Logic Characterization of Invisibly Structured Languages: the Case of Floyd Languages

Violetta Lonati[2] Dino Mandrioli[1] *Matteo Pradella*[1]

(1) DEIB - Politecnico di Milano
(2) DI - Università degli Studi di Milano

SOFSEM 2013

Outline

1 Background and motivations

2 Floyd's Operator-precedence languages

3 Automata for Operator-precedence languages

4 Logic

5 Applications and Conclusions

Background (1)

Floyd's Operator-precedence languages

- R. Floyd introduced *Operator-precedence (OP) grammars* in 1963, inspired by the structure of arithmetic expressions, and aimed at *deterministic parsing* of programming languages
- Early investigation on the algebraic and closure properties e.g. in [Crespi Reghizzi, Mandrioli and Martin 1978].

Background (1)

Floyd's Operator-precedence languages

- R. Floyd introduced *Operator-precedence (OP) grammars* in 1963, inspired by the structure of arithmetic expressions, and aimed at *deterministic parsing* of programming languages
- Early investigation on the algebraic and closure properties e.g. in [Crespi Reghizzi, Mandrioli and Martin 1978].
- Interest on OP languages was "virtually killed" by Knuth's LR algorithm.
- OP parsers are simple and efficient, still sometimes used in practice (e.g. for evaluating arithmetic expressions in gcc)

Background (2)

After 40+ years

- *Visibly Pushdown (VP) languages* [Alur and Madhusudan 2004]
- they exhibit many *closure properties* of regular languages (Boolean, concatenation, Kleene's $*$, ...)
- suitable to model *structured mark-up languages* such as XML; properties enable *model checking*.

Background (2)

After 40+ years

- *Visibly Pushdown (VP) languages* [Alur and Madhusudan 2004]
- they exhibit many *closure properties* of regular languages (Boolean, concatenation, Kleene's $*$, ...)
- suitable to model *structured mark-up languages* such as XML; properties enable *model checking*.

Floyd's OP languages are back

- OP languages actually *include* VP languages and enjoy the *same closure properties*

[Crespi Reghizzi, Mandrioli 2009]

Example: parsing arithmetic expressions

The classical context-free grammar for arithmetic expressions:

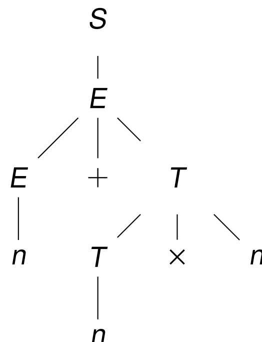
$$S \to E$$
$$E \to E + T$$
$$E \to T \times n$$
$$E \to n$$
$$T \to T \times n$$
$$T \to n$$

Example: parsing arithmetic expressions

The classical context-free grammar for arithmetic expressions:

$$
\begin{aligned}
S &\to E \\
E &\to E + T \\
E &\to T \times n \\
E &\to n \\
T &\to T \times n \\
T &\to n
\end{aligned}
$$

$$n + n \times n$$

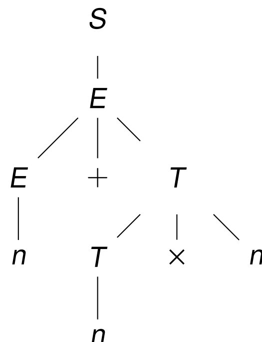


Example: parsing arithmetic expressions

The classical context-free grammar for arithmetic expressions:

$S \to E$
$E \to E + T$
$E \to T \times n$
$E \to n$
$T \to T \times n$
$T \to n$

$n + n \times n$



This grammar is such that $\times$ *takes precedence* over $+$ (as usual):

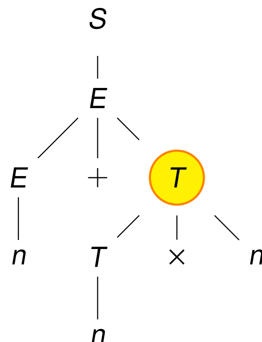
$n + \; n \times n$

Example: parsing arithmetic expressions

The classical context-free grammar for arithmetic expressions:

$$
\begin{aligned}
S &\to E \\
E &\to E + T \\
E &\to T \times n \\
E &\to n \\
T &\to T \times n \\
T &\to n
\end{aligned}
$$

$$n + n \times n$$



This grammar is such that $\times$ *takes precedence* over $+$ (as usual):

