

Лекция 3

Лексический анализ

§9. Основные понятия, связанные с текстом программы

Определение. *Строка* (string) – это последовательность кодовых точек. Номер кодовой точки в строке – *индекс* кодовой точки.

Определение. *Префикс* строки s (prefix) – это строка, полученная удалением нуля или нескольких последних кодовых точек строки s .

Определение. *Суффикс* строки s (suffix) – это строка, полученная удалением нуля или нескольких первых кодовых точек строки s .

Определение. *Подстрока* строки s (substring) – это строка, полученная удалением префикса и суффикса строки s .

Определение. *Правильные* префикс, суффикс и подстрока строки s – любая непустая строка, которая является соответственно префиксом, суффиксом и подстрокой строки s и не совпадает со строкой s .

Определение. *Текст программы* – это строка, полученная в результате чтения входного потока.

Определение. *Фрагмент* текста программы – это подстрока текста программы.

Определение. *Строчка* программы (line) – это фрагмент текста программы, не содержащий маркеров конца строки и не являющийся правильной подстрокой другой строчки программы.

Тем самым текст программы – это последовательность строчек, разделённых маркерами конца строки.

Индекс кодовой точки в строчке программы мы будем называть *позицией* кодовой точки.

Определение. *Координата кодовой точки* в тексте программы – это тройка $\langle \text{line}, \text{pos}, \text{index} \rangle$, в которой line – номер строки, в которой расположена кодовая точка, pos – позиция кодовой точки в строке, а index – индекс кодовой точки в тексте программы.

Координаты кодовых точек фигурируют в сообщениях об ошибках в виде пар $\langle \text{line}, \text{pos} \rangle$. Причём и line , и pos нумеруются, начиная с 1.

Компонента index координаты кодовой точки имеет смысл в том случае, если в память загружен весь текст программы.

Определение. *Координата фрагмента* текста программы – это пара $\langle \text{starting}, \text{ending} \rangle$ координат первой и последней кодовых точек фрагмента.

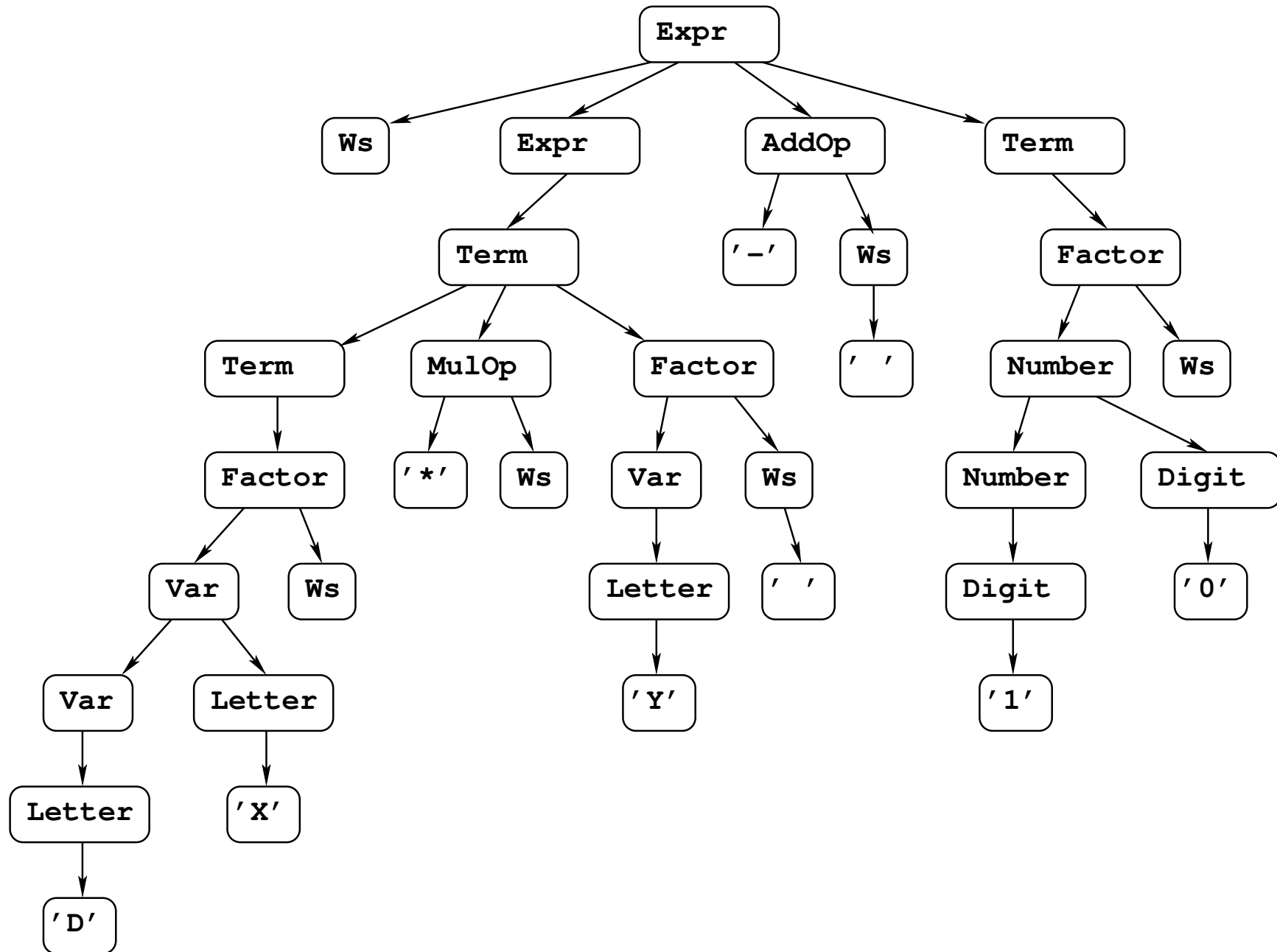
§10. Постановка задачи лексического анализа

Выделение из грамматики языка программирования его лексической структуры – проверенная временем декомпозиция задачи компиляции.

Пример. Грамматика языка арифметических выражений без выделения лексической структуры (БНФ):

```
Ws      ::= " " | Ws " " | .
Digit   ::= "0" | "1" | ... | "9".
Letter  ::= "A" | "B" | ... | "Z".
Number  ::= Number Digit | Digit.
Var      ::= Letter | Var Letter | Var Digit.
Expr     ::= Ws Expr AddOp Term | AddOp Term | Term.
Term     ::= Term MulOp Factor | Factor.
Factor   ::= "(" Expr ")" Ws | Var Ws | Number Ws.
MulOp    ::= "*" Ws | "/" Ws.
AddOp    ::= "+" Ws | "-" Ws.
```

"DX*Y - 10"



