SE 2A04 Fall 2002

01 Fundamental Programming Concepts

Instructor: W. M. Farmer

Revised: 6 September 2002

What is a Program?

- A program is most often viewed as a **sequence of instructions for a machine**
  - An understanding of a program requires an understanding of the machine

- A **machine language program** is a sequence of instructions for a physical machine
  - Usually represented as a sequence of 0s and 1s
  - Not intelligible to humans

- A **high-level language program** can be viewed as a sequence of instructions for a high-level abstract machine
  - Easier to understand because the machine is simpler
  - Ultimately executed on a physical machine via **interpretation** or **compilation**

Other Ways of Viewing Programs

★ • As a small abstract machine
  - Good because the machine can be simple

• As a function that maps inputs to outputs
  - Good if the program has no **side-effects**

• As an expression in a formal language
  - The **syntax** of the expression is the program
  - The **semantics** of the expression is the behavior of the program
  - Good if the language is well behaved

• As a constructive proof of an existential formula
  - Very impractical with today's technology

Ways of Classifying Programs

- Sequential vs. concurrent
- Terminating vs. nonterminating
- Subject-invoked vs. event-triggered
- Applicative vs. systemic

SE 2A04 focuses on programs that are sequential, terminating, subject-invoked, and applicative

Programming Languages

- Programming languages are intended to facilitate program implementation but not necessarily program design

- There are many kinds of programming languages
  - Imperative (Examples: Pascal, C, Basic, Fortran)
  - Object-oriented (Examples: Smalltalk, C++, Java)
  - Higher-order languages (Examples: Lisp, Scheme, ML)
  - Functional (Examples: ML, Haskell)
  - Logical (Examples: Prolog)

- Oberon is an imperative language with some elements of object-oriented and higher-order languages

- The design of a program should be tied to a specific programming language as little as possible

Components of a Powerful Language

1. Primitive expressions

2. Means of combination
   - Compound expressions are built from simpler ones via constructors
   - The expressions denote combinations of objects

3. Means of abstraction
   - Compound expressions are built from simpler ones via constructors
   - The expressions denote new objects

Taken from Abelson, Sussman, and Sussman, *Structure and Interpretation of Computer Programs* (see references)

Example: Oberon

- Primitive expressions:
  - Characters, numbers, identifiers
  - Basic types
  - Basic operators and system-supplied procedures
- Means of combination:
  - Expression formation
  - Procedure call
  - Assignment (`:=`)
  - Composition (`;`)
  - Conditional selection (`IF`, `CASE`)
  - Iteration (`WHILE`, `REPEAT`, `LOOP`,`FOR`)
- Means of abstraction:
  - Type declarations
  - Variable and constant declarations
  - Module and procedure declarations

Example: Lambda Notation

- Lambda notation is used in many languages to express ideas about functions
  - **Lambda Calculus** (a model of computability)
  - **Simple Type Theory** (a higher-order predicate logic)

- Primitive expressions: variable and constant symbols for denoting primitive functions and individuals

- Means of combination: **function application** $f(a)$

- Means of abstraction: **function abstraction** $(\lambda\, x\, .\, s[x])$

- Conversion rules
  - Alpha: $(\lambda\, x\, .\, s[x]) = (\lambda\, y\, .\, s[y])$ (with no variable captures)
  - Beta: $(\lambda\, x\, .\, s[x])(t) = s[t]$ (with no variable captures)

Data Structures

- A **data structure** is a structured collection of **values**
  - Values include booleans, characters, integers, and floating-point numbers (**atomic values**)
  - Values may also include some data structures (**compound values**)

- Various operators are associated with each kind of data structure:
  - **Constructors** for creating data structures
  - **Selectors** for retrieving the values in data structures
  - **Mutators** for modifying the values in data structures

- Some data structures do not have mutators

Data Structure Example: Pair

- Constructor: `pair(a,b)` creates a "pair" from two values `a` and `b`
- Selectors:
  - `first(p)` returns the first value of the pair `p`
  - `second(p)` returns the second value of the pair `p`
- Mutators:
  - `set-first(p,x)` sets the first value of the pair `p` to the value `x`
  - `set-second(p,x)` sets the second value of a pair `p` to the value `x`

Types

- A **type** is a syntactic object t that denotes a set s of values
  - t and s are often confused with each other

- Types are used in a variety of ways:
  - To classify values (latent types)
  - To classify variables (manifest types)
  - To control the formation of expressions
  - To classify expressions by value

- Types are also used as “mini-specifications”

Type Examples

- Mathematical types:
  - $\mathbf{Z}$: denotes the set of integers
  - $\mathbf{R}$: denotes the set of real numbers
  - $\mathbf{Z} \rightarrow \mathbf{R}$: denotes the set of functions from the integers to the real numbers

- Oberon types
  - `INTEGER`: set of machine integers between `-32768` and `32767`
  - `REAL`: set of floating point numbers between `-3.4E+38` and `3.4E+38`
  - `ARRAY OF CHAR`: set of arrays holding characters, i.e., members of the Oberon type `CHAR`

Variables

- The meaning of “variable” is different in logic, control theory, and programming

