iSAQB Advanced DSL - Syntax

Michael Sperber

Created: 2024-07-03 Wed 10:26

Lexical Analysis

Read the following program out loud:

```
public class Main {
  public static void main(String[] args) {
    // This is a comment
    System.out.println("Hello World");
  }
}
```

Most Programming Languages

- programs divided into **lexemes** ("words")
- spaces and line breaks often mostly irrelevant
- comments are irrelevant
- `"Hello World"` is a single unit
- `public`, `class` etc. are keywords - built into the language

Lexical Analysis

```
<public> <class> <identifier [Main]> <{>
<public> <static> <void> <identifier [main]>
<open paren> <identifier [String]> <open bracket> <closed bracket>
<identifier [args]> <closed paren>
<identifier [System]> <dot> <identifier [out]> <dot> <identifier [p
<open paren> <string [Hello World]> <closed paren> <semicolon>
<closed brace>
<closed brace>
```

Lexemes and Tokens

- the letter sequence `"Hello World"` is a **lexeme**
- its classification as a **string literal** is its **token**
- the text of the string literal `Hello World` is its **attribute**

(LG 1-2) Lexical Analysis

Let Σ be the alphabet of a programming language, T a finite set of **tokens**, and A an arbitrary set of **attributes**.

A **tokenizer** is a function

$$\mathit{tokenize} : \Sigma^* \to (T \times A)^*$$

such that there is a function

$$\mathit{untokenize} : (T \times A)^* \to \Sigma^*$$

Lexical Analysis

... with the following properties:

1. $\mathit{tokenize} \circ \mathit{untokenize} = \mathit{id}_{(T \times A)^*}$ and
2. there is a function $\mathit{untoken} : (T \times A) \to \Sigma^*$ so that

$$\mathit{untokenize}(t_1 t_2 \ldots) = \mathit{untoken}(t_1)\mathit{untoken}(t_2)\ldots$$

($\mathit{untokenize}$ is a **homomorphism**).

Regular Expressions

$RE(\Sigma)$ is the smallest set with:

- $\underline{\emptyset} \in RE(\Sigma)$
- $\underline{\varepsilon} \in RE(\Sigma)$
- if $a \in \Sigma$ then $\underline{\mathtt{a}} \in RE(\Sigma)$
- if $r_1, r_2 \in RE(\Sigma)$ then $r_1 r_2 \in RE(\Sigma)$
- if $r_1, r_2 \in RE(\Sigma)$ then $r_1 \mid r_2 \in RE(\Sigma)$
- if $r \in RE(\Sigma)$ then $r^* \in RE(\Sigma)$.

Language Defined by a Regular Expression

$$L : RE(\Sigma) \rightarrow \mathcal{P}(\Sigma^*)$$

$$
\begin{array}{lcl}
L(\underline{\emptyset}) & = & \emptyset \\
L(\underline{\varepsilon}) & = & \{\varepsilon\} \\
L(\underline{\mathtt{a}}) & = & \{a\} \\
L(r_1 r_2) & = & L(r_1) \cdot L(r_2) \\
 & := & \{w_1 w_2 \mid w_1 \in L(r_1), w_2 \in L(r_2)\} \\
L(r_1 \mid r_2) & = & L(r_1) \cup L(r_2) \\
L(r^*) & = & L(r)^* \\
 & := & \{w_1 w_2 \ldots w_n \mid n \in \mathbb{N}, w_i \in L(r)\} \\
 & = & \{\varepsilon\} \cup L(r) \cup L(r) \cdot L(r) \cup L(r) \cdot L(r) \cdot L(r) \cup \ldots
\end{array}
$$

Lexer Implementation Strategies

- translate to NFA, DFA (Dragon book)
- derivatives (Matt Might: A regular expression matcher in Scheme using derivatives)
- issue: keywords may cause automaton explosion

