CSC 326H1F, Fall 2008

Programming Languages

Instructor: Ali Juma
ajuma@cs.toronto.edu

What languages do you know?

- The usual suspects: C, C++, Java
  - fine languages
  - nearly the same
- Perhaps you've also learned some others?
  - assembler
  - Basic, Visual Basic, Turing, Pascal, Fortran, Modula, Eiffel
  - Perl, Python, PHP, C#, Ruby
- But these are also nearly the same.
- What about these?
  - Scheme/Lisp
  - Prolog
  - Haskell
  - ML

A survey of counted loops ("for loops"): C, C++, Java

```
int sum = 0;
int i;
for (i = 1; i < 11; i++)
  sum += i;
/* Note the semicolons everywhere. */
```


A survey of counted loops: FORTRAN

```
      SUM = 0
      DO 100 I = 1,10
100   SUM = SUM + I
```

A survey of counted loops: COBOL

```
   MOVE 0 TO SUM
   MOVE 1 TO I
   PERFORM ADD-I-TO-SUM
      UNTIL I = 11.

ADD-I-TO-SUM
   ADD I TO SUM
   ADD 1 TO I.
```


A survey of counted loops: Turing

```
var sum := 0

for i : 1..10
  sum += i
end for
```

-- or --

```
var sum := 0
type range : 1 .. 10
for i : range
  sum += i
end for
```


A survey of counted loops: Python

```
sum = 0

for i in range(11):
    sum += i
```


Looping over a list

- Python:

```
for i in list:
  # Do something with i.
```

- Java:

```
for (Iterator it = myList.iterator(); it.hasNext(); ) {
  MyClass mc = (MyClass) it.next();
  // Do something with mc.
}
```

- Java 1.5:

```
List<MyClass> myList; // It's a list of MyClass objects.
for (MyClass mc : myList ) {
  // Do something with mc.
}
```

What's the point?

- Not "Java is wonderful" (though it is).
  - Some of you are advocates of C++, C#, Python, Perl, etc.
  - They all have good points, and you can program well in them and enjoy it.
- Here's what our examples mean:
  - There are many languages, and they say the same thing in very different ways.
  - Over time, what you're allowed to say has changed, and the changes tend to make programming easier.
  - What you're allowed to say affects what you are able to think.
- And that's why we study programming languages: to be able to think better because we've seen different ways of expressing programs.

A few languages

- FORTRAN
- Algol
- LISP
- COBOL
- PL/I
- SNOBOL
- APL
- Pascal
- Simula
- Basic
- C
- Prolog
- Smalltalk
- Scheme
- Ada
- Turing
- C++
- ML
- Java
- Perl
- Python
- C#

References: Sebesta, chapter 2; and (a nice diagram)
http://www.oreilly.com/news/graphics/prog_lang_poster.pdf

Major topics of this course

- Introduction
- Object-oriented programming: Python
- Types and values
- Syntax and semantics
- Functional programming: Scheme
- Exceptions
- Logic programming: Prolog
- We won't be spending the same amount of time on each topic.

Administrative stuff

- Course website: http://www.cs.toronto.edu/~ajuma/326f08
  - Announcements, slides, and assignments will be posted here.
  - There will be a bulletin board for discussion too.
  - Read the announcements and the bulletin board regularly! I will assume that you are aware of any information posted there.
  - You'll need to sign up for a bulletin board account – use your ECF email address.
- Bulletin board: arranged by topics
  - Post in the correct location – mis-posted items may not be answered.
- Email to me: used only for personal issues
  - not for advice about a lab, project or test – these questions belong on the bulletin board.
  - Include "CSC326" in the subject line of your email message.
- My office hours: TBA

Text and other books

- Text: R.W. Sebesta, *Concepts of Programming Languages*, Addison-Wesley, 2008.
  - This is the **eighth** edition.
  - We will cover a lot of the text, but far from all of it. The chapter references in the slides are to this book.
- Other books: none required.
  - You will need to write programs in Python, Scheme and Prolog.
  - I bet you can manage without buying anything, but if you're serious about any of the languages you should buy something about that language.
    - Python is the most likely candidate. A good book is: Mark Lutz, *Learning Python*, O'Reilly, 2007.
      - When using any Python book, keep in mind that we will be using Python 2.5.

Grading scheme

- 25%: three labs, one project
  - due at **10:30 a.m.**: Sept. 29, Oct. 27, Nov. 17 (project), Dec. 1
    - all Mondays
  - submission is by ECF "submit" command
    - deadline is by ECF clock
  - weights: 5% for each lab, 10% for project
  - labs are to be done *individually*, and the project is to be done in a group of size at most 3.
  - late penalty: 20% per day up to 2 days, and not accepted after that.
- 25%: two tests
  - Oct. 6 and Nov. 3 (*to be confirmed*)
  - weights: 12.5% each
  - closed-book except that a single, double-sided aid ("cheat") sheet will be allowed
    - Produce this sheet any way you like, using letter-sized (that is, 8.5" x 11") paper.
- 50%: final exam
  - same aids allowed as on the tests

Labs and project

- Labs focus on the paradigm represented by the language, not on algorithm design.
- The project will be done in Python.
- what will get a good mark:
  - successful running *on the ECF workstations* (p1, p2, …)
    - Failure to run may result in a mark of 0, even if the code shows signs of intelligence.
  - completion of a significant part of the requirements
  - clear style
    - good variable names
    - helpful comments
  - good documentation
    - *within* the code unless otherwise specified

Re-mark requests

- Deadline for requests: one week after the marked work was available for return.
- Use the same standard form from the web site for labs, projects and tests.
- As usual: marks may go up or down – but I don't *plan* to lower them.
  - Markers are imperfect, and aren't insulted if you ask for a check.

