G53CMP: Recap of Basic Formal Language Notions

Henrik Nilsson

University of Nottingham, UK

About These Slides

The following slides give a brief recap on some central notions from the theory of formal languages, along with illustrative examples of specific relevance to G53CMP (including the coursework). This is material that has been covered in G52LAC and should be familiar to students taking G53CMP. This material will thus not be covered in detail in the G53CMP lectures, but is offered here for your convenience if you need to refresh these concepts. You may want to go back the G52LAC lecture notes if you need even more details.

Content

Languages (1)

- A ***symbol*** is a basic indivisible entity.
  Concrete examples of symbols are letters and digits.
- A ***string*** or ***word*** is a finite sequence of juxtapositioned symbols.
  For example: a, b, and c are symbols and $abcb$ is a string.
- An ***alphabet*** is a finite set of symbols.
  For example: $\{a, b, c\}$, $\emptyset$.

Languages (2)

- ϵ denotes the word of length 0, the ***empty word***.
- A ***language*** (over alphabet Σ) is a set of words (over alphabet Σ).
  For example: $\Sigma = \{a\}$; one possible language is $L = \{\epsilon, a, aa, aaa\}$.
- Σ^* denotes the set of ***all*** words over an alphabet Σ, including ϵ.

Languages: Examples

alphabet	$\Sigma = \{a, b\}$
words	$\epsilon, a, b, aa, ab, ba, bb,$ $aaa, aab, aba, abb, baa, bab, \ldots$
languages	$\emptyset, \{\epsilon\}, \{a\}, \{b\}, \{a, aa\},$ $\{\epsilon, a, aa, aaa\},$ $\{a^n \mid n \geq 0\},$ $\{a^n b^n \mid n \geq 0, n\,\text{even}\}$

Concatenation of Words

- Concatenation of words is denoted by juxtaposition. For example:
  Concatenation of ab and ba yields $abba$.
- Concatenation is associative and has unit ϵ:

$$u(vw) = (uv)w$$
$$\epsilon u = u = u\epsilon$$

where u, v, w are words.

Concatenation of Languages (1)

Concatenation of words is extended to languages by:

$$MN = \{uv \mid u \in M \wedge v \in N\}$$

Example:

$$\begin{aligned} M &= \{\epsilon, a, aa\} \\ N &= \{b, c\} \\ MN &= \{uv \mid u \in \{\epsilon, a, aa\} \wedge v \in \{b, c\}\} \\ &= \{\epsilon b, \epsilon c, ab, ac, aab, aac\} \\ &= \{b, c, ab, ac, aab, aac\} \end{aligned}$$

Concatenation of Languages (2)

- Concatenation of languages is associative:

$$L(MN) = (LM)N$$

- Concatenation of languages has unit $\{\epsilon\}$:

$$L\{\epsilon\} = L = \{\epsilon\}L$$

- Concatenation distributes through set union:

$$\begin{aligned} L(M \cup N) &= LM \cup LN \\ (L \cup M)N &= LN \cup MN \end{aligned}$$

Context-Free Grammars (1)

A ***Context-Free Grammar*** (CFG) is a way of formally describing ***Context-Free Languages*** (CFL):

- The CFLs captures ideas common in programming languages such as
  - nested structure
  - balanced parentheses
  - matching keywords like `begin` and `end`.
- Most "reasonable" CFLs can be recognised by a fairly simple machine: a ***deterministic pushdown automaton***.

Context-Free Grammars (2)

Thus, describing a programming language by a "reasonable" CFG

- allows context-free constraints to be expressed
- imparts a hierarchical structure to the words in the language
- allows simple and efficient ***parsing***:
  - determining if a word belongs to the language
  - determining its ***phrase structure*** if so.

Context-Free Grammars (3)

A ***Context-Free Grammar*** is a 4-tuple (N, T, P, S) where

- N is a finite set of ***nonterminals***
- T is a finite set of ***terminals*** (the ***alphabet*** of the language being described)
- $N \cap T = \emptyset$ (N and T are disjoint)
- S, the ***start symbol***, is a distinguished element of N
- P is a finite set of productions, written $A \to \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$

Context-Free Grammar: Example

$$G = (\{S, A\}, \{a, b\}, P, S)$$

where P consists of the productions

$$\begin{array}{rcl} S & \to & \epsilon \\ S & \to & aA \\ A & \to & bS \end{array}$$

