Case Study: Undefined Variables

CS 5010 Program Design Paradigms

Lesson 7.4

© Mitchell Wand, 2012-2016
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Learning Objectives

- At the end of this lesson the student should be able to:
  - explain the how defined and undefined variables work in our GarterSnake minilanguage
  - identify the undefined variables in a GarterSnake program
  - construct a data representation for a program in GarterSnake or a similar language
  - explain an algorithm for finding undefined variables in a GarterSnake program
  - understand how the algorithm follows the structure of the data representation
  - write similar algorithms for manipulating programs in GarterSnake or a similar simple programming language.

A Tiny Programming Language: GarterSnake

- We are writing a compiler for a tiny language, called GarterSnake.
- We want to write a program that checks a GarterSnake program for undefined variables.
- Let's describe the GarterSnake language:

The GarterSnake programming language: Programs

- A Program is a sequence of function definitions. The function defined in each definition is available for use in all of the following definitions.

Example: A GarterSnake program

```
def f1(x):f1(x)
; f1 is available in the body of f1
def f2 (x, y):f1(y)
; f1 is available in the body of f2
; spaces are ignored
def f3 (x,z): f1(f2(z,f1))
; f1 and f2 are available in the body of f3
; you can pass a function as an argument
def f4 (x, z):x(z,z)
; you can call an argument as a function
```

GarterSnake Definitions

- A Definition looks like

  def f(x1,..,xn):exp

- This defines a function named **f** with arguments **x1**, **x2**, etc., and body **exp**.
- The arguments of the function are available in the body of the function.
- The function **f** itself is also available in the body of the function.
- It is legal for a function to take no arguments.

GarterSnake Expressions

- An Expression is either a variable **v** or a function call **f(e1,..,en)** .
- **v** is a reference to the variable or function named **v** .
- **f(e1,e2,...)** is an application of **f** to the arguments **e1**, **e2**, etc.
- It is legal for a function to be applied to no arguments.
- There is no distinction between function names and argument names:
  - You can pass a function as an argument,
  - You can call an argument as a function.
  - You can return a function as the value of a function call.

The Problem: Undefined variables

An occurrence of a variable is *undefined* if it is in a place where the variable is not available. Examples:

I purposely called this **f7** to demonstrate that the names of the variables don't matter; it's just their position

```
def f7(x): f2(x)
; f2 is undefined in the body of f7
def f2(x,y): f3(y,x)
; f3 is undefined in the body of f2
def f3(x,z):f7(f2(z,y),z)
; y is undefined in the body of f3
```

The Requirements

Given a GarterSnake program p, determine whether there are any undefined variables in p.

```
;; program-all-defined?
;;  : Program -> Bool
;; GIVEN: A GarterSnake program p
;; RETURNS: true iff every variable
;; occurring in p is available at the
;; place it occurs.
```

Data Definitions

- We want to represent only as much information as we need to do the task.
- So we don't need to worry about spaces, details of syntax, etc.
- We just need to represent the structure of the programs.
- All the clues are already in the definitions

Data Definitions: Programs

- We said: A Program is a sequence of function definitions.
- So we write a corresponding data definition:

```
;; A Program is a ListOfDefinition
```

Data Definition: Definitions

- We wrote: A Definition looks like

  def f(x1,..,xn):exp

- So we write a data definition:

```
(define-struct def (name args body))
;; A Definition is a (make-def Variable ListOfVariable Exp)
;; INTERPRETATION:
;; name is the name of the function being defined
;; args is the list of arguments of the function
;; body is the body of the function.
```

Data Definition: Expressions

- We wrote: an Expression is either a variable **v** or a function call **f(e1,..,en)** .
- So we write a data definition

