COMP3012/G53CMP: Lecture 2

Defining Programming Languages

Henrik Nilsson

University of Nottingham, UK

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)

Syntax and Semantics (1)

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

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

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".

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.

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.

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.

Syntax and Semantics (2)

- ***Semantics***: the ***meaning*** of programs

Syntax and Semantics (2)

- ***Semantics***: the ***meaning*** of programs
  - ***Static Semantics***: the static, at ***compile-time***, meaning of programs and program fragments.

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.

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***.

Defining Programming Languages (1)

- In order to develop a compiler (or other language processor):

Defining Programming Languages (1)

- In order to develop a compiler (or other language processor):
  - the ***Source Language*** must be defined

Defining Programming Languages (1)

- In order to develop a compiler (or other language processor):
  - the ***Source Language*** must be defined
    - syntax
    - semantics

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

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

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.

Defining Programming Languages (2)

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

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.

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.

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***.

Object Language and Meta Language

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

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

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.

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!

Informal Specifications

- In an informal specification, the meta language is a natural language such as English.

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 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.

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.)

Formal Specifications

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

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))

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

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.

Context-Free Grammars

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

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`.

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***.

CFG Notation (1)

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

CFG Notation (1)

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

- ***Mathematical style*** (or “G52LAC style”):

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.

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

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.

CFG Notation (2)

- ***Programming Language Specification style:***

CFG Notation (2)

- ***Programming Language Specification style:***
  - Used for larger, more realistic examples.

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 $\underline{\mathit{this}}$

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.

CFG Notation (3)

For example:

$$AssignStmt \rightarrow \underline{Identifier}\ \mathbf{:=}\ Expr$$

Here,

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

BNF and Extended BNF

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

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.

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

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

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).

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.

EBNF: Example

The following EBNF grammar

$$Block \quad \rightarrow \quad \texttt{\textbf{begin}} \quad (Decl \mid Stmt)^* \quad \texttt{\textbf{end}}$$

(where $Decl$ and $Stmt$ are defined elsewhere)

EBNF: Example

The following EBNF grammar

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

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

$$\begin{aligned} Block &\rightarrow \texttt{begin}\; BlockRec\; \texttt{end} \\ BlockRec &\rightarrow \epsilon \mid BlockRec\; BlockAlts \\ BlockAlts &\rightarrow Decl \mid Stmt \end{aligned}$$

EBNF: Example

The following EBNF grammar

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

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

$$\begin{array}{lcl} Block & \rightarrow & \textbf{begin}\ BlockRec\ \textbf{end} \\ BlockRec & \rightarrow & \epsilon \mid BlockRec\ BlockAlts \\ BlockAlts & \rightarrow & Decl \mid Stmt \end{array}$$

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

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^*
$[\ A\ ]$	$(A \mid \epsilon)$

