Parsing Abstract Strings

```
String sql = "SELECT * FROM people "
    + "WHERE (age > 18)) ORDER BY name";
connection.prepareStatement(sql);
```


Andrey Breslav

University of Tartu / STACC

March 9th, 2010

Outline

- String-Embedded DSLs
- Abstract Strings
  - Three levels of abstraction
- Lexical analysis
  - Finite Automata
  - Finite-State Transducers
- Syntactical Analysis
  - Regular Approximation
  - Abstract Parsing
- Summary
- References

String-Embedded DSLs

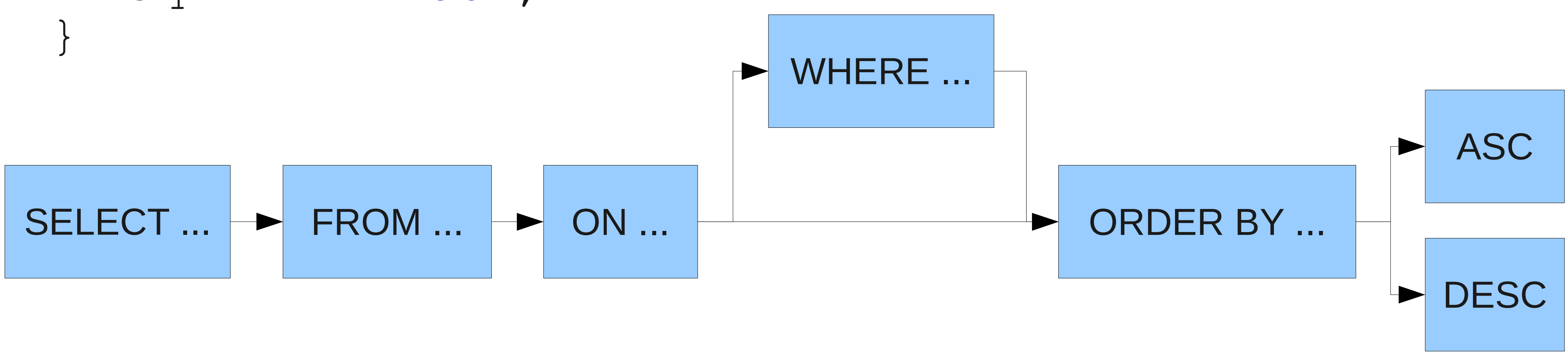
```
String sql = “SELECT name, age “ +
             ”FROM tab LEFT JOIN tab1 “ +
             “ON (tab.id = tab1.id) ”;
if (isFiltering()) {
   sql += “WHERE age >= 18 “;
}
sql += “ORDER BY age “;
if (isAscending()) {
   sql += “ASC”;
} else {
   sql += “DESC”;
}
Connection connection = connect();
connection.prepareStatement(sql);
```

Hotspot

Abstract Strings

```
String sql = “SELECT name, age “ +
             ”FROM tab LEFT JOIN tab1 “ +
             “ON (tab.id = tab1.id) ”;
if (...) {
   sql += “WHERE age >= 18 “;
}
sql += “ORDER BY age “;
if (...) {
   sql += “ASC”;
} else {
   sql += “DESC”;
}
```

Abstraction



Levels of Abstraction

Type 0: Arbitrary Turing machines

Context-Sensitive: Linear-bounded automata

Context-Free: Nondeterministic pushdown automata

$O(N^3)$, but usually **$O(N)$**

Regular: Finite automata (regular expressions)

Finite: Finite set of strings

$O(N)$

Program Constructs

- Finite
  - String Literals: `“SELECT * FROM t”`
  - Concatenation: `sql + “ WHERE x > 10”`
  - Conditionals: `if (b) {s+=“ASC”;} else {s+=“DESC”;}`
- Regular
  - Appending in a loop
    - ```
      for (String s : items) {
         buffer.append(“, “ + s);
      }
      ```
  - Appending in (effectively) tail recursion
- Context-Free
  - General loops and recursion
      ```

I am cheating!

Do you actually believe that

Arbitrary programs can generate ONLY context-free languages ?!

Please, reconsider this belief!

Problem Statement

- Input
  - Program P, with a hotspot E
    - E may have a value from a set of strings L(E)
  - **Regular** grammar Lex
    - Describes lexical structure of the embedded language (e.g. SQL)
  - **Context-Free** grammar G (over tokens produced by Lex)
- Output
  - OK – no errors found
  - ERROR(List of errors)

Solution Overview

- Find L(P)
  - Finite/Regular
  - Context-Free
- Check if L(P) $\subseteq$ L(Lex)*
  - Compute T := Lex(L(P)) – language of token sequences
    - Finite/Regular
    - Context-Free
- Check if T $\subseteq$ L(G)
  - REG-REG – Decidable
  - REG-CF – Undecidable

