Курс «Компиляторные технологии»

Лекция 11

Синтаксический анализ (2)

Курносов Михаил Георгиевич

www.mkurnosov.net

Сибирский государственный университет телекоммуникаций и информатики
Весенний семестр

Разбор (parsing)

- Для любой контекстно-свободной грамматики существует анализатор, который требует для разбора строки из n терминалов время, не превышающее $O(n^3)$
- Для разбора почти всех встречающихся на практике языков программирования можно построить алгоритм с линейным временем разбора $O(n)$
- **Основные типы синтаксических анализаторов**:
  - **универсальные**: алгоритм Кока-Янгера-Касами (Cocke-Younger-Kasami), алгоритм Эрли (Earley), редко используются на практике из-за низкой эффективности
  - **восходящие** (bottom-up)
  - **нисходящие** (top-down)

Методы разбора
(по порядку построения узлов дерева разбора)

Нисходящие
(сверху вниз, top-down)

- Построение узлов дерева разбора от корня к листьям
- Легко построить вручную (hand-written)

Восходящие
(снизу вверх, bottom-up)

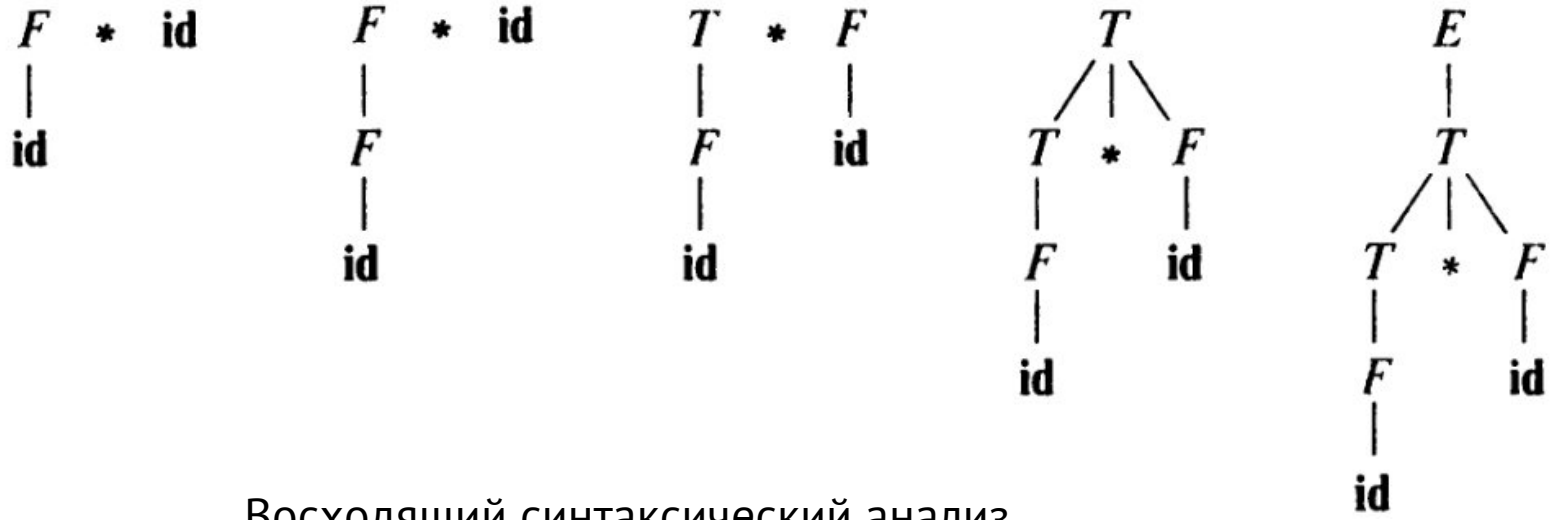
- Построение узлов дерева разбора от листьев к корню
- Применимы для большего класса грамматик
- Применяются в генераторах синтаксических анализаторов

Восходящий синтаксический анализ (bottom-up)

- **Восходящий синтаксический анализ** соответствует построению дерева разбора для входной строки, начиная с листьев (снизу) и идя по направлению к корню (вверх)

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

`id * id` ⟶



Восходящий синтаксический анализ

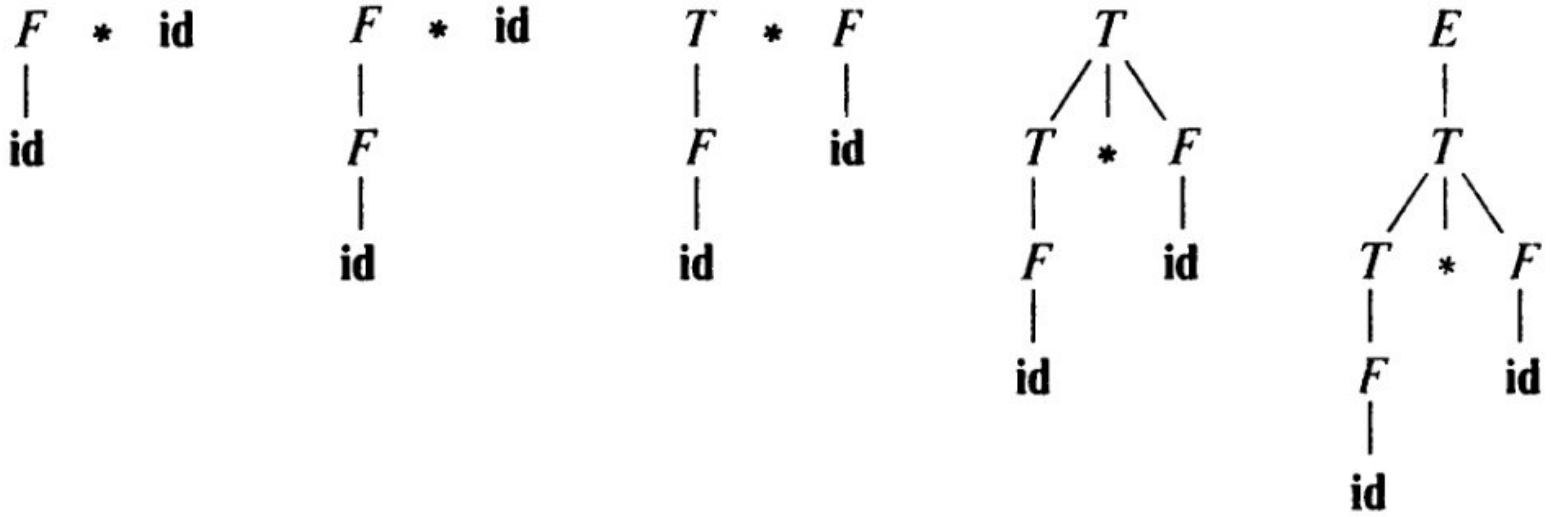
$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Свертки (reductions)

