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-assignment
- 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 nonterminal 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
  - In Prolog, '1 <2 <3' is meaningless because <doesn't work that way

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 (Extended Backus-Naur Form) 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 use 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 Ch4: Langauge Systems

4.1 The Classical Sequence

- Language System
  - Is what makes programming languages actually work
- Integrated Development Environment
  - Single interface for editing, running, and debugging programs
- Classical Sequence of Language-System steps
  - Editor creates source file
  - Compiler creates assembly-language file
    - Translates program from original high-level language into assembly langauge (instructions for processor)
  - Assembler creates object file
    - Converts each instruction into machine language
  - Linker creates executable file
    - Collect and combine different parts of program
  - Loader creates a running program in memory

4.2 Optimization

- Compiler automatically optimizes code to make it faster and/or take up less memory
- Loop invariant removal
  - Moves calculations of values which don't change outside the loop
- Code may be reorganized, removed, or added
  - Not obvious what assembly code will be generated

4.3 Variations on the Classical Sequence

- Hiding steps
  - gcc will compile, assemble, and link
  - Using gcc, as, and ld can recreate these steps
- Integrated Development Environment
  - Integrated editor will supply indenting and colors to make it easier to read
  - May also provide source-control features to track verisons and coordinate collaboration
  - May help with rebuilding automatically, especially when only a few files have changes
- Interpreters
  - Executes program without translating it beforehand
- Virtual Machines
  - Perform classical sequence of language-system steps to get an executable for a virtual machine
  - Virtual-machine simulator is an interpreter, and compiles high-level program into intermediate code
  - Virtual machines are often platform independent and more secure (b/c is never in control of physical processor)
  - Java virtual machine is intermediate code supported on many different physical machines
  - Wide spectrum of intermediate languages, ranging from pure interpreter to compiler
- Delayed Linking
  - Windows' load-time dynamic linking links program to all library functions just before program begins running (.dll)
  - Windows' run-time dynamic linking require explicit calls to find and load functions from library (.dll)
  - Unix shared libraries are linked by loader just before program begins running (.so)
  - Unix dynamically loaded libraries require explicit calls to find and load functions during runtime (.so)
  - Multiple programs can share library function (fewer copies)
  - Library function may be updated independently of the program
  - Only load what is necessary, so startup is faster

- Profiling
  - Instead of guessing what part of code is most important, compile twice
  - Compile program, run it and collect statistics, then recompile and generate better code
- Dynamic Compilation
  - Just-in-time compilation
  - Compile after program starts running, once the function is actually called
  - Java virtual machine interprets bytecode, which is slower, so it uses JIT compilation to get machine code to run
  - JVM will only compile what is used often, as determined by call count

4.4 Binding Times

- Binding
  - Act of associating properties with names
  - Properties are things like what are the set of values, type, memory location, value, etc
- Language-Definition Time
  - Keywords are part of language definition and language-system implementations must conform (void, for, etc)
- Language-Implementation Time
  - Properties left out of language definition (range for int in C and ML)
  - May introduce limits, such as on maximum length of name, levels of nesting, etc
- Compile Time
  - Type of variable is bound
  - Bind variable definitions to reference; nontrivial for large programs
- Link Time
  - Find definitions of library functions to match each reference
- Load Time
  - Load into memory (may make minor adjustments)
- Runtime
  - Early binding
  - Bound before runtime
    - More secure, less things go wrong
  - Late binding (runtime binding)
    - More runtime flexibility
  - Each high-level language has different binding-time implications
    - In lisp, may not know type until runtime

4.5 Debuggers

- Core dump
  - When langauge system hits fatal defect, writes copy of memory to a file
  - Useful for postmortem analysis
  - Language-system tool later extracts meaningful information from dump file, such as traceback, value of variables, etc
- Interactive debuggers allow programmer to inspect and modify variables while still running
  - Debugger must express things in terms of the original program, not the compiled changes
  - Must know the names of variables/functions, not just memory addresses
  - But the optimizations of modern compilers weaken the connection between source and machine code, meaning that debuggers must conceal the actions of compilers and give the programmer the impression of debugging directly
- Integrated language system can provide editing and recompiling program while still running

4.6 Runtime Support

- Runtime support
  - Additonal code included even though program does not explicitly refer to it, used in the following:
  - Startup processing, to set up processor and memory in thew ay high-level code expects
  - Exception handling, finding the exception handler or catching exceptions
  - Memory management, icluding allocation and freeing
  - Operating system input communication
  - Concurrrent execution