Datalog Lexer in Racket

```
#lang racket/base
(require parser-tools/lex
         (prefix-in : parser-tools/lex-sre))

(define-tokens dtokens
  (VARIABLE IDENTIFIER STRING))
(define-empty-tokens dpunct
  (LPAREN COMMA RPAREN TSTILE DOT EQUAL NEQUAL TILDE QMARK EOF))
(define-lex-abbrev line-break #\newline)
(define-lex-abbrev id-chars
  (char-complement (char-set "(,)=:.~?\"% \n")))
(define-lex-abbrev variable-re
  (:: upper-case
      (:* (:or upper-case lower-case (char-set "0123456789_")))))
(define-lex-abbrev identifier-re
  (:: id-chars (:* (:or upper-case id-chars))))
(define-lex-abbrev comment-re
  (:: "%"
      (complement (:: any-string line-break any-string))
      line-break))
```


https://github.com/racket/datalog

Datalog Lexer in Racket

```
(define get-string-token
  (lexer
   [(eof) (error 'datalog-lexer "Unterminated string")]
   [(:~ #\" #\\ #\newline)
    (cons (car (string->list lexeme))
          (get-string-token input-port))]
   [(:: #\\ #\\)
    (cons #\\ (get-string-token input-port))]
   [(:: #\\ #\newline)
    (cons #\newline (get-string-token input-port))]
   [(:: #\\ #\")
    (cons #\" (get-string-token input-port))]
   [#\" null]))
```

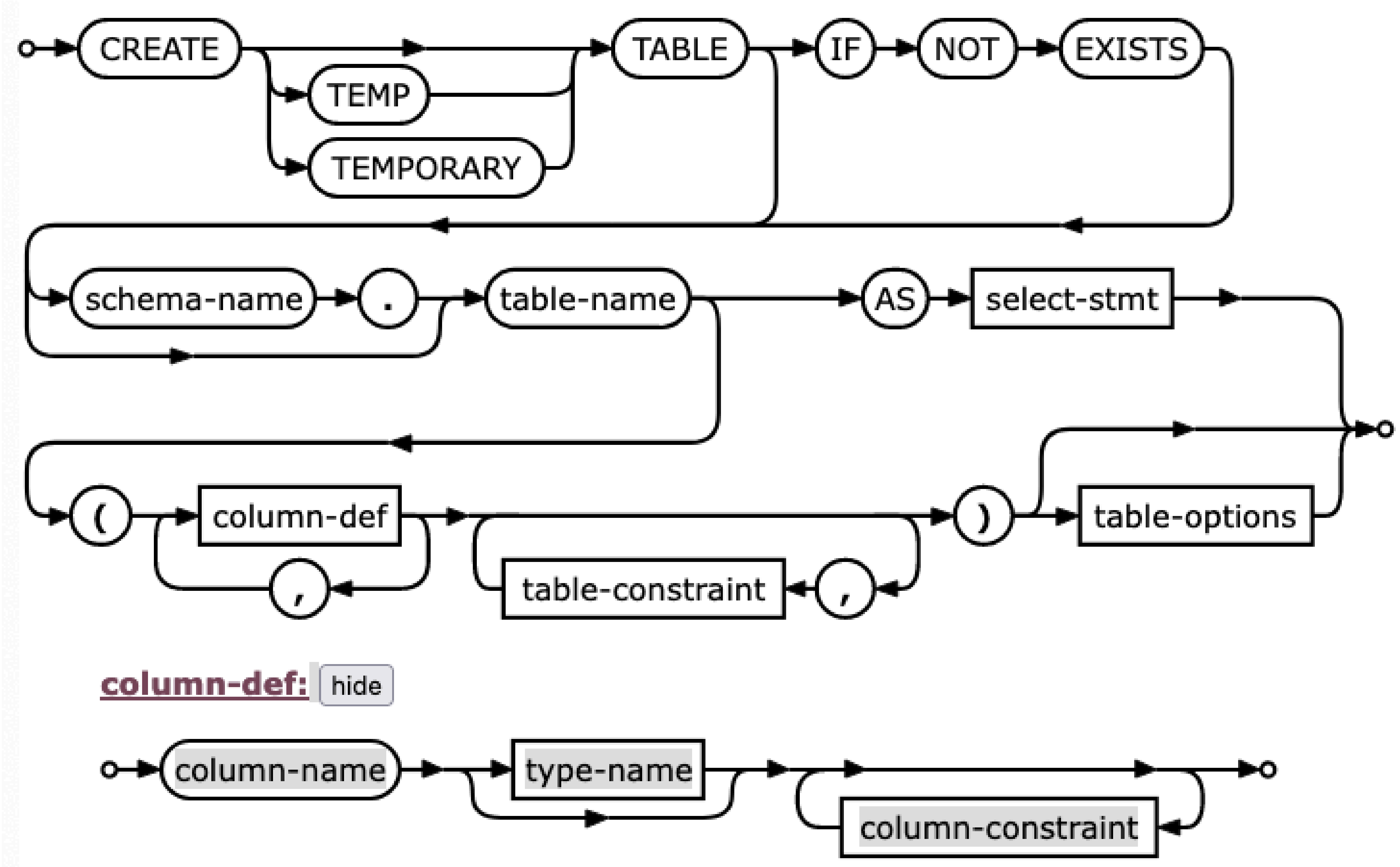
Datalog Lexer in Racket

```
(define dlexer
  (lexer-src-pos
   [whitespace
    (return-without-pos (dlexer input-port))]
   [comment-re
    (return-without-pos (dlexer input-port))]
   [variable-re
    (token-VARIABLE lexeme)]
   [identifier-re
    (token-IDENTIFIER lexeme)]
   [":-" (token-TSTILE)]
   [#\" (token-STRING (list->string (get-string-token input-port)))
   [#\( (token-LPAREN)]
   [#\, (token-COMMA)]
   [#\) (token-RPAREN)]
   [#\. (token-DOT)]
   [#\~ (token-TILDE)]
   [#\? (token-QMARK)]
   [#\= (token-EQUAL)]
   ["!=" (token-NEQUAL)]
   [(eof) (token-EOF)]))
```

