки могут быть записаны отрицательные значения, указывающие на сообщение об ошибке.

8.2. Подготовка к разбору

Перейдем в модуль syntaxan.h.

Для вывода сообщений анализатора на терминал добавим в класс новый метод, его можно скопировать из класса lexan:

```
// разбор потока токенов
void parse() {
} // parse
// выводит сообщение
void message_out(char * message) {
    printf("syntaxan[%d:%d] %s.\n", m_tlin, m_tpos, message);
} // message_out
```

Метод использует переменные m_tlin и m_tpos, которые не определены, поэтому их нужно описать в закрытой секции класса:

```
// текущая строка
int m_tlin;
// текущая позиция
int m_tpos;
```

Эти переменные получат свои значения при чтении очередного токена при помощи вспомогательного метода next_token:

```
void message_out(char * message) {
    printf("syntaxan[%d:%d] %s.\n", m_tlin, m_tpos, message);
} // message_out
// считывает очередной токен
void next_token() {
} // next_token
```

Кроме позиции токена, токен содержит также собственно значение токена, строковое значение и числовые значения. Для управления токенами нужно знать общее количество токенов и номер текущего токена. Поэтому в закрытой секции класса описываем следующие переменные:

```
// значение текущего токена
STTokenType m_token;
// строковое значение токена
char * m_str_val;
// целочисленное значение токена
int m_int_val;
// вещественное значение токена
double m_dbl_val;
// номер текущего токена
int tok_curr;
// количество токенов
int tok_count;
```

Теперь можно определить метод next_token:

```
void next_token() {
    if ( tok_curr > tok_count ) {
        tok_curr = tok_count;
    }
    m_token = m_toks[tok_curr].tt;
    m_tlin = m_toks[tok_curr].tlin;
    m_tpos = m_toks[tok_curr].tpos;
    m_str_val = m_toks[tok_curr].str_val;
    m_int_val = m_toks[tok_curr].int_val;
    m_dbl_val = m_toks[tok_curr].dbl_val;
    tok_curr++;
} // next_token
```

Конструктор класса syntaxan должен определить количество токенов:

```
// конструктор
syntaxan(sttokens toks) {
    m_toks = toks;
    tok_count = m_toks.count;
}
```

Метод parse проверяет количество токенов:

```
void parse() {
    if (tok_count < 2) {
        message_out("Nothing to parse");
        return;
    }
} // parse
```

Собственно МП-автомат будем реализовывать как отдельный метод:

```
} // parse
// МП-автомат
int parse_LL() {
    return 1;
} // parse_LL
```

Этот метод нужно вызвать в методе parse:

```
void parse() {
    if (tok_count < 2) {
        message_out("Nothing to parse.");
        return;
    }
    // номер первого токена
    tok_curr = 1;
    // считываем первый токен
    next_token();
    // МП-автомат
    int res = parse_LL();
    if (res == 1) {
        message_out("Success");
    }
} // parse
```

Переходим в метод parse_LL.
Для конструирования МП-автомата требуются переменные.
Первая переменная предотвращает зацикливание:

```
// МП-автомат
int parse_LL() {
    // счетчик итераций автомата
    int it_counter = 0;
    return 1;
} // parse_LL
```

Далее, для формирования циклов требуются переменные:

```
    // счетчик итераций автомата
    int it_counter = 0;
    // итератор и счетчик
    int i = 0, n = 0;
```

Затем нам нужно будет считывать состояние на стеке, а во время работы автомата проверять, какие символы выталкиваются со стека:

```
    // состояние на стеке
    STTokenType s = TOK_EOT;
    // вспомогательные символы
    STTokenType at = TOK_EOT, bt = TOK_EOT;
```

Для чтения значения в синтаксической таблице нужна переменная:

```
    // значение в таблице
    stack_t t = 0;
```

Перед началом работы МП-автомата нужно скорректировать длину правил. В случае, если константы ST_MAX_RULE_LEN окажется недостаточно для описания длины, функция корректировки возвращает значение меньше либо равное нулю:

```
    stack_t t = 0;
    // уточняем длину правил
    if ((n = get_rule_len()) <= 0) {
        // длина какого-то правила превышает размер массива RULE
        // требуемая длина равна n, если n > 0
        assert(0);
        message_out("Internal Grammar Error: ST_MAX_RULE_LEN");
        return 0;
    }
```

8.3. Моделирование МП-автомата с подбором альтернатив

В начале работы МП-автомата с подбором альтернатив на стек должны быть записаны два символа — маркер дна стека и целевой символ. Если внимательно изучить синтаксическую таблицу, то можно увидеть, что в качестве маркера в программе используется символ TOK_EOT.

Метод parse_LL, продолжаем наполнение метода:

```
    if ((n = get_rule_len()) <= 0) {
        . . .
    }
    // дно рабочего стека
    sta.push(TOK_EOT);
    // аксиома грамматики на рабочем стеке
    sta.push(START);
```

Затем формируем рабочий цикл МП-автомата:

```
    // рабочий цикл автомата LL
    while ( 1 ) {
    }
    return 1;
} // parse_LL
```

Цикл бесконечный, поэтому нужен предохранитель от зацикливания.

Для описания предохранителя в начале модуля syntaxan.h описываем константу максимального количества циклов:

```
// максимальное количество шагов МП-автомата
#define ST_MAX_PARSE_COUNT 1000
```

Теперь в рабочем цикле описываем предохраняющую конструкцию:

```
    // рабочий цикл автомата LL
    while ( 1 ) {
        // проверка на зацикливание
        if (it_counter++ > ST_MAX_PARSE_COUNT) {
            message_out("Internal Deadlock: ST_MAX_PARSE_COUNT");
            return 0;
        }
    }
    return 1;
} // parse_LL
```

Следите за тем, чтобы предохранитель всегда оставался в конце рабочего цикла. Его положение не существенно важно, но в конце цикла он не будет мешать чтению текста собственно автомата.

