

COMP3012/G53CMP: Lecture 2

Defining Programming Languages

Henrik Nilsson

University of Nottingham, UK

COMP3012/G53CMP: Lecture 2 – p.1/37

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**): What programs “look like”.
 - Usually **strings** of characters or symbols.
 - Some languages have graphical syntax.
 - **Abstract Syntax**: **trees** representing the essential structure of syntactically valid programs.

COMP3012/G53CMP: Lecture 2 – p.3/37

This Lecture

- Programming language definition basics.
- Backus-Naur Form (BNF) and Extended BNF (EBNF)
- Concrete Syntax
 - Lexical syntax for MiniTriangle
 - Context-free syntax for MiniTriangle
- Abstract Syntax
 - Abstract syntax for MiniTriangle
- Representing Abstract Syntax Trees (ASTs)

COMP3012/G53CMP: Lecture 2 – p.2/37

Syntax and Semantics (2)

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

COMP3012/G53CMP: Lecture 2 – p.4/37

Defining Programming Languages (1)

- In order to develop a compiler (or other language processor):
 - the **Source Language** must be defined
 - syntax
 - semantics
 - the **Target Language** must be defined
 - syntax
 - semantics
- Language definitions (aka **specifications**) can be **formal** or **informal**. Usually they are somewhere in between.

COMP3012/G53CMP: Lecture 2 – p.5/37

Object Language and Meta Language

In any language definition, informal or formal, a careful distinction must be made between

- the **Object Language**: the language being defined
- the **Meta Language**: the language of the definition itself.

Moreover, the semantics of the meta language must be well understood!

COMP3012/G53CMP: Lecture 2 – p.7/37

Defining Programming Languages (2)

Why is it important that the source and target languages are precisely defined?

- The source language syntax must be known to design the scanner and parser properly.
- The target language syntax must be known to generate syntactically correct target code.
- The semantics of both the source and target language must be known to ensure that the translation **preserves the meaning** of source programs; i.e. **compiler correctness**.

COMP3012/G53CMP: Lecture 2 – p.6/37

Informal Specifications

- In an informal specification, the meta language is a natural language such as English.
- Most programming languages are defined more or less informally.
- “Informal” does not mean “lack of rigour”: it is possible to be precise also in a natural language.
- An example of a well-written, predominantly informal language specification is that of Java:
<http://java.sun.com/docs/books/jls>
(See e.g. Third Edition, Section 14.9.)

COMP3012/G53CMP: Lecture 2 – p.8/37

Formal Specifications

A **Formal Specification** is mathematically precise. Usually, a **Formal Metalanguage** is used; e.g.:

- **EBNF** for specifying context-free syntax. (Int'l standard: ISO/IEC 14977:1996(E))
- **inference rules and logic** for specifying static and/or dynamic semantics
- **denotational semantics** for specifying dynamic semantics.

COMP3012/G53CMP: Lecture 2 – p.9/37

CFG Notation (1)

We will give CFGs by stating the productions in one of two styles:

- **Mathematical style** (or “G52LAC style”):
 - Used for: small, abstract examples; at **meta level** when talking **about** grammars.
 - Simple naming conventions used to distinguish terminals and non-terminals:
 - nonterminals: uppercase letters, like A, B, S
 - terminals: lowercase letters or digits, like $a, b, 3$
 - Start symbol usually called S .

COMP3012/G53CMP: Lecture 2 – p.11/37

Context-Free Grammars

A **Context-Free Grammar** (CFG) formally describes a **Context-Free Languages** (CFL):

- The CFLs capture common programming language ideas such as
 - nested structure
 - balanced parentheses
 - matching keywords like `begin` and `end`.
- Most “reasonable” CFLs can be recognised by a simple machine: a **deterministic pushdown automaton**.