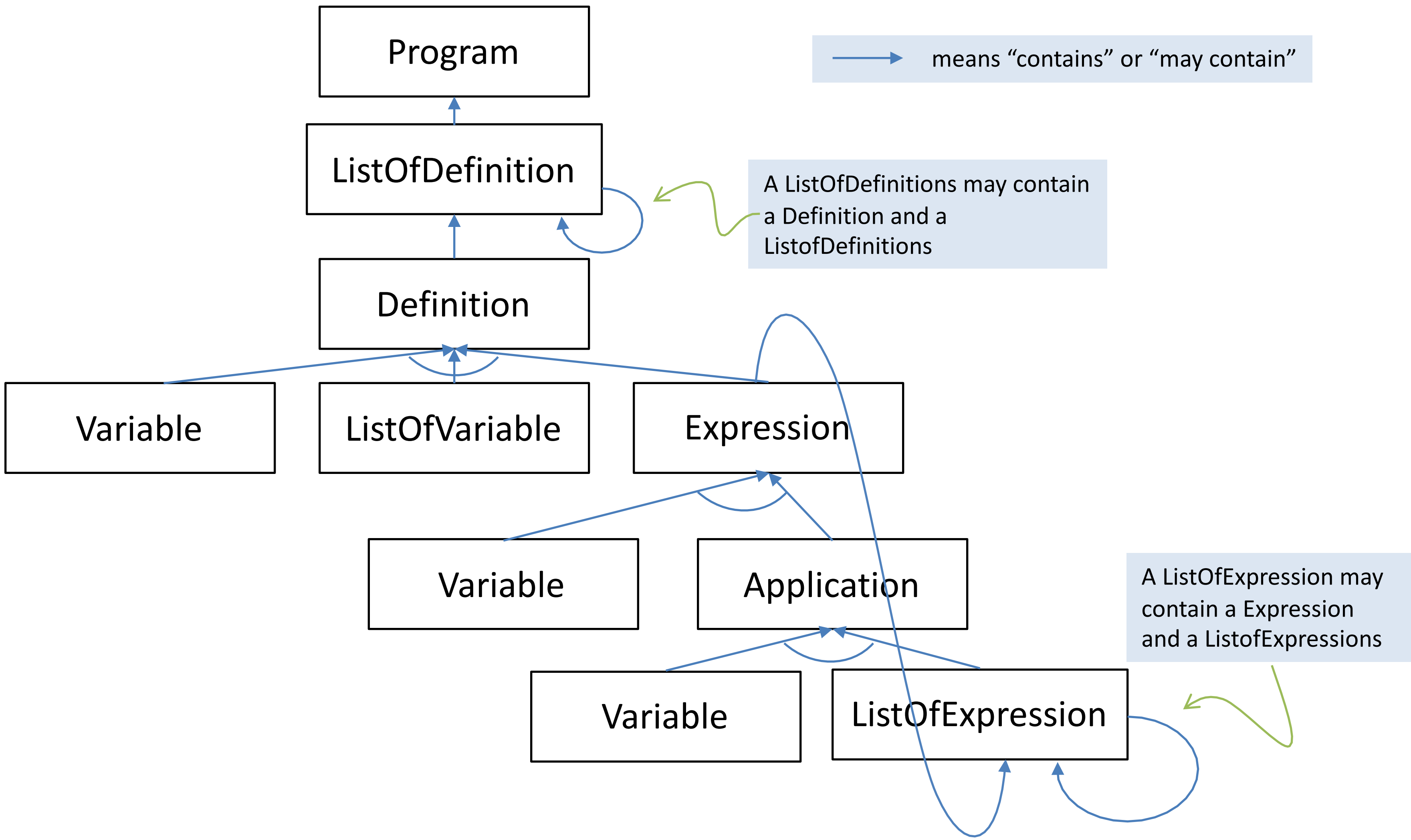
```
(define-struct varexp (name))
(define-struct appexp (fn args))

;; An Exp is one of
;; -- (make-varexp Variable)
;; -- (make-appexp Variable ListOfExp)
;; INTERPRETATION;
;; (make-varexp v)    represents a use of the variable v
;; (make-appexp f (list e1 ... en))
;;    represents a call to the function named f
;;    with arguments e1,..,en
```