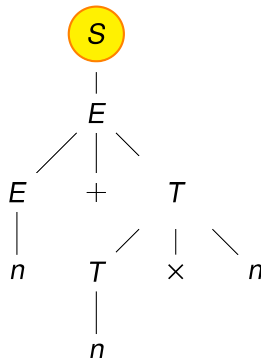
$n + n \times n$

Example: parsing arithmetic expressions

The classical context-free grammar for arithmetic expressions:

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow E + T \\
E &\rightarrow T \times n \\
E &\rightarrow n \\
T &\rightarrow T \times n \\
T &\rightarrow n
\end{aligned}
$$

$n + n \times n$



This grammar is such that $\times$ *takes precedence* over $+$ (as usual):

$n + \; n \times n$

Floyd's idea

A method for inserting parentheses into an expression is based on *precedence relations* between terminal symbols

- $a \lessdot b$ (a *yields precedence* over b)

Floyd's idea

A method for inserting parentheses into an expression is based on *precedence relations* between terminal symbols

- $a \lessdot b$ (a *yields precedence* over b)
- $a \gtrdot b$ (a *takes precedence* over b)

Floyd's idea

A method for inserting parentheses into an expression is based on *precedence relations* between terminal symbols

- $a \lessdot b$ (a *yields precedence* over b)
- $a \gtrdot b$ (a *takes precedence* over b)
- $a \doteq b$ (a *equals in precedence* b)

Floyd's idea

A method for inserting parentheses into an expression is based on *precedence relations* between terminal symbols

- $a \lessdot b$ (a *yields precedence* over b)
- $a \gtrdot b$ (a *takes precedence* over b)
- $a \doteq b$ (a *equals in precedence* b)

For each pair of symbols we determine the proper precedence and obtain the *precedence table*

Precedence table - example

In a sense, there are no *visible parentheses*, but precedence relations introduce *invisible ones*, e.g.:

Precedence table

$n + n \times n$

	n	$+$	$\times$	$\#$
n		$\gtrdot$	$\gtrdot$	$\gtrdot$
$+$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$\times$	$\doteq$			$\gtrdot$
$\#$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

Precedence table - example

In a sense, there are no *visible parentheses*, but precedence relations introduce *invisible ones*, e.g.:

Precedence table

	n	$+$	$\times$	$\#$
n		$\gtrdot$	$\gtrdot$	$\gtrdot$
$+$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$\times$	$\doteq$			$\gtrdot$
$\#$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

$n + n \times n$

$$\# \lessdot n \doteq + \underbrace{\lessdot n \gtrdot} \times \doteq n \gtrdot \#$$

$\#$ is the *boundary* symbol

Precedence table - example

In a sense, there are no *visible parentheses*, but precedence relations introduce *invisible ones*, e.g.:

Precedence table

	n	$+$	$\times$	$\#$
n		$\gtrdot$	$\gtrdot$	$\gtrdot$
$+$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$\times$	$\doteq$			$\gtrdot$
$\#$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

$n + n \times n$

$\# \lessdot n \doteq + \underbrace{\lessdot n \gtrdot} \times \doteq n \gtrdot \#$

$\# \lessdot n \doteq + \underbrace{\lessdot \times \doteq n \gtrdot} \#$

$\#$ is the *boundary* symbol

In a sense, there are no *visible parentheses*, but precedence relations introduce *invisible ones*, e.g.:

Precedence table

	n	$+$	$\times$	$\#$
n		$\gtrdot$	$\gtrdot$	$\gtrdot$
$+$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$\times$	$\doteq$			$\gtrdot$
$\#$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

$n + n \times n$

$\# \lessdot n \doteq + \underbrace{\lessdot n \gtrdot} \times \doteq n \gtrdot \#$

$\# \lessdot n \doteq + \underbrace{\lessdot \times \doteq n \gtrdot} \#$

$\# \underbrace{\lessdot n \doteq + \gtrdot} \#$

$\#$ is the *boundary* symbol

Precedence table - example

In a sense, there are no *visible parentheses*, but precedence relations introduce *invisible ones*, e.g.:

Precedence table

	n	$+$	$\times$	$\#$
n		$\gtrdot$	$\gtrdot$	$\gtrdot$
$+$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$\times$	$\doteq$			$\gtrdot$
$\#$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

$\#$ is the *boundary* symbol

$$n + n \times n$$

$$\# \lessdot n \doteq + \underbrace{\lessdot n \gtrdot} \times \doteq n \gtrdot \#$$

$$\# \lessdot n \doteq + \underbrace{\lessdot \times \doteq n \gtrdot} \#$$

$$\# \underbrace{\lessdot n \doteq + \gtrdot} \#$$

$$\# \doteq \#$$

Operator grammar

A CF grammar is an *operator grammar* if no right-hand side contains two consecutive nonterminals (and it is not empty).