- Восходящий синтаксический анализ — процесс «***свертки***» (reducing) строки *w* к стартовому символу грамматики
- На каждом шаге ***свертки*** (reduction) определенная подстрока, соответствующая телу продукции, заменяется нетерминалом из заголовка этой продукции
- Ключевой вопрос в процессе восходящего синтаксического анализ — когда выполнять свертку и какую продукцию применять

`id * id`

Свертки: $\mathbf{id} * \mathbf{id}, F * \mathbf{id}, T * \mathbf{id}, T * F, T, E$



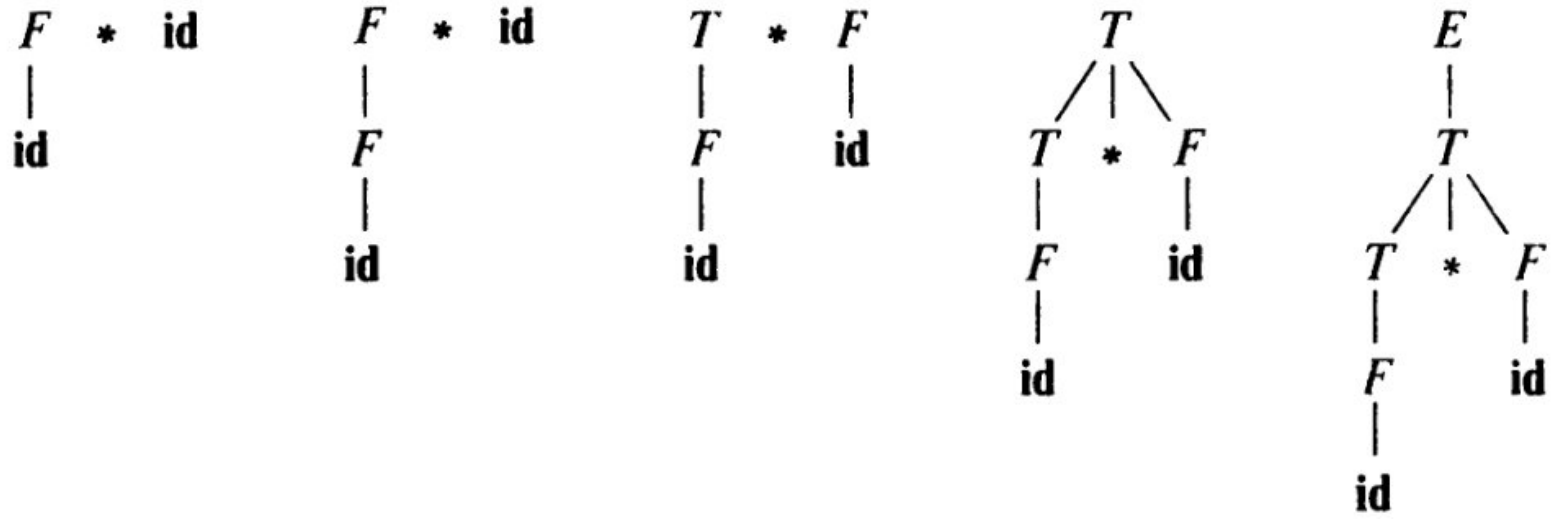
Свертки (reductions)

- **Свертка** — шаг, обратный порождению (выводу из корня дерева разбора)
- **Цель восходящего синтаксического анализа** — построение порождения в обратном порядке

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Свертки: $\mathbf{id} * \mathbf{id}, F * \mathbf{id}, T * \mathbf{id}, T * F, T, E$



Обрезка основ (handle pruning)

- **Восходящий синтаксический анализ** в процессе сканирования входного потока слева направо строит правое порождение в обратном порядке
- **Правосторонний вывод** (rightmost derivation) — вывод слова, в котором каждая последующая строка получена из предыдущей путем замены по одному из правил (продукций) самого правого встречающегося в строке нетерминала
- **Основа** (дескриптор, handle) — подстрока, которая соответствует телу продукции и свертка которой представляет собой один шаг правого порождения в обратном порядке

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Разбор строки: **id * id**

Правая сентенциальная форма	Основа	Сворачивающая продукция
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Обрезка основ (handle pruning)

- **Обращенное правое порождение** (rightmost derivation, RM) может быть получено посредством «обрезки основ»
- Мы начинаем процесс со строки терминалов, которую хотим проанализировать
- Если w — строка грамматики, то пусть $w = \gamma_n$, где γ_n — n-я правосентенциальная форма еще неизвестного правого порождения

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

- Для воссоздания порождения в обратном порядке мы находим основу γ_n в β_n и заменяем ее левой частью продукции $A_n \rightarrow \beta_n$ для получения предыдущей правосентенциальной формы γ_{n-1}
- Затем мы повторяем процесс: находим в γ_{n-1} основу β_{n-1} и свертываем ее для получения правосентенциальной формы γ_{n-2}, ...
- Если после очередного шага правосентенциальная форма содержит только стартовый символ S мы прекращаем процесс и сообщаем об успешном завершении анализа
- Каким образом искать основы?