Data Definition: Variables

- We never said anything about what is or isn’t a legal variable name. Based on the examples, we’ll choose to represent them as Racket symbols.
- We could have made other choices.
- Data Definition:

```
;; A Variable is a Symbol
```

Global View of the GarterSnake representation



Observer Templates

```
;; pgm-fn : Program -> ??
#;
(define (pgm-fn p)
  (lodef-fn p))

;; def-fn : Definition -> ??
#;
(define (def-fn d)
  (... (def-name d) (def-args d) (def-body d)))

;; exp-fn : Exp -> ??
#;
(define (exp-fn e)
  (cond
    [(varexp? e) (... (varexp-name e))]
    [(appexp? e) (... (appexp-fn e) (loexp-fn (appexp-args e)))]))

;; We omit the ListOf-* templates because they are standard and you should know
;; them by heart already.
```

In Racket, #; marks the next S-expression as a comment. So this definition is actually a comment. This is handy for templates.

Sidebar: Data Design in Racket

- We’ve chosen to represent GarterSnake programs as recursive structures.
- This is sometimes called “abstract syntax” because it abstracts away all the syntactic details of the programs we are manipulating.
- Recursive structures are our first-choice representation for information in Racket.
- You will almost never go wrong choosing that representation.

Sidebar: Symbols and Quotation

- Our data design uses *symbols*.
- A Symbol is a primitive data type in Racket.
- It looks like a variable.
- To introduce a symbol in a piece of code, we precede it with a quote mark. For example, **'z** is a Racket expression whose value is the symbol **z**.

Sidebar: Quotation (2)

- You can also use a quote in front of a list. Quotation tells Racket that the thing that follows it is a constant whose value is a symbol or a list. Thus
- Thus **‘(a b c)** and **(list ‘a ‘b ‘c)** are both Racket expressions that denote a list whose elements are the symbols **a**, **b**, and **c**.
- On the other hand, **(a b c)** is a Racket expression that denotes the application of the function named **a** to the values of the variables **b** and **c**.
- This is all you need to know about symbols and quotation for right now.
- There is lots more detail in HtDP/2e, in the Intermezzo entitled “Quote, Unquote”. But that chapter covers way more than you need for this course.

Data Design: Example

Now that we've briefly explained about symbols and quotation, we can give an example of the representation of a GarterSnake program

```
EXAMPLE:
def f1(x):f1(x)
def f2(x,y):f1(y)
def f3(x,y,z):f1(f2(z,y),z)
is represented by
(list
     (make-def 'f1 (list 'x)
               (make-appexp 'f1 (list (make-varexp 'x))))
     (make-def 'f2 (list 'x 'y) (make-appexp 'f1 (list (make-varexp 'y))))
     (make-def 'f3 (list 'x 'y 'z)
                (make-appexp 'f1 (list (make-appexp 'f2
                                             (list (make-varexp 'z)
                                                   (make-varexp 'y)))
                                       (make-varexp 'z)))))))
```

System Design (1)

```
;; We'll need to recur on the list structure of programs.  When we
;; analyze a definition, what information do we need to carry forward?
;; Let's look at an example.  We'll annotate each definition with a
;; list of the variables available in its body.

#|
def f1(x):f1(x)           ; f1 and x are available in the body.
def f2(u,y):f1(y)         ; f1, f2, u, and y, are available in the body.
def f3(x,z):f1(f2(z,f1)) ; f1, f2, f3, x, and z are available in the body.
def f4(x,z):x(z,z)        ; f1, f2, f3, f4, x, and z are available in the
body.
|#

;; In each case, the variables available in the body are the names of
;; the functions defined _before_ the current function, plus the names
;; of the current function and its arguments.
```

