

# Languages and Grammars

## COMP2600 — Formal Methods for Software Engineering

Jeremy Dawson

Australian National University  
Semester 2, 2011

## What's the Problem?

Examples of the difficulties faced by early compiler writers:

In *Fortran*, spaces are not significant (even as separators):

DO 5 I = 1.25 (assignment to variable D05I)

DO 5 I = 1,25 (count loop)

Not until the scanner reaches “.” or “,” can these statements be distinguished.

In *PL/1* there are no reserved words:

if then then then = else; else else = then;

Understanding language processing technology has influenced the *design* of programming languages syntax.

## Introduction

The development of high-level programming languages began in the 1950's with Fortran.

Fortran had *ad hoc* restrictions and non-orthogonal design — e.g. integer expressions were different in assignments, array subscripts, and DO-loop limits.

A major advance was Algol 60: “A vast improvement on all of its predecessors and most of its successors.”

The Algol team developed and used a formal definition of syntax — **Backus-Naur Form (BNF)**.

This started the interest in formal language theory among computer scientists, leading to standard and efficient translation techniques.

## Formal Languages

Terminology is similar to automata.

- The **alphabet** or **vocabulary** of a formal language is a set of **tokens** (or **letters**). It is usually denoted  $\Sigma$ .
- A **string** over  $\Sigma$  is a *sequence* of tokens, or the null-string  $\epsilon$ .  
For example, if  $\Sigma = \{a, b, c\}$  then *ababc* is a string over  $\Sigma$ .
- A **language** with alphabet  $\Sigma$  is some set of strings over  $\Sigma$ .
- The strings of a language are called the **sentences** of the languages.

### Notation:

- $\Sigma^*$  is the set of all strings over  $\Sigma$ .
- Therefore, every language with alphabet  $\Sigma$  is some **subset** of  $\Sigma^*$ .

## Specifying Languages

1. As an explicit set
2. As a set, by giving a predicate
3. Algebraically
4. **Grammars (e.g. regular or context-free)**
5. **Recognisers (automata)**

4, 5 are closely related:

- For each **grammar** we can derive an **automaton**, and *vice-versa*
- Each **type** of grammar corresponds to a particular **type** of automaton, and *vice-versa*

## Grammars

More definitions and terminology:

A **grammar** is a quadruple  $\langle V_t, V_n, S, P \rangle$  where:

- $V_t$  is a finite set of **terminal symbols** (the alphabet)
- $V_n$  is a finite set of **non-terminal symbols**  
( $V$  denotes  $V_t \cup V_n$ , and  $V_t \cap V_n = \emptyset$ )
- $S$  is a distinguished non-terminal symbol called the **goal** or **start** symbol
- $P$  is a set of **productions**, written  $\alpha \rightarrow \beta$  where  $\alpha \in V^*V_nV^*$  and  $\beta \in V^*$ .
  - $\alpha$  is a string of terminal and non-terminal symbols, *including at least one non-terminal.*
  - $\beta$  is a string of zero or more terminal and non-terminal symbols.

## Example:

$G = \langle \{a, b\}, \{S, A\}, S, \{S \rightarrow aAb, aA \rightarrow aaAb, A \rightarrow \epsilon\} \rangle$

- **Terminals:**  $\{a, b\}$
- **Non-terminals:**  $\{S, A\}$
- **Start symbol:**  $S$
- **Productions:**

$S \rightarrow aAb$

$aA \rightarrow aaAb$

$A \rightarrow \epsilon$

(Usually we only give the productions, using capital letters for non-terminals ( $S$  for the start symbol) and lower case for terminals — then the other details can be inferred.)

