LL Parsing

Traditional Parsing Algorithms

Eduardo Souza, Guido Wachsmuth, Eelco Visser

Recap: Lexical Analysis
lessons learned

What are the formalisms to describe regular languages?

- regular grammars
- regular expressions
- finite state automata

Why are these formalisms equivalent?

- constructive proofs

How can we generate compiler tools from that?

- implement DFAs
- generate transition tables

Overview

today's lecture

Overview
today's lecture

efficient parsing algorithms

- predictive/recursive descent parsing
- LL parsing

Overview
today's lecture

efficient parsing algorithms

- predictive/recursive descent parsing
- LL parsing

grammar classes

- LL(k) grammars
- LR(k) grammars

Overview
today's lecture

efficient parsing algorithms

- predictive/recursive descent parsing
- LL parsing

grammar classes

- LL(k) grammars
- LR(k) grammars

more top-down parsing algorithms

- Parser Combinators
- Parsing Expression Grammars
- ALL(*)

I

Predictive (Recursive Descent) Parsing

Recap: A Theory of Language

formal languages

Recap: A Theory of Language

formal languages

vocabulary Σ

finite, nonempty set of elements (words, letters)

alphabet

Recap: A Theory of Language

formal languages

vocabulary Σ

finite, nonempty set of elements (words, letters)

alphabet

string over Σ

finite sequence of elements chosen from Σ

word, sentence, utterance

Recap: A Theory of Language

formal languages

vocabulary Σ

finite, nonempty set of elements (words, letters)

alphabet

string over Σ

finite sequence of elements chosen from Σ

word, sentence, utterance

formal language λ

set of strings over a vocabulary Σ

$\lambda \subseteq \Sigma^*$

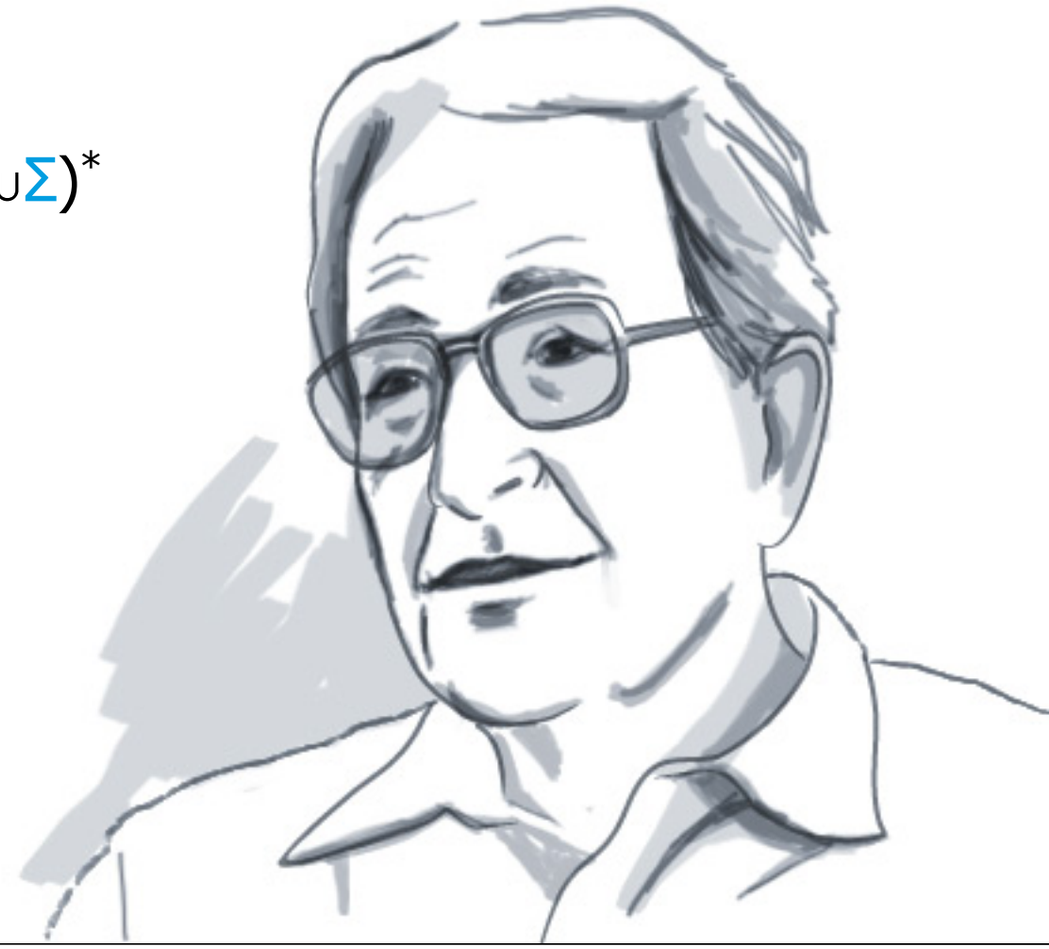
Recap: A Theory of Language

formal grammars

Recap: A Theory of Language
formal grammars

formal grammar $G = (N, \Sigma, P, S)$

- nonterminal symbols N
- terminal symbols Σ
- production rules $P \subseteq (N \cup \Sigma)^* \; N \; (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- start symbol $S \in N$