Precision

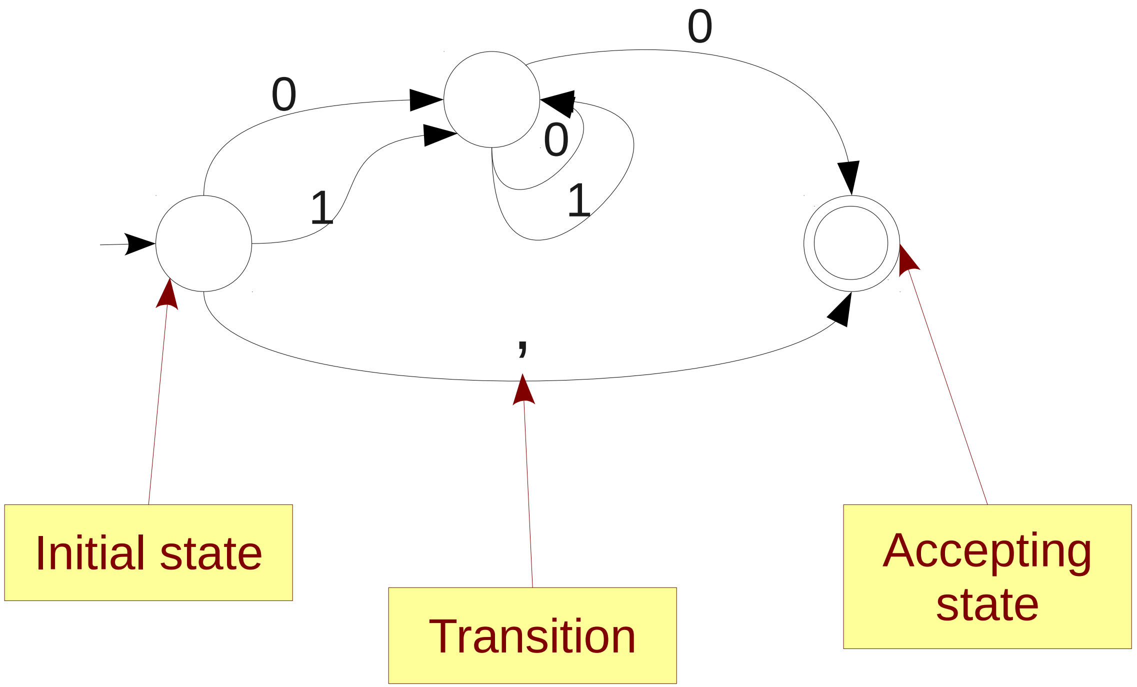
- Soundness
  - If we return OK, then there can be no errors when we run the program
- Completeness
  - If we return ERROR(...), then there will be some errors when we run the program
- Bad news (Rise's theorem):
  - We can not achieve completeness and soundness for unrestricted programs

Regular Input

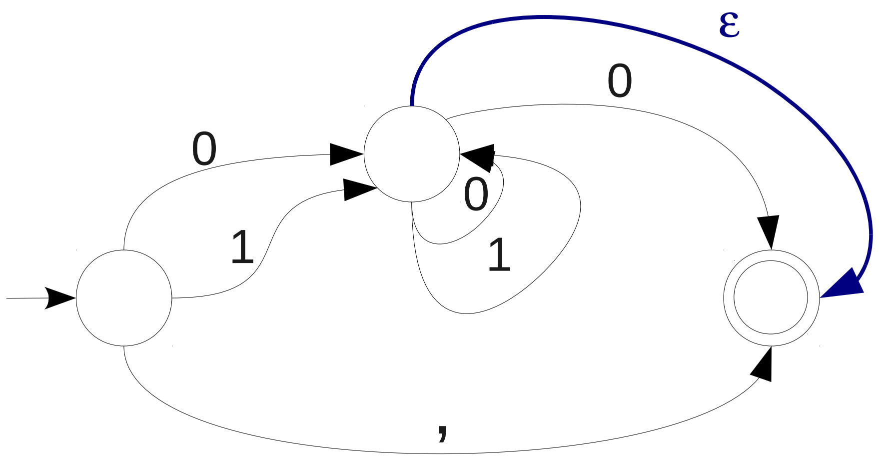
- L(E) is represented as a **finite automaton** A
- Lexical analyzer is represented as a **finite transducer** T
- TOK := T(A) is also a **finite automaton**
- **Problem**: is the given **regular language** a subset of a given **CF-language** (e.g. SQL)?
  - **AKA “Language Inclusion 3-2”, undecidable**
    - **We will use some approximation**

Finite Automata (FA)

- Regular expressions
  - , | [01]+0
- No loops => finite language
- Recognizing
  - A :: String -> Bool
- Generating
  - A :: [String]