We can write  $S \rightarrow \alpha \mid \beta$  for the two productions  $S \rightarrow \alpha$  and  $S \rightarrow \beta$

## Derivations

Productions are **substitution rules**:

If there is a production  $\alpha \rightarrow \beta$  then we can rewrite any string  $\gamma\alpha\rho$  as  $\gamma\beta\rho$

we say  $\gamma\alpha\rho \Rightarrow \gamma\beta\rho$

**Derivations** are the transitive closure of these substitutions:

$\alpha \xrightarrow{+} \beta$  ( $\beta$  derived from  $\alpha$  using 1 or more steps)

$\alpha \xrightarrow{*} \beta$  ( $\beta$  derived from  $\alpha$  using 0 or more steps)

The **language generated by a grammar** is the set of strings of  $V_t$  that can be derived from the start symbol:

$\{\alpha \mid S \xrightarrow{+} \alpha \wedge \alpha \in V_t^*\}$

The **sentential forms** are  $\{\alpha \mid S \xrightarrow{*} \alpha \wedge \alpha \in V^*\}$

## Example ctd

$$G = \langle \{a, b\}, \{S, A\}, S, \{S \rightarrow aAb, aA \rightarrow aaAb, A \rightarrow \epsilon\} \rangle$$

A simple derivation:

$$S \Rightarrow aAb \Rightarrow aaAbb \Rightarrow aaaAbbb \Rightarrow aaabbb$$

The last string is a *sentence*. The others are *sentential forms*.

The language generated by  $G$  can also be described as

$$\{a^n b^n \mid n \in \mathbb{N}, n \geq 1\}$$

The same language is generated by

$$G' = \langle \{a, b\}, \{S\}, S, \{S \rightarrow aSb, S \rightarrow ab\} \rangle$$

(Grammars are not 1-to-1 with languages.)

## Context-Free Grammars

Productions are all of the form:  $A \rightarrow \omega$  where  $A \in V_n$  and  $\omega \in V^*$

**Independent of its context**  $A$  can be replaced by  $\omega$ .

(Hence the name “context free”)

In contrast, context-sensitive grammars may have productions like  $\alpha A \beta \rightarrow \alpha \omega \beta$

$A$  may be replaced by  $\omega$  but *only in the context*  $\alpha\_ \beta$  ;

Theorem: we need only productions of this form for a type 2 language, (that is, a language that can be generated by a type 2 grammar), hence the name “context sensitive”

BNF, EBNF, railroad diagrams etc. are all context-free grammars.

## The Chomsky Hierarchy

Noam Chomsky classified grammars on the basis of the *form of their productions*:

**Regular:** (type 3) As for type 2, and the rhs of each derivation contains at most one non-terminal, and the non-terminal is always first or always last (exact definitions vary; see slide 12)

**Context-free:** (type 2) the lhs of each production must be a *single non-terminal*.

**Context-sensitive:** (type 1) the length of the lhs of each production must not exceed the length of the rhs.

**Unrestricted:** (type 0) no constraints.

Computer applications are based on **regular** and **context-free** grammars.

## Regular Grammars

Productions are *all* of the form (for a *left-linear* grammar):

$$A \rightarrow a \quad \text{or} \quad A \rightarrow Ba \quad \text{or} \quad A \rightarrow \epsilon$$

or *all* of the form: (for a *right-linear* grammar):

$$A \rightarrow a \quad \text{or} \quad A \rightarrow aB \quad \text{or} \quad A \rightarrow \epsilon$$

The essential feature of regular grammars is that they generate sentences left to right (or right to left), one symbol at a time.

Regular grammars are inconvenient.

Regular expressions are equivalent — they can define exactly the same set of languages and are much more convenient. (*Trust me...*)

These languages are called **regular languages**

## Regular Languages

**Definition:** A language is a **regular language** if it is accepted by some DFA

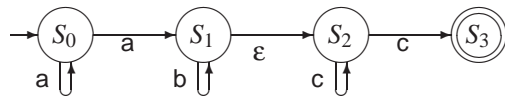
**Theorem:** The following are equivalent:

- $L$  is a regular language
- $L$  is the language accepted by some NFA
- $L$  is the language specified by a regular expression
- $L$  is the language generated by a right-linear grammar
- $L$  is the language generated by a left-linear grammar

We discussed the first three of these in connection with FSAs.

