Лекция 1

Введение в конструирование компиляторов

§1. Компиляторы и интерпретаторы: основные определения

Определение. *Компилятор* (compiler) из языка S в язык T – это программа, осуществляющая перевод программ с языка S на язык T. При этом:
S – *исходный язык* (source language);
T – *целевой язык* (target language, object language);
язык H, на котором написан компилятор, – *язык реализации* (implementation language, host language).

Мы будем использовать запись $S \underset{H}{\rightarrow} T$ для обозначения компилятора из языка S в язык T, написанного на языке H.

Транслятор (translator) – синоним компилятора.

Определение. *Интерпретатор* (interpreter) для некоторого языка L – это программа, осуществляющая выполнение программ, написанных на языке L.

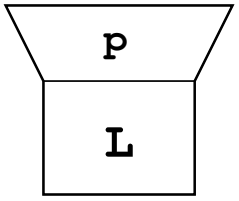
Компиляторы и интерпретаторы близки по структуре.

Компилятор из A в B можно считать интерпретатором языка A' такого, что синтаксис A' совпадает с синтаксисом A, а смысл программы p, написанной на A', заключается в порождении программы на языке B, вычисляющей ту же функцию, что и программа p, если понимать её написанной на языке A.

§2. Т-диаграммы

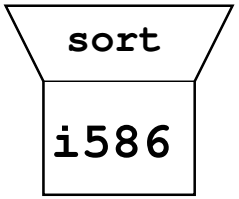
Т-диаграммы позволяют визуализировать те взаимодействия программ, в которые вовлечены компиляторы и интерпретаторы.

Программа p на языке L:



Машина, выполняющая язык M:

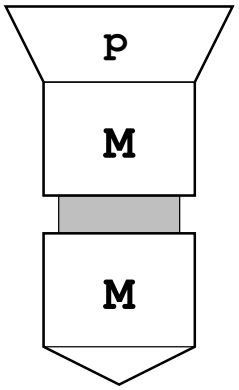
Пример:



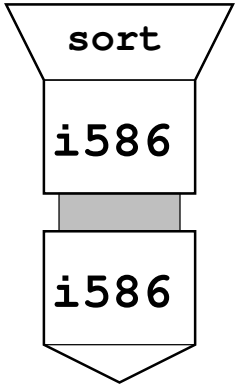
Пример:

Для того чтобы выполнить программу на некоторой машине, язык, на котором написана программа, должен совпадать с языком, который понимает машина.

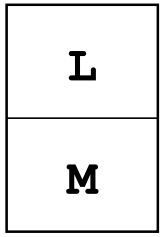
Выполнение программы p:



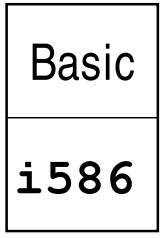
Пример:



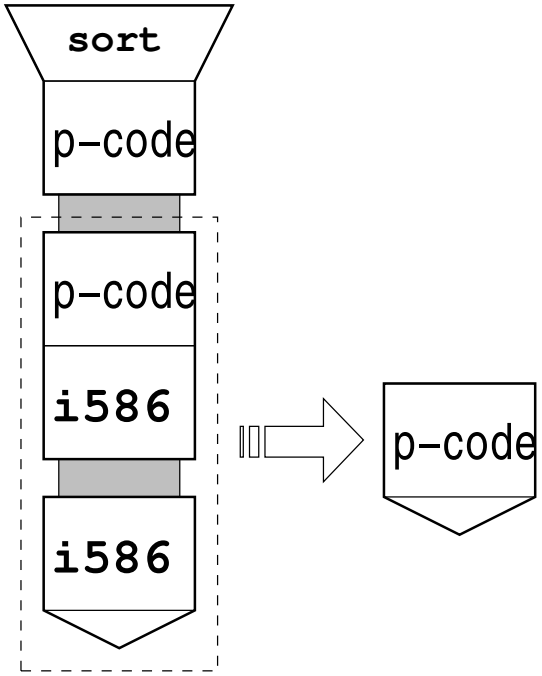
Интерпретатор языка L, написанный на языке M:



Пример:

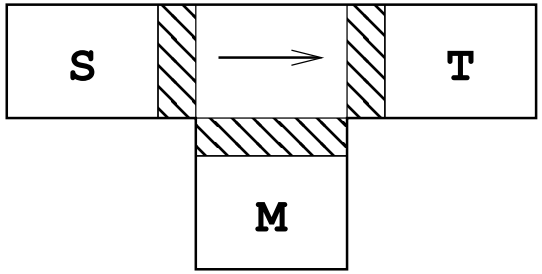


Пример (виртуальная p-машина):

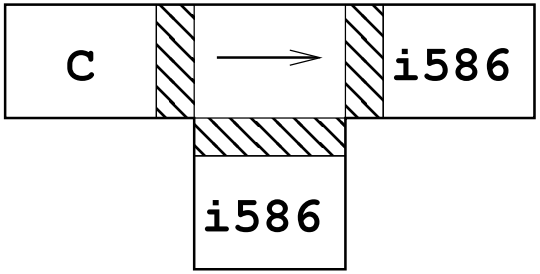


Определение. *Виртуальная машина* для языка L – это сочетание интерпретатора языка L, написанного на языке M, и машины, выполняющей язык M.

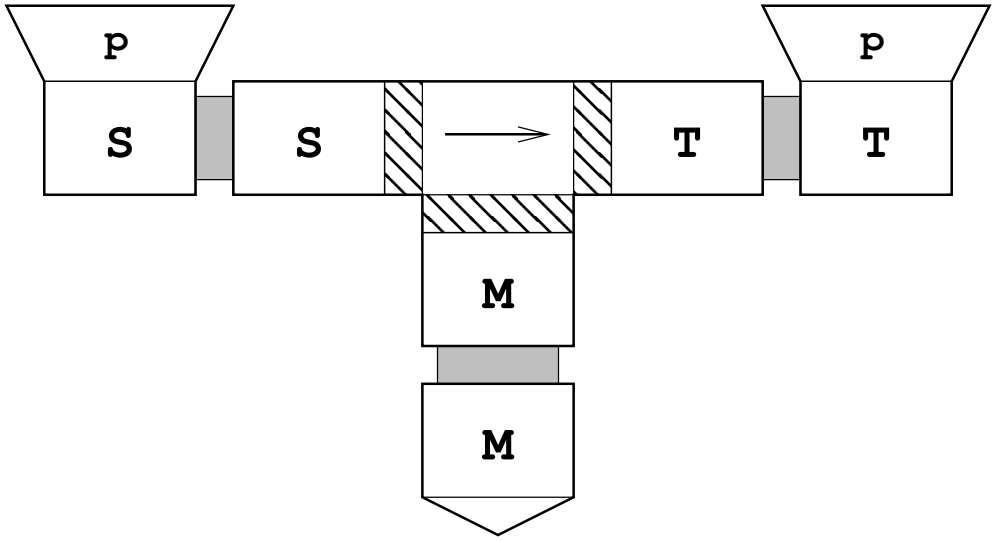
Компилятор $S \underset{M}{\rightarrow} T$:



Пример:

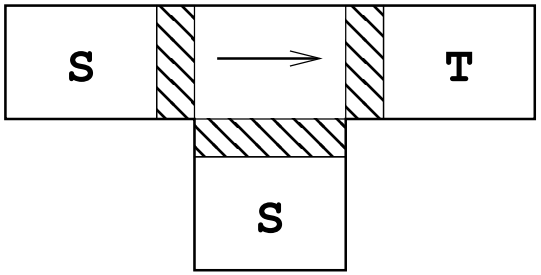


Компилятор $S \underset{M}{\rightarrow} T$, будучи запущен на машине, выполняющей язык M, переводит программы с языка S на язык T.



§3. Самоприменимые компиляторы: раскрутка и перенос

Определение. Компилятор $S \underset{S}{\rightarrow} T$ называется *самоприменимым* (self-applicable).

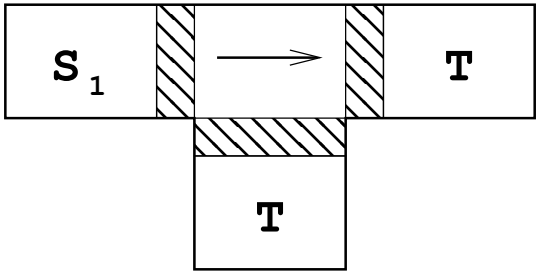


Если для языка S не существует другого компилятора или интерпретатора, то запуск самоприменимого компилятора является непростым делом.

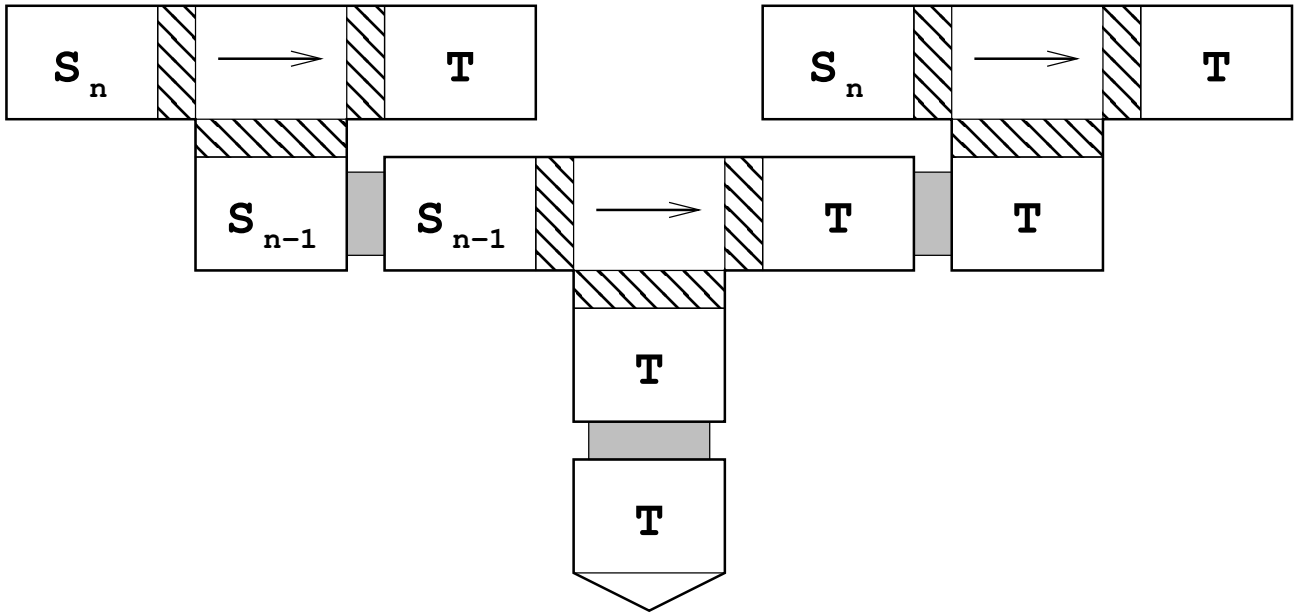
Определение. *Раскрутка* (bootstrapping) – это итерационный процесс создания самоприменимого компилятора $S \underset{S}{\rightarrow} T$, используемый в случае, если есть возможность запуска программ на языке T, а для программ на языке S такой возможности нет.

На каждом шаге раскрутки получается компилятор $S_i \underset{T}{\rightarrow} T$ для всё большего и большего подмножества S_i языка S.

Шаг 1.

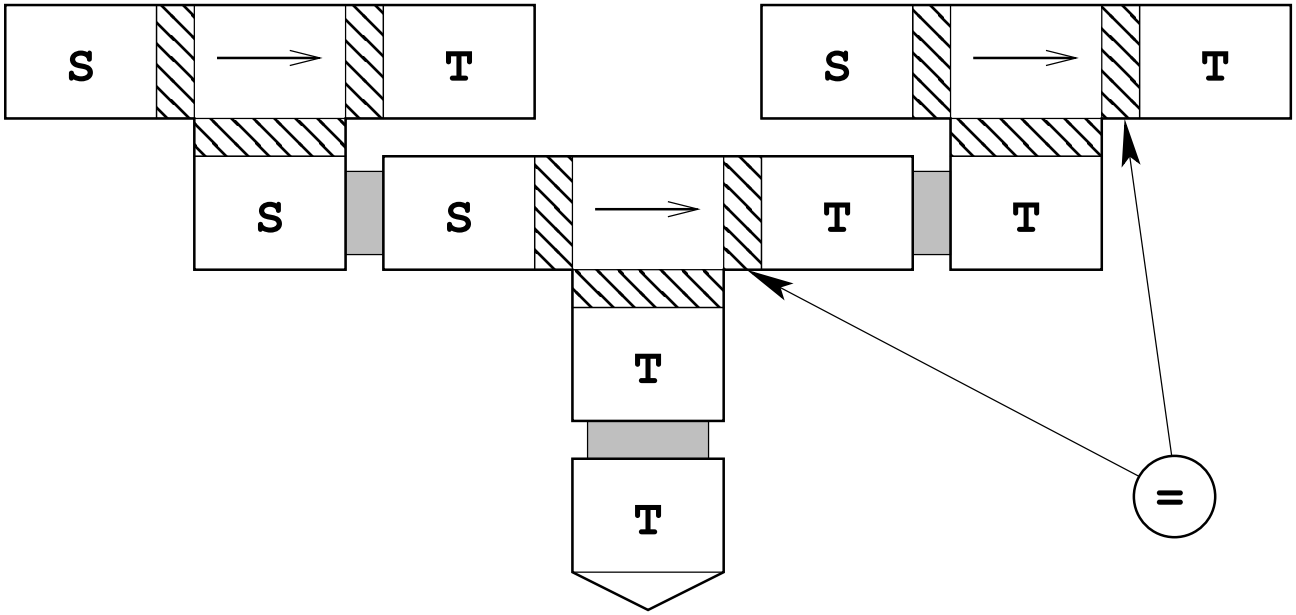


Шаг n.



$S_1 \subset \ldots \subset S_{n-1} \subset S_n \subset S$

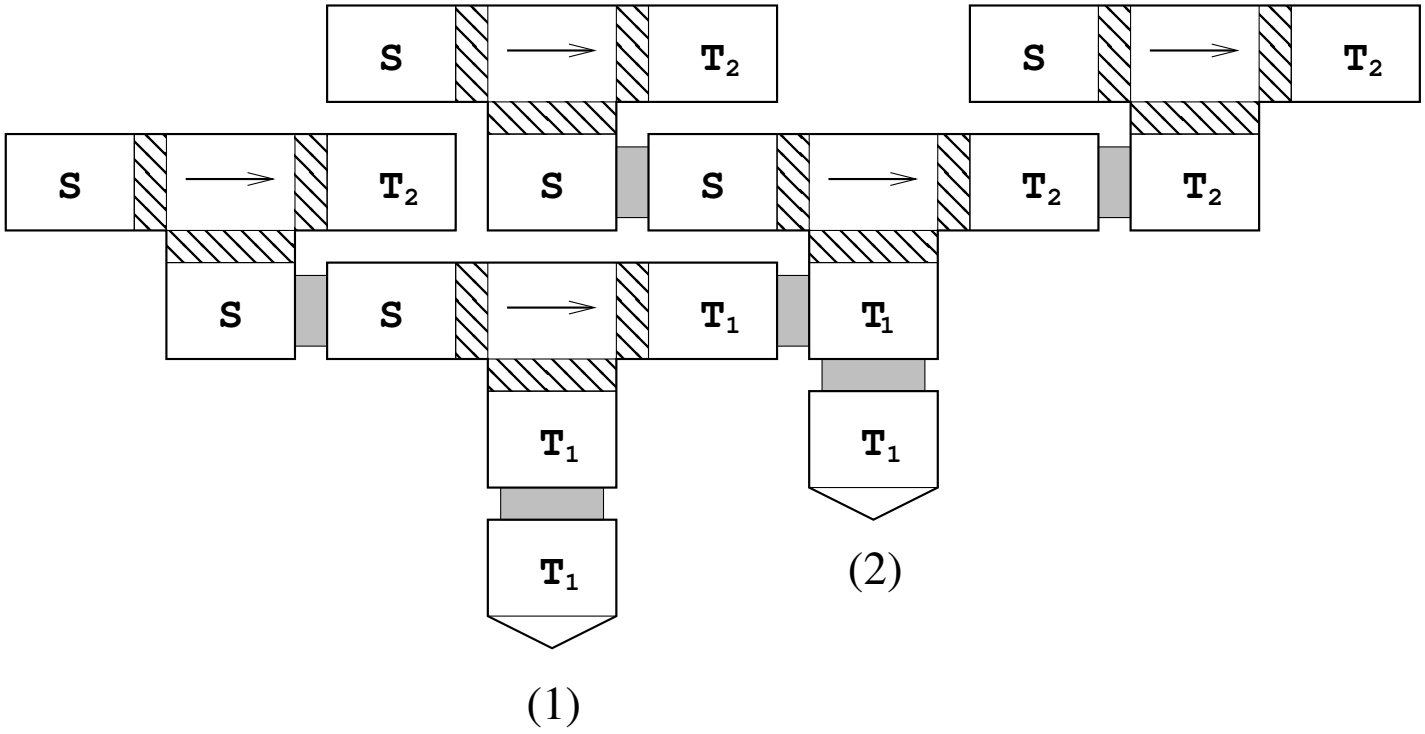
Определение. *Раскрученный самоприменимый компилятор* языка S на платформе T – это пара компиляторов $\left(S \underset{S}{\to} T,\ S \underset{T}{\to} T\right)$ такая, что применение компилятора $S \underset{T}{\to} T$ к компилятору $S \underset{S}{\to} T$ даёт $S \underset{T}{\to} T$.



Для переноса раскрученного самоприменимого компилятора

$$\left(S \underset{S}{\rightarrow} T_1,\ S \underset{T_1}{\rightarrow} T_1 \right)$$

на платформу T_2 достаточно на основе $S \underset{S}{\rightarrow} T_1$ разработать $S \underset{S}{\rightarrow} T_2$, а затем выполнить два шага:



Компилятор $S \underset{T_1}{\rightarrow} T_2$ называется *кросскомпилятором*.

§4. Внедрение «троянских коней» с помощью самоприменимых компиляторов

По статье:
Ken Thompson. *Reflections on Trusting Trust* // Communications of the ACM. Vol. 27, №8. August 1984.

В статье рассматривается раскрученный самоприменимый компилятор языка C:

$$\left(\mathbf{C} \underset{\mathbf{C}}{\rightarrow} \mathbf{Asm},\ \mathbf{C} \underset{\mathbf{Asm}}{\rightarrow} \mathbf{Asm}\right)$$

«Идеализированный» фрагмент компилятора C $\underset{C}{\rightarrow}$ Asm, обрабатывающий Escape-последовательности:

```
c = next();
if (c != '\\')
    return c;

c = next();
switch (c)
{
case '\\': return '\\';
case 'n':  return '\n';
}
```

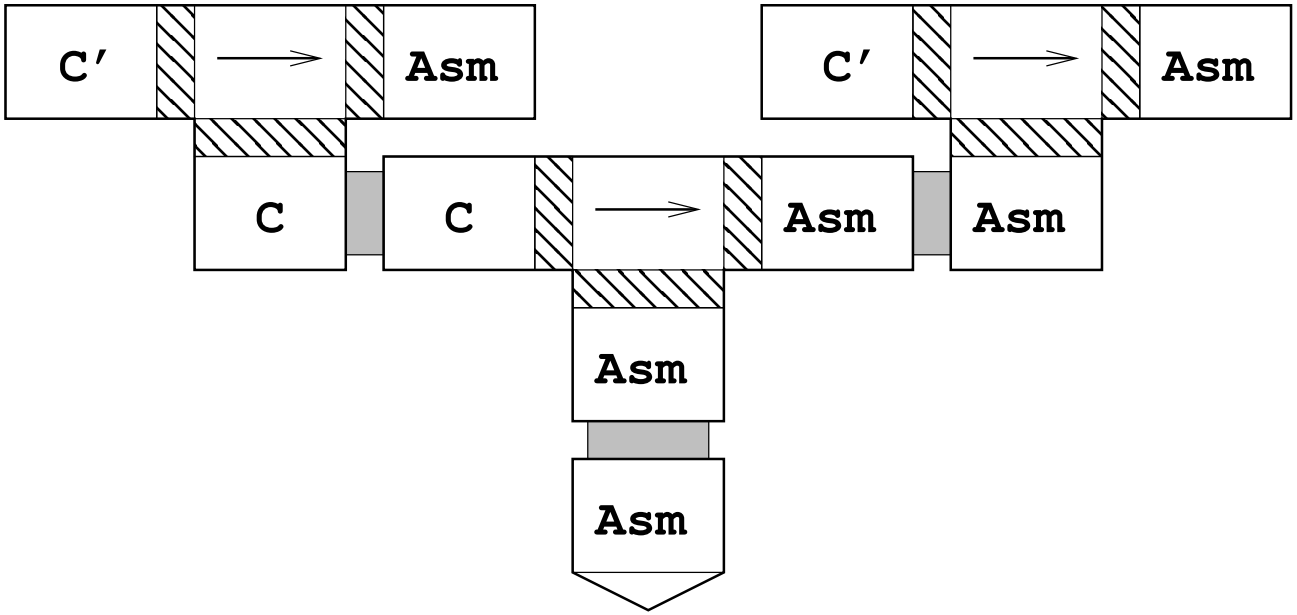
Попробуем добавить в $C \underset{C}{\rightarrow} Asm$ последовательность \t:

```
c = next();
if (c != '\\')
    return c;

c = next();
switch (c)
{
case '\\': return '\\';
case 'n':  return '\n';
case 't':  return 9;
}
```

В результате получим компилятор $C' \underset{C}{\rightarrow} Asm$.

Выполняем раскрутку:



Теперь мы можем «переписать» $C' \underset{C}{\rightarrow} \text{Asm}$ на C':

```
c = next();
if (c != '\\')
    return c;

c = next();
switch (c)
{
case '\\': return '\\';
case 'n':  return '\n';
case 't':  return '\t';
}
```

В результате получим самоприменимый компилятор $C' \underset{C'}{\rightarrow} \text{Asm}$, в тексте которого отсутствует ASCII-код символа '\t'. Дело в том, что этот ASCII-код «ушёл» в $C' \underset{\text{Asm}}{\rightarrow} \text{Asm}$!

Используем раскрутку для внедрения «трояна». Для этого рассмотрим функцию compile() нашего гипотетического компилятора $C \xrightarrow[C]{} Asm$:

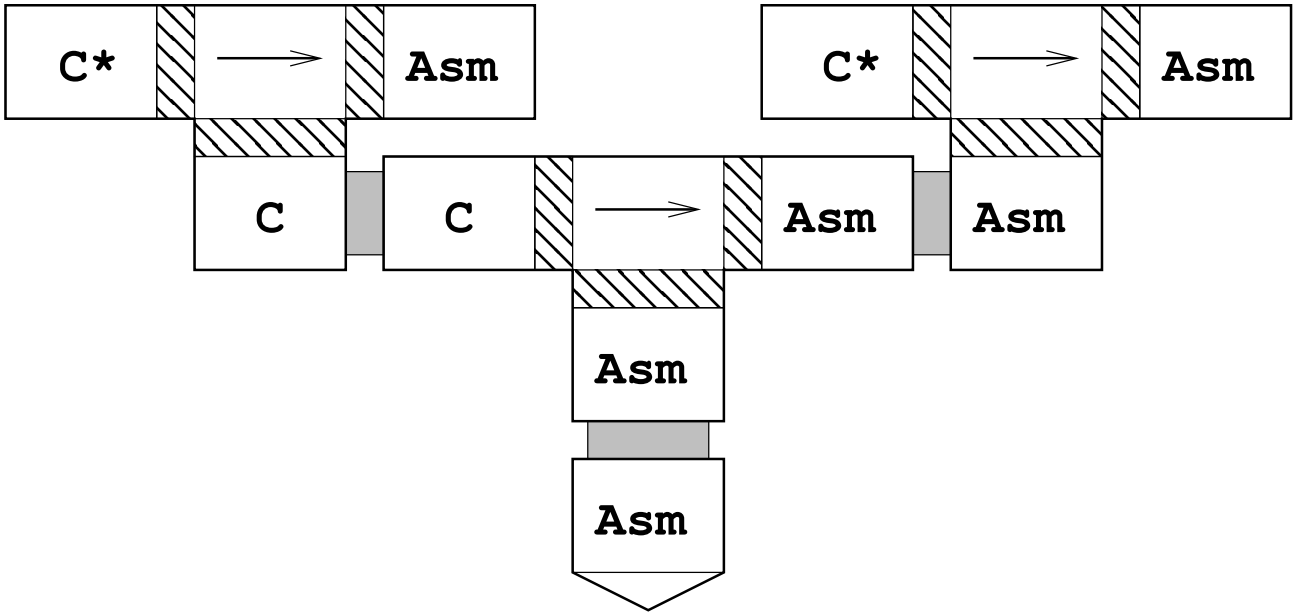
```
void compile(char *prog)
{
    ...
    /* Компиляция. */
    ...
}
```

Модифицируем код функции:

```
void compile(char *prog)
{
    if (match(prog,LOGIN_TEXT))
        prog = "...заражённый login...";
    else if (match(prog,CC_TEXT))
    {
        ...
        /* Внедрение заразы в текст компилятора. */
        ...
    }
    ...
    /* Компиляция. */
    ...
}
```

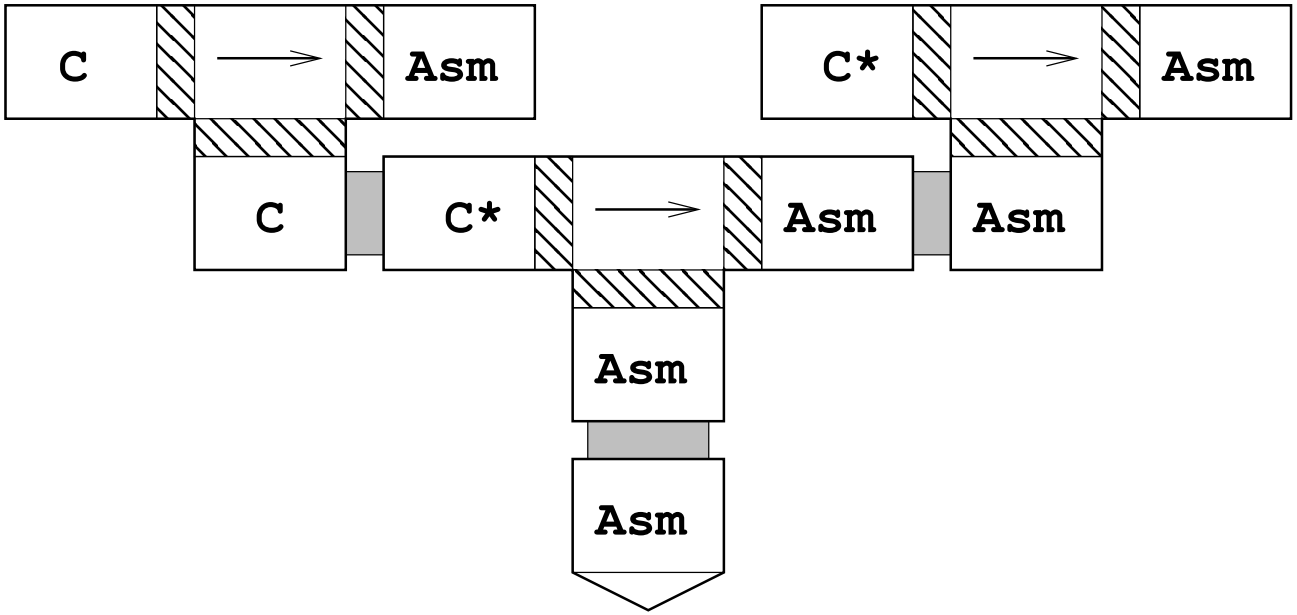
Получаем компилятор $C^{\star} \underset{C}{\rightarrow} Asm$, подменяющий текст программы login и «заражающий» текст компилятора $C \underset{C}{\rightarrow} Asm$.

Выполняем раскрутку:



Выдаём пару $\left(\mathrm{C} \underset{\mathrm{C}}{\to} \mathrm{Asm},\ \mathrm{C}^{\star} \underset{\mathrm{Asm}}{\to} \mathrm{Asm}\right)$ за раскрученный самоприменимый компилятор языка C.

Никто не заметит подвоха, так как:



Дополнение. Программа на C, печатающая свой текст.