В рабочем цикле автомат считывает значение в таблице на основании текущего токена и символа на стеке. Текущий токен в программе обозначен переменной m_token, а символ на стеке нужно прочитать:

```
while ( 1 ) {
    // состояние на стеке
    s = sta.top();
    // значение в таблице
    t = SYNTA[s][m_token];
    // проверка на зацикливание
        . . .
    }
}
```

Далее, в зависимости от считанного значения автомат выполняет одно из четырех действий: либо принимает входной текст, либо отвергает его, либо выполняет один из шагов — выбор альтернативы или выброс.

Если значение в таблице меньше или равно нулю, автомат фиксирует ошибку. При положительном значении, не превышающем максимальный номер правила, автомат выполняет выбор альтернативы.

Заметим, что на самом деле автомат будет выполнять не четыре действия, а пять. Пятое действие выполняется в случае, если считанное значение положительное, но не соответствует никакому правилу. Оно может возникнуть в результате ручной правки синтаксической таблицы.

Описываем разбор значения t, например, так:

```
while ( 1 ) {
    // состояние на стеке
    s = sta.top();
    // значение в таблице
    t = SYNTA[s][m_token];
    // анализ состояния
    if (t <= 0) {
        // ОШИБКА
    } else if (t == ACC) {
        // ДОПУСК
    } else if (t == ESC) {
        // ВЫБРОС
    } else if (t <= MAX_RULE) {
        // ВЫБОР АЛЬТЕРНАТИВЫ
    } else {
        // ОШИБКА СИИНТАКСИЧЕСКОЙ ТАБЛИЦЫ
    }
    // проверка на зацикливание
    if (it_counter++ > ST_MAX_PARSE_COUNT) {
        . . .
    }
}
```

Остается только описать соответствующие действия. По порядку.

В случае ошибки автомат формирует сообщение «Failure» и возвращает ложь:

```
if (t <= 0) {
    // ОШИБКА
    message_out("Failure");
    return 0;
} else if (ACC == t) {
```

В случае допуска выходим из цикла, и метод вернет истину:

```
} else if (ACC == t) {
    // ДОПУСК
    // выходим из цикла
    break;
} else if (ESC == t) {
```

В случае действия «выброс» выталкиваем с рабочего стека терминал и считываем следующий токен:

```
} else if (ESC == t) {
    // ВЫБРОС
    // выталкиваем терминал из рабочего стека
    at = sta.pop();
    // следующий токен
    next_token();
} else if (t <= MAX_RULE) {
```

В случае ошибки в синтаксической таблице выводим сообщение и возвращаем ложь:

```
} else {
    // ОШИБКА СИИНТАКСИЧЕСКОЙ ТАБЛИЦЫ
    message_out("Internal Syn Table Error: MAX_RULE");
    return 0;
}
```

Остается описать выбор альтернативы. На альтернативу указывает считанное из таблицы значение t, равное номеру правила, которое нужно применить в данном случае.

Выбор выполняется следующим образом. С рабочего стека снимается находящийся там нетерминал, и вместо него на стек записывается правая часть правила t так, чтобы первый символ правой части правила оказался на вершине стека (правило записывается в обратном порядке).

Снимаем со стека нетерминал:

```
} else if (t <= MAX_RULE) {
    // ВЫБОР АЛЬТЕРНАТИВЫ
    // выталкиваем нетерминал из стека
    at = sta.pop();
} else {
```

Нетерминал выталкивается в переменную at для контроля.

Для записи на стек правила нужно знать его длину. Длина считывается в переменную n для контроля во время отладки. Поэтому следующий оператор на самом деле не нужен, и после отладки его можно удалить вместе с комментарием:

```
// длина правила t
n = RLEN[t]; // [ оператор для контроля ]
```

Далее формируем цикл для проталкивания правила на стек в обратном порядке. В цикле используется переменная bt, в которую записывается очередной символ правила. Это также оператор для контроля:

```
// проталкиваем на стек правило t в обратном порядке
for (i = RLEN[t] - 1; i >= 0; i--) {
    bt = RULE[t][i]; // [ оператор для контроля ]
    sta.push(RULE[t][i]);
}
```

И это все. МП-автомат с подбором альтернатив смоделирован и на данном этапе он проверяет допустимость входного текста в смысле правильного порядка следования токенов. Формируем синтаксически правильные тексты 1.st, получаем таблицы токенов и тестируем анализатор.

В заключение можно сформировать последовательность правил.

Для этого скопируйте в папку проекта файл studeque.h и подключите его к модулю syntaxan.h.

В конце класса syntaxan объявите стек для правил:

```
// стек номеров правил
udeque m_rules;
```

В случае подбора альтернативы (в конце) запишите номер правила:

```
} else if (t <= MAX_RULE) {
    . . .
    // записываем правило
    m_rules.push_left(t);
} else {
```

Наконец, в методе parse добавьте вывод номеров правил:

```
int res = parse_LL();
if (res == 1) {
    message_out("Success");
}
// номера правил
stack_t r;
int n = m_rules.count();
for (int i = 0; i < n; i++) {
    r = m_rules.pop_right();
    printf("%d\n", r);
}
```

9. Работа 9. «Классы ПОЛИЗ»

Цель: разработка классов для формирования ленты ПОЛИЗ.
Задачи:
- формирование класса элемента ленты ПОЛИЗ;
- формирование класса ленты ПОЛИЗ.

9.1. ПОЛИЗ

ПОЛИЗ расшифровывается как «польская инверсная запись». Она формируется следующим образом.

Если есть некий оператор θ с n операндами α_1, α_2, …, α_n, то на ленту ПОЛИЗ записываются сначала элементы α_1, α_2, …, α_n, а затем элемент θ.

Например, оператор a=a+1 запишется на ленту в следующем порядке:
a a 1 + :=

При чтении ленты элементы данных записываются в стек, а операции выполняются над элементами стека. Поэтому сначала в стек будут помещены a, a и 1, затем операция «плюс» считает два элемента со стека, сложит их и запишет результат на стек, а последующая операция «присвоить» извлечет два элемента со стека, присвоит второму извлеченному элементу значение первого и запишет результат на стек.

Таким образом может быть записана любая операция, например, условный оператор, оператор цикла, вызов функции, описание функции и т.п., поэтому ПОЛИЗ может быть использована как способ записи внутреннего промежуточного представления программы. Цель нашего синтаксического анализатора — сформировать это представление.