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
```

Concrete Syntax

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

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***

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***.

MiniTriangle Lexical Syntax (1)

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & (\mathit{Token} \mid \mathit{Separator})^* \\
\mathit{Token} & \rightarrow & \mathit{Keyword} \mid \mathit{Identifier} \mid \mathit{IntegerLiteral} \mid \mathit{Operator} \\
 & & \mid \texttt{,} \mid \texttt{;} \mid \texttt{:} \mid \texttt{:=} \mid \texttt{=} \mid \texttt{(} \mid \texttt{)} \mid \underline{\mathit{eot}} \\
\mathit{Keyword} & \rightarrow & \texttt{begin} \mid \texttt{const} \mid \texttt{do} \mid \texttt{else} \mid \texttt{end} \mid \texttt{if} \mid \texttt{in} \\
 & & \mid \texttt{let} \mid \texttt{then} \mid \texttt{var} \mid \texttt{while} \\
\mathit{Identifier} & \rightarrow & \mathit{Letter} \mid \mathit{Identifier}\ \mathit{Letter} \mid \mathit{Identifier}\ \mathit{Digit} \\
 & & \text{except } \mathit{Keyword} \\
\mathit{IntegerLiteral} & \rightarrow & \mathit{Digit} \mid \mathit{IntegerLiteral}\ \mathit{Digit} \\
\mathit{Operator} & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{!} \\
\mathit{Separator} & \rightarrow & \mathit{Comment} \mid \underline{\mathit{space}} \mid \underline{\mathit{eol}} \\
\mathit{Comment} & \rightarrow & \texttt{//}\ (\text{any character except } \underline{\mathit{eol}})^*\ \underline{\mathit{eol}}
\end{array}
$$

MiniTriangle Lexical Syntax (2)

Notes:

MiniTriangle Lexical Syntax (2)

Notes:

- Essentially a (left-)linear grammar; i.e, the lexical syntax specifies a ***regular*** language.

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).

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:

$$\text{Keyword} \rightarrow \texttt{b e g i n} \mid \texttt{c o n s t} \mid \ldots$$

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 $\rightarrow$ `b e g i n` | `c o n s t` | ...

- Special characters are written like $\underline{\mathit{this}}$. Note! They are single terminal symbols!

MiniTriangle: Tokens

Some valid MiniTriangle tokens:

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

MiniTriangle: Tokens

Some valid MiniTriangle tokens:

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

Q: Is `const3` really a single token?

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***!

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!

MiniTriangle Context-Free Syntax (1)

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

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

MiniTriangle Context-Free Syntax (2)

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

MiniTriangle Context-Free Syntax (3)

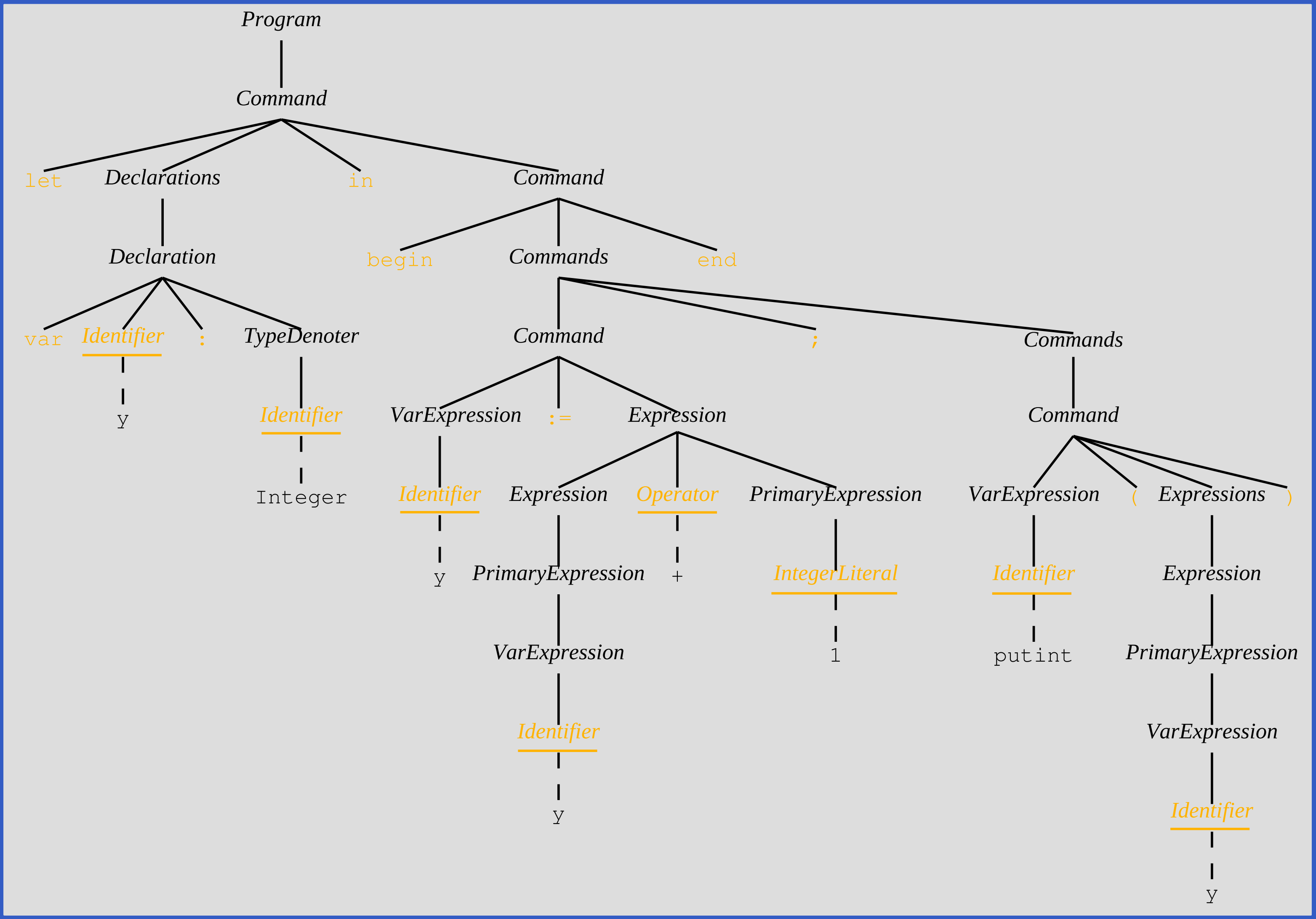
$$
\begin{array}{lcl}
\mathit{Declarations} & \rightarrow & \mathit{Declaration} \\
 & | & \mathit{Declaration}\ \texttt{;}\ \mathit{Declarations} \\
\mathit{Declaration} & \rightarrow & \texttt{const}\ \underline{\mathit{Identifier}}\ \texttt{:}\ \mathit{TypeDenoter}\ \texttt{=}\ \mathit{Expression}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

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
```