Синтаксический анализ «перенос-свертка» (Shift-Reduce Parsing)

- **Синтаксический анализ «перенос-свертка»** (ПС-анализом) — разновидность восходящего анализа, в котором для хранения символов грамматики используется стек, а для хранения остающейся непроанализированной части входной строки — входной буфер
- **Начальное состояние**: стек пуст, во входном буфере строка *w*

Стек	Вход
$	w

- В процессе сканирования входной строки слева направо анализатор выполняет нуль или несколько переносов символов в стек, пока не будет готов выполнить свертку строки β символов грамматики на вершине стека к заголовку соответствующей продукции
- Анализатор повторяет этот цикл до тех пор, пока не будет обнаружена ошибка или пока стек не будет содержать только стартовый символ, а входной буфер будет пуст

Стек	Вход
$$S$	$

- Всего ПС-анализатор может выполнять четыре действия: 1) перенос, 2) свертка, 3) принятие и 4) ошибка

Действия ПС-анализатора

- **Перенос** (shift) — перенос очередного входного символа на вершину стека
- **Свертка** (reduce) — правая часть сворачиваемой строки должна располагаться на вершине стека, определяется левый конец строки в стеке и принимается решение о том, каким нетерминалом будет заменена строка
- **Принятие** (accept) — объявление об успешном завершении синтаксического анализа
- **Ошибка** (error) — обнаружение синтаксической ошибки и вызов подпрограммы восстановления после ошибки
- Использование стека в ПС-анализаторе объясняется тем, что основа (handle) всегда находится на вершине стека и никогда — внутри него

Разбор строки:
id * id

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Стек	Вход	Действие
$	$\mathbf{id}_1 * \mathbf{id}_2\$$	Перенос
$\$\mathbf{id}_1$	$* \mathbf{id}_2\$$	Свертка по $F \rightarrow \mathbf{id}$
$\$F$	$* \mathbf{id}_2\$$	Свертка по $T \rightarrow F$
$\$T$	$* \mathbf{id}_2\$$	Перенос
$\$T *$	$\mathbf{id}_2\$$	Перенос
$\$T * \mathbf{id}_2$	$\$$	Свертка по $F \rightarrow \mathbf{id}$
$\$T * F$	$\$$	Свертка по $T \rightarrow T * F$
$\$T$	$\$$	Свертка по $E \rightarrow T$
$\$E$	$\$$	Принятие

Конфликты в процессе ПС-анализа (shift/reduce, reduce/reduce conflicts)

- Существуют контекстно-свободные грамматики, для которых восходящий ПС-анализ неприменим
- При работе с такой грамматикой ПС-анализатор может достичь конфигурации, в которой не может принять решение о том, следует ли выполнить ***перенос или свертку*** (конфликт «перенос-свертка», shift/reduce conflict) либо какое именно из нескольких приведений должно быть выполнено (конфликт «свертка/свертка», reduce/reduce conflict)

- Неоднозначная грамматика с «висящим else»

$$
\begin{aligned}
stmt \;\rightarrow\; & \textbf{if } expr \textbf{ then } stmt \\
\mid\; & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
\mid\; & \textbf{other}
\end{aligned}
$$

- **ПС-анализатор находится в конфигурации**

- Анализатор не может определить является ли **if** *expr* **then** *stmt* основой, без учета того, что находится ниже в стеке
- Конфликт «перенос/свертка» (shift/reduce conflict)

Конфликты в процессе ПС-анализа (shift/reduce, reduce/reduce conflicts)

- Имеется язык, в котором вызываются процедуры по именам с параметрами, заключенными в скобки, и что тот же синтаксис используется и для работы с массивами

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

- **Входная строка**: p(i, j)
- **Поток токенов**: **id** (**id**, **id**)
- После переноса первых трех токенов в стек ПС-анализатор окажется в конфигурации:

Стек	Вход
··· **id** (**id**	, **id**) ···

- В какую продукцию свернуть токен id на вершине стека?
- Правильный выбор:
  - ✓ продукцией (5), если p — процедура
  - ✓ продукцией (7), если p — массив

Конфликты в процессе ПС-анализа (shift/reduce, reduce/reduce conflicts)

- Имеется язык, в котором вызываются процедуры по именам с параметрами, заключенными в скобки, и что тот же синтаксис используется и для работы с массивами

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

- **Входная строка**: p(i, j)
- **Поток токенов**: **id** (**id**, **id**)
- Один из вариантов разрешения конфликта – замена токена **id** в продукции (1) на **procid**

Стек	Вход
··· **procid** (**id**	, **id**) ···

- Выбор определяется третьим от вершины символом в стеке, который не участвует в свертке
- Для управления разбором ПС-анализ
- может использовать информацию «из глубин» стека

Введение в LR-анализ: простой LR (simple LR)

- LR-анализаторы — наиболее распространенный тип восходящих синтаксических анализаторов
- **LR(*k*) parser:**
  - L — сканирование входного потока слева направо
  - R — построение правого порождения в обратном порядке (rightmost derivation in reverse)
  - *k* — количество предпросматриваемых символов входного потока, необходимое для принятия решения
- **Практический интерес:**
  - LR(0) — решение о действиях принимается только на основании содержимого стека, символы входной строки не учитываются
  - LR(1) — решение о действиях принимается на основании содержимого стека и одного символа предпросмотра входной строки
- LR-анализаторы управляются таблицами наподобие нерекурсивных LL-анализаторов
- Чтобы грамматика была LR-грамматикой, достаточно, чтобы синтаксический анализатор, работающий слева направо методом переноса/свертки, был способен распознавать основы правосентенциальных форм при их появлении на вершине стека

Введение в LR-анализ: простой LR (simple LR)

- LR-анализаторы могут быть созданы для распознавания практически всех конструкций языков программирования, для которых может быть написана контекстно-свободная грамматика
- Контекстно-свободные грамматики, не являющиеся LR-грамматиками, существуют, для типичных конструкций языков программирования их можно избежать
- Метод LR-анализа — наиболее общий метод ПС-анализа без возврата, который, кроме того, не уступает в эффективности другим, более примитивным ПС-методам
- LR-анализатор может обнаруживать синтаксические ошибки сразу же, как только это становится возможным при сканировании входного потока
- Класс грамматик, которые могут быть проанализированы LR-методами — надмножество класса грамматик, которые могут быть проанализированы с использованием предиктивных или LL-методов
- В случае грамматик, принадлежащих классу LR(*k*), мы должны быть способны распознать правую часть продукции в порожденной ею правосентенциальной форме с дополнительным предпросмотром к входных символов — это требование существенно мягче требования для LL(*k*)-грамматик, в которых мы должны быть способны распознать продукцию по первым к символам порождения ее тела
- LR-грамматики могут описать существенно больше языков, чем LL-грамматики
- Основной недостаток LR-метода — построение LR-анализатора для грамматики типичного языка программирования вручную требует очень большого объема работы
- Для решения этой задачи нужен специализированный инструмент — генератор LR-анализатора (Bison/Yacc,)

