CS558 Programming Languages

Winter 2013

Lecture 1

Goals of the Course

• Learn fundamental structure of programming languages.

• Understand key issues in language design and implementation.

• Increase awareness of the range of available languages and their uses.

• Learn how to learn a new language.

• Get a small taste of programming language theory.

Method of the Course

• Fairly conventional survey textbook, with broad coverage of languages.

• Homework exercises involve programming problems in real languages.

• Most homework problems will involve modifying **implementations** of "toy" languages that illustrate key features and issues.

• Exercises will use two modern languages: **Python** and **OCaml**.

• Between them, these languages illustrate many of the important concepts in current language designs.

Non-goals

• Teaching how to program.

• Teaching how to write significant programs in any particular language(s).

• Surveying/cataloging the features of lots of different languages.

• Comprehensive coverage of programming paradigms (e.g., will skip logic and concurrent programming material).

• Will mostly be concerned with interpreting **abstract syntax** for the toy languages, and will spend very little time on parsing and code generation. (Not a compiler course!)

Some Languages

What languages do you know?

FORTRAN, COBOL, (Visual) BASIC, ALGOL-60, ALGOL-68, PL/I, C, C++, RPG, Pascal, Modula, Oberon, Lisp, Scheme, ML, Haskell, Ada, Prolog, Goedel, Curry, Snobol, ICON, . . .

Don't forget things like:

scripting languages: perl, tcl, Python, . . .

SQL, other database query languages.

spreadsheet expression languages

text processing languages, tex, awk, etc.

application-specific languages.

"Higher-Level" Programming Languages

Consider a simple (dumb) algorithm for testing primality.

In Python:

```
def isprime(n):
    # return true if n has no divisor in interval [2,n-1]
    for d in range(2,n):
        if n % d == 0:
            return False
    return True
```

"Higher-Level" Programming Languages (2)

In OCaml (using a recursive function):

```
let isprime (n:int) : bool =
  (* return true if n has no divisor in interval [2,n-1] *)
  let rec no_divisor (d:int) : bool =
      (* return true if n has no divisor in interval [d,n-1] *)
      (d >= n) || ((n mod d <> 0) && no_divisor (d+1)) in
  no_divisor 2
```

In Intel X86 (32 bit) Assembler:

```
.globl isprime
isprime:
      pushl %ebp          ; set up procedure entry
      movl %esp,%ebp
      pushl %esi
      pushl %ebx
      movl 8(%ebp),%ebx   ; fetch arg n from stack
      movl $2,%esi        ; set divisor d := 2
      cmpl %ebx,%esi      ; compare n,d
      jge true            ; jump if d >= n
loop: movl %ebx,%eax      ; set n into ....
      cltd                ; ... dividend register
      idivl %esi          ; divide by d
      testl %edx,%edx     ; remainder 0?
      jne next            ; jump if remainder non-0
      xorl %eax,%eax      ; set ret value := false(0)
      jmp done
next: incl %esi           ; increment d
      cmpl %ebx,%esi      ; compare n,d
      jl loop             ; jump if d < n
true: movl $1,%eax        ; set ret value := true(1)
done: leal -8(%ebp),%esp ; clean up and exit
      popl %ebx
      popl %esi
      leave
      ret
```

High-level Languages: General Characteristics

- Complex Expressions (Arithmetic, Logical, ...)
- Structured Control Operators (Loops, Conditionals, Cases)
- Composite Types (Arrays, Records, etc.)
- Type Declarations and Type Checking
- Multiple storage classes (global/local/heap)
- Procedures/Functions, with private scope, maybe first-class
- Maybe high-level control mechanisms (Exceptions, Back-tracking, etc.)
- Maybe abstract data types, modules, objects, etc.

Machine Code Characteristics

- Low-level machine instructions to implement operations.
- Control flow based on labels and conditional branches.
- Explicit locations (e.g. registers) for values and intermediate results.
- Explicit memory management (e.g., stack management for procedures).

Language Implementation

Ultimately, we want to execute programs on real hardware.