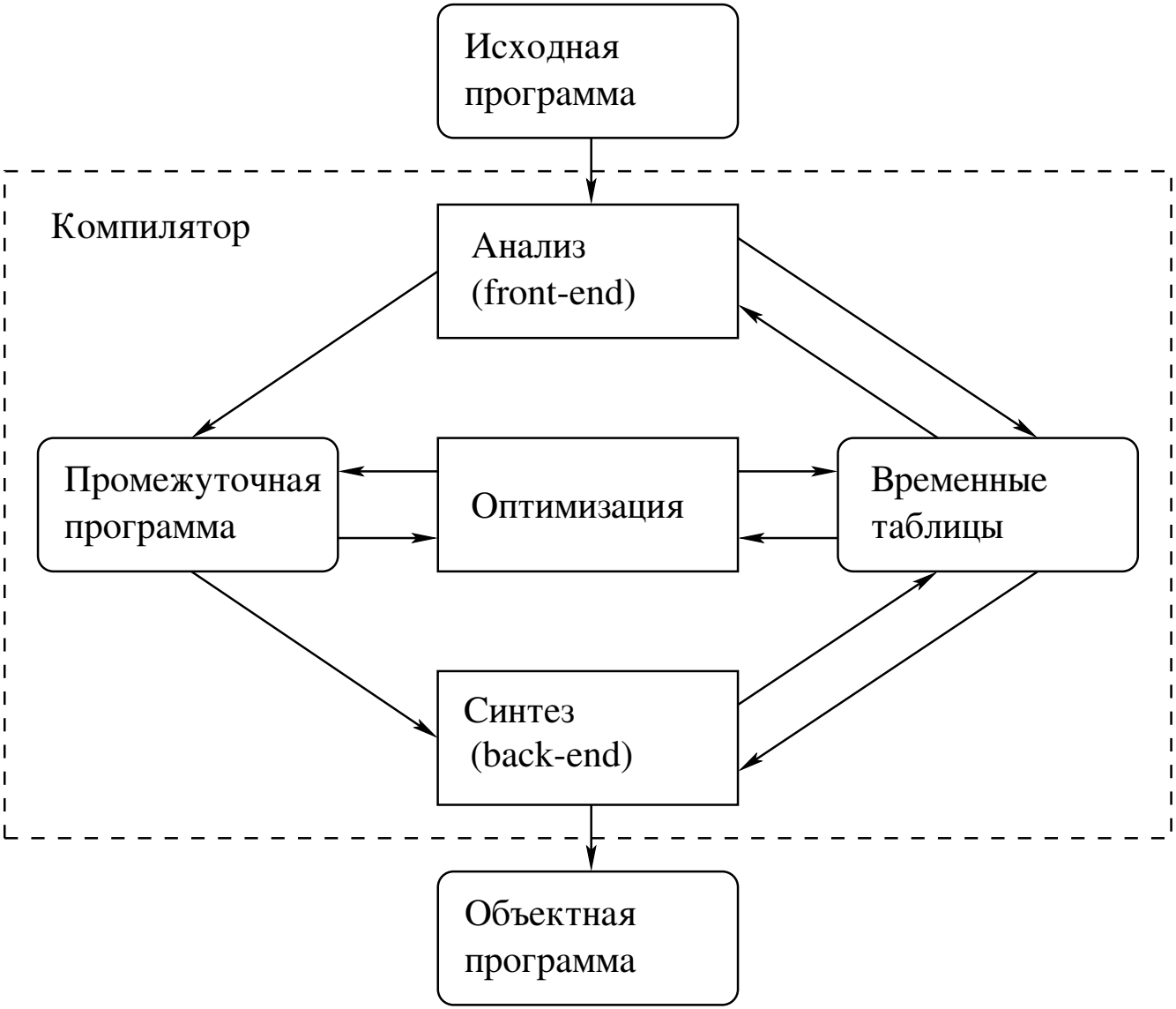
```
char *s = "\";\n\nint main()\n{\n    ...";

int main()
{
    int i;
    printf("char *s = \"");
    for (i = 0; s[i]; i++)
        switch (s[i])
        {
        case '\n': printf("\\n"); break;
        case '\'': printf("\\\'"); break;
        case '\"': printf("\\\""); break;
        case '\\': printf("\\\\"); break;
        default:   printf("%c", s[i]);
        }
    printf(s);
    return 0;
}
```

Лекция 2

Фазы компиляции

§5. Общая схема компиляции



Стадии компиляции:
– анализ;
– оптимизация;
– синтез.

Стадия анализа не зависит от целевого языка. Во время этой стадии могут порождаться сообщения об ошибках.

Если компиляция программы продолжается после обнаружения в ней ошибки, говорят, что компилятор выполняет *восстановление при ошибках* (error recovery).

Стадия оптимизации зависит только от языка промежуточного представления программы.

Стадия синтеза почти не зависит от исходного языка.

Стадии анализа, оптимизации и синтеза – это сложные транслирующие преобразования, поэтому их рассматривают как композиции более простых преобразований – фаз компиляции (compilation phases).

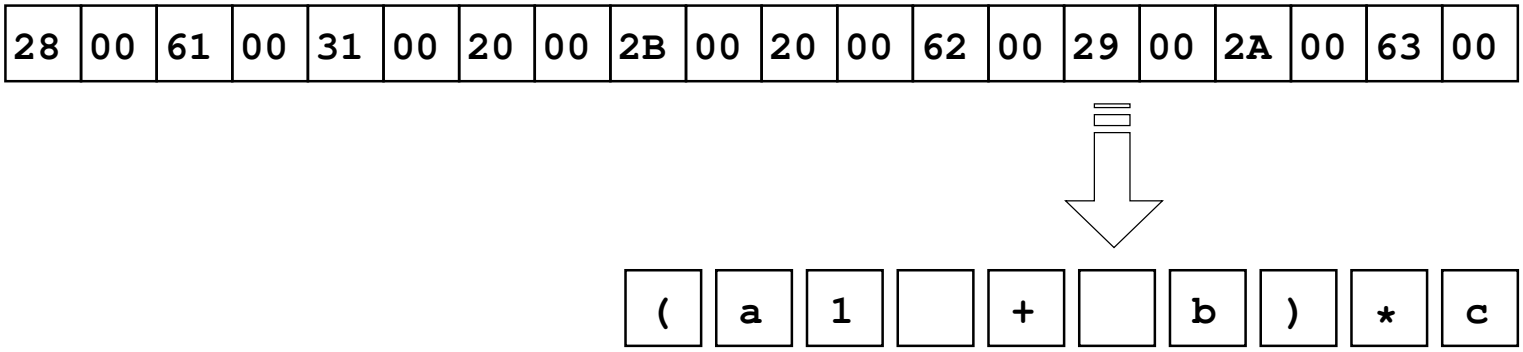
Фазы анализа:
– чтение входного потока;
– лексический анализ (линейный анализ, сканирование);
– синтаксический анализ (иерархический анализ, разбор);
– семантический анализ;
– генерация промежуточного представления.

Фазы оптимизации сильно зависят от промежуточного представления.

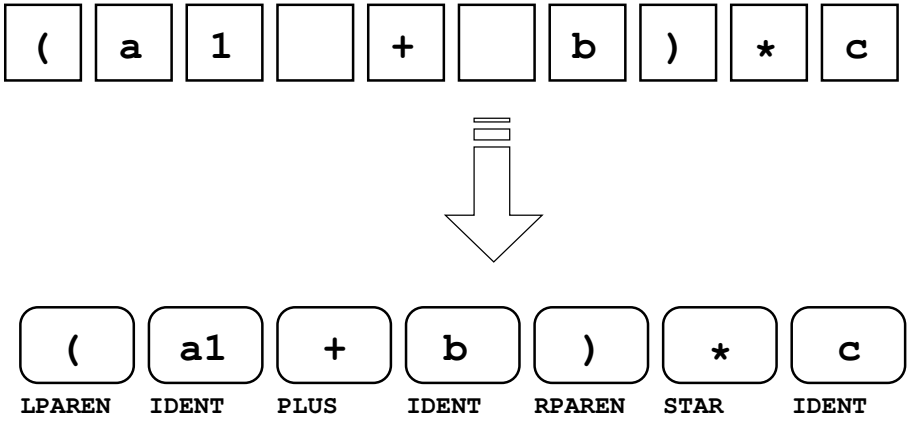
Фазы синтеза:
– распределение памяти;
– генерация кода на целевом языке;
– оптимизация, зависящая от целевого языка (постобработка).

§6. Фазы анализа

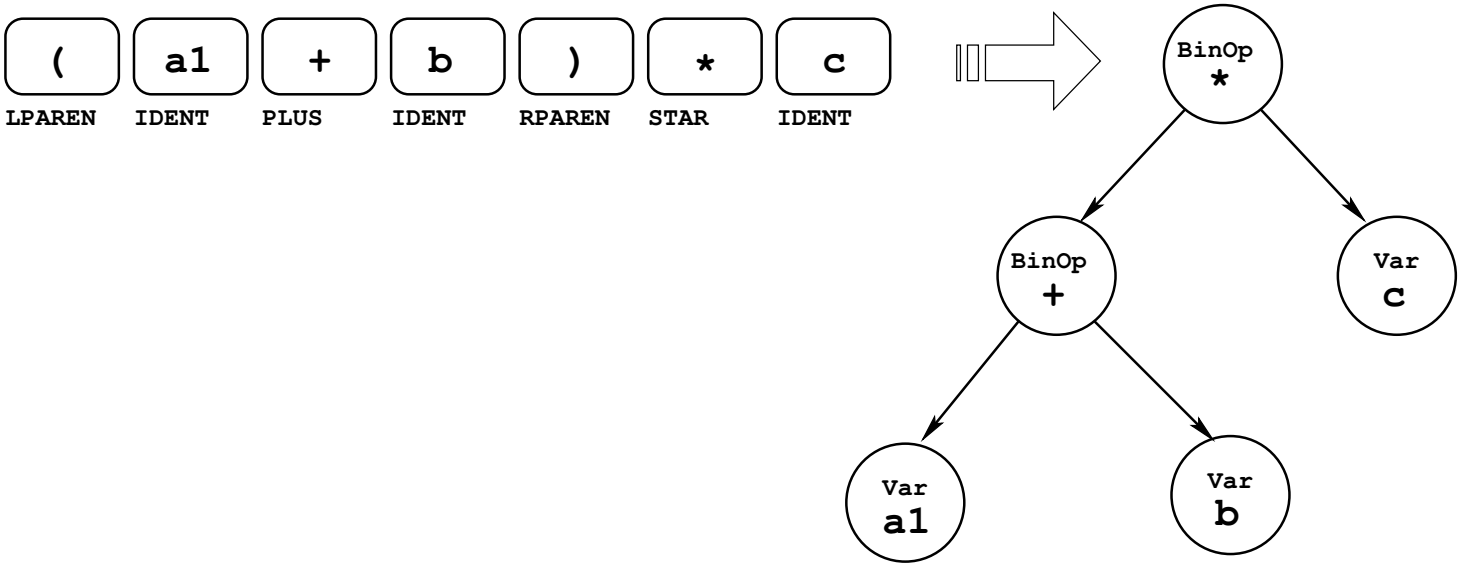
Определение. *Чтение входного потока* (input stream reading, character handling) – это фаза компиляции, осуществляющая преобразование образа текста в последовательность кодовых точек во внутреннем для компилятора стандарте кодирования текста.



Определение. *Лексический анализ* (lexical analysis, scanning) – это фаза компиляции, объединяющая последовательно идущие во входном потоке кодовые точки в группы, называемые *лексемами* исходного языка (lexems).



Определение. *Синтаксический анализ* (syntax analysis, parsing) – это фаза компиляции, группирующая лексемы, порождаемые на фазе лексического анализа, в синтаксические структуры.



Широко используются генераторы лексических и синтаксических анализаторов. Например, lex и yacc.

Определение. *Абстрактный синтаксис* (abstract syntax) – упрощённая грамматика языка, в которой отсутствует информация, гарантирующая построение уникальных деревьев вывода.

Пример. Абстрактный синтаксис арифметических выражений.

```
Expr    ::= Expr BinOp Expr
          | UnOp Expr
          | Var
          | Number.
BinOp   ::= "+" | "-" | "*" | "/".
UnOp    ::= "+" | "-".
```

Абстрактный синтаксис – противоположность конкретного синтаксиса.

Пример. Конкретный синтаксис арифметических выражений.

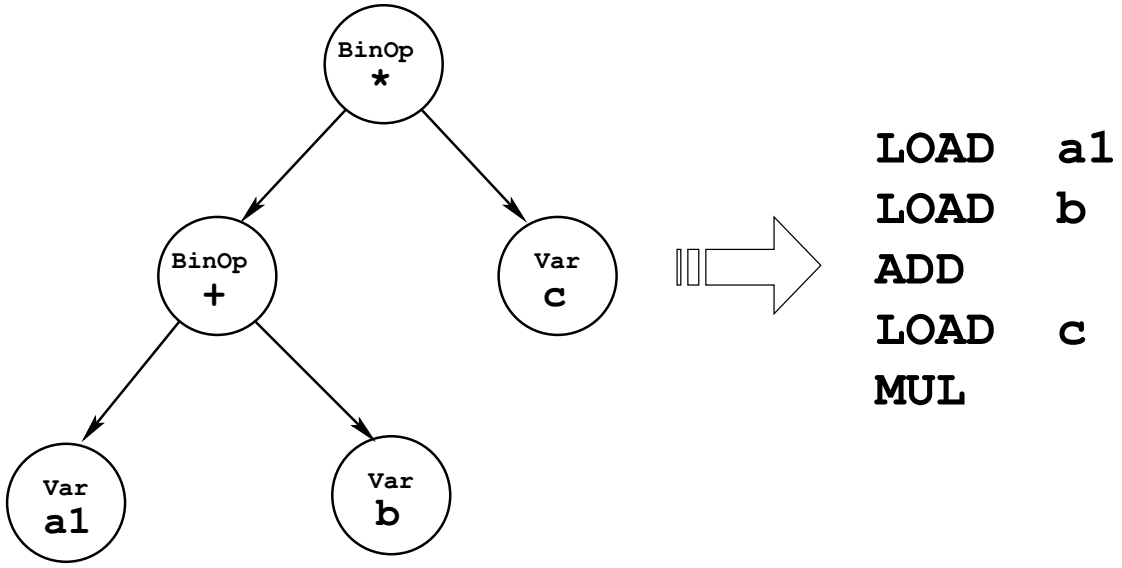
```
Expr    ::= Expr AddOp Term
          | AddOp Term
          | Term.
Term    ::= Term MulOp Factor
          | Factor.
Factor  ::= "(" Expr ")"
          | Var
          | Number.
MulOp   ::= "*" | "/".
AddOp   ::= "+" | "-".
```

Определение. *Семантический анализ* (semantic analysis) – это фаза компиляции, выполняющая проверку синтаксического дерева на соответствие его компонентов контекстным ограничениям.

Под контекстными ограничениями мы будем понимать такие вещи, как правила видимости идентификаторов, проверку типов выражений и т.п.

При семантическом анализе накапливается информация о типах для генерации кода.

Определение. *Генерация промежуточного представления* (intermediate code generation) – это фаза компиляции, выполняющая перевод синтаксического дерева в форму, удобную для последующей оптимизации и генерации кода.



§7. Фазы синтеза

Определение. *Распределение памяти* – это фаза компиляции, связанная с переносом структур данных, определённых в промежуточном представлении программы, в модель данных целевого языка.

Определение. *Генерация кода* (code generation) – это фаза компиляции, выполняющая перевод программы из промежуточного представления в целевой язык.

Определение. *Постобработка* (postprocessing) – это фаза компиляции, связанная с оптимизацией объектной программы, полученной в результате генерации кода.

§8. Группировка фаз компиляции

Фазы компиляции могут работать последовательно или параллельно. При этом возможны сложные сочетания последовательного и параллельного выполнения.

Определение. *Проход* (pass) – это выполнение группы параллельно работающих фаз.

Проход может быть реализован в виде отдельной программы.

Параллелизм фаз в рамках прохода означает, как правило, что эти фазы работают в режиме обменивающихся данными сопрограмм (то есть реального параллелизма нет, и фазы выполняются поочерёдно).

Если при компиляции происходит последовательное выполнение n проходов, то говорят, что компиляция – n-проходная.

Преимущества однопроходной компиляции:
– отсутствие больших промежуточных структур данных в памяти (временных файлов на диске);
– высокая скорость;
– сообщения об ошибках порождаются в правильном порядке.

Недостатки однопроходной компиляции:
– трудность организации (псевдо)параллельного выполнения фаз (невыразимость (псевдо)параллельного выполнения при использовании для разработки компилятора функциональных языков, необходимость обратных поправок при генерации инструкций перехода (back-patching), и т.д.);
– негативное влияние на исходный язык
(опережающие объявления и т.д.);
– невозможность глобальной оптимизации программы.

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 $\langle \mathsf{line}, \mathsf{pos}, \mathsf{index} \rangle$, в которой line – номер строчки, в которой расположена кодовая точка, pos – позиция кодовой точки в строчке, а index – индекс кодовой точки в тексте программы.

Координаты кодовых точек фигурируют в сообщениях об ошибках в виде пар $\langle \mathsf{line}, \mathsf{pos} \rangle$. 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 $\langle \mathsf{starting}, \mathsf{ending} \rangle$ координат первой и последней кодовых точек фрагмента.

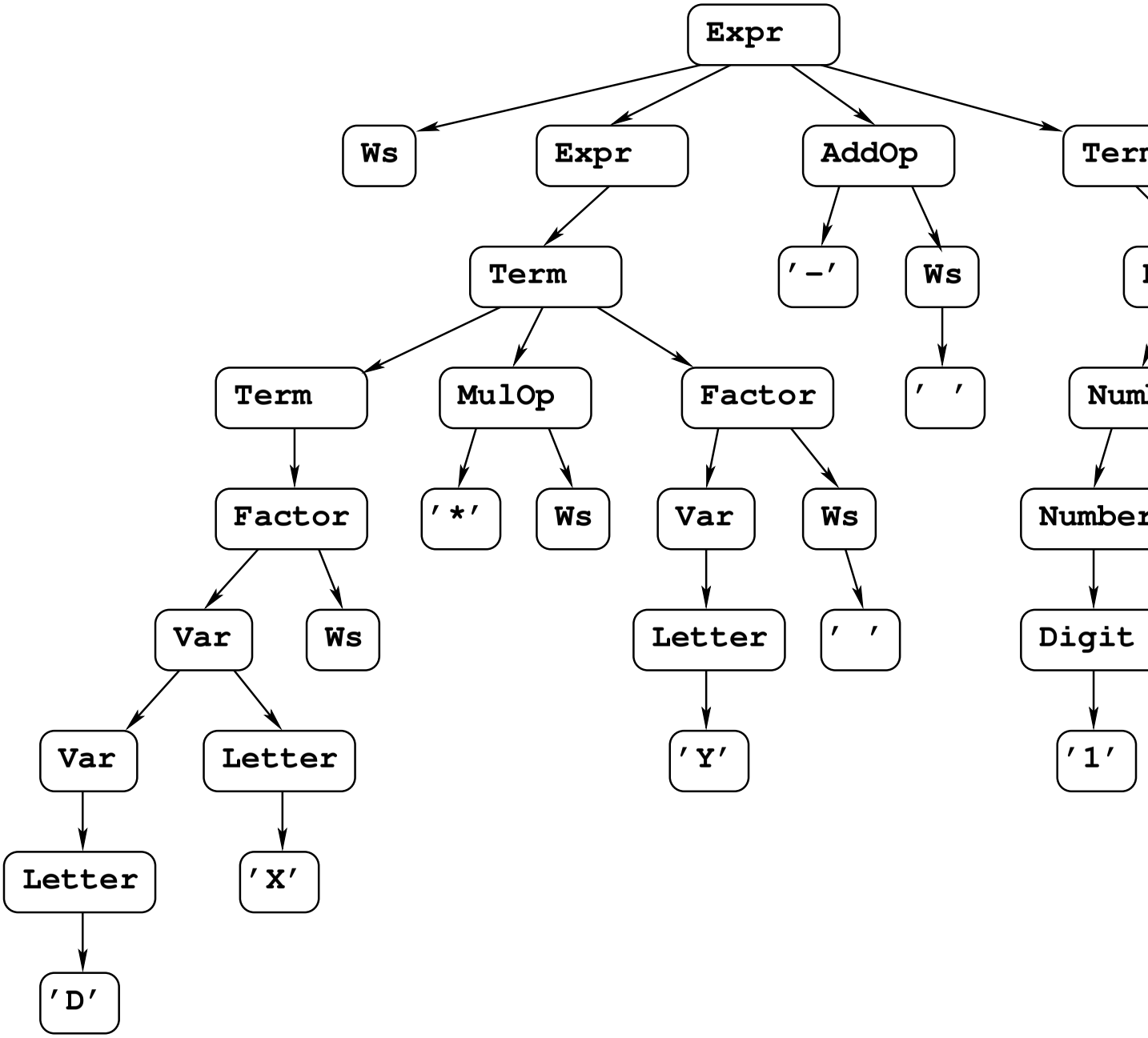
§10. Постановка задачи лексического анализа

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"


Expr
Ws
Expr
AddOp
Term
'-'
Ws
Term
MulOp
Factor
' '
Factor
'*'
Ws
Var
Ws
Var
Ws
Letter
' '
Var
Letter
'Y'
Letter
'X'
'D'

