

G54FOP: Lecture 2

Abstract Syntax and Induction on Terms

Henrik Nilsson

University of Nottingham, UK

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**

Example: MiniTriangle CFG (2)

Expressions → *Expression*
| *Expression* , *Expressions*

Expression → *PrimaryExpression*
| *Expression* *Operator* *PrimaryExpression*

PrimaryExpression → *IntegerLiteral*
| *VarExpression*
| *Operator* *PrimaryExpression*
| (*Expression*)

VarExpression → *Identifier*

Example: MiniTriangle CFG (3)

Declarations → *Declaration*
| *Declaration ; Declarations*

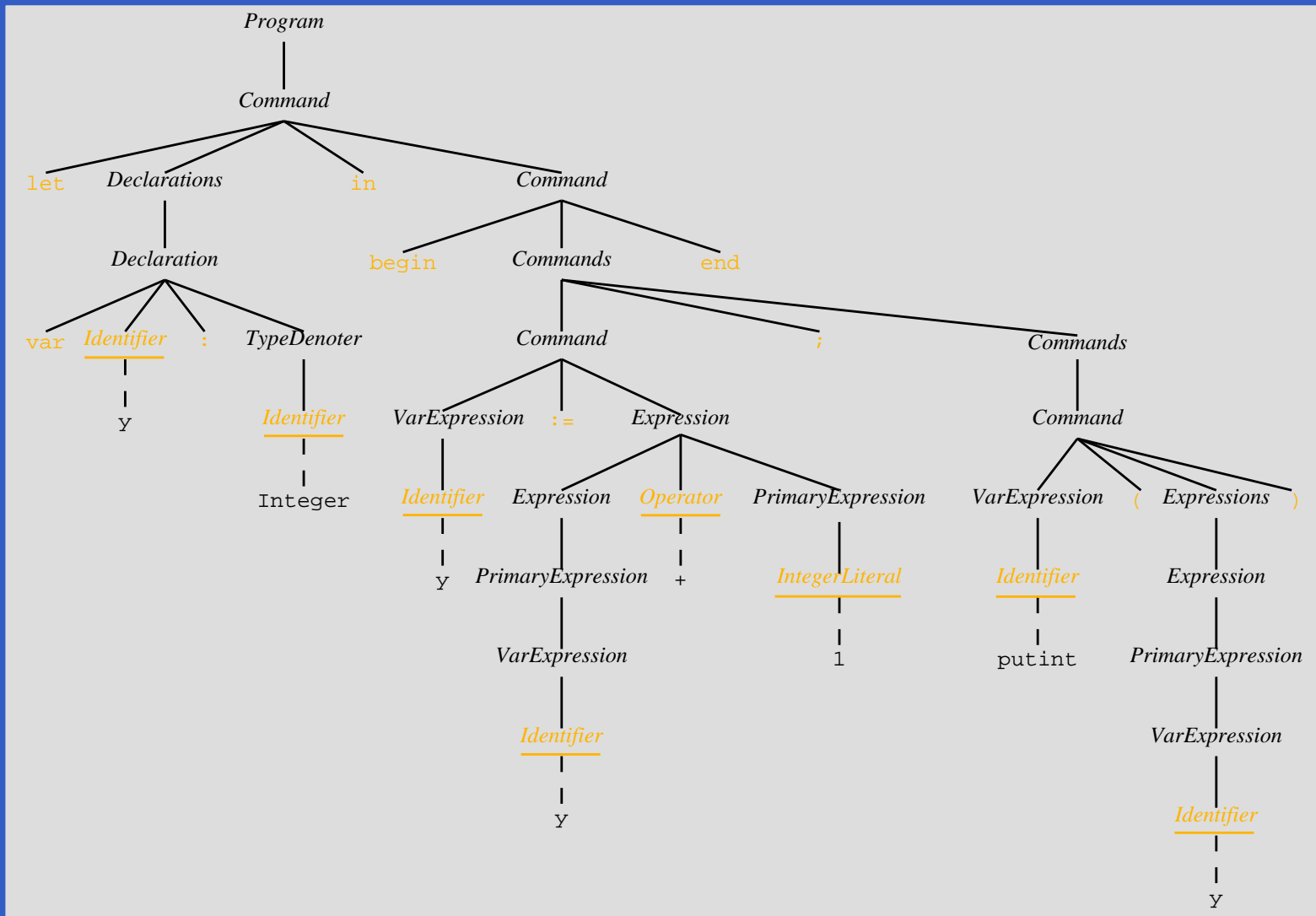
Declaration → **const** *Identifier* : *TypeDenoter* = *Expression*
| **var** *Identifier* : *TypeDenoter*
| **var** *Identifier* : *TypeDenoter* := *Expression*

