ГРАММАТИКИ

Формальные языки и грамматический разбор

К. Владимиров, Intel, 2021
mail-to: konstantin.vladimirov@gmail.com

- ➢ Регулярные выражения и автоматы
- ❑ Практика лексического анализа
- ❑ Грамматики
- ❑ Практика синтаксического анализа

Алфавиты и строки

- Алфавит это множество символов, например $\{a, b, c\}$
- Строкой называется последовательность символов, например $w = \{a, a, c, b\}$
- Для краткости можно записывать $w = aacb$. Пустая строка Λ

Алфавиты и строки

- Алфавит это множество символов, например $\{a, b, c\}$
- Строкой называется последовательность символов, например $w = \{a, a, c, b\}$
- Для краткости можно записывать $w = aacb$. Пустая строка Λ
- Конкатенация строк: $w = aacb$, $z = ba$, $wz = aacbba$, $zw = baaacb$
- Степень: $w^3 = www$, $w^0 = \Lambda$

Формальные языки

- Языком над данным алфавитом называется множество его строк

Язык L_{empty} = пустое множество строк

Язык L_{free} = все возможные строки алфавита (группа по конкатенации)

- Можно ли придумать более интересные языки?

Формальные языки

- Ограничимся простым алфавитом $\{a, b, c\}$
- Язык L_1 = $\{a^m b^n\}$: $a, ab, aab, aabb, abb, aaabbb,$ …
- Язык $L_2 = \{a, cab, caabc\}$
- Язык L_3 = $\{a^n b^n\}$: $ab, aabb, aaabbb,$ …
- Язык L_4 = $\{a^m b^n c a^m b^n\}$: $aca, abcab, aabcaab$ …
- Язык L_5 = $a, b, ba, bab, babba, babbabab,$ … (строки Фибоначчи, начиная с a, b)
- Такие описания несколько неформальны и их сложно расширять
- Но уже сейчас можно понять, что нам предстоит решать задачи на языках

Задачи для формальных языков

- Принадлежность: имея язык L и строчку w, определить принадлежит ли она языку
- Порождение: имея язык L, порождать все его строки последовательно
- Эквивалентность: имея язык L_1 и язык L_2, определить принадлежат ли им одинаковые элементы
- Отрицание: имея язык L_1, описать язык L_2, такой, что он содержит все строки, не принадлежащие L_1
- Чтобы решать все эти задачи, мы хотели бы простого и формального описания языка. И первой попыткой традиционно будут **регулярные выражения**

Регулярные выражения

- Любой алфавитный символ означает язык из этого символа: a это $\{a\}$
- Конкатенация $L_x L_y = \{wz | w \in L_x \wedge z \in L_y\}$
- Дизъюнкция $(L_x + L_y) \; = \{w | w \in L_x \vee w \in L_y\}$
- Замыкание $(L_x)^* = \{\{\Lambda\}, L_x, L_x L_x, L_x L_x L_x, \dots\}$
- Язык L_1 теперь можно описать как $a^* b^*$
- Упражнение: назовите любую строчку, принадлежащую языку $(c(a+b)^* ab)^* ca$
- Упражнение: принадлежит ли ему строчка $caabbca$? Как вы это установили?

Расширенные регулярные выражения

- Довольно часто регулярные выражения расширяются ещё двумя символами

$a?$ = $a + \Lambda$ (ноль или одно повторение)

a^{+} = aa^{*} (одно или больше повторений)

- В конкретных системах могут встречатся синонимы для групп алфавитных символов, например

`[[:digit:]] = ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )`

- Упражнение. Что описывает выражение: `(-)?([[:digit:]])+`
- Упражнение. Напишите выражение для чисел с плавающей точкой

Регулярные выражения в C++

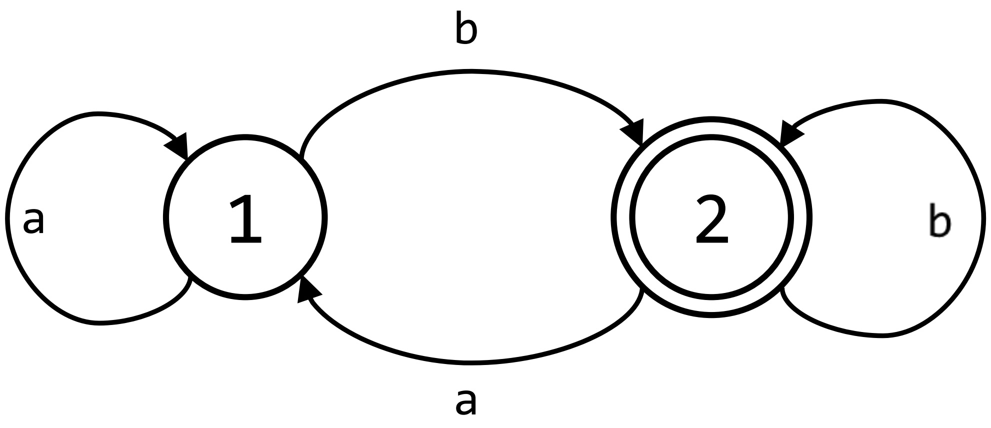
- Для того же выражения $(c(a+b)^*ab)^*ca$

```
const std::regex r1("(c(a|b)*ab)*ca"); // | вместо +
std::cmatch m;
bool res1 = std::regex_match ("caabca", m, r1);
bool res2 = std::regex_match ("cbbbab", m, r1);
```

- Как вы думаете, а как написать такую функцию?
- Очевидно надо сделать некое представление для регулярного выражения