COMP3012/G53CMP: Lecture 2 – p.10/37

CFG Notation (2)

- **Programming Language Specification style**:
 - Used for larger, more realistic examples.
 - Typographical conventions used to distinguish terminals and non-terminals:
 - nonterminals are written like *this*
 - terminals are written like **this**
 - terminals with **variable spelling** and special symbols are written like *this*
 - The start symbol is often implied by the context.

COMP3012/G53CMP: Lecture 2 – p.12/37

CFG Notation (3)

For example:

$$\text{AssignStmt} \rightarrow \text{Identifier} := \text{Expr}$$

Here,

- *AssignStmt* and *Expr* are nonterminals
- **:=** is a terminal
- *Identifier* is also a terminal, but its possible spellings are defined elsewhere (usually by a **lexical grammar**).

COMP3012/G53CMP: Lecture 2 – p.13/37

EBNF: Example

The following EBNF grammar

$$\text{Block} \rightarrow \text{begin } (\text{Decl} \mid \text{Stmt})^* \text{end}$$

(where *Decl* and *Stmt* are defined elsewhere) is equivalent to the following BNF grammar:

$$\text{Block} \rightarrow \text{begin } \text{BlockRec} \text{end}$$
$$\text{BlockRec} \rightarrow \epsilon \mid \text{BlockRec } \text{BlockAlts}$$
$$\text{BlockAlts} \rightarrow \text{Decl} \mid \text{Stmt}$$

Thus we see that EBNF can be quite a bit more concise and readable than plain BNF.

COMP3012/G53CMP: Lecture 2 – p.15/37

BNF and Extended BNF

The CFGs we have seen so far have (essentially) been expressed in **Backus-Naur Form** (BNF).

Extended BNF (EBNF) is a more **convenient** way of describing CFGs than is BNF.

- Additional EBNF constructs:
 - parentheses for grouping
 - | for alternatives **within** parentheses
 - * for iteration (W&B's notation).
- EBNF is **no more powerful** than BNF: any EBNF grammar can be transformed into BNF.

COMP3012/G53CMP: Lecture 2 – p.14/37

EBNF: ISO Notation

Watt & Brown use their own EBNF variant.

The more common variant is the ISO (International Organization for Standardization) version (ISO/IEC 14977:1996):

ISO	W&B
{ A }	A*
[A]	(A ε)

COMP3012/G53CMP: Lecture 2 – p.16/37

MiniTriangle

The source language in the coursework is called **MiniTriangle** (derived from Watt & Brown).

Example:

```
let
  var y: Integer := 0
in
  begin
    y := y + 1;
    putint(y)
  end
```

COMP3012/G53CMP: Lecture 2 – p.17/37

MiniTriangle Lexical Syntax (1)

Program → $(Token \mid Separator)^*$

Token → $Keyword \mid Identifier \mid IntegerLiteral \mid Operator$
| $, \mid ; \mid : \mid := \mid = \mid (\mid) \mid \underline{eol}$

Keyword → **begin** | **const** | **do** | **else** | **end** | **if** | **in**
| **let** | **then** | **var** | **while**

Identifier → $Letter \mid Identifier \ Letter \mid Identifier \ Digit$
except *Keyword*

IntegerLiteral → $Digit \mid IntegerLiteral \ Digit$

Operator → $+ \mid - \mid * \mid / \mid < \mid <= \mid == \mid != \mid >= \mid > \mid \&\& \mid || \mid !$

Separator → $Comment \mid \underline{space} \mid \underline{eol}$

Comment → $//$ (any character except eol)* eol

COMP3012/G53CMP: Lecture 2 – p.19/37

Concrete Syntax

The **Concrete Syntax**, or surface syntax, of a language is usually defined at two levels:

- The **Lexical syntax**: the syntax of
 - **language symbols** or **tokens**
 - **white space**
 - **comments**
- The **Context-Free syntax**.