We can easily show that regular grammars correspond to NFAs.

## Regular Grammars and NFAs



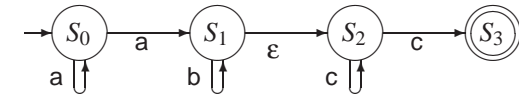
### A right-linear grammar accepting the same language

$$\begin{array}{ll}
 S \rightarrow S_0 & S_1 \rightarrow \epsilon S_2 (= S_2) \\
 S_0 \rightarrow a S_0 & S_2 \rightarrow c S_2 \\
 S_0 \rightarrow a S_1 & S_2 \rightarrow c S_3 \\
 S_1 \rightarrow b S_1 & S_3 \rightarrow \epsilon
 \end{array}$$

Idea : a non-terminal  $A$  expands to the set of strings which the NFA, *starting in state A*, "accepts"

We turn this into a grammar not containing productions of the form  $A \rightarrow B$  (see tutorial exercises on how to do this)

## Regular Grammars and NFAs — again



### A left-linear grammar accepting the same language

$$\begin{array}{ll}
 S_0 \rightarrow \epsilon & S_2 \rightarrow S_1 \epsilon (= S_1) \\
 S_0 \rightarrow S_0 a & S_2 \rightarrow S_2 c \\
 S_1 \rightarrow S_0 a & S_3 \rightarrow S_2 c \\
 S_1 \rightarrow S_1 b & S \rightarrow S_3
 \end{array}$$

Idea : a non-terminal  $A$  expands to the set of strings which the NFA, *finishing in state A*, "accepts"

We turn this into a grammar not containing productions of the form  $A \rightarrow B$

## Classification of Languages

A language is *type n* if it **can** be generated by a type  $n$  grammar.

Going up the hierarchy of grammars, there are more restrictions placed on the form of productions that are permitted.

For example, if there is a type 2 grammar for some language then there are also type 1 and type 0 grammars for that language.

To show that a *language* is type 2 we must provide a type 2 grammar for it.

To show that a *language* is not type 3 we must show that there *cannot* be a type 3 grammar for it.

Exception to the above: a *type 1 language* is also allowed to contain  $\epsilon$

## Example — language $\{a^n b^n \mid n \in \mathbb{N}, n \geq 1\}$

We already saw two grammars:

- Type 1 (well, nearly):

$$\begin{array}{l} S \rightarrow aAb \\ aA \rightarrow aaAb \\ A \rightarrow \varepsilon \end{array} \quad \text{type 1 equivalent:} \quad \begin{array}{l} S \rightarrow aAb \mid ab \\ aA \rightarrow aaAb \mid aab \end{array}$$

- Context-free (type 2):

$$\begin{array}{l} S \rightarrow ab \\ S \rightarrow aSb \end{array}$$

Last week we proved that there is no FSA (and therefore no regular grammar), so the *language* must be context-free.

## Exercise:

Give a regular expression for the language of integer literals.

## Exercise:

Define a DFA or NFA to recognise the language of integer literals.

## Exercise:

The language of Java identifiers was described in an earlier lecture with a railroad diagram. Give a corresponding context-free grammar. In fact the language is regular. Give a regular expression and/or a regular grammar.

## Example — Integer Literals

$$\begin{array}{l} I \rightarrow U \mid +U \mid -U \\ U \rightarrow D \mid DU \\ D \rightarrow 0 \mid 1 \mid \dots 9 \end{array}$$

The non-terminals are  $I$  (*integer*) (and start symbol),  $U$  (*unsigned integer*) and  $D$  (*digit*).