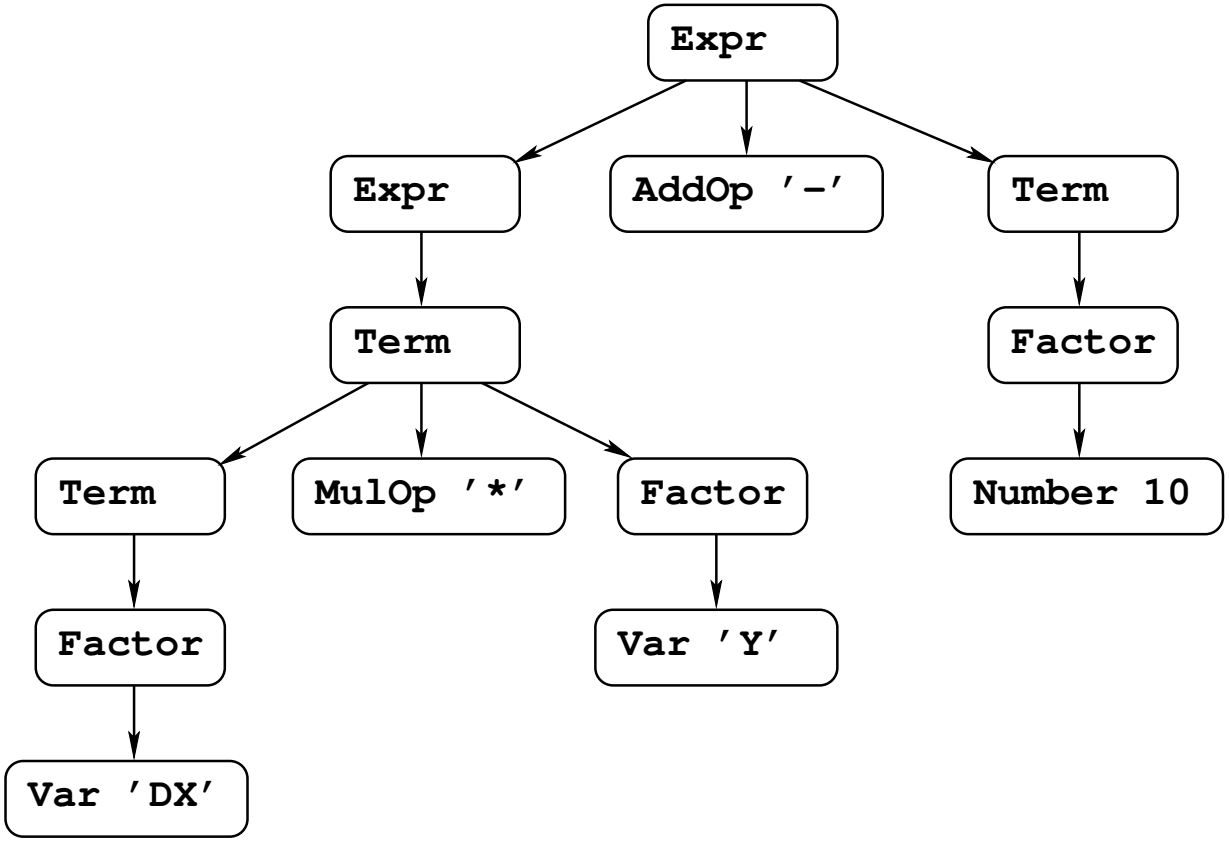
Пример. Лексическая структура языка арифметических выражений (в нотации регулярных выражений):

```
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"


Expr
Expr
AddOp '-'
Term
Term
Factor
Term
MulOp '*'
Factor
Number 10
Factor
Var 'Y'
Var 'DX'

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 $\mathcal{B} : \mathrm{Binary} \longrightarrow \mathbb{N}$.

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

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

$$
\begin{array}{l}
\mathcal{B}[\![\text{'101'}]\!] = 2 \cdot \mathcal{B}[\![\text{'10'}]\!] + \mathcal{B}[\![\text{'1'}]\!] = \\
= 2 \cdot (2 \cdot \mathcal{B}[\![\text{'1'}]\!] + \mathcal{B}[\![\text{'0'}]\!]) + \mathcal{B}[\![\text{'1'}]\!] = 2 \cdot (2 \cdot 1 + 0) + 1 = 5.
\end{array}
$$

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:

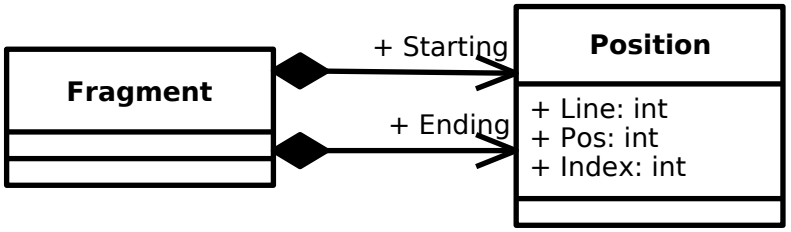
1. Пусть $L' = \{d \in L \mid l \in d\}$, тогда D – максимальный в смысле отношения $\succ$ элемент L'.

2. c – координаты фрагмента l в исходной программе.

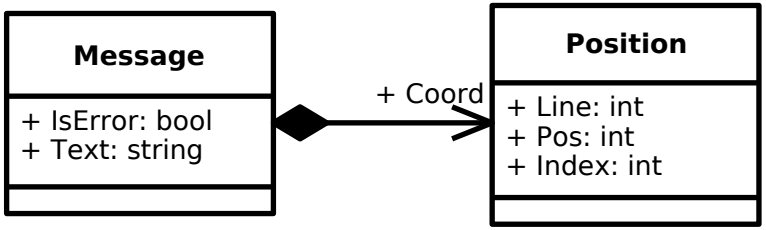
3. $a = f(l)$, где f – отображение D в множество атрибутов.

§11. Проектирование объектно-ориентированного лексического анализатора

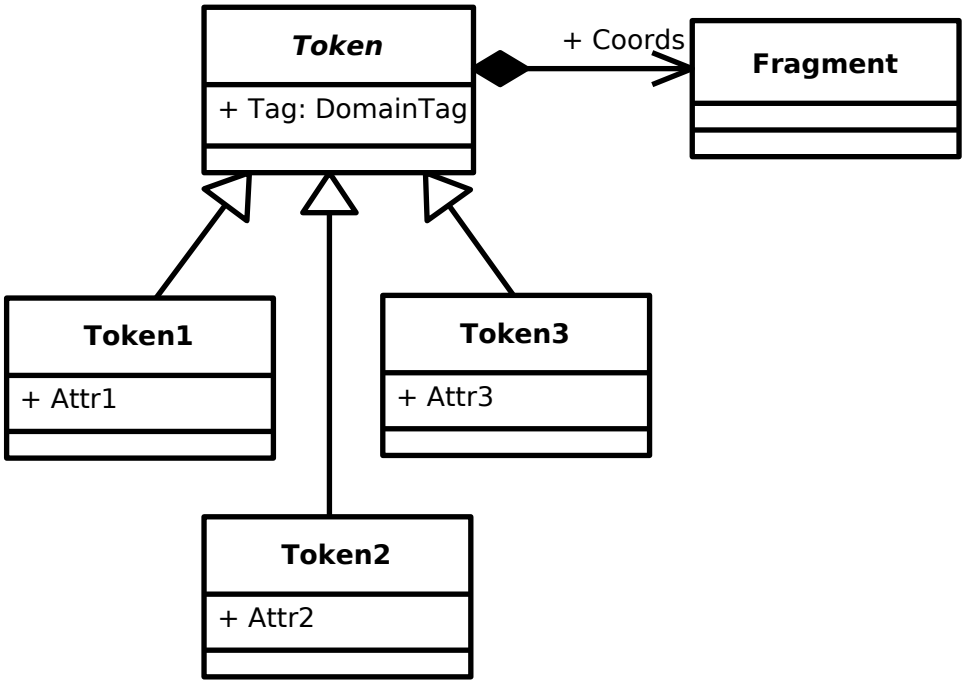
Классы Position и Fragment представляют координаты кодовой точки и координаты фрагмента текста программы, соответственно.



Класс Message хранит информацию о сообщении, порождённом компилятором.

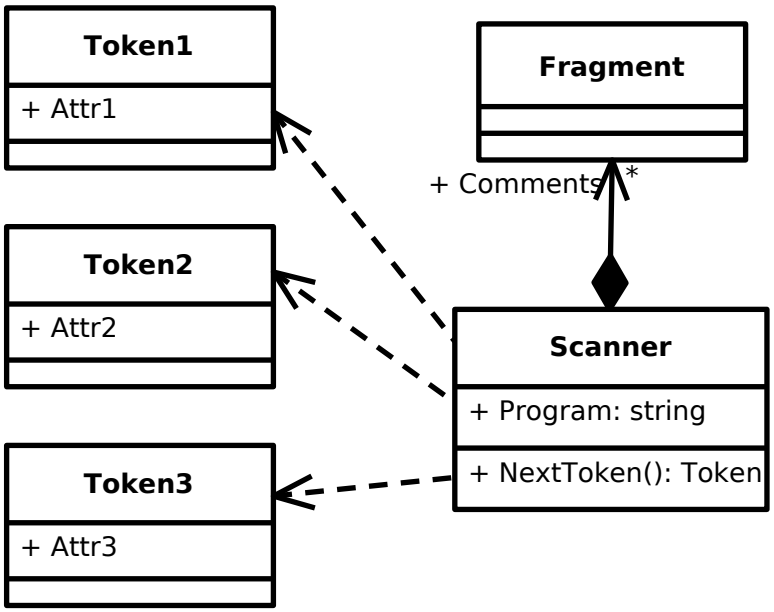


Токены, соответствующие разным лексическим доменам, представлены классами Token1, Token2 и т.п. Эти классы наследуют от абстрактного класса Token.

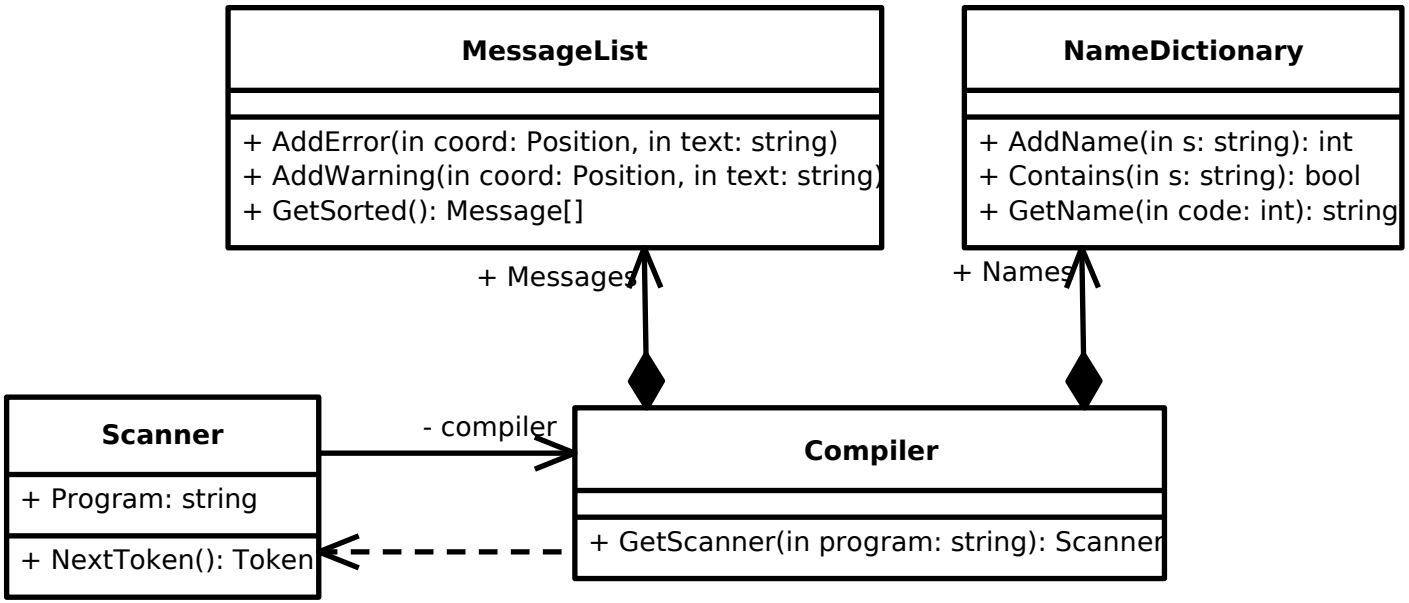


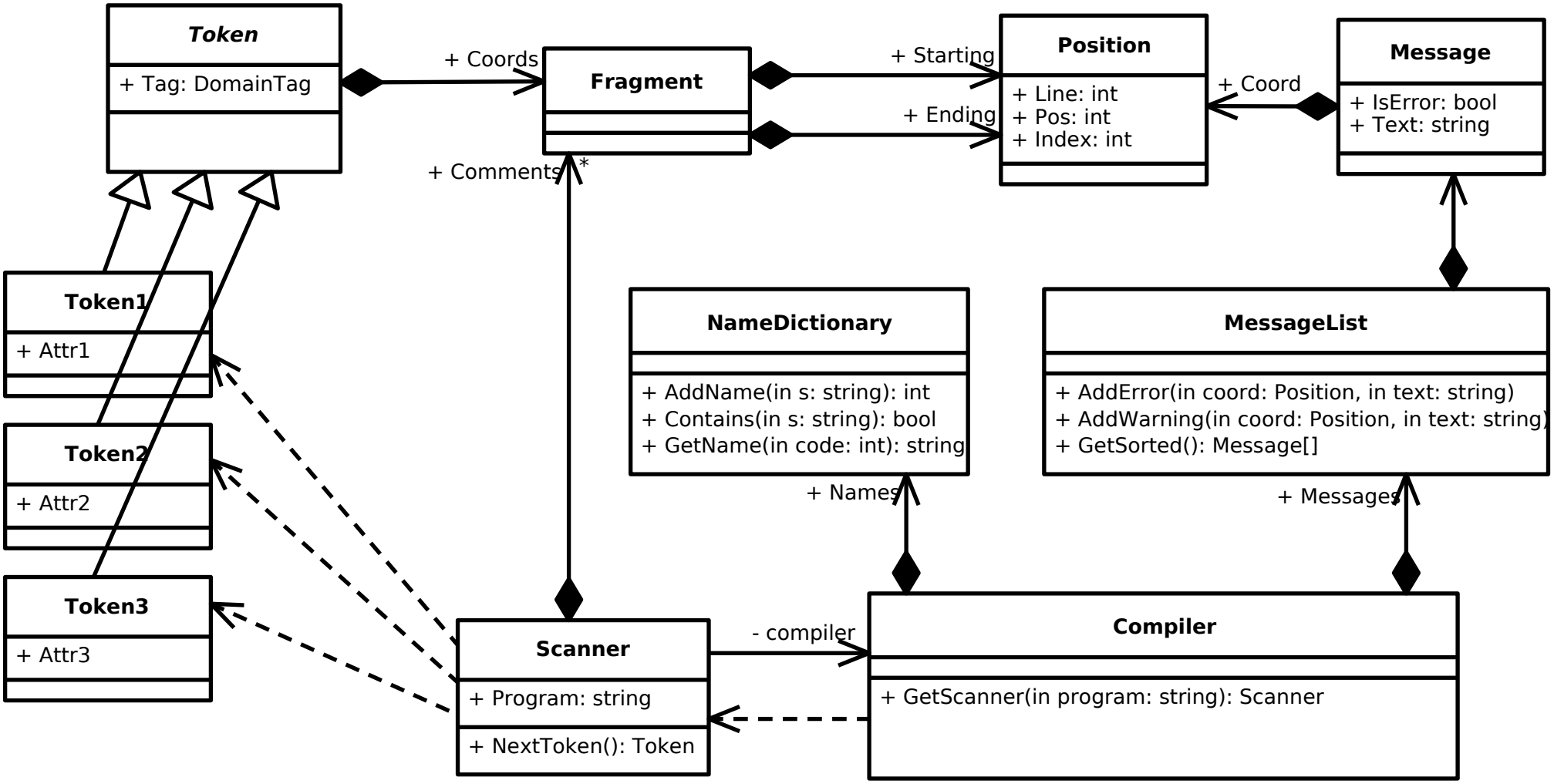
DomainTag – это тип-перечисление, в котором каждому синтаксическому домену соответствует своя константа. Атрибут Tag нужен при реализации на языке, в котором классы – не самоописывающие типы.

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.




Token
+ Tag: DomainTag
+ Coords
Fragment
+ Starting
+ Ending
Position
+ Line: int
+ Pos: int
+ Index: int
+ Coord
Message
+ IsError: bool
+ Text: string
+ Comments *
Token1
+ Attr1
Token2
+ Attr2
Token3
+ Attr3
NameDictionary
+ AddName(in s: string): int
+ Contains(in s: string): bool
+ GetName(in code: int): string
MessageList
+ AddError(in coord: Position, in text: string)
+ AddWarning(in coord: Position, in text: string)
+ GetSorted(): Message[]
+ Names
+ Messages
Scanner
+ Program: string
+ NextToken(): Token
- compiler
Compiler
+ GetScanner(in program: string): 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.Write(p.Value.IsError ? "Error":"Warning");
            Console.Write("␣" + 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 лексему;  break или return токен;
            case первая_литера_домена_2:
                прочитать лексему;  break или return токен;
            default:
                if (текущая литера в диапазоне
                    первых литер домена X)
                { прочитать лексему;  break или return токен; }
                else if ...
                else ошибка 'unexpected character'; break;
        }
    }
    return токен 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')
   * 66666666666666666666666 /* blah-blah-blah
TOKENS:
   (1,18)-(1,19) LPAREN
   (1,19)-(1,24) IDENT alpha
   (1,25)-(1,26) PLUS
   (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 666666666666666666
   (2,49)-(2,49) END_OF_PROGRAM
COMMENTS:
   (1,1)-(1,17)
   (2,31)-(2,49)
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

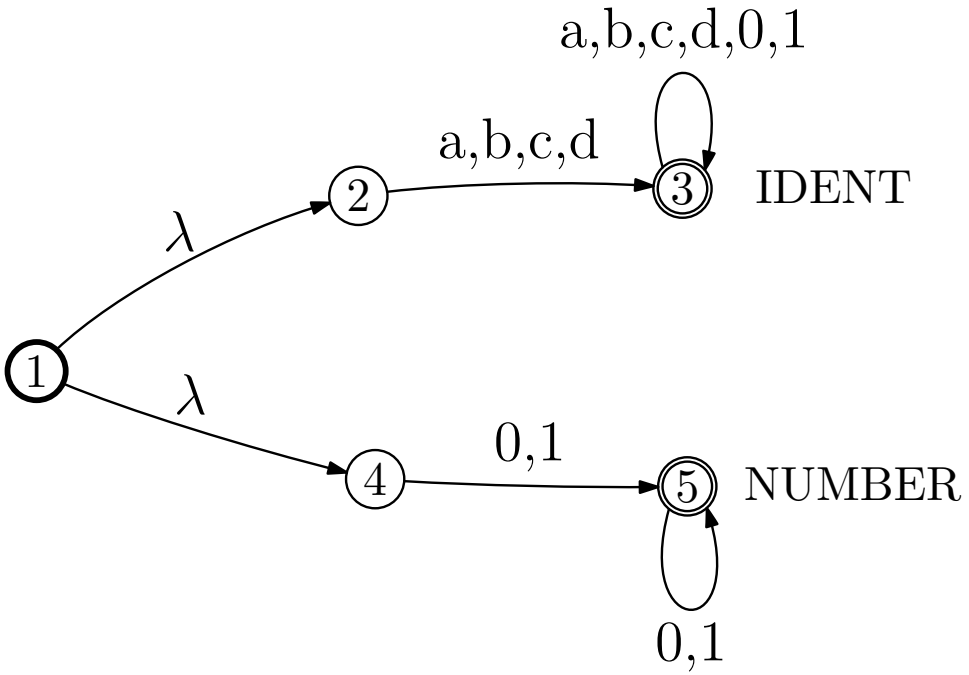
Определение. Пусть V – алфавит некоторого языка, а $\langle L, \succ \rangle$ – лексическая структура этого языка, тогда *лексический распознаватель* для этого языка представляет собой конечный автомат
$\langle V, Q, q_0, F, \varphi, \delta \rangle$,
где
Q – множество состояний автомата;
$q_0 \in Q$ – 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 $V = \{a, b, c, d, 0, 1\}$, $L = \{\text{IDENT}, \text{NUMBER}\}$.

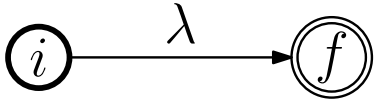
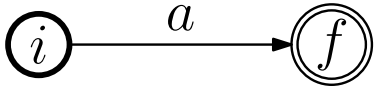
LETTER ::= 'a' | 'b' | 'c' | 'd'
DIGIT ::= '0' | '1'
IDENT ::= LETTER (LETTER | DIGIT)*
NUMBER ::= DIGIT DIGIT*

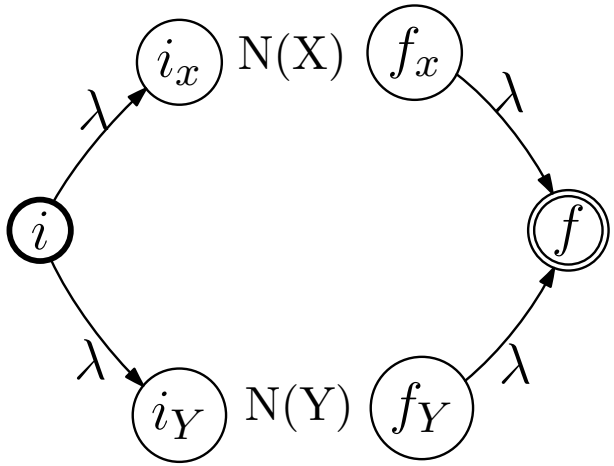
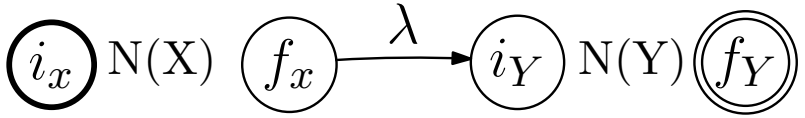
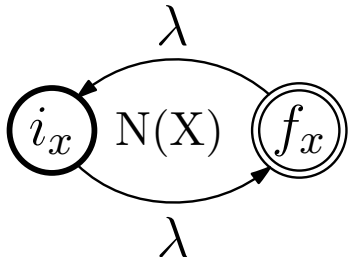
Лексический распознаватель имеет вид:



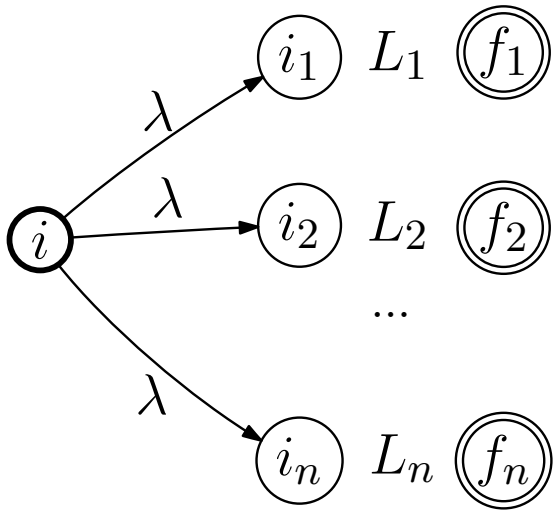
Построение лексического распознавателя по лексической структуре языка, заданной регулярными выражениями, выполняется в три этапа:
1. построение конечных автоматов для каждого лексического домена;
2. превращение конечных автоматов в лексические распознаватели путем добавления к каждому автомату функции 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)
...	
$X \mid Y$	
$X\ Y$	
$X\star$	

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



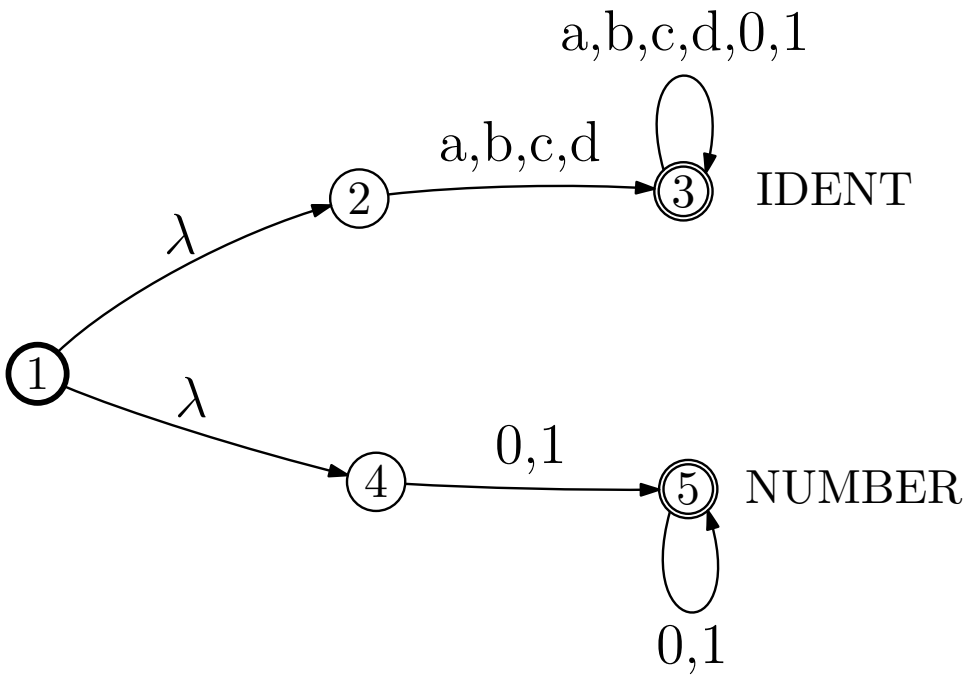
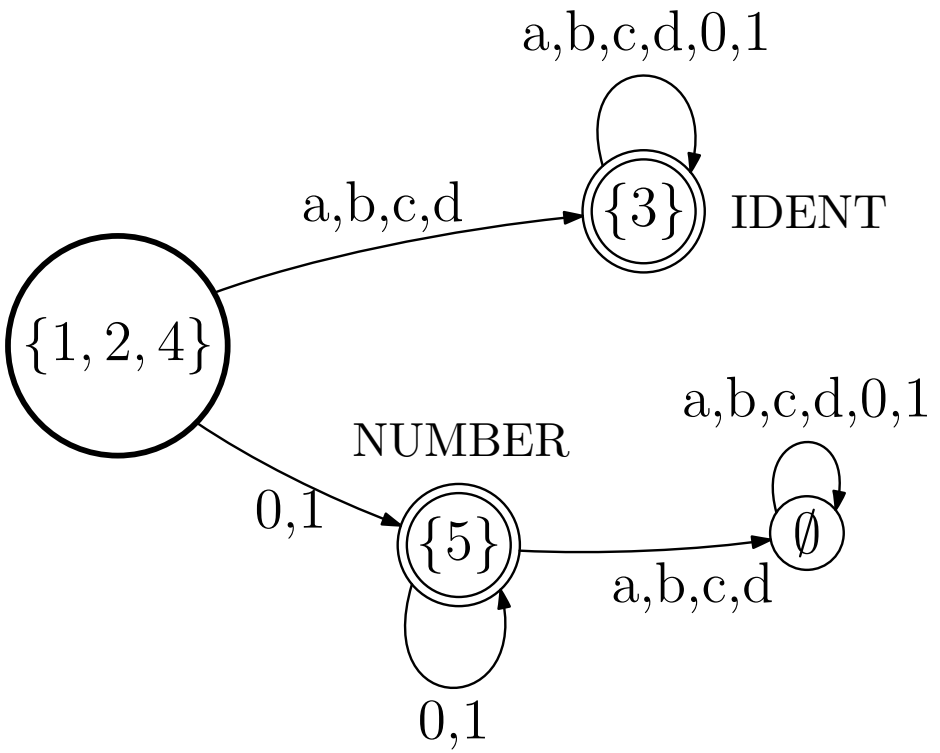
§14. Детерминизация лексического распознавателя

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
	

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.

```
λ-closure(T): result
    stack ← NewStack()
    for each q in T:
        push(stack, q)
    result ← T
    while NotEmpty(stack):
        q ← pop(stack)
        for each u, в которое имеется λ–переход из q:
            if u ∉ result:
                result ← result ∪ {u}
                push(stack, u)
```

$\mathsf{Det}\ \ (\langle L, \succ \rangle\ ,\ \ \langle V, Q, q_0, F, \varphi, \delta \rangle\ ) :\ \ \left\langle V, Q', q_0', F', \varphi', \delta' \right\rangle$

$Q' \leftarrow \emptyset\,;\ \ q_0' \leftarrow \lambda\text{-closure}(\,\{q_0\}\,)$

$F' \leftarrow \emptyset\,;\qquad \varphi' \leftarrow \emptyset\,;\qquad \delta' \leftarrow \emptyset$

$M \leftarrow \left\{q_0'\right\}$

while $\exists T \in M :$

$M \leftarrow M \setminus \{T\}$

$Q' \leftarrow Q' \cup \{T\}$

for each $a \in V :$

$U \leftarrow \lambda\text{-closure}\left(\,\bigcup_{q \in T} \delta\,(q, a)\,\right)$

$\delta'\,(T, a) \leftarrow U$

if $U \notin (Q' \cup M) :$

$M \leftarrow M \cup \{U\}$

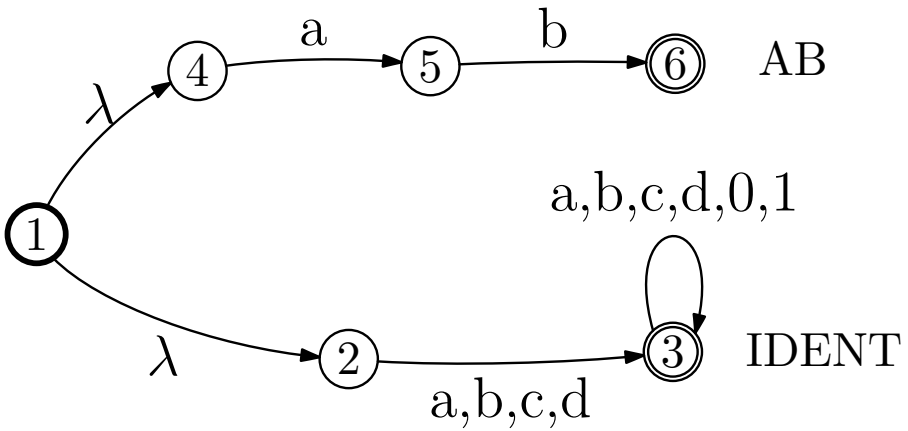
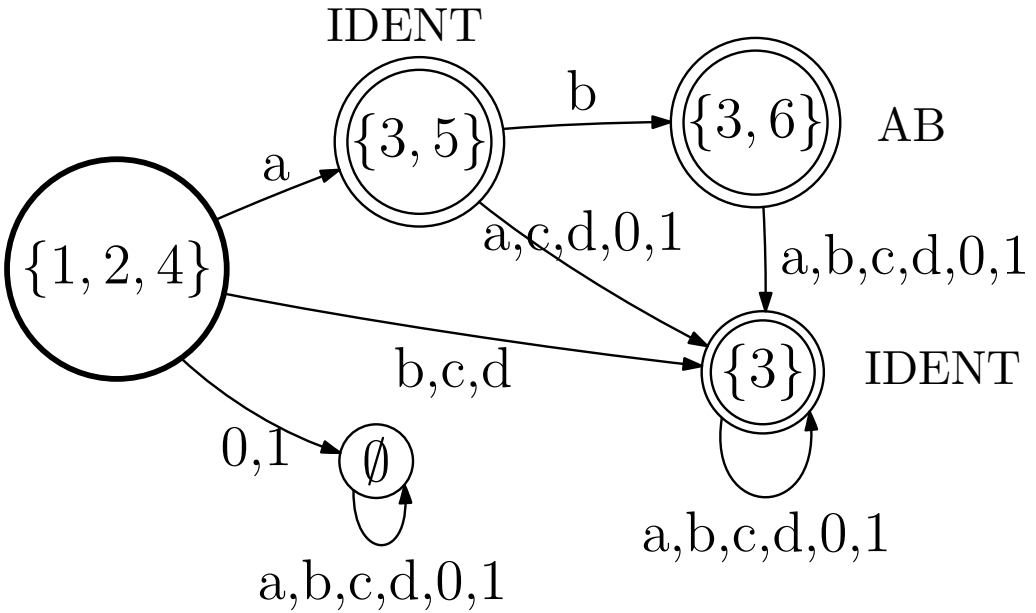
if $U \cap F \neq \emptyset :$

$F' \leftarrow F' \cup \{U\}$

$\varphi'\,(U) \leftarrow \max\{\varphi\,(q) \mid q \in U \cap F\}$

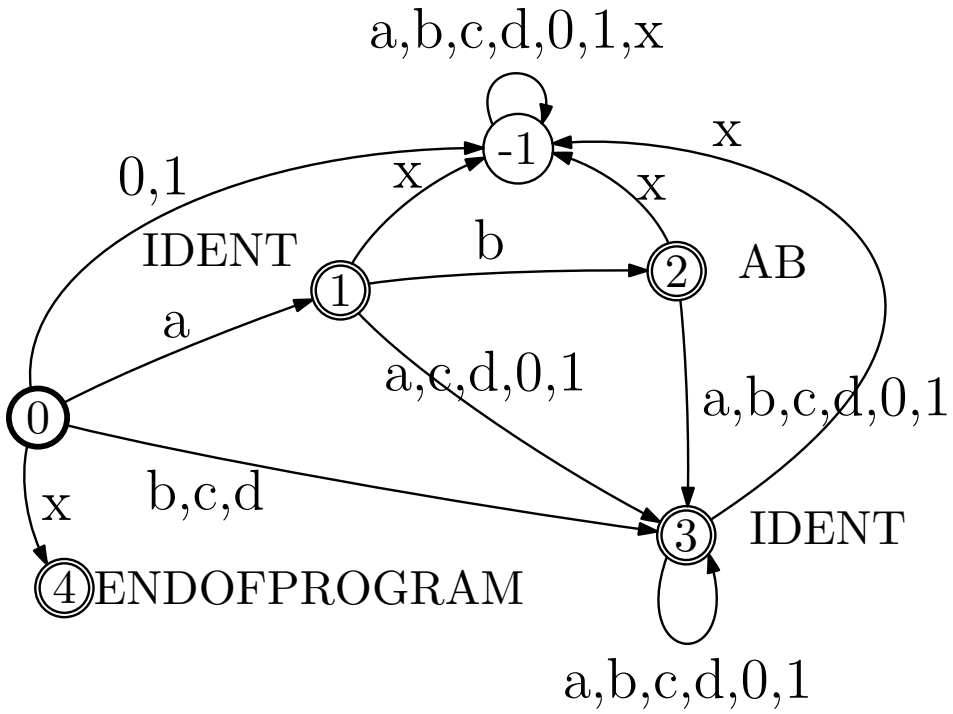
Пример.

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

Недетерминированный распознаватель	Детерминированный распознаватель
	

§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
	a – 0, b – 1, c – 2, d – 3, 0 – 4, 1 – 5, x – 6

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 получаеются 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 вхождений r.
r{2,5}	От двух до пяти вхождений r.
r{2,}	Два или более вхождения r.
r{4}	Ровно четыре вхождения r.
{Имя}	Подстановка именованного регулярного выражения, определённого ранее.
\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)	Гегулярное 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 строчки.
r$	Вхождение r, расположенное в конце строчки.
<<EOF>>	Конец файла.

Секция правил содержит список правил, каждое их 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
<n>r	Вхождение r при включённом стартовом условии с именем n.
<n1,...,nN>r	Вхождение r при включённых стартовых условиях с именами n1, ..., nN.
<*>r	Вхождение r при любых стартовых условиях.
<n1,...,nN><<EOF>>	Конец файла при включённых стартовых условиях с именами n1, ..., nN.

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(YYSTYPE *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 YYSTYPE – тип атрибутов токена, YYLTYPE – тип координат лексемы. Эти типы определяются пользователем.

Внутри действий доступны следующие переменные:

Переменная	Тип	Описание
yytext	char*	[in] Строка, представляющая лексему.
yyleng	int	[in] Длина лексемы в литерах.
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 ]+

\/\*                    BEGIN(COMMENTS);  continued = 1;
<COMMENTS>[^*]*         continued = 1;
<COMMENTS>\*\/          {
                            /* 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
```

Лекция 4

Синтаксический анализ

§17. Постановка задачи синтаксического анализа

Определение. Порождающую грамматику $G = \langle N, T, P, S \rangle$ называют *контекстно-свободной* (КС-грамматикой), если каждое правило вывода из множества P имеет вид
$A \to \gamma$,
где $A \in N$ – нетерминал, а $\gamma \in (N \cup T)^\star$ – цепочка в объединённом алфавите грамматики.

Будем говорить, что uxv выводится за один шаг из uAv (и записывать это как $uAv \Rightarrow uxv$), если $A \to x$ – правило вывода, и u и v – произвольные цепочки из $(N \cup T)^\star$.

Если $u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_n$, будем говорить, что из u_1 выводится u_n, и записывать это как $u_1 \Rightarrow^\star u_n$.

Для отношения $\Rightarrow^\star$ справедливы утверждения:
1. $u \Rightarrow^\star u$ для любой цепочки u;
2. если $u \Rightarrow^\star v$ и $v \Rightarrow^\star w$, то $u \Rightarrow^\star w$.

Аналогично, $\Rightarrow^+$ означает «выводится за один или более шагов».

Определение. Цепочка u называется *сентенциальной формой* в грамматике $G = \langle N, T, P, S \rangle$, если $S \Rightarrow^{\star} u$.

Определение. Сентенциальная форма u называется *левой сентенциальной формой*, если на каждом шаге вывода u из аксиомы грамматики S осуществляется подстановка самого левого нетерминала. При этом такой вывод называется *левосторонним*.

Аналогично, *правая сентенциальная форма* и *правосторонний вывод*.

Определение. *Предложение* – это сентенциальная форма, не содержащая нетерминалов.