Генераторы синтаксических анализаторов

Name	Parsing algorithm	Input grammar notation	Output languages	Grammar, code	Lexer	Development platform	IDE	License
ANTLR4	Adaptive LL(*)[3]	EBNF	C#, Java, Python, JavaScript, C++, Swift, Go, PHP	Separate	generated	Java virtual machine	Yes	Free, BSD
ANTLR3	LL(*)	EBNF	ActionScript, Ada95, C, C++, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby	Mixed	generated	Java virtual machine	Yes	Free, BSD
APG[4]	Recursive descent, backtracking	ABNF	Python, JavaScript, C, Java	Separate	none	All	No	Free, BSD
Beaver[5][6]	LALR(1)	EBNF	Java	Mixed	external	Java virtual machine	No	Free, BSD
Bison	LALR(1), LR(1), IELR(1), GLR	Yacc	C, C++, Java	Mixed	external	All	No	Free, GNU GPL with exception
BtYacc	Backtracking Bottom-up	?	C++	Mixed	external	All	No	Free, public domain
byacc	LALR(1)	Yacc	C	Mixed	external	All	No	Free, public domain
CL-Yacc[7][8]	LALR(1)	Lisp	Common Lisp	Mixed	external	All	No	Free, MIT
Coco/R	LL(1)	EBNF	C, C++, C#, F#, Java, Ada, Object Pascal, Delphi, Modula-2, Oberon, Ruby, Swift, Unicon, Visual Basic .NET	Mixed	generated	Java virtual machine, .NET framework, Windows, POSIX (depends on output language)	No	Free, GNU GPL
CppCC[9][10]	LL(k)	?	C++	Mixed	generated	POSIX	No	Free, GNU GPL
CUP[11][12]	LALR(1)	?	Java	Mixed	external	Java virtual machine	No	Free, BSD-like
Eli[13][14]	LALR(1)	?	C	Mixed	generated	POSIX	No	Free, GNU GPL, GNU LGPL

https://en.wikipedia.org/wiki/Comparison_of_parser_generators

Введение в LR-анализ: простой LR (simple LR)

- LR-анализаторы могут быть созданы для распознавания практически всех конструкций языков программирования, для которых может быть написана контекстно-свободная грамматика
- Контекстно-свободные грамматики, не являющиеся LR-грамматиками, существуют, для типичных конструкций языков программирования их можно избежать
- Метод LR-анализа — наиболее общий метод ПС-анализа без возврата, который, кроме того, не уступает в эффективности другим, более примитивным ПС-методам
- LR-анализатор может обнаруживать синтаксические ошибки сразу же, как только это становится возможным при сканировании входного потока
- Класс грамматик, которые могут быть проанализированы LR-методами — надмножество класса грамматик, которые могут быть проанализированы с использованием предиктивных или LL-методов
- В случае грамматик, принадлежащих классу LR(*k*), мы должны быть способны распознать правую часть продукции в порожденной ею правосентенциальной форме с дополнительным предпросмотром к входных символов — это требование существенно мягче требования для LL(*k*)-грамматик, в которых мы должны быть способны распознать продукцию по первым к символам порождения ее тела
- LR-грамматики могут описать существенно больше языков, чем LL-грамматики
- Основной недостаток LR-метода — построение LR-анализатора для грамматики типичного языка программирования вручную требует очень большого объема работы
- Для решения этой задачи нужен специализированный инструмент — генератор LR-анализатора (Bison/Yacc,)

Пункты (items) и LR(0)-автомат

- Каким образом ПС-анализатор выясняет, когда следует выполнять перенос, а когда — свертку?
- Стек содержит $T, а очередной входной символ *, каким образом синтаксический анализатор узнает, что *T* на вершине стека — не основа, так что корректное действие — перенос, а не свертка *T* в *E*?

- LR-анализатор принимает решение о выборе «перенос/свертка», поддерживая состояния, которые отслеживают, где именно в процессе синтаксического анализа мы находимся
- Состояния представляют собой множества «пунктов»
- **LR(0)-пункт** (item) грамматики *G* — это продукция с точкой в некоторой позиции правой части

$$A \to XYZ$$

Четыре пункта продукции *A*:

$$\begin{aligned} A &\to \cdot XYZ \\ A &\to X \cdot YZ \\ A &\to XY \cdot Z \\ A &\to XYZ \cdot \end{aligned}$$

Разбор строки:
id * id

$$\begin{aligned} E &\to E+T \mid T \\ T &\to T*F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned}$$

Стек	Вход	Действие
$	$\mathbf{id}_1 * \mathbf{id}_2\$$	Перенос
$\$\mathbf{id}_1$	$* \mathbf{id}_2\$$	Свертка по $F \to \mathbf{id}$
$\$F$	$* \mathbf{id}_2\$$	Свертка по $T \to F$
$\$T$	$* \mathbf{id}_2\$$	Перенос
$\$T *$	$\mathbf{id}_2\$$	Перенос
$\$T * \mathbf{id}_2$	$\$$	Свертка по $F \to \mathbf{id}$
$\$T * F$	$\$$	Свертка по $T \to T * F$
$\$T$	$\$$	Свертка по $E \to T$
$\$E$	$\$$	Принятие

Пункты (items) и LR(0)-автомат

- **LR(0)-пункт** (item) грамматики *G* — это продукция с точкой в некоторой позиции правой части
- Пункт указывает, какую часть продукции мы уже просмотрели в данной точке в процессе синтаксического анализа
- Примеры пунктов:

$$A \to \cdot XYZ$$

указывает, что во входном потоке мы ожидаем встретить строку, порождаемую *XYZ*

$$A \to X \cdot YZ$$

уже просмотрена строка, порожденная *X*, и мы ожидаем получить из входного потока строку, порождаемую *YZ*