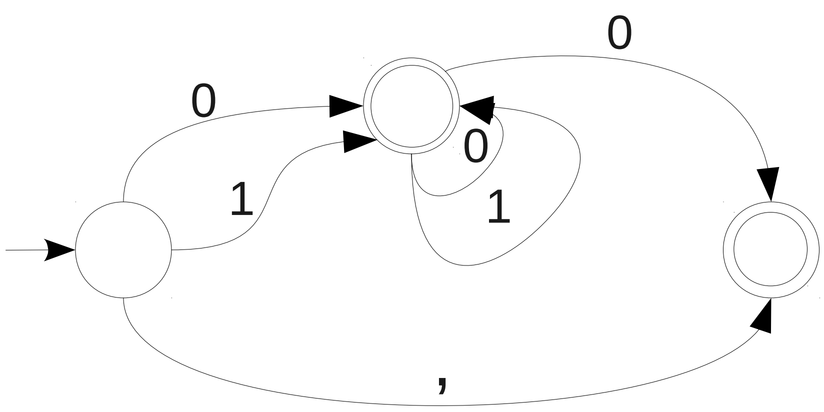


Empty Transitions



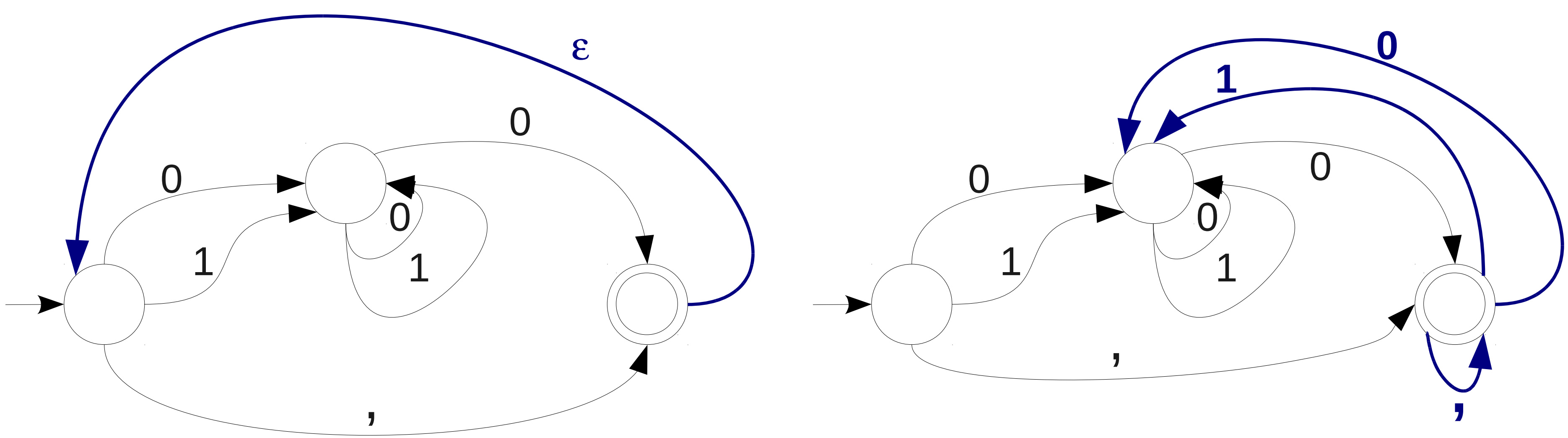
, | [01]+ 0?

ε–Transitions can always be eliminated:



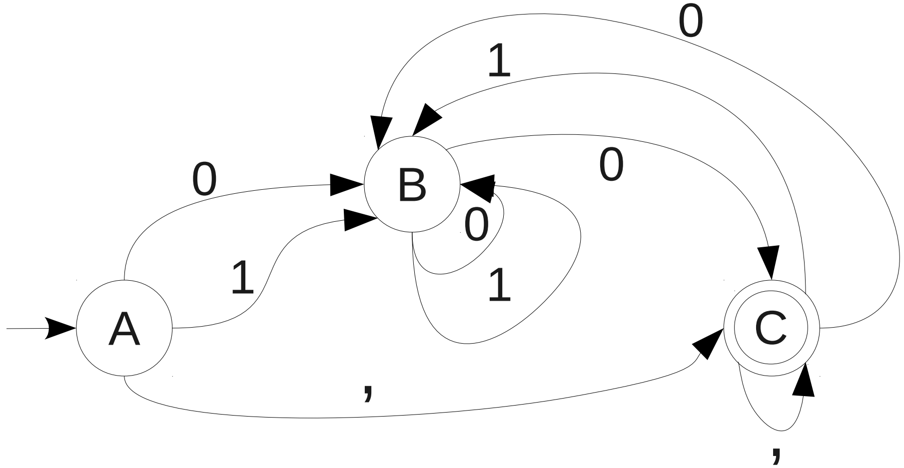
Recognizing Token Streams

- From Lex to Lex+
- For every accepting state A
  - Add an ε-transition from A to the initial state
- Eliminate all ε-transitions

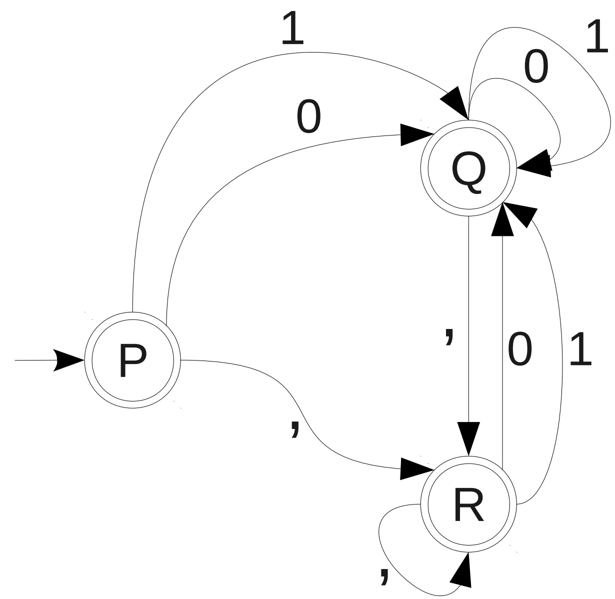


(, | [01]+ 0**)+**

Inclusion for Regular Languages



$\subseteq$



Generator: (, | [01]+ 0)+

Recognizer: ([01]+ | ,)*

$\texttt{S :: State}^{G}\texttt{ -> [State}^{R}\texttt{]}$
$\texttt{T :: Transition}^{G}\texttt{ -> [Transition}^{R}\texttt{]}$

