Introduction to Computer Science

Géza Horváth

Eleventh Lecture

Kuroda normal form for context-sensitive grammars

Definition (Kuroda normal form)

A grammar $G = (N, T, S, P)$ is in Kuroda normal form, if each of its rules is in one of the following forms:

1. $S \to \lambda$,
2. $A \to a$,
3. $A \to B$,
4. $A \to BC$,
5. $AB \to CD$,

where $A, B, C, D \in N$ and $a \in T$). In case $S \to \lambda$ is in the set of productions, the start symbol S cannot be in any right hand side of a rule.

Theorem

There is an equivalent grammar in Kuroda normal form for every context-sensitive grammar.

Révész normal form for unrestricted grammars

Definition (Révész normal form)

The grammar $G = (N, T, S, P)$ is in Révész normal form, if all of its production rules has the following forms:

1. $S \to \lambda$,
2. $A \to a$,
3. $A \to B$,
4. $A \to BC$,
5. $AB \to AC$,
6. $AB \to CB$,
7. $AB \to B$,

where $S, A, B, C \in N$ and $a \in T$.
In case $S \to \lambda$ is in the set of productions, the start symbol S cannot be in any right hand side of a rule.

This normal form for unrestricted grammars was introduced by György Révész.

Révész normal form for unrestricted grammars

Theorem

There is an equivalent grammar in Révész normal form for every grammar.

Compared to the Kuroda normal form, the differences are the following:

- instead of the rule $AB \to CD$ it allows two rules, namely $AB \to AC$ and $AB \to CB$,
- the only ,,really plus'' rule, which makes the difference between the monotone and unrestricted grammars is the last production rule: $AB \to B$.

As you can see, there is not a huge difference between the unrestricted grammars and the context-sensitive grammars. For generative grammars generating context-sensitive languages, only one production rule form is enough to be able to generate any recursively enumerable language.

Results of Geffert

Even more surprising results were proven by Viliam Geffert. His results put limitations not just to the form of the production rules, but also to the number of the nonterminal symbols. We introduce his results as theorems, without proofs.

Theorem

For each recursively enumerable language L we can give an unrestricted generative grammar $G = (N, T, S, H)$ such that

- *grammar G generates the language L, $(L(G) = L)$,*
- *G has exactly 5 nonterminal symbols, $(N = \{S, A, B, C, D\})$, and*
- *each rule has a form:*
  - *$S \to p$ where S is the start symbol, and p is a nonempty word, $(p \in (N \cup T)^+)$,*
  - *$AB \to \lambda$, or*
  - *$CD \to \lambda$.*

Results of Geffert

Theorem

For each recursively enumerable language L we can give an unrestricted generative grammar $G = (N, T, S, H)$ such that

- *grammar G generates the language L, $(L(G) = L)$,*
- *G has exactly 4 nonterminal symbols, $(N = \{S, A, B, C\})$, and*
- *each rule has a form:*
  - *$S \to p$ where S is the start symbol, and p is a nonempty word, $(p \in (N \cup T)^+)$,*
  - *$AB \to \lambda$, or*
  - *$CC \to \lambda$.*

Results of Geffert

Theorem

For each recursively enumerable language L we can give an unrestricted generative grammar $G = (N, T, S, H)$ such that

- *grammar G generates the language L, $(L(G) = L)$,*
- *G has exactly 3 nonterminal symbols, $(N = \{S, A, B\})$, and*
- *each rule has a form:*
  - *$S \to p$ where S is the start symbol, and p is a nonempty word, $(p \in (N \cup T)^+)$,*
  - *$AA \to \lambda$, or*
  - *$BBB \to \lambda$.*

Results of Geffert

Theorem

For each recursively enumerable language L we can give an unrestricted generative grammar $G = (N, T, S, H)$ such that

- *grammar G generates the language L, $(L(G) = L)$,*
- *G has exactly 3 nonterminal symbols, $(N = \{S, A, B\})$, and*
- *each rule has a form:*
  - *$S \to p$ where S is the start symbol, and p is a nonempty word, $(p \in (N \cup T)^+)$, or*
  - *$ABBBA \to \lambda$.*

Results of Geffert

Theorem

For each recursively enumerable language L we can give an unrestricted generative grammar $G = (N, T, S, H)$ such that