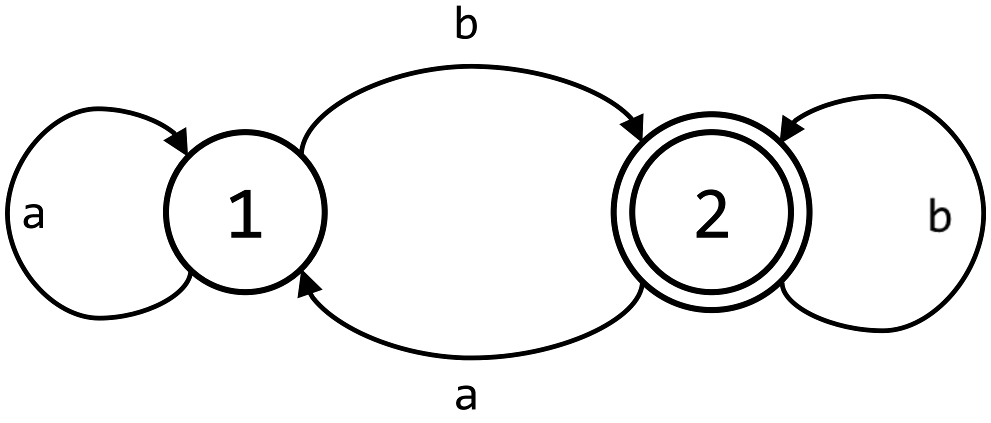
Конечные автоматы: ДКА

- Детерминированным конечным автоматом (ДКА) называется набор состояний и функция перехода между состояниями
- Некоторые состояния (2) называют принимающими (accepting)
- Ровно одно состояние (1) является стартовым
- Какие строки принимает автомат справа?



Конечные автоматы: ДКА

- Детерминированным конечным автоматом (ДКА) называется набор состояний и функция перехода между состояниями
- Некоторые состояния (2) называют принимающими (accepting)
- Ровно одно состояние (1) является стартовым
- Можем ли мы сделать такой автомат который принимал бы заданный регулярный язык?



`b, baaab, bbbab, aaab, bbaabab, ...`

$$(a+b)^{*}b$$

От регулярных выражений к автоматам

- Основная проблема сматчить выражение вроде $(a+b)^*b(b+c)^*$ это недетерминизм в том когда заканчивать матчинг первого замыкания

aabbcb

aabbbabcc

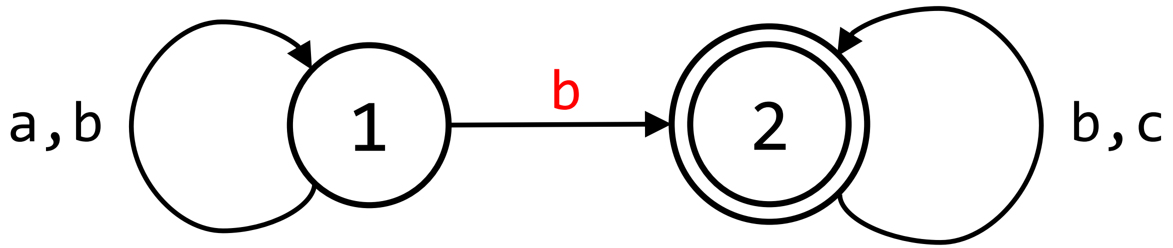
От регулярных выражений к автоматам

- Основная проблема сматчить выражение вроде $(a + b)^* b (b + c)^*$ это недетерминизм в том когда заканчивать матчинг первого замыкания

aabbcb

aabbbabcc

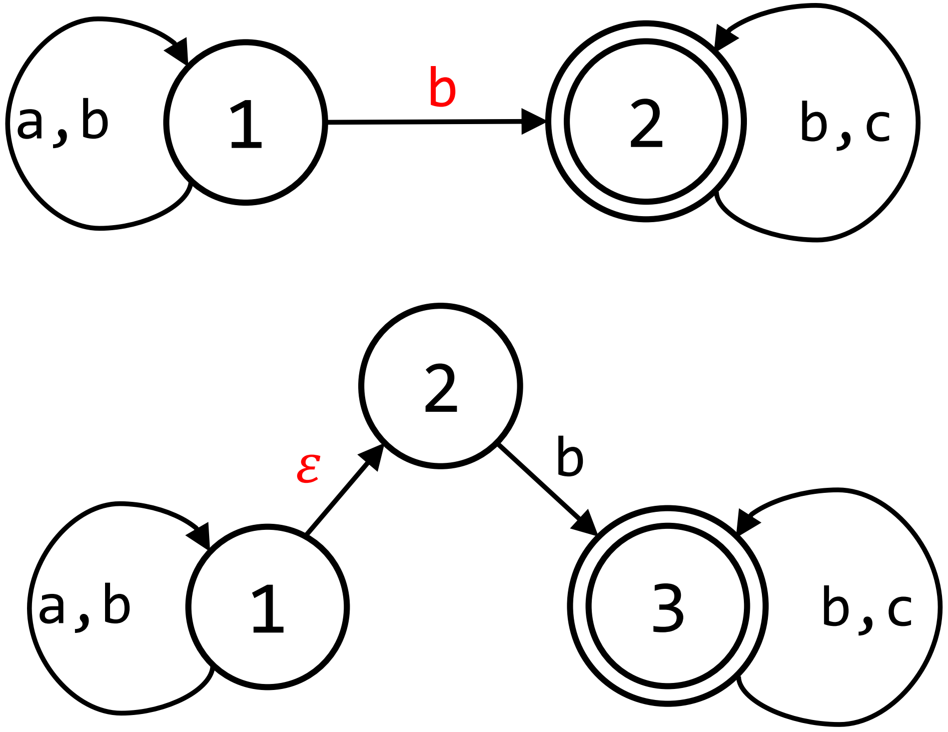
- Что если мы разрешим в конечных автоматах недетерминированные переходы? Например сразу в два состояния по символу b?



Алгоритм построения НКА по выражению

Конечные автоматы: НКА

- Есть два способа разрешить НКА
- Первый способ это разрешить неоднозначность в переходной функции напрямую
- Второй способ это разрешить спонтанные (эпсилон) переходы
- В любом случае можно доказать что всегда можно построить НКА из регулярного выражения
- Но кажется от НКА мало проку для программиста...



От НКА к ДКА

- К счастью всегда можно перейти от недетерминированного к детерминированному конечному автомату
- Алгоритм называется алгоритмом Рабина-Скотта или конструкцией подмножеств и довольно интересен но явно не укладывается в эту лекцию
- Увы, такой переход может привести к экспоненциальному росту числа состояний автомата
- Для того чтобы минимизировать число состояний конечного автомата тоже есть масса довольно сложных и интересных алгоритмов
- Очень хорошо, что std::regex делает всё это за нас. Проблема в том, что она делает это при каждом запуске

[Powerset construction]

- ❑ Регулярные выражения и автоматы
- ➢ Практика лексического анализа
- ❑ Грамматики
- ❑ Практика синтаксического анализа

Задача лексического анализа

- Лексический анализ это переход от текстового ввода к потоку лексем

```
a = 0; b = 1; n = ?;
```

VAR	ASSIGN	CST	EXPR	VAR	ASSIGN	CST	EXPR	VAR	...
a		0		b		1		n	...