Context-Free Grammars: Notation

- Productions with the same LHS are usually grouped together. For example, the productions for S from the previous example:

  $$S \to \epsilon \mid aA$$

  This is (roughly) what is known as ***Backus-Naur Form***.
- Another common way of writing productions is

  $$A ::= \alpha$$

The Directly Derives Relation (1)

To formally define the language generated by

$$G = (N, T, P, S)$$

we first define a binary relation $\underset{G}{\Rightarrow}$ on strings over $N \cup T$, read "directly derives in grammar G", being the least relation such that

$$\alpha A \gamma \underset{G}{\Rightarrow} \alpha \beta \gamma$$

whenever $A \to \beta$ is a production in G.
Note: a production can be applied regardless of context, hence ***context-free***.

The Directly Derives Relation (2)

When it is clear which grammar G is involved, we use $\Rightarrow$ instead of $\underset{G}{\Rightarrow}$.

Example: Given the grammar

$$\begin{array}{rcl} S & \to & \epsilon \mid aA \\ A & \to & bS \end{array}$$

we have

$$\begin{array}{rclcrcl} S & \Rightarrow & \epsilon & \quad & aA & \Rightarrow & abS \\ S & \Rightarrow & aA & \quad & SaAaa & \Rightarrow & SabSaa \end{array}$$

The Derives Relation (1)

The relation $\overset{*}{\underset{G}{\Rightarrow}}$, read "derives in grammar G", is the reflexive, transitive closure of $\underset{G}{\Rightarrow}$.

That is, $\overset{*}{\underset{G}{\Rightarrow}}$ is the least relation on strings over $N \cup T$ such that:

- $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ if $\alpha \underset{G}{\Rightarrow} \beta$
- $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$ (reflexive)
- $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ if $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma \wedge \gamma \overset{*}{\underset{G}{\Rightarrow}} \beta$ (transitive)

The Derives Relation (2)

Again, we use $\overset{*}{\Rightarrow}$ instead of $\overset{*}{\underset{G}{\Rightarrow}}$ when G is obvious.

Example: Given the grammar

$$\begin{array}{rcl} S & \to & \epsilon \mid aA \\ A & \to & bS \end{array}$$

we have

$$\begin{array}{rclcrcl} S & \overset{*}{\Rightarrow} & \epsilon & \quad & S & \overset{*}{\Rightarrow} & abS \\ S & \overset{*}{\Rightarrow} & aA & \quad & S & \overset{*}{\Rightarrow} & ababS \\ aA & \overset{*}{\Rightarrow} & abS & \quad & S & \overset{*}{\Rightarrow} & abab \end{array}$$

Language Generated by a Grammar

The language generated by a context-free grammar

$$G = (N, T, P, S)$$

denoted $L(G)$, is defined as follows:

$$L(G) = \{w \mid w \in T^* \ \wedge \ S \underset{G}{\overset{*}{\Rightarrow}} w\}$$

A language L is a ***Context-Free Language*** (CFL) iff $L = L(G)$ for some CFG G.

A string $\alpha \in (N \cup T)^*$ is a ***sentential form*** iff $S \overset{*}{\Rightarrow} \alpha$.

Language Generation: Example

Given the grammar $G = (N = \{S, A\}, T = \{a, b\}, P, S)$ where P are the productions

$$\begin{array}{rcl} S & \rightarrow & \epsilon \mid aA \\ A & \rightarrow & bS \end{array}$$

we have

$$\begin{array}{rcl} L(G) & = & \{(ab)^i \mid i \geq 0\} \\ & = & \{\epsilon, ab, abab, ababab, abababab, \ldots\} \end{array}$$

Equivalence of Grammars

Two grammars G_1 and G_2 are ***equivalent*** iff $L(G_1) = L(G_2)$.

Example:

$$G_1\text{:} \quad \begin{array}{rcl} S & \rightarrow & \epsilon \mid A \\ A & \rightarrow & a \mid aA \end{array} \qquad G_2\text{:} \quad \begin{array}{rcl} S & \rightarrow & A \\ A & \rightarrow & \epsilon \mid Aa \end{array}$$

$L(G_1) = \{a\}^* = L(G_2)$

Note: the equivalence of CFGs is in general ***undecidable***.