We start from

$\texttt{S = \{Init}^{G}\texttt{ |-> [Init}^{R}\texttt{]\}}$
$\texttt{T = \{\}}$

And compute S and T until a fixpoint

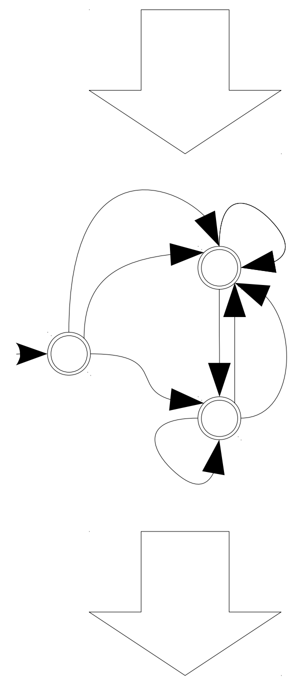
Algorithm

- Being in state X^G
  - For all transitions X^G -c-> Y^G
    - For all states X^R <- $S(X^G)$
      - Find a transition X^R -c-> Y^R
        - If no such transition exists, abort and return **NO**
      - Add it to $T(X^G)$
      - Add Y^R to $S(Y^G)$
        - If Y^G is accepting and Y^R is not, abort and return **NO**
    - If S or T has changed, recursive call from Y^G
  - Return **YES**
- Why a **fixpoint** will be reached eventually?
  - We only **add** to both maps
  - Sets of states and transitions are **finite**
- Time complexity:
  - $O(|States^G|*|States^R| + |Transitions^G|*|Transitions^R|)$

The Nature of Lexical Analysis

SELECT name FROM people WHERE people.age >= 18

Unicode alphabet

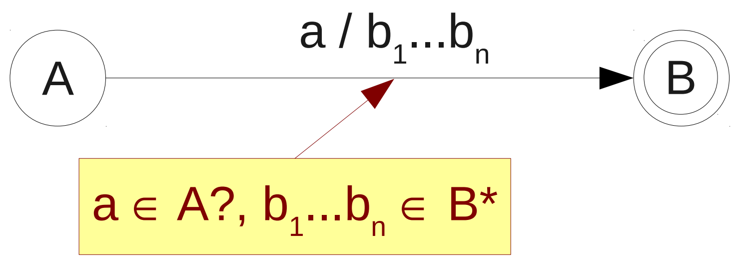


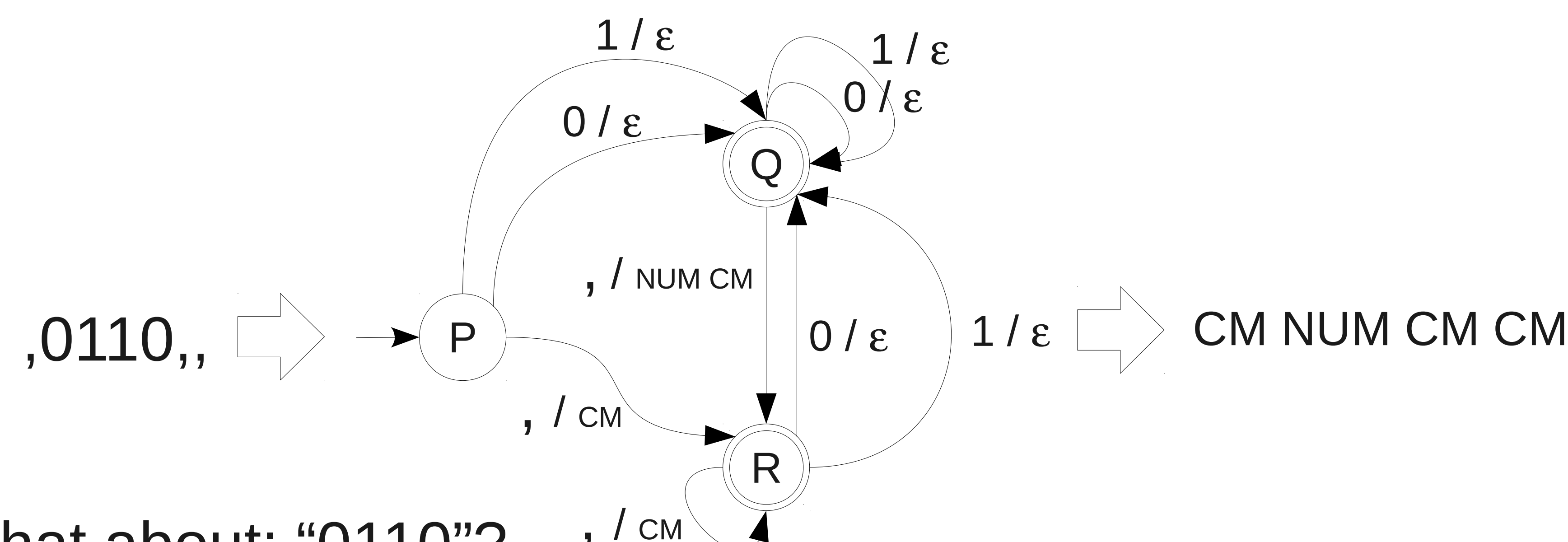
Tokens alphabet, includes **EOF**

SELECT WS ID(name) WS **FROM** ID(people) WS
WHERE WS ID(people) **DOT** ID(age) WS **GE** NUM(18) **EOF**

Finite-State Transducers (FST)

