Лекция 6. Пример языка программирования

Источники вдохновения для создания демонстрационного языка

Чтобы осуществить разработку транслятора, необходимо иметь язык программирования. Можно взять уже существующий язык программирования. Однако, создаваемый транслятор будет большим по объему, что затруднит изучение методов его разработки. Можно упростить любой из существующих языков программирования до уровня, удобного для изучения основных методов разработки трансляторов. Такой подход используется достаточно часто. Он удобен и позволяет легко разобрать различные методы построения трансляторов. Вместе с тем, организация таких языков программирования обладает определенными ограничениями, определяемыми "настоящим" языком, что не позволяет изучить ряд специфических приемов. Введение в упрощенный язык своих дополнительных конструкций нарушает общее гармоничное восприятие от урезанного первоисточника и заставляет относиться к нему как к помеси бульдога с носорогом. Поэтому, проще разработать собственный, достаточно простой язык, в котором можно определить все конструкции, необходимые для демонстрации различных аспектов разработки трансляторов.

История программирования полна идеями, которые могут служить источниками для создания своего языка. Мы же остановимся на тех из них, которые были предложены Дейкстрой в его книге "Дисциплина программирования" . В ней излагаются концепции безошибочного программирования и предлагаются языковые конструкции, поддерживающие такой подход. Этот язык также описан в книге Гриса , посвященной изучению разработке правильных программ и методов доказательства правильности программ. Назовем разрабатываемый язык DPL (Dejkstra Programming Language). Учитывая, что разработка языка носит учебный характер, внесем в него ряд изменений, противоречащих позициям автора, но позволяющих изучить необходимые методы разработки трансляторов.

Синтаксис и семантика DPL

DPL содержит основные операторы обработки данных и управления, которые позволяют строить простые программы. Вместе с тем, в нем отсутствуют конструкции, широко применяемые в развитых языках. В частности, нет процедур и функций, блочной структуры, типов данных, классов. Это не мешает изучению основных принципов разработки трансляторов, которые можно с успехом использовать и при разработке более сложных языков программирования. Описание языка построим по традиционному принципу. В начале рассмотрим элементарные конструкции, а затем структуру программы. Для описания синтаксиса DPL будем использовать РБНФ.

Элементарные конструкции

К элементарным конструкциям языка обычно относятся его понятия, состоящие из терминальных символов, принадлежащих алфавиту языка. Выделение элементарных

конструкций обусловлено целым рядом причин, среди которых можно отметить:

- мы имеем в своем распоряжении набор базовых "кирпичиков", опираясь на которые, легче изучать более сложные понятия;
- в большинстве языков программирования смысл и представление элементарных конструкций совпадают, поэтому, поняв их в ходе изучения одного языка, легче перейти к следующему.

К элементарным относятся такие понятия, как идентификатор, числа (целые, действительные, двоичные, десятичные), комментарии, метки, знаки операций, разделители, строки символов. Список можно продолжить и дальше. Эти понятия уточняются и конкретизируются при описании семантики языка. Например, идентификатор может служить в качестве имени переменной, процедуры, функции или типа. Элементарные конструкции обычно описываются с позиции, удобной для их изучения пользователем. Они могут распознаваться как во время лексического, так и синтаксического анализа, хотя большая их часть обычно распознается сканером.

Ниже приводится синтаксис элементарных конструкций DPL. Их описанию предшествует определение групп основных символов в качестве отдельных понятий, разделяющих эти символы на отдельные категории (классы). Аналогичные элементарные конструкции присутствуют в любом языке, поэтому дополнительное пояснение отсутствует.

$ буква = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|

"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|

"U"|"V"|"W"|"X"|"Y"|"Z"|"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|

"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z".

$ цифра = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

$ идентификатор = (буква | "_") { буква | цифра | "_" }.

$ число = целое | действительное.

$ целое = двоичное | восьмиричное |

десятичное | шестнадцатиричное.