- Позволяет рано выявить лексические ошибки и очистить задачу от мусора для более сложного синтаксического анализа

```
a = \0; 1b = 0; n = $; // лексические ошибки
```

Обсуждение

- Можем ли мы использовать регулярные выражения для лексического анализа?

Обсуждение

- Можем ли мы использовать регулярные выражения для лексического анализа?
- Можем, но такое чувство, что это будет одно гигантское регулярное выражение изо всех возможных вариантов лексем
- Мы должны начать с этого регулярного выражения, дальше сделать из него НКА, далее сделать из него ДКА, далее минимизировать ДКА
- Это лучше автоматизировать и сделать один раз где-нибудь до компиляции программы

FLEX

- Язык и система генерации лексических анализаторов для C++. Выходом flex является класс `yyFlexLexer` на C++ с интерфейсом анализа

```
%{
   // сюда можно вставить любые определения до паттернов
%}

// здесь дополнительные имена для регексов

%%

pattern  { /* action */ }

%%

// здесь любой код после паттернов
```


https://westes.github.io/flex/manual/

FLEX: простой пример

- Простой лексер для грамматики с операторами +, -, = и числами

```
200 + 2 = 400 - 0198 == 502 - 200
```

- Здесь `0198` это лексическая ошибка, а вот удвоенное = это просто две лексемы подряд
- Обратите внимание на использование `yytext` в тексте правила

```
cout << "number <" << yytext << ">" << endl;
```

- Это одна из особых переменных, доступных внутри сканера
- Снаружи через интерфейс она доступна как `lexer->YYText()`

numexample

FLEX: простой пример

```
%{
using std::cout;
using std::endl;
%}

WS       [ \t\n\v]+
OP       [\+\-\=]
DIGIT    [0-9]
DIGIT1   [1-9]

%%
{WS}               /* skip blanks and tabs */
{OP}               { cout << "operator <" << yytext[0] << ">" << endl; return 1; }
{DIGIT1}{DIGIT}*   { cout << "  number <" << yytext    << ">" << endl; return 1; }
.                  { cout << " UNKNOWN <" << yytext[0] << ">" << endl; return 1; }
%%
```

FLEX: немного сложнее

- На C++ хочется делать собственные классы лексеров, хранящие нужную информацию
- Для этого можно использовать

```
%option yyclass="NumLexer"
```

- Эта опция проинформирует flex, что вместо yyFlexLexer будет использоваться его кастомный наследник
- Поскольку все actions теперь будут вызываться внутри `NumLexer::yylex`, они могут быть приватными функциями

FLEX: пример посложнее

```
%option yyclass="NumLexer"
%option c++

%{
#include "numlexer.hpp"
%}

WS      [ \t\n\v]+
DIGIT   [0-9]
DIGIT1  [1-9]

%%
{WS}                /* skip blanks and tabs */
"+"                 return process_plus();
"-"                 return process_minus();
"="                 return process_eq();
{DIGIT1}{DIGIT}*    return process_digit();
.                   return process_unknown();
%%
```

Обсуждение

- Увы, не все языки являются регулярными
- Лемма о накачке гласит, что для любого достаточно длинного слова w в регулярном языке найдётся такая декомпозиция $w = xyz$, что все слова xy^nz также принадлежат этому языку
- Поэтому язык a^nb^m регулярный: вместе с $a^{n-1}ab^m$ ему принадлежат все $a^{n-1}a^kb^m$
- Но это значит, что язык a^nb^n **не регулярный**
- Также не регулярен язык всевозможных регулярных выражений
- К счастью, есть более совершенные способы описания языков

Pumping lemma

- ❑ Регулярные выражения и автоматы
- ❑ Практика лексического анализа
- ➢ Грамматики
- ❑ Практика синтаксического анализа

Грамматика регулярных выражений

- Все правильные регулярные выражения над $\{a, b, c\}$ и только их можно построить по следующему правилу

$$A \to A + A \mid (A) \mid A.A \mid A^* \mid a \mid b \mid c$$

- Например построим $(c.(a + b)^*.a.b)^*.c.a$

$$A \to A.A \to A.A.A \to (A)^*A.A \to (A.A)^*A.A \to (A.A.A)^*A.A \to (A.A.A.A)^*A.A \to$$

$$\to (A.(A)^*.A.A)^*A.A \to (A.(A + A)^*.A.A)^*.A.A \to \cdots \to (c.(a + b)^*.a.b)^*.c.a$$

- Мы получили вывод, использующий на каждом шаге одну продукцию.

Грамматика

- По определению грамматика состоит из продукций $\alpha \to \beta$.

$A \to A + A \mid (A) \mid A.A \mid A^* \mid a \mid b \mid c$

- Нетерминальные символы в общем случае могут стоять и слева и справа, терминальные только справа.
- Для языка регулярных выражений над $\{a, b, c\}$.
- Терминалы: a, b, c, $+$, (,), $*$
- Нетерминалы: пока только A.
- Обратим внимание: во всех продукциях языка регулярных выражений у нас слева всего один нетерминал.