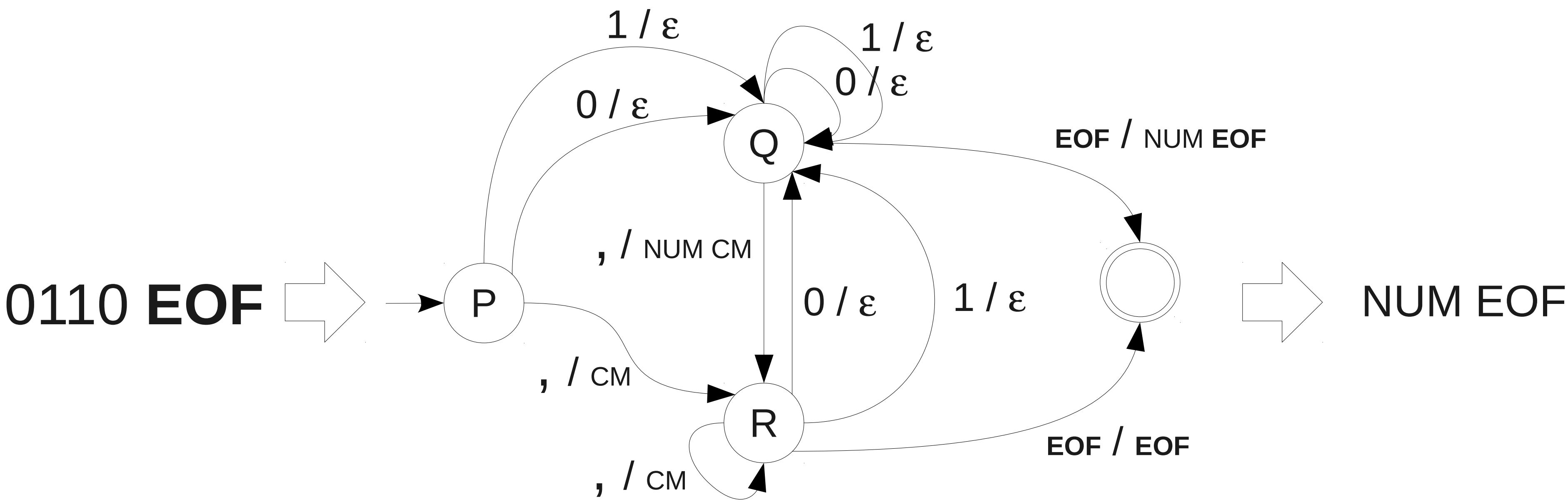
- Recognize and generate at the same time
  - FST :: A* -> B*
    - For finite alphabets A and B, both containing a special symbol **EOF**





What about: “0110”?

Dealing with EOF



From Inclusion to Transduction

- Inclusion check is simpler than a transduction
  - But not so much
- We can compute
  - $S :: State^{IN} \to [State^{FST}]$
  - $T :: Transition^{IN} \to [Transition^{FST}]$
- We need a resulting FA, OUT := FST(IN)
  - $State^{OUT}$:= Copy $State^{IN}$
  - For each $t^{FST} \in T(t^{IN} : A^{IN} \to B^{IN})$
    - Create $t^{OUT} : A^{OUT} \to B^{OUT}$

Abstract Lexical Analysis: Summary

- Convert an abstract string into a **N**FA
  - O(N)
- Compute FST(NFA)
  - O(|FST| * |NFA|)
- Loss of precision:
  - Only when creating the abstract string

Parsing Abstract Strings

- Inclusion ($A \subseteq B$) is undecidable if
  - A is regular
  - B is context-free
- Possible solutions
  - Check for disjointness (it is decidable)
    - Neither sound nor complete
    - But still useful
  - Loose precision (introduce more false alarms)

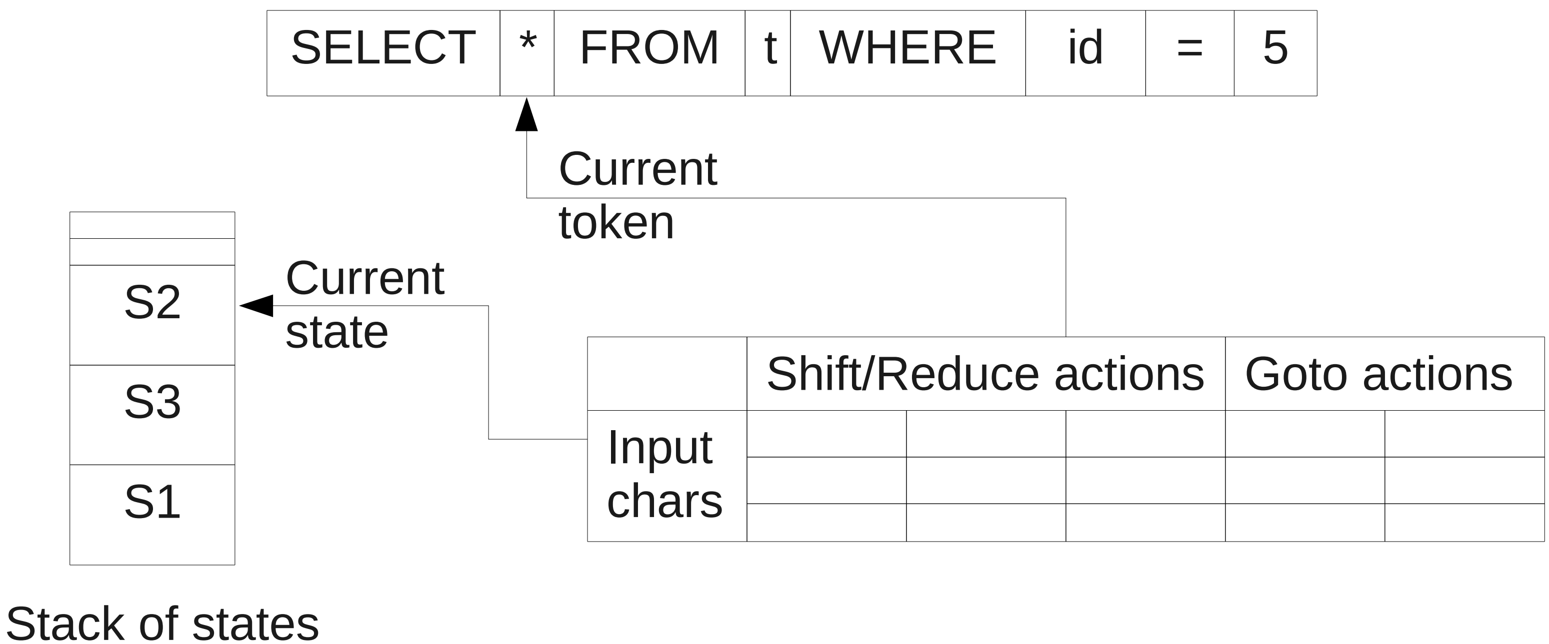
Two Principal Ways You ~~Shoot Yourself in the Foot~~ Loose Precision