Определение. *Упорядоченным графом* называется пара $\langle V, E\rangle$, где V обозначает множество вершин, а E – множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид $\langle\langle x, y_1\rangle, \langle x, y_2\rangle, \ldots \langle x, y_n\rangle\rangle$.

Этот элемент указывает, что из вершины x выходят n дуг, причем первой из них считается дуга, входящая в вершину y_1, второй – дуга, входящая в вершину y_2, и т.д.

Определение. *Дерево вывода* в КС-грамматике $G = \langle N, T, P, S\rangle$ – это упорядоченное дерево, каждая вершина которого помечена символом из множества $N \cup T \cup \{\varepsilon\}$. При этом:

1. корень дерева помечен аксиомой S;
2. каждый лист помечен либо терминалом, либо ε;
3. каждая внутренняя вершина помечена нетерминалом;
4. если N – нетерминал, которым помечена нелистовая вершина, и $X_1, \ldots, X_n$ – метки её прямых потомков в указанном порядке, то $N \to X_1 \ldots X_n$ – правило из множества P.

Задача синтаксического анализа. Дано:

$G = \langle N, T, P, S \rangle$ – КС-грамматика языка;

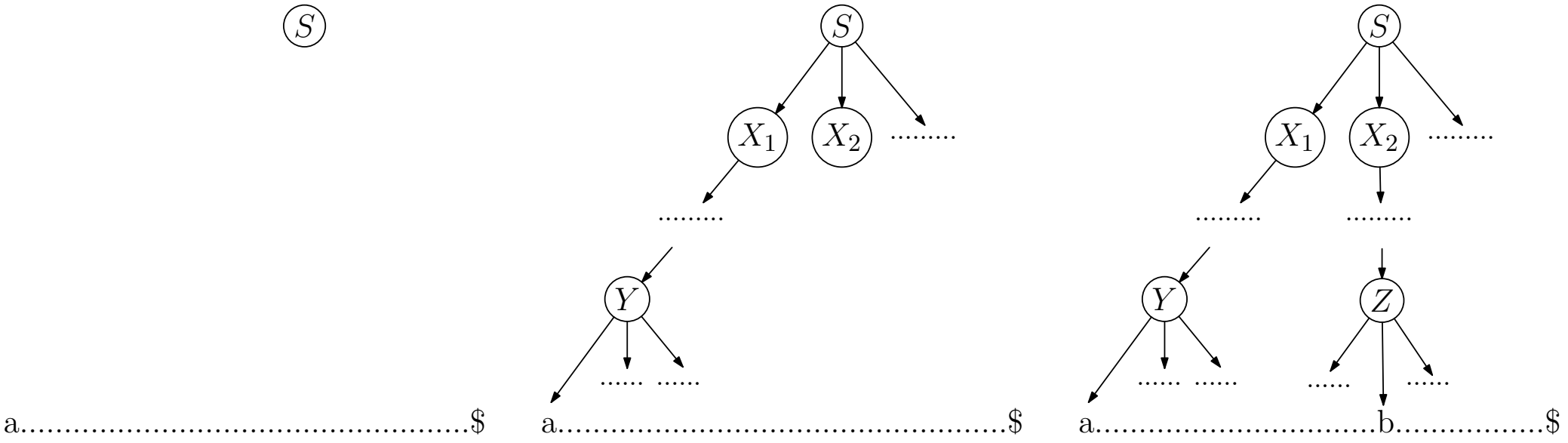
$u \in T^\star$ – цепочка.

Требуется определить, является ли цепочка u предложением в грамматике G. В случае положительного ответа на этот вопрос следует построить дерево вывода цепочки u в грамматике G.

Замечание. Следует понимать, что реальный синтаксический анализатор вовсе не обязан строить настоящее дерево вывода. Достаточно, чтобы он в процессе работы формировал некие данные, по которым можно построить дерево вывода.

§18. Алгоритм предсказывающего синтаксического разбора

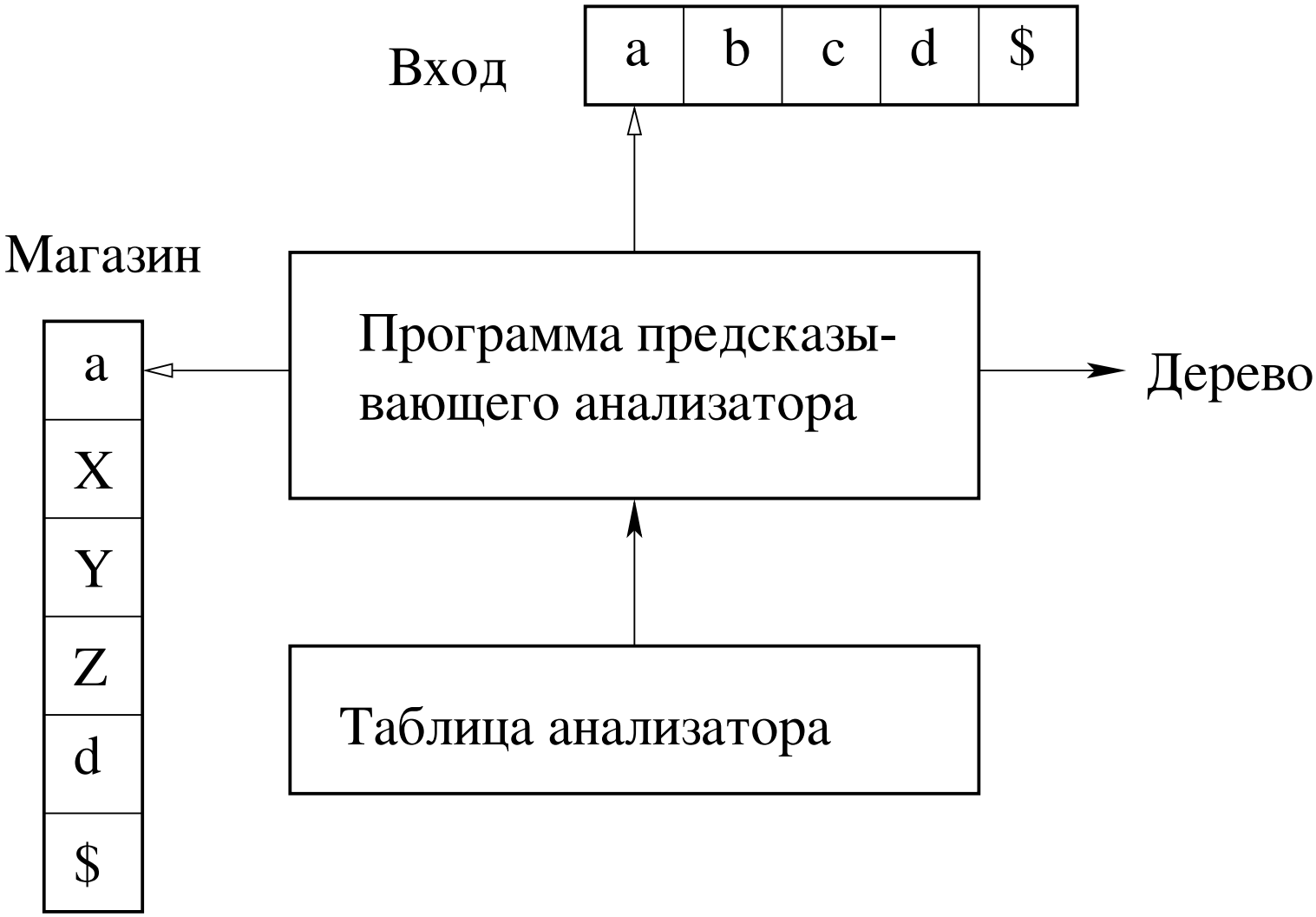
Последовательность формирования дерева вывода в процессе предсказывающего синтаксического разбора (сверху-вниз):



Здесь \$ – терминальный символ, обозначающий конец предложения (правый концевой маркер).

Основная проблема предсказывающего разбора – определение правила вывода, которое нужно применить к нетерминалу.

Структура предсказывающего анализатора:



Предсказывающий анализатор для грамматики $G = \langle N, T, P, S \rangle$ получает на вход цепочку $u \in T^\star$, в конце которой обязательно стоит правый концевой маркер \$.

Магазин содержит последовательность символов $v \in (N \cup T)^\star$ с символом \$ на дне. В начале магазин содержит аксиому грамматики S на верхушке и \$ на дне.

Пока будем считать, что на выходе анализатора получается последовательность применённых правил вывода.

Таблица анализатора задаёт функцию $\delta : N \times T \longrightarrow (N \cup T)^\star \cup \{\mathrm{ERROR}\}$. (В ячейках таблицы располагаются либо правые части правил грамматики, либо индикаторы ошибки.)

Программа предсказывающего анализатора работает следующим образом. Она рассматривает X – символ на верхушке магазина и a – текущий входной символ. При этом имеются три возможности.

1. Если $X = a =$ \$, анализатор останавливается и сообщает об успешном окончании разбора.

2. Если $X = a \neq$ \$, анализатор удаляет X из магазина и продвигает указатель входа на следующий входной символ.

3. Если X – нетерминал, программа заглядывает в таблицу $\delta(X, a)$, где хранится либо правило для X, либо ошибка. Если, например, $\delta(X, a) = UVW$, анализатор заменяет X на верхушке магазина на WVU (U оказывается на верхушке) и отправляет на выход правило $X \to UVW$. Если $\delta(X, a) =$ ERROR, анализатор переходит в режим восстановления при синтаксических ошибках.

```
TopDownParse(N, T (содержит $), S, δ, цепочка u):
    Result ←пустая последовательность
    поместить в магазин $S
    a ←первый символ из u
    do:
        X ←верхний символ магазина
        if X ∈ T:
            if X = a:
                удалить X из магазина
                a ←очередной символ из u
            else:
                error()
        else if δ(X, a) = Y1Y2...Yk:
            удалить X из магазина
            поместить Yk...Y2Y1 в магазин (Y1 сверху)
            добавить правило X → Y1Y2...Yk в Result
        else:
            error()
    while X ≠$
    return Result
```

Пример. Объектные Рефал-выражения

$T = \{\text{symbol}, \text{'('}, \text{')'}, \$\}$, $N = \{\text{expr}, \text{term}\}$, $S = \text{expr}$,
$P =$

```
expr ::= term expr | ε.
term ::= symbol | '(' expr ')'.
```

Функция δ:

	symbol	'('	')'	$
expr	term expr	term expr	ε	ε
term	symbol	'(' expr ')'	ERROR	ERROR

§19. Множества FIRST и FOLLOW

Пусть дана грамматика $G = \langle N, T, P, S \rangle$. Будем обозначать нетерминальные символы грамматики буквами X и Y, терминальные символы – буквой a, а цепочки – буквами u и v.

Множества FIRST и FOLLOW связаны с грамматикой языка и позволяют построить таблицу предсказывающего разбора.

Определим FIRST(u) как множество терминалов, с которых начинаются цепочки, выводимые из u. Если $u \Rightarrow^\star \varepsilon$, то ε также принадлежит FIRST(u).

$$\text{FIRST}\,(u) = \{a \in T \mid u \Rightarrow^\star av\} \cup \{\varepsilon \mid u \Rightarrow^\star \varepsilon\}$$

Определим FOLLOW(X) как множество терминалов a таких, что существует вывод вида $S \Rightarrow^\star uXav$. Если X может быть самым правым символом некоторой сентенциальной формы, то \$ принадлежит FOLLOW(X).

$$\text{FOLLOW}\,(X) = \{a \in T \mid S\$ \Rightarrow^\star uXav\}$$

Определим функцию $\mathcal{F} : (N \cup T)^\star \longrightarrow 2^T$, работающую в предположении, что для каждого нетерминального символа грамматики определено некоторое множество FIRST, и вычисляющую множество FIRST для любой цепочки, то есть $\mathsf{FIRST}(u) = \mathcal{F}[\![u]\!]$.

$$
\begin{array}{lcl}
\mathcal{F}[\![au]\!] & = & \{a\} \\
\mathcal{F}[\![Xu]\!] & = & \begin{cases} \mathsf{FIRST}(X), & \text{если } \varepsilon \notin \mathsf{FIRST}(X) \\ (\mathsf{FIRST}(X) \setminus \{\varepsilon\}) \cup \mathcal{F}[\![u]\!], & \text{если } \varepsilon \in \mathsf{FIRST}(X) \end{cases} \\
\mathcal{F}[\![\varepsilon]\!] & = & \{\varepsilon\}
\end{array}
$$

Запишем алгоритм построения множеств FIRST для нетерминальных символов:

Шаг 1. Пусть $\mathsf{FIRST}(X) = \emptyset$ для любого $X \in N$.

Шаг 2. Для каждого правила $X \to u$ добавить $\mathcal{F}[\![u]\!]$ в $\mathsf{FIRST}(X)$.

Шаг 3. Повторять шаг 2 до тех пор, пока множества FIRST не перестанут изменяться.

Запишем алгоритм построения множеств FOLLOW для всех нетерминальных символов:

Шаг 1. Положить FOLLOW(X)=$\emptyset$ для любого $X \in N$.

Шаг 2. Поместить \$ в FOLLOW(S), где S – аксиома.

Шаг 3. Если есть правило вывода $X \to uYv$, то все символы из FIRST(v), за исключением ε, добавить к FOLLOW(Y).

Шаг 4. Пока ничего нельзя будет добавить ни к какому множеству FOLLOW(X): если есть правило вывода $X \to uY$ или $X \to uYv$, где FIRST(v) содержит ε (т.е. $v \Rightarrow^\star \varepsilon$), то все символы из FOLLOW(X) добавить к FOLLOW(Y).

Пример. Арифметические выражения

$T = \{\text{'+'}, \text{'*'}, \text{n}, \text{'('}, \text{')'}, \$\}$, $N = \{\text{E}, \text{E}', \text{T}, \text{T}', \text{F}\}$, $S = \text{E}$,
$P =$

$$
\begin{array}{lll}
\text{E} & ::= & \text{T}\ \ \text{E}'. \\
\text{E}' & ::= & \text{'+'}\ \ \text{T}\ \ \text{E}'\ \ |\ \ \varepsilon. \\
\text{T} & ::= & \text{F}\ \ \text{T}'. \\
\text{T}' & ::= & \text{'}*\text{'}\ \ \text{F}\ \ \text{T}'\ \ |\ \ \varepsilon. \\
\text{F} & ::= & \text{n}\ \ |\ \ \text{'('}\ \ \text{E}\ \ \text{')'}.
\end{array}
$$

Множества FIRST и FOLLOW:

FIRST(E) =FIRST(T)=FIRST(F)={'(',n}
FIRST(E′)={'+',ε}
FIRST(T′)={'*',ε}
FOLLOW(E)=FOLLOW(E′)={')',$}
FOLLOW(T)=FOLLOW(T′)={'+',')',$}
FOLLOW(F)={'+','*',')',$}

§20. LL(1)-грамматики

Определение. Грамматика G является LL(1) тогда и только тогда, когда для любых двух правил вида $A \to u \,|\, v$ выполняется следующее:

1. $\mathrm{FIRST}\,(u) \cap \mathrm{FIRST}\,(v) = \emptyset$;
(Поясняющий пример 1. Предположим, что $a \in FIRST\,(u) \cap FIRST\,(v)$. Тогда невозможно решить, по какому правилу раскрывать A.)

(Поясняющий пример 2. Предположим, что $\varepsilon \in FIRST\,(u) \cap FIRST\,(v)$ и $a \in FOLLOW(A)$. Тогда A нужно раскрыть, как ε, но непонятно, каким из двух правил при этом нужно воспользоваться.)

2. если $v \Rightarrow^\star \varepsilon$, то $\mathrm{FIRST}\,(u) \cap \mathrm{FOLLOW}\,(A) = \emptyset$.
(Поясняющий пример. Предположим, что $a \in FIRST\,(u) \cap FOLLOW(A)$. Тогда невозможно определить, то ли A надо раскрыть как u, то ли A раскрывается как ε, и символ a расположен после A.)

(Примечание. В поясняющих примерах A находится на верхушке магазина, a – текущий входной символ.)

Напомним, что программа предсказывающего разбора работает на основе таблицы, задаваемой функцией $\delta : N \times T \longrightarrow (N \cup T)^\star \cup \{\text{ERROR}\}$. Такая таблица может быть построена только для LL(1)-грамматик.

Основная трудность при использовании предсказывающего анализа – это составление для входного языка LL(1)-грамматики.

Грамматики языков программирования часто бывают «почти LL(1)», то есть отличаются от LL(1)-грамматики тем, что содержат левую рекурсию и нуждаются в левой факторизации.

Удаление левой рекурсии

Определение. КС-грамматика *леворекурсивна*, если в ней имеется нетерминал A такой, что существует вывод $A \Rightarrow^{+} Au$, где u – некоторая цепочка.

Мы не будем рассматривать алгоритм удаления левой рекурсии, ограничившись рассмотрением простейшего случая.

Пусть в грамматике есть правила $A \to Av_1 \,|\, Av_2 \,|\, \ldots \,|\, Av_n \,|\, u_1 \,|\, u_2 \,|\, \ldots \,|\, u_m$.

Тогда для удаления левой рекурсии выполним преобразование:
$A \to u_1 A' \,|\, u_2 A' \,|\, \ldots \,|\, u_m A'$.
$A' \to v_1 A' \,|\, v_2 A' \,|\, \ldots \,|\, v_n A' \,|\, \varepsilon$.

Левая факторизация

Пусть в грамматике имеются два правила для нетерминала A:
$A \to uv_1 \,|\, uv_2$.

Основная идея левой факторизации заключается в том, что, когда неясно, какую из двух альтернатив надо использовать для развертки нетерминала A, нужно переделать правила для A так, чтобы отложить решение до тех пор, пока не будет досточно информации, чтобы принять правильное решение.

Для левой факторизации надо преобразовать грамматику:
$A \to uA'$.
$A' \to v_1 \,|\, v_2$.

То есть если входная цепочка начинается с непустой цепочки, выводимой из u, то в исходной грамматике мы не знаем, разворачивать ли A по uv_1 или по uv_2. Однако в факторизованной грамматике можно отложить решение, развернув $A \to uA'$. Тогда после анализа того, что выводимо из u, можно развернуть $A' \to v_1$ или $A' \to v_2$.

§21. Построение таблиц предсказывающего анализатора

Алгоритм построения таблиц работает для любой КС-грамматики, так как реально строит функцию $\delta' : N \times T \longrightarrow 2^{(N \cup T)^\star} \cup \{\text{ERROR}\}$. То есть в ячейке таблицы в общем случае могут находиться несколько правил.

Алгоритм построения таблиц предсказывающего анализатора

Пусть для начала $\delta'(X, a) = \text{ERROR}$ для любых $X \in N$ и $a \in T$.

Затем проходим по всем правилам грамматики, и для каждого правила $X \to u$ выполняем следующее:

а) перебираем все $a \in \text{FIRST}(u)$ и добавляем u к $\delta'(X, a)$.
б) если $\varepsilon \in \text{FIRST}(u)$, перебираем все $b \in \text{FOLLOW}(X)$ и добавляем u к $\delta'(X, b)$.

Замечание. Добавление u к $\delta'(X, a)$ заключается в том, что если $\delta'(X, a) = \text{ERROR}$, то $\delta'(X, a)$ становится равным $\{u\}$, а если $\delta'(X, a) = A$, то $\delta'(X, a)$ становится равным $A \cup \{u\}$.

Пример. Грамматика арифметических выражений

$$
\begin{aligned}
E &::= T\ E'. \\
E' &::= \text{'+'}\ T\ E' \mid \varepsilon. \\
T &::= F\ T'. \\
T' &::= \text{'*'}\ F\ T' \mid \varepsilon. \\
F &::= n \mid \text{'('}\ E\ \text{')'}.
\end{aligned}
$$

Инициализируем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
E′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
E ::= T E′

Имеем
FIRST(T E′) = {'(', n},
FOLLOW(E) = {')', $}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
$E' ::= \text{'+'}\ T\ E'$

Имеем
$\text{FIRST('+' T E')} = \{\text{'+'}\}$,
$\text{FOLLOW(E')} = \{\text{')'}, \$\}$.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ERROR	ERROR
T	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
$E' ::= \varepsilon$

Имеем
FIRST(ε) = {ε},
FOLLOW(E') = {')', \$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	\$
E	ERROR	ERROR	T E'	T E'	ERROR	ERROR
E'	'+' T E'	ERROR	ERROR	ERROR	$\underline{\varepsilon}$	$\underline{\varepsilon}$
T	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
T'	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
T ::= F T′

Имеем
FIRST(F T′) = {'(',n},
FOLLOW(T) = {'+',')',$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T′	F T′	ERROR	ERROR
T′	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
T′ ::= '*' F T′

Имеем
FIRST('*' F T′) = {'*'},
FOLLOW(T′) = {'+',')',$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T′	F T′	ERROR	ERROR
T′	ERROR	'*' F T′	ERROR	ERROR	ERROR	ERROR
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
$T' ::= \varepsilon$

Имеем
FIRST(ε) = {ε},
FOLLOW(T') = {'+',')',\$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	\$
E	ERROR	ERROR	T E'	T E'	ERROR	ERROR
E'	'+' T E'	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T'	F T'	ERROR	ERROR
T'	$\underline{\varepsilon}$	'*' F T'	ERROR	ERROR	$\underline{\varepsilon}$	$\underline{\varepsilon}$
F	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Рассматриваем правило
F ::= n

Имеем
FIRST(n) = {n},
FOLLOW(F) = {'+','*',')',$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T′	F T′	ERROR	ERROR
T′	ε	'*' F T′	ERROR	ERROR	ε	ε
F	ERROR	ERROR	n	ERROR	ERROR	ERROR

Рассматриваем правило
F ::= '(' E ')'

Имеем
FIRST('(' E ')') = {'('},
FOLLOW(F) = {'+','*',')',$}.

Заполняем таблицу:

	'+'	'*'	n	'('	')'	$
E	ERROR	ERROR	T E′	T E′	ERROR	ERROR
E′	'+' T E′	ERROR	ERROR	ERROR	ε	ε
T	ERROR	ERROR	F T′	F T′	ERROR	ERROR
T′	ε	'*' F T′	ERROR	ERROR	ε	ε
F	ERROR	ERROR	n	'(' E ')'	ERROR	ERROR

§23. Расширенная БНФ

Определение. Говорят, что КС-грамматика $G = \langle N, T, E, S \rangle$ записана в расширенной форме Бэкуса-Наура (РБНФ), если её правила имеют вид $X \to R$, где $X \in N$ – нетерминальный символ, а R – дерево, синтаксис которого задаётся грамматикой

$$
\begin{array}{rcll}
R & ::= & \varepsilon & \text{; пустое дерево} \\
 & | & a & \text{; терминальный символ} \\
 & | & X & \text{; нетерминальныйсимвол} \\
 & | & R\,R & \text{; конкатенация поддеревьев} \\
 & | & R\,\text{'|'}\,R & \text{; альтернатива} \\
 & | & R^{\text{'*'}} & \text{; вхождение 0 и более раз} \\
 & | & R^{\text{'+'}} & \text{; вхождение 1 и более раз} \\
 & | & R^{\text{'?'}}. & \text{; вхождение 0 или 1 раз}
\end{array}
$$

Множество таких синтаксических деревьев мы будем обозначать как RHS.

При текстовой записи правил наивысший приоритет имеют операции '*', '+' и '?', за ними следует конкатенация, а наименьший приоритет имеет альтернатива. При этом круглые скобки служат для изменения приоритета.

Пример. Арифметические выражения.

$$\begin{aligned}
\text{Expr} &::= (\text{'+'} \mid \text{'}-\text{'})^{?}\ \text{Term}\ ((\text{'+'} \mid \text{'}-\text{'})\ \ \text{Term})^{\star}. \\
\text{Term} &::= \text{Factor}\ ((\text{'*'} \mid \text{'/'})\ \ \text{Factor})^{\star}. \\
\text{Factor} &::= \text{Number} \mid \text{'('}\ \text{Expr}\ \text{')'}.
\end{aligned}$$

В книгах Н. Вирта используется альтернативная запись РБНФ, называемая синтаксической нотацией Вирта:

Метасимвол	Значение
=	Разделяет левую и правую части правила.
.	Конец правила.
|	Альтернатива
[ ]	Вхождение 0 или 1 раз.
{ }	Вхождение 0 или более раз.
( )	Изменение приоритета операций.

Приоритет операций – такой же, как и в РБНФ.

Пример. Арифметические выражения.

```
Expr   = ["+"|"-"] Term { ("+"|"-") Term }.
Term   = Factor { ("*"|"/") Factor }.
Factor = Number | "(" Expr ")".
```

§24. Множества FIRST для РБНФ

Пусть дана грамматика $G = \langle N, T, E, S \rangle$, правила которой заданы в РБНФ. Будем обозначать нетерминальные символы грамматики буквами X и Y, терминальные символы – буквой a, а выражения РБНФ – буквами u и v.

Определим функцию $\mathcal{F} : \mathsf{RHS} \longrightarrow 2^T$, работающую в предположении, что для каждого нетерминального символа грамматики определено некоторое множество FIRST, и вычисляющую множество FIRST для любого выражения РБНФ, то есть $\mathsf{FIRST}\,(u) = \mathcal{F}[\![u]\!]$.

$$
\begin{array}{lcl}
\mathcal{F}[\![a]\!] & = & \{a\} \\
\mathcal{F}[\![X]\!] & = & \mathsf{FIRST}\,(X) \\
\mathcal{F}[\![uv]\!] & = & \begin{cases} \mathcal{F}[\![u]\!], & \text{если } \varepsilon \notin \mathcal{F}[\![u]\!] \\ (\mathcal{F}[\![u]\!] \setminus \{\varepsilon\}) \cup \mathcal{F}[\![v]\!], & \text{если } \varepsilon \in \mathcal{F}[\![u]\!] \end{cases} \\
\mathcal{F}[\![u \,|\, v]\!] & = & \mathcal{F}[\![u]\!] \cup \mathcal{F}[\![v]\!] \\
\mathcal{F}[\![u^\star]\!] & = & \mathcal{F}[\![u]\!] \cup \{\varepsilon\} \\
\mathcal{F}[\![u^+]\!] & = & \mathcal{F}[\![u]\!] \\
\mathcal{F}[\![u^?]\!] & = & \mathcal{F}[\![u]\!] \cup \{\varepsilon\} \\
\mathcal{F}[\![\varepsilon]\!] & = & \{\varepsilon\}
\end{array}
$$

Алгоритм построения множеств FIRST
для всех нетерминальных символов грамматики:

Шаг 1. Пусть FIRST $(X) = \emptyset$ для любого $X \in N$.

Шаг 2. Для каждого правила $X \to u$ добавить $\mathcal{F}[\![u]\!]$ в FIRST (X).

Шаг 3. Повторять шаг 2 до тех пор, пока множества FIRST не перестанут изменяться.

§25. Рекурсивный спуск

Мы будем рассматривать метод рекурсивного спуска применительно к LL(1)-грамматикам, записанным в РБНФ.

В синтаксическом анализаторе, написанном методом рекурсивного спуска, каждому нетерминалу грамматики соответствует отдельная функция, в которой закодирован эффект применения соответствующего нетерминалу правила (мы будем считать, что такое правило единственно).

Цель этой функции – анализ последовательности токенов, которые по её запросу выдаёт лексический анализатор, и проверка соответствия этой последовательности правилу грамматики:

```
X() {
    PARSE(правило);
}
```

Функция, соответствующая нетерминалу X грамматики, составляется с учётом того, что:

- существует глобальная переменная Sym, в которую ДО вызова функции помещается токен, соответствующий первому символу цепочки, в которую раскрывается нетерминал X;

- функция должна либо полностью потребить последовательность токенов, в которую раскрывается нетерминал X, либо сгенерировать сообщение об ошибке;

- после завершения функции переменная Sym должна содержать токен, непосредственно следующий за раскрытым нетерминалом X.

Код функции, соответствующая нетерминалу X грамматики, порождается из правила грамматики согласно следующим шаблонам:

1. Пустое дерево

PARSE(ε) $\Rightarrow$

```
    ; /* Ничего не делать. */
```

2. Терминальный символ

PARSE(a) $\Rightarrow$

```
    {
        if (Sym == a)
            Sym = NextToken();
        else
            ReportError();
    }
```

3. Нетерминальный символ

PARSE(X) $\Rightarrow$
    X ( ) ;

4. Конкатенация поддеревьев

PARSE($R_1\, R_2$) $\Rightarrow$
    { PARSE(R_1) ; PARSE(R_2) ; }

5. Альтернатива ($\varepsilon \notin$ FIRST (R_1))

PARSE($R_1 \,|\, R_2$) $\Rightarrow$
    {
        if ($Sym \in$ FIRST (R_1))
            PARSE(R_1) ;
        else
            PARSE(R_2) ;
    }

6. Вхождение 0 или более раз

```
PARSE( R* ) ⇒
        {
            while (Sym ∈ FIRST(R)))
                PARSE( R );
        }
```

7. Вхождение 1 или более раз

```
PARSE( R+ ) ⇒
        {
            do
                PARSE( R );
            while (Sym ∈ FIRST(R)))
        }
```

8. Вхождение 0 или 1 раз

PARSE($R^?$) $\Rightarrow$

```
{
    if  ( Sym ∈ FIRST (R)) )
          PARSE( R ) ;
}
```

Пример. Арифметические выражения.