Floyd's Operator-precedence grammars

Operator grammar

A CF grammar is an *operator grammar* if no right-hand side contains two consecutive nonterminals (and it is not empty).

OP grammar

An operator grammar is a *operator-precedence grammar* if there are *no conflicts in its precedence matrix*

i.e. given a pair of symbols, there is *at most one* precedence relation.

Operator-precedence Automata

- OP automata are stack-based
- The stack alphabet consists of pairs *[a q]* where
  a is a symbol from the input alphabet,
  q is a state.

Operator-precedence Automata

- OP automata are stack-based
- The stack alphabet consists of pairs $[\,a \;\; q\,]$ where

  a is a symbol from the input alphabet,
  q is a state.
- Variant: the symbol may be marked $[\,a' \;\; q\,]$

Operator-precedence Automata

3 kinds of moves, depending on the precedence relation between the symbol *a on top of the stack* and *next input symbol b*:

Operator-precedence Automata

3 kinds of moves, depending on the precedence relation between the symbol *a on top of the stack* and *next input symbol b*:

push move when $a \doteq b$
b is pushed on the stack

Operator-precedence Automata

3 kinds of moves, depending on the precedence relation between the symbol *a on top of the stack* and *next input symbol b*:

push move when $a \doteq b$
b is pushed on the stack

mark move when $a \lessdot b$
b' is pushed on the stack

Operator-precedence Automata

3 kinds of moves, depending on the precedence relation between the symbol *a on top of the stack* and *next input symbol b*:

push move: when $a \doteq b$
b is pushed on the stack

mark move: when $a \lessdot b$
b' is pushed on the stack

flush move: when $a \gtrdot b$
pop from the stack until a *marked symbol c'* is found;

Operator-precedence Automata

3 kinds of moves, depending on the precedence relation between the symbol *a on top of the stack* and *next input symbol b*:

push move when $a \doteq b$
b is pushed on the stack

mark move when $a \lessdot b$
b' is pushed on the stack

flush move when $a \gtrdot b$
pop from the stack until a *marked symbol c'* is found;
note: *b is not consumed* (*look-ahead*);

Operator-precedence Automata

3 kinds of moves, depending on the precedence relation between the symbol *a on top of the stack* and *next input symbol b*:

push move when $a \doteq b$
b is pushed on the stack

mark move when $a \lessdot b$
b' is pushed on the stack

flush move when $a \gtrdot b$
pop from the stack until a *marked symbol c'* is found;
note: *b is not consumed* (*look-ahead*);
note: *c' is popped* as well (*look-back*).

Operator-precedence Automata

3 kinds of moves, depending on the precedence relation between the symbol *a on top of the stack* and *next input symbol b*:

push move when $a \doteq b$
b is pushed on the stack

mark move when $a \lessdot b$
b' is pushed on the stack

flush move when $a \gtrdot b$
pop from the stack until a *marked symbol c'* is found;
note: *b is not consumed* (*look-ahead*);
note: *c' is popped* as well (*look-back*).

e.g. in $n + \overbrace{\lessdot \times n \gtrdot} \#$, $\times$ is marked, $+$ is the look-back and $\#$ the look-ahead.

An example automaton

- Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

An example automaton

- Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.
- *High priority* means that $\underline{a}$ closes both an a and every pending b after it; e.g.

$$\overbrace{a\ b\ b\ \overbrace{b\ \overbrace{a\underline{a}}\ \underline{b}}\ \underline{a}}$$

An example automaton

- Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.
- *High priority* means that $\underline{a}$ closes both an a and every pending b after it; e.g.