- *grammar G generates the language L, $(L(G) = L)$,*
- *G has exactly 4 nonterminal symbols, $(N = \{S, A, B, C\})$, and*
- *each rule has a form:*
  - *$S \to p$ where S is the start symbol, and p is a nonempty word, $(p \in (N \cup T)^+)$, or*
  - *$ABC \to \lambda$.*

Parsing

In formal language theory, parsing – or the so called syntactic analysis – is a process when the parser determines if a given string can be generated by a given grammar. This is very important for compilers and interpreters. For example, it is not too difficult to create a context-free grammar G_P generating all syntactically correct Pascal programs. Then, we can use a parser to decide – about a Pascal program written by a programmer – if the program is in the generated language $L(G_P)$. When the program is in the generated language, it is syntactically correct.

CYK and Early algorithms

We have a given Chomsky normal form grammar $G = (N, T, S, P)$ and a word $p = a_1 a_2 \dots a_n$. The Cocke–Younger–Kasami algorithm is a well known method to decide wether $p \in L(G)$ or $p \notin L(G)$. To make our decision, we have to fill out an $n \times n$ size triangular matrix.

We have a given context-free grammar $G = (N, T, S, P)$ and a word $p = a_1 a_2 \dots a_n$. The Early algorithm is a well known method to decide wether $p \in L(G)$ or $p \notin L(G)$. To make our decision, we have to fill out an $(n+1) \times (n+1)$ size triangular matrix.

Both algorithms work with context-free grammars and run in **polynomial time**.

LL parsers

LL parsers run in **linear time** and work with a proper subset of the deterministic context-free languages.

An LL grammar is a grammar that can be parsed by an LL parser. (There are many context-free grammars that can't be so parsed.)

An LL(1) grammar is a grammar that can be parsed by an LL parser with one symbol of lookahead. That is, if a nonterminal has more than one right-hand side, the parser can decide which one to apply by looking at the next input symbol.

It is possible to have an LL(k) grammar, where k is some number other than 1, and which requires k symbols of lookahead, but such grammars are not very practical.

LL parsers

LL parsers use leftmost derivations. In a leftmost derivation, always the leftmost nonterminal is rewritten.

Parts of LL parsers:

- read-only input tape
- parsing table
- stack memory
- control head

Before the first use, a parsing table must be constructed.

LL(1) parser example

Let the generative grammar G be

$G = (\{S, A\}, \{a, b, c\}, S, P)$
$P = \{$
$S \to aAc,$
$S \to b,$
$A \to aAc,$
$A \to b$
$\}.$

Use the LL(1) parser to show that the grammar G generates the word $aabcc$.

LL(1) parser example – creating the parsing table

First we must create the parsing table based on the grammar G. We have to do it only once for one grammar, then we can use it with different input words.

$G = (\{S, A\}, \{a, b, c\}, S, P)$
$P = \{$
$S \to aAc,$
$S \to b,$
$A \to aAc,$
$A \to b$
$\}.$

LL(1) parser example – creating the parsing table

$G = (\{S, A\}, \{a, b, c\}, S, P)$
$P = \{$
$S \to aAc,$
$S \to b,$
$A \to aAc,$
$A \to b$
$\}.$

The parsing table:

N T	a	b	c
S	$S \to aAc$	$S \to b$	-
A	$A \to aSc$	$A \to b$	-

LL(1) parser example – calculation

The input word: *aabcc*

N T	a	b	c
S	$S \to aAc$	$S \to b$	-
A	$A \to aSc$	$A \to b$	-

The steps of the calculation:

input	stack	rule
aabcc	**S**	-
aabcc	**a**Ac	$S \to aAc$
abcc	**A**c	-
abcc	**a**Scc	$A \to aSc$
bcc	**S**cc	-
bcc	**b**cc	$S \to b$
cc	**c**c	-
c	**c**	-
-	-	-

LL(1) parser example – derivation

The steps of the calculation:

input	stack	rule
aabcc	**S**	-
aabcc	**a**Ac	$S \to aAc$
abcc	**A**c	-
abcc	**a**Scc	$A \to aSc$
bcc	**S**cc	-
bcc	**b**cc	$S \to b$
cc	**c**c	-
c	**c**	-
-	-	-