4.7 Conclusion

- Classical Sequence
  - Edit, compile, assemble, link, load, run
- Binding times
  - Language definition, language implementation
  - Early, late
- Debuggers and runtime support
- Programming language is not the same as programming-language system

5 Ch5: A First Look at ML

5.1 Introduction

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

5.2 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

5.3 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

5.4 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

5.5 Function Definitions

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

5.6 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

6 Ch6: Types

6.1 Menagerie of Types

- A Type is a Set
  - Can only have values from a certain set
  - Share common representation in how they are encoded in memory
  - Supports particular operations
- Primitive Types and Constructed Types
  - Primitive types are like int and real
  - Constructed types are defined by programmer
  - Definition of programming language may say exactly what set each primitive type must be, or may leave room for choosing different sets
    - Java wants programs to have same result across platforms, so defines all types completely
    - C and ML allow different implementations (int)
- Enumerations
  - Define new types by listing all elements of the type
  - Enums in C just wraps integers, allowing you to add elements as if integers
  - ML completely hides the implementation
- Tuples
  - Generalized to n-tuples of any type
  - Aggregate types are tuples, records, arrays, strings, lists
  - Scalar types are primitives and enumerations
- Arrays, Strings, Lists
  - Set A^n is set of all vectors of length n
  - Array indices are not always only integers (Pascal)
- Unions
  - Multiple elements in same block of memory
  - Discriminated Union supports values which can be one of many types
- Subtypes
  - Subset of an existing set
  - Can support superset of operations
- Function Types
  - From set A to set B

6.2 Uses for Types

- Type Annotations
  - Explicit type information used to determine type/documentation
  - Required in Java; allowed in ML; disallowed in Prolog, but comments serve the purpose
  - ANSI Basic has ultra-compact type annotation inside the variable name
- Type Inference
  - Most languages infer types based on operands or value
  - ML infers types for every expression
- Type Checking
  - Point is to prevent incorrect operations to operands
  - Static type checking determines type for everything before running
    - Generates consistent picture of types
  - Dynamic type checking checks types at runtime
    - Helpful for operators, etc
  - Many systems implement both static and dynamic bc hard to know exact type + is expensive to check all types
  - Strongly typed languages have type checking which is thorough enough to prevent any mistakes
  - Java is strongly typed, Pascal is somewhat weak, C is more weak

– Type-Equivalence Issues
  - Name-equivalence is when types are equivalent iff same name
  - Structural equivalence is when types are equivalent iff constructed from same primitive types using same constructors in same order

6.3 Conclusion

– Types are similar to mathematical sets
– Abstract type describes types defined only in terms of supported operations
– Static vs dynamic type checking
– Name vs structural equivalence
– Type annotation vs inference

7 Ch7: A Second Look at ML

7.1 Patterns

- Patterns match data and introduce new variables
- 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

7.2 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

7.3 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

7.4 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

8 Ch8: Polymorphism

8.1 Introduction

– Polymorphic functions can be applied to multiple types

8.2 Overloading

– Overloaded function is one that has at least two definitions, all of different types
  • Can overload operators like + to mean different things on different types (int, real, string)
  • Can overload functions to take int, double, etc
  • Often up to the programmer to use overloading sensibly

8.3 Parameter Coercion

– Implicit type conversion is called coercion
  • Specification for coercions can be very complex
  • Coercion is often useful to programmers, but sometimes leads to confusing errors due to unexpected coercion
– Coercion for functions
  • May be ambiguous, f(int, char) vs f(char, int)
    – f ('a', 'a')

8.4 Parametric Polymorphism

– Parametric polymorphism is when type contains one or more type variables
  • Substituting any type for "a will result in a valid function
– Type with type variables like "a * "a ->bool is a polytype
– Implementing Parametric Polymorphism
  • Generate separate copies of polymorphic function, one for each instatiation of type variables, and behave like overloading
    – Optimized but takes more memory
  • Or can create single general copy, which isn't optimized for each instantiation

8.5 Subtype Polymorphism

– A function or operator exhibits subtype polymorphism if one or more of its parameters have subtypes
– Polymorphic Functions
  • Subtypes are accepted in addition to the original type