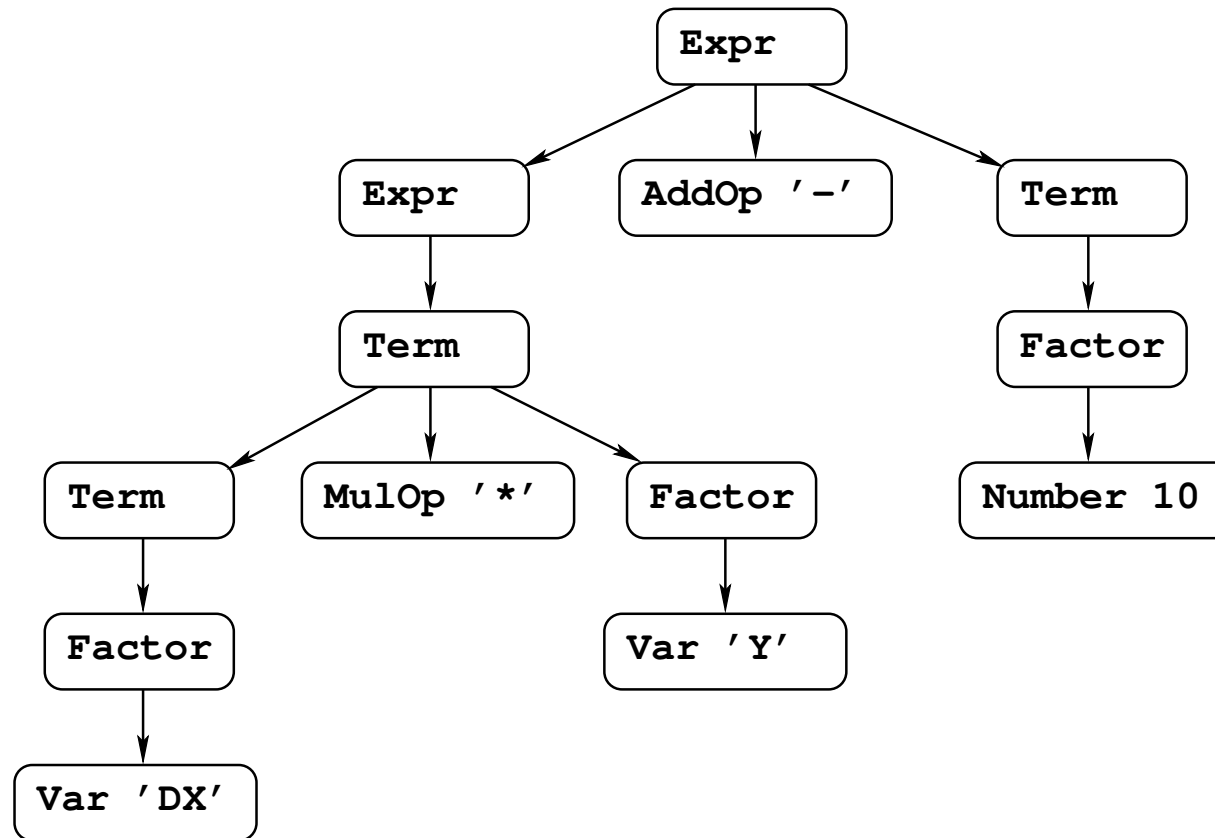
Пример. Лексическая структура языка арифметических выражений (в нотации регулярных выражений):

```
Ws      ::= ( ' ' ) *      – отбрасывается!  
Digit   ::= '0' | '1' | ... | '9'  
Letter  ::= 'A' | 'B' | ... | 'Z'  
Number  ::= Digit Digit *  
Var      ::= Letter ( Letter | Digit ) *  
MulOp   ::= '*' | '/'  
AddOp   ::= '+' | '-'
```

Пример. Синтаксис языка арифметических выражений (БНФ):

```
Expr    ::= Expr AddOp Term | AddOp Term | Term .  
Term    ::= Term MulOp Factor | Factor .  
Factor  ::= '(' Expr ')' | Var | Number .
```

"DX*Y - 10"



Определение. *Лексический домен* – это заданное некоторым формальным способом множество строк.

Лексический домен может быть задан с помощью порождающей грамматики. На практике лексический домен часто является регулярным языком.

Пример. Лексический домен «двоичное число» (БНФ).

`Digit ::= '0' | '1'.`

`Binary ::= Digit | Binary Digit.`

Определение. *Лексема* (lexem) – это фрагмент текста исходной программы, принадлежащий некоторому лексическому домену.

Определение. *Лексическая структура* языка – это множество лексических доменов, на котором задано отношение строгого линейного порядка \succ .

Говорят, что лексический домен D_1 имеет более высокий *приоритет*, чем лексический домен D_2 , если $D_1 \succ D_2$.

Приоритеты лексических доменов нужны для разрешения конфликтов: когда один и тот же участок программы принадлежит нескольким доменам, выбирается домен с наивысшим приоритетом.

Пример. В лексической структуре языка Pascal два домена содержат строку 'begin', а именно: домен ключевых слов и домен идентификаторов. Однако домен ключевых слов имеет более высокий приоритет, чем домен идентификаторов. Поэтому 'begin' в языке Pascal – ключевое слово.

Определение. *Токен* (token) – это описатель лексемы, представляющий собой кортеж вида $\langle \text{domain}, \text{coords}, \text{attr} \rangle$, где domain – лексический домен, которому принадлежит лексема, coords – координаты лексемы в тексте программы, а attr – атрибут токена.

Для вычисления атрибутов токенов задаётся функция $f : D \longrightarrow A$, отображающая лексический домен D в множество атрибутов A .

В простейшем случае A совпадает с D , и f – тождественное отображение, то есть атрибутами токенов являются сами лексемы.

Пример. Атрибуты токенов для домена «двоичных чисел» задаются отображением $\mathcal{B} : \text{Binary} \rightarrow \mathbb{N}$.

$$\begin{aligned}\mathcal{B}['0'] &= 0, \\ \mathcal{B}['1'] &= 1, \\ \mathcal{B}[\text{Binary Digit}] &= 2 \cdot \mathcal{B}[\text{Binary}] + \mathcal{B}[\text{Digit}].\end{aligned}$$

Пример. Вычисление атрибута токена, описывающего лексему '101'.

$$\begin{aligned}\mathcal{B}['101'] &= 2 \cdot \mathcal{B}['10'] + \mathcal{B}['1'] = \\ &= 2 \cdot (2 \cdot \mathcal{B}['1'] + \mathcal{B}['0']) + \mathcal{B}['1'] = 2 \cdot (2 \cdot 1 + 0) + 1 = 5.\end{aligned}$$

Задача лексического анализа. Дано:

$\langle L, \succ \rangle$ — лексическая структура языка;

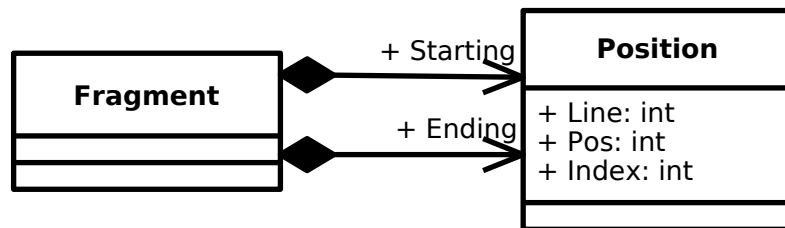
p — непустой суффикс исходной программы, полученный отщеплением от неё некоторого количества лексем.

Требуется найти самый длинный непустой префикс l строки p , принадлежащий хотя бы одному домену из L , и построить токен $\langle D, c, a \rangle$ по следующим принципам:

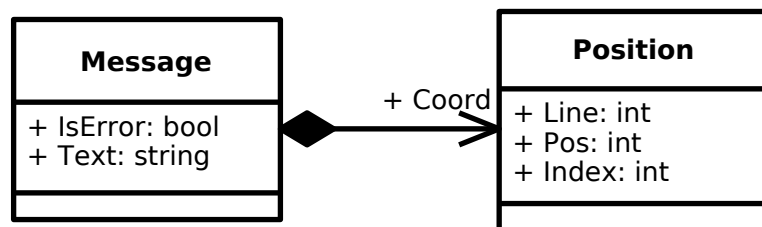
1. Пусть $L' = \{d \in L \mid l \in d\}$, тогда D — максимальный в смысле отношения \succ элемент L' .
2. c — координаты фрагмента l в исходной программе.
3. $a = f(l)$, где f — отображение D в множество атрибутов.

§11. Проектирование объектно-ориентированного лексического анализатора

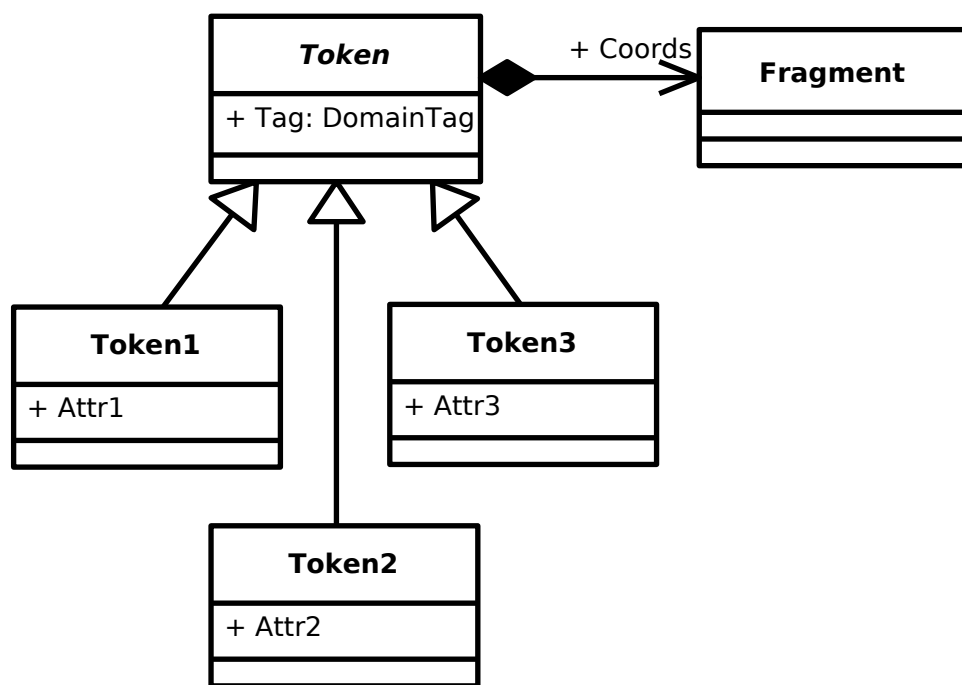
Классы Position и Fragment представляют координаты кодовой точки и координаты фрагмента текста программы, соответственно.