(LG 2-2) Limits of Regular Expressions

- no matched parens / brackets / braces
- no recursive definitions

Syntactic Structure



SQLite Syntax Documentation

Context-Free Grammar

A **context-free grammar** is a tuple $G = (N, T, P, S)$. N is the set of **nonterminals**, T the set of **terminals**, $S \in N$ the **start symbol**, $V = T \cup N$ the set of **grammar symbols**. P is the set of **productions**; productions have the form $A \to \alpha$ for a nonterminal A and a sequence α of grammar symbols.

ϵ is the empty sequence; $|\xi|$ is the length of sequence ξ. Furthermore, α^k denotes a sequence with k copies of α, and $\xi_{|k}$ is the sequence consisting of the first k terminals in ξ.

(LG 2-2) Chomsky Hierarchy

Type	languages	productions
Type-3	regular	$A \to a, A \to aB, A \to Ba$
Type-2	context-free	$A \to \alpha$
Type-1	context-sensitive	$\alpha A \beta \to \alpha \gamma \beta$
Type-0	recursively enumerable	$\gamma \to \alpha$

Conventions

Some letters denote elements of certain sets by default:

$$
\begin{aligned}
A, B, C, E &\in N \\
\xi, \rho, \tau &\in T^* \\
x, y, z &\in T \\
\alpha, \beta, \gamma, \delta, \nu, \mu &\in V^* \\
X, Y, Z &\in V
\end{aligned}
$$

All grammar rules in the text are implicitly elements of P.

Derives Relation

G induces the **derives relation** $\Rightarrow$ on V^* with

$$\alpha \Rightarrow \beta \;:\Leftrightarrow\; \alpha = \delta A \gamma \;\wedge\; \beta = \delta \mu \gamma \;\wedge\; A \rightarrow \mu$$

and $\stackrel{*}{\Rightarrow}$ denotes the reflexive and transitive closure of $\Rightarrow$.

A **derivation** from α_0 to α_n is a sequence $\alpha_0, \alpha_1, \ldots, \alpha_n$ where $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \leq i \leq n$.

A **sentential form** is a sequence appearing in a derivation beginning with the start symbol.

