LR(0) Parsers

CSCI 3130 Formal Languages and Automata Theory

Siu On CHAN

Fall 2019

Chinese University of Hong Kong

Parsing computer programs

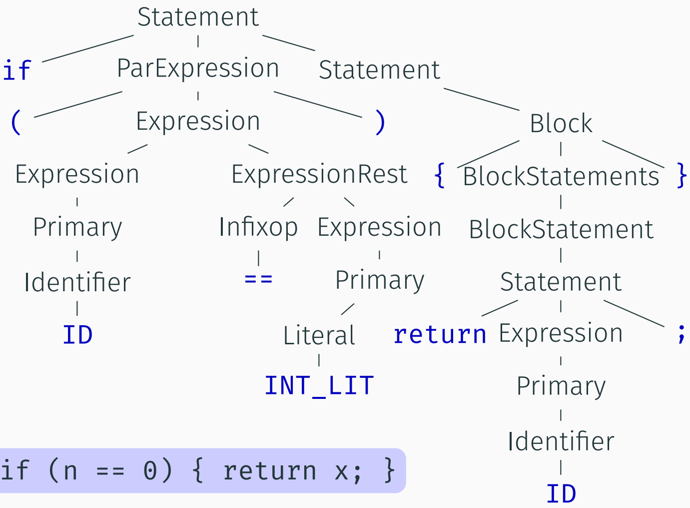
```
if (n == 0) { return x; }
```

First phase of `javac` compiler: lexical analysis

`if` `(` `ID` `==` `INT_LIT` `)` `{` `return` `ID` `;` `}`

The alphabet of Java CFG consists of tokens like

$$\Sigma = \{\texttt{if}, \texttt{return}, \texttt{(}, \texttt{)}, \texttt{\{}, \texttt{\}}, \texttt{;}, \texttt{==}, \texttt{ID}, \texttt{INT\_LIT}, \ldots\}$$




```
if (n == 0) { return x; }
```

Parse tree of a Java statement

CFG of the java programming language

```
Identifier:
   IdentifierChars but not a Keyword or BooleanLiteral or
NullLiteral
Literal:
   IntegerLiteral
   FloatingPointLiteral
   BooleanLiteral
   CharacterLiteral
   StringLiteral
   NullLiteral
Expression:
   LambdaExpression
   AssignmentExpression

AssignmentOperator:
   (one of) = *= /= %= += -= <<= >>= >>>= &= ^= |=
```

from http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html#52996

Parsing Java programs

```
class Point2d {
    /* The X and Y coordinates of the point--instance variables */
    private double x;
    private double y;
    private boolean debug;    // A trick to help with debugging

    public Point2d (double px, double py) { // Constructor
        x = px;
        y = py;

        debug = false;     // turn off debugging
    }

    public Point2d () {   // Default constructor
        this (0.0, 0.0);                    // Invokes 2 parameter Point2D constructor
    }
    // Note that a this() invocation must be the BEGINNING of
    // statement body of constructor

    public Point2d (Point2d pt) {             // Another consructor
        x = pt.getX();
        y = pt.getY();
    }
  ...
}
```

Simple Java program: about 1000 tokens

Parsing algorithms

How long would it take to parse this program?

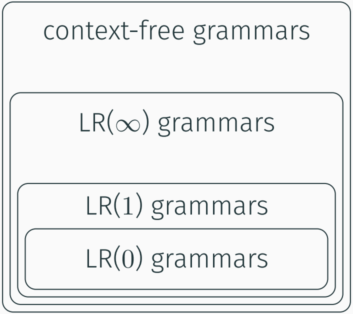
try all parse trees	$\geqslant 10^{80}$ years
CYK algorithm	hours

Can we parse faster?

CYK is the fastest known general-purpose parsing algorithm for CFGs

Luckily, some CFGs can be rewritten to allow for a faster parsing algorithm!

Hierarchy of context-free grammars



Java, Python, etc have LR(1) grammars

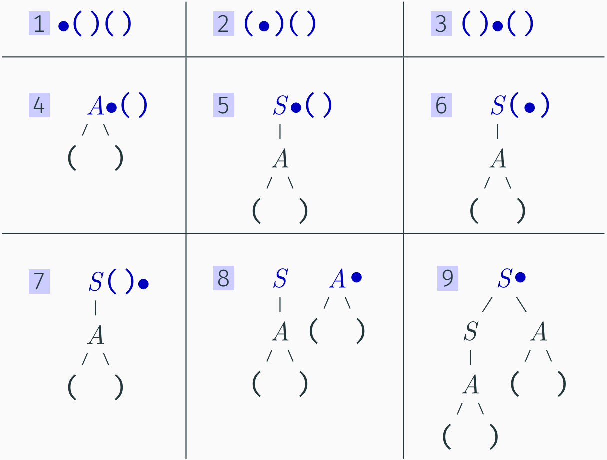
We will describe LR(0) parsing algorithm