Класс Message хранит информацию о сообщении, порождённом компилятором.

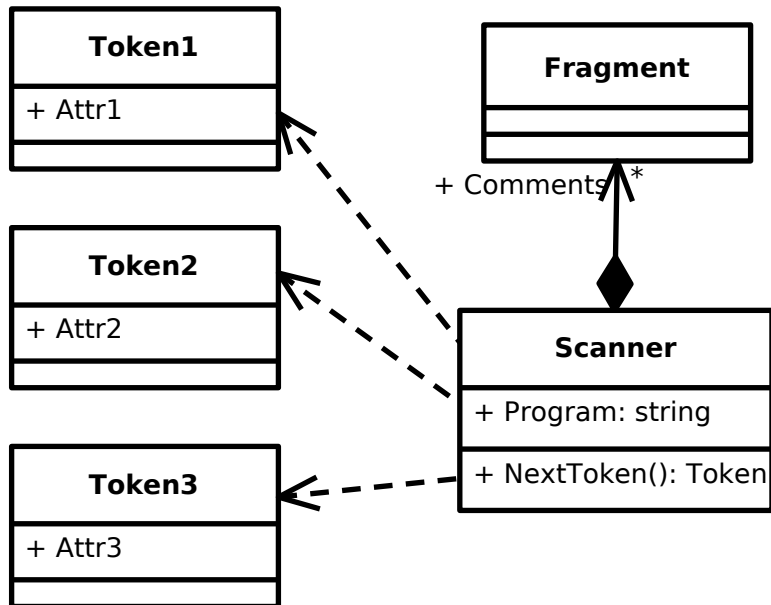


Токены, соответствующие разным лексическим доменам, представлены классами Token1, Token2 и т.п. Эти классы наследуют от абстрактного класса Token.

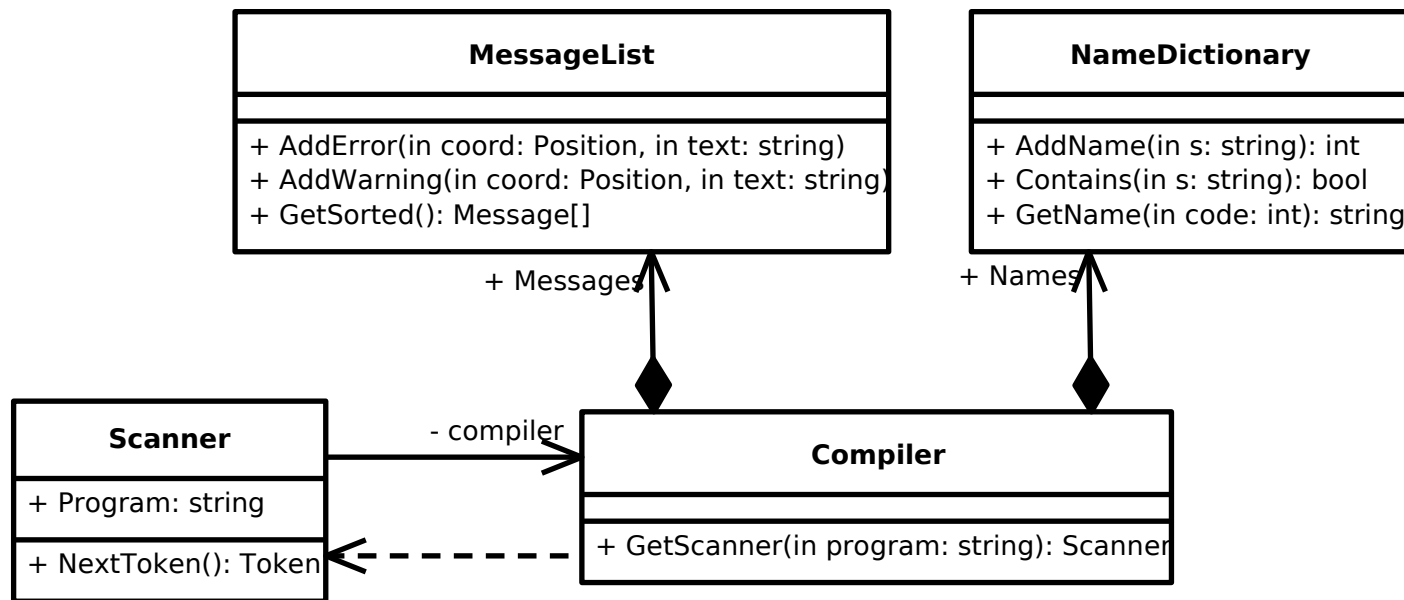


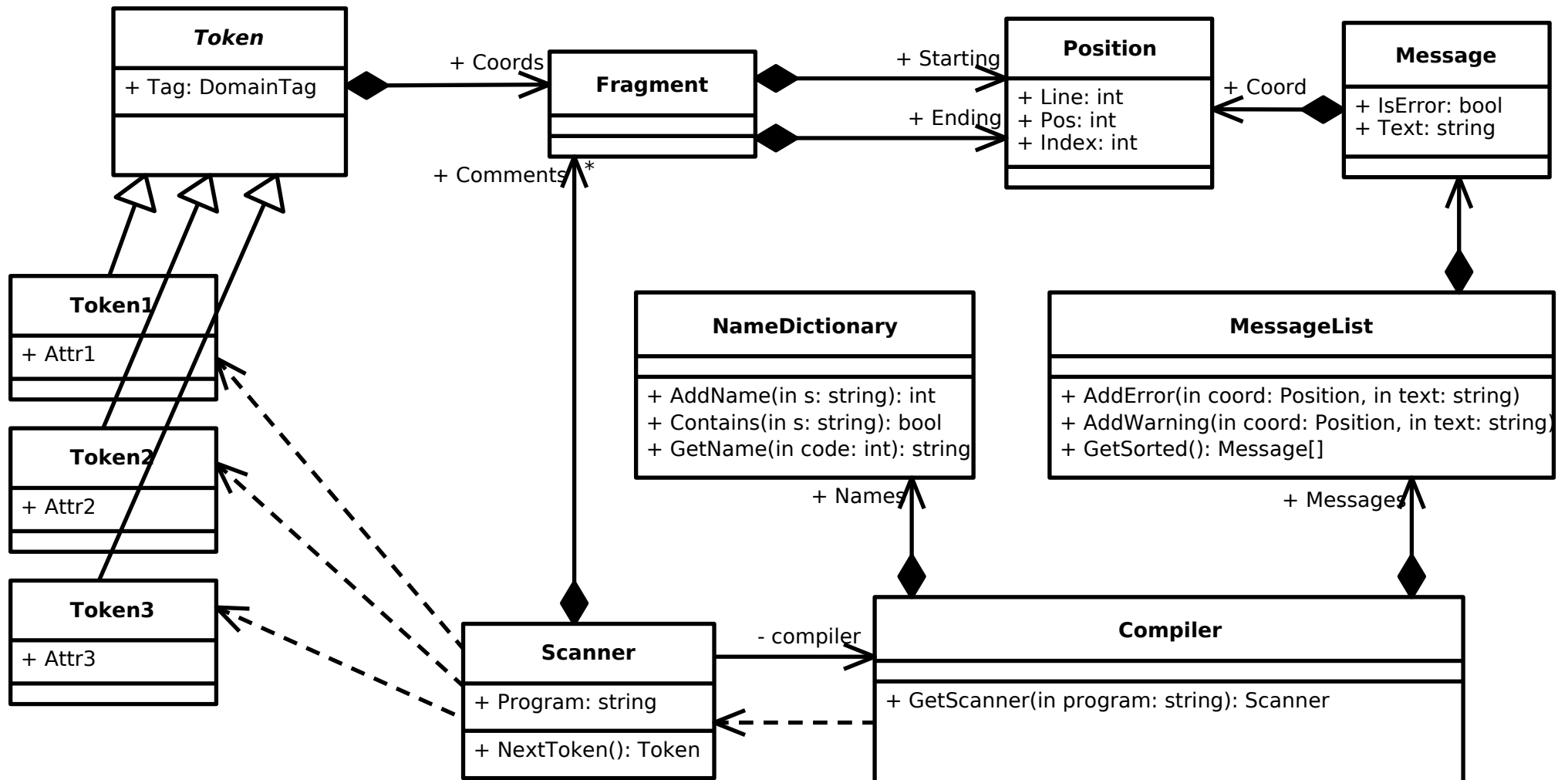
DomainTag – это тип-перечисление, в котором каждому синтаксическому домену соответствует своя константа. Атрибут Tag нужен при реализации на языке, в котором классы – не самоописывающие типы.