$ двоичное = "{2}" {/ "0" | "1" /}.

$ восьмиричное = "{8}"{ "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7" }.

$ десятичное = ["{10}"] {/ цифра /}.

$ шестнадцатиричное = "{16}" {/ цифра |"A"|"B"|"C"|"D"|"E"|"F"|

"a"|"b"|"c"|"d"|"e"|"f" /}.

$ действительное = числовая_строка порядок |

числовая_строка "." [числовая_строка] [порядок] |

"." числовая_строка [порядок].

$ числовая_строка = {/ цифра /}.

$ порядок = ("E"|"e")["+"|"-"] числовая_строка.

$ пробельный_символ = {/ пробел | табуляция |

перевод_строки | комментарий /}.

$ комментарий = "/*" { символ } "*/".

$ строка = """ { символ | """"} """.

Понятие **пробел** обозначает пустое место, которое можно задать пропуском **" "**. Однако, обычно это прямо не делается из-за опасения неправильной интерпретации или слияния кавычек при наборе текста и форматировании. Понятия **перевод_строки** и **табуляция** определяют невидимые символы, ASCII коды которых меньше кода пробела. Они могут присутствовать в тексте программы, обеспечивая его форматирование, но только в виде специальных знаков, и на экране или бумаге не отображаются. Понятие символа определяет все видимые символы кодовой таблицы, а также символы перевода строки и табуляции.

Следует отметить, что ряд понятий определен неформально. Эти определения ориентированы на то, чтобы раскрыть пользователю языка смысл написания и использования основных конструкций. Поэтому данное описание будем называть **пользовательским синтаксисом** языка. Для более полного понимания оно обычно снабжается дополнительным пояснительным текстом, определяющим и уточняющим семантику. Этим дополнительным описанием в данном случае является приведенный выше абзац, поясняющий понятия: **пробел**, **перевод_строки**, **табуляция**. Обычно пользовательский синтаксис напрямую не может быть использован для построения сканера или распознавателя. Его необходимо преобразовать к виду, удобного для эффективной реализации этих блоков транслятора.

Дополнительным описанием следует также снабдить понятия **комментария** и **строки**. Комментарий может задаваться в любом месте программы, где можно поставить пробел или разделитель. Он начинается парой символов **"/*"** и заканчивается другой парой **"*/"**. Между ними могут быть любые печатаемые символы, включая отдельные группы звездочек и наклонных линий, не образующих завершающую комбинацию, а также символы табуляции, перевода строки. Строка может содержать только видимые символы, заключенные между двумя кавычками. Если в строке необходимо поставить кавычку, то она дублируется при написании:

"Строка, в которой следующее ""слово"" взято в кавычки".

Ключевые слова и разделители используются для формирования выражений, описаний и операторов. В DPL они определяются следующим образом:

$ ключевое_слово = abort | begin | case | end | float | goto | int | loop |

or | read | skip | space | tab | var | write |.

$ разделитель = "(" | ")" | "[" | "]" | "," | ";" | ":" | ":=" | "*" | "/" | "%" |

"+" | "-" | "->" |"<" | "=" | ">" | "<=" | ">=" | "!=".

Составные конструкции. Организация программы

К составным конструкциям относятся понятия, определяющие структуру программы, ее операторов, описаний и выражений.

$ программа = begin (описание | оператор)

{ ";" (описание | оператор) } end.

$ описание = var идентификатор [размер]

{ "," идентификатор [размер] } ":" тип.

$ тип = int | float.

$ размер = целое.

$ оператор = метка непомеченный.

$ метка = идентификатор ":".

$ непомеченный = присваивания | условный | цикла | пустой |

ошибки | ввода | вывода | перехода.

$ присваивания = переменная ":=" выражение |

переменная "," присваивания "," выражение.

$ переменная = идентификатор ["[" выражение "]"].

$ выражение = ["-"] операнд { операция ["-"] операнд }.