1. Имеем правило для нетерминала Expr:

Expr ::= ('+'|'−')$^{?}$ Term (('+'|'−') Term)*.

Создаём функцию Expr:

```
Expr() {
    PARSE( ('+'|'−')? Term (('+'|'−') Term)* );
}
```

2. Правило для Expr представляет собой конкатенацию трёх поддеревьев:

```
Expr() {
    PARSE( ('+'|'−')? );
    PARSE( Term );
    PARSE( (('+'|'−') Term)* );
}
```

3. Раскрываем PARSE(('+'|'−')$^?$):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−'))
        PARSE( '+'|'−' );
    PARSE( Term );
    PARSE( (('+'|'−') Term)* );
}
```

4. Раскрываем PARSE(Term):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−'))
        PARSE( '+'|'−' );
    Term();
    PARSE( (('+'|'−') Term)* );
}
```

5. Раскрываем PARSE((('+'|'−') Term)*):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−'))
        PARSE( '+'|'−' );
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term))
        PARSE( ('+'|'−') Term );
}
```

6. Раскрываем PARSE('+'|'−'):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−')) {
        if (Sym ∈ FIRST('+'))
            PARSE( '+' );
        else
            PARSE( '−' );
    }
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term))
        PARSE( ('+'|'−') Term );
}
```

7. Раскрываем PARSE('+') и PARSE('−'):

```
Expr() {
    if (Sym ∈ FIRST('+'|'−')) {
        if (Sym ∈ FIRST('+')) {
            if (Sym == '+') Sym = NextToken();
            else ReportError();
        } else {
            if (Sym == '−') Sym = NextToken();
            else ReportError();
        }
    }
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term))
        PARSE( ('+'|'−') Term );
}
```

8. Упрощаем функцию:

```
Expr() {
      if (Sym == '+' || Sym == '-')
            Sym = NextToken();
      Term();
      while (Sym ∈ FIRST(('+'|'-') Term))
            PARSE( ('+'|'-') Term );
}
```

9. Раскрываем PARSE(('+'|'−') Term):

```
Expr() {
    if (Sym == '+' || Sym == '-')
        Sym = NextToken();
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term)) {
        PARSE( '+'|'−' );
        PARSE( Term );
    }
}
```

10. Раскрываем PARSE('+'|'−'):

```
Expr() {
    if (Sym == '+' || Sym == '-')
        Sym = NextToken();
    Term();
    while (Sym ∈ FIRST(('+'|'−') Term)) {
        if (Sym ∈ FIRST('+'|'−')) {
            if (Sym ∈ FIRST('+')) {
                if (Sym == '+') Sym = NextToken();
                else ReportError();
            } else {
                if (Sym == '-') Sym = NextToken();
                else ReportError();
            }
        }
        PARSE( Term );
    }
}
```

11. Упрощаем функцию и раскрываем PARSE(Term):

```
Expr() {
    if (Sym == '+' || Sym == '-')
        Sym = NextToken();
    Term();
    while (Sym == '+' || Sym == '-') {
        Sym = NextToken();
        Term();
    }
}
```

12. Имеем правило для нетерминала Term:
Term ::= Factor (('*'|'/') Factor)*.

Создаём функцию Term:

```
Term() {
    PARSE( Factor (('*'|'/') Factor)* );
}
```

13. После раскрытия PARSE(Factor (('*'|'/') Factor)*) и упрощения получаем:

```
Term() {
    Factor();
    while (Sym == '*' || Sym == '/') {
        Sym = NextToken();
        Factor();
    }
}
```

14. Имеем правило для нетерминала Factor:

Factor ::= Number|'(' Expr ')'.

Создаём функцию Factor:

```
Factor() {
    PARSE( Number|'(' Expr ')' );
}
```

15. Раскрываем PARSE(Number|'(' Expr ')'):

```
Factor() {
    if (Sym ∈ FIRST(Number))
        PARSE( Number );
    else
        PARSE( '(' Expr ')' );
}
```

15. Раскрываем PARSE(Number):

```
Factor() {
    if (Sym ∈ FIRST(Number)) {
        if (Sym == Number)
            Sym = NextToken();
        else
            ReportError();
    } else
        PARSE( '(' Expr ')' );
}
```

16. Упрощаем функцию:

```
Factor() {
    if (Sym == Number)
        Sym = NextToken();
    else
        PARSE( '(' Expr ')' );
}
```

17. Раскрываем PARSE('(' Expr ')'):

```
Factor() {
    if (Sym == Number)
        Sym = NextToken();
    else {
        PARSE( '(' );
        PARSE( Expr );
        PARSE( ')' );
    }
}
```

19. Раскрывая все вхождения PARSE, в итоге получаем:

```
Factor() {
    if (Sym == Number)
        Sym = NextToken();
    else {
        if (Sym == '(')
            Sym = NextToken();
        else
            ReportError();

        Expr();

        if (Sym == ')')
            Sym = NextToken();
        else
            ReportError();
    }
}
```

§22. Алгоритм Эрли

Earley, J. *An Efficient Context-Free Parsing Algorithm* [PhD Thesis]. – Carnegie-Mellon University, 1968.

Алгоритм Эрли подходит для любой КС-грамматики и для заданного предложения языка позволяет найти все левые выводы. Алгоритм Эрли относится к алгоритмам, работающих «сверху-вниз».

Оценка времени: $O(n)$ – для LL(1)-грамматик, $O\left(n^2\right)$ – для непротиворечивых грамматик, $O\left(n^3\right)$ – для всех остальных КС-грамматик.

Работа алгоритма основана на построении последовательности множеств, называемых *множествами Эрли*. Для входной цепочки $a_1 a_2 \ldots a_n$ конструируются $n+1$ множеств: начальное множество S_0 и по одному множеству S_i для каждого входного символа a_i.

Определение. *Элемент множества Эрли* (Earley item), принадлежащий множеству S_k – это тройка $\langle p, i, j \rangle$, в которой:
p – это правило грамматики;
i – позиция в правой части правила, показывающая, какая часть правила уже обработана;
$j \leq k$ – номер множества Эрли, в котором началось распознавание по правилу p.

Элемент $\langle p, i, j \rangle$ принято записывать в виде $[A \to u \bullet v,\ j]$, где $A \to uv$ – это правило p, а знак $\bullet$ задаёт позицию i.

В начале работы алгоритма Эрли в множество S_0 записывается единственный элемент $[S' \to \bullet S,\ 0]$, в котором S – это аксиома грамматики, а $S' \to S$ – вспомогательное правило, добавляемое в грамматику для удобства описания алгоритма.

В процессе работы алгоритма Эрли одни элементы порождают другие. Для того чтобы можно было восстановить последовательность применяемых правил, необходимо строить *дерево потомков*, в узлах которого находятся элементы Эрли, а дуги отражают тот факт, что один элемент породил другой элемент.

Алгоритм Эрли работает за $n+1$ шагов.

На каждом шаге происходит вычисление множества S_i и инициализация множества S_{i+1}:

```
S_{i+1} = ∅;
Q = S_i;
loop {
  Q' = ∅;
  for (каждый элемент x ∈ Q) {
    SCANNER(x);          /* Может добавлять в S_{i+1} */
    PREDICTOR(x, Q');    /* Может добавлять в Q' */
    COMPLETER(x, Q');    /* Может добавлять в Q' */
  }

  if (Q' ⊆ S_i) break;

  S_i = S_i ∪ Q';
  Q = Q';
}
```

Пусть $a_1 a_2 \ldots a_n$ – входная цепочка, A и B обозначают нетерминальные символы, b обозначает терминальный символ, u обозначает цепочку, составленную из терминальных и нетерминальных символов.

SCANNER(x). Если $x = [A \to \ldots \bullet b \ldots, j]$ и $b = a_{i+1}$,
добавить $[A \to \ldots b \bullet \ldots, j]$ в S_{i+1}.

PREDICTOR(x,Q). Если $x = [A \to \ldots \bullet B \ldots, j]$, то для каждого правила вида $B \to u$ добавить в Q элемент $[B \to \bullet u, i]$.
При этом, если $B \Rightarrow^\star \varepsilon$, то в Q также нужно добавить элемент $[A \to \ldots B \bullet \ldots, j]$.

COMPLETER(x,Q). Если $x = [A \to \ldots \bullet, j]$, то для каждого элемента $[B \to \ldots \bullet A \ldots, k] \in S_j$ добавить в Q элемент $[B \to \ldots A \bullet \ldots, k]$.

Пример. Грамматика $E \to E\,\text{'+'}\,E \mid n$ и входная цепочка $n\,\text{'+'}\,n$.

S_0

Q_1	S'	$\to$	$\bullet E$	, 0
Q_2	E	$\to$	$\bullet E\,\text{'+'}\,E$	, 0
	E	$\to$	$\bullet n$	, 0

n

S_1

Q_1	E	$\to$	$n\bullet$	, 0
Q_2	S'	$\to$	$E\bullet$	, 0
	E	$\to$	$E\bullet\,\text{'+'}\,E$	, 0

'+'

S_2

Q_1	E	$\to$	$E\,\text{'+'}\bullet\,E$	, 0
Q_2	E	$\to$	$\bullet E\,\text{'+'}\,E$	, 2
	E	$\to$	$\bullet n$	, 2

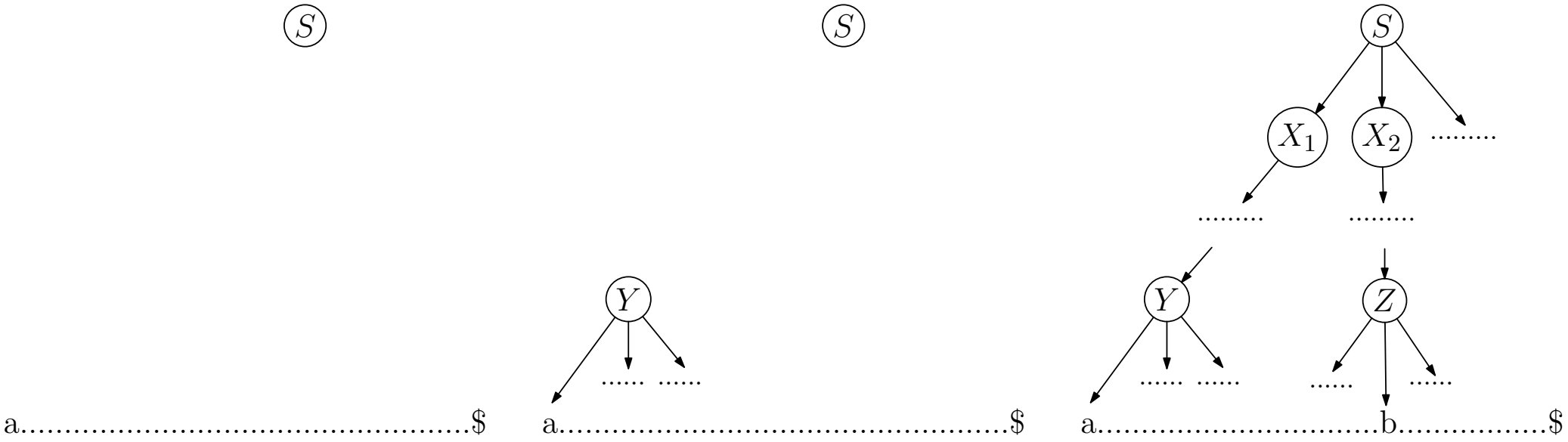
n

S_3

Q_1	E	$\to$	$n\bullet$	, 2
Q_2	E	$\to$	$E\,\text{'+'}\,E\bullet$	, 0
	E	$\to$	$E\bullet\,\text{'+'}\,E$	, 2
Q_3	S'	$\to$	$E\bullet$	, 0
	E	$\to$	$E\bullet\,\text{'+'}\,E$	, 0

§26. Синтаксический разбор типа «перенос–свёртка»

Последовательность формирования дерева вывода в процессе синтаксического разбора типа «перенос–свёртка» (снизу-вверх):



ПС-алгоритм (алгоритм синтаксического разбора типа «перенос–свёртка») строит правосторонний вывод предложения языка в обратном порядке.

Пример. Арифметические выражения.

$T = \{'+', '*', \mathsf{n}, '(', ')', \$\}$, $N = \{\mathsf{E}, \mathsf{T}, \mathsf{F}\}$, $S = \mathsf{E}$,

$P =$

$\mathsf{E} ::= \mathsf{E}\ '+'\ \mathsf{T} \mid \mathsf{T}.$

$\mathsf{T} ::= \mathsf{T}\ '*'\ \mathsf{F} \mid \mathsf{F}.$

$\mathsf{F} ::= \mathsf{n} \mid '('\ \mathsf{E}\ ')'.$

Для цепочки $\mathsf{n} * (\mathsf{n} + \mathsf{n})$ ПС-алгоритм строит вывод:

$\underline{\mathsf{n}} * (\mathsf{n} + \mathsf{n}) \Leftarrow \underline{\mathsf{F}} * (\mathsf{n} + \mathsf{n}) \Leftarrow \mathsf{T} * (\underline{\mathsf{n}} + \mathsf{n}) \Leftarrow \mathsf{T} * (\underline{\mathsf{F}} + \mathsf{n}) \Leftarrow \mathsf{T} * (\underline{\mathsf{T}} + \mathsf{n}) \Leftarrow$
$\Leftarrow \mathsf{T} * (\mathsf{E} + \underline{\mathsf{n}}) \Leftarrow \mathsf{T} * (\mathsf{E} + \underline{\mathsf{F}}) \Leftarrow \mathsf{T} * \underline{(\mathsf{E} + \mathsf{T})} \Leftarrow \mathsf{T} * \underline{(\mathsf{E})} \Leftarrow \underline{\mathsf{T} * \mathsf{F}} \Leftarrow \underline{\mathsf{T}} \Leftarrow \mathsf{E}.$

Определение. *Основа* правой сентенциальной формы u – это сочетание правила $X \to v$ и позиции цепочки v в u, такое, что вхождение v в u может быть заменено нетерминалом X для получения предыдущей правой сентенциальной формы в правостороннем выводе u.

После основы в правой сентенциальной форме идут только терминальные символы. В примере↑ все основы подчёркнуты.

ПС-алгоритм работает с двумя структурами данных:
– стек, в который помещаются символы грамматики ($N \cup T$), и дно которого помечено символом \$;
– входной буфер, содержащий нераспознанный суффикс входной цепочки и оканчивающийся символом \$.

В начале работы ПС-алгоритма стек содержит \$, а входной буффер – u\$, где u – распознаваемая цепочка.

ПС-алгоритм выполняет четыре действия:
1. *перенос* – символ в начале входного буфера переносится в стек;
2. *свёртка* – основа на вершине стека заменяется нетерминалом;
3. *допуск* – успешное окончание разбора (во входном буфере – \$, в стеке – \$$S$, где S – аксиома грамматики);
4. *ошибка* – вызов подпрограммы реакции на ошибку во входном потоке.

Основные проблемы ПС-алгоритма – поиск основы в стеке и выбор правила, по которому нужно осуществлять свёртку. В общем случае ПС-алгоритм осуществляет возвраты (см. [Ахо, Ульман]).

Пример. Разбор цепочки $n * (n + n)$.

№	Стек	Вход	Действие
1	$	n*(n+n)$	Перенос
2	$n	*(n+n)$	Свёртка по правилу F → n
3	$F	*(n+n)$	Свёртка по правилу T → F
4!	$T	*(n+n)$	Перенос ×3
7	$T*(n	+n)$	Свёртка по правилу F → n
8	$T*(F	+n)$	Свёртка по правилу T → F
9	$T*(T	+n)$	Свёртка по правилу E → T
10	$T*(E	+n)$	Перенос ×2
12	$T*(E+n	)$	Свёртка по правилу F → n
13	$T*(E+F	)$	Свёртка по правилу T → F
14!	$T*(E+T	)$	Свёртка по правилу E → E '+' T
15	$T*(E	)$	Перенос
16	$T*(E)	$	Свёртка по правилу F → '(' E ')'
17!	$T*F	$	Свёртка по правилу T → T '*' F
18	$T	$	Свёртка по правилу E → T
19	$E	$	Допуск

§27. Недетерминированные SLR-распознаватели

Определение. *Активный префикс* (viable prefix) – это префикс правосентенциальной формы, который не выходит за правый конец основы этой сентенциальной формы.

Если ПС-алгоритм работает правильно (то есть не пошёл по ложному пути), то его стек содержит знак \$, выше которого располагается активный префикс.

Пример. На 14-ом шаге разбора цепочки $n * (n + n)$ (см. пример в §26) в стеке содержится цепочка
\$T*(E+T
в которой T*(E+T является активным префиксом.

Определение. *SLR-ситуация* (SLR item) – это пара $\langle p, i \rangle$, в которой:

p – правило грамматики;

i – позиция в правой части правила, показывающая, какая часть правила уже обработана.

Далее мы будем называть SLR-ситуации просто *ситуациями*.

Ситуацию $\langle p, i \rangle$ принято записывать в виде $X \to u \bullet v$ или $[X \to u \bullet v]$, где $X \to uv$ – это правило p, а знак $\bullet$ задаёт позицию i.

Для каждого правила $X \to x_1 x_2 \dots x_n$ можно построить $(n + 1)$ ситуаций:
$X \to \bullet x_1 x_2 \dots x_n$,
$X \to x_1 \bullet x_2 \dots x_n$,
$X \to x_1 x_2 \bullet \dots x_n$,
...,
$X \to x_1 x_2 \dots \bullet x_n$,
$X \to x_1 x_2 \dots x_n \bullet$.

Для правила вида $X \to \varepsilon$ возможна единственная ситуация $X \to \bullet$.

Определение. *Расширенная грамматика* для грамматики $G = \langle N, T, P, S \rangle$ – это грамматика $G' = \langle N', T, P', S' \rangle$, в которой $N' = N \cup \{S'\}$, $P' = P \cup \{S' \to S\}$.

Основные проблемы ПС-алгоритма – поиск основы в стеке и выбор правила, по которому нужно осуществлять свёртку – для некоторых КС-грамматик можно решить с помощью распознавателей активных префиксов.

Определение. *Недетерминированный SLR-распознаватель активных префиксов* для расширенной грамматики $G' = \langle N', T, P', S' \rangle$ – это конечный автомат, состояниями которого являются все возможные SLR-ситуации для P', а переходы между ними задаются условиями:
1) $[X \to u \bullet xv] \overset{x}{\mapsto} [X \to ux \bullet v]$, где $x \in N' \cup T$;
2) $[X \to u \bullet Yv] \overset{\varepsilon}{\mapsto} [Y \to \bullet w]$, где $Y \in N'$.
При этом начальным состоянием автомата является ситуация $[S' \to \bullet S]$, а заключительными состояниями являются все ситуации вида $[X \to u\bullet]$.

Пример.

Правила грамматики:

$$
\begin{array}{lcl}
S' & \to & S \\
S & \to & L = R \\
S & \to & R \\
L & \to & \star R \\
L & \to & v \\
R & \to & L
\end{array}
$$

Примеры выводимых предложений:

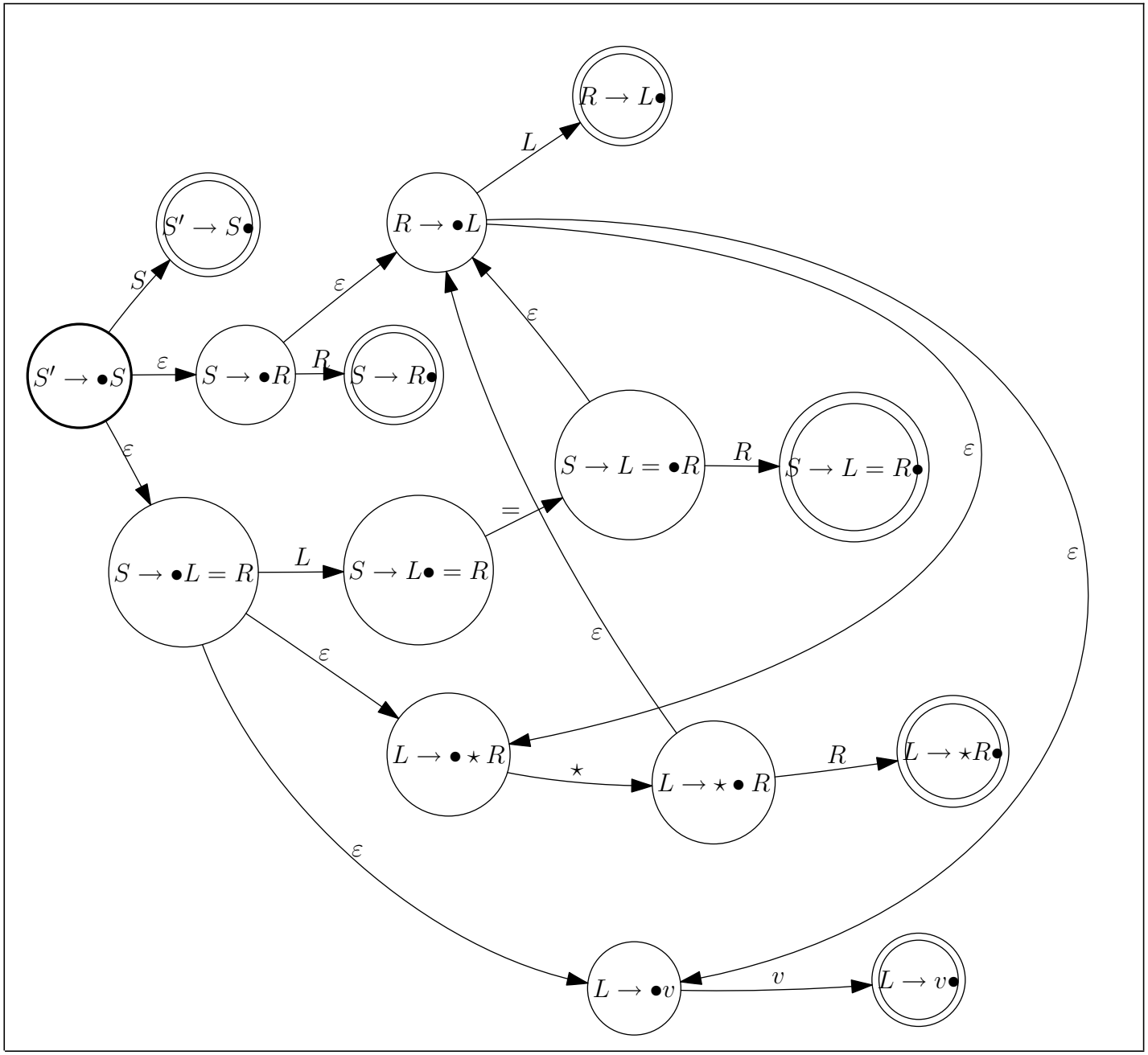
v

$\star v$

$\star \star v$

$v = v$

$\star \star v = \star v$



Если «натравить» SLR-распознаватель на активный префикс, лежащий в стеке ПС-алгоритма, то в результате мы можем получить некоторую ситуацию, которая является подсказкой для ПС-алгоритма:
– заключительная ситуация вида $[X \to u\bullet]$ говорит о том, что нужно выполнить свёртку по правилу $X \to u$;
– ситуация вида $[X \to u \bullet xv]$ означает, что нужно выполнить перенос.

Пример. Пусть выполняется синтаксический анализ цепочки $\star\star v = \star v$, и на некотором шаге работы ПС-алгоритма на стеке лежит активный префикс $L = \star R$.

Распознавание активного префикса:
$[S' \to \bullet S] \overset{\varepsilon}{\mapsto} [S \to \bullet L = R] \overset{L}{\mapsto} [S \to L\bullet = R] \overset{=}{\mapsto} [S \to L = \bullet R] \overset{\varepsilon}{\mapsto} [R \to \bullet L] \overset{\varepsilon}{\mapsto}$
$\overset{\varepsilon}{\mapsto} [L \to \bullet \star R] \overset{\star}{\mapsto} [L \to \star \bullet R] \overset{R}{\mapsto} [L \to \star R\bullet]$.

$[L \to \star R\bullet]$ – заключительное состояние, означающее свёртку основы $\star R$ по правилу $L \to \star R$.

Если распознавание активного префикса оказалось невозможным, это означает, что обнаружена ошибка.

Пример. Пусть выполняется синтаксический анализ цепочки $\star = \star v$, и на некотором шаге работы ПС-алгоритма на стеке лежит активный префикс $\star =$.

Распознавание активного префикса:
$[S' \to \bullet S] \overset{\varepsilon}{\mapsto} [S \to \bullet L = R] \overset{\varepsilon}{\mapsto} [L \to \bullet \star R] \overset{\star}{\mapsto} [L \to \star \bullet R] \mapsto ?$

Активный префикс не может быть распознан – обнаружена ошибка.

Если в результате распознавания активного префикса мы получили более одной ситуации, это означает, что КС-грамматика слишком сложна для SLR-распознавателя.

Пример. Пусть выполняется синтаксический анализ цепочки $v = v$, и на некотором шаге работы ПС-алгоритма на стеке лежит активный префикс L.

Распознавание активного префикса:
1. $[S' \to \bullet S] \overset{\varepsilon}{\mapsto} [S \to \bullet L = R] \overset{L}{\mapsto} [S \to L\bullet = R]$ – перенос;
2. $[S' \to \bullet S] \overset{\varepsilon}{\mapsto} [S \to \bullet R] \overset{\varepsilon}{\mapsto} [R \to \bullet L] \overset{L}{\mapsto} [R \to L\bullet]$ – свёртка по правилу $R \to L$.

Имеем так называемый конфликт «перенос/свёртка».

Кроме конфликтов «перенос/свёртка» бывают ещё конфликты «свёртка/свёртка».

§28. Детерминированные SLR-распознаватели

Если выполнить детерминизацию недетерминированного SLR-распознавателя, то получится *детерминированный SLR-распознаватель активного префикса*, состояниями которого являются не отдельные ситуации, а множества ситуаций.

При этом начальным состоянием детерминированного SLR-распознавателя является состояние, содержащее ситуацию $[S' \to \bullet S]$, а заключительными состояниями являются состояния, содержащие заключительные состояния соответствующего недетминированного SLR-распознавателя.

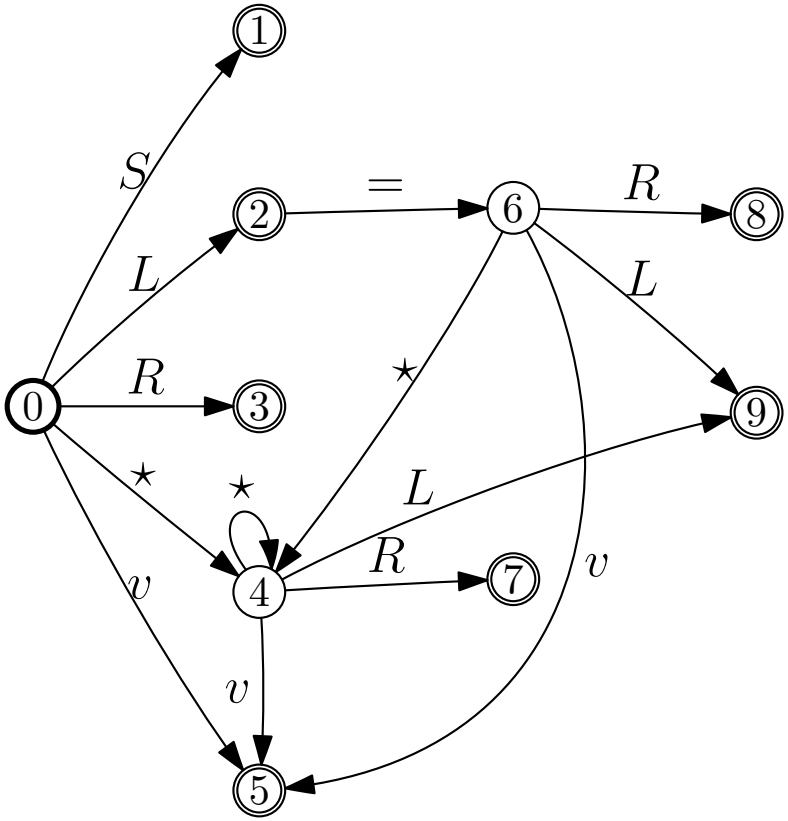
На практике используются только детерминированные SLR-распознаватели, потому что они позволяют распознавать активные префиксы за время, пропорциональное длине префикса.

Пример.

Правила грамматики:

$$
\begin{array}{lcl}
S' & \to & S \\
S & \to & L = R \\
S & \to & R \\
L & \to & \star R \\
L & \to & v \\
R & \to & L
\end{array}
$$

(В состоянии 2 – конфликт «перенос/свёртка».)