The derivation: $S \Rightarrow aAc \Rightarrow aaScc \Rightarrow aabcc$

LL(2) parser example – creating the parsing table

$G = (\{S, A, B\}, \{a, b, c\}, S, P)$
$P = \{$
$S \to AB,$
$A \to acA,$
$A \to a,$
$B \to bcB,$
$B \to b$
$\}.$

LL(2) parser example – creating the parsing table

$G = (\{S, A, B\}, \{a, b, c\}, S, P)$
$P = \{$
$S \to AB,$
$A \to acA,$
$A \to a,$
$B \to bcB,$
$B \to b$
$\}.$

The parsing table:

N T	a	b	c	aa	ab	ac	ba	bb	bc	ca	cb	cc
S	-	-	-	-	$S \to AB$	$S \to AB$	-	-	-	-	-	-
A	-	-	-	-	$A \to a$	$A \to acA$	-	-	-	-	-	-
B	-	$B \to b$	-	-	-	-	-	-	$B \to bcB$	-	-	-

LL(2) parser example – creating equivalent LL(1) grammar

$G = (\{S, A, B\}, \{a, b, c\}, S, P)$
$P = \{$
$S \to AB,$
$A \to acA,$
$A \to a,$
$B \to bcB,$
$B \to b$
$\}.$

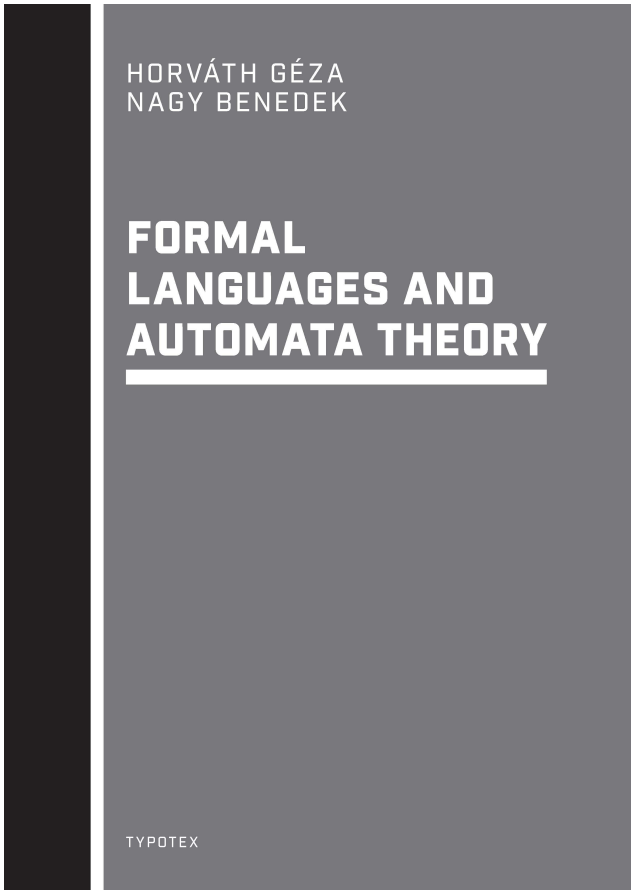
$G' = (\{S, A, B, X, Y\}, \{a, b, c\}, S, P)$
$P = \{$
$S \to AB,$
$A \to aX,$
$X \to cA,$
$X \to \lambda,$
$B \to bY,$
$Y \to cB,$
$Y \to \lambda$
$\}.$

LL(2) parser example – creating parsing table for the equivalent LL(1) grammar

$G' = (\{S, A, B, X, Y\}, \{a, b, c\}, S, P)$
$P = \{$
$S \to AB,$
$A \to aX,$
$X \to cA,$
$X \to \lambda,$
$B \to bY,$
$Y \to cB,$
$Y \to \lambda$
$\}.$

N $T \cup \{\lambda\}$	a	b	c	λ
S	$S \to AB$	-	-	-
A	$A \to aX$	-	-	-
X	-	$X \to \lambda$	$X \to cA$	-
B	-	$B \to bY$	-	-
Y	-	-	$Y \to cB$	$Y \to \lambda$

References



This work was supported by the construction EFOP-3.4.3-16-2016-00021. The project was supported by the European Union, co-financed by the European Social Fund.