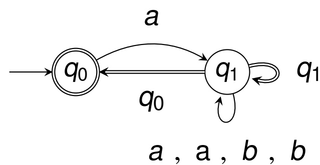
$$\overbrace{a\, b\, b\, \overbrace{b\, \overbrace{a \underline{a}}\, \underline{b}}\, \underline{a}}$$

- also, it is required that the string has only top-level a pairs (e.g. it cannot start with b).

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



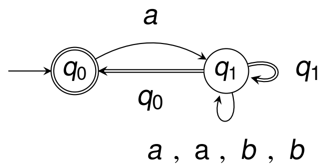
$abba\underline{aba}\#$

An example automaton

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



$[\# \; q_0]$

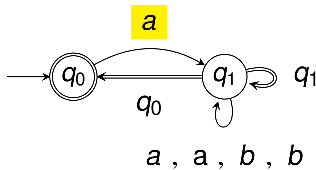
$abba\underline{aba}\#$

restart

skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



mark

$[\# \; q_0]$

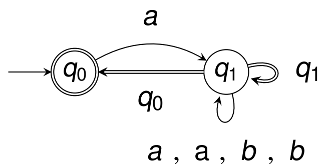
$abba\underline{aba}\#$

An example automaton

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



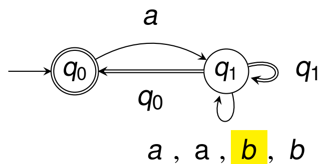
$[\# \; q_0][a' \; q_1]$

$abba\underline{aba}\#$

restart   skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



mark

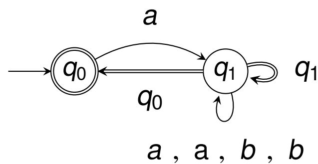
$[\# \; q_0][a' \; q_1]$

$abba\underline{a}\underline{b}\underline{a}\#$

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



$[\# \; q_0][a' \; q_1][b' \; q_1]$

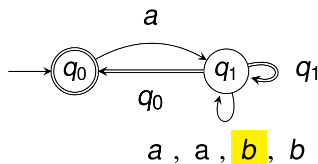
$abba\underline{aba}\#$

An example automaton

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



mark

$[\# \; q_0][a' \; q_1][b' \; q_1]$

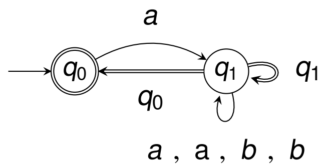
$ab\,b\,a\,\underline{a}\,b\,\underline{a}\#$

An example automaton

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



$[\# \; q_0][a' \; q_1][b' \; q_1][b' \; q_1]$

$abba\underline{aba}\#$

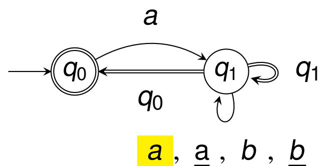
An example automaton

restart

skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



mark

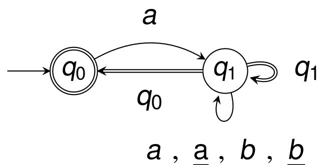
$[\# \; q_0][a' \; q_1][b' \; q_1][b' \; q_1]$

$abba\underline{a}\underline{b}\underline{a}\#$

◂ restart ⏩ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$

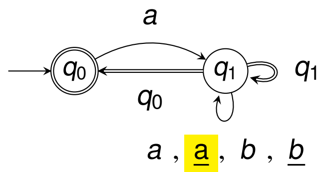


$[\# \; q_0][a' \; q_1][b' \; q_1][b' \; q_1][a' \; q_1]$ $\quad$ $abba\underline{a}b\underline{a}\#$

restart skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



push

$[\# \ q_0][a' \ q_1][b' \ q_1][b' \ q_1][a' \ q_1]$

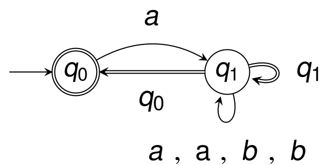
abba$\underline{aba}$#

An example automaton

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$

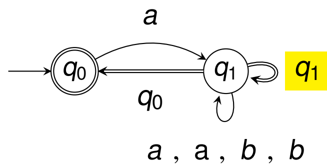


$[\# \; q_0][a' \; q_1][b' \; q_1][b' \; q_1][a' \; q_1][\underline{a} \; q_1]$ $abbaa\underline{a}\underline{b}\underline{a}\#$

restart skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$

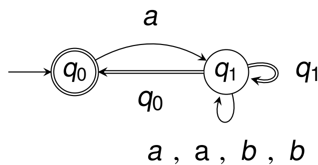


flush $[\# \ q_0][a' \ q_1][b' \ q_1][b' \ q_1][a' \ q_1][\underline{a} \ q_1]$ abbaa$\underline{b}\underline{a}$#