There is also a *regular* grammar:

$$\begin{array}{l} I \rightarrow +U \mid -U \mid 0U \mid 1U \mid \dots 9U \mid 0 \mid 1 \mid \dots 9 \\ U \rightarrow 0U \mid 1U \mid \dots 9U \mid 0 \mid 1 \mid \dots 9 \end{array}$$

(In fact we can derive this grammar from the context-free one by substitution.)

So the *language* of integer literals is *regular*.

## Example — Arithmetic Expressions

$$\begin{array}{l} E \rightarrow T \mid E + T \mid E - T \\ T \rightarrow F \mid T * F \mid T / F \\ F \rightarrow \mathbf{id} \mid (E) \end{array}$$

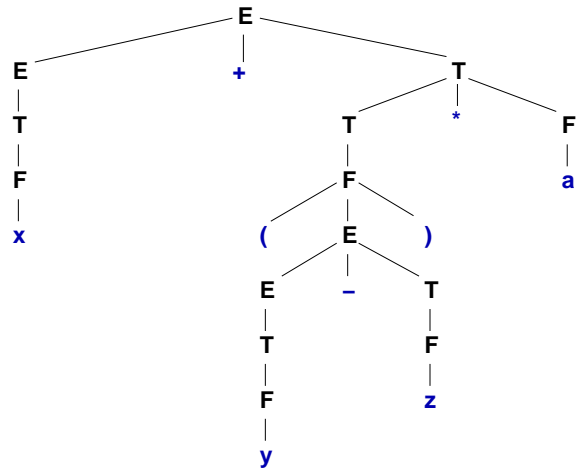
$E$  is *expression*,  $T$  is *term* and  $F$  is *factor*. Assume identifiers have been defined elsewhere and treat **id** as a terminal symbol.

The grammar is context-free and the language is context-free. *(In fact, without the need to match an arbitrary number of parentheses, it would have been a regular language.)*

This grammar encodes the priority and associativity of the arithmetic operators and parentheses — some *semantics* as well as the *syntax*.

## Example Parse Tree

The parse tree for the expression  $x + (y - z) * a$  is as follows:



## Ambiguity

The grammar for arithmetic expressions on the previous slide look fairly complicated. Is there an easier way?

$$E \rightarrow \mathbf{id} \mid E + E \mid E - E \mid E * E \mid E / E \mid (E)$$

Yes, but this grammar is *ambiguous*.

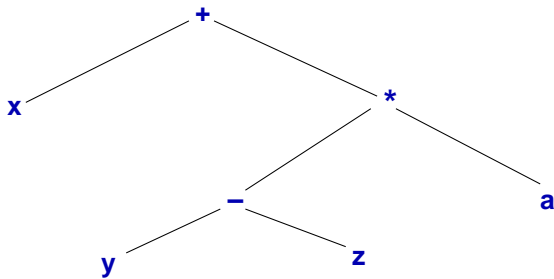
### Definition:

A grammar is *ambiguous* if some sentence has more than one parse tree.

### So what?

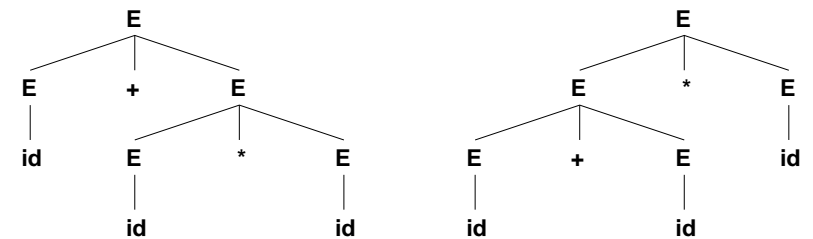
It must be impossible to produce deterministic parsing algorithms for ambiguous grammars.

If we remove the “intermediate” nodes in the tree, it becomes clearer how the order of evaluation of  $x + (y - z) * a$  is represented in the tree structure:



## Example:

Two parses of the sentence  $\mathbf{id + id * id}$ :



For the other grammar there is only *one parse tree* for any sentence.