0 :
$$
\begin{array}{lcl}
S' & \to & \bullet S \\
S & \to & \bullet L = R \\
S & \to & \bullet R \\
L & \to & \bullet \star R \\
L & \to & \bullet v \\
R & \to & \bullet L
\end{array}
$$

1 :
$$
S' \to S\bullet
$$

2 :
$$
\begin{array}{lcl}
S & \to & L\bullet = R \\
R & \to & L\bullet
\end{array}
$$

3 :
$$
S \to R\bullet
$$

4 :
$$
\begin{array}{lcl}
L & \to & \star \bullet R \\
R & \to & \bullet L \\
L & \to & \bullet \star R \\
L & \to & \bullet v
\end{array}
$$

5 :
$$
L \to v\bullet
$$

6 :
$$
\begin{array}{lcl}
S & \to & L = \bullet R \\
R & \to & \bullet L \\
L & \to & \bullet \star R \\
L & \to & \bullet v
\end{array}
$$

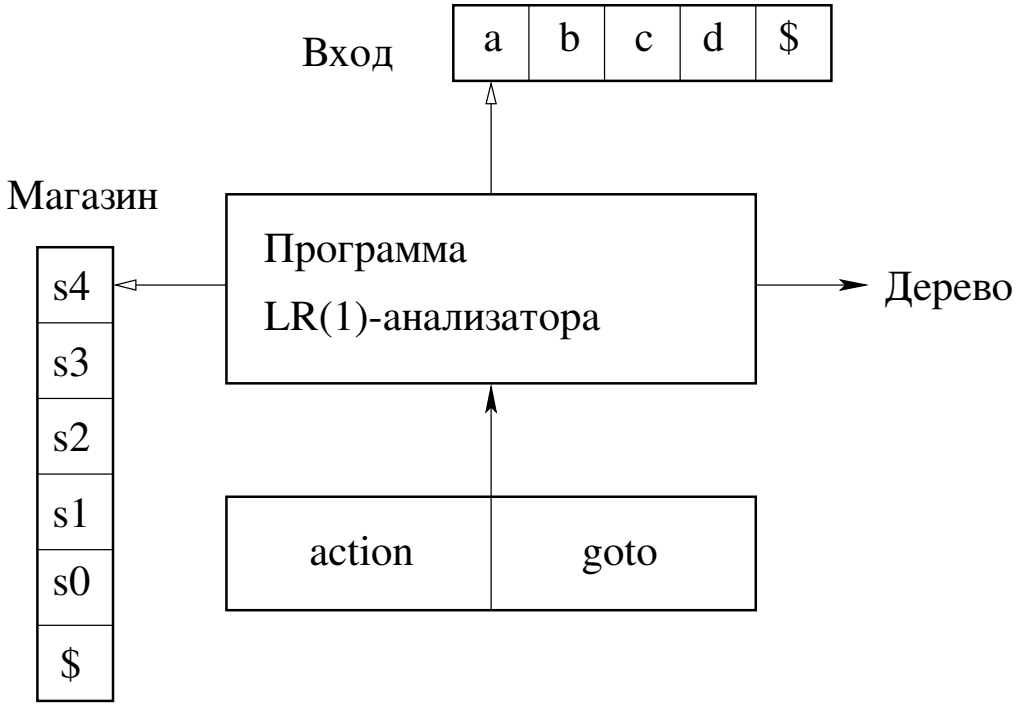
7 :
$$
L \to \star R\bullet
$$

8 :
$$
S \to L = R\bullet
$$

9 :
$$
R \to L\bullet
$$

§29. Алгоритм LR(1)-анализа

Алгоритм LR(1)-анализа является вариантом ПС-алгоритма, работающим за линейное время. В стеке LR(1)-анализатора содержится последовательность состояний, через которые прошёл SLR-распознаватель (или более сложный LR-распознаватель) в процессе распознавания активного префикса. LR(1)-анализатор управляется таблицами распознавателя, не содержащими конфликтов.



action:

$State \times T \longrightarrow$ перенос $State$
| свёртка $X \to u$
| допуск
| ошибка

goto:

$State \times N' \longrightarrow State$
| ошибка

```
LR1Parse(N, T (содержит $), S,
         s0, action, goto, цепочка x) {
  Result = пустая последовательность;
  поместить в стек $, а затем s0;
  a = первый символ из x;
  loop {
      s = состояние на вершине стека;
      if (action[s,a] == "перенос s'") {
          поместить в стек s';
          a = следующий символ из x;
      } else if (action[s,a] == "свёртка X → u") {
          снять со стека |u| состояний.
          s' = состояние на вершине стека;
          поместить в стек goto[s',X];
          добавить правило X → u в Result;
      } else if (action[s,a] == "допуск")
          return Result;
      else error();
  }
}
```

Таблица action[s, a], где s – номер состояния SLR-распознавателя, а $a \in T \cup \{\$\}$, задаётся следующим образом:
– action[s,a] $\ni$ «перенос s'», если $s \overset{a}{\mapsto} s'$;
– action[s,a] $\ni$ «свёртка $X \to u$», если $[X \to u\bullet] \in s$ и $a \in \mathrm{FOLLOW}(X)$;
– action[s,\$] $\ni$ «допуск», если $[S' \to S\bullet] \in s$;
– action[s,a] $\ni$ «ошибка», если ни одна из вышеприведённых альтернатив не сработала.

Таблица goto[s,X], где s – номер состояния SLR-распознавателя, а $X \in N'$, задаётся следующим образом:
– goto[s,X] $= s'$, если $s \overset{X}{\mapsto} s'$;
– goto[s,X] $=$ «ошибка» в противном случае.

Пример.

FOLLOW $(S') = \{\$\}$,
FOLLOW $(S) = \{\$\}$,
FOLLOW $(L) = \{=, \$\}$,
FOLLOW $(R) = \{=, \$\}$

№	action				goto			
	$=$	$\star$	v	\$	S'	S	L	R
0	ош.	пер.4	пер.5	ош.	ош.	1	2	3
1	ош.	ош.	ош.	доп.	ош.	ош.	ош.	ош.
2	пер.6, св.$R \to L$	ош.	ош.	св.$R \to L$	ош.	ош.	ош.	ош.
3	ош.	ош.	ош.	св.$S \to R$	ош.	ош.	ош.	ош.
4	ош.	пер.4	пер.5	ош.	ош.	ош.	9	7
5	св.$L \to v$	ош.	ош.	св.$L \to v$	ош.	ош.	ош.	ош.
6	ош.	пер.4	пер.5	ош.	ош.	ош.	9	8
7	св.$L \to \star R$	ош.	ош.	св.$L \to \star R$	ош.	ош.	ош.	ош.
8	ош.	ош.	ош.	св.$S \to L = R$	ош.	ош.	ош.	ош.
9	св.$R \to L$	ош.	ош.	св.$R \to L$	ош.	ош.	ош.	ош.

Лекция 5

Синтаксически управляемая трансляция

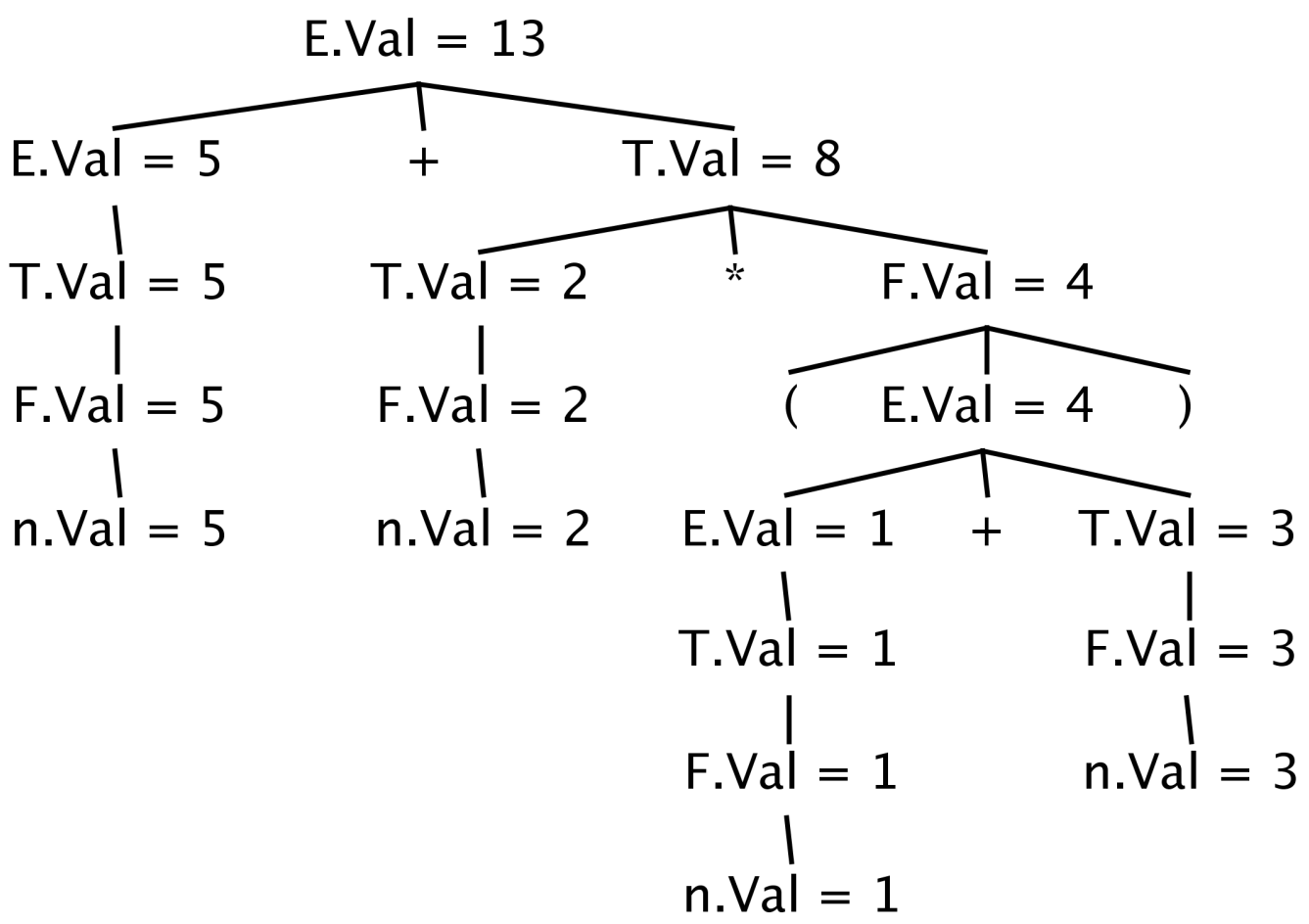
§30. Синтаксически управляемые определения

Синтаксический анализ позволяет ответить на вопрос, является ли цепочка предложением языка, и построить дерево вывода предложения. Можно расширить алгоритмы синтаксического анализа, приспособив их для перевода предложений в другой язык.

Определение. *Аннотированное дерево синтаксического разбора* – это дерево разбора для некоторого предложения языка, с каждым узлом которого связана структура, состоящая из именованных полей – *атрибутов*.

Значениями атрибутов может быть всё, что угодно – числа, строки, деревья, и т.д. Заставив алгоритм синтаксического анализатора вычислять значения атрибутов по специальным *семантическим правилам*, ассоциированным с правилами грамматики, мы можем добиться перевода предложений на другой язык. Это называется *синтаксически управляемой трансляцией*.

Пример.



$$
\begin{aligned}
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T \star F \\
T &\rightarrow F \\
F &\rightarrow (E) \\
F &\rightarrow n
\end{aligned}
$$

Значения атрибутов в узлах аннотированного дерева вычисляются с помощью синтаксически управляемого определения.

Определение. *Синтаксически управляемое определение* (syntax-directed definition) – это тройка $\langle G, A, R\rangle$, в которой:
$G = \langle N, T, S, P\rangle$ – КС-грамматика;
A – конечное множество атрибутов, при этом с каждым символом $x \in N \cup T$ связано 0 или более атрибутов;
R – конечное множество семантических правил, при этом с каждым правилом грамматики связано 0 или более семантических правил.
Семантическое правило, связанное с правилом грамматики
$X \to x_1 x_2 \dots x_n$, имеет вид $b := f(c_1, c_2, \dots, c_k)$,
где b – атрибут символа, входящего в правило,
$c_1, c_2, \dots, c_k$ – атрибуты любых других символов, входящих в правило,
f – функция, вычисляющая значение атрибута b по значениям атрибутов $c_1, c_2, \dots, c_k$.

Определение. *Атрибутная грамматика* (attribute grammar) – это синтаксически управляемое определение, в котором функции в семантических правилах не имеют побочных эффектов.

Пример. Атрибутная грамматика арифметических выражений.

Правила грамматики	Семантические правила
$E \to E_1 + T$	$E.Val := E_1.Val + T.Val$
$E \to T$	$E.Val := T.Val$
$T \to T_1 \star F$	$T.Val := T_1.Val \times F.Val$
$T \to F$	$T.Val := F.Val$
$F \to (E)$	$F.Val := E.Val$
$F \to n$	$F.Val := n.LexVal$

Говорят, что атрибут b одного из символов правила грамматики p, *зависит* от атрибутов $c_1, c_2, \ldots, c_k$ других символов правила p, если с правилом p связано семантическое правило $b := f(c_1, c_2, \ldots, c_k)$.

Определение. *Синтезируемый атрибут*:

– для терминального символа это атрибут, вычисляемый на этапе лексического анализа;

– для нетерминального символа это атрибут, вычисляемый на основе синтезируемых атрибутов дочерних узлов дерева синтаксического разбора.

Определение. *Наследуемый атрибут* – это атрибут символа, вычисляемый на основе атрибутов родительского узла или соседних узлов (то есть, узлов, имеющих того же родителя).

Пример. Атрибутная грамматика объявлений переменных.

Правила грамматики	Семантические правила
$D \to T\ L$	$L.In := T.Type$
$T \to$ int	$T.Type := integer$
$T \to$ float	$T.Type := real$
$L \to L_1,\ v$	$L_1.In := L.In$ $v.Type = L.In$
$L \to v$	$v.Type := L.In$

Атрибут $T.Type$ – синтезируемый, атрибуты $L.In$ и $v.Type$ – наследуемые.

§31. Порядок выполнения семантических правил

Преимущество синтаксически управляемых определений – компактность, достигаемая за счёт того, что не указывается порядок вычисления семантических правил.

Методы выбора порядка выполнения семантических правил:
– метод, связанный с анализом зависимостей атрибутов в дереве разбора;
– метод, основанный на внедрении семантических действий в правила грамматики.

Метод выбора порядка выполнения семантических правил, связанный с анализом зависимостей атрибутов в дереве разбора

Определение. *Граф зависимостей атрибутов* для дерева синтаксического разбора – это направленный граф, в узлах которого расположены атрибуты узлов дерева, а дуги обозначают зависимость атрибутов (дуга ведёт из узла c в узел b, если атрибут b зависит от атрибута c).

Если синтаксически управляемое определение не содержит ошибок, то граф зависимостей атрибутов будет ациклическим для любого дерева синтаксического разбора.

Любая топологическая сортировка графа зависимостей атрибутов даёт правильный порядок выполнения семантических правил, вычисляющих атрибуты. (Топологическая сортировка – это упорядочивание узлов направленного ациклического графа согласно частичному порядку, заданному дугами этого графа на множестве его узлов.)

Метод задания порядка выполнения семантических правил с помощью схемы трансляции

Определение. *Семантическое действие* для правила p грамматики – это семантическое правило для p, привязанное к определённой позиции в правой части p.

Пример.

```
Loop ::= 'while' { Expr.SymT := Loop.SymT; } Expr
         'do' { Block.SymT := Loop.SymT; } Block.
```

Семантическое действие разбивает правую часть правила на две подцепочки, которые мы будем называть *префиксом и суффиксом семантического действия*.

Подразумевается, что семантическое действие должно выполняться в тот момент синтаксического анализа, когда полностью распознана часть входной цепочки, соответствующая префиксу этого действия.

Определение. *Схема трансляции* – это синтаксически управляемое определение, семантические правила которого являются семантическими действиями и удовлетворяют условиям:

1. наследуемый атрибут для символа x вычисляется в действии, предшествующем символу x;
2. действие не должно обращаться к синтезируемому атрибуту символа, расположенного справа от действия;
3. синтезируемый атрибут для нетерминала в левой части правила вычисляется в действии, расположенном справа от последнего символа правила.

Пример. Схема трансляции объявлений переменных.

```
D ::= T { L.In := T.Type; } L.
T ::= 'int' { T.Type := integer; }
    | 'float' { T.Type := real; }.
L ::= { L1.In := L.In; } L1 ',' { v.Type := L.In } v
    | { v.Type := L.In } v.
```

§32. Восходящее выполнение S-атрибутных определений

Определение. *S-атрибутное определение* – это синтаксически управляемое определение, в котором применяются только синтезируемые атрибуты.

Синтезируемые атрибуты могут быть вычислены восходящим синтаксическим анализатором в процессе разбора входной цепочки.

K символам грамматики (или состояниям SLR-распознавателя), содержащимся в стеке ПС-анализатора, добавляется дополнительная информация: значения синтезируемых атрибутов.

Возможно выделение специального стека для синтезируемых атрибутов.

При выполнении свёртки по правилу $X \to u$ со стека снимается $|u|$ символов (или состояний), и на основе значений их атрибутов вычисляется значение синтезируемого атрибута символа X.

Пример. Разбор цепочки $3 * (4 + 5)$.

№	Стек	Атрибуты	Вход	Действие
1	$		3*(4+5)$	Перенос
2	$n	3	*(4+5)$	Свёртка по правилу F → n
3	$F	3	*(4+5)$	Свёртка по правилу T → F
4	$T	3	*(4+5)$	Перенос ×3
7	$T*(n	3, , , 4	+5)$	Свёртка по правилу F → n
8	$T*(F	3, , , 4	+5)$	Свёртка по правилу T → F
9	$T*(T	3, , , 4	+5)$	Свёртка по правилу E → T
10	$T*(E	3, , , 4	+5)$	Перенос ×2
12	$T*(E+n	3, , , 4, , 5	)$	Свёртка по правилу F → n
13	$T*(E+F	3, , , 4, , 5	)$	Свёртка по правилу T → F
14	$T*(E+T	3, , , 4, , 5	)$	Свёртка по правилу E → E + T
15	$T*(E	3, , , 9	)$	Перенос
16	$T*(E)	3, , , 9 ,	$	Свёртка по правилу F → (E)
17	$T*F	3, , 9	$	Свёртка по правилу T → T * F
18	$T	27	$	Свёртка по правилу E → T
19	$E	27	$	Допуск

Лекция 6

Семантический анализ

§33. Постановка задачи семантического анализа

Определение. *Символ* (symbol) – это именованная сущность в программе, определяемая парой $\langle \text{name}, \text{info} \rangle$, где name – идентификатор сущности, info – описание сущности.

Примеры сущностей: переменные, типы данных, функции, модули и т.п.

Две и более сущности могут иметь один идентификатор. При этом им соответствуют разные символы: $\langle \text{name}, \text{info1} \rangle$, $\langle \text{name}, \text{info2} \rangle$ и т.д.

Реже бывает, что два и более идентификатора обозначают одну сущность (тогда говорят, что сущность имеет псевдонимы). При этом каждому идентификатору соответствует свой символ: $\langle \text{name1}, \text{info} \rangle$, $\langle \text{name2}, \text{info} \rangle$ и т.д.

Семантика языка программирования определяет, в каких узлах дерева синтаксического разбора программы может использоваться идентификатор того или иного символа.

Если идентификатор символа может использоваться в некотором узле, говорят, что символ *виден* в этом узле.

В большинстве языков программирования видимость символа – это статическая характеристика символа, то есть она определяется во время компиляции.

Категории символов, для которых происходит выделение памяти во время выполнения программы (например, переменные), в дополнение к видимости имеют ещё одну характеристику – время жизни.

Определение. *Время жизни* (lifetime, extent) символа – это часть времени выполнения программы от момента выделения памяти для сущности, соответствующей символу, до момента освобождения этой памяти.

Время жизни – это динамическая характеристика, то есть оно определяется во время выполнения программы.

Следует понимать, что во время выполнения программы символ может присутствовать в памяти в нескольких экземплярах (например, локальные переменные в рекурсивных функциях).

Определение. *Таблица символов* (symbol table) для узла дерева синтаксического разбора – это отображение идентификаторов символов, видимых в этом узле, в описания соответствующих этим символам сущностей.

Таблицы символов в компиляторах реализованы в виде хеш-таблиц. Идентификаторы являются ключами этих хеш-таблиц.

Лексический анализатор связывает с каждым идентификатором уникальное целое число – код идентификатора. Это существенно ускоряет вычисление хеш-функции.

Иногда семантика языка программирования допускает, что множества идентификаторов разных категорий сущностей, видимых в одном и том же узле дерева, могут пересекаться.

Пример. В C теги структур могут совпадать с именами переменных:

```
struct A
{
    int x, y;
};

void f()
{
    struct A A;
    A.x = 5;
}
```

В таком случае нужно иметь по отдельной таблице символов для каждой категории сущностей.

Семантический анализ программы решает две задачи:

1. построение таблицы символов для каждой области в дереве синтаксического разбора программы и генерация ошибок для узлов дерева, в которых используются идентификаторы, не являющиеся ключами соответствующих таблиц символов;

2. проверка допустимости применения операций к сущностям программы (для языков со статической типизацией может потребоваться вычисление типов для всех узлов, соответствующих выражениям в программе).

§34. Области и локальные таблицы символов

Определение. *Область* (scope) – это множество узлов дерева синтаксического разбора программы, в которых по семантике языка программирования видимы одни и те же символы.

Использование областей увеличивает эффективность семантического анализатора благодаря тому, что для всех узлов, принадлежащих области, строится одна таблица символов.

Определение. *Вложенная область* (nested scope) – это область, узлы которой являются дочерними по отношению к узлу другой области.

Вложенная область наследует таблицу символов родительского узла, дополняя её собственными символами. При этом собственные символы могут экранировать часть унаследованных символов.

Определение. Область A является *открытой* для области B, если область B непосредственно вложена в область A, или если специальная конструкция языка открывает область A для области B.

Пример. Операция доступа к полю структуры открывает область, содержащую символы, описывающие поля. В языке Pascal поля структуры открываются оператором with:

```
type
  R = record
        a, b: integer
      end;
var
  x: R;
begin
  with x do
  begin
    a := 5;
    b := 10;
  end;
end.
```

Определение. *Эффективные символы* области – это все символы, видимые в узлах области.

Определение. *Собственные символы* области – это символы, объявленные внутри области.

Определение. *Унаследованные символы* области – это эффективные символы областей, открытых для этой области.

Определение. *Экранированные символы* области – это унаследованные символы, идентификаторы которых совпадают с идентификаторами собственных символов.

Таким образом, множество E эффективных символов области вычисляется по формуле
$E = P \cup I \setminus S$,
где P – множество собственных символов;
I – множество унаследованных символов;
S – множество экранированных символов.

Пример. Вложенные области в программе на языке Pascal.

```
program P;
var x, y: integer;

  procedure F;
  var x, a, b: real;

    procedure G;
    var a, c: char;
    begin
      (* [a->char, c->char] +
         унаследованные [x->real, a->real, b->real] +
         унаследованные [x->integer, y->integer] *)
    end;

  begin
    (* [x->real, a->real, b->real] +
       унаследованные [x->integer, y->integer] *)
  end;

begin
  (* [x->integer, y->integer] *)
end.
```

Определение. *Локальная таблица символов* (local symbol table) – это таблица символов, в которой перечисляются только собственные символы области и имеются ссылки на локальные таблицы областей, открытых для данной области.

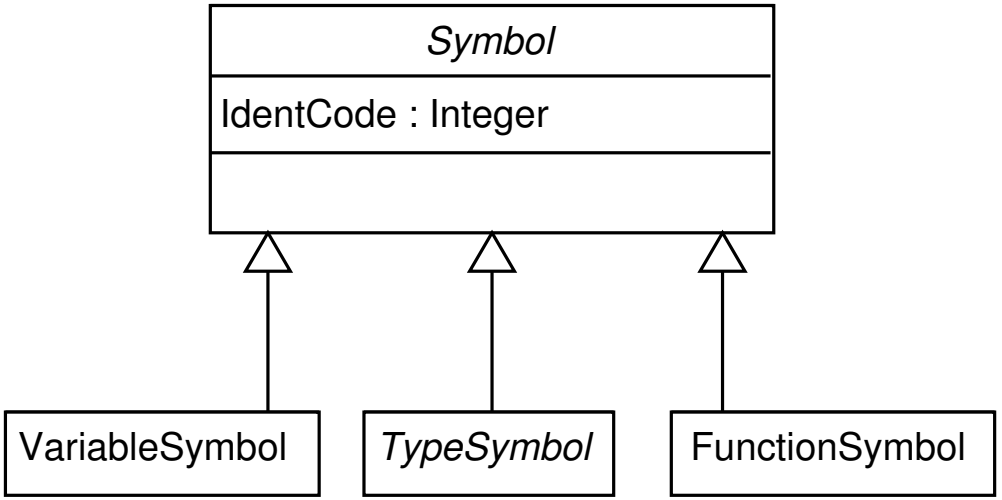
Локальные таблицы позволяют экономить память за счёт того, что каждый символ входит только в одну таблицу.

На выходе семантического анализатора, строящего локальные таблицы, получается ациклический направленный граф локальных таблиц. Корнями этого графа являются локальные таблицы самых внешних областей.

Поиск символа для узла дерева синтаксического разбора начинается с обращения к локальной таблице области X, в которую входит узел, и если символ в ней не найден, то поиск рекурсивно запускается для локальных таблиц областей, открытых для X.

§35. Проектирование объектно-ориентированных таблиц символов

Символ представляется абстрактным классом Symbol, от которого наследуются классы основных категорий символов.



Поле IdentCode содержит код, присвоенный идентификатору лексическим анализатором.

Класс SymbolTable представляет локальную таблицу символов.

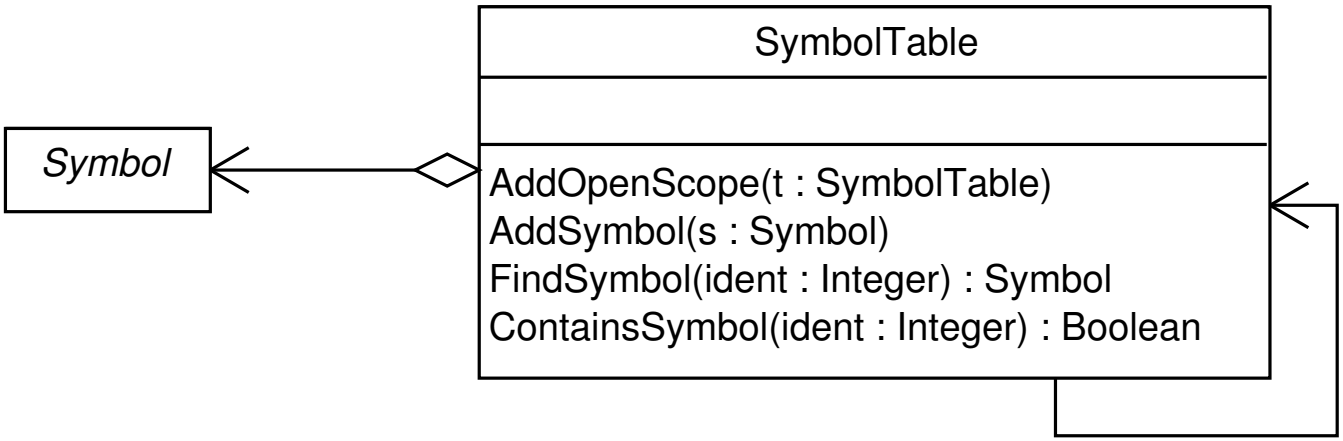
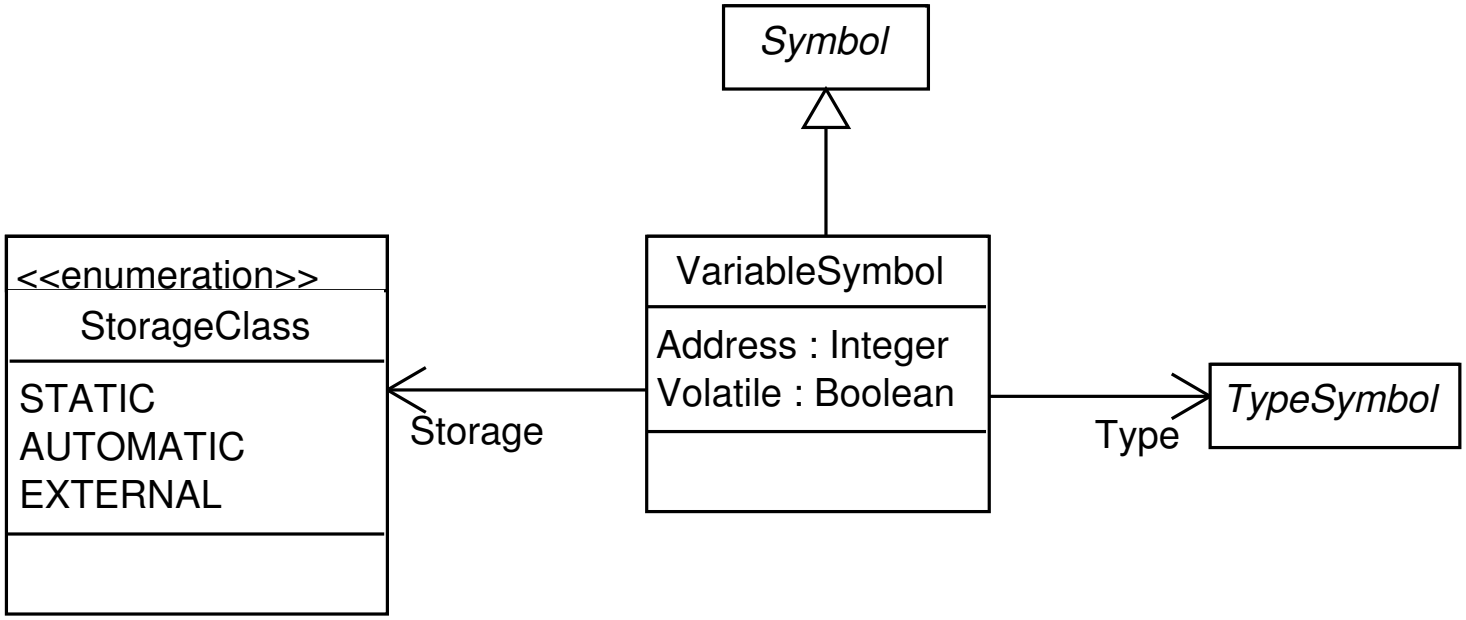


Таблица символов содержит хеш-таблицу символов и ссылки на таблицы символов открытых областей.