$$A \to XYZ \cdot$$

уже обнаружено тело *XYZ* и что, возможно, пришло время свернуть *XYZ* в *A*

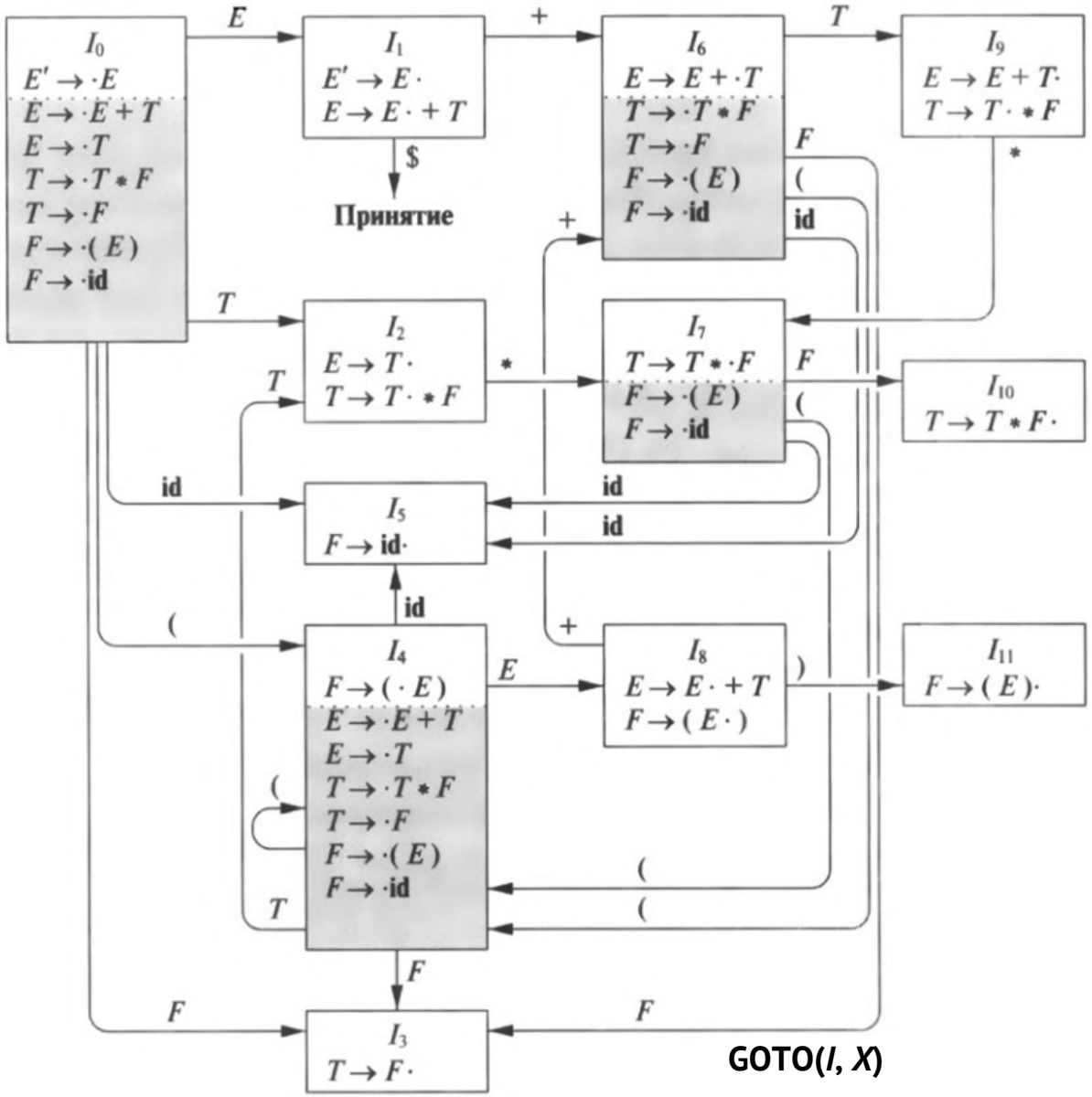
LR(0)-автомат

- **LR(0)-автомат** (LR(0) automaton) — детерминированный конечный автомат, который используется для принятия решений в процессе синтаксического анализа
- Каждое состояние LR(0)-автомата — множество пунктов в **каноническом наборе пунктов LR(0)** (canonical LR(0) collection)

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

- *небазисные пункты* (nonkernel) — в заштрихованных частях прямоугольников состояний

LR(0)-автомат для грамматики выражений



Канонический набор пунктов LR(0)

- **Канонический набор LR(0)** — набор множеств LR(0)-пунктов
- Для построения канонического LR(0)-набора введем расширенную грамматику и две функции: CLOSURE и GOTO
- **Расширенная грамматика *G'*** — грамматика *G* с новым стартовым символом *S'* и продукцией *S'* —> *S*
- Назначение новой стартовой продукции — указание синтаксическому анализатору, когда следует прекратить анализ и сообщить о принятии входной строки (принятие осуществляется тогда и только тогда, когда синтаксический анализатор выполняет свертку с использованием продукции *S'* —> *S*)
- Каждое состояние LR (0)-автомата представляет множество пунктов в каноническом наборе LR(0)
- Пусть *I* — некоторое множество пунктов грамматики *G*
- **Алгоритм вычисления функции CLOSURE(*I*) — замыкание множеств пунктов**
- **CLOSURE**(*I*) — множество пунктов, построенное из *I* согласно двум правилам:
  - В CLOSURE(*I*) добавляются все пункты из *I*
  - Если *A* —> α•*B*β входит в CLOSURE(*I*), а *B* —> γ является продукцией, то в CLOSURE(*I*) добавляется пункт *B* —> •γ, если его там еще нет
  - Правило применяется до тех пор, пока не останется пунктов, которые могут быть добавлены в CLOSURE(*I*)

Канонический набор пунктов LR(0)

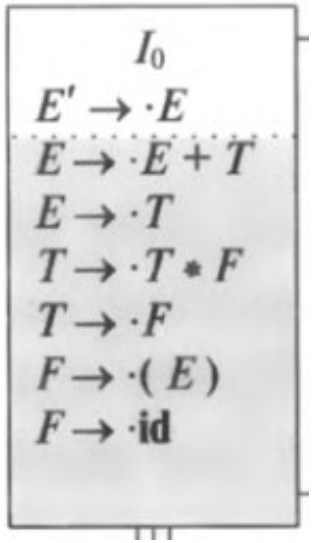
Расширенная грамматика *G'*

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

I — множество из одного пункта

$$\{[E' \rightarrow \cdot E]\}$$

CLOSURE(*I*) = I_0



Канонический набор пунктов LR(0)

- **Канонический набор LR(0)** — набор множеств LR(0)-пунктов
- Для построения канонического LR(0)-набора введем расширенную грамматику и две функции: CLOSURE и GOTO
- **Алгоритм вычисления функции GOTO(*I, X*)**
  - *I* — множество пунктов грамматики *G*, *X* — грамматический символ
  - GOTO(*I*, *X*) — замыкание множества всех пунктов [*A* —> α*X*•β], таких, что [*A* —> α•*X*β] находится в *I*