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



$[\# \; q_0][a' \; q_1][b' \; q_1][b' \; q_1]$

$abba\underline{a}\underline{ba}\#$

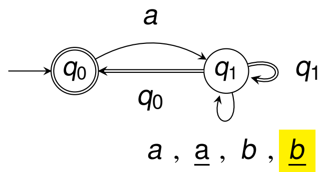
An example automaton

restart

skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



push

$[\# \; q_0][a' \; q_1][b' \; q_1][b' \; q_1]$

$abba\underline{a}\underline{b}\underline{a}\#$

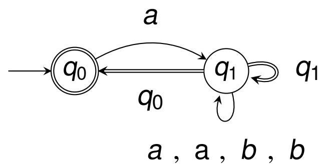
An example automaton

restart

skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$

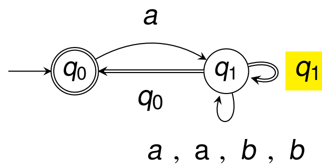


$[\# \; q_0][a' \; q_1][b' \; q_1][b' \; q_1][\underline{b} \; q_1]$ $abba\underline{ab}\underline{a}\#$

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



flush

$[\# \; q_0][a' \; q_1][b' \; q_1][b' \; q_1][\underline{b} \; q_1]$

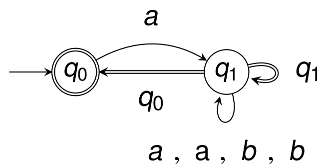
$abbaab\underline{a}\#$

An example automaton

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



$[\# \ q_0][a' \ q_1][b' \ q_1]$

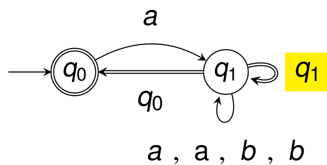
$abba\underline{ab}\underline{a}\#$

An example automaton

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



flush $[\# \; q_0][a' \; q_1][b' \; q_1]$

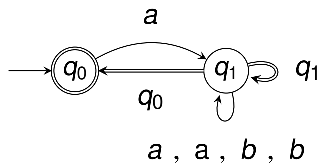
$abbaab\underline{a}\#$

An example automaton

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



$[\# \; q_0][a' \; q_1]$

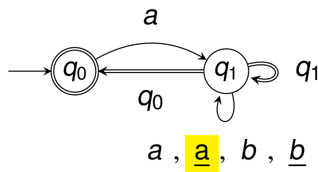
$abba\underline{ab}\underline{a}\#$

restart

skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



push

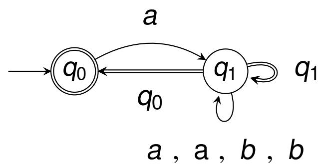
$[\# \ q_0][a' \ q_1]$

$abbaab\underline{a}\#$

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



$[\# \; q_0][a' \; q_1][\underline{a} \; q_1]$

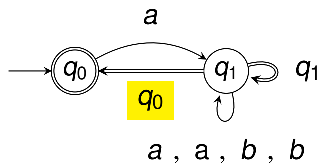
$abbaa\underline{ba}\#$

An example automaton

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



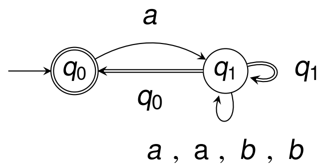
flush $[\# \, q_0][a' \, q_1][\underline{a} \, q_1]$

$abba\underline{aba}\#$

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



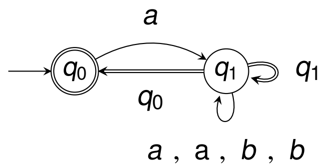
$[\# \; q_0]$

$abbaa\underline{ba}\#$

◂ restart ▸▸ skip

Dyck language with parentheses pairs $a, \underline{a}$ (high priority) and $b, \underline{b}$.

	a	$\underline{a}$	b	$\underline{b}$	$\#$
a	$\lessdot$	$\doteq$	$\lessdot$		
$\underline{a}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
b	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$	
$\underline{b}$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	
$\#$	$\lessdot$		$\lessdot$		$\doteq$



accept $[\# \; q_0]$

$abbaaba\#$

Equivalence theorems

- Deterministic OP automata are equivalent to nondeterministic ones.
- OP grammars and automata have the *same expressive power*.