Recap: A Theory of Language
formal grammars

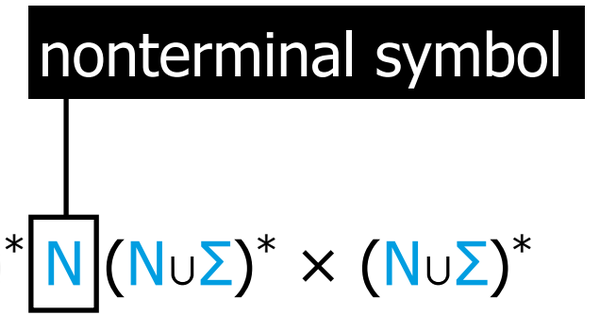
formal grammar $G = (N, \Sigma, P, S)$

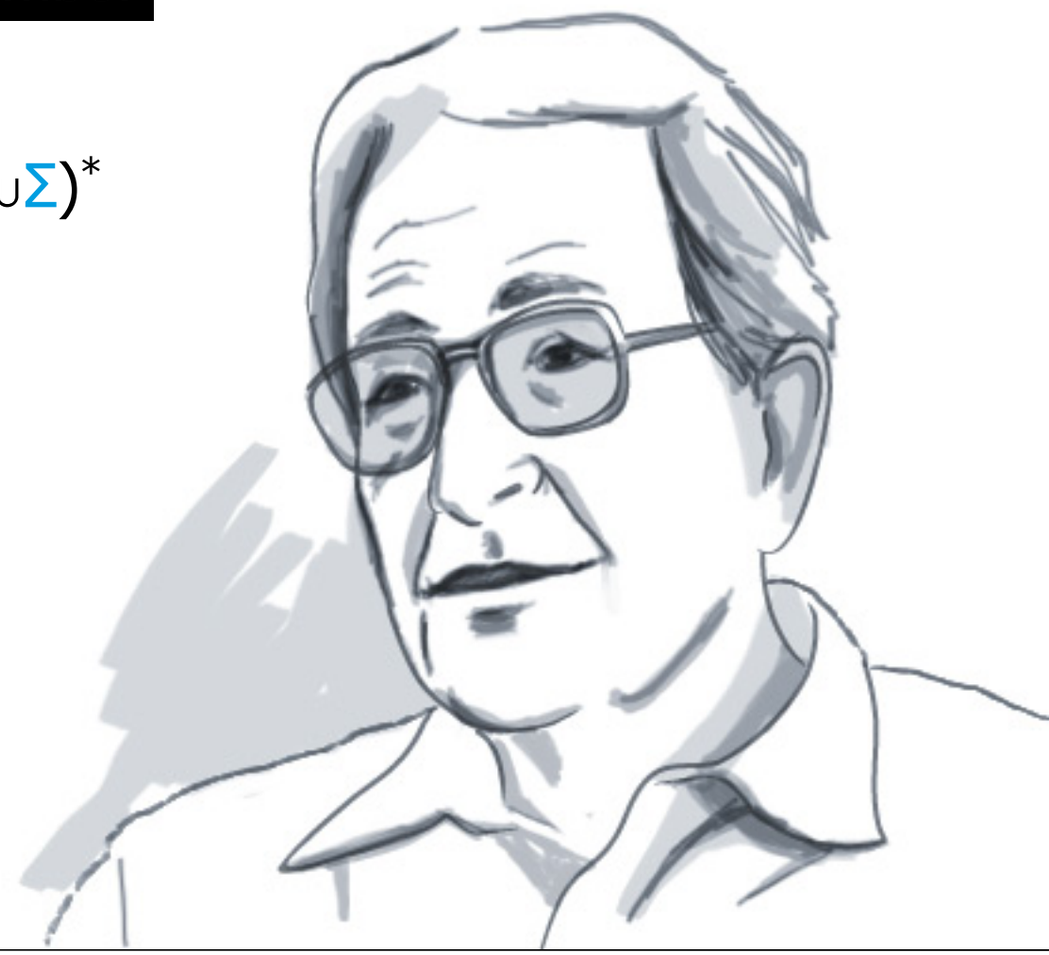
nonterminal symbols N

terminal symbols Σ

production rules $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$

start symbol $S \in N$





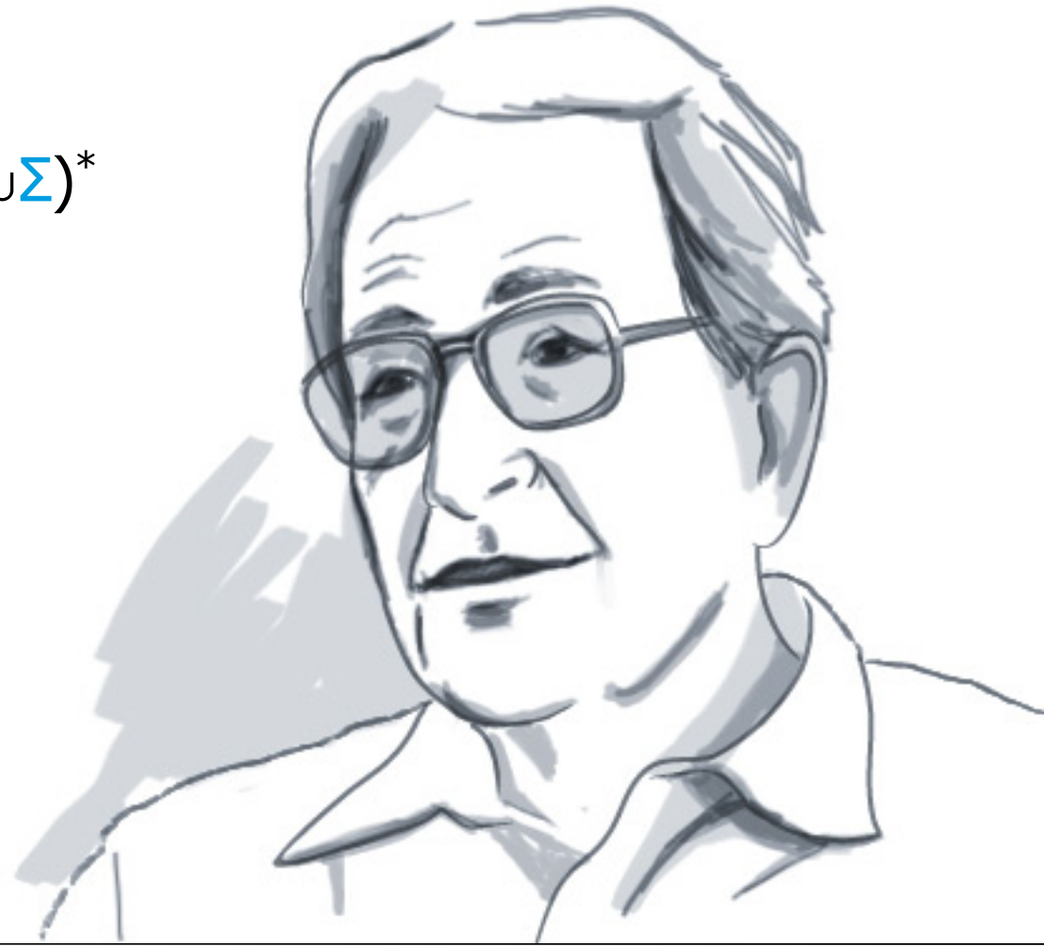
Recap: A Theory of Language

formal grammars

formal grammar $G = (N, \Sigma, P, S)$

- nonterminal symbols N
- terminal symbols Σ
- production rules $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- start symbol $S \in N$

context



Recap: A Theory of Language
formal grammars

formal grammar $G = (N, \Sigma, P, S)$

- nonterminal symbols N
- terminal symbols Σ
- production rules $P \subseteq (N \cup \Sigma)^* \; N \; (N \cup \Sigma)^* \times \boxed{(N \cup \Sigma)^*}$
- start symbol $S \in N$

replacement

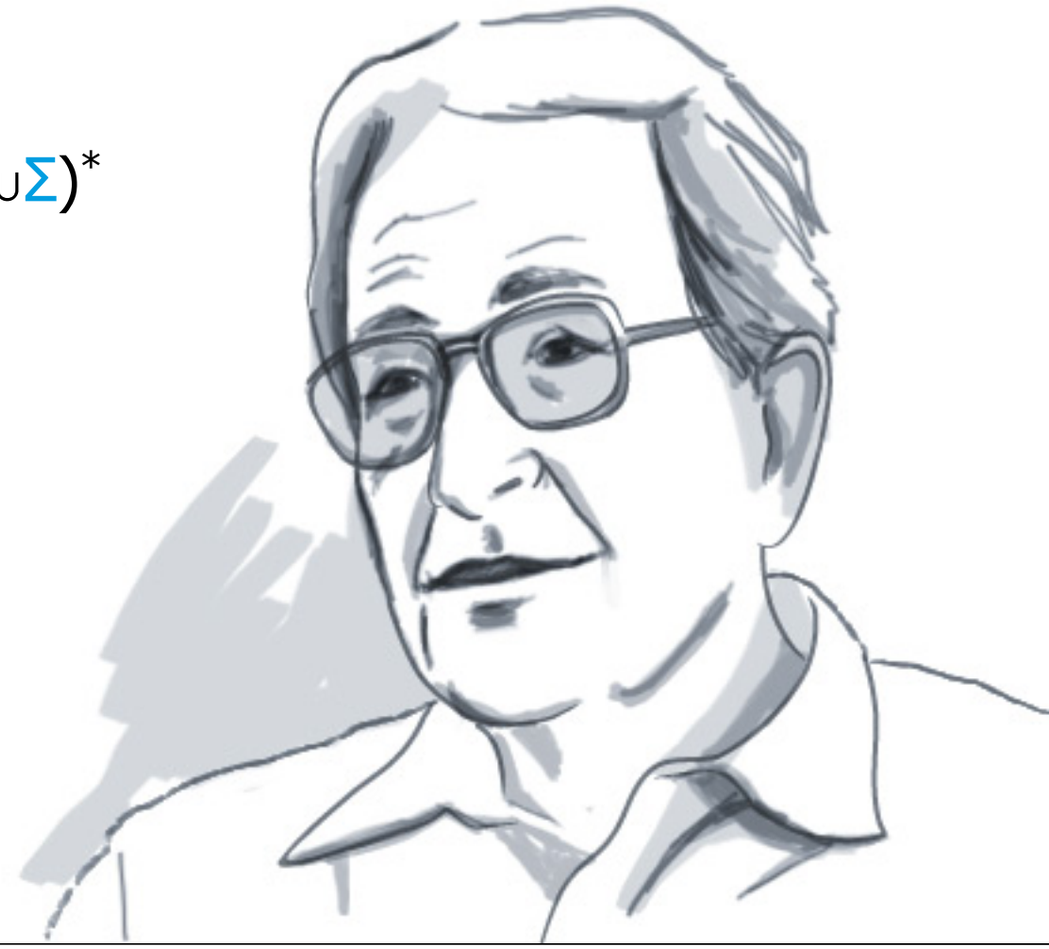
Recap: A Theory of Language
formal grammars

formal grammar $G = (N, \Sigma, P, S)$

- nonterminal symbols N
- terminal symbols Σ
- production rules $P \subseteq (N \cup \Sigma)^* \; N \; (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- start symbol $S \in N$

grammar classes

- type-0, unrestricted
- type-1, context-sensitive: (a A c, a b c)
- type-2, context-free: $P \subseteq N \times (N \cup \Sigma)^*$
- type-3, regular: (A, x) or (A, xB)



Recap: A Theory of Language

formal languages

Recap: A Theory of Language

formal languages

formal grammar G = (N, Σ, P, S)

Recap: A Theory of Language
formal languages

formal grammar $G = (N, \Sigma, P, S)$

derivation relation $\Rightarrow_G \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$

$w \Rightarrow_G w' \Leftrightarrow$

$\exists (p, q) \in P: \exists u, v \in (N \cup \Sigma)^*:$

$w = u\ p\ v \wedge w' = u\ q\ v$

Recap: A Theory of Language
formal languages

formal grammar $G = (N, \Sigma, P, S)$

derivation relation $\Rightarrow_G \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$

$w \Rightarrow_G w' \Leftrightarrow$

$\exists (p, q) \in P: \exists u, v \in (N \cup \Sigma)^*:$

$w = u\ p\ v \wedge w' = u\ q\ v$

formal language $L(G) \subseteq \Sigma^*$

$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$

Recap: A Theory of Language
formal languages

formal grammar $G = (N, \Sigma, P, S)$

derivation relation $\Rightarrow_G \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$

$w \Rightarrow_G w' \Leftrightarrow$

$\exists (p, q) \in P: \exists u, v \in (N \cup \Sigma)^*:$

$w = u\ p\ v \wedge w' = u\ q\ v$

formal language $L(G) \subseteq \Sigma^*$

$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$

classes of formal languages

Predictive parsing
recursive descent

Exp → “while” Exp “do” Exp

```
public void parseExp() {
    consume(WHILE);
    parseExp();
    consume(DO);
    parseExp();
}
```

Predictive parsing

look ahead

Exp → “while” Exp “do” Exp

Exp → “if” Exp “then” Exp “else” Exp

```
public void parseExp() {

   switch current() {
      case WHILE: consume(WHILE); parseExp(); ...; break;
      case IF   : consume(IF); parseExp(); ...; break;
      default   : error();
   }
}
```