Класс Scanner является итератором по токенам. Его метод NextToken осуществляет лексический анализ и попутно ведёт список комментариев.



Объект класса `Compiler` содержит изменяемые таблицы: таблицу сообщений, представленную классом `MessageList`, и словарь имён, представленный классом `NameDictionary`. Эти таблицы заполняются в методе `NextToken` класса `Scanner`.





§12. Реализация объектно-ориентированного лексического анализатора вручную

Возьмём для примера лексическую структуру некоторого языка:

```
Uni          ::= все литеры стандарта Unicode
WhiteSpace   ::= все пробельные литеры | '\r\n' | '\n'
Comment      ::= '/' * (Uni* - (Uni* '/' Uni*)) '/'
LetterUni    ::= все буквенные литеры
DigitUni     ::= все цифровые литеры
Digit10      ::= '0' | '1' | ... | '9'
CharBody     ::= Uni - '\n' - '\\ ' - '\\ '
               | '\\n' | '\\\\ ' | '\\\\ '
Ident        ::= LetterUni (LetterUni | DigitUni)*
Number       ::= Digit10 Digit10*
Char         ::= '\\' CharBody '\\'
Spec         ::= '(' | ')' | '+' | '-' | '*' | '/'
```

Реализуем лексический анализатор для этого языка на C#.

Рекомендация. Вычисление координат кодовых точек – довольно трудоёмкая задача. Поэтому лучше реализовать её в отдельной функции, чем «размазывать» по коду лексического анализатора.

Учитывая эту рекомендацию, мы сделаем объекты класса `Position` итераторами по тексту программы.

При этом доступ к литере, соответствующей текущим координатам, будет осуществляться через свойство `Cp`.

```
Position p = new Position("qwerty");  
Console.WriteLine(p.Cp);    // 'q'
```

Переход к следующей литере реализуем в виде операции `++`:

```
p++;  
Console.WriteLine(p.Cp);    // 'w'
```

```

struct Position: IComparable<Position>
{
    private string text;
    private int line, pos, index;

    public Position(string text)
    {
        this.text = text;
        line = pos = 1;
        index = 0;
    }

    public int Line { get { return line; } }
    public int Pos { get { return pos; } }
    public int Index { get { return index; } }

    public int CompareTo(Position other)
    { return index.CompareTo(other.index); }

    public override string ToString()
    { return "(" + line + "," + pos + ")"; }

    ...
}

```

Рекомендация. Лексический анализатор становится гораздо проще, если дописать в конец текста программы специальную литеру, которая не может встречаться в тексте.

Мы будем использовать UTF-32 в качестве внутренней формы кодирования текста. При этом специальной литерой, обозначающей конец текста программы, у нас будет кодовая точка -1 (0xFFFFFFFF).

Пример. Если не следовать рекомендации:

```
while (cur.Index < program.Length && cur.Cp == ' ')  
    cur++;
```

Пример. Если следовать рекомендации:

```
while (cur.Cp == ' ')  
    cur++;
```

Реализация свойства Cp «понимает» суррогатные пары.

```
struct Position : IComparable<Position>
{
    ...
    public int Cp
    {
        get
        {
            return (index == text.Length) ?
                -1 : Char.ConvertToUtf32(text, index);
        }
    }

    public UnicodeCategory Uc
    {
        get
        {
            if (index == text.Length)
                return UnicodeCategory.OtherNotAssigned;
            return Char.GetUnicodeCategory(text, index);
        }
    }
    ...
}
```

```

struct Position : IComparable<Position>
{
    ...

    public bool IsWhiteSpace
    {
        get
        {
            return index != text.Length &&
                Char.IsWhiteSpace(text , index);
        }
    }

    public bool IsLetter
    {
        get
        {
            return index != text.Length &&
                Char.IsLetter(text , index);
        }
    }

    ...
}

```



```

struct Position : IComparable<Position>
{
    ...

    public bool IsLetterOrDigit
    {
        get
        {
            return index != text.Length &&
                Char.IsLetterOrDigit(text, index);
        }
    }

    public bool IsDecimalDigit
    {
        get
        {
            return index != text.Length &&
                text[index] >= '0' && text[index] <= '9';
        }
    }

    ...
}

```

Рекомендация. Конец файла удобно считать маркером конца строки.

```
struct Position: IComparable<Position>
{
    ...
    public bool IsNewLine
    {
        get
        {
            if (index == text.Length)
                return true;

            if (text[index] == '\r' &&
                index + 1 < text.Length)
                return (text[index + 1] == '\n');

            return (text[index] == '\n');
        }
    }
    ...
}
```

```

struct Position : IComparable<Position>
{
    ...
    public static Position operator ++ (Position p)
    {
        if (p.index < p.text.Length)
        {
            if (p.IsNewLine)
            {
                if (p.text[p.index] == '\r')
                    p.index++;
                p.line++; p.pos = 1;
            }
            else
            {
                if (Char.IsHighSurrogate(p.text[p.index]))
                    p.index++;
                p.pos++;
            }
            p.index++;
        }
        return p;
    }
}

```

```
struct Fragment
{
    public readonly Position Starting , Following ;

    public Fragment(Position starting , Position following)
    {
        Starting = starting ;
        Following = following ;
    }

    public override string ToString()
    {
        return Starting.ToString() + "-" + Following.ToString();
    }
}
```

Из класса Message «исчезли» координаты, потому что мы будем использовать SortedList<Position,Message> для представления списка сообщений компилятора.

```
class Message
{
    public readonly bool IsError;
    public readonly string Text;

    public Message(bool isError , string text)
    {
        IsError = isError;
        Text = text;
    }
}
```

Рекомендация. Для каждого специального символа и для каждого знака операции нужно завести отдельный лексический домен. Кроме того, лексический анализатор должен генерировать специальный токен, обозначающий конец файла. Всё это пригодится на этапе синтаксического анализа.

```
enum DomainTag
{
    IDENT          = 0,    /* Ident */
    NUMBER         = 1,    /* Number */
    CHAR           = 2,    /* Char */
    LPAREN         = 3,    /* Spec '(' */
    RPAREN         = 4,    /* Spec ')' */
    PLUS           = 5,    /* Spec '+' */
    MINUS          = 6,    /* Spec '-' */
    MULTIPLY       = 7,    /* Spec '*' */
    DIVIDE         = 8,    /* Spec '/' */
    END_OF_PROGRAM = 9
};
```

```

abstract class Token
{
    public readonly DomainTag Tag;
    public readonly Fragment Coords;

    protected Token(DomainTag tag ,
                    Position starting , Position following)
    {
        Tag = tag ;
        Coords = new Fragment(starting , following );
    }
}

class IdentToken : Token
{
    public readonly int Code;

    public IdentToken(int code ,
                    Position starting , Position following)
        : base(DomainTag.IDENT, starting , following)
    {
        Code = code;
    }
}

```

```

class NumberToken : Token
{
    public readonly long Value;

    public NumberToken(long val ,
                      Position starting , Position following)
        : base(DomainTag.NUMBER, starting , following)
    {
        Value = val;
    }
}

```

```

class CharToken : Token
{
    public readonly int CodePoint;

    public CharToken(int codePoint ,
                    Position starting , Position following)
        : base(DomainTag.CHAR, starting , following)
    {
        CodePoint = codePoint;
    }
}