COMP3012/G53CMP: Lecture 2 – p.18/37

MiniTriangle Lexical Syntax (2)

Notes:

- Essentially a (left-)linear grammar; i.e, the lexical syntax specifies a **regular** language.
- Not completely formal (e.g. the use of “except” for excluding keywords from identifiers).
- Note! Each individual character of a terminal is actually a terminal symbol! I.e., really:
Keyword → **b e g i n** | **c o n s t** | ...
- Special characters are written like *this*.
Note! They are single terminal symbols!

COMP3012/G53CMP: Lecture 2 – p.20/37

MiniTriangle: Tokens

Some valid MiniTriangle tokens:

- `const3` (Identifier)
- `const` (Keyword)
- `42` (Integer-Literal)
- `+` (Operator)

Q: Is `const3` really a single token?
The grammar is **ambiguous**!

A: An implicit “**maximal munch rule**” used to disambiguate!

COMP3012/G53CMP: Lecture 2 – p.21/37

MiniTriangle Context-Free Syntax (1)

(Small version: other (extended) versions later.)

```
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
```

COMP3012/G53CMP: Lecture 2 – p.22/37

MiniTriangle Context-Free Syntax (2)

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

COMP3012/G53CMP: Lecture 2 – p.23/37

MiniTriangle Context-Free Syntax (3)

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

COMP3012/G53CMP: Lecture 2 – p.24/37

Another MiniTriangle Program

The following is a **syntactically** valid MiniTriangle program (slightly changed from earlier to save some space):

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

COMP3012/G53CMP: Lecture 2 – p.25/37

Why a Lexical Grammar? (1)

Together, the lexical grammar and the context-free grammar specify the **concrete syntax**.

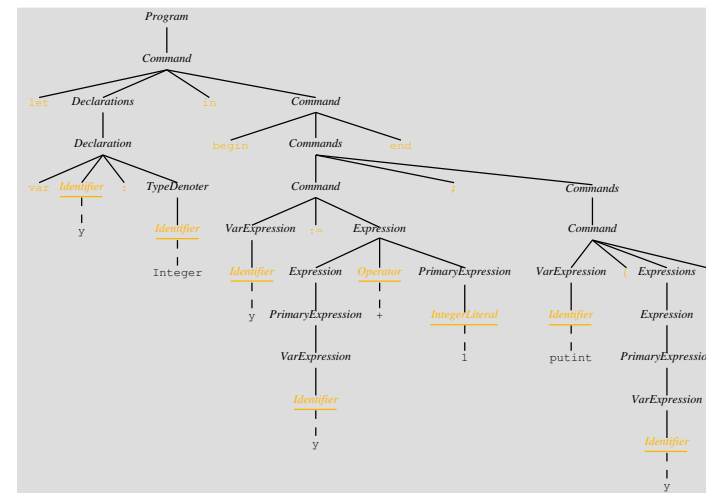
In our case, both grammars are expressed in (E)BNF and looks similar.

So ...

- Why not join them?
- Why not do away with scanning, and just do parsing?

COMP3012/G53CMP: Lecture 2 – p.27/37

Parse Tree for the Program



COMP3012/G53CMP: Lecture 2 – p.28/37

Why a Lexical Grammar? (2)

Answer:

- **Simplicity**: dealing with white space and comments in the context free grammar becomes extremely complicated. (Try it!)
- **Efficiency**:
 - Working on classified groups of characters (tokens) facilitates parsing: may be possible to use a simpler parsing algorithm.
 - Grouping and classifying characters by as simple means as possible increases efficiency.

COMP3012/G53CMP: Lecture 2 – p.28/37

MiniTriangle Abstract Syntax (1)

This grammar specifies the **phrase structure** of MiniTriangle. In addition, it gives node labels to be used when drawing Abstract Syntax Trees.