Метод AddOpenScope добавляет открытую область, метод AddSymbol добавляет новый символ, метод FindSymbol выполняет поиск эффективного символа по идентификатору, а метод ContainsSymbol проверяет наличие собственного символа по идентификатору.

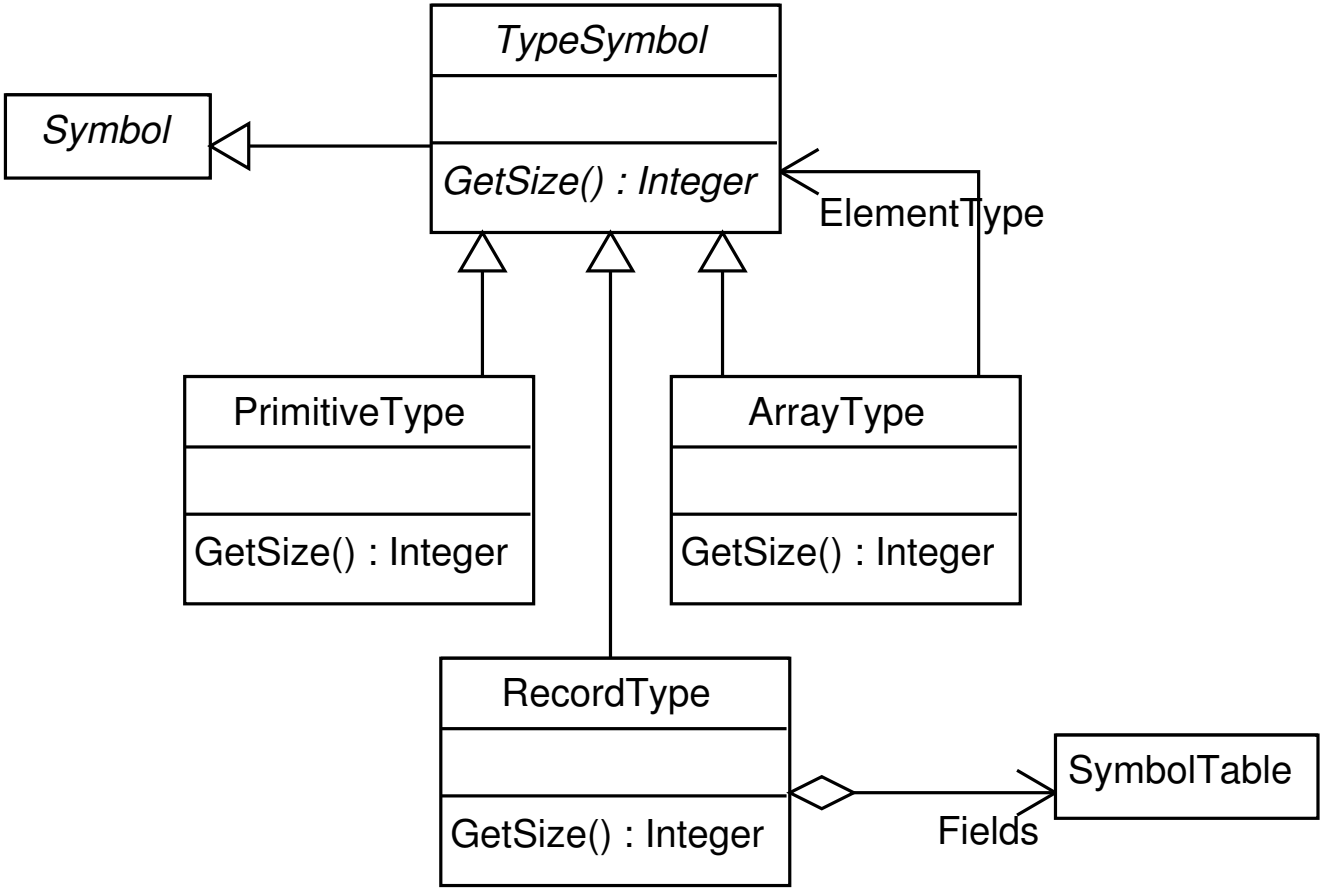
Класс VariableSymbol служит для представления переменных.



Перечисление StorageClass содержит варианты классов памяти для переменной.

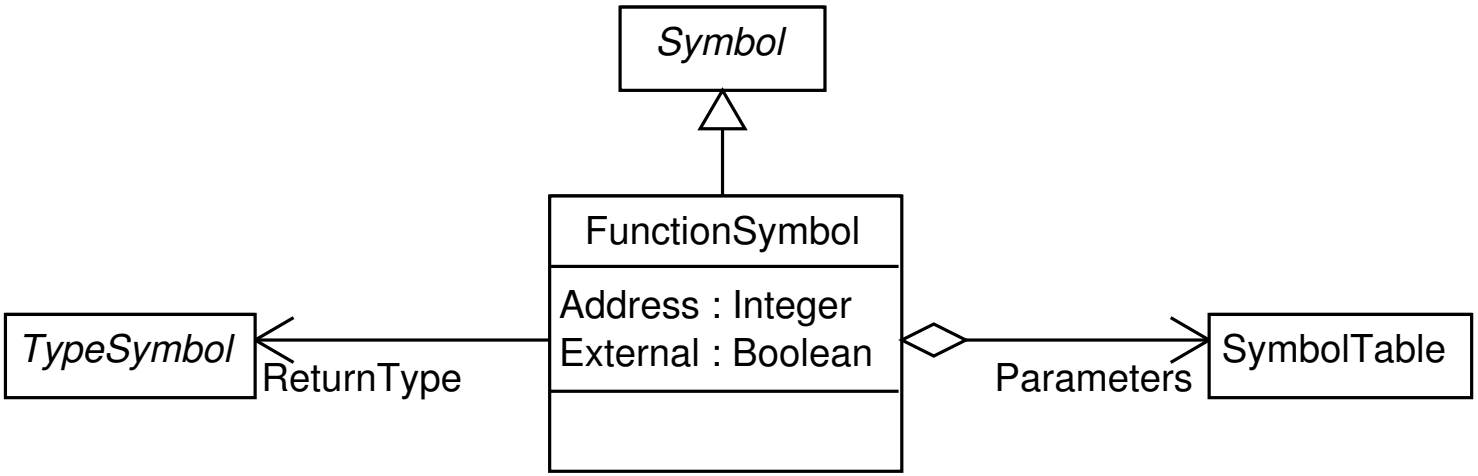
Свойство Address зарезервировано для адреса переменной и заполняется во время работы фазы распределения памяти.

От абстрактного класса TypeSymbol наследуют классы, представляющие примитивные и составные типы.



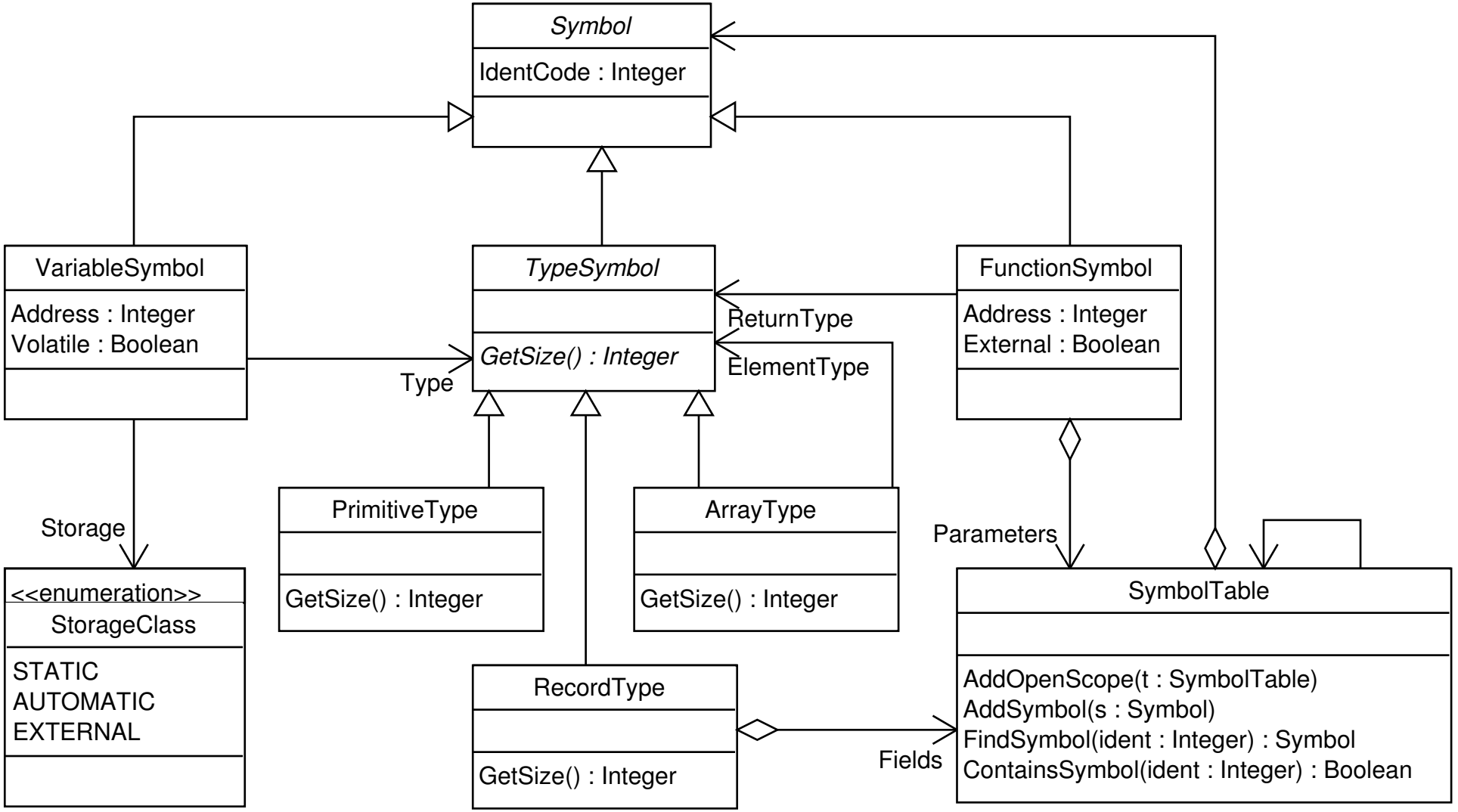
Предполагается, что у безымянных типов поле IdentCode содержит какое-нибудь специальное значение (например, -1).

Класс FunctionSymbol служит для представления функций.



Свойство Address зарезервировано для адреса функции и заполняется во время работы фазы генерации кода на целевом языке.

Сводная диаграмма классов:



§1. Математическая индукция

Исходные понятия арифметики:
единица (1), *натуральное число* и *следовать за* ($\prec$).

Аксиомы арифметики:

1. $(\forall n \in \mathbb{N})\,,\ (\exists_1\, n' \in \mathbb{N}) :\ n \prec n'$;

2. $1 \in \mathbb{N}$, причём $(\nexists n \in \mathbb{N}) :\ n \prec 1$;

3. $(\forall n, k, p \in \mathbb{N})$: $(k \prec n\ \wedge\ p \prec n) \Rightarrow (k \equiv p)$;

4. Если $1 \in A$ и $(\forall k \in \mathbb{N}) : (k \in A) \Rightarrow (k' \in A)$, то $\mathbb{N} \subseteq A$
   (аксиома математической индукции).

Утверждение. (*Метод математической индукции.*) Если высказывание $P(1)$ истинно, а из истинности $P(k)$ следует истинность высказывания $P(k+1)$, то $(\forall n \in \mathbb{N}) :\ P(n)$ – истинно.

Примечание. Предположение истинности $P(k)$, из которого выводится истинность $P(k+1)$, называется *гипотезой индукции*.

Доказательство. Пусть $A = \{n \in \mathbb{N} \mid P(n) -$ истинно$\}$.

Так как $P(1)$ истинно, то $1 \in A$.

Пусть некоторое $k \in A$, то есть $P(k)$ – истинно.

Тогда $P(k+1)$ тоже истинно, откуда следует, что $k' \equiv k+1 \in A$.

Согласно аксиоме математической индукции $\mathbb{N} \subseteq A$.

Так как $A \subseteq \mathbb{N}$, то $A \equiv \mathbb{N}$, и $P(n)$ истинно для любого $n \in \mathbb{N}$. $\square$

Определение. Говорят, что функция $u : \mathbb{N} \longrightarrow X$ задана индуктивно, если вычисление $u(n)$ определяется соотношениями:
1. $u\,(1) = x$;
2. $u\,(k) = F\,(u\,(k-1))$, где $k > 1$, $F : X \longrightarrow X$.

Легко доказать, что $(\forall k \in \mathbb{N}) : \exists_1 u\,(k)$.

Пример. Функция $\gamma : \mathbb{N} \longrightarrow \mathbb{Z}$ индуктивно определяет члены геометрической прогрессии с первым членом a и знаменателем q:
1. $\gamma(1) = a$;
2. $\gamma(k) = q \cdot \gamma(k-1)$.
Здесь $F(x) = q \cdot x$, $F : \mathbb{Z} \longrightarrow \mathbb{Z}$.

Пример. Функция $\varphi : \mathbb{N} \longrightarrow \mathbb{N}$ вычисляет числа Фибоначчи:
$\varphi\,(k) = n$, где $\langle n, m\rangle = f\,(k)$.
При этом функция $f : \mathbb{N} \longrightarrow \mathbb{N}^2$ задана индуктивно:
1. $f(1) = \langle 1, 1\rangle$;
2. $f(k) = \langle b, a+b\rangle$, где $\langle a, b\rangle = f(k-1)$.
Здесь $F(x, y) = \langle y, x+y\rangle$, $F : \mathbb{N}^2 \longrightarrow \mathbb{N}^2$.

§2. Абстрактный синтаксис

Определение. Мы будем называть *абстрактным синтаксисом* упрощённую грамматику языка, в которой отсутствует информация, гарантирующая построение уникальных деревьев вывода.

Определение. Пусть $G = \langle T, N, S, P \rangle$ – грамматика.
Мы будем называть *синтаксическим доменом*, соответствующим нетерминальному символу $x \in N$, множество деревьев вывода, полученных из x по правилам P.

Замечание. Деревья вывода – конечные. Они содержат терминальные символы в качестве листовых вершин.

Абстрактный синтаксис языка While:

$n \in$ Num – числовые константы;
$x \in$ Var – переменные;
$a \in$ Aexp – арифметические выражения;
$b \in$ Bexp – логические выражения;
$S \in$ Stm – операторы.

$n ::= \mathtt{0} \mid \mathtt{1} \mid \ldots \mid \mathtt{9} \mid n\,\mathtt{0} \mid n\,\mathtt{1} \mid \ldots \mid n\,\mathtt{9}$
$x ::= \mathtt{var}\,n$
$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \cdot a_2 \mid a_1 - a_2$
$b ::= \mathtt{true} \mid \mathtt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$
$S ::= x := a \mid \mathtt{skip} \mid S_1; S_2 \mid \mathtt{if}\,b\,\mathtt{then}\,S_1\,\mathtt{else}\,S_2 \mid \mathtt{while}\,b\,\mathtt{do}\,S$

Пример. Вычисление факториала для начального значения, связанного с переменной x_2.
$x_1 := \mathtt{1};\ \mathtt{while}\,\neg\,(x_2 = \mathtt{1})\ \mathtt{do}\,(x_1 := x_1 \cdot x_2;\ x_2 := x_2 - \mathtt{1})$.

§3. Структурная индукция

Утверждение. (*Метод структурной индукции.*) Если высказывание P истинно для деревьев нулевой глубины, а из истинности P для деревьев, глубина которых меньше k, следует истинность P для деревьев, глубина которых равна k, то P истинно для любого *конечного* дерева.

Доказательство. Воспользуемся методом м.и. по глубине дерева.

Случай $n = 1$ (*глубина дерева меньше 1*):

Имеем дерево, глубина которого равна нулю, для которого P истинно согласно условию.

Случай $n = k$ (P *истинно для всех деревьев глубиной меньше* k; *надо доказать, что* P *истинно для всех деревьев глубиной меньше* $k + 1$):

Из условия имеем истинность P для деревьев глубины k.

Из гипотезы индукции имеем истинность P для деревьев, глубина которых меньше k. $\square$

Понятие индуктивно заданной функции может быть расширено для структурной индукции.

Пример. Функция $\mathcal{N} : \mathsf{Num} \longrightarrow \mathbb{Z}$:

$$
\begin{array}{lcl}
\mathcal{N}[\![0]\!] & = & 0, \\
\mathcal{N}[\![1]\!] & = & 1, \\
\ldots & & \\
\mathcal{N}[\![9]\!] & = & 9, \\
\mathcal{N}[\![n\,0]\!] & = & 10 \cdot \mathcal{N}[\![n]\!], \\
\mathcal{N}[\![n\,1]\!] & = & 10 \cdot \mathcal{N}[\![n]\!] + 1, \\
\ldots & & \\
\mathcal{N}[\![n\,9]\!] & = & 10 \cdot \mathcal{N}[\![n]\!] + 9.
\end{array}
$$

Легко доказать, что $(\forall n) : \exists_1 \mathcal{N}\,(n)$.

§4. Системы переходов

Определение. *Система переходов* – это упорядоченная тройка

$$\langle \Gamma, T, \triangleright \rangle$$

где Γ – это множество *конфигураций*, $T \subseteq \Gamma$ – множество *терминальных* конфигураций, а $\triangleright \subseteq \Gamma \times \Gamma$ – *отношение переходов*.

Для отношения переходов должно выполняться
$(\forall \gamma \in T)\,(\nexists \gamma' \in \Gamma) : \gamma \triangleright \gamma'$.

Все конфигурации $\gamma \in \Gamma \setminus T$ такие, что $(\nexists \gamma' \in \Gamma) : \gamma \triangleright \gamma'$, называются *тупиковыми*.

Важные свойства систем переходов:

- *детерминизм* $((\forall \gamma \in \Gamma)\,(\exists_1\, \gamma') : \gamma \rhd \gamma')$;

- *достижимость* из определённых начальных конфигураций;

  – *безопасность* («плохие» конфигурации недостижимы);

- *правильная завершаемость* (все терминальные конфигурации – «хорошие»).

§5. Среды

Определение. *Среда* (environment) – это функция, отображающая множество переменных в множество их значений.

Пример. Множество Env сред для языка While состоит из функций вида $\mathrm{Var} \hookrightarrow \mathbb{Z}$, которые мы будем записывать в виде списка пар «переменная↦значение»:
$[x_1 \mapsto 2,\ x_2 \mapsto 3,\ x_3 \mapsto -10]$

Обозначение. Если $f : X \longrightarrow Y$, $x \in X$, $y \in Y$, то функция $f\,[x \mapsto y] : X \longrightarrow Y$ определяется как

$$f\,[x \mapsto y]\,(x') = \begin{cases} y, & \text{если } x = x'; \\ f\,(x') & \text{в противном случае.} \end{cases}$$

Пример. Если имеется среда σ, то в результате присвоения некоторого значения i переменной x получается среда $\sigma\,[x \mapsto i]$.

§6. Семантики и семантические функции

Определение. *Семантика* для синтаксического домена D – это кортеж

$\left\langle D, \Sigma, \Sigma_{start}, \Sigma_{final}, \Gamma, \triangleright \right\rangle$, где

Σ – множество состояний вычисления;
$\Sigma_{start} \subseteq \Sigma$ – множество входных состояний вычисления;
$\Sigma_{final} \subseteq \Sigma$ – множество выходных состояний вычисления;

$\left\langle \Gamma, \Sigma_{final}, \triangleright \right\rangle$ – система переходов, множество конфигураций Γ которой состоит из конфигураций вида:

$\langle d, \sigma \rangle$, означающей, что синтаксическая конструкция $d \in D$ должна быть выполнена для состояния $\sigma \in \Sigma$;

σ, представляющей одно из финальных состояний вычисления (это терминальная конфигурация, то есть $\sigma \in \Sigma_{final}$).

Пример. Семантика для арифметических выражений языка While:

$\langle$Aexp, Env $\cup\, \mathbb{Z}$, Env, $\mathbb{Z}$, Г, $\triangleright\rangle$, где

Aexp – синтаксический домен арифметических выражений;
Env $\cup\, \mathbb{Z}$ – множество состояний вычисления;
Env – множество начальных состояний вычисления;
$\mathbb{Z}$ – множество конечных состояний вычисления;
Г – множество конфигураций.

В случае естественной семантики («большие шаги») отношение переходов $\triangleright$ задаёт, например, такие переходы:
$\langle 6 \cdot (2 - x)\,, [x \mapsto 10]\rangle \triangleright -48$,
$\langle 10 \cdot 20, [x_1 \mapsto -1, x_2 \mapsto 4]\rangle \triangleright 200$.

Для редукционной семантики («малые шаги») отношение $\triangleright$ содержит переходы:
$\langle 6 \cdot (2 - x)\,, [x \mapsto 10]\rangle \triangleright \langle 6 \cdot (2 - 10)\,, [x \mapsto 10]\rangle$,
$\langle 10 \cdot 20, [x_1 \mapsto -1, x_2 \mapsto 4]\rangle \triangleright \langle 200, [x_1 \mapsto -1, x_2 \mapsto 4]\rangle$,
$\langle 200, [x_1 \mapsto -1, x_2 \mapsto 4]\rangle \triangleright 200$.

Определение. *Семантическая функция* $\mathcal{S} : D \longrightarrow \left(\Sigma_{start} \hookrightarrow \Sigma_{final}\right)$, выражающая *наблюдаемое поведение* детерминированной семантики $\left\langle D, \Sigma, \Sigma_{start}, \Sigma_{final}, \Gamma, \triangleright \right\rangle$, определяется как

$$\mathcal{S}[\![d]\!](\sigma) = \begin{cases} \sigma', & \text{если } \langle d, \sigma \rangle \triangleright^{\star} \sigma'; \\ \mathbf{undef} & \text{в противном случае.} \end{cases}$$

Определение. Говорят, что детерминированные семантики $\left\langle D, \Sigma, \Sigma_{start}, \Sigma_{final}, \Gamma, \triangleright_1 \right\rangle$ и $\left\langle D, \Sigma, \Sigma_{start}, \Sigma_{final}, \Gamma, \triangleright_2 \right\rangle$ эквивалентны,

если $(\forall d \in D)\,(\forall \sigma \in \Sigma_{start}) : (\mathcal{S}_1[\![d]\!](\sigma) = \sigma') \Leftrightarrow (\mathcal{S}_2[\![d]\!](\sigma) = \sigma')$,

где $\mathcal{S}_1$ и $\mathcal{S}_2$ – семантические функции, выражающие наблюдаемое поведение этих семантик.

Пример. Семантическая функция $\mathcal{A} : \mathsf{Aexp} \longrightarrow (\mathsf{Env} \longrightarrow \mathbb{Z})$ для арифметических выражений языка While (задаётся индуктивно):

$$\begin{array}{lcl}
\mathcal{A}[\![n]\!](\sigma) & = & \mathcal{N}(n), \\
\mathcal{A}[\![x]\!](\sigma) & = & \sigma(x), \\
\mathcal{A}[\![a_1 + a_2]\!](\sigma) & = & \mathcal{A}[\![a_1]\!](\sigma) + \mathcal{A}[\![a_2]\!](\sigma), \\
\mathcal{A}[\![a_1 \cdot a_2]\!](\sigma) & = & \mathcal{A}[\![a_1]\!](\sigma) \cdot \mathcal{A}[\![a_2]\!](\sigma), \\
\mathcal{A}[\![a_1 - a_2]\!](\sigma) & = & \mathcal{A}[\![a_1]\!](\sigma) - \mathcal{A}[\![a_2]\!](\sigma).
\end{array}$$

Пример. Если добавить в язык операцию «унарный минус», то определение функции $\mathcal{A}$ нужно расширить предложением:

$$\mathcal{A}[\![-a]\!](\sigma) = -\mathcal{A}[\![a]\!](\sigma).$$

Альтернативный способ, противоречащий индуктивному заданию функции:

$$\mathcal{A}[\![-a]\!](\sigma) = \mathcal{A}[\![0 - a]\!](\sigma).$$

Пример. Семантическая функция $\mathcal{B} : \mathrm{Bexp} \longrightarrow (\mathrm{Env} \longrightarrow \mathsf{T})$ для логических выражений языка While (задаётся индуктивно):

$$\mathcal{B}[\![\mathtt{true}]\!](\sigma) = \mathbf{tt},$$

$$\mathcal{B}[\![\mathtt{false}]\!](\sigma) = \mathbf{ff},$$

$$\mathcal{B}[\![a_1 = a_2]\!](\sigma) = \begin{cases} \mathbf{tt}, & \text{если } \mathcal{A}[\![a_1]\!](\sigma) = \mathcal{A}[\![a_2]\!](\sigma), \\ \mathbf{ff}, & \text{если } \mathcal{A}[\![a_1]\!](\sigma) \neq \mathcal{A}[\![a_2]\!](\sigma), \end{cases}$$

$$\mathcal{B}[\![a_1 \leq a_2]\!](\sigma) = \begin{cases} \mathbf{tt}, & \text{если } \mathcal{A}[\![a_1]\!](\sigma) \leq \mathcal{A}[\![a_2]\!](\sigma), \\ \mathbf{ff}, & \text{если } \mathcal{A}[\![a_1]\!](\sigma) > \mathcal{A}[\![a_2]\!](\sigma), \end{cases}$$

$$\mathcal{B}[\![\neg b]\!](\sigma) = \begin{cases} \mathbf{tt}, & \text{если } \mathcal{B}[\![b]\!](\sigma) = \mathbf{ff}, \\ \mathbf{ff}, & \text{если } \mathcal{B}[\![b]\!](\sigma) = \mathbf{tt}, \end{cases}$$

$$\mathcal{B}[\![b_1 \wedge b_2]\!](\sigma) = \begin{cases} \mathbf{tt}, & \text{если } \mathcal{B}[\![b_1]\!](\sigma) = \mathbf{tt} \text{ и } \mathcal{B}[\![b_2]\!](\sigma) = \mathbf{tt}, \\ \mathbf{ff}, & \text{если } \mathcal{B}[\![b_1]\!](\sigma) = \mathbf{ff} \text{ или } \mathcal{B}[\![b_2]\!](\sigma) = \mathbf{ff}. \end{cases}$$

§7. Естественная семантика

В естественной семантике все переходы имеют вид $\langle d, \sigma \rangle \rhd \sigma'$. Это означает, что выполнение синтаксической конструкции d из состояния вычисления σ завершается, и результирующим состоянием будет σ'.

Определение. Правило естественной семантики $\left\langle D, \Sigma, \Sigma_{start}, \Sigma_{final}, \Gamma, \to \right\rangle$ записывается в виде

$$\frac{\left\langle \widehat{d_1},\, \widehat{\sigma_1} \right\rangle \to \widehat{\sigma'_1}, \cdots, \left\langle \widehat{d_n},\, \widehat{\sigma_n} \right\rangle \to \widehat{\sigma'_n}}{\left\langle \widehat{d},\, \widehat{\sigma} \right\rangle \to \widehat{\sigma'}}, \text{ если } \ldots$$

Здесь $\widehat{d}, \widehat{d_i} \subseteq D$, $\widehat{\sigma}, \widehat{\sigma_i} \subseteq \Sigma_{start}$, $\widehat{\sigma'_i} \subseteq \Sigma_{final}$ – обобщённые записи подмножеств множеств D и Σ (образцы с метапеременными).

При этом $\widehat{d_1}, \cdots, \widehat{d_n}$ – это либо непосредственные поддеревья $\widehat{d}$, либо деревья, сконструированные из непосредственных поддеревьев $\widehat{d}$.

Над чертой – *предпосылки*, под чертой – *следствия*, справа – *условия*, правила без предпосылок – *аксиомы* (записываются без гор. черты).

Пример. Естественная семантика для операторов языка While.

