

# G54FOP: Lecture 1

## Basic Formal Language Notions and Abstract Syntax

Henrik Nilsson

University of Nottingham, UK

G54FOP: Lecture 1 – p.1/32

## About These Slides

We give a brief recap on some central notions from the theory of formal languages, covered in a typical undergraduate course on that topic such as G52MAL, before moving on to abstract syntax.

To recap, consult the G52MAL lecture notes:

<http://www.cs.nott.ac.uk/~nhn/G52MAL>

or a book on the topic, such as

John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation, 2nd edition*, Addison Wesley, 2001.

G54FOP: Lecture 1 – p.2/32

## Syntax and Semantics (1)

The notions of **Syntax** and **Semantics** are central to any discourse on languages. Focusing on **programming languages**:

- **Syntax**: the **form** of programs
  - **Concrete Syntax** (or **Surface Syntax**): (typically) the exact character sequences that are syntactically valid programs.
  - **Abstract Syntax**: the essential structure of syntactically valid programs.

G54FOP: Lecture 1 – p.3/32

## Syntax and Semantics (2)

- **Semantics**: the **meaning** of programs
  - **Static Semantics**: the static, at **compile-time**, meaning of programs and program fragments. E.g. types.
  - **Dynamic Semantics**: what programs and program fragments mean (or do) when executed, at **run-time**.

G54FOP: Lecture 1 – p.4/32

## Syntax and Semantics (3)

- Methods for defining the syntax and semantics of programming languages are thus the very foundation for systematic study of programming languages.
- A large part of this module will thus be concerned with various aspects of syntax and semantics for programming languages.
- We will start by looking at syntax, recapitulating some notions from the theory of formal languages related to **concrete syntax**, and then move on to **abstract syntax**.

G54FOP: Lecture 1 – p.5/32

## Formal Languages

In the context of **formal** languages, the terms **language** and **word** are used in a strict technical sense:

- A **language** is a (possibly infinite) set of words.
- A **word** is a **finite** sequence (or string) of symbols.

$\epsilon$  denotes the **empty word**, the sequence of zero symbols.

The term **string** is often used interchangeably with the term **word**.

G54FOP: Lecture 1 – p.6/32

## Symbols and Alphabets

What is a symbol, then?

Anything, but it has to come from an **alphabet**  $\Sigma$  which is a **finite** set.

A common (and important) instance is  $\Sigma = \{0, 1\}$ .

$\epsilon$ , the empty word, is **never** a symbol of an alphabet.

G54FOP: Lecture 1 – p.7/32

## Languages: Examples

alphabet  $\Sigma = \{a, b\}$   
words

Note the distinction between  $\epsilon$ ,  $\emptyset$ , and  $\{\epsilon\}$ !

G54FOP: Lecture 1 – p.8/32

## Exercises

- Is the set of natural numbers,  $\mathbb{N}$ , a possible alphabet? Why/why not?
- What about the set of all natural numbers smaller than some given number  $n$ ?
- Homework:
  - Suggest an alphabet of a handful of **drink ingredients**.
  - List some words over your alphabet.
  - Does all possible words describe "interesting" drinks?
  - Define a language of interesting drinks.

G54FOP: Lecture 1 – p.9/32

## All Words over an Alphabet (1)

Given an alphabet  $\Sigma$  we define the set  $\Sigma^*$  as set of words (or sequences) over  $\Sigma$ :

- The empty word  $\epsilon \in \Sigma^*$ .
- given a symbol  $x \in \Sigma$  and a word  $w \in \Sigma^*$ ,  $xw \in \Sigma^*$ .
- These are all elements in  $\Sigma^*$ .

This is called an **inductive definition**.

Inductive definitions and reasoning by induction over inductively defined structures will be recurring themes in this module.

054FOP: Lecture 1 – p.10/32

## Examples of Languages

Some examples of languages:

- The set  $\{0010, 00000000, \epsilon\}$  is a language over  $\Sigma = \{0, 1\}$ .  
This is an example of a **finite** language.
- The set of words with odd length over  $\Sigma = \{1\}$ . (Finite or infinite?)
- The set of words that contain the same number of 0s and 1s is a language over  $\Sigma = \{0, 1\}$ . (Finite or infinite?)
- The set of correct Java programs. This is a language over the set of UNICODE characters.