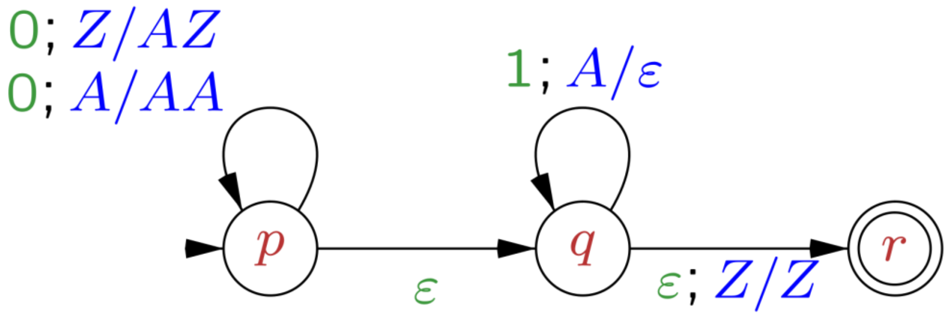
Контекстная свобода и зависимость

- **Контекстно свободной** грамматикой называется такая, которую можно представить так, чтобы слева в каждой продукции был ровно один нетерминал
- Интуитивно это означает, что "if везде if"
- Любой регулярный язык тривиально является контекстно свободным
- Язык L_4 = $\{a^m b^n c a^m b^n\}$ не является контекстно свободным
- Контекстно-свободный язык эквивалентен автомату с магазинной памятью (естественному обобщению обычного автомата)

[wiki page on context-free grammars]

Автоматы с магазинной памятью

- Pushdown automata это обычный НКА, к которому добавлен стек
- На рисунке справа стек изображён синим
- Переход делается по входному символу и по стековому символу
- При этом при переходе можно запушить или извлечь символ
- Z/AZ означает "если верхушка Z то перейти и запушить A"



Автомат для разбора языка $\{0^n1^n\}$

Вывод в грамматике

- Мы говорим, что в данной грамматике выражения бывают выводимые и нет

$A \to A + A \mid (A) \mid A.A \mid A^* \mid a \mid b \mid c$

- Обратим внимание, что у нас есть две естественные стратегии вывода: брать самый левый или самый правый нетерминал на каждом шаге

$A \to A + A \to A + A + A \to A + A + c \to A + b + c \to a + b + c$

$A \to A + A \to A + A + A \to a + A + A \to a + b + A \to a + b + c$

- Это нормально: мы можем задаться левым или правым выводом и если он единственный, у нас всё хорошо
- А вот если нет, то грамматика неоднозначна

Грамматика: неоднозначность

- Можно заметить много проблем у такой грамматики

$$A \to A + A \,|(A)|\, A.A \mid A^*| \; a \mid b \mid c$$

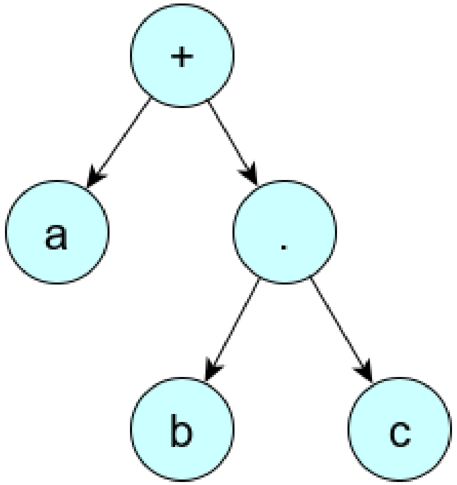
- Строчка $a + bc$ имеет два разных левых вывода

$$A \to A.A \to A + A.A \to a + A.A \to a + b.A \to a + b.c$$

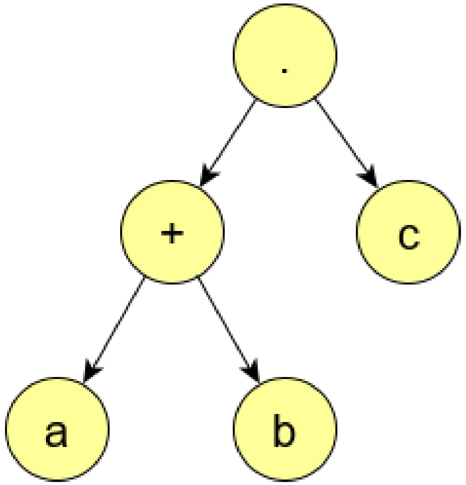
$$A \to A + A \to a + A \to a + A.A \to a + b.A \to a + b.c$$

- Эта неоднозначность очень неприятно выглядит, если взглянуть на получившиеся деревья вывода

Грамматика: неоднозначность



$$A \to A + A \to \cdots$$



$$A \to AA \to \cdots$$

- У приведенной ниже грамматики регулярных выражений

$$A \to A + A\ |(A)|\ A.A\ |\ A^*|\ a\ |\ b\ |\ c$$

- Строчка $a + b.c$ имеет два разных левых вывода

$$A \to AA \to A + A.A \to \cdots$$

$$A \to A + A \to A + A.A \to \cdots$$

- В первом случае она будет означать $a + (b.c)$, а во втором случае это будет $(a + b).c$

Грамматика: приоритеты операций

- Добавив нетерминалов мы получаем приоритеты у операций
- Строчка $a + bc$ имеет по сути единственный вывод с точностью до порядка выбора продукций
- Логично выбирать продукцию либо для самого левого нетерминала либо для самого правого (leftmost/rightmost)

$A \to B + A \mid B$

$B \to C.B \mid C$

$C \to D^* \mid D$

$D \to (E) \mid E$

$E \to a \mid b \mid c$

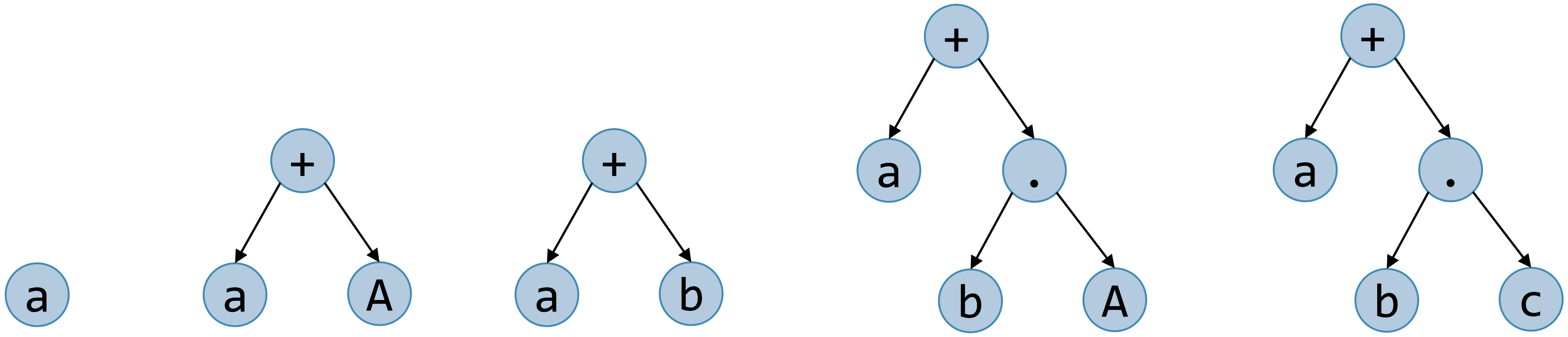
Заметим также, что эта грамматика факторизована слева, у неё нет продукций вида $A \to A\alpha$