Predictive parsing

parse table

rows

- nonterminal symbols N

- symbol to parse

columns

- terminal symbols Σ^k

- look ahead k

entries

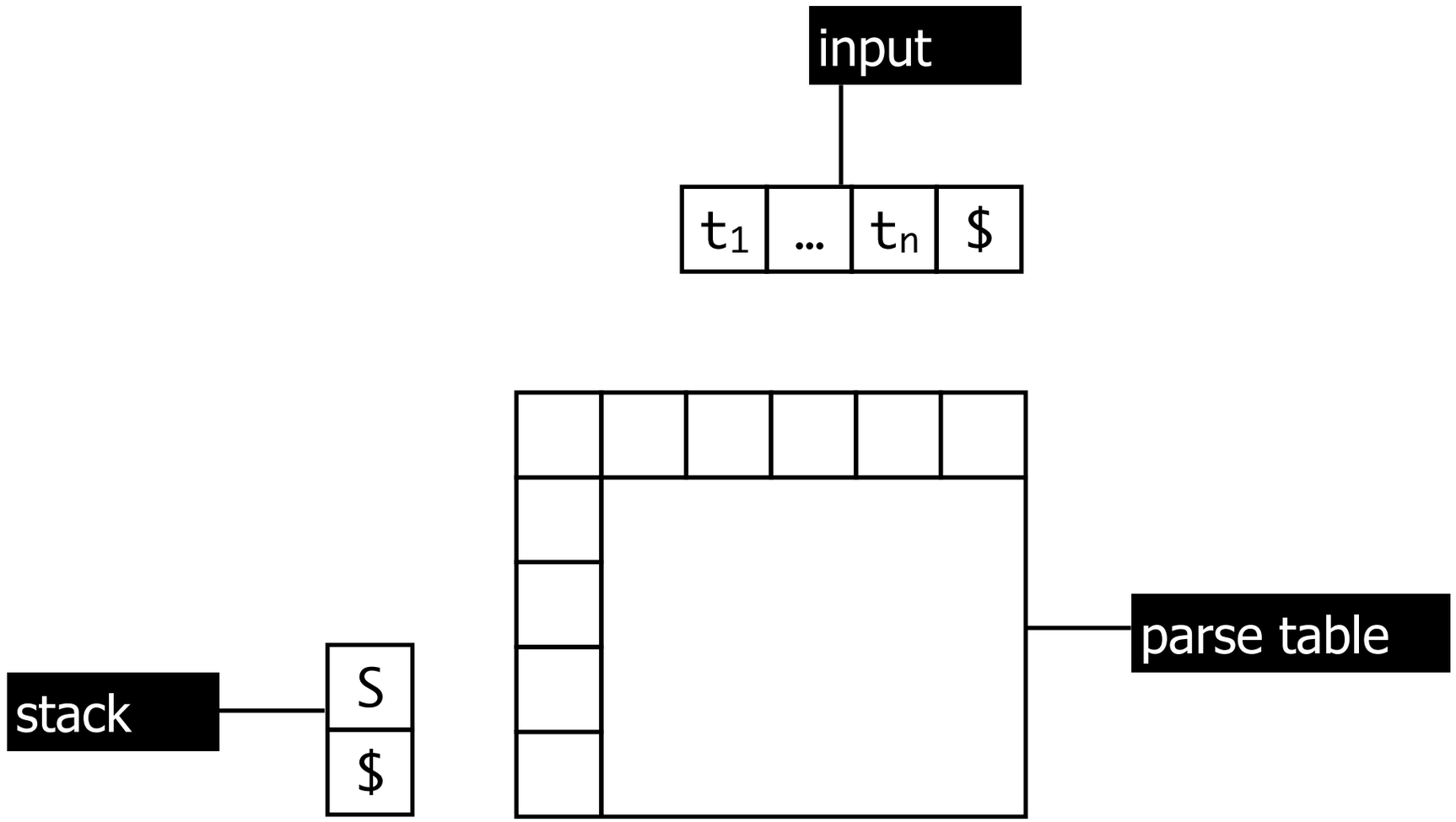
- production rules P

- possible conflicts

	T_1	T_2	T_3	...
N_1	$N_1 \rightarrow$...		$N_1 \rightarrow$...	
N_2		$N_2 \rightarrow$...		
N_3		$N_3 \rightarrow$...	$N_3 \rightarrow$...	
N_4	$N_4 \rightarrow$...			
N_5		$N_5 \rightarrow$...		
N_6	$N_6 \rightarrow$...	$N_6 \rightarrow$...		
N_7			$N_7 \rightarrow$...	
N_8	$N_8 \rightarrow$...	$N_8 \rightarrow$...	$N_8 \rightarrow$...	
...				

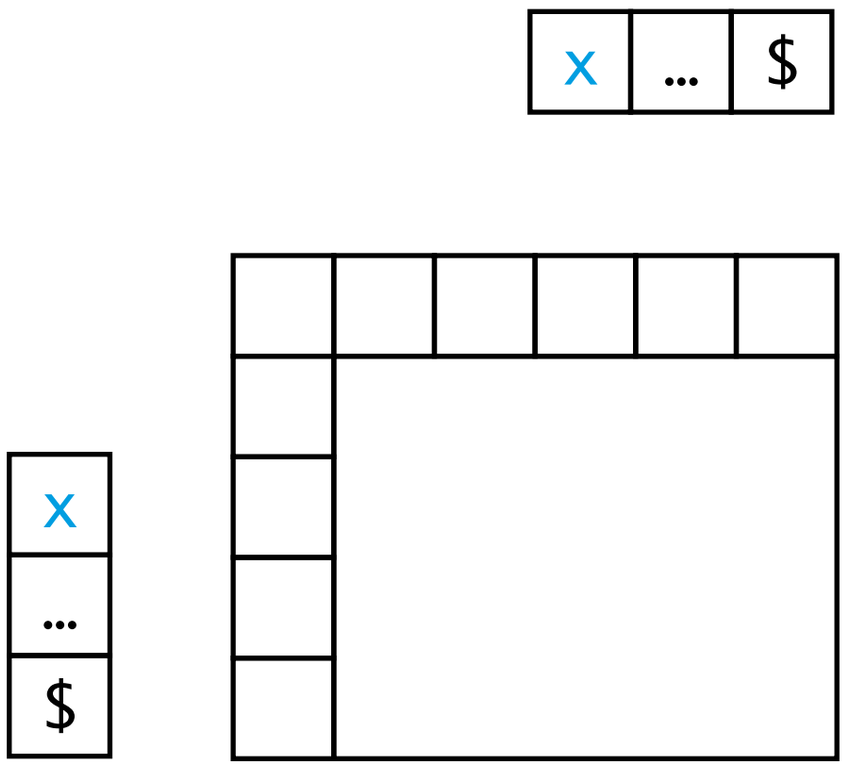
Predictive parsing

automaton



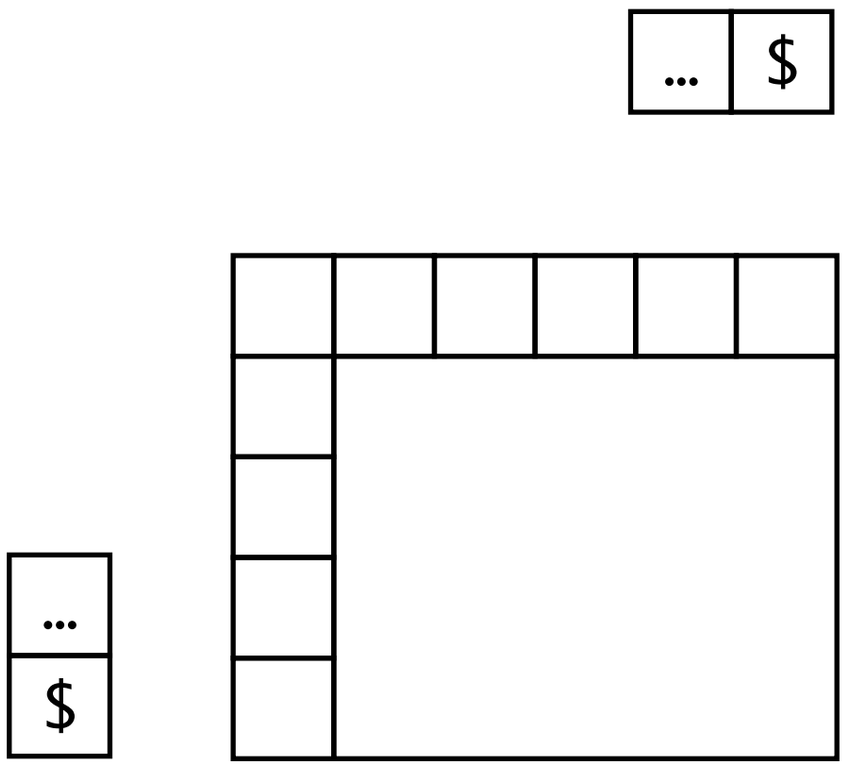
Predictive parsing

automaton



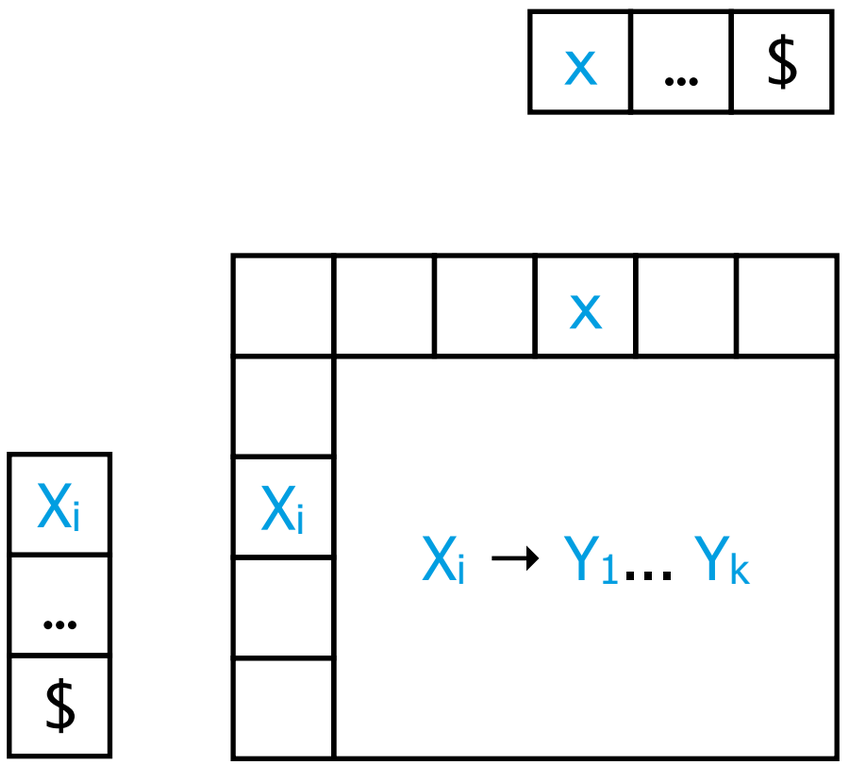
Predictive parsing

automaton



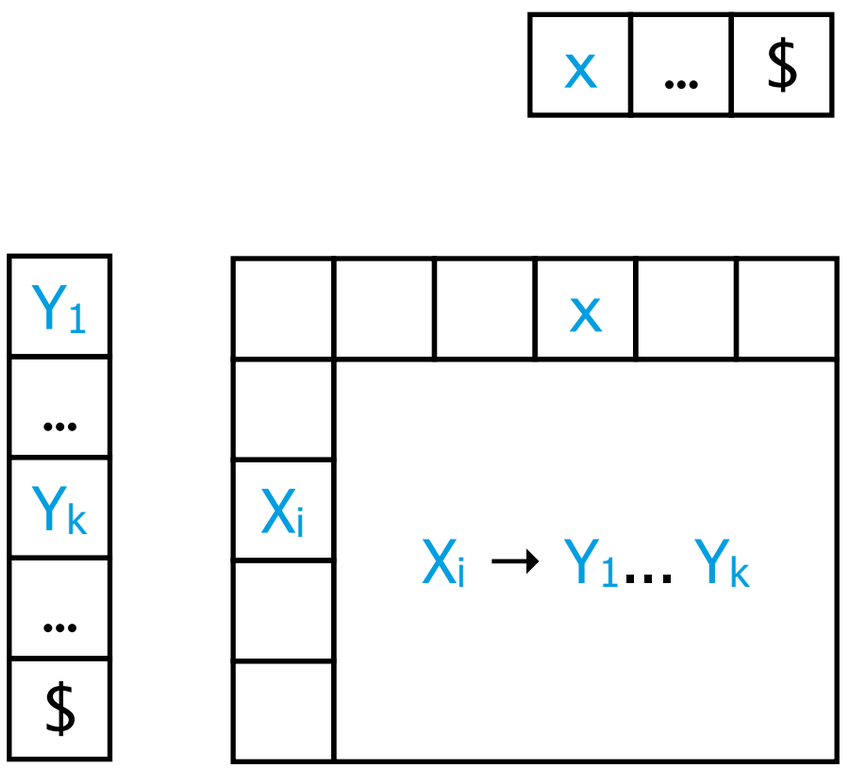
Predictive parsing

automaton



Predictive parsing

automaton



II

LL Parse Tables

Predictive parsing
filling the table

entry $(X, w) \in P$ at row X and column T

$T \in \text{FIRST}(w)$

$\text{nullable}(w) \wedge T \in \text{FOLLOW}(X)$

Predictive parsing
filling the table

entry $(X, w) \in P$ at row X and column T

$T \in$ FIRST(w) — letters that w can start with

nullable$(w) \wedge T \in$ FOLLOW(X)

