G54FOP: Lecture 1

Basic Formal Language Notions
and
Abstract Syntax

Henrik Nilsson

University of Nottingham, UK

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.

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.

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***.

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***.

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.

ϵ denotes the ***empty word***, the sequence of zero symbols.

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

Symbols and Alphabets

What is a symbol, then?

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

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

ϵ, the empty word, is ***never*** a symbol of an alphabet.

Languages: Examples

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

words

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

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.

All Words over an Alphabet (1)

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

- 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 Σ^*.

This is called an ***inductive definition***.

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

All Words over an Alphabet (2)

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

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

We are just applying the inductive definition.

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

Languages Revisited

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

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

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.

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 $A \rightarrow \alpha$, 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

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

Context-Free Grammars: Notation

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

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

  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 $A \rightarrow \beta$ 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

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

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

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

we have

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

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:

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

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

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

Example: MiniTriangle CFG (1)

Concrete syntax for MiniTriangle:

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Command} \\
\mathit{Commands} & \rightarrow & \mathit{Command} \\
 & \mid & \mathit{Command}\ \mathtt{;}\ \mathit{Commands} \\
\mathit{Command} & \rightarrow & \mathit{VarExpression}\ \mathtt{:=}\ \mathit{Expression} \\
 & \mid & \mathit{VarExpression}\ \mathtt{(}\ \mathit{Expressions}\ \mathtt{)} \\
 & \mid & \mathtt{if}\ \mathit{Expression}\ \mathtt{then}\ \mathit{Command}\ \mathtt{else}\ \mathit{Command} \\
 & \mid & \mathtt{while}\ \mathit{Expression}\ \mathtt{do}\ \mathit{Command} \\
 & \mid & \mathtt{let}\ \mathit{Declarations}\ \mathtt{in}\ \mathit{Command} \\
 & \mid & \mathtt{begin}\ \mathit{Commands}\ \mathtt{end}
\end{array}
$$

Example: MiniTriangle CFG (2)

$$
\begin{array}{lcl}
\mathit{Expressions} & \rightarrow & \mathit{Expression} \\
 & \mid & \mathit{Expression}\ \mathtt{,}\ \mathit{Expressions} \\
\mathit{Expression} & \rightarrow & \mathit{PrimaryExpression} \\
 & \mid & \mathit{Expression}\ \underline{\mathit{Operator}}\ \mathit{PrimaryExpression} \\
\mathit{PrimaryExpression} & \rightarrow & \underline{\mathit{IntegerLiteral}} \\
 & \mid & \mathit{VarExpression} \\
 & \mid & \underline{\mathit{Operator}}\ \mathit{PrimaryExpression} \\
 & \mid & \mathtt{(}\ \mathit{Expression}\ \mathtt{)} \\
\mathit{VarExpression} & \rightarrow & \underline{\mathit{Identifier}}
\end{array}
$$

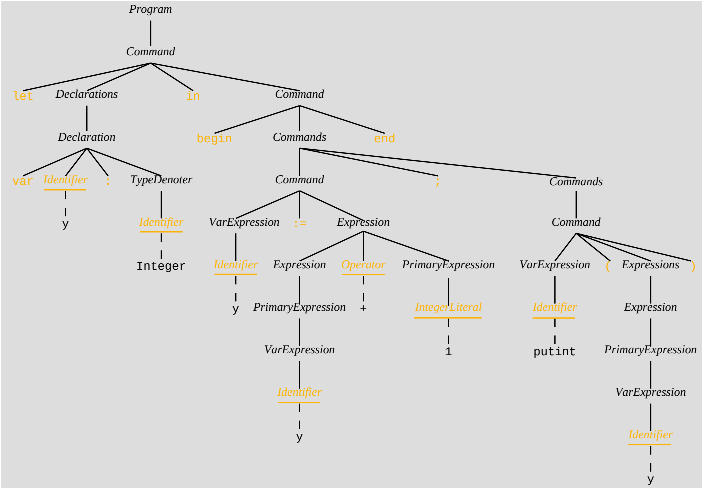
Example: MiniTriangle CFG (3)

$$
\begin{array}{lcl}
\mathit{Declarations} & \rightarrow & \mathit{Declaration} \\
 & \mid & \mathit{Declaration}\ \mathtt{;}\ \mathit{Declarations} \\
\mathit{Declaration} & \rightarrow & \mathtt{const}\ \underline{\mathit{Identifier}}\ \mathtt{:}\ \mathit{TypeDenoter}\ \mathtt{=}\ \mathit{Expression} \\
 & \mid & \mathtt{var}\ \underline{\mathit{Identifier}}\ \mathtt{:}\ \mathit{TypeDenoter} \\
 & \mid & \mathtt{var}\ \underline{\mathit{Identifier}}\ \mathtt{:}\ \mathit{TypeDenoter}\ \mathtt{:=}\ \mathit{Expression} \\
\mathit{TypeDenoter} & \rightarrow & \underline{\mathit{Identifier}}
\end{array}
$$

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:

Program	→	*Command*	Program
Command	→	*Expression* **:=** *Expression*	CmdAssign
	|	*Expression* **(** *Expression** **)**	CmdCall
	|	*Command**	CmdSeq
	|	**if** *Expression* **then** *Command*	CmdIf
		else *Command*	
	|	**while** *Expression* **do** *Command*	CmdWhile
	|	**let** *Declaration** **in** *Command*	CmdLet

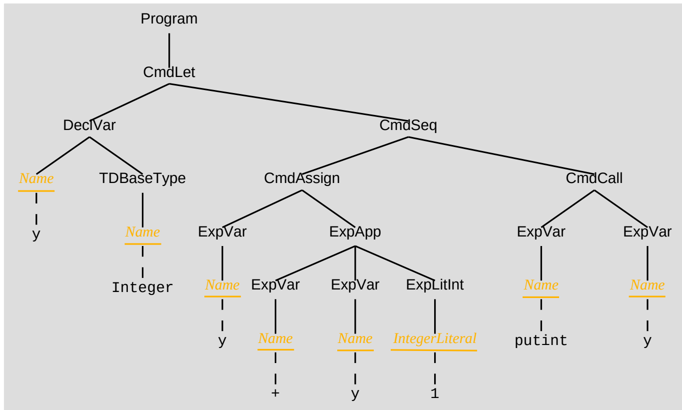
MiniTriangle Abstract Syntax (2)

Expression	→	*IntegerLiteral*	ExpLitInt
	|	*Name*	ExpVar
	|	*Expression* **(** *Expression** **)**	ExpApp
Declaration	→	**const** *Name* **:** *TypeDenoter*	DeclConst
		= *Expression*	
	|	**var** *Name* **:** *TypeDenoter*	DeclVar
		(**:=** *Expression* | ϵ)	
TypeDenoter	→	*Name*	TDBaseType

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

Identifier $\subseteq$ *Name*, *Operator* $\subseteq$ *Name*

Abstract Syntax Tree for the Program



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