По ходу разбора токенов входного текста синтаксический анализатор должен записывать на ленту ПОЛИЗ операнды и операции. Элементами ленты ПОЛИЗ могут быть не только операнды и операции, но также переходы к другим элементам ленты и метки для этих переходов.

Для формирования класса элемента ленты это учитывается с помощью перечисления STTokenType, которое должно содержать константы для описания всех возможных типов элементов. Элемент ленты может содержать дополнительную информацию. Например, если на ленте записана числовая константа, элемент должен содержать ее значение, а если на ленту записывается идентификатор, то элемент должен иметь строковое представление этого идентификатора или его индекс в таблице символов.

9.2. Класс элемента ПОЛИЗ

Для создания классов ленты ПОЛИЗ используем имеющиеся классы токена, так как эти классы похожи. Найдем в папке C:\SIMPLET\SIMPLET файл sttoken.h и получим его копию ststrip.h.

При помощи меню Project — Add Existing Item добавляем в проект новый модуль ststrip.h.

Произведем в новом модуле замены:
sttoken заменим на exstel,
sttokens заменим на ststrip,
MAX_TOKEN заменим на MAX_RPN.
Добавим в класс exstel копирование строкового значения:

```
// копирование идентификатора
void set_str(char * s) {
    strcpy(str_val, s);
}
```

9.3. Класс ленты ПОЛИЗ

Далее в модуле следует класс ленты ПОЛИЗ ststrip. Добавим в элементы данных еще один — счетчик меток:

```
template <int N>
class ststrip {
    // лента
    exstel ST[N];
    // счетчик меток
    int label;
};
```

Конструктор очищает счетчики:

```
public:
    // счетчик токенов
    int count;
    // конструктор
    ststrip() {
        count = 0;
        label = 0;
    }
```

Следующий метод генерирует новый идентификатор для метки:

```
// возвращает новый идентификатор метки
int new_label() {
    return ++label;
}
```

Все элементы ленты можно подразделить на следующие виды:
- управляющие коды, такие, как переход или определение метки;
- метки, нужные для переходов по ленте;
- идентификаторы, представляющие собой переменные программы;
- константы.

Следующие методы упрощают запись этих типов элементов. Они содержат предопределенные для ПОЛИЗ типы элементов, которые сейчас нужно добавить в модуль sttypes.h в перечисление STTokenType:

```
// Символы
typedef enum _STTokenType {
    . . .
    . . .
    // символы генератора кода
    OUT_ID,
    OUT_I4,
    OUT_LABEL,
    OUT_DEFL,
    OUT_END
} STTokenType;
```

Если у вас есть константы типа TOK_R8, то дополнительно нужно добавить идентификатор OUT_R8.

Метод, записывающий управляющий код:

```
// добавляет на ленту управляющий код
int add_CTRL(STTokenType tt) {
    if (count >= MAX_RPN) assert(0);
    ST[++count].tt = tt;
    return count;
}
```

Метод, добавляющий метку с идентификатором id:

```
// добавляет на ленту метку с идентификатором id
int add_LABEL(int id) {
    if (count >= MAX_RPN) assert(0);
    ST[++count].tt = OUT_LABEL;
    ST[count].int_val = id;
    return count;
}
```

Метод, добавляющий идентификатор:

```
// добавляет на ленту идентификатор
int add_ID(char * id, int tlin, int tpos) {
    if (count >= MAX_RPN) assert(0);
    ST[++count].tt = OUT_ID;
    ST[count].tlin = tlin;
    ST[count].tpos = tpos;
    ST[count].set_str(id);
    return count;
}
```

Следующий метод добавляет константу:

```
// добавляет на ленту константу типа tt
int add_const(STTokenType tt, int int_val, double dbl_val) {
    if (count >= MAX_RPN) assert(0);
    ST[++count].tt = tt;
    ST[count].int_val = int_val;
    ST[count].dbl_val = dbl_val;
    return count;
}
```

Наконец, нужен метод, который найдет на ленте метку с идентификатором id и последующим управляющим кодом OUT_DEFL:

```
// ищет на ленте определение метки с идентификатором id
int find_DEF(int id) {
    for (int i = 1; i <= count; i++) {
        if (ST[i].tt == OUT_END) return -1; // конец ленты
        if (ST[i].tt == OUT_LABEL && ST[i].int_val == id) {
            if (ST[i + 1].tt == OUT_DEFL) {
                return (i + 2);
            }
        }
    }
    return -1;
}
```

Для проверки нужно создать представителя ленты в классе syntaxan:

```
// лента ПОЛИЗ
ststrip strip;
// стек номеров правил
udeque m_rules;
```

В анализаторе при допуске входной цепочки нужно записать на ленту управляющий код конца ленты:

```
        } else if (ACC == t) {
            // ДОПУСК
            // записываем конец ленты ПОЛИЗ
            strip.add_CTRL(OUT_END);
            // выходим из цикла
            break;
```

Нам нужно сформировать текстовое представление ленты ПОЛИЗ. Для этой цели в классе ststrip есть метод print.

```
// запись в текстовый файл
void print(FILE * f) {
    for (int i = 1; i <= count; i++) {
        fprintf(f, "%d ", i);
        switch (ST[i].tt) {
        case TOK_UNKNOWN: fprintf(f, "UNKNOWN"); break;
        case OUT_END:     fprintf(f, "END"); break;
        case OUT_LABEL:   fprintf(f, "LABEL"); break;
        case OUT_DEFL:    fprintf(f, "DEFL"); break;
        case OUT_ID:      fprintf(f, "ID"); break;
        case OUT_I4:      fprintf(f, "I4"); break;
        default: fprintf(f, "UNEXPECTED");
        }
        fprintf(f, " %d", ST[i].tlin);
        fprintf(f, " %d", ST[i].tpos);
        fprintf(f, " %d", ST[i].int_val);
        fprintf(f, " %.20f", ST[i].dbl_val);
        fprintf(f, " \"%s\"\n", ST[i].str_val);
    }
}
```

В этом методе указаны лишь часть кодов, которые могут оказаться на ленте. По мере необходимости коды нужно добавлять.

Переходим в метод parse класса syntaxan.