[Lonati, Mandrioli, Pradella 2011]

A logic for operator-precedence languages

- The $\mathrm{MSO}_{\Sigma,M}$ (*monadic second-order logic*) is interpreted over (Σ, M) *strings*, and it is defined by the following syntax
- where $a \in \Sigma$, x, y are first-order variables (representing *positions* in the string) and X is a set variable.
- $\varphi :=$ $a(x) \mid x \in X \mid x \leq y \mid x = y + 1 \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi \mid$ $x \curvearrowright y$ *matching relation*

A logic for operator-precedence languages

- The $\mathrm{MSO}_{\Sigma,M}$ (*monadic second-order logic*) is interpreted over (Σ, M) *strings*, and it is defined by the following syntax
- where $a \in \Sigma$, x, y are first-order variables (representing *positions* in the string) and X is a set variable.
- $\varphi :=$ $a(x) \mid x \in X \mid x \leq y \mid x = y + 1 \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi \mid$
  $x \curvearrowright y$ *matching relation*
- Intuitively, $x \curvearrowright y$ holds iff positions x and y are to become *“adjacent” during parsing*

A logic for operator-precedence languages

- The $MSO_{\Sigma,M}$ (*monadic second-order logic*) is interpreted over (Σ, M) *strings*, and it is defined by the following syntax
- where $a \in \Sigma$, x, y are first-order variables (representing *positions* in the string) and X is a set variable.
- $\varphi :=$ $a(x) \mid x \in X \mid x \leq y \mid x = y + 1 \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi \mid$ $x \curvearrowright y$ *matching relation*
- Intuitively, $x \curvearrowright y$ holds iff positions x and y are to become *"adjacent" during parsing*
- i.e. the symbol at x is used as *look-back*, and the one at y as *look-ahead* in a *flush* move.

Logic-Automata equivalence

- The construction from logic to automata is quite standard
- From automata to logic, it is not:

- The construction from logic to automata is quite standard
- From automata to logic, it is not:
  - Finite-state and VP automata are *real-time* machines;
  - OP automata are not, so it is not easy to represent their runtime configuration in logic;
  - Indeed $\curvearrowright$ represents a relation that is *not* one-to-one (the structure can be *invisible*).

Logic-Automata equivalence

- The construction from logic to automata is quite standard
- From automata to logic, it is not:
  - Finite-state and VP automata are *real-time* machines;
  - OP automata are not, so it is not easy to represent their runtime configuration in logic;
  - Indeed $\curvearrowright$ represents a relation that is *not* one-to-one (the structure can be *invisible*).
- In parenthesis languages (like VPL) the structure is *visible*; the analogous of $\curvearrowright$ in VPL matches parentheses (*one-to-one*).

Applications

Parallel and incremental parsing

OP languages are *locally parsable*: i.e. we can parse substrings without modifying the outcome when considering larger strings (*PAPAGENO*)

[Barenghi, Crespi Reghizzi, Mandrioli, Pradella 2012; 2013]

Applications

Parallel and incremental parsing

OP languages are *locally parsable*: i.e. we can parse substrings without modifying the outcome when considering larger strings (*PAPAGENO*)

[Barenghi, Crespi Reghizzi, Mandrioli, Pradella 2012; 2013]

Model checking

Closure properties enable *infinite-state model checking*.

OP languages can model some characteristics exhibited by real-world systems, as *transactions*, *interrupts* and *hierarchical management* (e.g. in operating systems, distributed databases, privilege-based accesses, versioning systems).

Conclusions

OP automata and logic

We defined OP automata and a MSO logic that perfectly matches the generative power of OP grammars

Conclusions

OP automata and logic

We defined OP automata and a MSO logic that perfectly matches the generative power of OP grammars

Operator-precedence languages

OP languages have *almost the full generality of deterministic context-free* but *enjoy all the nice properties* of regular and VP languages.