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

COMP3012/G53CMP: Lecture 2 – p.29/37

MiniTriangle Abstract Syntax (2)

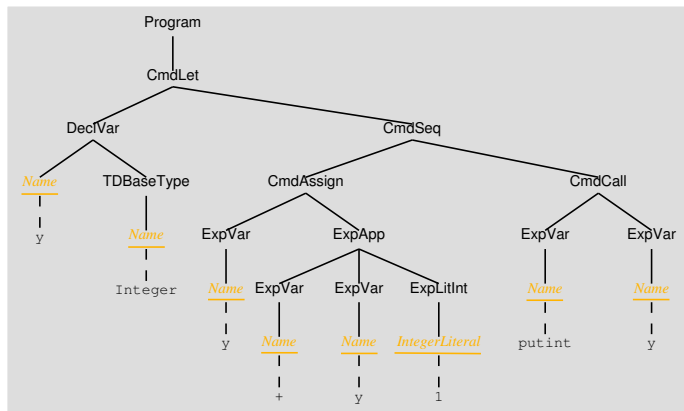
<i>Expression</i>	→	<u><i>IntegerLiteral</i></u>	ExpLitInt
		<u><i>Name</i></u>	ExpVar
		<i>Expression</i> (<i>Expression</i> *)	ExpApp
<i>Declaration</i>	→	const <u><i>Name</i></u> : <i>TypeDenoter</i>	DeclConst
		= <i>Expression</i>	
		var <u><i>Name</i></u> : <i>TypeDenoter</i>	DeclVar
		(:= <i>Expression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

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

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

COMP3012/G53CMP: Lecture 2 – p.30/37

Abstract Syntax Tree for the Program



Note: **fixed-spelling** terminals are **omitted** because they are implied by the node labels.

COMP3012/G53CMP: Lecture 2 – p.31/37

Concrete vs. Abstract Syntax

Key points:

- Concrete syntax: **string** (generated from the lexical and context-free grammars)
- Abstract syntax: **tree**

(Ways to describe **graphical** concrete syntax are more varied.)

COMP3012/G53CMP: Lecture 2 – p.32/37

Concrete AST Representation

Mapping of abstract syntax to algebraic datatypes

- Each non-terminal is mapped to a *type*.
- Each label is mapped to a *constructor* for the corresponding type.
- The constructors get one argument for each non-terminal and “variable” terminal in the RHS of the production.
- Sequences are represented by lists.
- Options are represented by values of type `Maybe`.
- “Literal” terminals are ignored.

COMP3012/G53CMP: Lecture 2 – p.33/37

Concrete AST Representation (3)

```
data Expression
  = ExpLitInt Integer
  | ExpVar Name
  | ExpApp Expression [Expression]

data Declaration
  = DeclConst Name TypeDenoter Expression
  | DeclVar Name TypeDenoter (Maybe Expression)
```

COMP3012/G53CMP: Lecture 2 – p.35/37

Concrete AST Representation (2)

```
data Command
  = CmdAssign Expression Expression
  | CmdCall Expression [Expression]
  | CmdSeq [Command]
  | CmdIf Expression Command Command
  | CmdWhile Expression Command
  | CmdLet [Declaration] Command
```

COMP3012/G53CMP: Lecture 2 – p.34/37

Concrete AST Representation (4)

In fact, the lab code uses labelled fields:

```
data Command
  = CmdAssign {
      caVar      :: Expression,
      caVal      :: Expression,
      cmdSrcPos  :: SrcPos
    }
  | CmdCall {
      ccProc     :: Expression,
      ccArgs     :: [Expression],
      cmdSrcPos  :: SrcPos
    }
```

COMP3012/G53CMP: Lecture 2 – p.36/37

Haskell Representation of the Program

```
CmdLet
  (DeclVar "y" (TDBaseName "Integer") Nothing)
  (CmdSeq [CmdAssign (ExpVar "y")
           (ExpApp (ExpVar "+")
                   [ExpVar "y",
                    ExpLitInt 1]),
          CmdCall (ExpVar "putint")
                  [ExpVar "y"]])
```

Assumption:

```
type Name = String
```