Predictive parsing
filling the table

entry $(X, w) \in P$ at row X and column T

$T \in \text{FIRST}(w)$

$\text{nullable}(w) \wedge T \in \text{FOLLOW}(X)$

$w \Rightarrow_G^* \varepsilon$

Predictive parsing
filling the table

entry $(X, w) \in P$ at row X and column T

$T \in \text{FIRST}(w)$

$\text{nullable}(w) \wedge T \in \text{FOLLOW}(X)$ — letters that can follow X

Predictive parsing
nullable

nullable(X)

$(X, \varepsilon) \in P \Rightarrow \text{nullable}(X)$

$(X_0, X_1 \ldots X_k) \in P \wedge \text{nullable}(X_1) \wedge \ldots \wedge \text{nullable}(X_k) \Rightarrow \text{nullable}(X_0)$

nullable(w)

nullable(ε)

$\text{nullable}(X_1 \ldots X_k) = \text{nullable}(X_1) \wedge \ldots \wedge \text{nullable}(X_k)$

Predictive parsing

first sets

FIRST(X)

$X \in \Sigma$: FIRST(X) = {X}

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \wedge \text{nullable}(X_1 \ldots X_i) \Rightarrow \text{FIRST}(X_0) \supseteq \text{FIRST}(X_{i+1})$

FIRST(w)

FIRST(ε) = {}

¬nullable(X) ⇒ FIRST(Xw) = FIRST(X)

nullable(X) ⇒ FIRST(Xw) = FIRST(X) ∪ FIRST(w)

Predictive parsing

follow sets

FOLLOW(X)

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \wedge \text{nullable}(X_{i+1} \ldots X_k) \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FOLLOW}(X_0)$

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FIRST}(X_{i+1} \ldots X_k)$

Example

p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	nullable	FIRST	FOLLOW
Start			
Exp			
Exp’			
Term			
Term’			
Fact			

Example
nullable

$(X, \varepsilon) \in P \Rightarrow \text{nullable}(X)$

$(X_0, X_1 \ldots X_k) \in P \wedge$

$\text{nullable}(X_1) \wedge \ldots \wedge \text{nullable}(X_k) \Rightarrow \text{nullable}(X_0)$

p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	nullable	FIRST	FOLLOW
Start			
Exp			
Exp’			
Term			
Term’			
Fact			

Example
nullable

$(X, \varepsilon) \in P \Rightarrow \text{nullable}(X)$

$(X_0, X_1 \ldots X_k) \in P \wedge$

$\text{nullable}(X_1) \wedge \ldots \wedge \text{nullable}(X_k) \Rightarrow \text{nullable}(X_0)$

p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	nullable	FIRST	FOLLOW
Start	no		
Exp	no		
Exp’	yes		
Term	no		
Term’	yes		
Fact	no		

Example

FIRST sets

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \wedge$

$\text{nullable}(X_1 \ldots X_i) \Rightarrow \text{FIRST}(X_0) \supseteq \text{FIRST}(X_{i+1})$

p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	nullable	FIRST	FOLLOW
Start	no		
Exp	no		
Exp’	yes		
Term	no		
Term’	yes		
Fact	no		

Example

FIRST sets

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \wedge$

$\text{nullable}(X_1 \ldots X_i) \Rightarrow \text{FIRST}(X_0) \supseteq \text{FIRST}(X_{i+1})$

p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	nullable	FIRST	FOLLOW
Start	no	Num (	
Exp	no	Num (	
Exp’	yes	+	
Term	no	Num (	
Term’	yes	*	
Fact	no	Num (	

Example
FOLLOW sets

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \wedge$

$\text{nullable}(X_{i+1} \ldots X_k) \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FOLLOW}(X_0)$

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FIRST}(X_{i+1} \ldots X_k)$

p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	nullable	FIRST	FOLLOW
Start	no	Num (	
Exp	no	Num (	
Exp’	yes	+	
Term	no	Num (	
Term’	yes	*	
Fact	no	Num (	

Example
FOLLOW sets

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \wedge$

$\text{nullable}(X_{i+1} \ldots X_k) \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FOLLOW}(X_0)$

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FIRST}(X_{i+1} \ldots X_k)$

p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	nullable	FIRST	FOLLOW
Start	no	Num (	
Exp	no	Num (	) EOF
Exp’	yes	+	) EOF
Term	no	Num (	+) EOF
Term’	yes	*	+) EOF
Fact	no	Num (	* +) EOF

Example

LL parse table

entry (X, w)∈P at row X and column T

T∈ FIRST(w)

nullable(w) ∧ T∈ FOLLOW(X)

p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	+	*	Num	(	)	EOF
Start						
Exp						
Exp’						
Term						
Term’						
Fact						

Example

LL parse table

entry (X, w)∈P at row X and column T

T∈ FIRST(w)

nullable(w) ∧ T∈ FOLLOW(X)

p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	+	*	Num	(	)	EOF
Start			p0	p0		
Exp			p1	p1		
Exp’	p2				p3	p3
Term			p4	p4		
Term’	p6	p5			p6	p6
Fact			p7	p8		