В конце метода parse добавим запись ленты в текстовый файл:

```
void parse() {
    . . .
    // запись ленты ПОЛИЗ
    FILE * out = fopen("strip.txt", "wt");
    strip.print(out);
    fclose(out);
} // parse
```

9.4. Класс вычислительного стека

Для выполнения операций ПОЛИЗ требуется вычислительный стек.

Добавляем его в конце модуля ststrip.h:

```
// вычислительный стек ПОЛИЗ
template <int N>
struct exstack {
private:
    exstel ST[N];
public:
    int count;
    exstack() {
        count = 0;
    }
    int push(exstel & e) {
        if (count >= N) return 0;
        ST[count++] = e;
        return 1;
    }
    int pop(exstel & e) {
        if ( count <= 0 ) return 0;
        e = ST[--count];
        return 1;
    }
};
```

Как видим, этот простой класс выполняет всего две операции: протал-кивание и выталкивание вычислительного элемента.

10. Работа 10. «Трансляционная грамматика»

Цель: преобразование синтаксической грамматики в трансляционную.
Задачи:
- трансляция выражений.

10.1. Подготовка МП-автомата

Сначала нам нужно подготовить МП-автомат к тому, что правила будут содержать символы OUT_XXX, называемые *операционными*.

В случае обнаружения таких символов автомат должен извлекать их из потока и формировать ленту ПОЛИЗ.

Нам нужен метод для записи элементов ПОЛИЗ по их коду.

Класс syntaxan. Добавляем новый метод после метода parse_LL:

```
// запись символа на ленту ПОЛИЗ
void out(STTokenType tt) {
    switch (tt) {
    case OUT_ID:
        strip.add_ID(m_str_val, m_tlin, m_tpos);
        break;
    case OUT_I4:
        strip.add_const(tt, m_int_val, m_dbl_val);
        break;
    default:;
        // запишем код элемента ПОЛИЗ
        strip.add_CTRL(tt);
    }
} // out
```

Метод добавляет только два кода.

Другие коды будем добавлять по мере необходимости.

Заметим, что если код не разобран оператором switch, он будет добавлен как управляющий.

Еще один метод удаляет операционные символы со стека. Добавляем его после метода parse_LL:

```
// выводит операционные символы со стека
STTokenType clear_stack() {
    STTokenType s;
    while ((s = sta.top()) > SYM_LAST) {
        s = sta.pop();
        out(s);
    }
    return s;
} // clear_stack
```

Для определения того, что символ операционный, используется константа SYM_LAST, соответствующая последнему символу грамматики.

Символ извлекается из стека и если он операционный, то помещается на ленту. Если символ не операционный, то он возвращается.

Эти два метода нужно применить в МП-автомате.

Сначала заменим добавление кода конца ленты в случае окончания разбора:

```
} else if (ACC == t) {
    // ДОПУСК
    // записываем конец ленты ПОЛИЗ
    out(OUT_END);
    // выходим из цикла
    break;
```

Метод clear_stack применяется в двух случаях.

Первый раз он необходим при определении символа на стеке:

```
// рабочий цикл автомата LL
while ( 1 ) {
    // состояние на стеке
    s = clear_stack();
    // значение в таблице
    t = SYNTA[s][m_token];
    // анализ состояния
    if (t <= 0) {
```

Здесь, если переменная s получит значение операционного символа, произойдет ошибка, поскольку s используется как индекс таблицы.

Второй раз метод должен применяться непосредственно после операции «выброс», так как в этом случае при выбрасывании идентификатора или константы до следующего выбрасывания атрибуты токена доступны для записи на ленту:

```
} else if (ESC == t) {
    // ВЫБРОС
    // выталкиваем терминал из рабочего стека
    at = sta.pop();
    // выталкиваем операционные символы на стеке
    clear_stack();
    // следующий токен
    next_token();
} else if (t <= MAX_RULE) {
```

10.2. Преобразование грамматики в трансляционную

МП-автомат готов к формированию ленты ПОЛИЗ. Остается только записать операционные символы в грамматику.

Для определенности будем вычислять выражение и присваивать его значение некоторой переменной. Сейчас нужно подготовить поток токенов для допустимого текста на заданном языке, который содержит оператор присваивания:

```
A = B + 1
```

Лента ПОЛИЗ должна содержать следующие элементы:

```
ID(A) ID(B) I4(1) ADD ASS END
```

Сейчас на ленту записывается только элемент END.

Чтобы добавить на ленту элемент ID(B), нужно записать в грамматику символ OUT_ID в то правило, которое считывает токен TOK_ID как примитивный элемент данных. Например, если в грамматике есть правило

```
<P> = [id]
```

то в это правило добавляется символ OUT_ID:

```
<P> = [id]{ID}
```

Операционные символы записываются в фигурных скобках.

Токен TOK_I4 точно таким же образом должен формировать операционный символ OUT_I4.

Рассмотрим конкретный пример.

В моей грамматике есть правила:

```
#23 <P>=[id]
#24 <P>=[I4]
#25 <P>=[(]<E>[)]
```

Эти правила должны быть записаны следующим образом:

```
#23 <P>=[id]{ID}
#24 <P>=[I4]{I4}
#25 <P>=[(]<E>[)]
```

Изменения в грамматику вносятся вручную.

Вот как это выглядит в модуле syntab.h:

```
/*23 SYM_P  */{ TOK_ID, OUT_ID, TOK_EOT },
/*24 SYM_P  */{ TOK_I4, OUT_I4, TOK_EOT },
/*25 SYM_P  */{ TOK_LP, SYM_E, TOK_RP, TOK_EOT }
```

Прежде, чем формировать на ленте символы операций, их нужно определить. Соответствие между операциями и символами следующее:

присваивание	OUT_ASS
сложение	OUT_ADD
вычитание	OUT_SUB
умножение	OUT_MUL
деление	OUT_DIV

Эти символы нужно определить в модуле sttypes.h в перечислении символов STTokenType, например, так:

```
    OUT_DEFL,
    OUT_ASS,
    OUT_ADD,
    OUT_SUB,
    OUT_MUL,
    OUT_DIV,
    OUT_END
```

Не забываем, что каждый добавляемый в перечисление операционный символ должен выводиться в методе print класса ststrip.

Учитывая, что операция на ленте ПОЛИЗ записывается после операндов, нужно найти правильное положение символов операций в правилах.