(LG 1-2) Parsing

$$[X] : T^* \to \mathcal{P}(T^*)$$

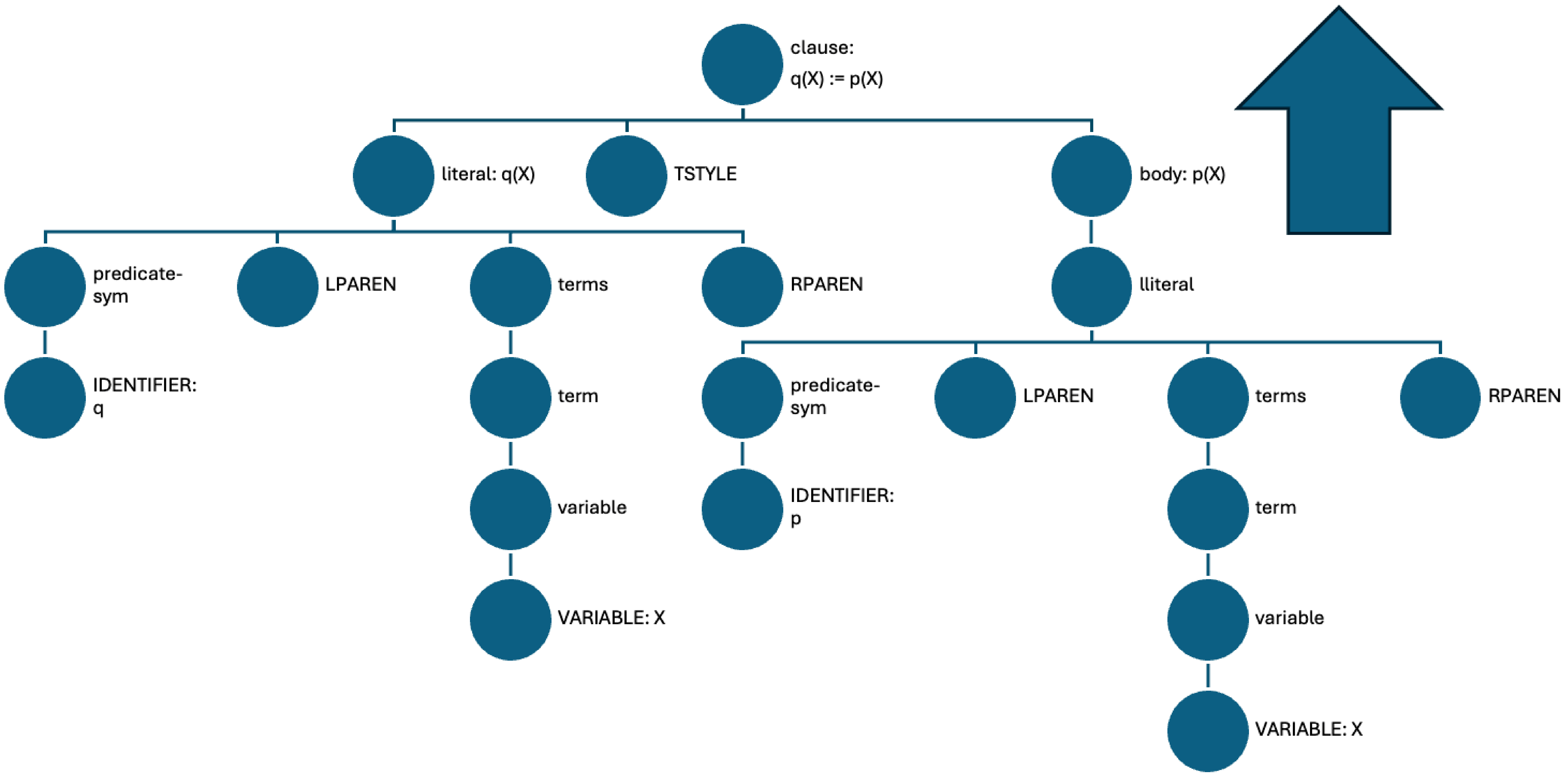
$$[X](x_1 \ldots x_n) = \{x_{k+1} \ldots x_n \mid X \stackrel{*}{\Rightarrow} x_1 \ldots x_k\}$$

ξ is in the language defined by the grammar iff $\epsilon \in [S](\xi)$.

Datalog Parser

```
(define-values
  (program-parser statement-parser clause-parser literal-parser)
  (apply
   values
   (parser
    (start program statement clause literal)
    (end EOF)
    (tokens dtokens dpunct)
    (src-pos) (error ...)
    (grammar
     (program [(statements) $1])
     (statements [() empty]
                 [(statement statements) (list* $1 $2)])
     (statement [(assertion) $1]
                [(query) $1]
                [(retraction) $1]
                [(requirement) $1])
     ...
     (requirement [(LPAREN IDENTIFIER RPAREN DOT)
                   (make-requirement (make-srcloc $1-start-pos $4-e
     ...)))
```