System Design (2)

```
;; Let's look at the "middle" of the calculation.
;; When we analyze the definition of f3, we need to know that f1 and
;; f2 are defined.  When we analyze the body of f3, we need to know
;; that f1, f2, x, and z are defined.

;; So we generalize our functions to take a second argument, which is
;; the set of defined variables.

;; We'll have a family of functions that follow the data definitions;

;; program-all-defined : Program                            -> Boolean
;; lod-all-defined?    : ListOfDefinition SetOfVariable -> Boolean
;; def-all-defined?    : Definition       SetOfVariable -> Boolean
;; exp-all-defined?    : Exp              SetOfVariable -> Boolean
```

lod-all-defined?

```
;; lod-all-defined? : ListOfDefinition SetOfVariable -> Boolean
;; GIVEN: a list of definitions 'defs' from some program p and a set of
;; variables 'vars'
;; WHERE: vars is the set of variables available at the start of defs in
;; p.
;; RETURNS: true iff there are no undefined variables in defs.
;; EXAMPLES: See examples above (slide 8)
;; STRATEGY: Use template for ListOfDefinition on defs.  The names
;; available in (rest defs) are those in vars, plus the variable
;; defined in (first defs).

(define (lod-all-defined? defs vars)
  (cond
    [(null? defs) true]
    [else
     (and
      (def-all-defined? (first defs) vars)
      (lod-all-defined? (rest  defs)
                        (set-cons (def-name (first defs))
                                  vars)))]))
```

You can't tell if a variable is undefined unless you know something about the program it occurs in! The WHERE invariant captures this information.

Don't say "see examples above" or "see tests below" unless there really are such examples or tests.

def-all-defined?

```
;; def-all-defined? : Definition SetOfVariable -> Boolean
;; GIVEN: A definition 'def' from some program p and a set of
;; variables 'vars'
;; WHERE: vars is the set of variables available at the start of def in
;; p.
;; RETURNS: true if there are no undefined variables in the body of
;; def.  The available variables in the body are the ones in def, plus
;; the name and arguments of the definition.
;; EXAMPLES: See examples above (slide 8)
;; STRATEGY: Use template for Definition on def

(define (def-all-defined? def vars)
  (exp-all-defined? (def-body def)
                    (set-cons
                     (def-name def)
                     (set-union (def-args def) vars))))
```

exp-all-defined?