```


Рекомендация. Для токенов, соответствующих разным лексическим доменам, но имеющих одинаковые атрибуты, достаточно одного класса.

```
class SpecToken : Token
{
    public SpecToken(DomainTag tag ,
                     Position starting , Position following)
        : base(tag , starting , following)
    {
        Debug.Assert(tag == DomainTag.LPAREN ||
                     tag == DomainTag.RPAREN ||
                     tag == DomainTag.PLUS ||
                     tag == DomainTag.MINUS ||
                     tag == DomainTag.MULTIPLY ||
                     tag == DomainTag.DIVIDE ||
                     tag == DomainTag.END_OF_PROGRAM);
    }
}
```

```

class Scanner
{
    public readonly string Program;

    private Compiler compiler;
    private Position cur;
    private List<Fragment> comments;

    public IEnumerable<Fragment> Comments
    {
        get { return comments; }
    }

    public Scanner(string program, Compiler compiler)
    {
        this.compiler = compiler;
        cur = new Position(Program = program);
        comments = new List<Fragment>();
    }

    ...
}

```

Оставим реализацию метода GetToken «на потом», и обратимся к классу Compiler.

```
class Compiler
{
    private SortedList<Position , Message> messages ;

    private Dictionary<string , int> nameCodes ;
    private List<string> names ;

    public Compiler()
    {
        messages = new SortedList<Position , Message>();
        nameCodes = new Dictionary<string , int>();
        names = new List<string>();
    }
    ...
}
```

Словарь имён реализован в виде двух «таблиц»: nameCodes отображает имена в коды, а names — коды в имена.

```

class Compiler
{
    ...

    public int AddName(string name)
    {
        if (nameCodes.ContainsKey(name))
            return nameCodes[name];
        else
        {
            int code = names.Count;
            names.Add(name);
            nameCodes[name] = code;
            return code;
        }
    }

    public string GetName(int code)
    {
        return names[code];
    }

    ...
}

```

```

class Compiler
{
    ...
    public void AddMessage(bool isErr , Position c, string text)
    {
        messages[c] = new Message(isErr , text);
    }

    public void OutputMessages()
    {
        foreach (KeyValuePair<Position , Message> p in messages)
        {
            Console.WriteLine(p.Value.IsError ? "Error" : "Warning");
            Console.WriteLine(p.Key + ":");
            Console.WriteLine(p.Value.Text);
        }
    }

    public Scanner GetScanner(string program)
    {
        return new Scanner(program , this);
    }
    ...
}

```

Рекомендация. Алгоритм лексического анализа можно организовать по следующей схеме:

```
Token NextToken()  
{  
    while (не конец файла)  
    {  
        пропускаем пробельные литеры;  
        switch (текущая литера)  
        {  
            case первая_литера_домена_1:  
                прочитать литеру; break или return token;  
            case первая_литера_домена_2:  
                прочитать литеру; break или return token;  
            default:  
                if (текущая литера в диапазоне  
                    первых литер домена X)  
                { прочитать литеру; break или return token; }  
                else if ...  
                else ошибка 'unexprcted character'; break;  
        }  
    }  
    return token конца файла;  
}
```

```

public Token NextToken()
{
    while (cur.Cp != -1)
    {
        while (cur.IsWhiteSpace)
            cur++;

        Position start = cur;

        switch (cur.Cp)
        {
            case '(':
                return new SpecToken(DomainTag.LPAREN, start, ++cur);

            case ')':
                return new SpecToken(DomainTag.RPAREN, start, ++cur);

            ...

            case '*':
                return new SpecToken(DomainTag.MULTIPLY, start, ++cur);

            ...
        }
    }
    return new SpecToken(DomainTag.END_OF_PROGRAM, cur, cur);
}

```

Рекомендация. Восстановление при ошибках в лексическом анализаторе осуществляется с помощью сочетания двух приёмов:

- если некоторая литера является для лексического анализатора «непонятной», то он может её пропустить;
- если лексический анализатор обнаруживает, что в распознаваемой лексеме не хватает важной литеры, он может эту литеру добавить.

Случай операции деления осложнён тем, что комментарии тоже начинаются с косой черты.

```
...
case '/':
    if ((++cur).Cp != '*')
        return new SpecToken(DomainTag.DIVIDE, start, cur);

    do
    {
        do
            cur++;
        while (cur.Cp != '*' && cur.Cp != -1);

        cur++;
    }
    while (cur.Cp != '/' && cur.Cp != -1);

    if (cur.Cp == -1)
        compiler.AddMessage(true, cur,
            "end_of_program_found, '/' expected");

    comments.Add(new Fragment(start, ++cur));
    break;
```

```

case  '\ '':
    if ((++cur).IsNewLine)
    {
        compiler.AddMessage(true, cur, "newline_in_constant");
        return new CharToken(0, start, cur++);
    }
    else if (cur.Cp == '\ ')
    {
        compiler.AddMessage(true, cur, "empty_character_literal");
        return new CharToken(0, start, cur++);
    }
    else
    {
        ...
    }

```

```

int ch = cur.Cp;
if (ch == '\\')
    switch ((++cur).Cp)
    {
        case 'n': ch = '\n'; break;
        case '\': ch = '\\'; break;
        case '\\': ch = '\\\'; break;
        default:
            compiler.AddMessage(true, cur,
                "unrecognized_escape_sequence");
            break;
    }

if ((++cur).Cp != '\\')
{
    compiler.AddMessage(true, cur,
        "too_many_characters_in_character_literal");
    while (cur.Cp != '\\' && !cur.IsNewLine)
        cur++;
    if (cur.Cp != '\\')
        compiler.AddMessage(true, cur, "newline_in_constant");
}

return new CharToken(ch, start, ++cur);

```

```
...
default :
    if (cur.IsLetter)
    {
        do
            cur++;
        while (cur.IsLetterOrDigit);

        string name = Program.Substring(start.Index ,
                                         cur.Index - start.Index);
        return new IdentToken(compiler.AddName(name), start, cur);
    }
...
```

Рекомендация. При разработке лексического анализатора следует иметь в виду, что в лексической структуре многих языков присутствуют дополнительные ограничения на «примыкание» лексем друг к другу. Например, числовая константа должна чем-то отделяться от идентификатора.

```

else if (cur.IsDecimalDigit)
{
    long val = cur.Cp - '0';

    try
    {
        while ((++cur).IsDecimalDigit)
            val = checked(val * 10 + cur.Cp - '0');
    }
    catch (System.OverflowException)
    {
        compiler.AddMessage(true, start,
            "integral_constant_is_too_large");
        while ((++cur).IsDecimalDigit)
            ;
    }

    if (cur.IsLetter)
        compiler.AddMessage(true, cur, "delimiter_required");
    return new NumberToken(val, start, cur);
}
else if (cur.Cp != -1)
    compiler.AddMessage(true, cur++, "unexpected_character");
break;

```

Пример работы лексического анализатора.

PROGRAM:

```
/* Expression */ (alpha + 'beta' - '\n')  
    * 66666666666666666666666666666666 /* blah-blah-blah
```

TOKENS:

[illegible]

COMMENTS:

$$\begin{array}{l} (1, 1) - (1, 17) \\ (2, 31) - (2, 49) \end{array}$$

MESSAGES :

```
Error (1,29): too many characters in character literal
Error (2,5): integral constant is too large
Error (2,49): end of program found, '*'/' expected
```

§13. Лексические распознаватели

Определение. Пусть V – алфавит некоторого языка, а $\langle L, \succ \rangle$ – лексическая структура этого языка, тогда *лексический распознаватель* для этого языка представляет собой конечный автомат

$$\langle V, Q, q_0, F, \varphi, \delta \rangle,$$

где

Q – множество состояний автомата;

$q_0 \in Q$ – начальное состояние;

$F \subseteq Q$ – множество заключительных состояний;

$\varphi : F \longrightarrow L$ – функция разметки заключительных состояний;

$\delta : Q \times (V \cup \{\lambda\}) \longrightarrow 2^Q$ – функция переходов.

Пример.

Пусть $V = \{a, b, c, d, 0, 1\}$, $L = \{\text{IDENT}, \text{NUMBER}\}$.