I множество из двух пунктов

$$\{[E' \to E\cdot], [E \to E \cdot + T]\}$$

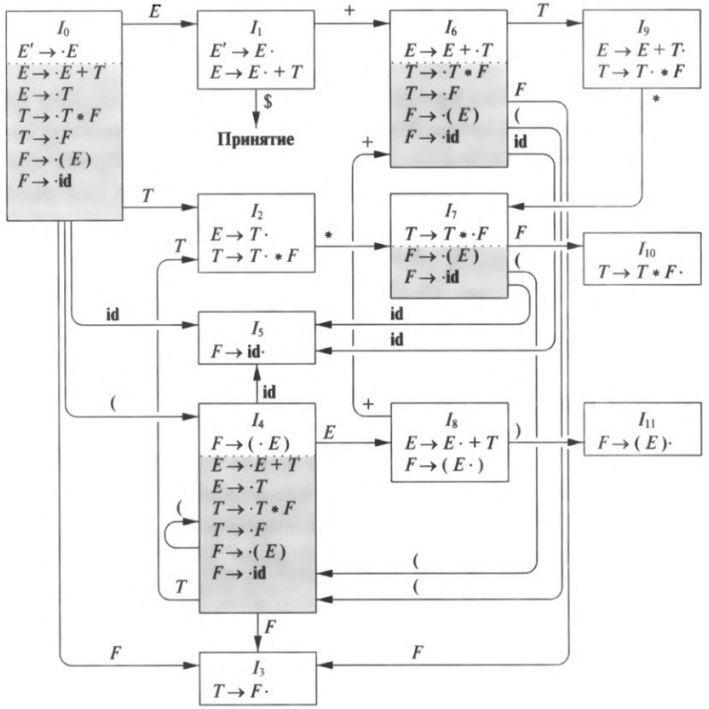
GOTO(*I*, +) =

$$\begin{aligned} E &\to E + \cdot T \\ T &\to \cdot T * F \\ T &\to \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

Построение канонического набора пунктов LR(0)

```
void items (G') {
    C = {CLOSURE ({[S' → ·S]})};
    repeat
        for ( каждое множество пунктов I в C )
            for ( каждый грамматический символ X )
                if ( множество GOTO (I, X) не пустое и не входит в C )
                    Добавить GOTO (I, X) в C;
    until нет новых множеств пунктов для добавления в C за один проход;
}
```

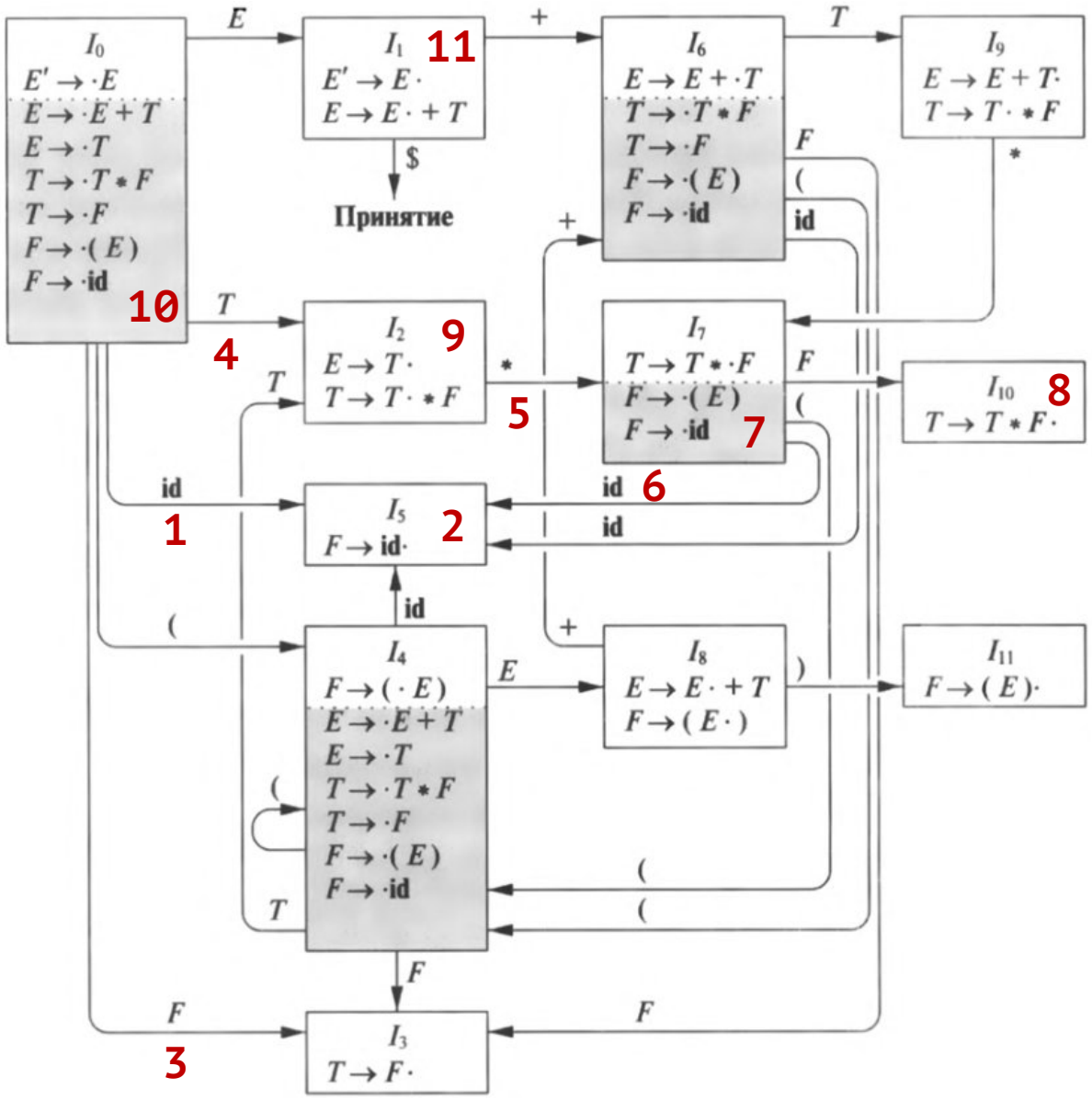
Канонический набор C для LR(0)-автомата



Применение LR(0)-автомата

- Для заданной грамматики строится LR (0)-автомат
- Все состояния являются принимающими
- Под состоянием *j* подразумевается состояние, соответствующее множеству пунктов I_j
- **Как LR(0)-автомат помогает в принятии решения «перенос/свертка»?**
- Пусть строка γ из символов грамматики переводит LR(0)-автомат из состояния 0 в состояние *j*
- Если состояние *j* имеет переход для данного символа *a*, то выполним *перенос (shift)* входного символа *a*
- В противном случае выбирается *свертка (reduce)*; пункт в состоянии *j* говорит какую продукцию следует использовать

