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 "d". Эта грамматика корректна. Например, можно построить следующие выводы:

$M \Rightarrow DS\ SS \Rightarrow SS \Rightarrow s$
$M \Rightarrow DS\ SS \Rightarrow D\ SS \Rightarrow d\ s$
$M \Rightarrow DS\ SS \Rightarrow DS\ D\ SS \Rightarrow D\ SS \Rightarrow^{*} d\ d\ s$

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]
```

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 вещественной 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]
```