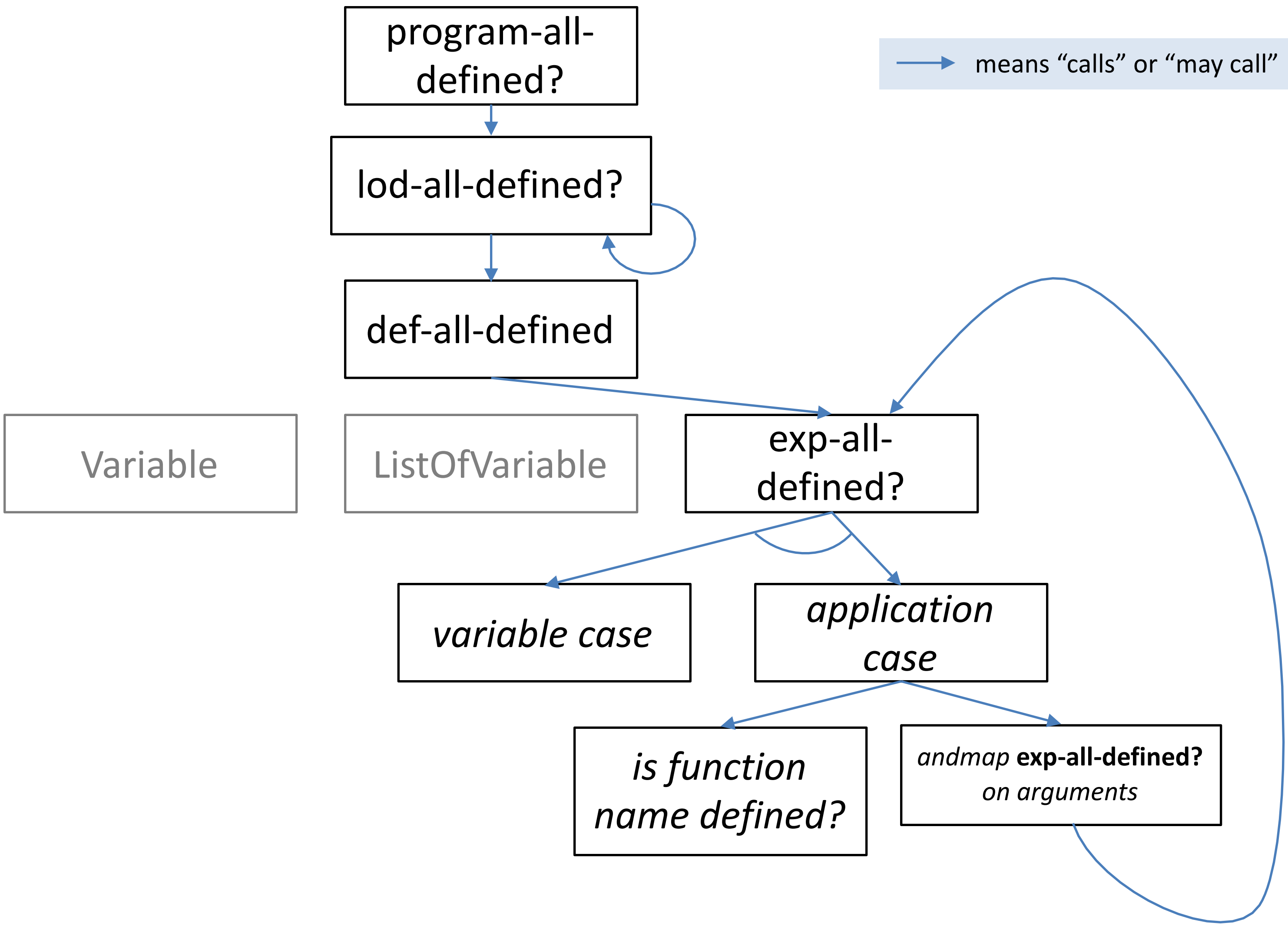
```
;; exp-all-defined? : Exp SetOfVariable -> Boolean
;; GIVEN: A GarterSnake expression e, occurring in some program
;;   p, and a set of variables vars
;; WHERE: vars is the set of variables that are available at the
;;   occurrence of e in p
;; RETURNS: true iff all the variable in e are defined
;; STRATEGY: Use template for Exp on e

(define (exp-all-defined? e vars)
  (cond
    [(varexp? e) (my-member? (varexp-name e) vars)]
    [(appexp? e)
     (and (my-member? (appexp-fn e) vars)
          (andmap
           (lambda (e1) (exp-all-defined? e1 vars))
           (appexp-args e)))]))
```

program-all-defined?

```
;; And finally, we can write program-all-defined?, which
;; initializes the invariant information for the other
;; functions.

;; program-all-defined? : Program -> Bool
;; GIVEN: A GarterSnake program p
;; RETURNS: true iff there every variable occurring in p
;; is defined at the place it occurs.
;; STRATEGY: Initialize the invariant of lod-all-defined?

(define (program-all-defined? p)
  (lod-all-defined? p empty))
```