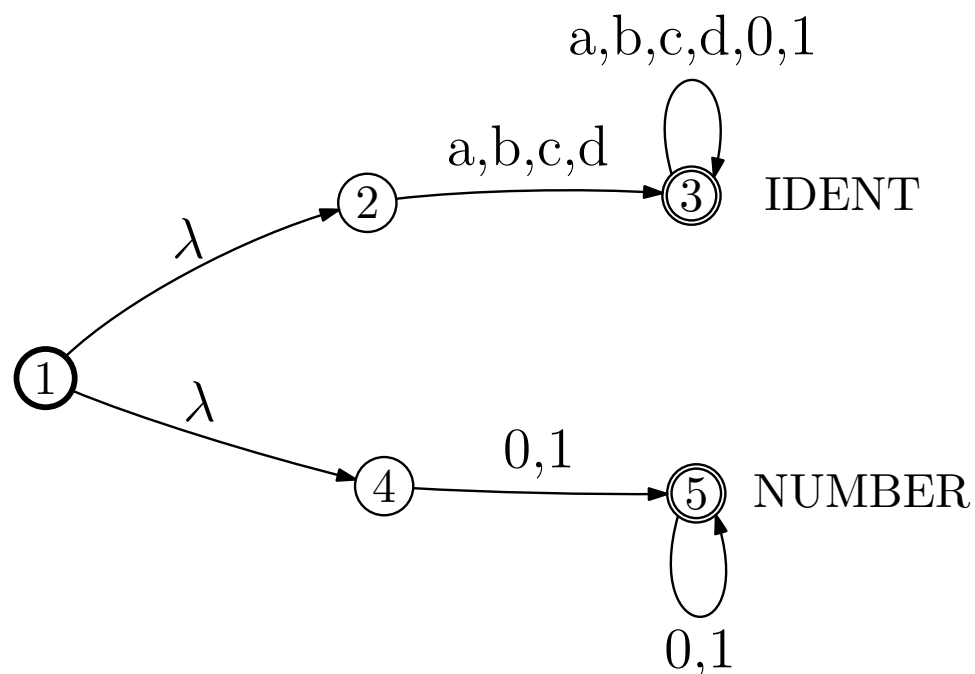
$\text{LETTER} ::= 'a' \mid 'b' \mid 'c' \mid 'd'$

$\text{DIGIT} ::= '0' \mid '1'$

$\text{IDENT} ::= \text{LETTER} (\text{LETTER} \mid \text{DIGIT})^*$

$\text{NUMBER} ::= \text{DIGIT} \text{ DIGIT}^*$

Лексический распознаватель имеет вид:

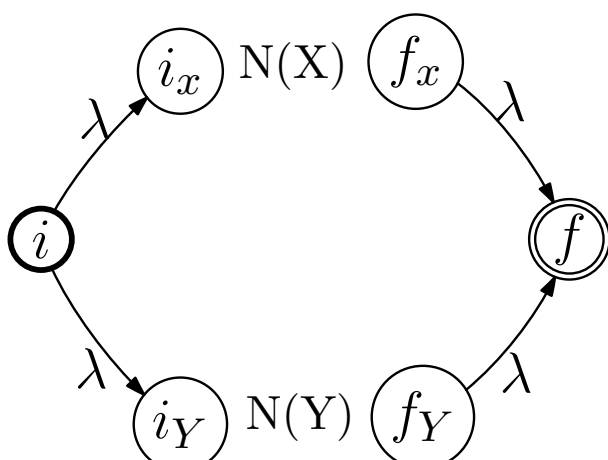
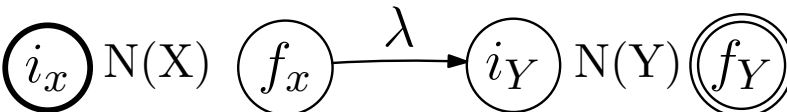
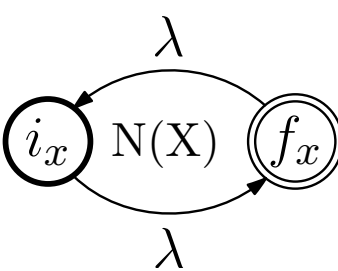


Построение лексического распознавателя по лексической структуре языка, заданной регулярными выражениями, выполняется в три этапа:

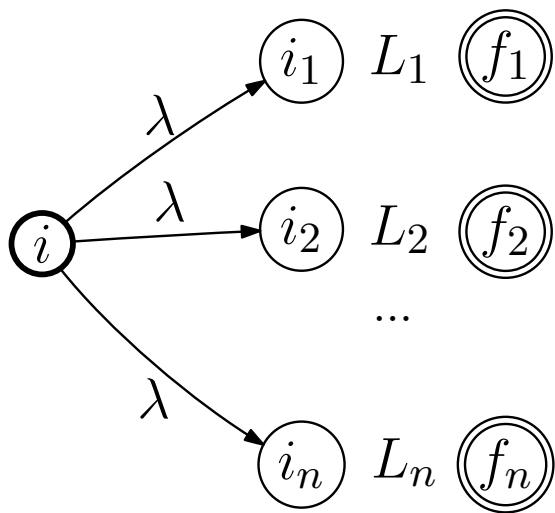
1. построение конечных автоматов для каждого лексического домена;
2. превращение конечных автоматов в лексические распознаватели путем добавления к каждому автомату функции разметки, привязывающей заключительное состояние автомата (оно одно в силу алгоритма построения автомата) к соответствующему лексическому домену;
3. объединение построенных лексических распознавателей в один лексический распознаватель.

Первый этап (построение конечного автомата по регулярному выражению):

Регулярное выражение R	Конечный автомат $N(R)$
λ	
a	
...	

Регулярное выражение R	Конечный автомат N(R)
...	
$X \mid Y$	
$X Y$	
X^*	

Третий этап: имеем набор лексических распознавателей L_1, L_2, \dots, L_n , строим общий распознаватель.



§14. Детерминизация лексического распознавателя

Определение. Лексический распознаватель называется *детерминированным*, если в нём нет дуг с меткой λ , и из любого состояния по любому входному символу возможен переход в точности в одно состояние, т.е. $(\forall q \in Q) (\forall a \in V) (|\delta(q, a)| = 1)$.

Для построения детерминированного распознавателя по недетерминированному применяется известный алгоритм детерминизации конечных автоматов, модифицированный для правильного вычисления функции разметки заключительных состояний.

Состояния детерминированного распознавателя, построенного по недетерминированному, представляют собой подмножества множества состояний недетерминированного распознавателя.

Пример.

Недетерминированный распознаватель	Детерминированный распознаватель
<p>Diagram of a Non-deterministic Finite Automaton (NFA) with states 1, 2, 3, 4, 5. State 1 is the start state. State 3 is the final state (double circle). Transitions: 1 to 2 on λ, 1 to 4 on λ, 2 to 3 on a,b,c,d, 3 to 3 on $a,b,c,d,0,1$, 4 to 5 on $0,1$, 5 to 5 on $0,1$. Labels IDENT and NUMBER are next to states 3 and 5 respectively.</p>	<p>Diagram of a Deterministic Finite Automaton (DFA) with states $\{1, 2, 4\}$, $\{3\}$, $\{5\}$, and \emptyset. State $\{1, 2, 4\}$ is the start state. States $\{3\}$ and $\{5\}$ are final states (double circles). Transitions: $\{1, 2, 4\}$ to $\{3\}$ on a,b,c,d, $\{1, 2, 4\}$ to $\{5\}$ on $0,1$, $\{3\}$ to $\{3\}$ on $a,b,c,d,0,1$, $\{5\}$ to $\{5\}$ on $0,1$, $\{5\}$ to \emptyset on a,b,c,d, \emptyset to \emptyset on $a,b,c,d,0,1$. Labels IDENT and NUMBER are next to states $\{3\}$ and $\{5\}$ respectively.</p>

В алгоритме детерминизации распознавателя используется операция λ -closure : $2^Q \longrightarrow 2^Q$.