Derivation Tree

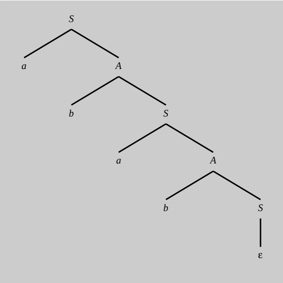
A tree is a ***derivation*** or ***parse tree*** for CFG $G = (N, T, P, S)$ if:

- every vertex has a ***label*** from $N \cup T \cup \{\epsilon\}$
- the label of the root is S
- labels of interior vertices belong to N
- if vertex n has label A and vertices $n_1, n_2, \ldots, n_k$ are the children of n, from left to right, with labels $X_1, X_2, \ldots, X_k$, then $A \rightarrow X_1 X_2 \cdots X_k$ is a production in P
- if a vertex n has label ϵ, then n is a leaf and the only child of its parent.

Derivation Tree: Example

Derivation tree for the string $abab \in L(G)$:

$$G\text{:} \quad \begin{array}{rcl} S & \rightarrow & \epsilon \mid aA \\ A & \rightarrow & bS \end{array}$$



Derivations and Derivation Trees

Given a derivation tree for a grammar G:

- The string of leaf labels read from left to right is the ***yield*** of the tree.
- The yield is a sentential form of G.

The derives relation and derivation trees are related as follows:

> A string α is the yield of some derivation tree for a grammar G iff $S \underset{G}{\overset{*}{\Rightarrow}} \alpha$.

Regular Grammars

- ***Lexical syntax*** is usually defined through ***Regular Languages***.
- The regular languages are a proper subset of the context-free languages.
- Context-free grammars can thus be used to describe regular languages.
- If a grammar G is ***left-linear*** or ***right-linear***, then G is a ***regular grammar*** and $L(G)$ is a regular language.
- Regular languages are easy to recognize (DFA).

Right-linear Grammar

A CFG $G = (N, T, P, S)$ is ***right-linear*** if all its productions are of the forms

$$\begin{array}{rcl} A & \rightarrow & wB \\ A & \rightarrow & w \end{array}$$

where $A, B \in N$ and $w \in T^*$.

Example: The regular language $\mathbf{0}(\mathbf{10})^*$ is generated by the right-linear grammar

$$\begin{array}{rcl} S & \rightarrow & 0A \\ A & \rightarrow & 10A \mid \epsilon \end{array}$$

Left-linear Grammar

A CFG $G = (N, T, P, S)$ is ***left-linear*** if all its productions are of the forms

$$\begin{array}{rcl} A & \rightarrow & Bw \\ A & \rightarrow & w \end{array}$$

where $A, B \in N$ and $w \in T^*$.

Example: The regular language $\mathbf{0}(\mathbf{10})^*$ is generated by the left-linear grammar

$$S \rightarrow S10 \mid 0$$

Leftmost and Rightmost Derivations

- A derivation is ***leftmost*** if productions are always applied to the leftmost nonterminal at each step in a derivation.
- A derivation is ***rightmost*** if productions are always applied to the rightmost nonterminal at each step in a derivation.

$$G: \begin{array}{lcl} S & \rightarrow & AB \mid BA \\ A & \rightarrow & a \\ B & \rightarrow & Ab \end{array}$$

Leftmost derivation:

$$S \underset{lm}{\Rightarrow} BA \underset{lm}{\Rightarrow} AbA \underset{lm}{\Rightarrow} abA \underset{lm}{\Rightarrow} aba$$

Ambiguous Grammars (1)

A CFG G is ***ambiguous*** if some word in $L(G)$ has ***more than one derivation tree***.

A derivation tree determines a unique leftmost and a unique rightmost derivation.

Thus, equivalently: A CFG G is ***ambiguous*** if some word in $L(G)$ has

- ***more than one leftmost derivation***, or
- ***more than one rightmost derivation***.

Ambiguous Grammars (2)

- A CFL for which every CFG is ambiguous is ***inherently ambiguous***.
  - The following language L is inherently ambiguous:

$$\begin{array}{lcl} L & = & \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \\ & & \cup\{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\} \end{array}$$

  - Reason: All but a finite number of strings of the form $a^n b^n c^n d^n$ must be generated in two different ways. (The proof is not easy!)

Ambiguous Grammars (3)

- Most CFLs are not inherently ambiguous; i.e., an ambiguous CFG G for a language L can often be ***transformed*** into an ***equivalent*** but unambiguous grammar G'.
- The ambiguity of a CFG is in general ***undecidable***.