A grammar is LR(0) if LR(0) parser works correctly for it

LR(0) parser: overview

$$S \to SA \mid A$$
$$A \to (\,S\,) \mid (\,)$$

input: ( )( )



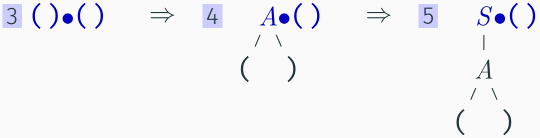
LR(0) parser: overview

$$S \to SA \mid A$$
$$A \to (\,S\,) \mid (\,)$$

input: ( )( )

Features of LR(0) parser:

- Greedily reduce the recently completed rule into a variable
- Unique choice of reduction at any time



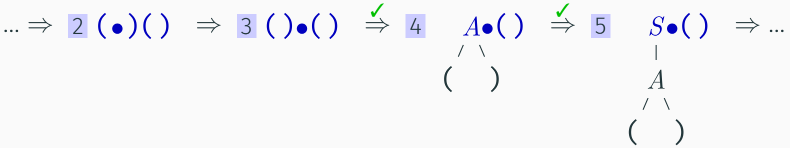
LR(0) parsing using a PDA

To speed up parsing, keep track of partially completed rules in a PDA P

In fact, the PDA will be a simple modification of an NFA N

The NFA accepts if a rule $B \to \beta$ has just been completed

and the PDA will reduce β to B



✓: NFA N accepts

NFA acceptance condition

$$S \to SA \mid A$$
$$A \to (\,S\,) \mid (\,)$$

A rule $B \to \beta$ has just been completed if

Case 1 input/buffer so far is exactly β

Examples: 3 $(\,)\bullet(\,)$ and 4 $A\bullet(\,)$

```
A•( )
/ \
(   )
```

Case 2 Or buffer so far is $\alpha\beta$ and there is another rule $C \to \alpha B\gamma$

Example: 7 $S(\,)\bullet$

```
S( )•
|
A
/ \
(   )
```

This case can be chained

Designing NFA for Case 1

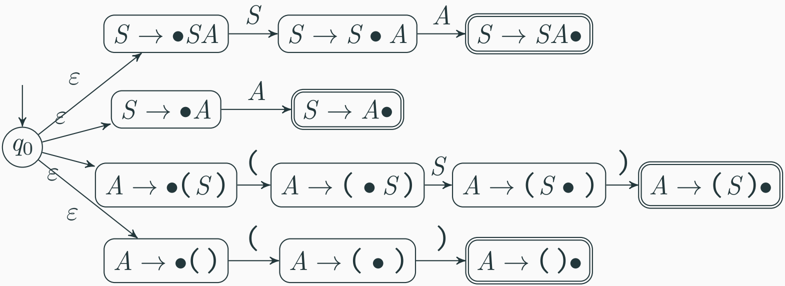
$$S \to SA \mid A$$
$$A \to (S) \mid ()$$

Design an NFA N' to accept the right hand side of some rule $B \to \beta$

Designing NFA for Case 1

$$S \to SA \mid A$$
$$A \to (S) \mid ()$$

Design an NFA N' to accept the right hand side of some rule $B \to \beta$



Designing NFA for Cases 1 & 2

$$S \to SA \mid A$$
$$A \to (\, S\,) \mid (\,)$$

Design an NFA N to accept $\alpha\beta$ for some rules $C \to \alpha B\gamma, \quad B \to \beta$
and for longer chains

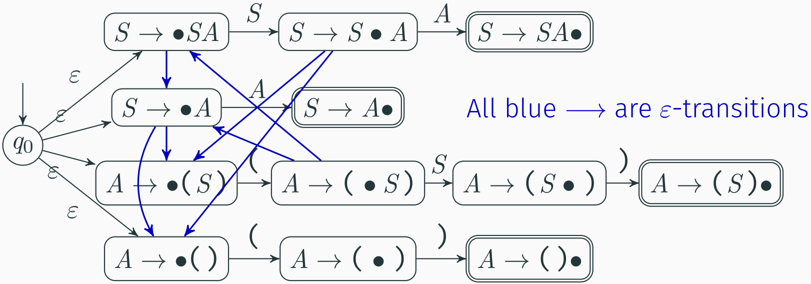
Designing NFA for Cases 1 & 2

$S \to SA \mid A$

$A \to (\,S\,) \mid (\,)$

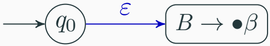
Design an NFA N to accept $\alpha\beta$ for some rules $C \to \alpha B \gamma, \quad B \to \beta$
and for longer chains