Parse Tree for the Program



Why a Lexical Grammar? (1)

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

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 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 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?

Why a Lexical Grammar? (2)

Answer:

- ***Simplicity***: dealing with white space and comments in the context free grammar becomes extremely complicated. (Try it!)

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.

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.

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.

$\mathit{Program}$	$\rightarrow$	$\mathit{Command}$	Program
$\mathit{Command}$	$\rightarrow$	$\mathit{Expression}$ `:=` $\mathit{Expression}$	CmdAssign
	$\mid$	$\mathit{Expression}$ `(` $\mathit{Expression}^*$ `)`	CmdCall
	$\mid$	$\mathit{Command}^*$	CmdSeq
	$\mid$	`if` $\mathit{Expression}$ `then` $\mathit{Command}$ `else` $\mathit{Command}$	CmdIf
	$\mid$	`while` $\mathit{Expression}$ `do` $\mathit{Command}$	CmdWhile
	$\mid$	`let` $\mathit{Declaration}^*$ `in` $\mathit{Command}$	CmdLet

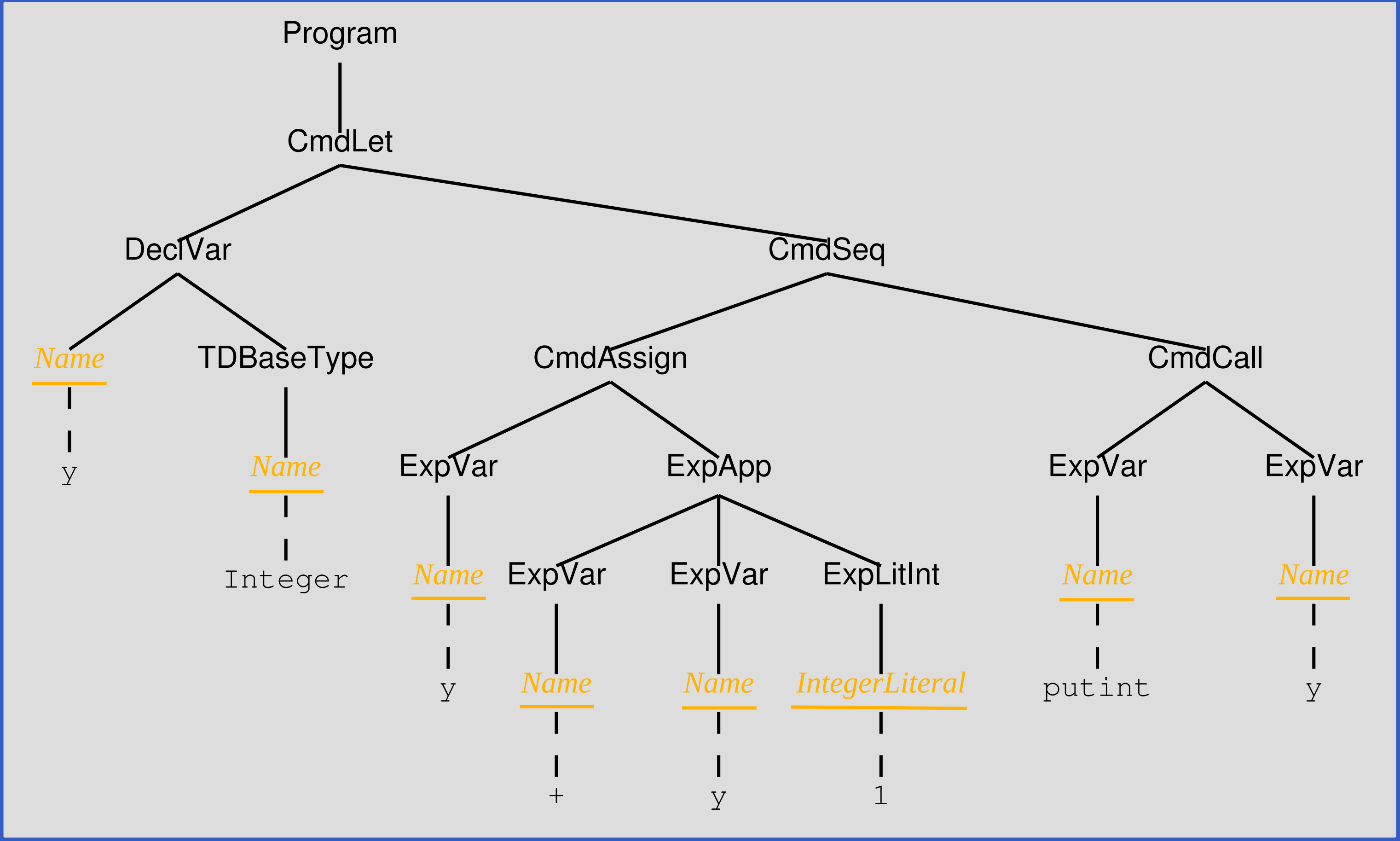
MiniTriangle Abstract Syntax (2)

Expression	$\rightarrow$	*IntegerLiteral*	ExpLitInt
	$\vert$	*Name*	ExpVar
	$\vert$	*Expression* **(** *Expression** **)**	ExpApp
Declaration	$\rightarrow$	**const** *Name* **:** *TypeDenoter* **=** *Expression*	DeclConst
	$\vert$	**var** *Name* **:** *TypeDenoter* (**:=** *Expression* $\vert$ ϵ)	DeclVar
TypeDenoter	$\rightarrow$	*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



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

Concrete vs. Abstract Syntax

Key points:

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

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.)

Concrete AST Representation

Mapping of abstract syntax to algebraic datatypes

Concrete AST Representation

Mapping of abstract syntax to algebraic datatypes

- Each non-terminal is mapped to a ***type***.

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.

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.

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.

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`.

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.

Concrete AST Representation (2)

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

Concrete AST Representation (3)

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

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

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
      }
    ...
```

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
```