Пример применения LR(0)-автомата



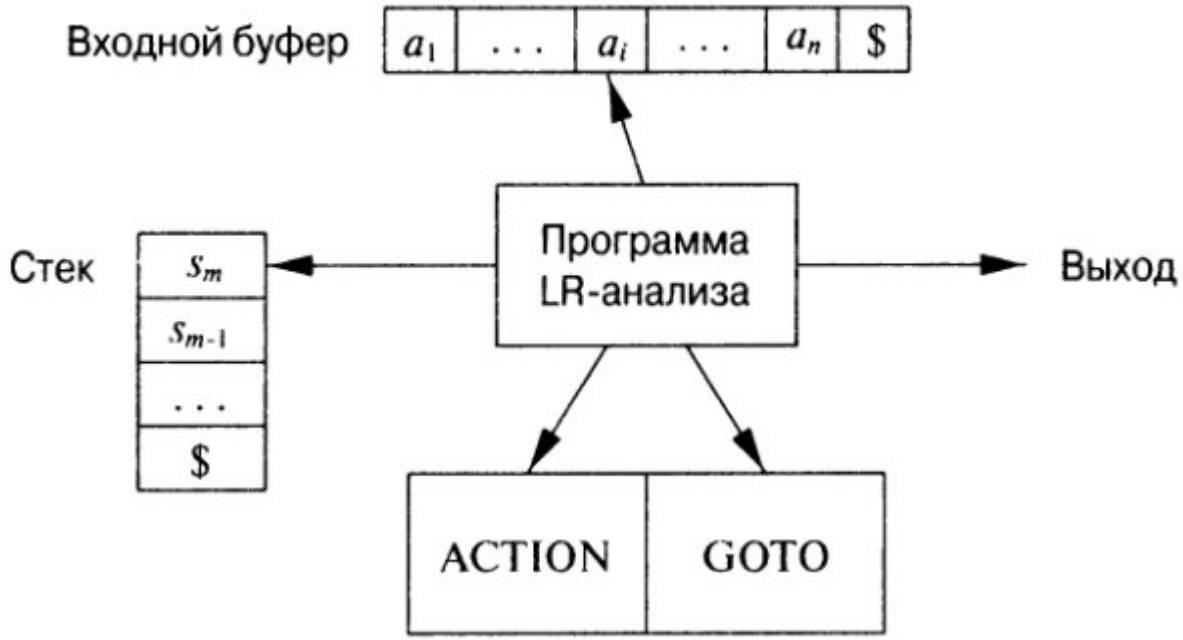
- Шаги синтаксического анализатора методом «перенос/свертка» с использованием LR(0)-автомата для входной строки

  `id * id`

- Стек хранит состояния

Строка	Стек	Символы	Вход	Действие
(1)	0	\$	**id** $*$ **id**\$	Перенос в 5
(2)	0 5	\$**id**	$*$**id**\$	Свертка по $F \to$ **id**
(3)	0 3	\$$F$	$*$**id**\$	Свертка по $T \to F$
(4)	0 2	\$$T$	$*$**id**\$	Перенос в 7
(5)	0 2 7	\$$T*$	**id**\$	Перенос в 5
(6)	0 2 7 5	\$$T*$**id**	\$	Свертка по $F \to$ **id**
(7)	0 2 7 10	\$$T*F$	\$	Свертка по $T \to T * F$
(8)	0 2	\$$T$	\$	Свертка по $E \to T$
(9)	0 1	\$$E$	\$	Принятие

Алгоритм LR-анализа (LR-Parsing Algorithm)



- **Входной буфер**: анализатор читает по одному символу предпросмотра
- **Стек**: В случае методов SLR, LR, LALR стек хранит состояния LR(0)-автомата
- **Таблица синтаксического анализа** (parsing table): функции действий синтаксического анализа ACTION и функции переходов GOTO

Структура таблицы LR-анализа (LR parsing table)

- **Функция** ACTION[*i*, *a*]:
  - *i* — состояние и терминал *a* (или маркер $ конца входной строки)
- **Значения** ACTION[*i*, *a*]:
  - **Перенос** *j* (shift): перено входного символ *a* в стек, но для представления *a* используется состояние *j*
  - **Свертка** A —> β (reduce): свёртка β на вершине стека в заголовок *A*
  - **Принятие** (accept): анализатор принимает входную строку и завершает анализ
  - **Ошибка** (error): синтаксический анализатор обнаруживает ошибку во входной строке и предпринимает корректирующее действие

- **Расширенная функция GOTO**, определенная на множествах пунктов, распространяется на состояния: если GOTO[I_i, A] = I_j, то GOTO отображает также состояние *i* и нетерминал *A* на состояние *j*

Алгоритм LR-анализа

- **Вход**: входная строка *w* и таблица LR-анализа с функциями ACTION и GOTO для грамматики *G*
- **Выход**: если $w \in L(G)$, шаги сверток восходящего синтаксического анализа *w*, в противном случае — сообщение об ошибке
- **Начальное состояние**: в стеке начальное состояние s_0, а во входном буфере *w*$

```
Пусть a — первый символ w$.
while(1) { /* Бесконечный цикл */
    Пусть s — состояние на вершине стека.
    if ( ACTION[s, a] = перенос t ) {
        Внести t в стек.
        Присвоить a очередной входной символ.
    } else if ( ACTION[s, a] = свертка A → β ) {
        Снять |β| символов со стека.
        Пусть теперь на вершине стека находится состояние t.
        Внести в стек GOTO[t, A].
        Вывести продукцию A → β.
    } else if ( ACTION[s, a] = принятие ) {
        break; /* Синтаксический анализ завершен */
    } else Вызов подпрограммы восстановления после ошибки.
}
```

Пример таблицы для LR-анализа

Грамматика для выражений:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Продукции:

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$

- si — перенос и размещение в стеке состояния i
- rj —свертка в соответствии с продукцией с номером j
- acc — принятие
- пустое поле — ошибка