$[\text{assign}_{ns}]$ $\quad \langle x := a,\ \sigma\rangle \to \sigma\,[x \mapsto \mathcal{A}[\![a]\!](\sigma)]$

$[\text{skip}_{ns}]$ $\quad \langle \texttt{skip},\ \sigma\rangle \to \sigma$

$[\text{comp}_{ns}]$ $\quad \dfrac{\langle S_1,\ \sigma\rangle \to \sigma', \quad \langle S_2,\ \sigma'\rangle \to \sigma''}{\langle S_1; S_2,\ \sigma\rangle \to \sigma''}$

$\left[\text{if}^{\mathbf{tt}}_{ns}\right]$ $\quad \dfrac{\langle S_1,\ \sigma\rangle \to \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \to \sigma'}$, если $\mathcal{B}[\![b]\!](\sigma) = \mathbf{tt}$

$\left[\text{if}^{\mathbf{ff}}_{ns}\right]$ $\quad \dfrac{\langle S_2,\ \sigma\rangle \to \sigma'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \to \sigma'}$, если $\mathcal{B}[\![b]\!](\sigma) = \mathbf{ff}$

$\left[\text{while}^{\mathbf{tt}}_{ns}\right]$ $\quad \dfrac{\langle S,\ \sigma\rangle \to \sigma', \quad \langle \texttt{while}\ b\ \texttt{do}\ S,\ \sigma'\rangle \to \sigma''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ \sigma\rangle \to \sigma''}$, если $\mathcal{B}[\![b]\!](\sigma) = \mathbf{tt}$

$\left[\text{while}^{\mathbf{ff}}_{ns}\right]$ $\quad \langle \texttt{while}\ b\ \texttt{do}\ S,\ \sigma\rangle \to \sigma$, если $\mathcal{B}[\![b]\!](\sigma) = \mathbf{ff}$

Каждое правило семантики является схемой для порождения некоторого множества переходов.

Определение. Мы будем называть переходы, порождаемые правилами семантики, *экземплярами* этих правил.

Пример. Одним из экземпляров аксиомы [assign$_{ns}$] является переход $\langle x_1 := 1,\ [x_1 \mapsto 0,\ x_2 \mapsto 0]\rangle \to [x_1 \mapsto 1,\ x_2 \mapsto 0]$

Пример. Одним из экземпляров правила [comp$_{ns}$] является переход $\langle x_1 := 5;\ x_2 := 3,\ [x_1 \mapsto 4]\rangle \to [x_1 \mapsto 5,\ x_2 \mapsto 3]$

Вывод перехода $\langle d, \sigma\rangle \to \sigma'$ в некоторой семантике связан с построением *дерева вывода*, корнем которого является выводимый переход, листьями – экземпляры аксиом семантики.

Пример. Дерево вывода для перехода

$$\langle (x_3 := x_2;\ x_2 := x_1)\,;\ x_1 := x_3,\ \sigma_1\rangle \to \sigma_4$$

записывается так:

$$\dfrac{\dfrac{\langle x_3 := x_2,\ \sigma_1\rangle \to \sigma_2, \quad \langle x_2 := x_1,\ \sigma_2\rangle \to \sigma_3}{\langle x_3 := x_2;\ x_2 := x_1,\ \sigma_1\rangle \to \sigma_3}, \quad \langle x_1 := x_3,\ \sigma_3\rangle \to \sigma_4}{\langle (x_3 := x_2;\ x_2 := x_1)\,;\ x_1 := x_3,\ \sigma_1\rangle \to \sigma_4},$$ где

$\sigma_1 = [x_1 \mapsto 4,\ x_2 \mapsto 5]$,
$\sigma_2 = [x_1 \mapsto 4,\ x_2 \mapsto 5,\ x_3 \mapsto 5]$,
$\sigma_3 = [x_1 \mapsto 4,\ x_2 \mapsto 4,\ x_3 \mapsto 5]$,
$\sigma_4 = [x_1 \mapsto 5,\ x_2 \mapsto 4,\ x_3 \mapsto 5]$.

Пусть требуется построить дерево вывода для некоторой синтаксической конструкции d из состояния вычисления σ. Для этого требуется найти правило семантики, левую часть следствия которого можно отождествить с конфигурацией $\langle d, \sigma \rangle$. При этом возможны два случая:

1. Если найденное правило является аксиомой, и условия этой аксиомы выполняются, то мы можем сразу же определить выходное состояние вычисления. Тем самым построение дерева вывода завершается.

2. Если найденное правило содержит предпосылки, то мы пытаемся построить деревья вывода для каждой предпосылки. В случае успешного построения этих деревьев мы обязаны проверить условия, связанные с правилом, и, если эти условия выполняются, мы можем определить выходное состояние вычисления.

Пример. Покажем процесс построения дерева вывода T для оператора

$$\texttt{while}\,\neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)$$

из состояния вычисления $\sigma_1 = [x_1 \mapsto 2,\ x_2 \mapsto 2]$.

1. Согласно правилу $\left[\text{while}^{\mathbf{tt}}_{ns}\right]$:

$$T = \frac{T_1, \quad T_2}{\langle \texttt{while}\,\neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\ \sigma_1\rangle \to \sigma_6}, \text{ где}$$

$$T_1 = \frac{\ldots}{\langle x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1,\ \sigma_1\rangle \to \sigma_3},$$

$$T_2 = \frac{\ldots}{\langle \texttt{while}\,\neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\ \sigma_3\rangle \to \sigma_6}.$$

2. Согласно правилу [comp$_{ns}$]:

$$T_1 = \frac{T_3, \quad T_4}{\langle x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1,\ \sigma_1\rangle \to \sigma_3}, \text{ где}$$

$T_3 = \langle x_2 := x_2 \cdot 2,\ \sigma_1\rangle \to \sigma_2$ (аксиома [assign$_{ns}$]),
$T_4 = \langle x_1 := x_1 - 1,\ \sigma_2\rangle \to \sigma_3$ (аксиома [assign$_{ns}$]).

Учитывая, что $\sigma_1 = [x_1 \mapsto 2,\ x_2 \mapsto 2]$, получаем
$\sigma_2 = \sigma_1\,[x_2 \mapsto 4] = [x_1 \mapsto 2,\ x_2 \mapsto 4]$,
$\sigma_3 = \sigma_2\,[x_1 \mapsto 1] = [x_1 \mapsto 1,\ x_2 \mapsto 4]$.

3. Согласно правилу $\left[\text{while}^{\mathbf{tt}}_{ns}\right]$:

$$T_2 = \frac{T_5, \quad T_6}{\langle \texttt{while}\, \neg\, (x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\ \sigma_3 \rangle \to \sigma_6}, \text{ где}$$

$$T_5 = \frac{\ldots}{\langle x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1,\ \sigma_3 \rangle \to \sigma_5},$$

$$T_6 = \frac{\ldots}{\langle \texttt{while}\, \neg\, (x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\ \sigma_5 \rangle \to \sigma_6}.$$

4. Согласно правилу [comp$_{ns}$]:

$$T_5 = \frac{T_7, \quad T_8}{\langle x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1,\ \sigma_3\rangle \to \sigma_5}, \text{ где}$$

$T_7 = \langle x_2 := x_2 \cdot 2,\ \sigma_3\rangle \to \sigma_4$ (аксиома [assign$_{ns}$]),
$T_8 = \langle x_1 := x_1 - 1,\ \sigma_4\rangle \to \sigma_5$ (аксиома [assign$_{ns}$]).

Учитывая, что $\sigma_3 = [x_1 \mapsto 1,\ x_2 \mapsto 4]$, получаем
$\sigma_4 = \sigma_3\,[x_2 \mapsto 8] = [x_1 \mapsto 1,\ x_2 \mapsto 8]$,
$\sigma_5 = \sigma_4\,[x_1 \mapsto 0] = [x_1 \mapsto 0,\ x_2 \mapsto 8]$.

5. Согласно аксиоме $\left[\text{while}^{\mathbf{ff}}_{ns}\right]$:

$T_6 = \langle \texttt{while}\ \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\ \sigma_5\rangle \to \sigma_6$, где

$\sigma_6 = \sigma_5 = [x_1 \mapsto 0,\ x_2 \mapsto 8]$.

§8. Проведение доказательств по форме дерева вывода

Если отношение переходов семантики не задано индуктивно, то структурная индукция по деревьям абстрактного синтаксиса невозможна, и приходится проводить доказательства по форме дерева вывода.

Пример. Естественная семантика языка While содержит неиндуктивное правило $\left[\text{while}^{\mathbf{tt}}_{ns}\right]$, поэтому доказательство свойств семантики языка While с помощью структурной индукции по деревьям абстрактного синтаксиса невозможно.

Схема доказательства по форме дерева вывода:

1. Необходимо доказать, что некое свойство семантики выполняется для всех элементарных деревьев вывода, то есть для аксиом семантики.

2. Для каждого правила, не являющегося аксиомой, предполагаем, что нужное свойство выполняется для всех его предпосылок. Базируясь на этом предположении, доказываем свойство для следствия.

Пример. Докажем, что естественная семантика языка While является детерминированной.

Доказательство. Предполагая $\langle S, \sigma \rangle \to \sigma'$, мы докажем, что из $\langle S, \sigma \rangle \to \sigma''$ следует $\sigma' = \sigma''$.

Для доказательства используем индукцию по форме дерева вывода для $\langle S, \sigma \rangle \to \sigma'$.

Случай [**assign**$_{ns}$]:
$S = x := a$ и $\sigma' = \sigma\,[x \mapsto \mathcal{A}[\![a]\!]\,(\sigma)]$.
Единственное правило, которое может быть применено для вывода $\langle S, \sigma \rangle \to \sigma''$, – это [assign$_{ns}$]. Поэтому $\sigma'' = \sigma\,[x \mapsto \mathcal{A}[\![a]\!]\,(\sigma)] = \sigma'$.

Случай [**skip**$_{ns}$]:
аналогично.

Случай [**comp**$_{ns}$]:
Предположим, что $\langle S_1; S_2, \sigma\rangle \to \sigma'$.
Значит $\exists \sigma_0 : \langle S_1, \sigma\rangle \to \sigma_0, \quad \langle S_2, \sigma_0\rangle \to \sigma'$.

Единственное правило, которое может быть применено для вывода $\langle S_1; S_2, \sigma\rangle \to \sigma''$, – это [comp$_{ns}$].
Отсюда $\exists \sigma_1 : \langle S_1, \sigma\rangle \to \sigma_1, \quad \langle S_2, \sigma_1\rangle \to \sigma''$.

Применяя гипотезу индукции к предпосылке $\langle S_1, \sigma\rangle \to \sigma_0$, получаем, что $\langle S_1, \sigma\rangle \to \sigma_1 \Rightarrow \sigma_0 = \sigma_1$.
Применяя гипотезу индукции к предпосылке $\langle S_2, \sigma_0\rangle \to \sigma'$, получаем, что $\langle S_2, \sigma_0\rangle \to \sigma'' \Rightarrow \sigma' = \sigma''$.

Случай $\left[\mathbf{if}^{\mathbf{tt}}_{ns}\right]$:
Предположим, что $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \to \sigma'$.
То есть $\mathcal{B}[\![b]\!]\,(\sigma) = \mathbf{tt}$ и $\langle S_1,\ \sigma\rangle \to \sigma'$.

Так как $\mathcal{B}[\![b]\!]\,(\sigma) = \mathbf{tt}$, то единственное правило, которое может быть применено для вывода $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \to \sigma''$ – это $\left[\mathbf{if}^{\mathbf{tt}}_{ns}\right]$.
Отсюда $\langle S_1,\ \sigma\rangle \to \sigma''$.

Применяя гипотезу индукции к предпосылке $\langle S_1,\ \sigma\rangle \to \sigma'$, получаем, что $\langle S_1,\ \sigma\rangle \to \sigma'' \Rightarrow \sigma' = \sigma''$.

Случай $\left[\mathbf{if}^{\mathbf{ff}}_{ns}\right]$:
аналогично.

Случай $\left[\textbf{while}^{\mathbf{tt}}_{ns}\right]$:
Предположим, что $\langle \texttt{while}\, b\, \texttt{do}\, S,\, \sigma\rangle \to \sigma'$.
То есть $\mathcal{B}[\![b]\!]\,(\sigma) = \mathbf{tt}$ и $\exists \sigma_0 : \langle S,\, \sigma\rangle \to \sigma_0,\, \langle \texttt{while}\, b\, \texttt{do}\, S,\, \sigma_0\rangle \to \sigma'$.

Так как $\mathcal{B}[\![b]\!]\,(\sigma) = \mathbf{tt}$, то единственное правило, которое может быть применено для вывода $\langle \texttt{while}\, b\, \texttt{do}\, S,\, \sigma\rangle \to \sigma''$ – это $\left[\textbf{while}^{\mathbf{tt}}_{ns}\right]$.
Отсюда $\exists \sigma_1 : \langle S,\, \sigma\rangle \to \sigma_1,\, \langle \texttt{while}\, b\, \texttt{do}\, S,\, \sigma_1\rangle \to \sigma''$.

Применяя гипотезу индукции к предпосылке $\langle S,\, \sigma\rangle \to \sigma_0$, получаем, что $\langle S,\, \sigma\rangle \to \sigma_1 \Rightarrow \sigma_0 = \sigma_1$.
Применяя гипотезу индукции к предпосылке $\langle \texttt{while}\, b\, \texttt{do}\, S,\, \sigma_0\rangle \to \sigma'$, получаем, что $\langle \texttt{while}\, b\, \texttt{do}\, S,\, \sigma_0\rangle \to \sigma'' \Rightarrow \sigma' = \sigma''$.

Случай $\left[\textbf{while}^{\mathbf{ff}}_{ns}\right]$:
элементарно.$\square$

§9. Редукционная семантика

В редукционной семантике используются два вида переходов:

$\langle d, \sigma \rangle \triangleright \langle d', \sigma' \rangle$ – частичное выполнение синтаксической конструкции d из состояния вычисления σ (оставшаяся часть вычислений выражается промежуточной конфигурацией $\langle d', \sigma' \rangle$);

$\langle d, \sigma \rangle \triangleright \sigma'$ – выполнение синтаксической конструкции d из состояния вычисления σ завершается, и результирующим состоянием становится σ'.

Правила редукционной семантики записываются в том же виде, что и правила естественной семантики.

Из естественной семантики в редукционную также переходит понятие *дерева вывода*.

Пример. Редукционная семантика для операторов языка While.

$[\text{assign}_{rs}]$ $\quad \langle x := a,\ \sigma\rangle \Rrightarrow \sigma\,[x \mapsto \mathcal{A}[\![a]\!](\sigma)]$

$[\text{skip}_{rs}]$ $\quad \langle \texttt{skip},\ \sigma\rangle \Rrightarrow \sigma$

$$\left[\text{comp}^1_{rs}\right] \quad \frac{\langle S_1,\ \sigma\rangle \Rrightarrow \left\langle S_1',\ \sigma'\right\rangle}{\langle S_1; S_2,\ \sigma\rangle \Rrightarrow \left\langle S_1'; S_2,\ \sigma'\right\rangle}$$

$$\left[\text{comp}^2_{rs}\right] \quad \frac{\langle S_1,\ \sigma\rangle \Rrightarrow \sigma'}{\langle S_1; S_2,\ \sigma\rangle \Rrightarrow \langle S_2,\ \sigma'\rangle}$$

$\left[\text{if}^{\mathbf{tt}}_{rs}\right]$ $\quad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \Rrightarrow \langle S_1,\ \sigma\rangle$, если $\mathcal{B}[\![b]\!](\sigma) = \mathbf{tt}$

$\left[\text{if}^{\mathbf{ff}}_{rs}\right]$ $\quad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \Rrightarrow \langle S_2,\ \sigma\rangle$, если $\mathcal{B}[\![b]\!](\sigma) = \mathbf{ff}$

$[\text{while}_{rs}]$ $\quad \langle \texttt{while}\ b\ \texttt{do}\ S,\ \sigma\rangle \Rrightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\ \sigma\rangle$

Пример. Дерево вывода для перехода

$$\langle (x_3 := x_2;\ x_2 := x_1)\,;\ x_1 := x_3,\ \sigma\rangle \Rrightarrow \langle x_2 := x_1;\ x_1 := x_3,\ \sigma'\rangle$$

записывается так:

$$\dfrac{\dfrac{\langle x_3 := x_2,\ \sigma\rangle \Rrightarrow \sigma'}{\langle x_3 := x_2;\ x_2 := x_1,\ \sigma\rangle \Rrightarrow \langle x_2 := x_1,\ \sigma'\rangle}}{\langle (x_3 := x_2;\ x_2 := x_1)\,;\ x_1 := x_3,\ \sigma\rangle \Rrightarrow \langle x_2 := x_1;\ x_1 := x_3,\ \sigma'\rangle}$$, где

$\sigma = [x_1 \mapsto 4,\ x_2 \mapsto 5]$,
$\sigma' = [x_1 \mapsto 4,\ x_2 \mapsto 5,\ x_3 \mapsto 5]$.

Определение. *Последовательность вывода* для синтаксической конструкции d из состояния σ – это либо конечная последовательность конфигураций $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k$, либо бесконечная последовательность конфигураций $\gamma_0, \gamma_1, \gamma_2, \ldots$.
При этом $\gamma_0 = \langle d, \sigma \rangle$, $\gamma_i \Rrightarrow \gamma_{i+1}$ и γ_k – это либо терминальная, либо тупиковая конфигурация.

Обозначение. Мы будем использовать запись $\gamma_0 \Rrightarrow^i \gamma_i$, чтобы показать, что γ_0 переходит в γ_i за i шагов. Запись $\gamma_0 \Rrightarrow^\star \gamma_i$ будет означать, что между γ_0 и γ_i – конечное число шагов. При этом надо понимать, что $\gamma_0 \Rrightarrow^i \gamma_i$ и $\gamma_0 \Rrightarrow^\star \gamma_i$ являются последовательностями вывода только в том случае, если γ_i – терминальная или тупиковая конфигурация.

Определение. Мы будем говорить, что выполнение синтаксической конструкции d из состояния вычисления σ *завершается*, если существует конечная последовательность вывода, начинающаяся с $\langle d, \sigma \rangle$.
При этом выполнение *успешно*, если $\langle d, \sigma \rangle \Rrightarrow^{\star} \sigma'$.

Определение. Мы будем говорить, что выполнение синтаксической конструкции d из состояния вычисления σ *зацикливается*, если существует бесконечная последовательность вывода, начинающаяся с $\langle d, \sigma \rangle$.

Пример. Покажем процесс построения последовательности вывода для оператора

$$\texttt{while}\,\neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)$$

из состояния вычисления $\sigma_1 = [x_1 \mapsto 2,\ x_2 \mapsto 2]$.

1. Согласно правилу [while$_{rs}$]:

$$\begin{array}{l}\langle \texttt{while}\,\neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\ \sigma_1\rangle \Rrightarrow \\ \quad \langle \texttt{if}\,\neg\,(x_1 = 0)\ \texttt{then}\,(\\ \qquad (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,; \\ \qquad \texttt{while}\,\neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1) \\ \qquad) \\ \quad \texttt{else} \\ \qquad \texttt{skip}, \\ \quad \sigma_1\rangle\end{array}$$

2. Согласно правилу $\left[\mathsf{if}^{\mathbf{tt}}_{rs}\right]$:

$$
\begin{array}{l}
\langle \texttt{if}\ \neg\,(x_1 = 0)\ \texttt{then}\,(\\
\quad (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,; \\
\quad \texttt{while}\ \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1) \\
\quad) \\
\texttt{else} \\
\quad \texttt{skip}, \\
\sigma_1\rangle \Rrightarrow \\
\quad \langle (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,; \\
\quad \texttt{while}\ \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,, \\
\quad \sigma_1\rangle
\end{array}
$$

3. Согласно правилу $\left[\mathrm{comp}^1_{rs}\right]$:

$$\dfrac{\dfrac{\langle x_2 := x_2 \cdot 2,\, \sigma_1\rangle \Rrightarrow \sigma_2}{\langle x_2 := x_2 \cdot 2;\, x_1 := x_1 - 1,\, \sigma_1\rangle \Rrightarrow \langle x_1 := x_1 - 1,\, \sigma_2\rangle}}{\begin{array}{l}\langle (x_2 := x_2 \cdot 2;\, x_1 := x_1 - 1)\,; \\ \ \texttt{while}\, \neg (x_1 = 0)\, \texttt{do}\, (x_2 := x_2 \cdot 2;\, x_1 := x_1 - 1)\,, \\ \ \sigma_1\rangle \Rrightarrow \\ \quad \langle x_1 := x_1 - 1; \\ \quad\ \texttt{while}\, \neg (x_1 = 0)\, \texttt{do}\, (x_2 := x_2 \cdot 2;\, x_1 := x_1 - 1)\,, \\ \quad\ \sigma_2\rangle\end{array}}$$

Учитывая, что $\sigma_1 = [x_1 \mapsto 2,\, x_2 \mapsto 2]$, получаем $\sigma_2 = \sigma_1\,[x_2 \mapsto 4] = [x_1 \mapsto 2,\, x_2 \mapsto 4]$.

4. Согласно правилу $\left[\mathsf{comp}^1_{rs}\right]$:

$$\frac{\langle x_1 := x_1 - 1,\, \sigma_2\rangle \Rrightarrow \sigma_3}{\begin{array}{l}\langle x_1 := x_1 - 1;\\ \ \mathtt{while}\, \neg (x_1 = 0)\ \mathtt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\\ \ \sigma_2\rangle \Rrightarrow\\ \quad \langle \mathtt{while}\, \neg (x_1 = 0)\ \mathtt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\, \sigma_3\rangle\end{array}}$$

Учитывая, что $\sigma_2 = [x_1 \mapsto 2,\, x_2 \mapsto 4]$, получаем $\sigma_3 = \sigma_2\,[x_1 \mapsto 1] = [x_1 \mapsto 1,\, x_2 \mapsto 4]$.

5. Согласно правилу [while$_{rs}$]:

$$\begin{array}{l}\langle \texttt{while}\ \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\ \sigma_3\rangle \Rrightarrow \\ \quad \langle \texttt{if}\ \neg\,(x_1 = 0)\ \texttt{then}\,(\\ \qquad (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,; \\ \qquad \texttt{while}\ \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1) \\ \qquad) \\ \quad \texttt{else} \\ \qquad \texttt{skip}, \\ \quad \sigma_3\rangle\end{array}$$

6. Согласно правилу $\left[\mathsf{if}^{\mathbf{tt}}_{rs}\right]$:

$$
\begin{array}{l}
\langle \texttt{if} \neg (x_1 = 0) \texttt{ then} (\\
\quad (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,; \\
\quad \texttt{while} \neg (x_1 = 0) \texttt{ do } (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1) \\
\quad) \\
\texttt{else} \\
\quad \texttt{skip}, \\
\sigma_3 \rangle \Rrightarrow \\
\quad \langle (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,; \\
\quad \texttt{while} \neg (x_1 = 0) \texttt{ do } (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,, \\
\quad \sigma_3 \rangle
\end{array}
$$

7. Согласно правилу $\left[\mathrm{comp}^1_{rs}\right]$:

$$\dfrac{\dfrac{\langle x_2 := x_2 \cdot 2,\, \sigma_3\rangle \Rrightarrow \sigma_4}{\langle x_2 := x_2 \cdot 2;\, x_1 := x_1 - 1,\, \sigma_3\rangle \Rrightarrow \langle x_1 := x_1 - 1,\, \sigma_4\rangle}}{\begin{array}{l}\langle (x_2 := x_2 \cdot 2;\, x_1 := x_1 - 1)\,; \\ \texttt{while}\, \neg (x_1 = 0)\, \texttt{do}\, (x_2 := x_2 \cdot 2;\, x_1 := x_1 - 1)\,, \\ \sigma_3\rangle \Rrightarrow \\ \quad \langle x_1 := x_1 - 1; \\ \quad \texttt{while}\, \neg (x_1 = 0)\, \texttt{do}\, (x_2 := x_2 \cdot 2;\, x_1 := x_1 - 1)\,, \\ \quad \sigma_4\rangle\end{array}}$$

Учитывая, что $\sigma_3 = [x_1 \mapsto 1,\, x_2 \mapsto 4]$, получаем $\sigma_4 = \sigma_3\,[x_2 \mapsto 8] = [x_1 \mapsto 1,\, x_2 \mapsto 8]$.

8. Согласно правилу $\left[\mathsf{comp}^1_{rs}\right]$:

$$\frac{\langle x_1 := x_1 - 1,\, \sigma_4\rangle \Rrightarrow \sigma_5}{\begin{array}{l}\langle x_1 := x_1 - 1;\\ \ \texttt{while}\, \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\\ \ \sigma_4\rangle \Rrightarrow\\ \quad \langle \texttt{while}\, \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\, \sigma_5\rangle\end{array}}$$

Учитывая, что $\sigma_4 = [x_1 \mapsto 1,\, x_2 \mapsto 8]$, получаем $\sigma_5 = \sigma_4\,[x_1 \mapsto 0] = [x_1 \mapsto 0,\, x_2 \mapsto 8]$.

9. Согласно правилу [while$_{rs}$]:

$$
\begin{array}{l}
\langle \texttt{while}\ \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,,\ \sigma_5\rangle \Rrightarrow \\
\quad \langle \texttt{if}\ \neg\,(x_1 = 0)\ \texttt{then}\ (\\
\qquad (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,; \\
\qquad \texttt{while}\ \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1) \\
\qquad) \\
\quad \texttt{else} \\
\qquad \texttt{skip}, \\
\quad \sigma_5\rangle
\end{array}
$$

10. Согласно правилу $\left[\mathsf{if}^{\mathbf{ff}}_{rs}\right]$:

$$\begin{array}{l}\langle \texttt{if}\ \neg\,(x_1 = 0)\ \texttt{then}\,(\\ \quad (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\,;\\ \quad \texttt{while}\ \neg\,(x_1 = 0)\ \texttt{do}\ (x_2 := x_2 \cdot 2;\ x_1 := x_1 - 1)\\ \quad)\\ \texttt{else}\\ \quad \texttt{skip},\\ \sigma_5\rangle \Rrightarrow\\ \quad \langle \texttt{skip},\ \sigma_5\rangle\end{array}$$

11. И, наконец, по правилу $[\mathsf{skip}_{rs}]$:

$\langle \texttt{skip},\ \sigma_5\rangle \Rrightarrow \sigma_5$.

§10. Проведение доказательств по длине последовательности вывода

Доказательство свойств редукционных семантик удобно проводить индукцией по длине последовательности вывода.

Схема доказательства по длине последовательности вывода:

1. Необходимо доказать, что нужное свойство выполняется для всех последовательностей вывода нулевой длины.

2. На базе предположения, что нужное свойство выполняется для всех последовательностей вывода, длина которых не превышает k, доказываем, что оно также верно и для последовательностей длины $k+1$.

Пример. Докажем для семантики языка While, что если $\langle S_1; S_2, \sigma\rangle \Rrightarrow^k \sigma''$, то существует состояние σ' и натуральные числа k_1 и k_2 такие, что $\langle S_1, \sigma\rangle \Rrightarrow^{k_1} \sigma'$ и $\langle S_2, \sigma'\rangle \Rrightarrow^{k_2} \sigma''$, причём $k_1 + k_2 = k$.

Доказательство. Проведём доказательство индукцией по числу k, то есть индукцией по длине последовательности вывода $\langle S_1; S_2, \sigma\rangle \Rrightarrow^k \sigma''$.
Случай $k = 0$ (*последовательности нулевой длины*):
Существование последовательности $\langle S_1; S_2, \sigma\rangle \Rrightarrow^0 \sigma''$ означало бы, что $\langle S_1; S_2, \sigma\rangle \equiv \sigma''$. Это невозможно, поэтому доказываемое свойство автоматически верно для $k = 0$.

Случай $k = k_0$ (*доказываемое свойство истинно для всех последовательностей, длина которых меньше или равна* k_0; *надо доказать, что* оно *истинно для всех последовательностей длины* $k_0 + 1$):
Пусть
$\langle S_1; S_2, \sigma\rangle \Rrightarrow^{k_0+1} \sigma''$.
Мы можем записать эту последовательность вывода в виде
$\langle S_1; S_2, \sigma\rangle \Rrightarrow \gamma \Rrightarrow^{k_0} \sigma''$.

Конфигурация γ может быть получена двумя способами:
по правилу $\left[\text{comp}^1_{rs}\right]$ или по правилу $\left[\text{comp}^2_{rs}\right]$.

1. В первом случае $\langle S_1; S_2, \sigma\rangle \Rrightarrow \left\langle S_1'; S_2, \sigma_0\right\rangle$, так как $\langle S_1, \sigma\rangle \Rrightarrow \left\langle S_1', \sigma_0\right\rangle$.

Тем самым мы имеем последовательность вывода $\left\langle S_1'; S_2, \sigma_0\right\rangle \Rrightarrow^{k_0} \sigma''$, к которой может быть применена гипотеза индукции. Это означает, что существует состояние σ' и натуральные числа k_1 и k_2 такие, что $\left\langle S_1', \sigma_0\right\rangle \Rrightarrow^{k_1} \sigma'$ и $\langle S_2, \sigma'\rangle \Rrightarrow^{k_2} \sigma''$, причём $k_1 + k_2 = k_0$.

Учитывая, что $\langle S_1, \sigma\rangle \Rrightarrow \left\langle S_1', \sigma_0\right\rangle$ и $\left\langle S_1', \sigma_0\right\rangle \Rrightarrow^{k_1} \sigma'$, получаем
$\langle S_1, \sigma\rangle \Rrightarrow^{k_1+1} \sigma'$.
Кроме того, мы знаем, что $\langle S_2, \sigma'\rangle \Rrightarrow^{k_2} \sigma''$.
А так как $(k_1 + 1) + k_2 = k_0 + 1$, то доказываемое свойство выполняется.

2. Во втором случае $\langle S_1; S_2, \sigma\rangle \Rrightarrow \langle S_2, \sigma_0\rangle$, так как $\langle S_1, \sigma\rangle \Rrightarrow \sigma_0$.

Тем самым $\langle S_2, \sigma_0\rangle \Rrightarrow^{k_0} \sigma''$, и доказываемое свойство выполняется, если мы выберем $k_1 = 1$ и $k_2 = k_0$. $\square$