Рассмотрим операцию сложения.

В моей грамматике она задана следующими правилами:

```
#15 <R>=<T><T.>
#16 <T.>=[+]<T><T.>
#18 <T.>=.
#19 <T>=<P><P.>
```

Операционный символ сложения добавляется так, чтобы он следовал за вторым операндом. Рассмотрим вывод в грамматике:

```
R => TT. => PT. => P+TT. => P+PT. =>* P+P+P+PT.
```

Очевидно, что нетерминал P ведет к формированию элемента данных.

Символ операции сложения должен следовать за вторым операндом, то есть предшествовать нетерминалу T.:

```
#15 <R>=<T><T.>
#16 <T.>=[+]<T>{ADD}<T.>
#18 <T.>=.
#19 <T>=<P><P.>

R => TT. => PT. => P+T{ADD}T. => P+P{ADD}T. =>* P+P{ADD}+P{ADD}T.
```

Если же символ операции поставить за нетерминалом T., то вывод будет другим:

```
#15 <R>=<T><T.>
#16 <T.>=[+]<T><T.>{ADD}
#18 <T.>=.
#19 <T>=<P><P.>

R => TT. => PT. => P+TT.{ADD} => P+PT.{ADD} =>* P+P+PT.{ADD}{ADD}
```

Заметим, что оба вывода верны, но порядок выполнения операций в этих случаях будет различным, в связи с чем различают лево-ассоциативный и право-ассоциативный порядок вычисления.

В модуле syntab.h изменения выглядят следующим образом:

```
/*16 SYM_T_ */{ TOK_ADD, SYM_T, SYM_T_, OUT_ADD, TOK_EOT },
```

Аналогичным образом добавляются символы других арифметических и прочих операций. Для обозначения прочих операций следует использовать такие же обозначения, как у соответствующих токенов:

AND	OUT_AND
OR	OUT_OR
NOT	OUT_NOT
MOD	OUT_MOD
равно	OUT_EQ
не равно	OUT_NE
меньше	OUT_LT
больше	OUT_GT
меньше либо равно	OUT_LE
больше либо равно	OUT_GE
унарный минус	OUT_NEG

Не забываем добавлять описание вывода всех добавляемых символов операций в методе print класса ststrip.

Все операции должны быть протестированы, а лента ПОЛИЗ должна быть проанализирована на правильность.

11. Работа 11. «Исполняющая стековая машина ПОЛИЗ»

Цель: проектирование исполняющей машины ПОЛИЗ.
Задачи:
- чтение ленты ПОЛИЗ и выполнение операций.

11.1. Стековая машина ПОЛИЗ

Под исполняющей стековой машиной ПОЛИЗ здесь понимается алгоритм, читающий ленту ПОЛИЗ и выполняющий записанные на ней операции. Результатом выполнения операций является выходной текст, являющийся результатом перевода входного текста. Выходной текст может быть как машинный (объектный) код, так и непосредственное вычисление. В нашем случае исполняющая машина должна выполнять непосредственные вычисления, так как наш транслятор является интерпретатором.

Будем выполнять вычисления не в отдельном проекте, а в текущем.

Нам нужно сформировать метод класса syntaxan, который прочитает ленту ПОЛИЗ и выполнит вычисления. Поскольку при этом должна выполняться проверка семантических правил, нам нужна таблица символов, в которой будут храниться символы и их значения.

Добавим в проект модуль stsymt.h и включим его в модуль syntaxan:

```
// syntaxan.h
#pragma once
// синтаксическая таблица
#include "syntab.h"
// лента ПОЛИЗ
#include "ststrip.h"
// таблица символов
#include "stsymt.h"
// стек для правил
#include "studeque.h"
```

Данная таблица символов поддерживает только скалярные типы данных, поэтому исходный код не должен содержать массивов.

Опишем константу максимального количества операций ПОЛИЗ:

```
// максимальное количество шагов МП-автомата
#define ST_MAX_PARSE_COUNT 1000
// максимальное количество циклов ПОЛИЗ
#define MAX_PN_COUNT 100000
```

В конце класса syntaxan объявим таблицу символов и вычислительный стек ПОЛИЗ:

```
    // таблица символов
    stsyms<MAX_TABLE> syms;
    // рабочий стек ПОЛИЗ
    exstack<MAX_EXE_STACK> exe;
    // стек номеров правил
    udeque m_rules;
};
```

Описываем вычислительную машину ПОЛИЗ как функцию:

```
// исполняющая машина ПОЛИЗ
void execute() {
    // счетчик операций ПОЛИЗ
    int it_counter = 0;
    // указатель ленты ПОЛИЗ
    int strip_pointer = 1;
    // текущий элемент ПОЛИЗ
    STTokenType tok;
    // переменные для вычислений
    exstel X, Y;
    // вспомогательные
    int j = 0;
    // рабочий цикл выполнения ленты ПОЛИЗ
    while (1) {
        // очистим переменные
        X.clear();
        Y.clear();
        if (++it_counter > MAX_PN_COUNT) {
            message_out("EXE DEADLOCK");
            return;
        }
    }
} // execute
```

Вызываем эту функцию в методе parse:

```
    int res = parse_LL();
    if (res == 1) {
        message_out("Success");
        execute();
    }
```

11.2. Распознавание оператора объявления переменной

Сейчас мы должны распознать объявление скалярной переменной.

Будем исходить из предположения, что грамматика оператора объявления скалярной переменной имеет следующий вид:

```
#4 <S>=[dim][long][id]<D.>[LF]
#5 <D.>=[,][id]<D.>
#6 <D.>=.
```

Это дает возможность написать следующее объявление:

```
Dim Long A, B, C LF
```

При этом должна быть сформирована следующая лента ПОЛИЗ:

```
A LONG DIM B LONG DIM C LONG DIM
```

Грамматика должна быть преобразована в трансляционную грамматику следующим образом:

```
#4 <S>=[dim][long][id]{ID}{LONG}{DIM}<D.>[LF]
#5 <D.>=[,][id]{ID}{LONG}{DIM}<D.>
```

В модуль syntab.h нужно внести изменения, например:

```
/* 4 SYM_S  */{ TOK_DIM, TOK_LONG, TOK_ID, OUT_ID, OUT_LONG,
               OUT_DIM, SYM_D_, TOK_LF, TOK_EOT },
/* 5 SYM_D_ */{ TOK_COMMA, TOK_ID, OUT_ID, OUT_LONG, OUT_DIM,
               SYM_D_, TOK_EOT },
```

Сейчас нужно отредактировать входной текст так, чтобы он содержал только объявление одной скалярной переменной целого типа, и проанализировать этот текст лексическим анализатором.