$ операнд = "(" выражение ")" | число | переменная.

$ операция = "*" | "/" | "%" | "+" | "-" | "<" | "=" | ">" | ">=" |

"<=" | "!=".

$ условный = case [набор_охраняемых] end.

$ цикла = loop [набор_охраняемых] end.

$ набор_охраняемых = охраняемые [or охраняемые].

$ охраняемые = выражение "->" оператор { ";" оператор }.

$ пустой = skip |.

$ прерывания = abort [строка].

$ ввода = read переменная { "," переменная }.

$ вывода = write (выражение | спецификатор | строка)

{ "," (выражение | спецификатор | строка) }.

$ перехода = goto идентификатор.

$ спецификатор = (space | tab | skip) [выражение].

Приведенные правила требуют некоторых дополнительных пояснений, раскрывающих особенности семантики языка и мотивирующих решения, принятые из чисто субъективных соображений.

Краткое описание семантики языка.

Оператор присваивания имеет нетрадиционную форму и, в общем случае, обеспечивает одновременное присваивание нескольким переменным, расположенным слева от знака **":="** значений предварительно вычисленных выражений, расположенных в правой части. Каждой переменной соответствует свое выражение. Присваивания начинаются только после вычисления всех выражений, результаты которых временно сохраняются. Это позволяет произвести обмен значений переменных с использованием только одного оператора присваивания:

x, y := y, x

В обычных языках программирования необходимо написать три отдельных оператора:

t := x; x := y; y := t

Выражения задаются в традиционной инфиксной форме. Порядок выполнения операций определяется их приоритетом и скобками. В начале выполняются выражения в скобках. Наивысший приоритет имеет унарный минус **"-"**, далее следуют мультипликативные операции **"*"**, **"/"**, **"%"** (вычисление остатка), затем аддитивные **"+"**, **"-"** и, наконец операции отношения **"<"**, **"="**, **">"**, **"<="**, **">="**, **"!="**.

Для представления пустого оператора Дейкстра намеренно использовал специальное ключевое

слово **skip** (что мотивировал соответствующим текстом), хотя в большинстве языков программирования пустой оператор - это пустое место:

$ пустой = .

Для того, чтобы продемонстрировать решения проблемы неоднозначности синтаксиса языка, возникающие из-за наличия "традиционного" пустого оператора, я ввел оба варианта написания. Но это не самый большой мой грех по отношению к Дейкстре.

Для экстренного выхода из любой точки программы в языке используется оператор прерывания **abort**. Необязательная строка символов предназначена для пояснения причины выхода из программы.

В соответствии с концепциями безошибочного программирования, разработанными Дейкстрой, определены условный оператор и оператор цикла. Их тела содержат наборы операторов, выполнение которых возможно только при истинности условий, задаваемых предваряющими их охраняющими выражениями. Выражения отделяются от охраняемых ими операторов стрелками **"->"** и, начиная с первого, последовательно анализируются до тех пор, пока не встретится "истинное". Истинным считается ненулевое значение выражения. Предполагается, что в рассматриваемой версии языка операции отношения возвращают в качестве результата целое число, равное 1, при выполнении условия и, равное 0, если условие не выполняется. Если в условном операторе все охраняющие выражения дают ложь, то он выполняется как оператор ошибки (**abort**). Оператор цикла в данной ситуации эквивалентен пустому оператору (**skip**). Возникновение такой ситуации обеспечивает выход из цикла. При наличии истинного охраняющего выражения происходит выполнения охраняемых операторов и повторное выполнение оператора цикла. Оператор **abort** также эквивалентен конструкции **case end** (пустое тело в условном операторе), а оператор **skip** - оператору **loop end**.

Спецификаторы **space**, **tab** и **skip** используются в операторе вывода для форматирования выходного потока данных и означают пробел, табуляцию и перевод строки. Выражение, следующее за спецификатором, определяет количество его повторений. Строка символов используется для вывода пояснительного текста.