Attribute Grammar

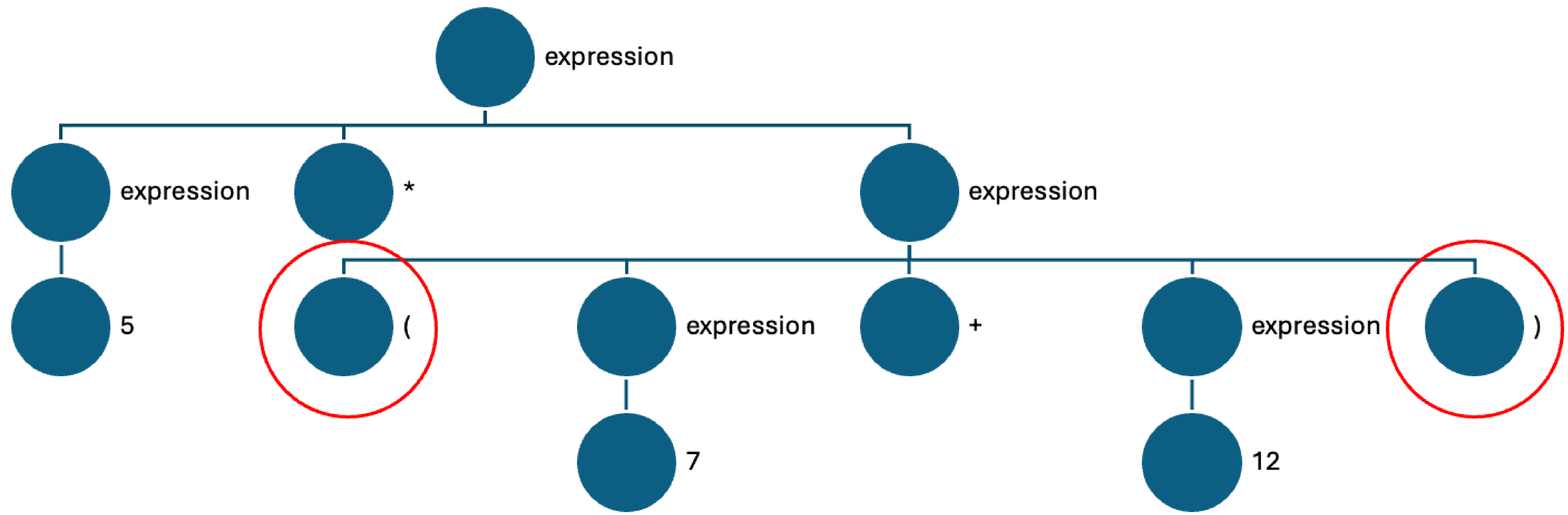


(LG 2-3) Datalog Abstract Syntax

```
(struct node (srcloc))

(struct assertion node (clause))
(struct retraction node (clause))
(struct query node (question))
(struct requirement node (lib))
...
```

(LG 1-2, LG 2-3) Abstract Syntax



(LG 2-3) Abstract Syntax

Let Σ be the alphabet of the language of a program. **Abstract syntax generation** is a function

$$parse : \Sigma^* \to D$$

where D is a suitable set such that a function

$$unparse : D \to \Sigma^*$$

exists with

$$parse \circ unparse = id_D.$$

(LG 2-3) Exercise: Abstract Syntax for Tim

Design an abstract syntax for Tim and implement it in Racket!

"Parser"

$$[x](x_1 \ldots x_n) = \{x_2 \ldots x_n\} \text{ if } x_1 = x, \varnothing \text{ otherwise}$$

$$\begin{aligned}
[\epsilon](\xi) &= \{\xi\} \\
[X\alpha](\xi) &= \bigcup\{[\alpha](\rho) \mid \rho \in [X](\xi)\} \\
[A](\xi) &= \bigcup_{A \to \alpha}[\alpha](\xi)
\end{aligned}$$

- non-deterministic
- doesn't work for left recursion

Lookahead

$$
\begin{aligned}
\mathrm{first}_k &: V^* \to T^k \\
\mathrm{first}_k(\alpha) &:= \{\xi_{|k} \mid \alpha \overset{*}{\Rightarrow} \xi\} \\
\mathrm{follow}_k &: N \to T^k \\
\mathrm{follow}_k(A) &:= \{\xi \mid S \overset{*}{\Rightarrow} \beta A \gamma \;\wedge\; \xi \in \mathrm{first}_k(\gamma)\}
\end{aligned}
$$

Recursive-Descent Parser

$$[A](\xi) = \bigcup_{A \to \alpha, \xi_{|k} \in (\mathrm{first}_k(\alpha)\ \mathrm{follow}_k(A))_{|k}} [\alpha](\xi)$$