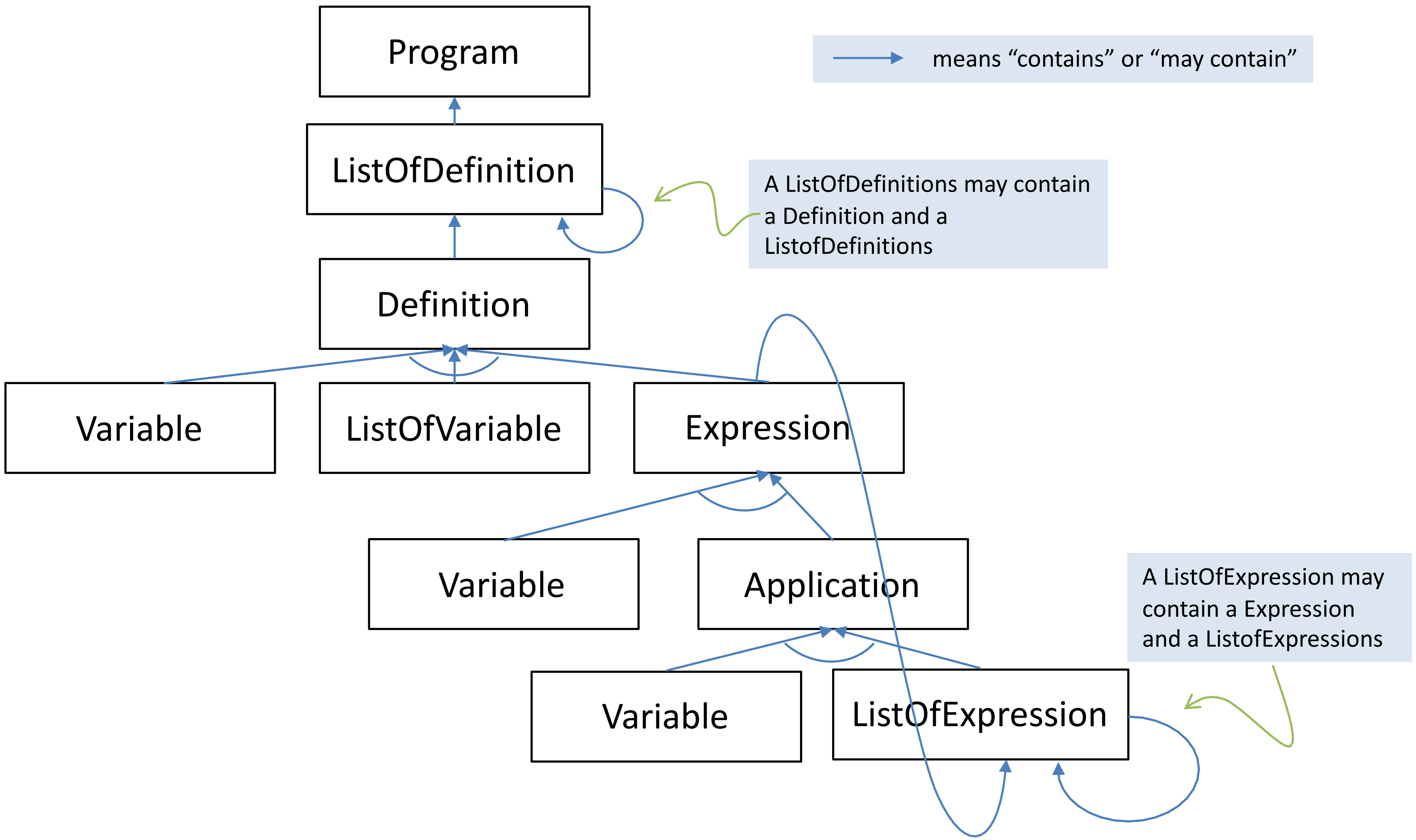
It would be ok to write “call a more general function” here, but this is more informative.

Call Graph for this Program



We used an **andmap** instead of defining **listofexp-all-defined?**

See how the call graph follows the structure of the data!



Summary

- You should now be able to:
  - explain how defined and undefined variables work in our GarterSnake minilanguage
  - identify the undefined variables in a GarterSnake program
  - construct a data representation for a program in GarterSnake or a similar language
  - explain an algorithm for finding undefined variables in a GarterSnake program
  - understand how the algorithm follows the structure of the data representation
  - write similar algorithms for manipulating programs GarterSnake or a similar simple programming language.

Next Steps

- Study Examples/07-3-gartersnake.rkt
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practices 7.2 and 7.3
- Go on to the next lesson