И, наконец, прокомментирую акт вандализма, осуществленный мною по отношению к Дейкстре: включение в язык оператора перехода **goto**. И это после того, как он воздвиг фундамент структурного программирования и написал целый ряд обличительных статей о вреде данного рудимента и атавизма одновременно! Однако использование данного оператора в учебном языке осуществляется только для того, чтобы ***продемонстрировать ряд особенностей, связанных с построением транслятора***. Поэтому, после нескольких лет сомнений и метаний (когда **goto** отсутствовал), я решился на столь тяжкий грех (хорошо, что Дейкстра не видел моих исходных текстов синтаксического анализатора, а не то гореть мне в аду).

Примеры программ на DPL

Ниже приведены примеры программ, раскрывающие особенности использования языка.

Пример 1. Алгоритм Евклида (нахождение наибольшего общего делителя).

```
begin
    var x, y: int; /* описание переменных */
    read x, y; /* ввод операндов */
    /* выполнять до равенства аргументов */
    loop x != y ->
        case
             x > y -> x := x - y
        or
             y > x -> y := y - x
        end
    end;
    write x /* полученный НОД */
end
```

Пример 2. Одновременное нахождение наибольшего общего делителя (НОД) и наименьшего общего кратного (НОК).

```
begin
    var x, y, u, v: int; /* описание переменных */
    read x, y; /* ввод операндов */
    u, v := y, x;
    /* выполнять до равенства аргументов */
    loop
        x > y -> x, v := x - y, v + u
    or
        y > x -> y, u := y - x, u + v
    end;
    write "НОД = ", x; /* НОД */
    write skip, "НОК = ", (u + v) / 2 /* НОК */
end
```

Пример 3. Суммирование n элементов из входного потока.

```
begin
    var a, i, s, n: int;
    i,s := 0,0;
    read n;
    loop
        i < n -> read a; s,i := s+a,i+1
    end;
    write s
end
```

Пример 4. Сортировка элементов вектора.

```
begin
    var A[100], i, n: int;
    i := 0;
    read n; /* чтение числа элементов */
    /* ввод вектора */
    loop
```

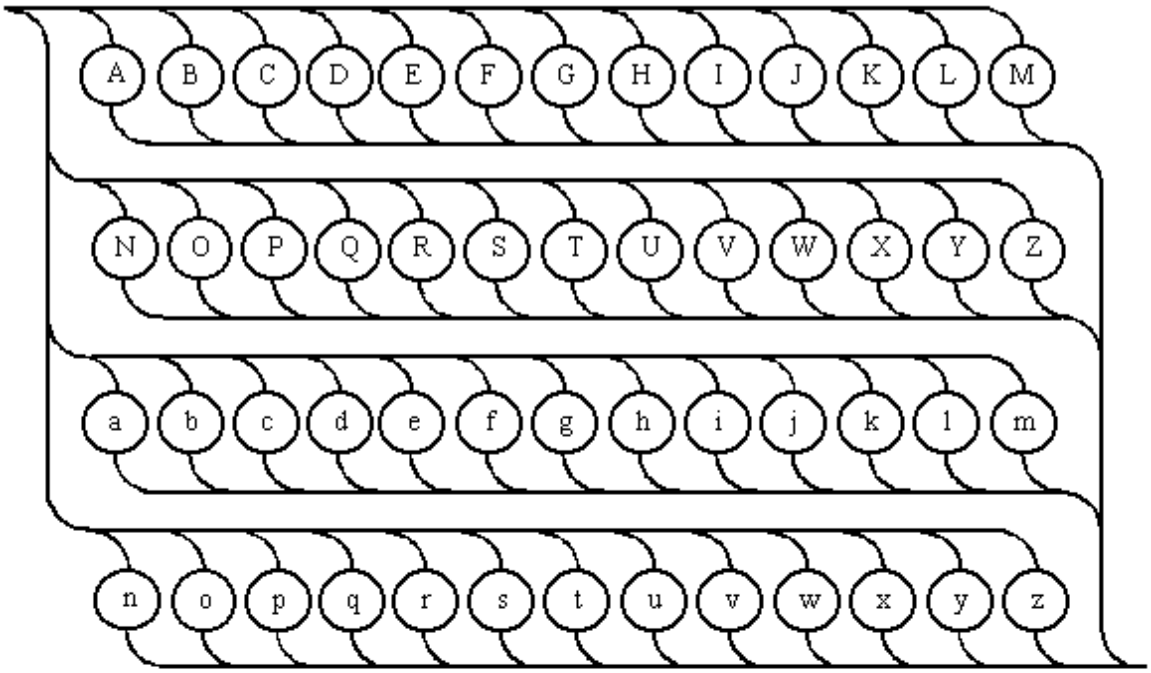
```
        i < n -> read A[ i ];
        i := i +1
    end;
    i := 0;
    /* сортировка методом пузырька */
    loop i < n-1 ->
        case
            A[ i ] > A[ i+1 ] ->
                A[ i ], A[ i+1 ],i := A[ i + 1], A[ i ], 0
        or
            A[ i ] <= A[ i+1 ] ->
                 i := i + 1

        end
    end;
    /* вывод вектора */
    i := 0;
    loop
        i < n ->
        write A[ i ], skip;
        i := i + 1
    end
end
```