Таксономия L/R

- Первая буква означает направление
  - L означает слева направо
  - R означает справа налево
- Вторая буква означает выбранный нетерминал
  - L означает берём самый левый
  - R означает берём самый правый
- Далее могут следовать скобки в которых стоит сколько символов предпросматриваем
- Есть также префиксы например `LALR (LA = look ahead)`
- Мы можем формулировать вопросы про языки в терминах их классов

Нисходящий парсинг LL(k)

- Нисходящий парсинг контекстно-свободных грамматик состоит в построении деревьев сверху вниз



$a + b.c$ $\quad$ $a + b.c$ $\quad$ $a + b.c$ $\quad$ $a + b.c$ $\quad$ $a + b.c$

$A \to A + A \mid (A) \mid AA \mid A^* \mid a \mid b \mid c$

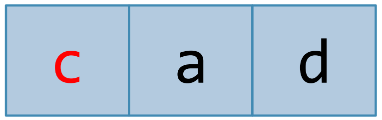
Рекурсивный спуск LL(k)

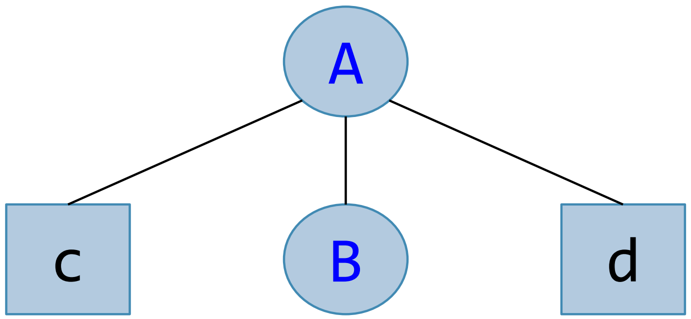
- Для достаточно хороших грамматик для построения дерева можно использовать метод рекурсивного спуска
- Для каждого нетерминала X_i делается отдельная функция X_i

```
void Xi() {
  // выбрать продукцию Xi → Y1Y2 … Yk
  for (i = 1 to k) {
    if (Yi nonterminal) Yi();
    else if (Yi equals current input symbol) advance input
    else ???
  }
}
```

Рекурсивный спуск

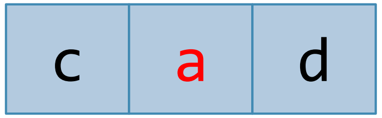
- Первая главная проблема нисходящего разбора: может понадобиться откат если продукция была выбрана неудачно
- Рассмотрим грамматику $A \to cBd$; $B \to ab \mid a$
- Как выводить строчку cad?
- Первая продукция очевидно $A \to cBd$

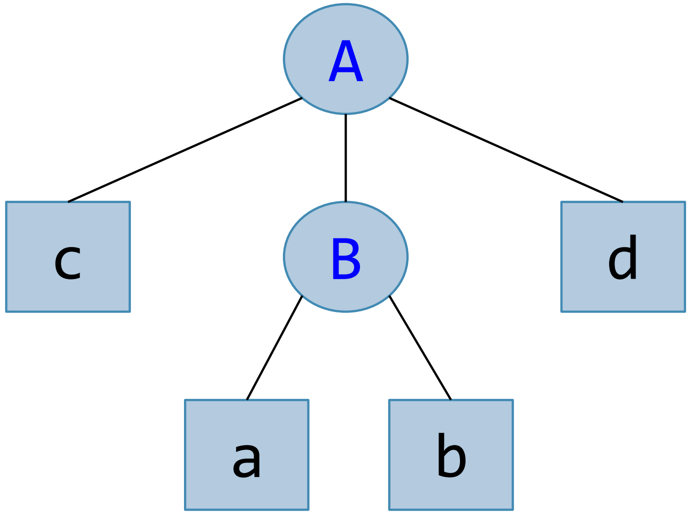




Рекурсивный спуск

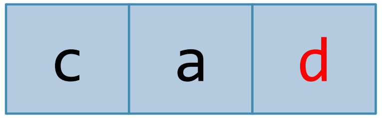
- Первая главная проблема нисходящего разбора: может понадобиться откат если продукция была выбрана неудачно
- Рассмотрим грамматику $A \to cBd$; $B \to ab \mid a$
- Как выводить строчку cad?
- Первая продукция очевидно $A \to cBd$
- Вторую продукцию можно выбрать как $B \to ab$

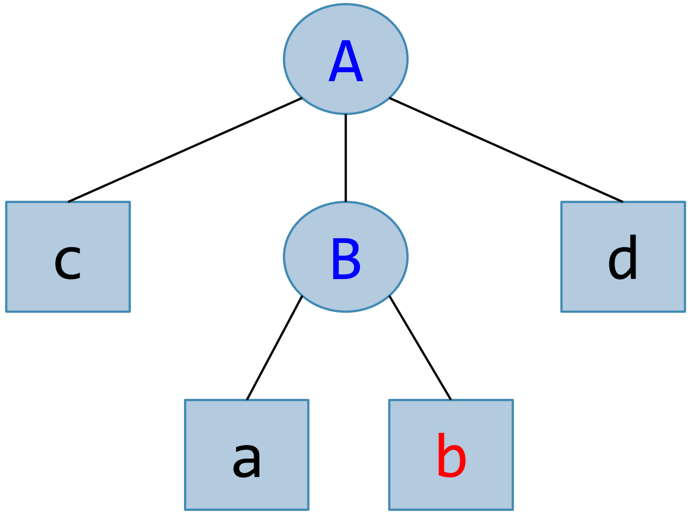




Рекурсивный спуск

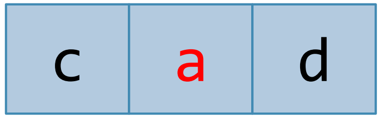
- Первая главная проблема нисходящего разбора: может понадобиться откат если продукция была выбрана неудачно
- Рассмотрим грамматику $A \to cBd$; $B \to ab \mid a$
- Как выводить строчку cad?
- Первая продукция очевидно $A \to cBd$
- Вторую продукцию можно выбрать как $B \to ab$
- Увы, на третьем шаге d не с чем матчить
- Делаем откат ко второму шагу