Таблица синтаксического анализа (функции ACTION и GOTO)

Состояние	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Пример действий в ходе LR-анализа

Действия LR-анализатора для строки **id * id + id**

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift
(2)	0 5	**id**	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \to F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \to T * F$
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \to T$
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \to E + T$
(14)	0 1	E	$\$$	accept

Методы LR-анализа

- **LR** // Donald Knuth, 1965, «On the translation of languages from left to right»
- **SLR** (Simple LR)
  // Frank DeRemer, 1969, PhD dissertation
- **LALR** (lookahead LR, lookahead, left-to-right, rightmost derivation parser)
  // Frank DeRemer, 1969, PhD dissertation «Practical Translators for LR(k) languages»
  https://web.archive.org/web/20130819012838/http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-065.pdf

Сравнительные характеристики реальных языков программирования

Язык	Количество			
	Токены	Продукций (непустых)	Терминалы	Ключевые слова
Algol-60	890	102(90)	88	24
Pascal	1004	109(85)	84	35
Modula-2	887	70(69)	88	39
Ada 95	2999	327(258)	98	69
Turbo Pascal 6.0	1479	141(117)	89	55
Delphi 7.0	2041	186(165)	92	107
C (K&R)	913	52(49)	122	27
C99	1413	110(106)	133	37
C++ (Страуструп, 1990)	1654	124(117)	131	48
C++ (ISO/IEC 14882-1998)	2292	176(166)	136	63
Java	1813	172(158)	121	48
C#	3036	295(268)	115	88

Использование неоднозначных грамматик

- **Неоднозначная грамматика не может быть LR-грамматикой и, следовательно, не может относиться ни к одному из рассмотренных ранее классов грамматик**
- Ряд типов неоднозначных грамматик, однако, вполне пригоден для определения и реализации языков
- Для некоторых неоднозначных грамматик задаются специальные правила разрешения неоднозначности, обеспечивающие для каждого предложения только одно дерево разбора

- **Использование приоритетов и ассоциативности для разрешения конфликтов**
- Грамматики для выражений с операторами + и *

Неоднозначная грамматика

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Однозначная грамматика

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

Плюсы использования неоднозначной грамматики:

1. Можно заменить ассоциативность и уровень приоритета операторов + и * без изменения продукций или количества состояний получающегося синтаксического анализатора
2. Анализатор для однозначной грамматики будет тратить много времени на свертку по продукциям *E* —> *T* и *T* —> *F*

Неоднозначность «висящего else» (dangling-else ambiguity)

- **Неоднозначная грамматика не может быть LR-грамматикой и, следовательно, не может относиться ни к одному из рассмотренных ранее классов грамматик**
- Ряд типов неоднозначных грамматик, однако, вполне пригоден для определения и реализации языков
- Для некоторых неоднозначных грамматик задаются специальные правила разрешения неоднозначности, обеспечивающие для каждого предложения только одно дерево разбора

- **Использование приоритетов и ассоциативности для разрешения конфликтов**
- Грамматики для выражений с операторами + и *

Неоднозначная грамматика

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$$

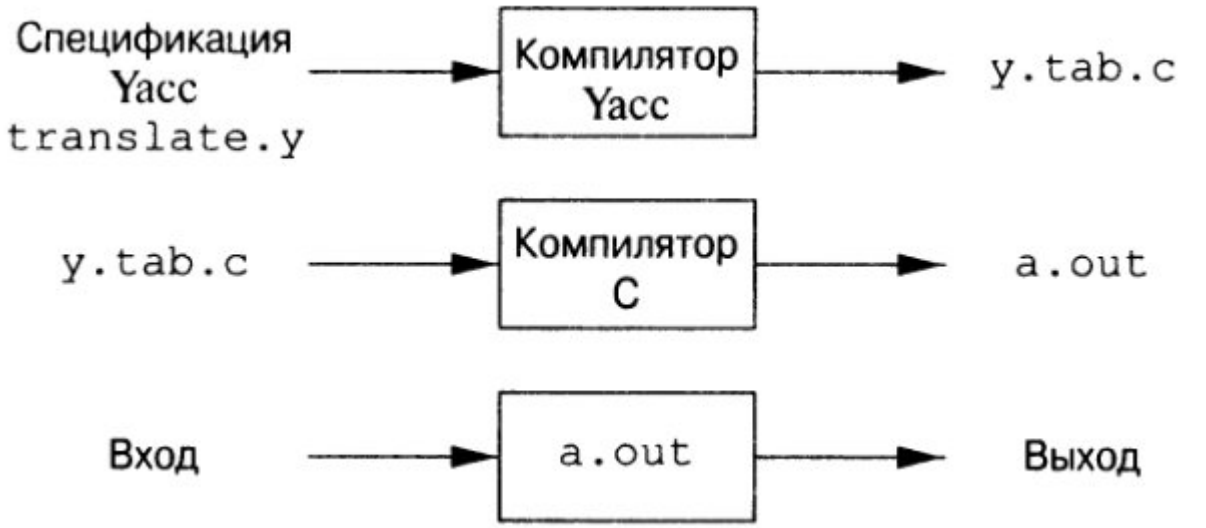
Однозначная грамматика

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned}$$

Плюсы использования неоднозначной грамматики:

1. Можно заменить ассоциативность и уровень приоритета операторов + и * без изменения продукций или количества состояний получающегося синтаксического анализатора
2. Анализатор для однозначной грамматики будет тратить много времени на свертку по продукциям *E* —> *T* и *T* —> *F*

Генератор синтаксических анализаторов Yacc



Преобразует файл translate. y в программу y. tab. c на языке C с использованием LALR-метода

Использование Yacc с неоднозначной грамматикой

- Конфликт «свертка/свертка» разрешается выбором продукции, находящейся первой в спецификациях Yacc
- Конфликт «перенос/свертка» разрешается в пользу переноса, правило корректно разрешает конфликт, возникающий из-за неоднозначности «висящего else»

- Yacc обеспечивает общий механизм для разрешения конфликтов — в части объявлений терминалам можно назначить приоритеты и ассоциативность
- Токены получают уровни приоритетов в том порядке, в котором они встречаются в части объявлений, начиная с наинизшего

```
%left '+' '-'
%right '^'
%nonassoc '<'
```

- Обычно приоритет продукции устанавливается таким же, как у ее крайнего справа терминала

```
%prec (терминал)
```