054FOP: Lecture 1 – p.10/32

## 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  $A \rightarrow \alpha$ , where  $A \in N$  and  $\alpha \in (N \cup T)^*$

054FOP: Lecture 1 – p.10/32

## All Words over an Alphabet (2)

Example: Given  $\Sigma = \{0, 1\}$ , some elements of  $\Sigma^*$  are

- $\epsilon$  (the empty word)
- 0, 1
- 00, 10, 01, 11
- 000, 100, 010, 110, 001, 101, 011, 111
- ...

We are just applying the inductive definition.

Note: although there are infinitely many words in  $\Sigma^*$ , each word has a **finite** length!

054FOP: Lecture 1 – p.11/32

## 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**.

054FOP: Lecture 1 – p.11/32

## Context-Free Grammar: Example

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

where  $P$  consists of the productions

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow aA \\ A &\rightarrow bS \end{aligned}$$

054FOP: Lecture 1 – p.11/32

## Languages Revisited

The notion of a language  $L$  of a set of words over an alphabet  $\Sigma$  can now be made precise:

- $L \subseteq \Sigma^*$ , or equivalently
- $L \in \mathcal{P}(\Sigma^*)$ .

054FOP: Lecture 1 – p.12/32

## 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.

054FOP: Lecture 1 – p.12/32

## Context-Free Grammars: Notation

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

$$S \rightarrow \epsilon \mid aA$$

This is (roughly) what is known as **Backus-Naur Form**.

- Another common way of writing productions is

$$A ::= \alpha$$

054FOP: Lecture 1 – p.12/32

## The Directly Derives Relation (1)

To formally define the language generated by

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

we first define a binary relation  $\Rightarrow_G$  on strings over

$N \cup T$ , read “**directly derives** in grammar  $G$ ”, being the least relation such that

$$\alpha A \gamma \Rightarrow_G \alpha \beta \gamma$$

whenever  $A \rightarrow \beta$  is a production in  $G$ .

**Note:** a production can be applied regardless of context, hence **context-free**.

054FOP: Lecture 1 – p.19/32

## The Derives Relation (2)

Again, we use  $\xRightarrow{*}_G$  instead of  $\Rightarrow_G$  when  $G$  is obvious.

Example: Given the grammar

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

we have

$$\begin{aligned} S &\xRightarrow{*} \epsilon & S &\xRightarrow{*} abS \\ S &\xRightarrow{*} aA & S &\xRightarrow{*} ababS \\ aA &\xRightarrow{*} abS & S &\xRightarrow{*} abab \end{aligned}$$

054FOP: Lecture 1 – p.22/32

## Example: MiniTriangle CFG (1)

Concrete syntax for MiniTriangle:

```

Program    → Command
Commands  → Command
           | Command ; Commands
Command   → VarExpression := Expression
           | VarExpression ( Expressions )
           | if Expression then Command else Command
           | while Expression do Command
           | let Declarations in Command
           | begin Commands end
    
```

054FOP: Lecture 1 – p.25/32

## The Directly Derives Relation (2)

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

Example: Given the grammar

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

we have

$$\begin{aligned} S &\Rightarrow \epsilon & aA &\Rightarrow abS \\ S &\Rightarrow aA & SaAaa &\Rightarrow SabSaa \end{aligned}$$

054FOP: Lecture 1 – p.20/32

## 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:

$$L(G) = \{w \mid w \in T^* \wedge S \xRightarrow{*}_G w\}$$

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  $S \xRightarrow{*} \alpha$ .

054FOP: Lecture 1 – p.23/32

## Example: MiniTriangle CFG (2)

```

Expressions → Expression
              | Expression , Expressions
Expression  → PrimaryExpression
              | Expression Operator PrimaryExpression
PrimaryExpression → IntegerLiteral
                  | VarExpression
                  | Operator PrimaryExpression
                  | ( Expression )
VarExpression → Identifier
    
```

054FOP: Lecture 1 – p.26/32

## The Derives Relation (1)

The relation  $\xRightarrow{*}_G$ , read “**derives** in grammar  $G$ ”, is the reflexive, transitive closure of  $\Rightarrow_G$ .

