Programming Languages
UCLA-CS131-S18

Quentin Truong
Taught by Professor Eggert

Spring 2018

Contents

1 Ch1: Programming Languages

1.1 Introduction

- Practical Magic
  - Useful and beautiful
- Programming Languages
  - ML, Java, Prolog

1.2 The Amazing Variety

- Imperative Languages (C)
  - Hallmarks: assignment and iteration
- Functional Languages (ML, Lisp)
  - Hallmarks: recursion and single-valued variables
  - Factorial is natural to functional
- Logic Programming Languages (Prolog)
  - Express program in terms of rules about logical inferences and proving things
  - Factorial is very not natural to logic programming; not well suited to mathematical functions
- Object-oriented Programming Languages (Java)
  - Object is a bundle of data which knows how to do things to itself
  - Helps keep large programs organized
- Other categories
  - Applicative, concurrent, constraint, declarative, definitional, procedural scripting, single-asstgnmen
- Multi-paradigm
  - JavaScript, OCaml, Python, Ruby
- Others
  - FORTH is stack-oriented
  - APL is a unique functional language relying on large character sets with many symbols that most users don't have

1.3 The Odd Controversies

- Partisans
  - For every language b/c some advantages
  - But all languages have advantages and disadvantages
  - Disagreement even on basic terminology, like object oriented

1.4 The Intriguing Evolution

- Programming languages change
  - All change; new ones evolve from old ones
  - Many have several dialects
  - Fortran is entirely only dialects (sequence of standards)

1.5 The Many Connections

- Styles
  - Object Oriented, like Java -¿ objects
  - Functional, like ML -¿ many small functions
  - Logic, like Prolog -¿ express problem as searches in logically defined space of solutions
- Language evolution driven by hardware + applications
  - AI encouraged Lisp; Classes bc Simula

1.6 A Word about Application Programming Interfaces

- Application Programming Interfaces (API)
  - May implement data structures, GUI, network input/output, encryption, security, other services
  - Is much of language; more than the printed specification of the language

2 Ch2: Defining Program Syntax

2.1 Introduction

- Syntax
  - Language definition that says how programs look (form and structure)
  - Appearance, delimiters, etc
- Semantics
  - language definition that says what programs do (behavior and meaning)
  - How it works, what can go wrong, etc
- Formal grammar
  - Used to define programming language syntax

2.2 A Grammar Example for English

- English
  - Article, noun, noun phrase, verb, sentence composes subset of unpunctuated English
  - Grammar used as set of rules that say how to build a parse tree (sentence at root)
  - Language defined by grammar is set of all strings that can be formed as fringes of parse trees

2.3 A Grammar Example for a Programming Language

- Infinite language have arbitrarily long expressions
  - Recursive grammar where exprsesions can be children of expressions
    - Expressions can be sum/product/enclosed/variable of two expressions

2.4 A Definition of Grammars: Backus-Naur Form

- Tokens
  - Smallest units of syntax
  - Strings and symbols not consisting of smaller parts (cat, if, !=)
- Non-terminal symbols
  - Correspond to different language constructs (sentences, noun pharses, statements)
  - Special non terminal symbol <empty¿
- Productions
  - Possible way of building parse tree
  - LHS is non-terminal; RHS is sequence of one or more things
- Start symbol
  - Special non-terminal symbol
- <if-stmt >::= if <expr >then <stmt >else-part >
  - <else-part >::= else <stmt >— <empty >

2.5 Writing Grammars

- Divide and Conquer
  - <var-dec >::= <type-name ><declarator-list >;
  - <declarator-list >::= <declarator >— <declarator >, <declarator-list >
  - BNF syntax defines programming language constructs

2.6 Lexical Structure and Phrase Structure

- Lexical Structure
  - How to divide program text into tokens
- Phrase Structure
  - How to construct parse trees with tokens at leaves

- Separate lexical and phrase structure
  - Otherwise, is ugly, hard to read, and complicated
- Lexer
  - Reads input file and converts to stream of tokens, discarding white space and comments
- Parser
  - Reads stream of tokens and forms parse tree
- Free-format languages
  - End-of-line is no more special than space or tab
  - Most modern languages don't care for column position, so could write program as a single line
  - Python is an exception

2.7 Other Grammar Forms

- Backus-Naur Form (BNF)
  - Has many minor variations, use = or ->instead of ::=
  - Metasymobls are part of language of the definition, not of the language being defined
- Extended Backus-Naur Form (EBNF)
  - Might use brackets, parentheses, etc
  - [optional], {repeatable}, (group)
  - Use quotes to denote tokens as not metasymbols
- Syntax Diagrams (Railroad diagram)
  - Way to express grammars graphically
  - Uses circles, rectangles, and arrows to show flow and possible control flows
  - Railroad diagram bc many many arrows
  - Good for casual use; hard for machines + parse trees
- Formal, Context-free Grammars
  - Formal languages study formal grammars
  - Context-free b/c children of node in parse tree depend only on that node's non-terminal symbol (not on context of neighboring nodes in tree)
  - Regular grammars (less expressive, good for lexical structure) and context-sensitive grammars (more expressive, good for phrase structure) both exist