- Approximations
  - Find a **regular** language contained in the CF one
    - **Bounding the depth of recursion** [CMS03]
  - Try to run a well-known parsing algorithm on **N**FAs
    - Earley parsing (done in [Thi05] in the form of a type system)
    - **LR-parsing** (called “Abstract Parsing” in [DKS09, KCY09])

Bounded Recursion [CMS03]

- Example of a non-regular grammar
  - E ::= int
  - E ::= (E)
- If we bound the recursion depth to D = 3, we get
  - E^{REG} ::= int | (int) | ((int)) | (((int)))
  - This is a regular set of strings
- False alarms
  - “(((((int)))))” $\in$ E, but $\notin E^{REG}$
  - The bigger D is, the less false alarms we get

Introduction to LR-Parsing



- Action table does not change
- Parser state is characterized by
  - Stack of states
  - Current offset in the input stream

Abstract Parsing

- Input
  - TOK :: NFA generating strings of tokens, which end with EOF
  - Action table of an LR parser P^G
- Output
  - OK – $L(TOK) \subseteq L(G)$
  - ERROR – L(TOK) **may be** not a subset of L(G)
- Algorithm
  - For each state of TOK find a set of possible stacks of P^G

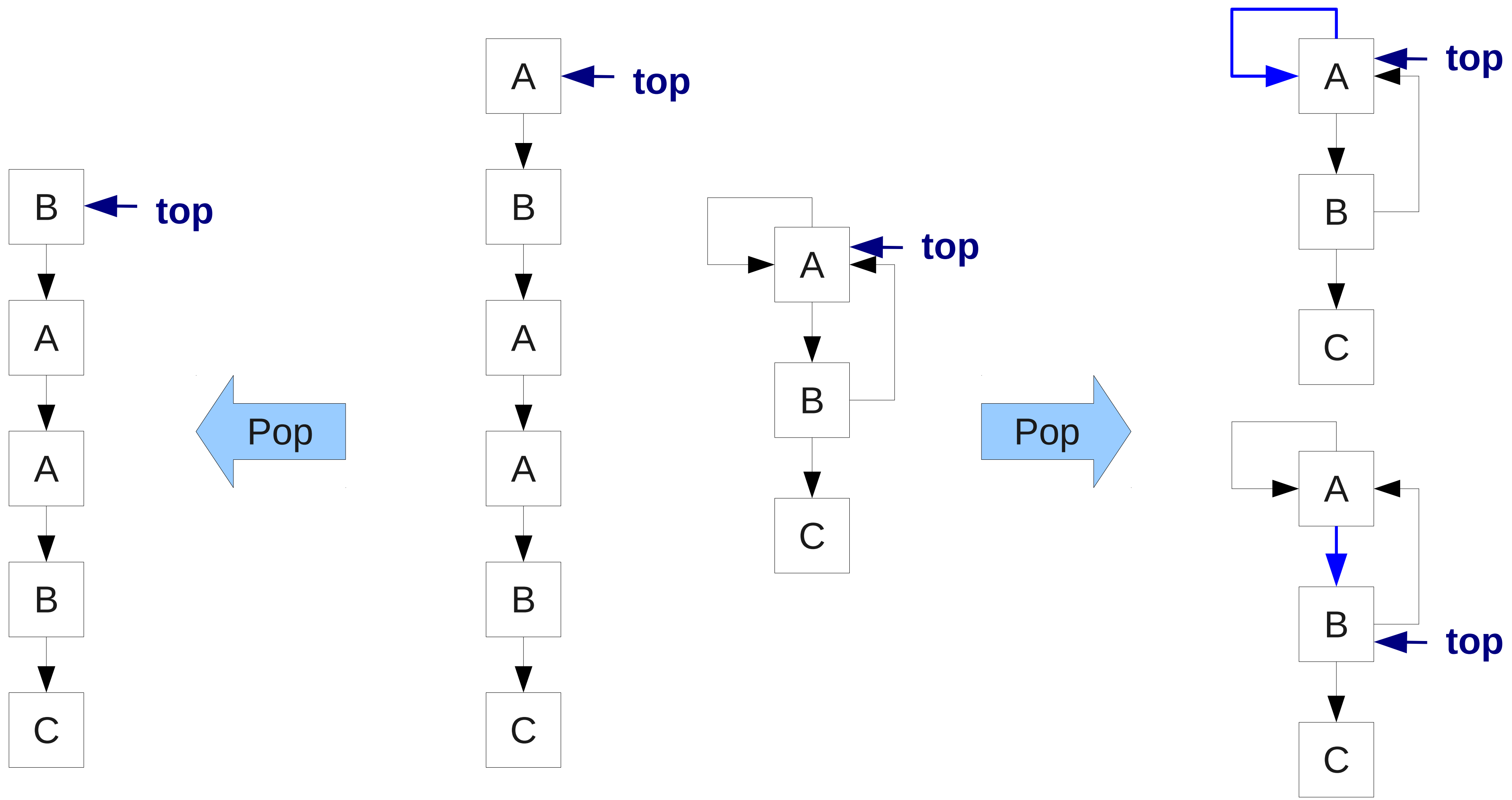
Abstract Parsing (Algorithm)