λ -closure(T) – множество состояний недетерминированного распознавателя, достижимых из какого-либо состояния $q \in T$ по λ -переходам.

```
 $\lambda$ -closure( $T$ ) {  
    Внести все состояния множества  $T$  в стек  $stack$ ;  
     $Result = T$ ;  
    while ( $stack$  не пуст) {  
        снять со стека верхний элемент  $t$ ;  
        for (каждое состояние  $u$  с дугой от  $t$  к  $u$ ,  
            помеченной  $\lambda$ ) {  
            if ( $u \notin Result$ ) {  
                добавить  $u$  к  $Result$ ;  
                поместить  $u$  в  $stack$ ;  
            }  
        }  
    }  
    return  $Result$ ;  
}
```

```

Det  ( $\langle L, \succ \rangle$  ,   $\langle V, Q, q_0, F, \varphi, \delta \rangle$ ) {
   $q'_0 = \lambda\text{-closure}(\{q_0\})$ ;   $F' = \emptyset$ ;   $\varphi' = \emptyset$ ;   $\delta' = \emptyset$ ;
   $Q' = \{ \text{непомеченное состояние } q'_0 \}$ ;
  while (в  $Q'$  имеется непомеченное состояние  $T$ ) {
    пометить  $T$ ;
    for (каждый символ  $a \in V$ ) {
       $U = \lambda\text{-closure}(\bigcup_{q \in T} \delta(q, a))$ ;

      if ( $U \notin Q'$ ) {
        Добавить  $U$  как
        непомеченное состояние в  $Q'$ ;
        if ( $U \cap F \neq \emptyset$ ) {
          Добавить  $U$  в  $F'$ ;
           $\varphi'(U) = \max \{ \varphi(q) \mid q \in U \cap F \}$ ;
        }
      }
       $\delta'(T, a) = U$ ;
    }
  }
}
return  $\langle V, Q', q'_0, F', \varphi', \delta' \rangle$ ;
}

```

Пример.

Пусть $V = \{a, b, c, d, 0, 1\}$, $L = \{\text{IDENT}, \text{AB}\}$, $\text{AB} \succ \text{IDENT}$.

Недетерминированный распознаватель	Детерминированный распознаватель
<pre>graph LR 1((1)) -- λ --> 4((4)) 4 -- a --> 5((5)) 5 -- b --> 6(((6))) 1 -- λ --> 2((2)) 2 -- "a,b,c,d" --> 3(((3))) 3 -- "a,b,c,d,0,1" --> 3</pre>	<pre>graph LR S(({1, 2, 4})) -- a --> A((({3, 5}))) S -- "b,c,d" --> B((({3}))) S -- "0,1" --> E((∅)) A -- b --> C((({3, 6}))) A -- "a,c,d,0,1" --> B C -- "a,b,c,d,0,1" --> B E -- "a,b,c,d,0,1" --> E B -- "a,b,c,d,0,1" --> B</pre>

§15. Представление лексического распознавателя в программе

Применяют три способа представления функции переходов лексического распознавателя:

- 1. таблица переходов;
- 2. списки переходов;
- 3. алгоритм переходов.

Пример распознавателя	Коды символов алфавита
<pre>graph LR 0((0)) -- "a IDENT" --> 1((1)) 0 -- "x ENDOFFPROGRAM" --> 4(((4))) 0 -- "b,c,d IDENT" --> 3(((3))) 1 -- "x" --> m1((-1)) 1 -- "b AB" --> 2((2)) 1 -- "a,c,d,0,1" --> 3 2 -- "a,b,c,d,0,1" --> 3 3 -- "x" --> m1 3 -- "a,b,c,d,0,1" --> 3 m1 -- "a,b,c,d,0,1,x" --> m1 m1 -- "x" --> 2 m1 -- "x" --> 3</pre>	<p>a – 0, b – 1, c – 2, d – 3, 0 – 4, 1 – 5, x – 6</p>

Применение таблицы переходов:

```
private int [,] table = new int [,]  
{  
    /*      a      b      c      d      0      1      x */  
    /* state 0 */ { 1, 3, 3, 3, -1, -1, 4 },  
    /* state 1 */ { 3, 2, 3, 3, 3, 3, -1 },  
    /* state 2 */ { 3, 3, 3, 3, 3, 3, -1 },  
    /* state 3 */ { 3, 3, 3, 3, 3, 3, -1 },  
    /* state 4 */ { -1, -1, -1, -1, -1, -1, -1 }  
};
```

```
private DomainTag[] final = new DomainTag []  
{  
    /* state 0 */ NOT_FINAL,  
    /* state 1 */ IDENT,  
    /* state 2 */ AB,  
    /* state 3 */ IDENT,  
    /* state 4 */ ENDOFPROGRAM  
};
```

Определение. Символы x и y алфавита детерминированного лексического распознавателя $\langle V, Q, q_0, F, \varphi, \delta \rangle$ называются *эквивалентными*, если для любого состояния $q \in Q$ выполняется $\delta(q, x) = \delta(q, y)$.

Для уменьшения таблицы переходов применяют *метод факторизации алфавита* лексического распознавателя, который заключается в том, что группе эквивалентных символов соответствует один столбец таблицы.

§16. Генератор лексических анализаторов flex

flex – вариант генератора лексических анализаторов lex.

Мы будем использовать flex 2.5.35.

Описание лексического анализатора на flex имеет вид:

```
Секция определений
%%
Секция правил
%%
Пользовательский код
```

Генератор flex транслирует описание в код лексического анализатора на языке C – lex.yy.c.

Все директивы генератора flex должны располагаться в строчке, начиная с первой позиции. Содержимое секции пользовательского кода, а также все строчки, начинающиеся с пробелов, и строчки, заключённые в %{ и %}, копируются в выходной файл.

Секция определений содержит объявления именованных регулярных выражений и стартовых условий.

Именованные регулярные выражения имеют вид

Имя РегулярноеВыражение

Стартовые условия объявляются с помощью директивы %x:

%x ИмяУсловия1 ИмяУсловия2 ... ИмяУсловияN

Стартовые условия позволяют переключать лексический анализатор в специальные режимы распознавания: режим распознавания комментариев, режим распознавания строковых констант и т.п.

Регулярные выражения задаются на расширенном языке регулярных выражений:

Регулярное выражение	Описание
x	Соответствует литере 'x'.
.	Любая литера, кроме перевода строки.
[xyz]	Класс литер, распознающий литеры 'x', 'y' или 'z'.
[abj-oZ]	Класс литер с интервалом, распознаёт литеры 'a', 'b', любые литеры в интервале от 'j' до 'o', а также литеру 'Z'.
[^A-Z]	Инвертированный класс литер, содержащий все литеры, кроме перечисленных.
[a-z]{-}[aeiou]	Вычитания классов литер; в данном случае получаются латинские прописные согласные.
...	

Регулярное выражение	Описание
...	
r^*	Ноль или более вхождений r , где r – произвольное регулярное выражение.
r^+	Один или более вхождений r .
$r^?$	Ноль или одно вхождение r .
$r\{2,5\}$	От двух до пяти вхождений r .
$r\{2,\}$	Два или более вхождения r .
$r\{4\}$	Ровно четыре вхождения r .
$\{\text{Имя}\}$	Подстановка именованного регулярного выражения, определённого ранее.
$\backslash X$	Escape-последовательность; в качестве X допустимы 'a', 'b', 'f', 'n', 'r', 't', 'v' (их смысл аналогичен Escape-последовательностям языка C) или любой символ, используемый в качестве метасимвола расширенного языка регулярных выражений ('(', ')', '[', ']', '{', '}', '.', '*', '+', '?', '\', и т.д.).
...	

Регулярное выражение	Описание
...	
\123	Литера с восьмиричным кодом 123.
\x2A	Литера с шестнадцатеричным кодом 2A.
(r)	Регулярное выражение r (круглые скобки служат для задания приоритета).
rs	Конкатенация регулярных выражений r и s.
r s	Либо r, либо s.
r/s	Вхождение r, за которым следует вхождение s.
^r	Вхождение r, расположенное в начале строки.
r\$	Вхождение r, расположенное в конце строки.
<<EOF>>	Конец файла.

Секция правил содержит список правил, каждое из которых определяет некоторую лексему.