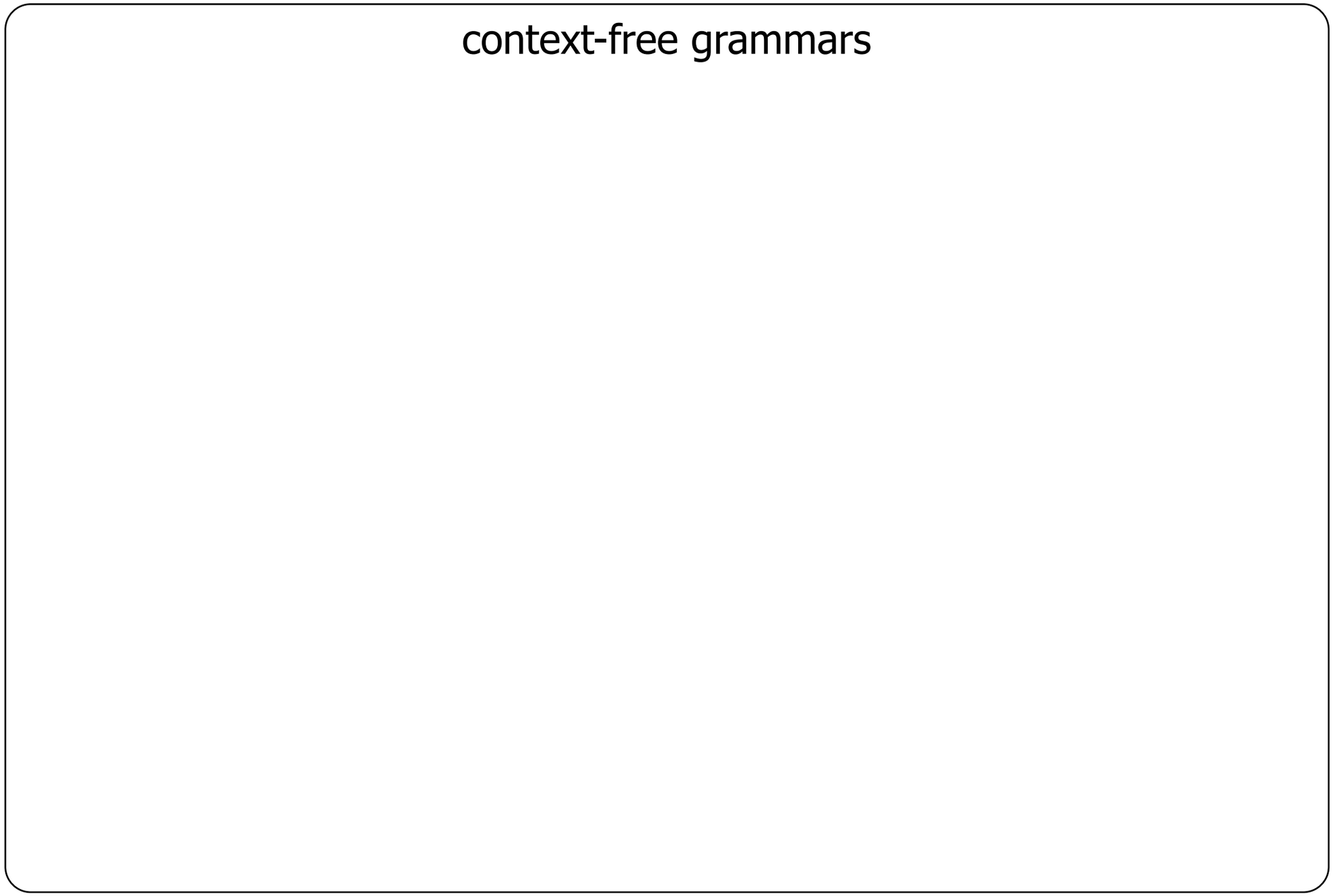
Example

parsing (4+3)*6 EOF

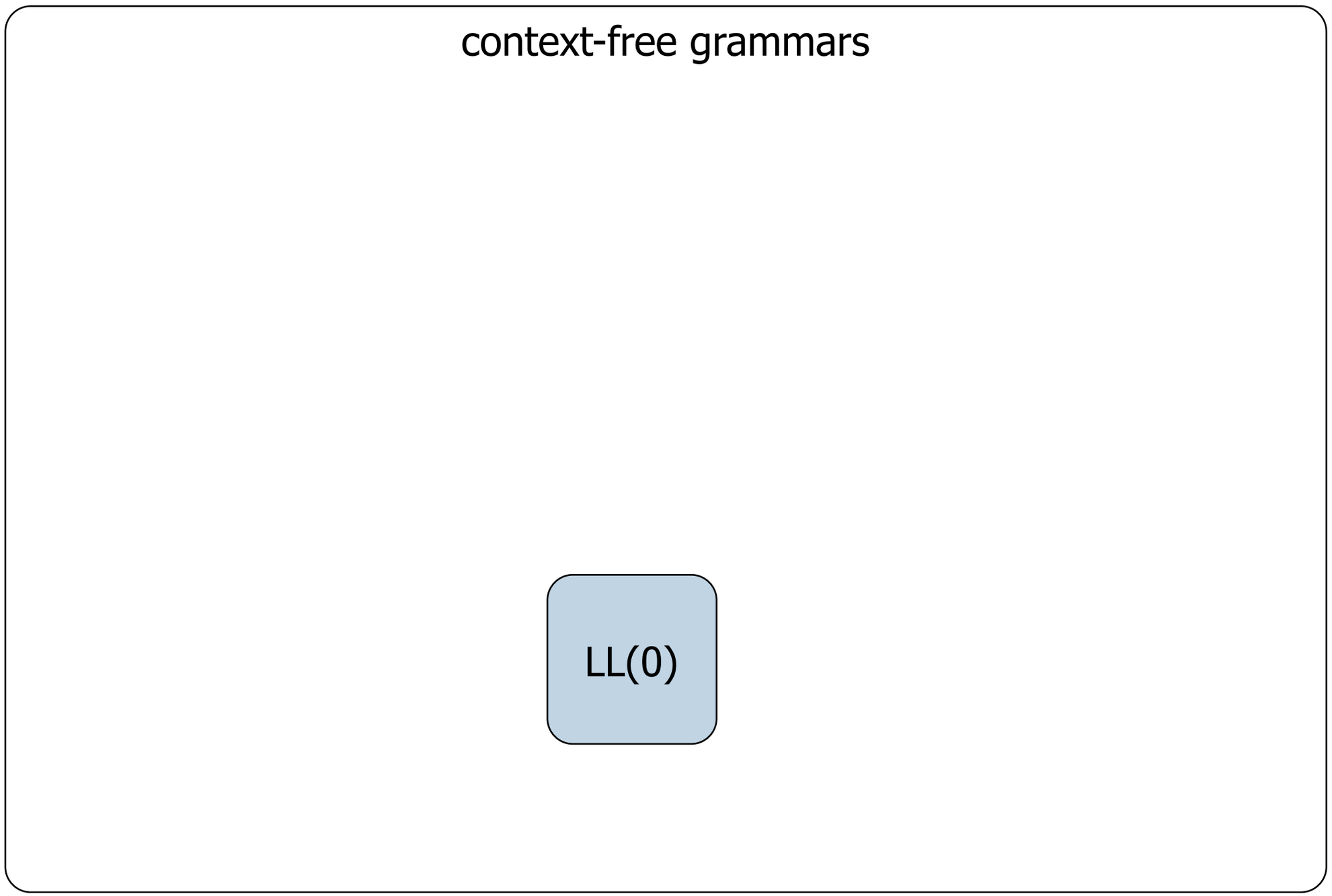
p0: Start → Exp EOF
p1: Exp → Term Exp’
p2: Exp’ → “+” Term Exp’
p3: Exp’ →
p4: Term → Fact Term’
p5: Term’ → “*” Fact Term’
p6: Term’ →
p7: Fact → Num
p8: Fact → “(” Exp “)”

	+	*	Num	(	)	EOF
Start			p0	p0		
Exp			p1	p1		
Exp’	p2				p3	p3
Term			p4	p4		
Term’	p6	p5			p6	p6
Fact			p7	p8		