TypeDenoter → *Identifier*

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>	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

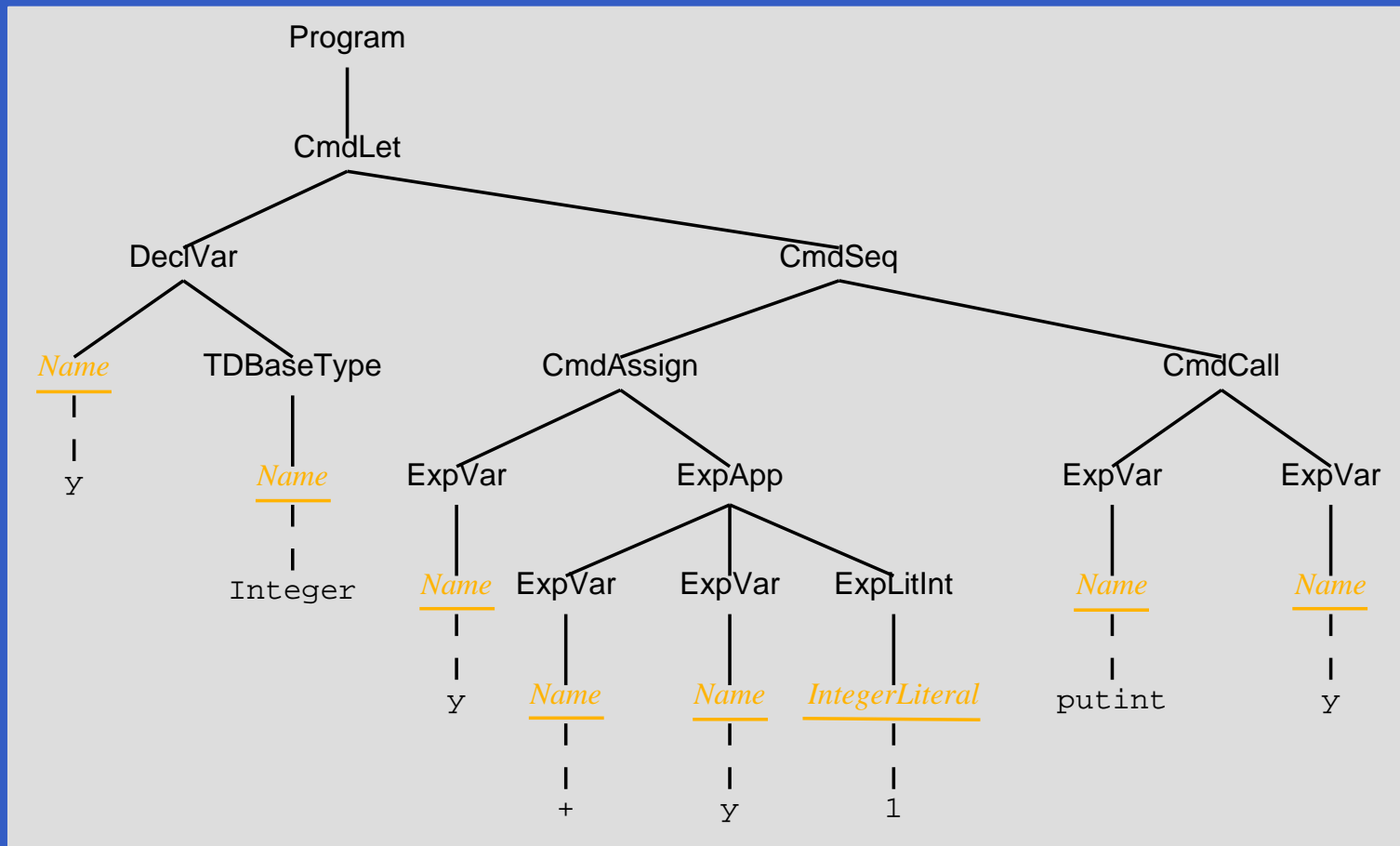
MiniTriangle Abstract Syntax (2)