Переходим в метод execute класса syntaxan.

Формируем оператор разбора очередного символа ленты:

```
while ( 1 ) {
    // очистим переменные
    . . .
    // считываем тип элемента ленты
    tok = strip[strip_pointer].tt;
    switch (tok) {
    case OUT_ID: case OUT_I4: case OUT_LONG:
        // проталкиваем в стек
        exe.push(strip[strip_pointer]);
        // следующий элемент ленты
        strip_pointer++;
        break;
    }
    . . .
}
```

Понимать это надо так: если на ленте идентификатор, константа или тип, мы проталкиваем их в вычислительный стек ПОЛИЗ и переходим к следующему элементу ленты.

В результате после выполнения ленты в вычислительном стеке должны находиться идентификатор и тип, в моем случае LONG.

Следующим на ленте должен находиться символ DIM, указывающий на определение переменной. Этот символ является операцией. Операция заключается в том, чтобы вытолкнуть из стека тип и запомнить его, а затем вытолкнуть из стека переменную и записать ее в таблицу символов, задавая при этом запомненный тип. Если при добавлении символа окажется, что переменная уже есть в таблице, то программа должна сформировать сообщение о семантической ошибке и завершить выполнение.

Заметим, что возможен и другой подход к формированию ленты. Он заключается в том, чтобы для оператора

```
Dim Long A, B, C LF
```

сформировать такую ленту ПОЛИЗ:

```
A B C 3 LONG DIM
```

В этом случае на стеке находится также число «три», равное количеству формируемых переменных. Чтобы подсчитать число переменных, в правила грамматики нужно добавить три операционных символа. Один из этих символов обнуляет счетчик идентификаторов (OUT_N_CLR), другой подсчитывает идентификаторы (OUT_N_INC), третий записывает идентификатор на ленту (OUT_N_OUT). Это немного сложнее, мы реализуем более простой подход.

Разбираем операцию DIM в методе execute:

```
switch (tok) {
    . . .
case OUT_DIM:
    // выталкиваем тип переменной
    exe.pop(Y);
    // выталкиваем идентификатор
    exe.pop(X);
    // добавляем символ в таблицу символов
    if (j = syms.insert(X.str_val) == ST_EXISTS) {
        // идентификатор существует
        message_out("EXE Duplicate identifier");
        return;
    }
    // устанавливаем тип символа
    syms[j].data_type = Y.tt;
    // следующий элемент ленты
    strip_pointer++;
    break;
}
```

Нам нужно также распознать конец ленты:

```
switch (tok) {
    . . .
case OUT_END:
    message_out("EXE DONE");
    return;
}
```

11.3. Выполнение присваивания

Изменим входной текст так, чтобы он содержал объявление переменной A и оператор присваивания A = 1. Проанализируем входной текст лексическим анализатором.

Если грамматика не содержит символа OUT_ASS, его нужно добавить сейчас. Например, если в грамматике есть правило

```
#7 <S>=[id][=]<E>[LF]
```

то нужно преобразовать его следующим образом:

```
#7 <S>=[id]{ID}[=]<E>{ASS}[LF]
```

В модуле syntab.h это может выглядеть следующим образом:

```
/*7 SYM_S*/{ TOK_ID, OUT_ID, TOK_EQ, SYM_E, OUT_ASS, TOK_LF, TOK_EOT },
```

Добавим разбор символа OUT_ASS в оператор switch метода execute:

```
        case OUT_ASS:
            // вытолкнуть из стека значение в Y
            // вытолкнуть из стека переменную в X
            // найти переменную X в таблице символов
            // присвоить переменной значение
            // следующий элемент ленты
            break;
```

Первое действие при выполнении присваивания — получение значения, находящегося на стеке. Поскольку на стеке могут быть как переменные, так и константы, нам нужен метод, который извлечет константное значение. Добавляем новый метод в класс syntaxan после метода execute:

```
// извлекает из стека значение
int exe_pop(exstel & e) {
    exe.pop(e);
    if (e.tt == OUT_I4) {
        return 1;
    } else if (e.tt == OUT_ID) {
        int j = syms.find(e.str_val);
        if (j == ST_NOTFOUND) {
            // символ не найден
            message_out("EXE POP Identifier not found");
            return 0;
        }
        // извлекаем значение из таблицы символов
        e.int_val = syms[j].int_val;
        e.dbl_val = syms[j].dbl_val;
        // меняем тип элемента на константный
        e.tt = OUT_I4;
        return 1;
    } else {
        // неправильная лента ПОЛИЗ
        message_out("EXE Internal error");
        return 0;
    }
} // exe_pop
```

В случае, если на ленте константа, то мы возвращаем ее.

В случае, если на ленте идентификатор, то ищем его в таблице символов. Если символ не найден, это семантическая ошибка, выводим сообщение и завершаем выполнение ленты. Если символ найден, то извлекаем из таблицы символов его текущее значение, записываем его в извлеченный элемент и меняем тип элемента на константный.

В случае, если на ленте окажется элемент другого типа, фиксируем ошибку ленты и завершаем ее выполнение.

Возвращаемся к разбору символа OUT_ASS. Описываем действия:

```
case OUT_ASS:
    // вытолкнуть из стека значение в Y
    if (!exe_pop(Y)) return;
    // вытолкнуть из стека переменную в X
    exe.pop(X);
    // найти переменную X в таблице символов
    j = syms.find(X.str_val);
    if (j == ST_NOTFOUND) {
        message_out("EXE Identifier not found");
        return;
    }
    // присвоить переменной значение
    syms[j].int_val = Y.int_val;
    // следующий элемент ленты
    strip_pointer++;
    break;
```

Тестируем программу, убеждаемся, что присваивание выполняется.