For every rule $C \to \alpha B\gamma$, $B \to \beta$, add $\boxed{C \to \alpha \bullet B\gamma} \xrightarrow{\varepsilon} \boxed{B \to \bullet\beta}$

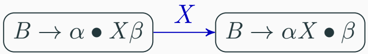


Summary of the NFA

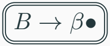
For every rule $B \to \beta$, add



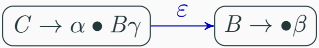
For every rule $B \to \alpha X \beta$ (X may be terminal or variable), add



Every completed rule $B \to \beta$ is accepting



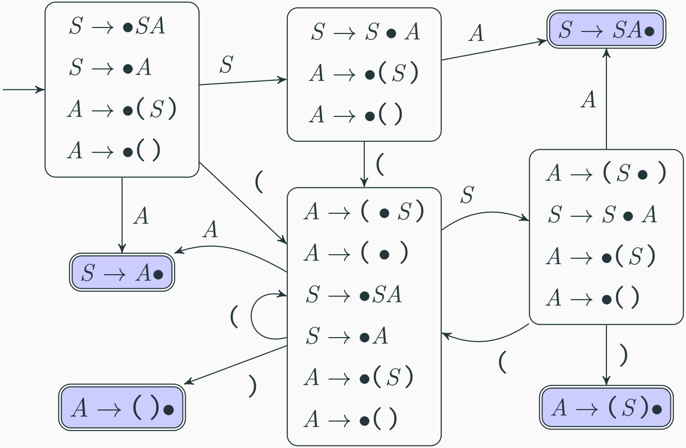
For every rule $C \to \alpha B \gamma$, $B \to \beta$, add



The NFA N will accept whenever a rule has just been completed

Equivalent DFA D for the NFA N

Dead state (empty set) not shown for clarity



Observation: every accepting state contains only one rule:

a completed rule $B \to \beta\bullet$, and such rules appear only in accepting states

LR(0) grammars

A grammar G is LR(0) if its corresponding D_G satisfies:

Every accepting state contains only one rule:
a completed rule of the form $B \to \beta\bullet$
and completed rules appear only in accepting states

Shift state:

no completed rule

$S \to S \bullet A$

$A \to \bullet(S)$

$A \to \bullet()$

Reduce state:

has (unique) completed rule

$A \to (S)\bullet$

Simulating DFA D

Our parser P simulates state transitions in DFA D

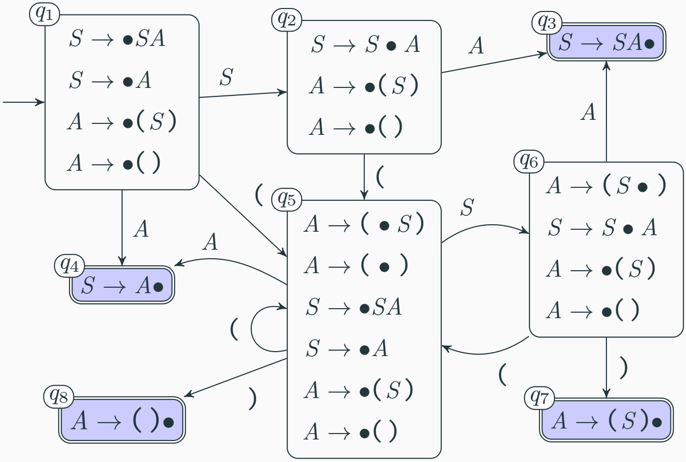
```
(()•)    ⇒    (A•)
               / \
              (   )
```

After reducing ( ) to A, what is the new state?