- Stacks(S^{TOK}) :: $State^{TOK}$ -> [$Stack^{G}$]
- Being in the state A^{TOK}
  - For each t^{TOK} : A^{TOK} -T-> B^{TOK}
    - For each stack ∈ Stacks(A^{TOK})
      - Perform actions of P^{G} with token T
        - If P^{G} returns an error, return ERROR
      - Add resulting stacks to States(B^{TOK})
  - If Stacks did not change
    - Return OK
  - Recursive call from B^{TOK}
- Termination
  - **NOT** guaranteed, because the set of possible stacks may be infinite

Summary So Far

- For finite inputs
  - Precise result
- For infinite inputs
  - No result
- Solution: loose precision
  - Represent sets of stacks as finite objects
    - e.g. regular approximation (stacks are also strings over the state alphabet) [DKS09]
    - e.g. consider only stacks of finite depth [KCY09]

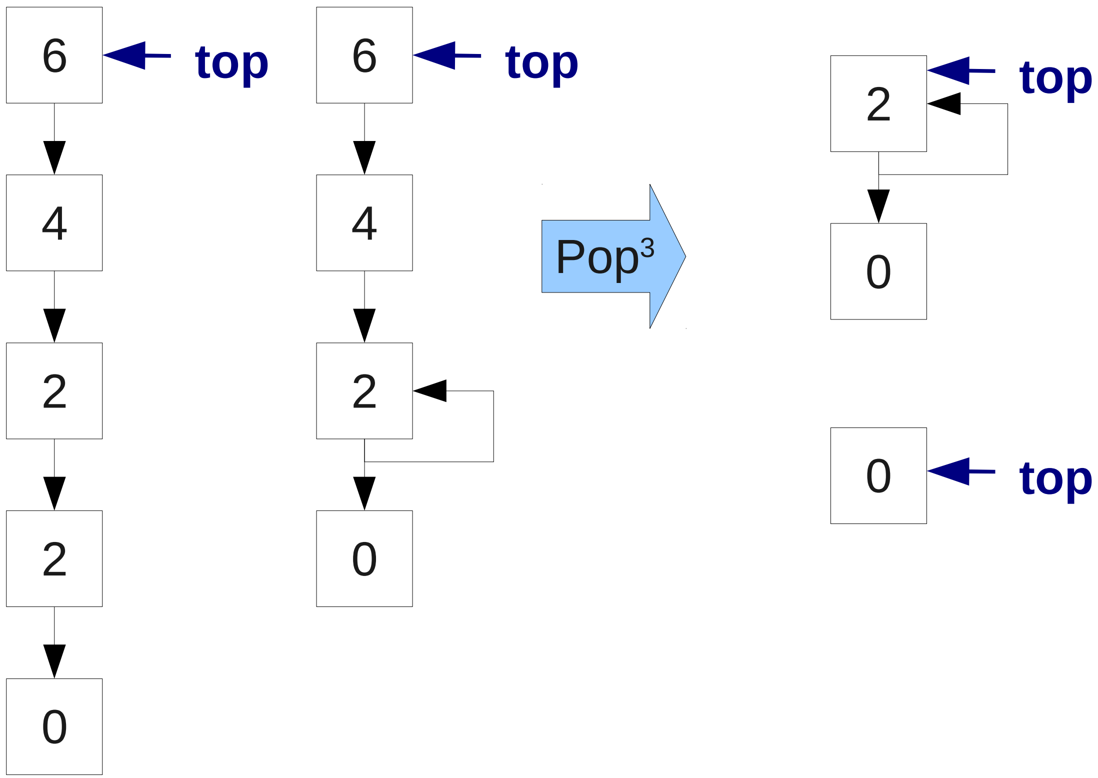
Regular Approximation for Stacks [DKS09]