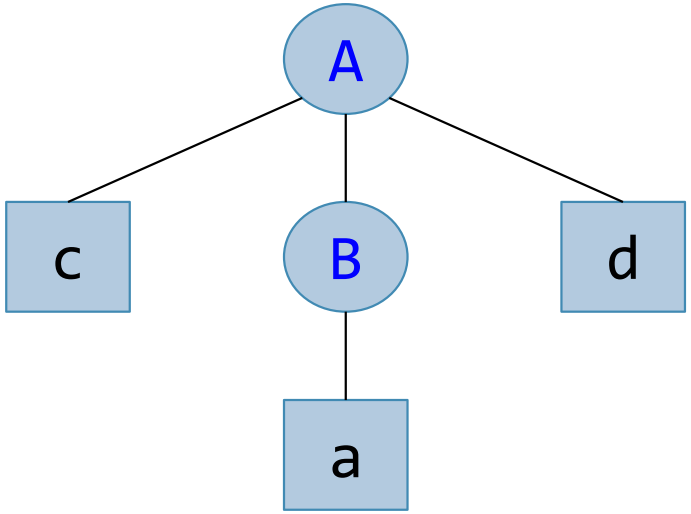




Рекурсивный спуск

- Первая главная проблема нисходящего разбора: может понадобиться откат если продукция была выбрана неудачно
- Рассмотрим грамматику $A \to cBd$; $B \to ab \mid a$
- Как выводить строчку cad?
- Первая продукция очевидно $A \to cBd$
- Вторую продукцию можно выбрать как $B \to a$
- Теперь на третьем шаге всё тоже будет успешно
- Но сколько откатов потребует нетривиальная грамматика?





Обсуждение: $LL(1)$

- Можем ли мы построить парсер без откатов?
- Да, но не для всех грамматик
- Рассмотрим следующую грамматику арифметических выражений:

$E \to E \mathbf{+} T \mid T$

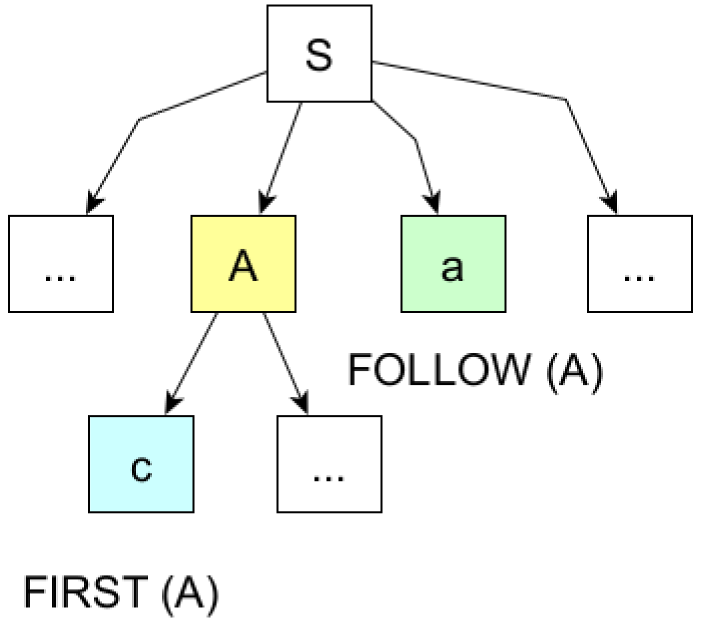
$T \to T \mathbf{*} F \mid F$

$F \to (\ E\ ) \mid \boldsymbol{id}$

- Понятно ли почему тут $LL(1)$ не получится?

Ограничения на $LL(1)$

- Можно доказать, что язык относится к $LL(1)$ если и только если для двух разных продукций $A \rightarrow \alpha$ и $A \rightarrow \beta$
- Не существует терминала a, такого, чтобы α и β выводили строки начинающиеся с a
- Не более чем одна из α и β выводит пустую строку
- Если β выводит пустую строку, то α не выводит строк из $FOLLOW(A)$



Построение FIRST / FOLLOW

$E \to T\ E'$

$E' \to +\, T\ E' \mid \epsilon$

$T \to F\ T'$

$T' \to *\ F\ T' \mid \epsilon$

$F \to (\ E\ ) \mid id$

$FIRST(F) = FIRST(T) = FIRST(E) = \{\ (\ , id\}$

$FIRST(E') = \{+, \epsilon\}$

$FIRST(T') = \{*, \epsilon\}$

$FOLLOW(E) = FOLLOW(E') = \{\ )\ , \$\}$

$FOLLOW(T) = FOLLOW(T') = \{+,)\ , \$\}$

$FOLLOW(F) = \{+,\ *,\ ), \$\}$

Таблица для $LL(1)$ разбора

- По множествам FIRST и FOLLOW можно построить таблицу выбора продукций

$E \to T\ E'$

$E' \to +T\ E' \mid \epsilon$

$T \to F\ T'$

$T' \to *\ F\ T' \mid \epsilon$

$F \to (\ E\ ) \mid id$

	id	**+**	*****	**(**	**)**	**$**
E	$E \to T\ E'$			$E \to T\ E'$		
E'		$E' \to +T\ E'$			$E' \to\ \epsilon$	$E' \to\ \epsilon$
T	$T \to F\ T'$			$T \to F\ T'$		
T'		$T' \to\ \epsilon$	$T' \to *\ F\ T'$		$T' \to\ \epsilon$	$T' \to\ \epsilon$
F	$F \to id$			$F \to (\ E\ )$		