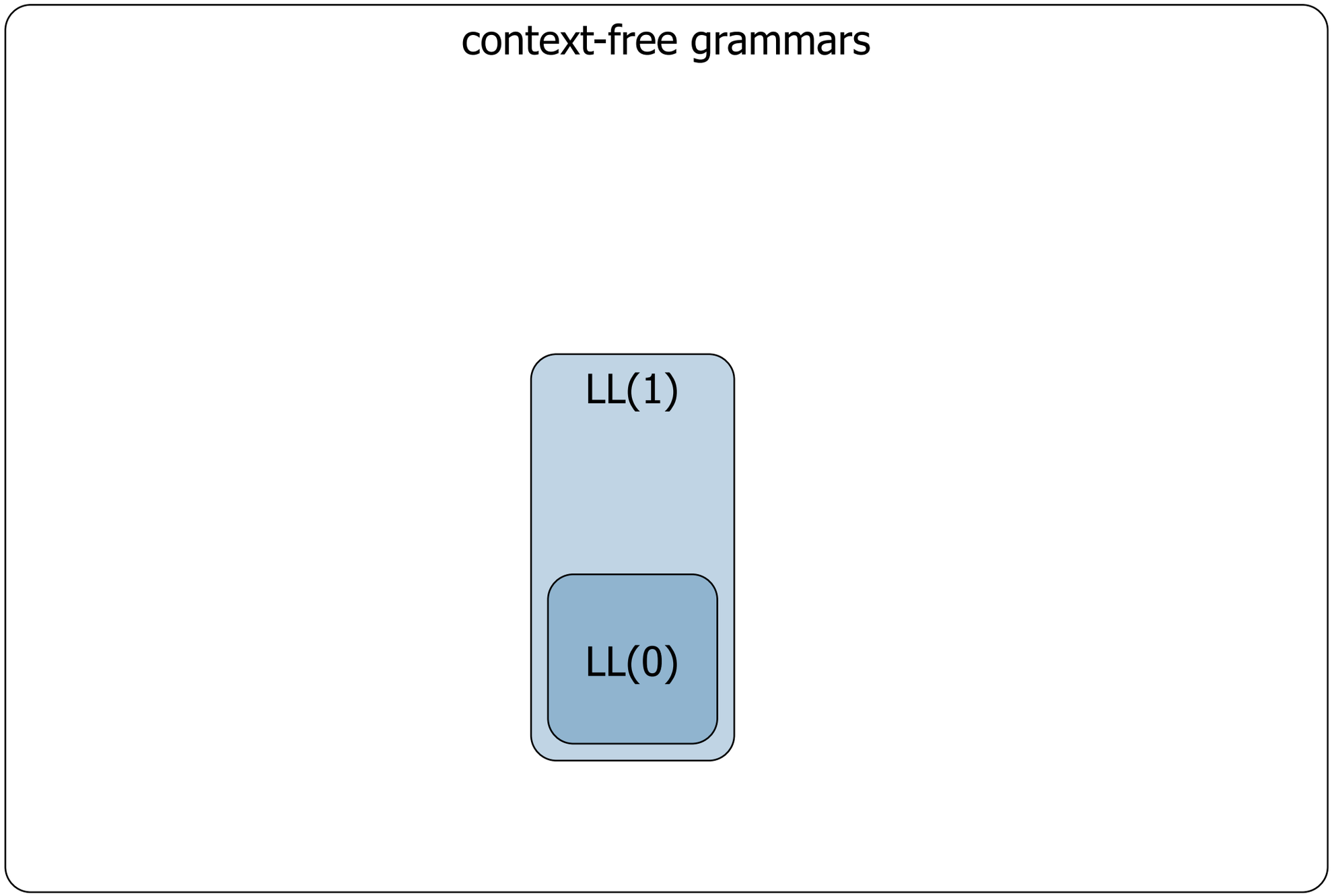
Grammar classes



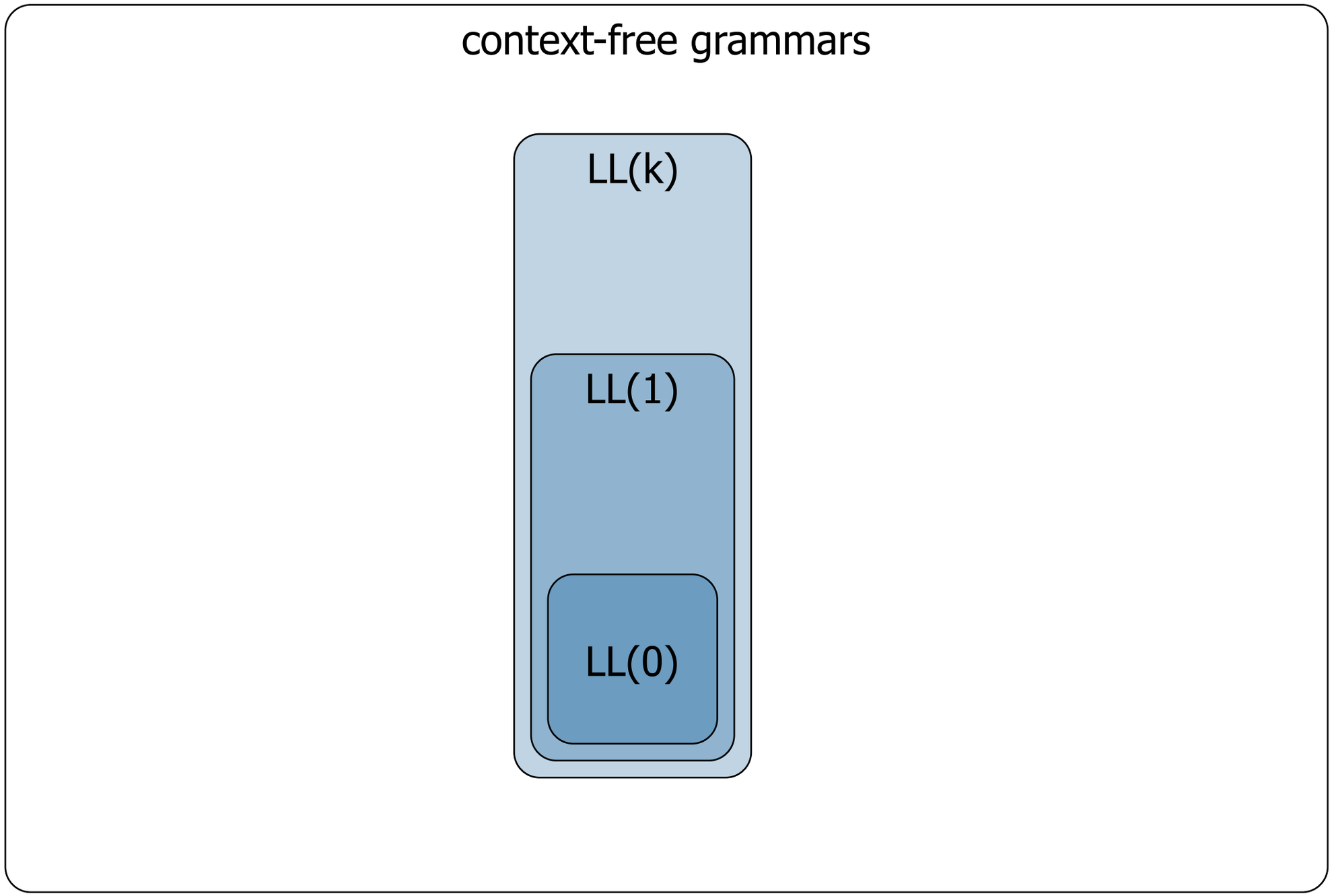
Grammar classes



Grammar classes



Grammar classes



Predictive parsing
encoding precedence

Exp → Num
Exp → "(" Exp ")"
Exp → Exp "*" Exp
Exp → Exp "+" Exp

Fact → Num
Fact → "(" Exp ")"
Term → Term "*" Fact
Term → Fact
Exp → Exp "+" Term
Exp → Term

Predictive parsing
eliminating left recursion

Term → Term "*" Fact
Term → Fact
Exp → Exp "+" Term
Exp → Term

Term′ → "*" Fact Term′
Term′ →
Term → Fact Term′
Exp′ → "+" Term Exp′
Exp′ →
Exp → Term Exp′

Predictive parsing
left factoring

Exp → “if” Exp “then” Exp “else” Exp
Exp → “if” Exp “then” Exp

Exp → “if” Exp “then” Exp Else
Else → “else” Exp
Else →

III

Parser Combinators

Parser Combinators

- Parsers are modeled as functions, and larger parsers are built from smaller ones using higher-order functions.

- Can be implemented in any lazy functional language with higher-order style type system (e.g. Haskell, Scala).

- Parser combinators enable a recursive descent parsing strategy that facilitates modular piecewise construction and testing.

Parser Combinators

```
type Parser = String -> Tree
```

Parser Combinators

What about intermediate parsers?

type Parser = String -> Tree

Parser Combinators

Return the remainder of the input!

~~type Parser = String -> Tree~~

type Parser = String -> (Tree, String)

Parser Combinators

What about parsers that fail?

~~type Parser = String -> Tree~~

type Parser = String -> (Tree, String)

Parser Combinators

Returns either a singleton or
an empty list!

~~type Parser = String -> Tree~~

~~type Parser = String -> (Tree, String)~~

type Parser = String -> [(Tree, String)]

Parser Combinators

~~type Parser = String -> Tree~~

~~type Parser = String -> (Tree, String)~~

~~type Parser = String -> [(Tree, String)]~~

type Parser a = String -> [(a, String)]

A parser p returns either a singleton list [(a, String)] indicating success, or an empty list [] indicating failure. Returning a list of results, also allow to express ambiguities.

Primitive Parsers

success

```
success :: a -> Parser a
success v = \inp -> [(v, inp)]
```

or

```
success :: a String -> [(a, String)]
success v inp -> [(v, inp)]
```

fail

```
fail :: Parser a
fail = \inp -> []
```

match

```
match :: Parser Char
match = \inp -> case inp of
                  []     -> []
                  (x:xs) -> [(x,xs)]
```

Parser Combinators

alternation

```
alt :: Parser a -> Parser a -> Parser a
p 'alt' q = \inp -> (p inp ++ q inp)
```

sequencing

```
seq :: Parser a -> Parser b -> Parser (a, b)
p ‘seq‘ q = \inp -> [((v,w), inp’’) | (v, inp’) <- p inp
                                    , (w, inp’’) <- q inp’]
```

Parser Combinators

- Parser combinators can use semantic actions to manipulate intermediate results, being able to produce values rather than trees.

- More powerful combinators can be build to allow parsing context-sensitive languages (by passing the context together with parsing results).

- Parser Combinators in Scala:
  A. Moors, F. Piessens, Martin Odersky. Parser combinators in Scala. Technical Report Department of Computer Science, K.U. Leuven, February 2008.

IV

Parsing Expression Grammars

Parsing Expression Grammars (PEGs)

Rules are of the form:

$$A \leftarrow \mathbf{e}$$

where A is a non-terminal and **e** is a parsing expression.

PEG operators

Operator	Type	Precedence	Description
' '	primary	5	Literal string
" "	primary	5	Literal string
[ ]	primary	5	Character class
.	primary	5	Any character
(e)	primary	5	Grouping
$e?$	unary suffix	4	Optional
$e*$	unary suffix	4	Zero-or-more
$e+$	unary suffix	4	One-or-more
$\&e$	unary prefix	3	And-predicate
$!e$	unary prefix	3	Not-predicate
$e_1\ e_2$	binary	2	Sequence
e_1 / e_2	binary	1	Prioritized Choice

Syntactic Predicates

&**e** and !**e** : preserves the knowledge whether e fails or not.

Comment ← '//' (!EndOfLine .)* EndOfLine

Parsing Expression Grammars

- Prioritized choice '/'.

S ← **a b** / **a**

S ← **a** / **a b**

Parsing Expression Grammars

- Prioritized choice '/'.

S ← **a b** / **a** ≠ S ← **a** / **a b**

- Backtracks when parsing an alternative fails.
- Uses memoization of intermediate results to parse in linear time, called packrat parsing.
- Does not allow left-recursion.

V

ANTLR / ALL(*)

LL(*)

- Parses LL-regular grammars: for any given non-terminal, parsers can use the entire remaining of the input to differentiate the alternative productions.

- Statically builds a DFA from the grammar productions to recognize lookahead.

- When in conflict, chooses the first alternative and backtracks when DFA lookahead recognition fails.

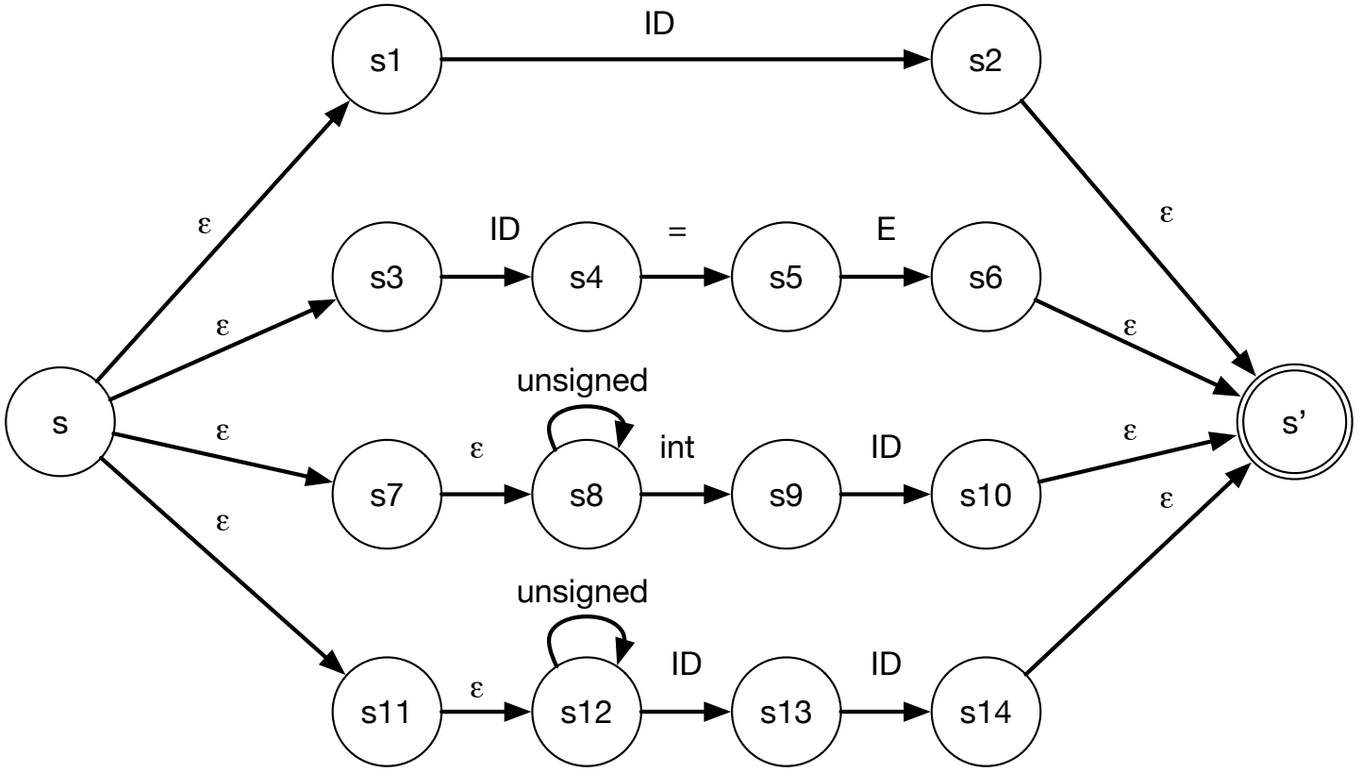
LL(*) Grammar Analysis

S : ID
  | ID **=** E
  | **unsigned*** **int** ID
  | **unsigned*** ID ID

LL(*) Grammar Analysis

terminals: ID, **=**, **int**, **unsigned**

S : ID (1)
| ID **=** E (2)
| **unsigned*** **int** ID (3)
| **unsigned*** ID ID (4)