Если лексема не подразумевает никакой реакции на её обнаружение, то правило состоит просто из одного регулярного выражения. Например:

```
%%  
[\\n\\t ]+
```

Код, который нужно выполнить при распознавании лексемы, записывается в виде *действия*, которое отделяется от регулярного выражения одним или несколькими пробелами. Например:

```
%%  
[A-Z][A-Z0-9]*    printf("Identifier\\n");  
  
[0-9]+            {  
                    /* Действие может занимать  
                     несколько строк. */  
                    printf("Number\\n");  
                    }
```

В секции правил разрешено использование регулярных выражений, зависящих от стартовых условий:

Регулярное выражение	Описание
$\langle n \rangle r$	Вхождение r при включённом стартовом условии с именем n .
$\langle n_1, \dots, n_N \rangle r$	Вхождение r при включённых стартовых условиях с именами n_1, \dots, n_N .
$\langle * \rangle r$	Вхождение r при любых стартовых условиях.
$\langle n_1, \dots, n_N \rangle \langle \langle \text{EOF} \rangle \rangle$	Конец файла при включённых стартовых условиях с именами n_1, \dots, n_N .

Изначально включено стартовое условие INITIAL, которое в правилах не указывается. Переключение стартовых условий осуществляется путём вызова макроса BEGIN. Например:

```
%x  STRING
```

```
%%
```

```
\ "          printf("Opening quote\n"); BEGIN(STRING);  
<STRING>[^\n"]*  printf("String body\n");  
<STRING>\n      printf("Error: newline in string");  
<STRING>\ "      printf("Closing quote\n"); BEGIN(0);
```

Вызов BEGIN(0) опять включает стартовое условие INITIAL.

flex по набору правил генерирует тело функции `yylex()`.

Прототип функции `yylex()` зависит от опций, задаваемых либо при вызове генератора flex, либо с помощью директивы `%option`.

Мы будем использовать следующий набор опций:

```
%option noyywrap bison-bridge bison-locations
```

Из-за включённого «моста» с генератором парсеров bison и включённой поддержкой отслеживания координат лексем, прототип функции `yylex()` в нашем случае будет таким:

```
int yylex(YSTYPE *yylval, YYLTYPE *yylloc);
```

Функция `yylex()` возвращает тег распознанного токена (или 0 в случае конца программы). Кроме того, атрибуты токена возвращаются через параметр `yylval`, а координаты токена – через параметр `yylloc`.

Здесь `YSTYPE` – тип атрибутов токена, `YYLTYPE` – тип координат лексем. Эти типы определяются пользователем.

Внутри действий доступны следующие переменные:

Переменная	Тип	Описание
yyltext	char*	[in] Строка, представляющая лексему.
yyleng	int	[in] Длина лексемы в литерлах.
yylval	YYSTYPE*	[out] Атрибуты возвращаемого токена.
yylloc	YYLTYPE *	[out] Координаты лексемы.

Если в начале каждого действия требуется выполнить одни и те же операции (например, вычислить координаты лексемы), то эти операции можно записать в макрос YY_USER_ACTION.

```
%option noyywrap bison-bridge bison-locations
```

```
%{
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#define TAG_IDENT 1
```

```
#define TAG_NUMBER 2
```

```
#define TAG_CHAR 3
```

```
#define TAG_LPAREN 4
```

```
#define TAG_RPAREN 5
```

```
#define TAG_PLUS 6
```

```
#define TAG_MINUS 7
```

```
#define TAG_MULTIPLY 8
```

```
#define TAG_DIVIDE 9
```

```
char *tag_names[] =
```

```
{
```

```
    "END_OF_PROGRAM", "IDENT", "NUMBER",
```

```
    "CHAR", "LPAREN", "RPAREN", "PLUS",
```

```
    "MINUS", "MULTIPLY", "DIVIDE"
```

```
};
```

```

struct Position
{
    int line, pos, index;
};

void print_pos(struct Position *p)
{
    printf("(%d,%d)",p->line,p->pos);
}

struct Fragment
{
    struct Position starting, following;
};

typedef struct Fragment YYLTYPE;

void print_frag(struct Fragment *f)
{
    print_pos(&(f->starting));
    printf("-");
    print_pos(&(f->following));
}

```

```
union Token
{
    char *ident; /* Бандитизм: должно быть int code; */
    long num;
    char ch;
};

typedef union Token YYSTYPE;
```



```

int continued;
struct Position cur;

#define YY_USER_ACTION \
{ \
    int i; \
    if (! continued) \
        yylloc->starting = cur; \
    continued = 0; \
    \
    for (i = 0; i < yyleng; i++) \
    { \
        if (yytext[i] == '\n') \
        { \
            cur.line++; \
            cur.pos = 1; \
        } \
        else \
        { \
            cur.pos++; \
            cur.index++; \
        } \
        \
        yylloc->following = cur; \
    }

```

```

void init_scanner(char *program)
{
    continued = 0;
    cur.line = 1;
    cur.pos = 1;
    cur.index = 0;
    yy_scan_string(program);
}

```

```

void err(char *msg)
{
    /* Бандитизм: ошибки нужно класть в список ошибок. */
    printf("Error ");
    print_pos(&cur);
    printf(": %s\n", msg);
}
%}

```

```

LETTER    [a-zA-Z]
DIGIT     [0-9]
IDENT     {LETTER}({LETTER}|{DIGIT})*
NUMBER    {DIGIT}+

```

```

%x          COMMENTS  CHAR_1  CHAR_2

```

%%

[\n\t]+

\/*

<COMMENTS>[~]*

<COMMENTS>*\

```
BEGIN(COMMENTS); continued = 1;
```

```
continued = 1;
```

```
{
```

```
    /* Бандитизм: координаты комментариев
```

```
       нужно класть в список комментариев. */
```

```
    print_frag(yylloc);
```

```
    printf(" comment\n");
```

```
    BEGIN(0);
```

```
}
```

<COMMENTS>*

```
continued = 1;
```

<COMMENTS><<EOF>>

```
{
```

```
    err("end of program found, '*/' expected");
```

```
    return 0;
```

```
}
```

```

\ (      return TAG_LPAREN;
\ )      return TAG_RPAREN;
\ +      return TAG_PLUS;
-        return TAG_MINUS;
\ *      return TAG_MULTIPLY;
\ /      return TAG_DIVIDE;

{IDENT}  {
            /* Бандитизм: здесь нужно поместить
               идентификатор в словарь имён. */
            yylval->ident = yytext;
            return TAG_IDENT;
        }

{NUMBER} {
            /* Бандитизм: нет проверки на переполнение. */
            yylval->num = atoi(yytext);
            return TAG_NUMBER;
        }

\ '      BEGIN(CHAR_1);   continued = 1;

.        err("unexpected character");

```

```

<CHAR_1 , CHAR_2 > \n      {
                                err("newline in constant");
                                BEGIN(0);
                                yylval->ch = 0;
                                return TAG_CHAR;
                                }

<CHAR_1 > \\n                yylval->ch = '\n';   BEGIN(CHAR_2);   continued = 1;
<CHAR_1 > \\\'                yylval->ch = '\'';   BEGIN(CHAR_2);   continued = 1;
<CHAR_1 > \\\'\'              yylval->ch = '\\\'';   BEGIN(CHAR_2);   continued = 1;
<CHAR_1 > \\\'\'\'            {
                                err("unrecognized Escape sequence");
                                yylval->ch = 0;
                                BEGIN(CHAR_2);
                                continued = 1;
                                }

<CHAR_1 > \'                  {
                                err("empty character literal");
                                BEGIN(0);
                                yylval->ch = 0;
                                return TAG_CHAR;
                                }

<CHAR_1 > .                  yylval->ch = yytext[0]; BEGIN(CHAR_2); continued = 1;
<CHAR_2 > \'                  BEGIN(0); return TAG_CHAR;
<CHAR_2 > [^\n\']*           err("too many characters in literal"); continued = 1;

```

```

%%

#define PROGRAM "/* Expression */ (alpha + 'beta' - '\\n')\n" \
               " * 666666666 /* blah-blah-blah "

int main()
{
    int tag;
    YYSTYPE value;
    YYLTYPE coords;

    init_scanner(PROGRAM);

    do
    {
        tag = yylex(&value,&coords);
        if (tag != 0)
        { ... }
    }
    while (tag != 0);

    return 0;
}

```

(1,1)-(1,17) comment
(1,18)-(1,19) LPAREN
(1,19)-(1,24) IDENT alpha
(1,25)-(1,26) PLUS
Error (1,32): too many characters in literal
(1,27)-(1,33) CHAR 98
(1,34)-(1,35) MINUS
(1,36)-(1,40) CHAR 10
(1,40)-(1,41) RPAREN
(2,3)-(2,4) MULTIPLY
(2,5)-(2,30) NUMBER 666666666
Error (2,49): end of program found, '*/' expected