- Попробуйте прогнать по этой таблице $a + b * c$

Dangling else

- Классический пример неоднозначности не позволяющей построить LL(1) парсер это dangling else

$$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt$$
$$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt$$
$$stmt \rightarrow S_i$$
$$expr \rightarrow E_i$$

- Тогда в следующей строчке будет непонятно к какому if отнести else

$$\textbf{if}\ E_1\ \textbf{then}\ \textbf{if}\ E_2\ \textbf{then}\ S_1\ \textbf{else}\ S_1$$

- Неоднозначность из грамматики можно убрать, сделав из неё нормальную форму Хомского (CNF)

Более совершенные методы

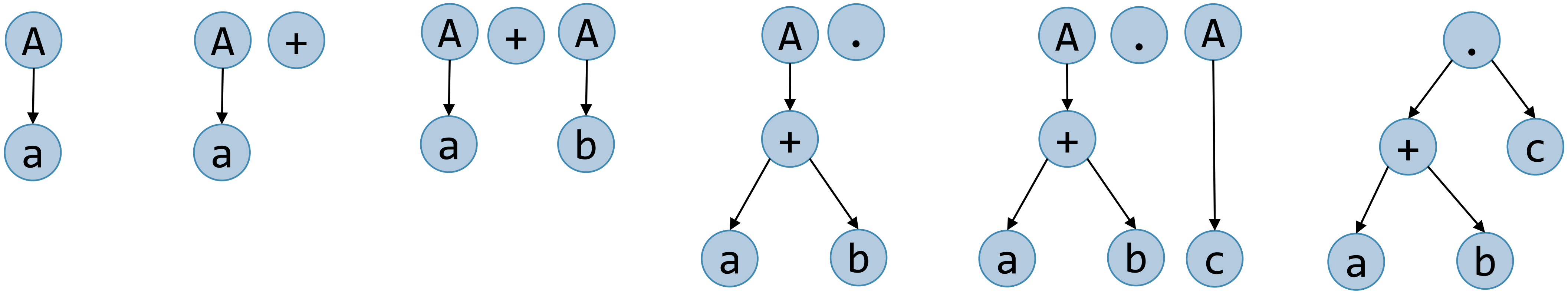
- Рекурсивный спуск $LL(k)$ может потребовать откатов
- Ошибки в грамматике (неоднозначности, циклы) не видны пока в них не наступишь в рантайм
- Нисходящий $LL(1)$ эффективен но не для всех языков применим
- Более совершенными являются восходящие методы

Восходящий парсинг $LR(k)$

- Восходящий разбор основан на двух операциях
- shift – сдвинуть текущий элемент входного потока в стек
- reduce – использовать продукцию чтобы изменить содержимое стека на терминал слева от продукции
- Главная хитрость этого метода когда делать shift и когда reduce
- Для принятия этого решения сначала строятся множества FIRST/FOLLOW, потом для них строится $LR(0)$ автомат
- Всё это технически сложные действия, которые проще автоматизировать

Восходящий парсинг LR

- Восходящий парсинг контекстно-свободных грамматик состоит в построении деревьев снизу вверх. Также известен как shift-reduce подход



shift	shift	shift	reduce, shift	shift	reduce
$a + b.c$	$a + b.c$	$a + b.c$	$a + b.c$	$a + b.c$	$a + b.c\$$

$A \to A + A \mid (A) \mid A.A \mid A^* \mid a \mid b \mid c$

Обсуждение

- К сожалению написать руками LR парсер очень нелегко, там нужно делать $LR(0)$ автомат и использовать его в парсинге
- Если мы генерируем лексер из регулярных выражений, можно ли сделать также парсер из правил продукции?

- ❑ Регулярные выражения и автоматы
- ❑ Практика лексического анализа
- ❑ Грамматики
- ➢ Практика синтаксического анализа

Задача синтаксического анализа

- Основная задача это построить синтаксическое дерево из потока лексем
- Мы не хотели бы делать LL(k) из-за бэктрекинга
- Хочется сделать LR или LALR, но делать их руками может быть крайне неудобно
- Поэтому мы хотели бы точно также сгенерировать парсер

BISON

- Язык и система генерации синтаксических анализаторов для C++

```
%code qualifier { /* C++ code */ }
// определения для bison
%%
grammar rule  { /* action */ }
%%
// здесь любой код после правил грамматики
```

- qualifier всегда можно посмотреть в грамматике бизона

bison manual

BISON: простой пример

- Допустим нам необходимо проверить серию равенств

```
200 + 2 = 400 - 198; 100 + 1 = 100 - 2;
```

- Для простоты всё что нужно это вывести на экран результаты
- Обратите внимание на (увы, неправильную) явную грамматику для expr:

```
expr: NUMBER                { $$ = $1; }
    | expr PLUS expr        { $$ = $1 + $3; }
    | expr MINUS expr       { $$ = $1 - $3; };
```

- Здесь использованы бизоновские сокращения для семантических значений
- Ясно что если мы можем в правилах складывать, можно там и AST строить

Токены и нетерминалы

```
%language "c++"

%token
  EQUAL   "="
  MINUS   "-"
  PLUS    "+"
  SCOLON  ";"
  ERR;

%token <int> NUMBER
%nterm <int> equals
%nterm <int> expr

%left '+' '-'

%start program
```

- Здесь пропущены технические куски кода с заголовочными файлами и всем прочим
- Токен без типа это просто токен
- Токен с типом несёт в себе семантическое значение
- Операторы сделаны левоассоциативными
- Задана стартовая точка

Грамматические правила

```
program: eqlist;

eqlist: equals SCOLON eqlist
| equals SCOLON
;

equals: expr EQUAL expr { $$ = ($1 == $3);}
;

expr: NUMBER                { $$ = $1; }
| expr PLUS expr            { $$ = $1 + $3; }
| expr MINUS expr           { $$ = $1 - $3; }
;
```

- Видно что почти точно повторяются продукции грамматики

$EqList \to Equals; EqList \mid Equals$

$Equals \to Expr = Expr$

$Expr \to NUM \mid Expr + Expr \mid Expr - Expr$

- Текущее семантическое значение это `$$`
- Символами `$i` обозначаются значения параметров продукций (нумерация с единицы)