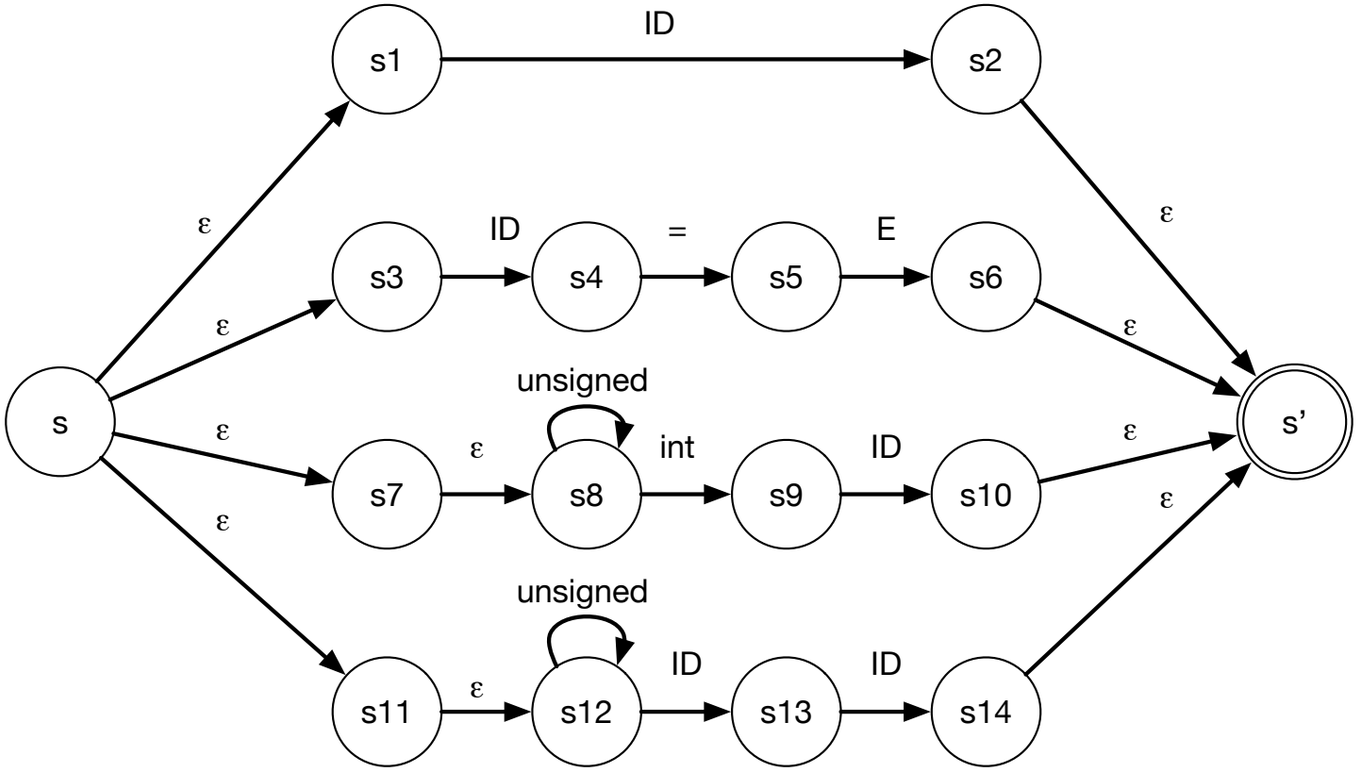


augmented recursive transition network (ATN)

LL(*) Grammar Analysis

terminals: ID, **=**, **int**, **unsigned**

S : ID (1)
| ID **=** E (2)
| **unsigned*** **int** ID (3)
| **unsigned*** ID ID (4)



converts ATN to DFA

(i) Initial state = closure(s);

(ii) Create a final state for each production;

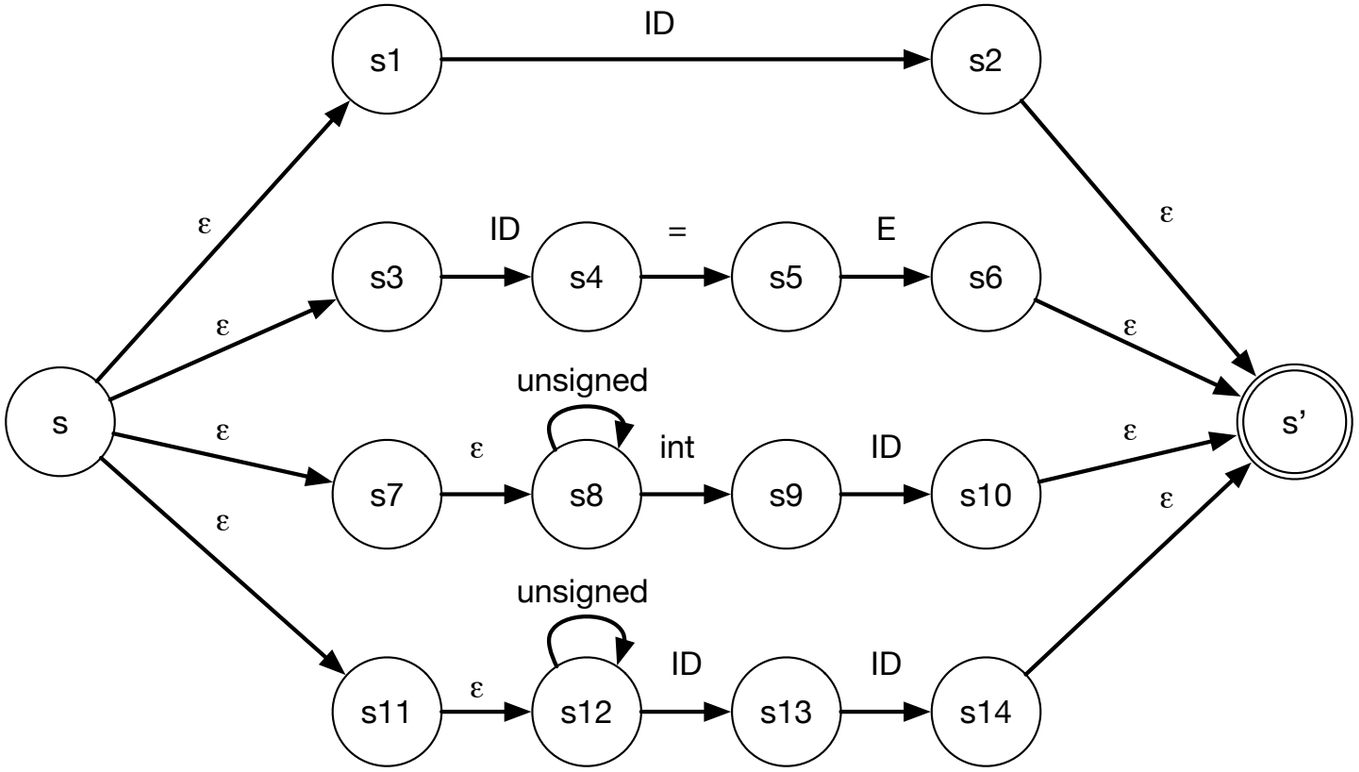
(iii) While DFA state contains ATN states from multiple paths, create transitions with terminals (or EOF if DFA contains s');

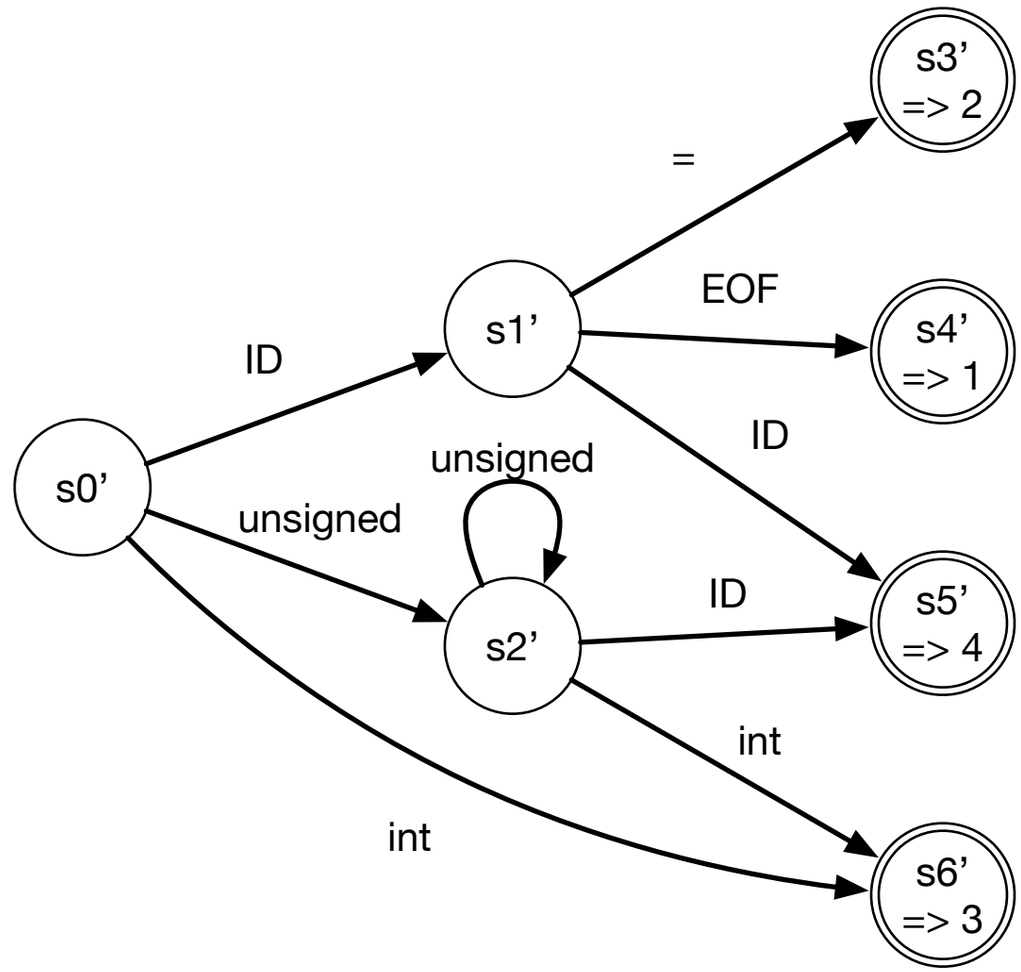
(iv) If DFA state contain ATN states from a single path, add transition to corresponding final DFA state;

LL(*) Grammar Analysis

terminals: ID, **=**, **int**, **unsigned**

S : ID (1)
| ID **=** E (2)
| **unsigned*** **int** ID (3)
| **unsigned*** ID ID (4)