Eliminating Ambiguity: Dangling-Else

Consider the following "dangling-else" grammar:

$$\begin{array}{lcl} \mathit{Stmt} & \rightarrow & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stmt} \\ & \mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \\ & \mid & \texttt{other} \end{array}$$

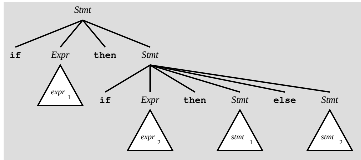
and the following program fragment:

if $expr_1$ then if $expr_2$ then $stmt_1$ else $stmt_2$

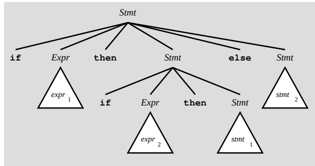
Two possible parse trees!
Hence the grammar is ambiguous!

Elim. Ambiguity: Dangling-Else (2)

Tree 1:



Tree 2:



Elim. Ambiguity: Dangling-Else (3)

Note that the distinction is important, as the two trees suggest ***different semantics***.

For example, suppose $expr_1$ evaluates to true, and $expr_2$ evaluates to false. Which, if any, of $stmt_1$ and $stmt_2$ gets executed?

Elim. Ambiguity: Dangling-Else (4)

Preferred interpretation:

"Match each `else` with the closest previous unmatched `then`"

That is, ***Tree 1*** is preferred.

Q: How can that be achieved?

A: Transform the grammar into an ***equivalent*** but ***unambiguous*** grammar.

Exercise: convince yourself that the following grammar indeed is equivalent!

Elim. Ambiguity: Dangling-Else (5)

Idea: a statement appearing between a `then` and an `else` must be a "matched" statement.

$$\begin{array}{lcl} \mathit{Stmt} & \rightarrow & \mathit{MatchedStmt} \\ & \mid & \mathit{UnmatchedStmt} \\ \mathit{MatchedStmt} & \rightarrow & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{MatchedStmt} \\ & & \texttt{else}\ \mathit{MatchedStmt} \\ & \mid & \texttt{other} \\ \mathit{UnmatchedStmt} & \rightarrow & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stmt} \\ & \mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{MatchedStmt} \\ & & \texttt{else}\ \mathit{UnmatchedStmt} \end{array}$$

Elim. Ambiguity: Dangling-Else (6)

Compare with the grammar for `if`-statements given in section 14.9 of the Java Language Specification, Third Edition:

`http://java.sun.com/docs/books/jls`

It uses the grammar structure of the previous slide to solve the dangling-else problem, even if the names of the non-terminals are somewhat different.

Eliminating Ambiguity: Associativity

It is standard practice to leave out unnecessary parentheses when writing down mathematical expressions:

$1 + 2 + 3$ instead of $(1 + 2) + 3$
$47 - 3 - 2$ instead of $(47 - 3) - 2$

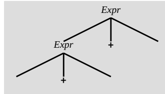
We would like to do the same when writing programs!

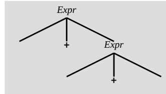
Elim. Ambiguity: Associativity (2)

The following grammar achieves that:

$$\begin{array}{lcl} Expr & \rightarrow & \texttt{integer} \\ & | & Expr \; \texttt{+} \; Expr \\ & | & Expr \; \texttt{-} \; Expr \\ & | & \texttt{(} \; Expr \; \texttt{)} \end{array}$$

But ambiguous! Parse trees for `1 + 2 + 3`:





(Slightly simplified: `1`, `2`, etc. considered terminals.)

Elim. Ambiguity: Associativity (3)

If we make the ***choice*** of letting the parse tree structure impart the bracketing structure, we see that the two parse trees correspond to

- `(1 + 2) + 3`
- `1 + (2 + 3)`

Similarly, `47 - 3 - 2` can be parsed in two ways:

- `(47 - 3) - 2`
- `47 - (3 - 2)`

Clearly the choice affects the of the code!

Elim. Ambiguity: Associativity (4)

- The choice might not seem important for $+$ since, mathematically, $+$ is ***associative***:

$$(1 + 2) + 3 = 1 + (2 + 3) = 6$$

But the ***computer implementation*** of $+$ might not be so well-behaved!

- Floating-point addition is ***not*** associative!
- Integer addition is not associative if e.g. overflow is trapped.

Elim. Ambiguity: Associativity (5)

- The choice clearly matters for $-$:

$$(47 - 3) - 2 \neq 47 - (3 - 2)$$

Elim. Ambiguity: Associativity (6)

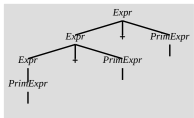
To disambiguate, we want to make both + and − ***left-associative***.