11.4. Вычисление выражений

Изменим входной текст так, чтобы он содержал объявление переменной A и оператор присваивания A = A + 1.

Добавим разбор операционных символов арифметических операций в оператор switch метода execute:

```
case OUT_ADD: case OUT_SUB: case OUT_MUL: case OUT_DIV:
    // вытолкнуть из стека значение второго операнда в Y
    // вытолкнуть из стека значение первого операнда в X
    // выполнить операцию
    // протолкнуть в стек результат как константу
    // следующий элемент ленты
    break;
```

Операнды извлекаются методом exe_pop. При этом операнды являются константами, и атрибут tt (*token type*) у них равен OUT_I4.

Далее нужно снова выполнить разбор значения tok, чтобы отделить одну операцию от другой:

```
    // выполнить операцию
    switch (tok) {
    case OUT_ADD:
        break;
    case OUT_SUB:
        break;
    case OUT_MUL:
        break;
    case OUT_DIV:
        break;
    }
```

Собственно операция выполняется над элементами X и Y примерно следующим образом:

```
switch (tok) {
case OUT_ADD:
    X.int_val += Y.int_val;
    break;
case OUT_SUB:
```

Неявно подразумевается, что все элементы имеют один и тот же тип данных, целочисленный или вещественный. В случае, если это не так, то должно выполняться приведение типов и изменение типа результата при необходимости. На самом деле нужно доопределить класс элемента ленты так, чтобы операции выполнялись в классе exstel.

Ниже приведен пример описания операции в классе:

```
// операция сложения
exstel operator += (exstel & a) {
    if (tt == OUT_I4) {
        if (a.tt == OUT_I4) {
            int_val += a.int_val;
        }
    }
    return *this;
}
```

При этом здесь не показано, как выполнить приведение типов, если типы данных не совпадают, но как это сделать, надеюсь, понятно.

Тогда операция сложения, например, может быть выполнена так:

```
switch (tok) {
case OUT_ADD:
    X += Y;
    break;
case OUT_SUB:
```

Сейчас можно не описывать выполнение операций в классе exstel.

Другие виды операций выполняются аналогично.

При выполнении операции деления нужно учесть, что частное не может быть равным нулю:

```
case OUT_DIV:
    if (Y.int_val == 0) {
        message_out("EXE Division by zero");
        return;
    }
    X.int_val /= Y.int_val;
    break;
```

Если при делении используются вещественные типы, то проверку на ноль нужно выполнять корректно, то есть проверять машинный ноль.

Операции отношений и логические операции при этом должны формировать на ленте целочисленные значения «ноль» для лжи и «единица» для истины.

11.5. Оператор вывода выражения

Изменим входной текст и добавим в него оператор вывода значения переменной A оператором print.

Если в грамматике еще не добавлен символ OUT_PRINT, то это нужно сделать сейчас. Например, если в грамматике есть правило

```
#10 <S>=[print]<E>[LF]
```

то его нужно преобразовать следующим образом:

```
#10 <S>=[print]<E>{print}[LF]
```

В модуле syntab.h это может иметь вид:

```
/*10 SYM_S   */{ TOK_PRINT, SYM_E, OUT_PRINT, TOK_LF, TOK_EOT },
```

При этом символ OUT_PRINT должен быть добавлен в перечисление символов STTokenType и в метод print класса ststrip.

Класс syntaxan.

В метод execute добавляем разбор символа OUT_PRINT:

```
case OUT_PRINT:
    // вытолкнуть из стека значение в Y
    if (!exe_pop_val(Y)) return;
    // вывести значение
    printf("EXE PRINT %d\n", Y.int_val);
    // следующий элемент ленты
    strip_pointer++;
    break;
```

Остается протестировать программу и убедиться, что можно вычислять арифметические выражения любой сложности.

12. Работа 12. «Действия с метками»

Цель: работа с метками на ленте ПОЛИЗ.
Задачи:
- операционные символы операторов управления;
- операции с метками на ленте ПОЛИЗ.

12.1. Операторы управления

К операторам управления будем в данном контексте относить операторы, которые приводят к ветвлениям в алгоритме (к появлению веток).

Поскольку ветки располагаются на ленте ПОЛИЗ (равно как и в памяти вычислительного устройства) последовательно, нужен механизм для перехода к веткам, и для обхода веток. Этот механизм в теоретическом курсе называется «линеаризация».

Основным понятием линеаризации является метка. Под меткой в данном случае понимается некоторая точка в последовательности элементов ПОЛИЗ, специальным образом обозначенная. Это позволяет делать переходы к меткам, и обходить, таким образом, другие ветки элементов.

Рассмотрим оператор IF-THEN-ELSE.

В операторе такой конструкции есть две ветки. Ветку, которая выполняется при истинности выражения, будем называть «веткой истины», соответственно, ветку, которая выполняется при лжи выражения, будем называть «веткой лжи».

В памяти компьютера такой оператор располагается следующим образом:

```
    вычисление выражения
    переход по лжи к метке L1
    ветка истины
    переход к метке L2
L1:
    ветка лжи
L2:
```

При этом при вычислении выражения должен быть сформирован логический результат. Обычно нулевое значение выражения принимается за ложь, а любое другое значения выражения — это истина.

Если выражение ложно, то осуществляется так называемый условный переход к точке памяти, обозначенной как «L1» и далее выполняется ветка лжи. В противном случае выполняется ветка истины, и осуществляется безусловный переход к точке памяти, обозначенной как «L2».

Таким образом, для вычисления условного оператора нужны метки и два действия: условный переход по лжи и безусловный переход.

Рассмотрим теперь, как должна выглядеть лента ПОЛИЗ для условного оператора IF-THEN-ELSE.

e l_1 BZ ветка-истина l_2 BR l_1 DEFL ветка-ложь l_2 DEFL

Лента начинается с вычисления выражения e.

Далее на ленте должна быть размещена метка, условно обозначенная как l_1, за которой следует операция перехода по лжи BZ (*branch if zero*).

Затем следуют элементы, соответствующие ветке истины.

Поскольку после ветки истины должен следовать безусловный переход к концу оператора, далее на ленте расположены метка l_2 и операция BR (*branch*).