Solution: keep track of previous states in a stack

go back to the correct state by looking at the stack

Let's label D's states



LR(0) parser: a "PDA" P simulating DFA D

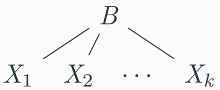
P's stack contains labels of D's states to remember progress of partially completed rules

At D's non-accepting state q_i

1. P simulates D's transition upon reading terminal or variable X
2. P pushes current state label q_i onto its stack

At D's accepting state with completed rule $B \to X_1 \dots X_k$

1. P pops k labels $q_k, \dots, q_1$ from its stack
2. constructs part of the parse tree



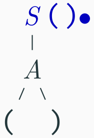
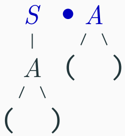
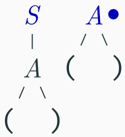
3. P goes to state q_1 (last label popped earlier), pretend next input symbol is B

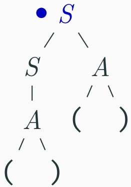
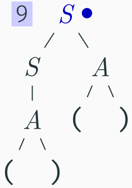
Example

		state	stack
1	$\bullet(\,)(\,)$	q_1	$
2	$(\bullet)(\,)$	q_5	$1
3	$(\,)\bullet(\,)$	q_8	$15
	$\bullet A(\,)$ / \ ( )	q_1	$
4	$A\bullet(\,)$ / \ ( )	q_4	$1
	$\bullet\ S(\,)$ | A / \ ( )	q_1	$

		state	stack
5	$S\bullet(\,)$ | A / \ ( )	q_2	$1
6	$S(\bullet)$ | A / \ ( )	q_5	$12

Example

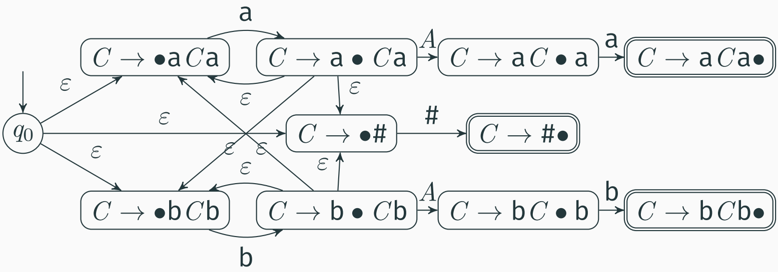
		state	stack
7		q_8	\$125
		q_2	\$1
8		q_3	\$12

		state	stack
		q_1	\$
9		q_2	\$1

parser's output is the parse tree

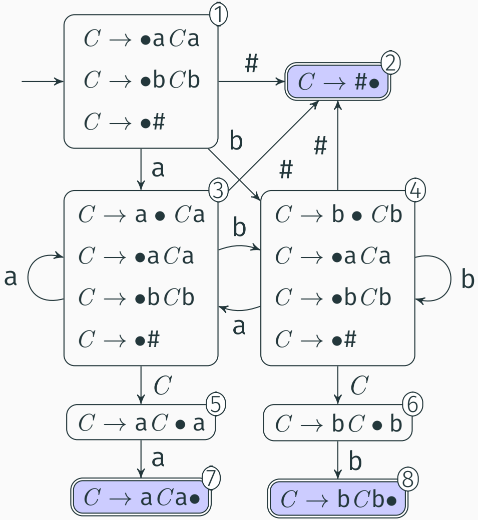
$L = \{w\#w^R \mid w \in \{\mathsf{a}, \mathsf{b}\}^*\}$ $C \to \mathsf{a}C\mathsf{a} \mid \mathsf{b}C\mathsf{b} \mid \#$

NFA N:



Another LR(0) grammar

$C \to \text{a}C\text{a} \mid \text{b}C\text{b} \mid \#$



input: ba#ab

stack	state	action
\$	1	S
\$1	4	S
\$14	3	S
\$143	2	R
\$143	5	S
\$1435	7	R
\$14	6	S
\$146	8	R

Deterministic PDAs

PDA for LR(0) parsing is deterministic

Some CFLs require non-deterministic PDAs, such as

$$L = \{ww^R \mid w \in \{\mathsf{a}, \mathsf{b}\}^*\}$$

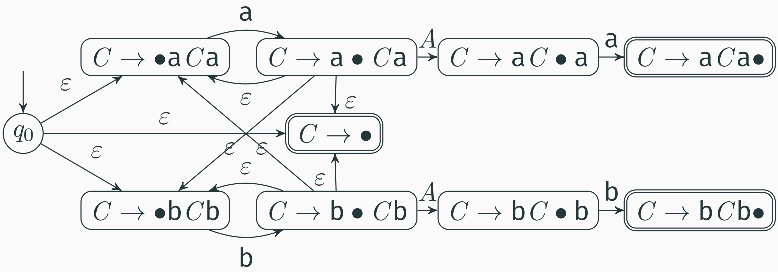
What goes wrong when we do LR(0) parsing on L?