That can be achieved by making the relevant grammar productions ***left-recursive***:

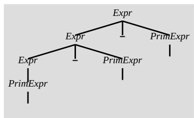
$$\begin{array}{lcl} Expr & \rightarrow & PrimExpr \\ & | & Expr \; \texttt{+} \; PrimExpr \\ & | & Expr \; \texttt{-} \; PrimExpr \\ PrimExpr & \rightarrow & \texttt{integer} \\ & | & \texttt{(} \; Expr \; \texttt{)} \end{array}$$

Elim. Ambiguity: Associativity (7)

Thus, `1 + 2 + 3` is parsed as `(1 + 2) + 3`:



And `47 - 3 - 2` is parsed as `(47 - 3) - 2`:



Elim. Ambiguity: Associativity (8)

Some operators are usually considered ***right-associative***.

Consider an arithmetic exponentiation operator `^`. We would like

`3 ^ 2 ^ 3`

to be parsed as

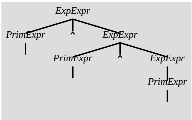
`3 ^ (2 ^ 3)`

so that the meaning is $3^{2^3} = 3^{(2^3)} = 6561$ rather than $(3^2)^3 = 729$.

Elim. Ambiguity: Associativity (9)

An operator can be made ***right-associative*** through ***right-recursive*** grammar productions:

$$\begin{array}{lcl} \mathit{ExpExpr} & \rightarrow & \mathit{PrimExpr} \\ & | & \mathit{PrimExpr}\ \texttt{\^{}}\ \mathit{ExpExpr} \\ \mathit{PrimExpr} & \rightarrow & \texttt{integer} \\ & | & \texttt{(}\ \mathit{Expr}\ \texttt{)} \end{array}$$



Eliminating Ambiguity: Precedence (1)

We would also like to be able to rely on standard rules for ***operator precedence*** to make it clear what is meant.

For example, it should be possible to write

`1 + 2 * 3`

instead of having to write out the fully parenthesized version

`1 + (2 * 3)`

Eliminating Ambiguity: Precedence (2)

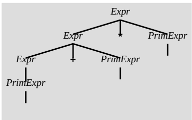
We chose to make $\star$ left-associative (standard). The following grammar accepts expressions like `1 + 2 * 3`:

$$\begin{array}{lcl} \mathit{Expr} & \rightarrow & \mathit{PrimExpr} \\ & | & \mathit{Expr}\ \texttt{+}\ \mathit{PrimExpr} \\ & | & \mathit{Expr}\ \texttt{*}\ \mathit{PrimExpr} \\ \mathit{PrimExpr} & \rightarrow & \texttt{integer} \\ & | & \texttt{(}\ \mathit{Expr}\ \texttt{)} \end{array}$$

Eliminating Ambiguity: Precedence (3)

However, the meaning is ***not*** what we want!

`1 + 2 * 3` gets parsed as `(1 + 2) * 3`:



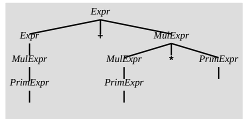
Eliminating Ambiguity: Precedence (4)

We rewrite the grammar so that expressions involving ***high-precedence*** operators only can occur as ***subexpressions*** of expressions involving low-precedence operators.

$$\begin{array}{lcl} \mathit{Expr} & \rightarrow & \mathit{MulExpr} \\ & | & \mathit{Expr}\ \texttt{+}\ \mathit{MulExpr} \\ \mathit{MulExpr} & \rightarrow & \mathit{PrimExpr} \\ & | & \mathit{MulExpr}\ \texttt{*}\ \mathit{PrimExpr} \\ \mathit{PrimExpr} & \rightarrow & \texttt{integer} \\ & | & \texttt{(}\ \mathit{Expr}\ \texttt{)} \end{array}$$

Eliminating Ambiguity: Precedence (5)

Now `1 + 2 * 3` gets parsed as `1 + (2 * 3)`:



Other ways of dealing with ambiguity

Transforming a grammar to eliminate ambiguity is not always desirable:

- Can be quite hard to do correctly.
- The transformed grammar might be less easy to understand than the original.

Parser generator tools often provide alternative disambiguation mechanisms:

- Meta-rules that favours the longest RHS among a group of conflicting productions.
- Explicit declaration of operator precedence.