Adaptive LL(*)

```
S : E = E;
  | E;

E : E * E
  | E + E
  | E ( E )
  | ID
```

```
void S() {
  switch (adaptivePredict("S", callStack)){
    case 1 :
      E(); match('='); E();
      match(';'); break;
    case 2:
      E(); match(';'); break;
  }
}
```

Adaptive LL(*)

Dynamically builds the lookahead DFA.

```
S : E = E ;
  | E ;

E : E * E
  | E + E
  | E ( E )
  | ID
```

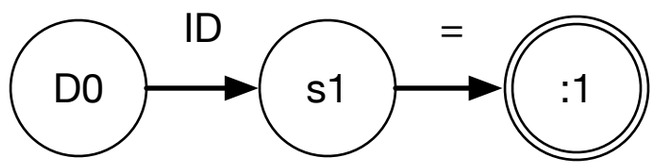
terminals: **=**, **;**, ***, **+**, **(**, **)**, ID

Adaptive LL(*)

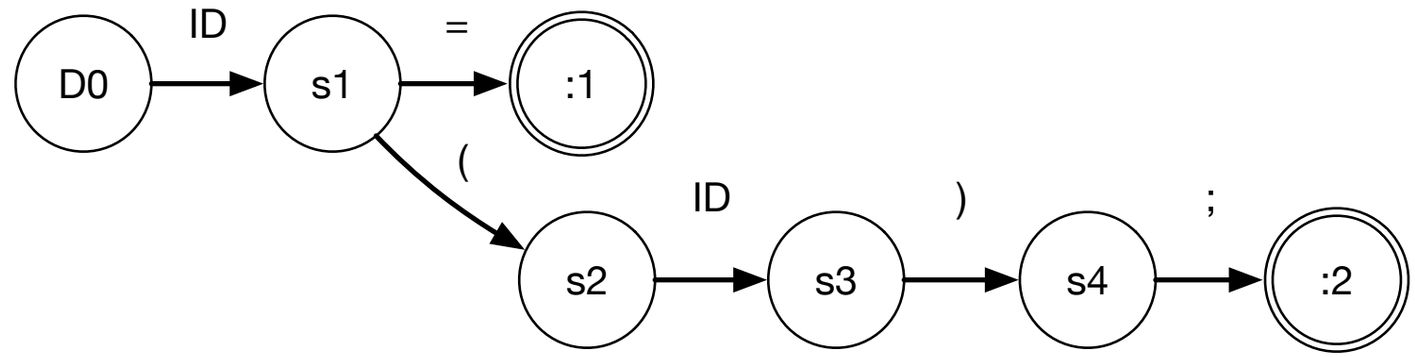
Dynamically builds the lookahead DFA.

After parsing:

x = y;



f(x);



Adaptive LL(*)

parse: **yba**

S : **x** B
  | **y** C

B : A **a**

C : A **b a**

A : **b**
  | ε

Adaptive LL(*)

S : **x** B
  | **y** C

B : A **a**

C : A **b a**

A : **b**
  | ε

input: **yba**

stack: S

Adaptive LL(*)

S : **x** B
  | **y** C

B : A **a**

C : A **b a**

A : **b**
  | ε

input: **yba**

stack: **y**C

Adaptive LL(*)

S : **x** B
  | **y** C

B : A **a**

C : A **b a**

A : **b**
  | ε

input: **ba**

stack: C

Adaptive LL(*)

S : **x** B
  | **y** C

B : A **a**

C : A **b a**

A : **b**
  | ε

input: **ba**

stack: A **ba**

Adaptive LL(*)

S : **x** B
  | **y** C

B : A **a**

C : A **b a**

A : **b** A_1
  | ε A_2

input: **ba**

stack: A **ba**

b in First(A)
b in Follow(A)

A_1 or A_2?

Adaptive LL(*)

- Tries to parse using **strong LL** (without looking at the call stack), when there are multiple alternatives, splits the parser to try all possibilities.

- Parsing continues with a graph, handling multiple stacks in paralell to explore all alternatives.

- Eventually, if the grammar is unambiguous, only one stack "survives". If multiple stacks survive, an ambiguty has been found and the parser picks the production with lower value.

ANTLR 4

- Accepts as input any context-free grammar that does not contain indirect or hidden left-recursion.
- Generates ALL(*) parses in Java or C#.
- Grammars contain lexical and syntactic rules and generate a lexer and a parser.
- ANTLR4 grammars can be scannerless and composable by using individual characters as input symbols.

VI

Summary

Summary

lessons learned

Summary
lessons learned

How can we parse context-free languages effectively?

- predictive parsing algorithms

Summary
lessons learned

How can we parse context-free languages effectively?

- predictive parsing algorithms

Which grammar classes are supported by these algorithms?

- LL(k) grammars, LL(k) languages

Summary
lessons learned

How can we parse context-free languages effectively?

- predictive parsing algorithms

Which grammar classes are supported by these algorithms?

- LL(k) grammars, LL(k) languages

How can we generate compiler tools from that?

- implement automaton
- generate parse tables

Summary
lessons learned

How can we parse context-free languages effectively?

- predictive parsing algorithms

Which grammar classes are supported by these algorithms?

- LL(k) grammars, LL(k) languages

How can we generate compiler tools from that?

- implement automaton
- generate parse tables

What are other techniques for implementing top-down parsers?

- Parser Combinators
- PEGs
- ALL(*)

Literature

learn more

Literature

learn more

formal languages

Noam Chomsky: Three models for the description of language. 1956

J. E. Hopcroft, R. Motwani, J. D. Ullman: Introduction to Automata Theory, Languages, and Computation. 2006

Literature

learn more

formal languages

Noam Chomsky: Three models for the description of language. 1956

J. E. Hopcroft, R. Motwani, J. D. Ullman: Introduction to Automata Theory, Languages, and Computation. 2006

syntactical analysis

Andrew W. Appel, Jens Palsberg: Modern Compiler Implementation in Java, 2nd edition. 2002

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Monica S. Lam: Compilers: Principles, Techniques, and Tools, 2nd edition. 2006

Literature

learn more

Literature

learn more

ALL(*)

> Terence John Parr, Sam Harwell, Kathleen Fisher. Adaptive LL(*) parsing: the power of dynamic analysis. In OOPSLA 2014.

Literature

learn more

ALL(*)

> Terence John Parr, Sam Harwell, Kathleen Fisher. Adaptive LL(*) parsing: the power of dynamic analysis. In OOPSLA 2014.

Parsing Expression Grammars

> Bryan Ford. Parsing Expression Grammars: a recognition-based syntactic foundation. In POPL 2004.

Literature

learn more

ALL(*)

Terence John Parr, Sam Harwell, Kathleen Fisher. Adaptive LL(*) parsing: the power of dynamic analysis. In OOPSLA 2014.

Parsing Expression Grammars

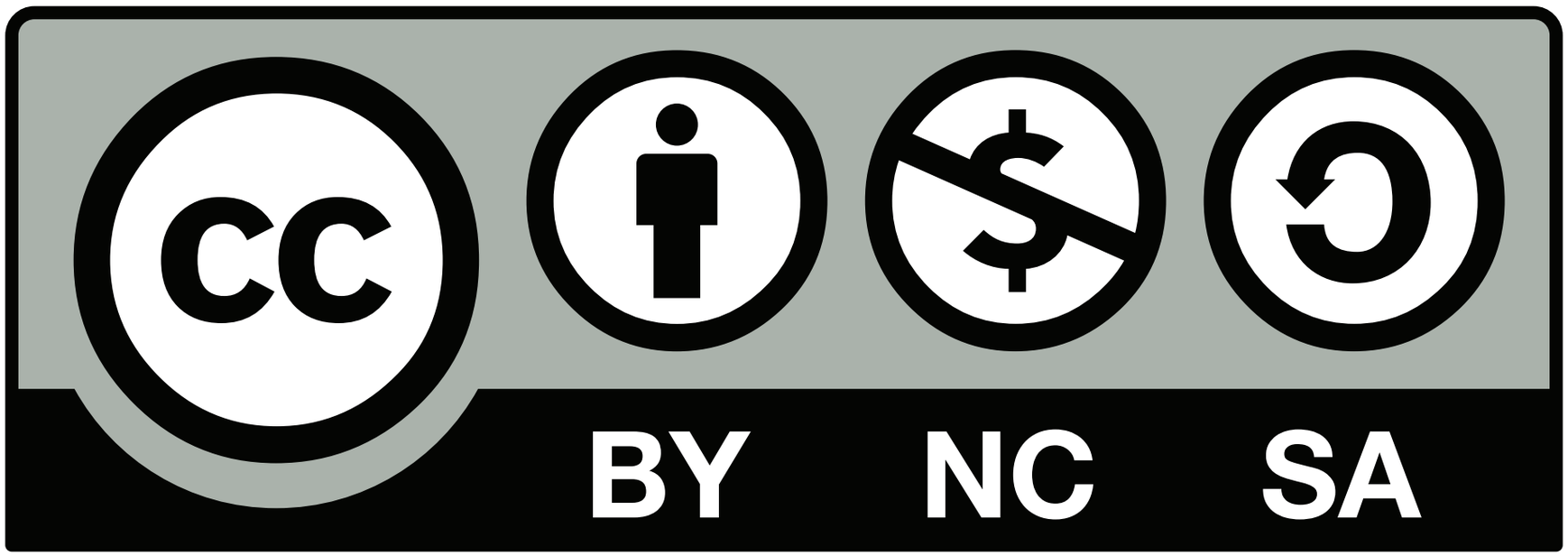
Bryan Ford. Parsing Expression Grammars: a recognition-based syntactic foundation. In POPL 2004.

Parser Combinators

Graham Hutton. Higher-Order Functions for Parsing. Journal of Functional Programming, 1992.

A. Moors, F. Piessens, Martin Odersky. Parser combinators in Scala. Technical Report Department of Computer Science, K.U. Leuven, February 2008.

copyrights


BY
NC
SA

Pictures

copyrights

Slide 1: Book Scanner by Ben Woosley, some rights reserved

Slides 5-7: Noam Chomsky by Fellowsisters, some rights reserved

Slide 8, 9, 26-28: Tiger by Bernard Landgraf, some rights reserved

Slide 32: Ostsee by Mario Thiel, some rights reserved