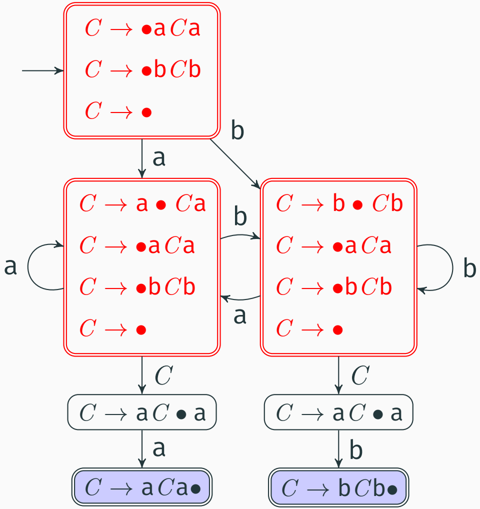
Example 2

$L = \{ww^R \mid w \in \{\mathsf{a},\mathsf{b}\}^*\}$

$C \to \mathsf{a}C\mathsf{a} \mid \mathsf{b}C\mathsf{b} \mid \varepsilon$

NFA N:

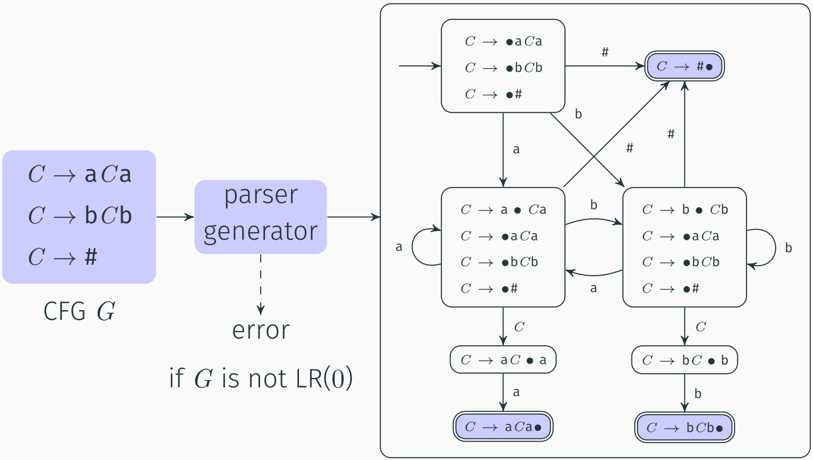




$C \to \mathtt{a}\,C\,\mathtt{a} \mid \mathtt{b}\,C\,\mathtt{b} \mid \varepsilon$

shift-reduce conflicts

Parser generator



“PDA” for parsing G

Motivation: Fast parsing for programming languages

LR(1) Grammar: A few words

LR(0) grammar revisited

LR(1) grammars

LR(0) grammars

LR(0) parser: **L**eft-to-right read, **R**ightmost derivation, **0** lookahead symbol

$$S \to SA \mid A$$
$$A \to (\,S\,) \mid (\,)$$

Derivation

$S \Rightarrow SA \Rightarrow S(\,) \Rightarrow A(\,) \Rightarrow (\,)(\,)$

Reduction (derivation in reverse)

$(\,)(\,) \rightarrowtail A(\,) \rightarrowtail S(\,) \rightarrowtail SA \rightarrowtail S$

LR(0) parser looks for rightmost derivation

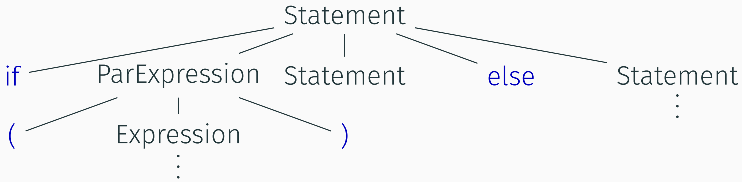
Rightmost derivation $=$ Leftmost reduction

Parsing computer programs

```
if (n == 0) { return x; }
```

Parsing computer programs

```
if (n == 0) { return x; }
else { return x + 1; }
```



CFGs of most programming languages are not LR(0)

LR(0) parser cannot tell apart

`if` ...`then` from `if` ...`then` ...`else`

LR(1) grammar

LR(1) grammars resolve such conflicts by one symbol lookahead

States in NFA N

LR(0):	LR(1):
$A \to \alpha \bullet \beta$	$[A \to \alpha \bullet \beta, a]$

States in DFA D

LR(0):	LR(1):
no shift-reduce conflicts no reduce-reduce conflicts	some shift-reduce conflicts allowed some reduce-reduce conflicts allowed as long as can be resolved with lookahead symbol a

We won't cover LR(1) parser in this class; take CSCI 3180 for details