Описание пользовательского синтаксиса с использованием диаграмм Вирта

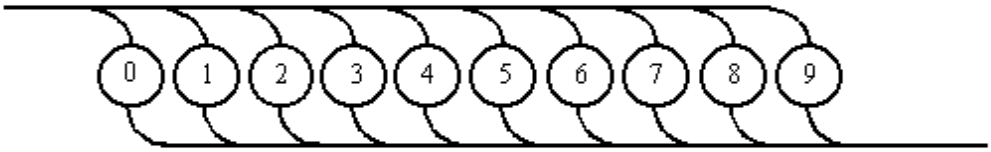
Разработка транслятора, рассматриваемая в последующих разделах, опирается на описание синтаксиса DPL с использованием диаграмм Вирта. Поэтому, необходимо осуществить перевод с одного метаязыка на другой. Сам синтаксис пока по-прежнему остается пользовательским, а его изменение будет осуществляться в дальнейшем по мере надобности. Окончательное представление синтаксиса DPL с применением диаграмм Вирта приведено на рис. 3.1 без каких либо дополнительных комментариев.

Элементарные конструкции

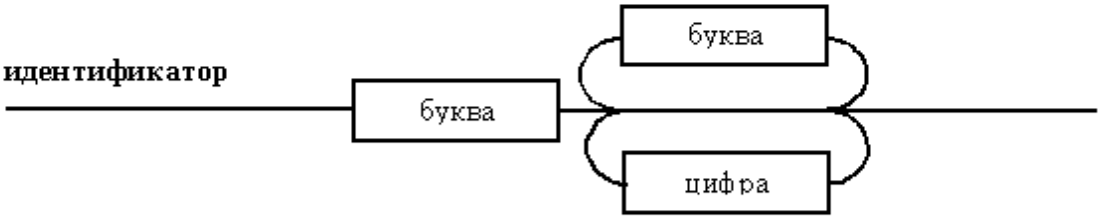
буква



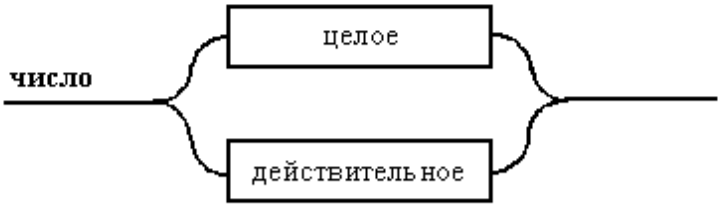
цифра



идентификатор



число



целое