Взаимодействие с лексером

- Поскольку лексер должен распознавать те же токены которые являются основой для парсера, в bison есть возможность сгенерировать заголовочный файл, где все они перечислены
- Это оказывается очень удобно, достаточно просто включить его в лексер

Код лексера

```
%option c++

%{
#include "numgrammar.tab.hh"
%}

WS      [ \t\n\v]+
DIGIT   [0-9]
DIGIT1  [1-9]

%%
{WS}                /* skip blanks and tabs */
"+"                 return yy::parser::token_type::PLUS;
"-"                 return yy::parser::token_type::MINUS;
"="                 return yy::parser::token_type::EQUAL;
";"                 return yy::parser::token_type::SCOLON;
{DIGIT1}{DIGIT}*    return yy::parser::token_type::NUMBER;
.                   return yy::parser::token_type::ERR;
%%
```

Драйвер: связующее звено

- Поскольку и лексер и парсер в данном случае являются классами, драйвер должен соединять их воедино:

```
parser::token_type yylex(parser::semantic_type* yylval) {
  // вызвали лексер
  auto tt = static_cast<parser::token_type>(plex_->yylex());
  // установили семантическое значение
  if (tt == yy::parser::token_type::NUMBER)
    yylval->as<int>() = std::stoi(plex_->YYText());
  // вернули тип токена
  return tt;
}
```

Ошибки shift и reduce

- Два вида ошибок
  - менее мрачные shift/reduce
  - более мрачные reduce/reduce
- Бизон предупреждает о тех и о других и лучше их убирать из грамматики
- Их удобно посмотреть на демо парсера для арифметических выражений

simplest-arith

BISON: семантические значения

- Допустим мы не хотим в узлах сразу считать, а хотим накапливать вектора
- Нет ничего проще. Благодаря объявлению

```
%define api.value.type variant
```

- Мы можем иметь сколь угодно сложные семантические типы, в частности

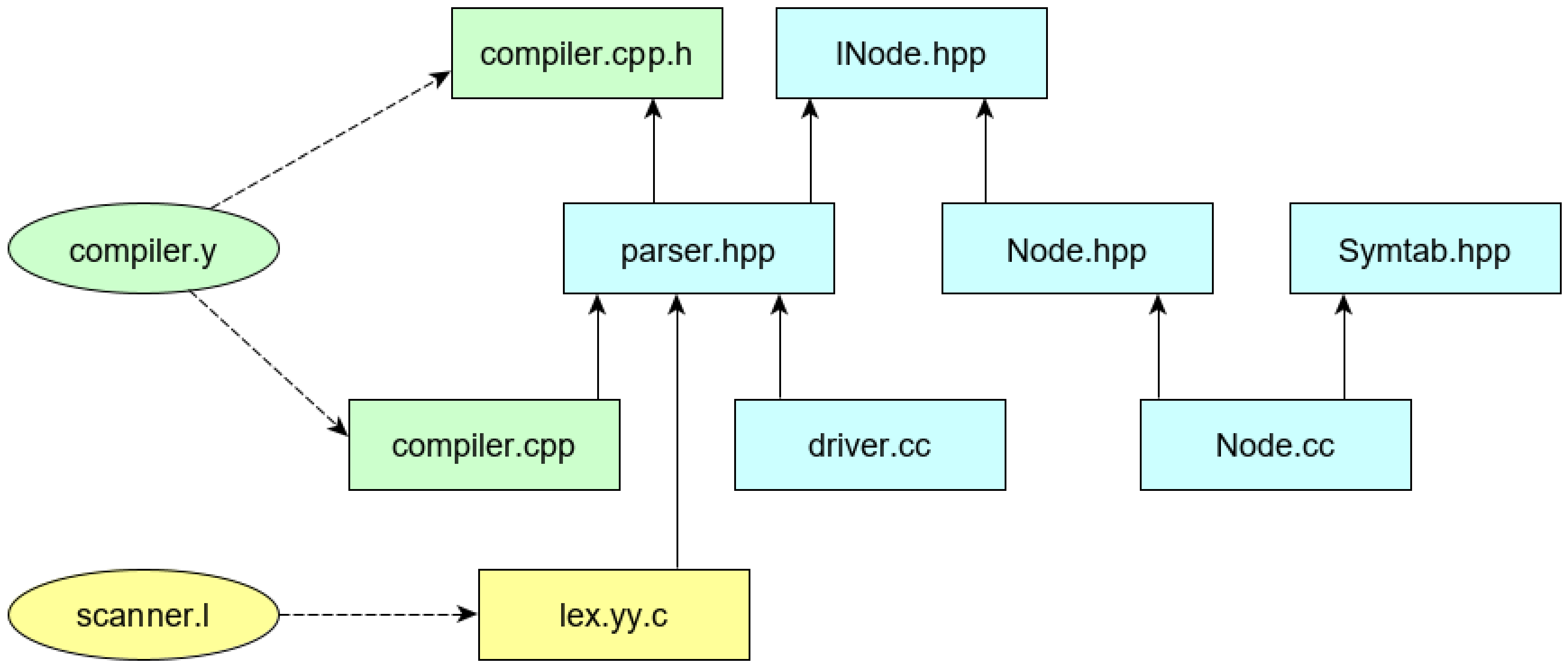
```
%nterm <vector<int>> expr
%nterm <pair<vector<int>, vector<int>>> equals
%nterm <vector<pair<vector<int>, vector<int>>>> eqlist
```

- И это тоже будет работать

Обсуждение

- Что насчёт обработки ошибок?
- В системе bison она концептуально не слишком сложна, но требует много технической работы (как впрочем и в любом компиляторе)

Простая архитектура компилятора



Литература

- $[CC11]$ ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- $[BS]$ Bjarne Stroustrup – The C++ Programming Language (4th Edition) , 2013
- $[DB]$ Alfred Aho, Jeffrey Ullman – Compilers: Principles, Techniques, and Tools (2nd Edition), 2006
- $[Aut]$ Jeffrey Ullman – Automata Theory online course, online.stanford.edu
- $[Comp]$ Alex Aiken – Compilers online course, online.stanford.edu