2.8 Conclusion

- Grammars
  - Used to define syntax (lexical and phrase structure)
  - Lexical is division of program text into meaningful tokens
  - Phrase is organization of tokens into parse tree for meaningful structures
- Good grammars
  - If grammar is in the correct form, can be fed into parser-generator
  - Simple, readable, short grammars are more memorable + easier to learn/use

3 Ch3: Where Syntax Meets Semantics

3.1 Introduction

- Grammar
  - Set of rules for constructing parse trees
  - Language defined by grammar is set of fringes of parse trees
- Equivalent Grammars
  - Different grammars may generate identical languages (bc identical fringes despite different internal structure)
- Internal structure of parse tree
  - Semantics must be unambiguous

3.2 Operators

- Operator (+, *)
  - Refers to both the tokens for the operation, and the operation itself
  - Unary, binary, ternary take one, two, three operands
- Operands
  - Inputs to operator
- Infix Notation
  - Operator between operands
  - Postfix has operator after operands

3.3 Precedence

- Higher precedence performed before lower precedence
  - Use different non-terminal symbol for each precedence level
  - Non-terminal symbols in this chain are in order of precedence, from lowest to highest (is generalizable)
- Precedence Levels
  - Smalltalk has 1 precedence level (no precedence)
  - C has 15, Pascal has 5
  - Can add unnecessary parentheses to make expressions more readable

3.4 Associativity

- Grammar for a language must generate only one parse tree for each expression
  - So need to implement left/right-associative (eliminate the other direction)
  - <exp >::= <exp >+ <exp >
  - <exp >::= <exp >+ <mulexp >| <mulexp >
  - Only recursive on left side, so that it is left associative and grows tree left
- Nonassociative operator
  - Has no defined behavior when used in sequence in expression
  - Prolog 'a :- b :- c' is just a syntax error; also '1 <2 <3'

3.5 Other Ambiguities

- Ambiguous Grammars
  - Allows construction of two different parse trees for same string
- Dangling else
  - Optional else part, so grammar may be ambiguous
  - Can fix using <fullstmt >
  - Or use indenting (Python) or other ways to fix it

3.6 Cluttered Grammars

- Novice users
  - Just want to find out what legal programs look like
- Advanced users and language-system implementers
  - Need exact, detailed definition
- Automatic tools
  - Complete and unambiguous
  - Not sensitive to clutter

3.7 Parse Trees and EBNF

- EBNF can be easier to read
  - Eliminates some confusing recursions, but obscures structure of parse tree
  - Must quote many tokens, bc they are tokens in the language being defined
  - Grammar still incomplete without explanation of intended associativity for binary operators

3.8 Abstract Syntax Trees

- Abstract Syntax Trees (AST)
  - Node for every operation, with subtree for every operand
  - Many language systems ust AST as internal representation of program
  - Type-checking and post-parsing carried out on AST

3.9 Conclusion

- Grammar defines more than syntax
- Unique parse trees allow us to begin to define semantics
- Parse trees and ASTs are where syntax meet semantics

4 Ch5: A First Look at ML

4.1 Introduction

- Standard ML is the popular functional language
  - Mostly learned just for knowledge of programming languages

4.2 Getting Started with an ML Language System

- -1 + 2 * 3
  - val it = 7 : int
  - Very powerful type-inference system
  - 'it' is a variable whose value is the value of the last expression

4.3 Constants, Operators, Conditional Expressions, Type Conversion and Function Application, Variable Definition

- Constants
  - real if decimal, int if integer, negation operator is a tilde
  - ML is case sensitive
- Operators
  - Integers: + - * div mod
  - Reals: + - * tilde /
  - String: ^for concat
  - Inequality: ><
- If-then-else
  - if <expr >then <expr >else <expr >;
- Type conversions
  - Never does conversions automatically
- Parentheses around function parameters are unnecessary
- Variable definition
  - val x = 1 + 2;
  - Letter followed by zero or more additional letters, digits, underscores
  - ML expects input to be a series of definitions, so if type just an expression, will set val it = exp