- In logic, a variable is a **symbol** that denotes an **unspecified value**

- In control theory, a variable is a **changing value** that is a component of the **state** of a system
  - A **monitored variable** is a variable the system can observe but not change
  - A **controlled variable** is a variable the system can both observe and change

- In programming, a variable is a **data structure** composed of a single value and with the following attributes:
  - **Name**: An identifier bound to the variable
  - **Value**: The single value stored in the variable
  - **Type**: The type of the values that can be stored

Oberon Variables

- A **variable declaration** such as

  ```
  VAR sum: INTEGER;
  ```

  serves as the **constructor** for a variable

  - `sum` is the **name** of the variable
  - `INTEGER` is the **type** of the variable
  - The **value** of the variable is initially empty

- The **name** of a variable (e.g., `sum`) serves as the **selector** for a variable

- An **assignment statement** such as

  ```
  sum := 17;
  ```

  serves as the **mutator** for a variable

Binding vs. Assignment

- **Binding** associates an identifier with a value
  - An identifier i bound to a value v means that i is a name for v
  - Several identifiers can be bound to the same value
  - Binding does not modify data structures
- **Assignment** changes a value in a data structure
- An Oberon variable declaration binds an identifier to a variable, while an Oberon assignment statement changes the value of a variable

Constants

- The meaning of "constant" is different in logic, control theory, and programming
- In logic, a constant is a **symbol** that denotes a **specified value**
- In control theory, a constant is an **unchanging value**
- In programming, a constant is a **variable without mutators**
  - The use of constants is essential for code readability and software maintenance

Oberon Constants

- A **constant declaration** such as

  ```
  CONST pi = 3.14;
  ```

  serves as the **constructor** for a constant

  - `pi` is the **name** of the constant
  - 3.14 is the **value** of the constant
  - The **type** of the constant is the type of 3.14, i.e., `REAL`

- The **name** of a constant (e.g., `pi`) serves as the **selector** for a constant

- The value of a constant cannot be changed (at run time): there is no mutator for a constant

Scope

- The **scope** of an identifier i bound to a value v is the region of program code in which the binding is effective
  - The scope is usually the region of code from the place where i was first bound to the end of the smallest enclosing "block" of code
  - An identifier i is only visible in its scope, i.e., outside of its scope i will normally not be bound to v

- If i is rebound within its scope, a new scope of i is created in which the old binding is not visible

- In Oberon, module and procedure declarations serve as blocks

- In accordance with the **Principle of Least Privilege**, the scope of a variable name should be as narrow as possible

Persistence

- The **persistence** of a data structure (e.g., a variable) is the period of time the data structure is available to a running program

- Examples:
  - The persistence of a running function procedure begins when it is called and ends when it returns a value
  - The persistence of a variable declared in a procedure normally has the same persistence as the procedure
  - The persistence of an Oberon module is normally from when it is first imported to the termination of the program

Argument Passing Conventions

The most common conventions for passing arguments to procedures are:

- **Call-by-name**: the argument is passed without being evaluated
  - Arguments to macros are usually passed this way
- **Call-by-value**: the value of the argument is passed
  - If the argument is a name of a variable x, assignments to its corresponding formal parameter have no effect effect on x
- **Call-by-reference** when the argument is a name of a variable x, the corresponding formal parameter of the procedure is also bound to x
  - Assignments to the formal parameter are effectively assignments to x

Argument Passing in Oberon

- The variables declared in a procedure heading are called the **formal parameters** of the procedure

- The arguments passed to the formal parameters in a procedure call are called the **actual parameters** of the procedure call

- Oberon procedures can have two kinds of formal parameters:
  - A **value parameter** is passed an argument using call-by-value
  - A **variable parameter** is passed an argument using call-by-reference

- A value parameter or variable parameter is indicated by the absence or presence of the keyword `VAR`, respectively

A Simple Pseudocode (ASP) 1

Declarations:

1. Type: `type <typename> = <typeexpr>`

2. Variable: `var <varname> : <typeexpr>`

3. Constant: `const <constname> : <typeexpr> = <expr>`

4. Procedure:

```
proc <procname>(vardecllist) : <typeexpr>
  <statement>
end
```

A Simple Pseudocode (ASP) 2

Statements:

1. Declaration: `<typedecl>`, `<vardecl>`, `<constdecl>`, or `<procdecl>`

2. Procedure call: `<procname>(<exprlist>)`

3. Return: `return <expr>`

4. Assignment: `<varname> := <expr>`

5. Composition: `<statement> ; <statement>`

A Simple Pseudocode (ASP) 3

6. Conditional selection:

```
case
  (<expr> , <statement>),
    .
    .
    .
  (<expr> , <statement>)
end
```

7. Iteration:

```
loop
  <condition>,
  <statement>
end
```

Backus-Naur Form (BNF)

- The **Backus-Naur Form (BNF)** is a formal metasyntax used to describe the syntax of a context-free language

- A line of BNF syntax has the form

  <*meta-variable-name*> ::= *bnf-expression*

  where *bnf-expression* is build from metavariables; the meta-symbols |, [, and]; and the symbols of the language

  - | means disjunction
  - [*bnf-expression*] means *bnf-expression* is optional

- There are many variations of BNF including **Extended BNF (EBNF)** with the additional metasymbols *, +, {, and }