That is,  $\xRightarrow{*}_G$  is the least relation on strings over  $N \cup T$  such that:

- $\alpha \xRightarrow{*}_G \beta$  if  $\alpha \Rightarrow_G \beta$
- $\alpha \xRightarrow{*}_G \alpha$  (reflexive)
- $\alpha \xRightarrow{*}_G \beta$  if  $\alpha \xRightarrow{*}_G \gamma \wedge \gamma \xRightarrow{*}_G \beta$  (transitive)

054FOP: Lecture 1 – p.21/32

## Language Generation: Example

Given the grammar

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

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

we have

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

054FOP: Lecture 1 – p.24/32

## Example: MiniTriangle CFG (3)

```

Declarations → Declaration
              | Declaration ; Declarations
Declaration  → const Identifier : TypeDenoter = Expression
              | var Identifier : TypeDenoter
              | var Identifier : TypeDenoter := Expression
TypeDenoter  → Identifier
    
```

054FOP: Lecture 1 – p.27/32

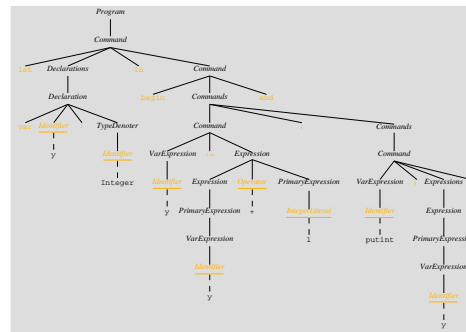
## A MiniTriangle Program

```

let
  var y: Integer
in
  begin
    y := y + 1;
    putint(y)
  end

```

## Parse Tree for the Program



## MiniTriangle Abstract Syntax (1)

The details of the concrete syntax often obscure the essence of the structure of a program. In contrast, **abstract syntax** describe this directly:

<i>Program</i>	→	<i>Command</i>	<i>Program</i>
<i>Command</i>	→	<i>Expression := Expression</i>	<i>CmdAssign</i>
		<i>Expression ( Expression* )</i>	<i>CmdCall</i>
		<i>Command*</i>	<i>CmdSeq</i>
		<b>if</b> <i>Expression</i> <b>then</b> <i>Command</i>	<i>CmdIf</i>
		<b>else</b> <i>Command</i>	
		<b>while</b> <i>Expression</i> <b>do</b> <i>Command</i>	<i>CmdWhile</i>
		<b>let</b> <i>Declaration*</i> <b>in</b> <i>Command</i>	<i>CmdLet</i>

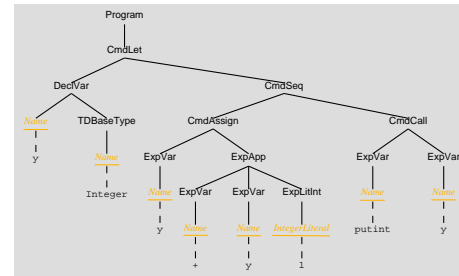
## MiniTriangle Abstract Syntax (2)

<i>Expression</i>	→	<i>IntegerLiteral</i>	<i>ExpLitInt</i>
		<i>Name</i>	<i>ExpVar</i>
		<i>Expression ( Expression* )</i>	<i>ExpApp</i>
<i>Declaration</i>	→	<b>const</b> <i>Name</i> : <i>TypeDenoter</i>	<i>DeclConst</i>
		<b>=</b> <i>Expression</i>	
		<b>var</b> <i>Name</i> : <i>TypeDenoter</i>	<i>DeclVar</i>
		<b>( := Expression   ε )</b>	
<i>TypeDenoter</i>	→	<i>Name</i>	<i>TDBaseType</i>

Note: Keywords and other fixed-spelling terminals serve only to make the connection with the concrete syntax clear.

*Identifier* ⊆ *Name*, *Operator* ⊆ *Name*

## Abstract Syntax Tree for the Program



Key Point: The abstract syntax specifies **trees**, not strings.