<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 \subseteq *Name*, *Operator* \subseteq *Name*

Abstract Syntax Tree for the Program



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

Abstract Syntax for a Small Language

$t \rightarrow$

true

false

if t **then** t **else** t

0

succ t

pred t

iszero t

terms:

constant true

constant false

conditional

constant zero

successor

predecessor

zero test

Abstract Syntax for a Small Language

$t \rightarrow$		<i>terms:</i>
	true	<i>constant true</i>
	false	<i>constant false</i>
	if t then t else t	<i>conditional</i>
	0	<i>constant zero</i>
	succ t	<i>successor</i>
	pred t	<i>predecessor</i>
	iszero t	<i>zero test</i>

Exercise: State some terms according to this grammar?

Some Terms

Examples of terms:

- `true`
- `if true then 0 else succ 0`
- `if (iszero 0) then 0 else (succ (succ 0))`

Some Terms

Examples of terms:

- `true`
- `if true then 0 else succ 0`
- `if (iszero 0) then 0 else (succ (succ 0))`

Note: For **convenience**, terms will often be written as strings, with **parentheses where necessary** to make the structure clear. However, they **are** trees!

Some Terms

Examples of terms:

- `true`
- `if true then 0 else succ 0`
- `if (iszero 0) then 0 else (succ (succ 0))`

Note: For **convenience**, terms will often be written as strings, with **parentheses where necessary** to make the structure clear. However, they **are** trees!

Exercise: Draw the last term above as a tree.

Inductive Definition of Terms (1)

The set of trees described by the abstract syntax can be characterized by an **inductive definition**:

The set of terms is the **smallest** set τ such that:

1. $\{\mathbf{true}, \mathbf{false}, \mathbf{0}\} \in \tau$
2. if $t_1 \in \tau$, then
 $\{\mathbf{succ} t_1, \mathbf{pred} t_1, \mathbf{iszero} t_1\} \subseteq \tau$
3. if $t_1 \in \tau, t_2 \in \tau, t_3 \in \tau$, then
 $\{\mathbf{if} t_1 \mathbf{then} t_2 \mathbf{else} t_3\} \subseteq \tau$

This formulation makes it clear that the terms are an inductively defined set of **trees**.

Inductive Definition of Terms (2)

We will often use *inference rules* to state inductive definitions:

$$\mathbf{true} \in \tau \quad \mathbf{false} \in \tau \quad \mathbf{0} \in \tau$$

$$\frac{t_1 \in \tau}{\mathbf{succ} \ t_1 \in \tau}$$

$$\frac{t_1 \in \tau}{\mathbf{pred} \ t_1 \in \tau}$$

$$\frac{t_1 \in \tau}{\mathbf{iszero} \ t_1 \in \tau}$$

$$\frac{t_1 \in \tau \quad t_2 \in \tau \quad t_3 \in \tau}{\mathbf{if} \ t_1 \ \mathbf{then} \ t_2 \ \mathbf{else} \ t_3 \in \tau}$$

Principles of Induction on Terms

1. Structural induction:

If, for each term s , given $P(r)$ for all immediate subterms r of s we can show $P(s)$, then $P(s)$ holds for all terms s .

2. Induction on depth:

If, for each term s , given $P(r)$ for all r such that $\text{depth}(r) < \text{depth}(s)$ we can show $P(s)$, then $P(s)$ holds for all terms s .

3. Induction on size: similar to induction on depth, but using a suitable notion of size instead: $\text{size}(r) < \text{size}(s)$.