Перед веткой лжи должна быть расположена метка, она на ленте задается при помощи элементов l_1 и DEFL (*define label*). Аналогичным образом в конце оператора задана метка l_2.

Все метки на ленте должны быть уникальными. Поэтому нужен механизм для генерации меток. Он есть во всех без исключения трансляторах.

У нас механизм самый простой.

Номер метки генерируется методом new_label класса ststrip.

DEFL — это специальный управляющий код ПОЛИЗ.

BR и BZ — операции, которые могут помещаться на ленту.

Теперь возникает вопрос, как получить данную последовательность элементов ПОЛИЗ из грамматики.

Для этой цели будем использовать активные операционные символы.

Это такие символы, которые при распознавании записывают на ленту необходимые последовательности элементов, выполняя при этом некие простые алгоритмы. Для формирования ПОЛИЗ условного оператора с веткой лжи требуется три таких символа-генератора.

Обозначим эти символы PUSH, POPL и SWAP.

Когда в правиле грамматики встречается символ PUSH, то генерируется новая метка, которая проталкивается на специальный стек меток, и вставляется на ленту как элемент LABEL.

Когда в правиле грамматики встречается символ POPL, то из стека меток выталкивается метка и вставляется на ленту ПОЛИЗ.

Когда в правиле грамматики встречается символ SWAP, то две верхних метки на стеке меняются местами, после чего выталкивается метка, которая записывается на ленту.

Канитель со стеком меток нужна для того, чтобы иметь возможность вставлять один оператор в другой, например if (e) if (e)… В противном случае метки перепутаются и вычисления будут неверными.

Таким образом, требуется еще стек меток.

12.2. Формирование грамматики

В перечисление STTokenType нужно добавить символы:
OUT_BZ, OUT_BR, OUT_PUSH, OUT_POPL, OUT_SWAP.
Эти символы следует включить также в метод print класса ststrip.
Сформируем трансляционную грамматику для оператора IF.

```
/*  */{ TOK_IF, SYM_E, OUT_PUSH, OUT_BZ, TOK_THEN, TOK_LF,
                SYM_M, SYM_A, TOK_EOT },
/*  */{ TOK_END, TOK_LF, OUT_POPL, OUT_DEFL, TOK_EOT },
/*  */{ TOK_ELSE, TOK_LF, OUT_PUSH, OUT_BR, OUT_SWAP, OUT_DEFL,
                SYM_M, TOK_END, TOK_LF, OUT_POPL, OUT_DEFL, TOK_EOT },
```

Здесь показано, как правильно вставить операционные символы. Дополнительно см. учебно-методическое пособие по курсу [1].

12.3. Обработка активных символов

В класс syntaxan добавляем стек меток:

```
    // стек исполняющей машины ПОЛИЗ
    exstack<MAX_EXE_STACK> exe;
    // стек меток
    ststack<int, MAX_STACK> stl;
    // стек номеров правил
    udeque m_rules;
};
```

Теперь нужно перейти к методу out и обработать активные операционные символы грамматики:

```
    void out(STTokenType tt) {
        int i = 0, j = 0;
        switch (tt) {
        case OUT_PUSH:
            j = strip.new_label();
            stl.push(j);
            strip.add_LABEL(j);
            break;
        case OUT_POPL:
            j = stl.pop();
            strip.add_LABEL(j);
            break;
        case OUT_SWAP:
            i = stl.pop();
            j = stl.pop();
            stl.push(i);
            strip.add_LABEL(j);
            break;
            . . .
            . . .
            . . .
            . . .
    } // out
```

12.4. Вычисление операторов

Далее переходим к методу execute (вычислительной машине ПОЛИЗ) и описываем действия при обнаружении новых элементов.

Если на ленте обнаружена метка, на стек нужно протолкнуть ее числовое значение как константу:

```
case OUT_LABEL:
    // на стек помещаем значение метки
    X.int_val = strip[strip_pointer].int_val;
    // тип значения - константа
    X.tt = OUT_I4;
    exe.push(X);
    strip_pointer++;
    break;
```

Если на ленте обнаружено определение метки, то значение метки нужно вытолкнуть из стека, так как оно не требуется, а оставлять что-то на стеке нельзя:

```
case OUT_DEFL:
    // выталкиваем значение метки, оно не требуется
    exe.pop(X);
    strip_pointer++;
    break;
```

Если на ленте операция перехода по лжи, то нужно вытолкнуть из стека сначала значение метки, затем значение выражения, проверить значение выражения, и если оно равно нулю, то найти на ленте метку и выполнить переход на нее.

При проверке значения выражения нужно, конечно, учитывать его тип, в примере предполагается целочисленный тип:

```
case OUT_BZ:
    // значение метки на стеке
    if (!exe_pop(Y)) return;
    // значение выражения на стеке
    if (!exe_pop(X)) return;
    if (X.int_val == 0) {
        // ищем метку на ленте
        j = strip.find_DEF(Y.int_val);
        if (j == -1) {
            message_out("EXE LABEL not found");
            return;
        }
        // переход
        strip_pointer = j;
    } else {
        // нет перехода
        strip_pointer++;
    }
    break;
```

Наконец, если на ленте операция безусловного перехода, выталкиваем из стека значение метки, ищем метку на ленте и выполняем переход к найденной метке:

```
case OUT_BR:
    exe.pop(X);
    j = strip.find_DEF(X.int_val);
    if (j == -1) {
        message_out("EXE LABEL not foind");
        return;
    }
    strip_pointer = j;
    break;
```

Для тестирования нужно использовать пример кода с вложенными условными операторами, например:

```
Dim Long A
If 0 Then
    If 0 Then
        A = 1
    Else
        A = 2
    End
Else
    If 0 Then
        A = 3
    Else
        A = 4
    End
End
Print A
```

Варьируя значения выражений в операторах IF, нужно получить вывод всех четырех значений переменной A.

Операторы циклов реализуются похожим образом.

Дополнительно см. учебно-методическое пособие по курсу [1].

Литература

1. Пономарев В.В. Конспективное изложение теории языков программирования и методов трансляции. Учебно-методическое пособие. В двух книгах. Книга 1. Озерск: ОТИ НИЯУ МИФИ, 2017. — 204 с., ил.