AKA *top-down* parser

(LG 2-2) LL(k) Grammars

The **LL(k) lookahead** of a production **A→α** is computed as follows:

$$\mathrm{LLLA}_k(A \to \alpha) := (\mathrm{first}_k(\alpha)\,\mathrm{follow}_k(A))_{|k}$$

A context-free grammar G is **LL(k)** if, for productions $A \to \alpha$ and $A \to \beta$ with $\alpha \neq \beta$,

$$\mathrm{LLLA}_k(A \to \alpha) \cap \mathrm{LLLA}_k(A \to \beta) = \varnothing.$$

Caveats of LL/Recursive Descent Parsing

- decision on production is made on the **left**
- no left recursion

(LG 2-1) Exercise: Tim Grammar und Parser

Design a grammar for the Tim language, along with a reursive-descent parser for it in Racket.

(LG 2-2) LR Parsing

- also works from left to right
- ... but only decides on a production at its **right** edge, i.e. *later*
- ... and therefore works on a larger class of grammars

LR Parsing

An **LR(k)** item} (or just **item**) is a triple consisting of a production, a position within its right-hand side, and a terminal string of length k - -the **lookahead**.

An item is written as $A \to \alpha \cdot \beta\ (\rho)$ where the dot indicates the position, and ρ is the lookahead.

If the lookahead is not used (or $k = 0$), it is omitted.

A **kernel item** has the form $A \to \alpha \cdot \beta\ (\rho)$ with $|\alpha| > 0$.

A **predict item** has the form $A \to \cdot\alpha\ (\rho)$.

An **LR(k) state** (or just **state**) is a non-empty set of LR(k) items.

iSAQB International Software Architecture Qualification Board

@ctive group

LR Parsing

Each state q has an associated set of **predict items**:

$$\text{predict}(q) := \left\{ B \rightarrow \cdot\nu\ (\tau) \;\middle|\; \begin{array}{r} A \rightarrow \alpha \cdot \beta\ (\rho) \Downarrow^{+} B \rightarrow \cdot\nu\ (\tau) \\ \text{for } A \rightarrow \alpha \cdot \beta\ (\rho) \in q \end{array} \right\}$$

where $\Downarrow^{+}$ is the transitive closure of the relation $\Downarrow$ defined by

$$A \rightarrow \alpha \cdot B\beta\ (\rho) \Downarrow B \rightarrow \cdot\delta\ (\tau) \text{ for all } \tau \in \text{first}_k(\beta\rho)$$

The union of q and $\text{predict}(q)$ is called the **closure** of q:

$$\overline{q} := q \cup \text{predict}(q)$$

denotes the closure of a state q.

LR Parsing

For a state q and a grammar symbol X:

$$\text{goto}(q, X) := \{A \rightarrow \alpha X \cdot \beta\ (\rho) \mid A \rightarrow \alpha \cdot X\beta\ (\rho) \in \overline{q}\}$$

$$\text{nextterm}(q) := \{x \mid A \rightarrow \alpha \cdot x\beta \in \overline{q}\}$$

$$\text{nactive}(q) := \max\{|\alpha| : A \rightarrow \alpha \cdot \beta \in q\}$$

Recursive-Ascent Parsing

$$[q](\xi, c_1, \ldots, c_{\mathrm{nactive}(q)}) :=$$

$$\textbf{letrec } c_0(X, \xi) = [\mathrm{goto}(q, X)](\xi, c_0, c_1, \ldots, c_{\mathrm{nactive}(\mathrm{goto}(q,X))-1})$$

$$\textbf{in}$$

$$A \to \alpha \cdot \ (\rho) \in \overline{q} \wedge \xi_{|k} = \rho \rhd c_{|\alpha|}(A, \xi) \text{ (reduce)}$$

$$\xi = x\xi' \wedge x \in \mathrm{nextterm}(q) \rhd c_0(x, \xi') \text{ (shift)}$$

(LG 5-1) Exercise: Tim Grammar und Parser

Rewrite the grammar using `parser-tools/yacc`, and generate abstract syntax.

(LG 2-3, LG 5-2, LG 5-4) Exercise: Xtext DSL

Implement the grammar for the previous exercise in Xtext, and evolve it into a DSL implementation.

(LG 2-3, LG 5-4) Exercise: MPS DSL

Design a projectional representation for the syntax from the previous exercise, and evolve it into a DSL implementation.