8.6 Conclusion

– Dynamic Type Checking
  • Polymorphism is a way to gain the freedom of dynamic type checking without losing benefits of static type checking
– Four types of polymorphism
  • Overloading - multiple definitions of function each with different types
  • Parameter Coercion - type conversion
  • Parametric polymorphism - Substituting a specific type for a general type
  • Subtype polymorphism - Functions can accept subtype even if asking for parent type
– Additional types of Polymorphic Langauges
  • Polymorphic variables can be of different classes
  • Polymorphism to call different function depending on actual class of object
  • Temmplates for class definitions are parametric polymorphism
– Ad Hoc Polymorphism

    - Is at least two, but finitely many types
    - Overloading, parameter coercion
- Universal Polymorphism
    - Has infinitely many types (no limit to what is possible)
    - Parametric polymorphism, subtype polymorphism

9 Ch9: A Third Look at ML

9.1 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

9.2 More Pattern Matching

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

9.3 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 <

9.4 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

9.5 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

10 Ch10: Scope

10.1 Introduction

- Nearly all languages allow reuse of variable names
  - Thus nearly all languages use scoping to some extent

10.2 Definitions and Scope

- Definitions
  - Bindings for names to variables, functions, types, classes, etc
  - Also referred to as declarations and other things
- Scope
  - In the scope of a given definition of that name whenever that definition governs the binding for the occurrence

10.3 Scoping with Blocks

- Blocks
  - Any language construct that contains definitions and the region of the program where those definitions apply
  - Inner blocks override outer blocks
  - Classic block scope rule: Scope of a definition is block containing that definition, starting from point of definition to end of block, minus scope of any redefinitions of the same name
  - Other variations exist
- ML
  - Let, fun, and pattern matching all create blocks
- C
  - Braces create compound statements

10.4 Scoping with Labeled Namespaces

- Labeled Namespaces
  - Any language construct that contains definitions and a region of the program where those definitions apply
  - And has name that can be used to access those definitions outside of context
  - Helps prevent namespace pollution (which is caused by everyone using the same short, memorable names)
- ML
  - structures
- C
  - namespace creates labeled namespaces
- Visibility within namespaces
  - ML uses interfaces
  - C uses public/private

10.5 Scoping with Primitive Namespaces

- Primitive Namespaces
  - Not created using the language, are part of the language definition
  - When searching for things like type, it is in the primitive namespace
  - May have many different primitive namespaces, such as for packages types, methods, etc (Java)

10.6 Dynamic Scoping

- Classic Dynamic Scope Rule
  - Scope of a definition is the function containing that definition, from the point of definition to the end of the function, along with any functions when they are called (including indirectly), from within that scope, minus scopes of any redefinitions
  - Basically, variables are defined by their own scope; if not there, then its defined by caller's scope; if not there, keep going up
- Dynamic Scoping
  - Hard to implement efficiently bc need to look up bindings at runtime
  - Few languages use it b/c leads to large scopes + debugging headaches

10.7 A Word about Separate Compilation

- Separate Compilation
  - Program is written as many small files
  - Pieces are compiled separately, linked together
  - Recent trends are showing 'separate compilation' as less and less separate, bc they may depend on each other, or need results of multiple, etc

10.8 Conclusion

- Scope
  - Cannot ask 'what is the scope of x'
  - Must ask 'what is the scope of this definition of x'
  - Names do not have scopes, definitions do

11 Ch11: A Fourth Look at ML

11.1 Enumerations

- Enumerations
  - Simple use of datatype is to create enumerations
  - datatype flip = Heads | Tails;
  - flip is a type constructor
  - Heads is a data constructor
  - Enumerations have enforced types, so normal type checking applies to them

11.2 Data Constructors with Parameters

- Data constructors can have parameters
  - datatype exint = Value of int | Plusinf | Minusinf
  - Use pattern matching to extract the value
  - Allows us to wrap values together

11.3 Type Constructors with Parameters

- Polymorphic type constructors
  - Allows us to change the type of our type constructor just by passing a different variable
  - datatype 'a option = NONE | SOME of 'a
- option type constructor
  - Useful for functions whose result is not always defined (division may result in NONE if div by 0)

11.4 Recursively Defined Type Constructors

- In data constructor, can use type constructor being defined
  - Allows us to define things like list and binary trees
    - datatype intlist = INTNIL | INTCONS of int * intlist

12 Ch4: Langauge Systems

12.1 Introduction

–