Tutorials

- Most weeks, the tutorial will be run as a help session where a TA will be present to answer questions.
  - SF 1012, Fridays at 12, starting Sept. 12
    - We also have SF 1013 available if SF 1012 gets full.
- Any exceptions to this will be announced in class.

Academic offenses

It is not OK to "work together". Don't even *look* at anyone else's code.

- The labs are to be done individually. The project is to be done by your group only.
- All work must include the name and student number of the author or authors.
- You may use material from other sources if you include attribution.
  - This includes the textbook and all other references, including online and printed.
- If you include full attribution, you have not committed an academic offense.
- If your attributions indicate that you haven't done much work, then the mark you get may be rather low.
- I'll be checking in two cases:
  - all submissions together, where possible
  - in more detail, where there's reason to wonder
- (Why do I care?)

The current topic: introduction

- Introduction
  - ✓ reasons for studying languages
  - language classifications
    - level, natural data types, paradigm
  - simple syntax specification
- Object-oriented programming: Python
- Types and values
- Syntax and semantics
- Functional programming: Scheme
- Exceptions
- Logic programming: Prolog

Kinds of languages

- We can classify by:
  - level
  - natural data types
  - paradigm
- Levels:
  - machine
  - assembly
  - high-level
- Data types:
  - COBOL: records
  - FORTRAN: "real" numbers
  - LISP and Scheme: lists, functions

Terminology: "first class"

- *First-class* "objects" are things that the language allows programs to:
  - declare
  - assign values to
  - return as function values
    - or: create without a name
- Example: functions/methods
  - C: can't be created *by programs* (that is, C functions can't *construct* new C functions) or assigned values
    - but you can have pointers to functions
  - Java: can't even have function pointers
  - Scheme: functions are first-class objects.
- Example: types
  - C: types can't be created or manipulated at runtime, but can be declared in your code (typedefs and structs).
  - C++, Java: you can have template arguments (known as type parameters in Java) which take on a type as their value, but you still can't create or manipulate types at runtime.
  - Who would want to create types at runtime?

Programming paradigms

- imperative: program is sequence of orders to be carried out
  - Fortran, Cobol, C, Basic
- object-oriented: program is an interacting community
  - Smalltalk, C++, Java, ***Python***
- functional: program consists *only* of function definitions and function calls; in particular, there are no assignment statements, no loops.
  - Lisp, ***Scheme***, ML
- logic: program finds variables that satisfy predicates
  - ***Prolog***

How does this affect a language's power?

- It doesn't. According to the Church-Turing thesis, anything that can be computed can be computed by a *Turing machine*, and all of our languages are capable of doing all that a Turing machine can do. That is, all of our languages have equivalent computational power.
- But our languages may differ in how easy it is to express particular computations.
- So we choose a language by how well it suits what we're trying to do, not for whether or not it permits us to do what we want.

The current topic: introduction

- Introduction
  - ✓ reasons for studying languages
  - ✓ language classifications
  - simple syntax specification
- Object-oriented programming: Python
- Types and values
- Syntax and semantics
- Functional programming: Scheme
- Exceptions
- Logic programming: Prolog

Syntax

- What are you allowed to say? -- *syntax*.
- What does it mean? -- *semantics*.
- We'll come back to both syntax and semantics later on. For now, we just need to know a little about describing syntax.
- Reference: Sebesta, Chapter 3.

Describing syntax

The standard notation used in computing is *Backus-Naur form* or BNF.

- Example 1 (in C/Java):

  ```
  x = y + z;
  ```

  <asst stmt> ::= <vbl> '=' <expr> ';'
  - This *rule* says that an assignment statement is a variable name, followed by an = operator, followed by an expression, followed by a semicolon. Of course, we'd also need to give definitions for <vbl> and <expr>.
- Example 2 (for a simplified language):

  <expr> ::= <vbl>
  | <vbl> '+' <vbl>
  | <vbl> '-' <vbl>
  - This rule says in our simplified language, an expression is a variable name, or a variable name followed by a + operator followed by a variable name, or a variable name followed by a - operator followed by a variable name. Note that the symbol "|" means "or".

Backus-Naur form (BNF)

- On the left, the form being described.
- Then, the ::= (or →) symbol.
- Then, the components allowed in the form, with pipes used to separate multiple allowed definitions of the same form.
- Components that must be exactly as shown are put in quotation marks: e.g. '='.
- The names of the form and of components that are forms themselves are put in angle brackets <…> or distinguished by a special typeface.
- In other words, BNF is like this:

  <BNF description> ::= <form> '::=' <components>

Extended BNF

- Repetition: use curly brackets, which mean "zero or more" of their contents.

  <comma expr> ::= <expr> { ',' <expr> }
  - A comma expression consists of one or more expressions, separated by commas.
- Optional parts: use square brackets, which mean "zero or one" of their contents.

  <if stmt> ::= 'if' '(' <expr> ')' <stmt> ['else' <stmt>]
  - An "if" statement has at most one "else" clause.
- Alternatives: use brackets and pipes, which mean "choose exactly one"

  <expr> ::= <vbl> ('+' | '-' | '*' | '\') <vbl>
  - An expression consists of a variable followed by exactly one symbol followed by another variable.
- BNF and Extended BNF are equivalent in what they can describe, but Extended BNF improves readability.

BNF usage

- BNF style varies:
  - <…> may be replaced by font choice or subscripts
  - quotation marks may be replaced by font choice or lack of subscripts
- Sometimes usage can be a little loose.
  - The reader is supposed to figure it out from context.