4.4 Garbage Collection, Tuples and Lists

- GC #0.0.0.0.1.3 (0 ms)
  - ML prints this if performing garbage collection (SML/NJ)
- Tuples
  - Ordered collection of values of different types (tuples OK too)
  - val p1 = ("red", (300, 200))
    - val p1 = ("red", (300,200)) : string * (int * int)
  - * is type constructor
  - To extract ith element of tuple named v, write #i v
- List
  - Elements must be of same type
  - Uses square brackets
  - Empty list is nil or [] and has unknown type
  - @ symbol does concatenation for lists
  - cons (construct) is written as :: and wil glue elements onto front of list
  - head (hd) and tail (tl) will extact first/all except first parts of a list
- Type variables
  - Type that is unknown

- 'a list might be any type
- x = [] is restricted to type, so use null x instead

– Recursive Functions
- Could use hd to take first element, and recursively call, resulting in iteration

4.5 Function Definitions

– Polymorphic function
- Paramters allow different types
- Ex: List length function

4.6 ML Types and Type Annotations

– Types: int, real, bool, char, string
– Constructors: * for tuples, list for lists, -¿ for function types
– Type annotations: necessary for ambiguous situations
- fun prod(a:real, b:real) : real = a * b;
- Many larger ML projects use type annotations heavily

5 Ch7: A Second Look at ML

5.1 Patterns

- A variable is a pattern that matches anything and bnids to it
- Underscore character matches anything and does not introduce new variables
- Constant is pattern that matches only that constant value
- Tuple of patterns is pattern that matches tuple of any right size, whose contents match subpatterns
- List of patterns is pattern that matches list of rgiht size, whose contents match subpatterns
- Cons of patterns is pattern that matches non-empty list whose head and tail match the subpatterns
  - x :: xs matches any non-empty list; binds x to head; binds xs to tail

5.2 Using Multiple Patterns for Functions

- fun f 0 = "zero" | f 1 = "one";
  - Two different function bodies (still nonexhaustive)
  - If overlapping, tries patterns in order they are listed, using the first one that matches

5.3 Pattern-Matching Style

- If-else
  - Equivalent to the multiple patterns
  - Multiple patterns is often preferred and cleaner
- Functions
  - null l returns true if the list l is empty
  - length l returns the number of elements in the list l
  - hd l returns the first element of l
  - tl l returns all but the first element of l
- Variable name cannot be used more than once unless you want them to be legal
- Patterns can be used in definitions
  - val (a, b) = (1, 2.3);
    - val a = 1 : int
    - val b = 2.3 : real
  - val a :: b = [1, 2, 3, 4, 5];
    - val a = 1 : int
    - val b = [2, 3, 4, 5] : int list

5.4 Local Variable Definitions and Nested Function Definitions

- <let-exp >::= let <definitions >in <expression >end
  - <definitions >hold only within <let-exp >
- Generally don't need to use # to extract from tuple, can use pattern matching
- Use halve function to divide list into pair of half-lists
  - half [1]
    - val it = ([1], []) : int list * int list
- Use merge function to merge
- Local functions
  - Can define functions inside other function definitions

6 Ch9: A Third Look at ML

6.1 Introduction

- Case expressions
  - Pattern matching can be used in many other places, including case expressions
- Higher order functions
  - Take other functions as parameters or produces them as returned values
  - Used more often in functional languages than in imperative languages

6.2 More Pattern Matching

- Rule
  - <rule >::= <pattern >= ><expression >
- Match
  - <match >::= <rule >| <rule >'|' <match >
- Case
  - <case-exp >::= case <expression >of <match >

6.3 Function Values and Anonymous Functions

- Predefined functions like ord and -:
  - Variables just like others, but initially bound to functions
- New functions and bind name to function automatically
  - fun f x = x + 2
- Anonymous function
  - fn x =>x + 2
  - Useful when need small function in one place and don't want to clutter
- op <
  - Extracts the function used by the operator <

6.4 Higher-Order Functions and Currying

- Order
  - Function that does not take any functions as parameters and does not return a function value has order 1
  - Function that takes a function as a parameter or returns a function value has order n + 1, where n is the order of its highest-order parameter or returned value
  - Higher-order function is an nth order function where n is greater than 1
- Currying
  - Use higher order functions to pass multiple parameters into a function
  - Function takes first parameter and returns another function, which takes second parameter and returns final result
  - Is an alternative way to passing multiple parameters (could pass tuple)
  - Main advantages is that we can pass only some of the parameters, and save the function
  - fun g a b c is equivalent to fun g a =>fn b =>fn c

6.5 Predefined Higher Order Functions

- map
  - Applies some function to every element in the list
- foldr
  - Combines all elements into one value with starting value
  - foldr (op *) 1 [1, 2, 3, 4];
    - val it = 24 : int

    - foldr (op ::) [5] [1, 2, 3, 4];
        - val it = [1, 2, 3, 4, 5] : int list
    - foldr (op ^) "" ["abc", "def", "ghi"]
        - val it = "abcdefghi" : string
- foldl
    - same as foldr, except proceeds from left to right

7 Ch9:

7.1 Introduction

–