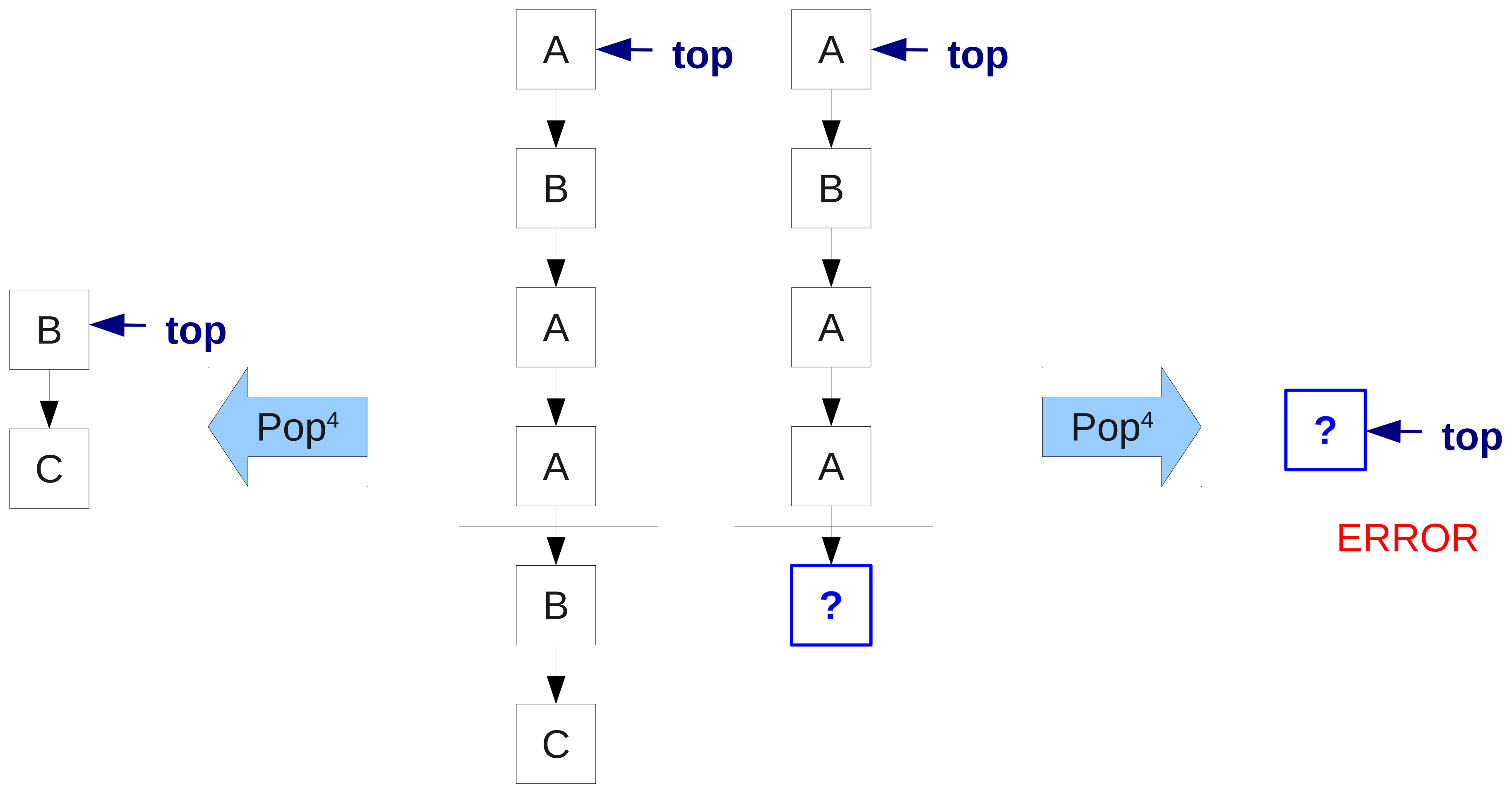
False Alarm Example for [DKS09]

- Grammar
  - E ::= num | (E)
- Input: “((num))”
- Trace:

Stack	Actions
0	Push 2
0 2	Push 2
0 2 2	Push 1
0 2 2 1	Pop, Push 4
0 2 2 4	Push 6
0 2 2 4 6	Pop^3, Push 4
0 2 4	Push 6
0 2 4 6	Pop^3, Push 3
0 3	Push 5
0 3 5	Accept



Stacks of Bounded Depth [KCY09]



False Alarm Example for [KCY09]

- Grammar
  - E ::= num | (E)
- D = 3
- Input: “((((((num))))))”
  - The bottom of the stack is lost
- With D = 1000 it is unlikely to loose anything
  - **NB:** Time and memory are O(|States|^D)
  - We have to experiment to look for reasonable D
    - In progress :)

Comparing the Two Abstractions

- Regular approximation
  - Does not handle nested parentheses at all
  - Even worse than bounded recursion
- Bounded stack depth
  - Does not handle nested parentheses of certain depths
  - Same as bounded recursion

Why Abstract Parsing

- We have two options:
  - Approximate SQL grammar with a regular one (by bounding recursion depth)
  - Apply abstract parsing with bounded stack depth (D)
    - Time complexity: $O(|States^{TOK}| * |States^{G}|^{D})$
- These two raise the same false alarms on infinite inputs
- Advantages of Abstract Parsing
  - Precision guaranteed on finite inputs
  - Helpful error reporting
  - Support for IDE features (e.g., content assist)

Reporting Errors

- Types of errors
  - **Unexpected token**: no action for the input token is present in the action table
    - Good: we have an erroneous token
  - **Non-accepting state**: all input characters are consumed, but the current state is not accepting
    - Not so good: we do not know what the errors is
- Error annotation positioning
  - Input characters are coming with their positions
  - Transducer collects the characters which form tokens

Overall Summary

- Convert an abstract string into a **N**FA
  - O(N)
- Compute FST(NFA)
  - O(|FST| * |NFA|)
- Perform abstract parsing on FST(NFA)
  - O(|NFA| * |Parser States|^{D})
- Loss of precision:
  - On creating the abstract string
  - On abstract parsing

References

- **[DKS09]** Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. *Abstract parsing: Static analysis of dynamically generated string output using LR-parsing technology*. In Jens Palsberg and Zhendong Su, editors, SAS, volume 5673 of Lecture Notes in Computer Science, pages 256–272. Springer, 2009.
- **[KCY09]** Soonho Kong, Wontae Choi, and Kwangkeun Yi. *Abstract parsing for two-staged languages with concatenation*. In GPCE ’09: Proceedings of the eighth international conference on Generative programming and component engineering, pages 109–116, New York, NY, USA, 2009. ACM.
- **[Thi05]** Peter Thiemann. *Grammar-based analysis of string expressions.* In TLDI ’05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation, pages 59–70, New York, NY, USA, 2005. ACM.
- **[CMS03]** Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. *Precise analysis of string expressions.* In Proc. 10th International Static Analysis Symposium, SAS ’03, volume 2694 of LNCS, pages 1–18. Springer-Verlag, June 2003. Available from http://www.brics.dk/JSA/.