Two classic approaches:

- A **compiler** translates high-level language programs into a lower-level language (e.g. machine code).
- An **interpreter** is a fixed program that can read (the representation of) an arbitrary high-level program and execute it.

Very generally, compilers can generate code that runs faster than interpreted code, but interpreters are quicker and easier to write and maintain than compilers.

Interpreters are also easier to **understand**, which is why we will be using them for the toy languages in this course. But interpreters can also obscure important machine-level implementation issues, so we will also sometimes consider compiled code.

Stack Machines

A **stack machine** is a simple architecture based on a **stack** of operand values.

- All machine instructions pop their operands from the stack, and push their results back onto the stack
- This makes instructions very simple, becuase there's no need to specify operand locations.
- This architecture is often used in **abstract** machines, such as the Java, Python, or OCaml virtual machines. (Most **real** machines use register-based architectures instead.)
- Often compile from high-level language to stack machine **byte code** which is then interpreted (or perhaps compiled further to machine code).

STACK MACHINE EXAMPLE

Here's the instruction set for a very simple stack machine:

```
Instruction  Stack Before        Stack After

CONST i       s1 ... sn           i s1 ... sn
LOAD x        s1 ... sn           Vars[x] s1 ... sn
STORE x       s1 ... sn           s2 ... sn
PLUS          s1 s2 s3 ... sn     (s1+s2) s3 ... sn
MINUS         s1 s2 s3 ... sn     (s2-s1) s3 ... sn
```

Note that `STORE x` also sets `Vars[x] = s1`.

STACK MACHINE EXAMPLE (2)

And here's a stack machine program corresponding to the simple statement `c = 3 - a + (b - 7)`:

```
CONST 3
LOAD a
MINUS
LOAD b
CONST 7
MINUS
PLUS
STORE c
```

Is this code sequence unique?

This illustrates the expressiveness of high-level **expressions** compared to machine code.

PROGRAMMING LANGUAGE CLASSIFICATIONS

Programming paradigms

- Imperative (including object-oriented)
- Functional
- Logic
- Concurrent/Parallel
- Scripting

Programming Contexts

Programming "in the Small"
- Expressions
- Structured Control Flow
- Structured Data
- Types

Programming "in the Large"
- Modules and Separate Compilation
- Code Re-use; Polymorphism
- Object-oriented Programming
- Types

SOME THEMES IN LANGUAGE DESIGN

Expressiveness

- Vocabulary of operations, expressions, statements, etc.
- Support for abstraction, extensibility

Efficiency

- Mapping to the hardware

Program Correctness

- Types
- Reasoning about program behavior

Language Description and Documentation

For programmers, compiler-writers, and students . . .

Syntax (Easy)

What do programs look like?

- Grammars; BNF and Syntax Charts

Semantics (Hard)

What do programs do?

- Informal
- Formal: Operational, Denotational, Axiomatic

Learning about a Language

- Reference Manuals, User Guides, Texts and tutorials
- Experimentation

Syntax: Concrete & Abstract

- Language syntax describes the legal form and structure of programs
- Concrete syntax describes is what a program looks like on the page or screen
- Abstract syntax describes the essential contents of a program as it might be represented internally (e.g. by an interpreter or compiler)
- In this course, we won't worry much about concrete syntax, but it is worth some brief discussion
- (And we will need to choose some concrete syntax for the programs we interpret)
- Syntax is specified by a **grammar**.

Context-free Grammars

- Used for description, parsing, analysis, etc.
- Based on **recursive** definition of program structure.
- A grammar is defined by:
  - a set of **terminal** symbols (strings of characters)
  - a set **nonterminal** variables, which represent sets of terminals
  - a set of **production** rules that map nonterminals to strings of terminals and non-terminals
- The **language** defined by a grammar is the **set of strings of terminals** that can be derived by applying production rules, starting from a specified nonterminal.
- Grammars have rich theory with connections to automatic parser generation, push-down automata, etc.
- Many possible representations, including **BNF** (Backus-Naur Form), **EBNF** (Extended BNF), syntax charts, etc.

BNF

BNF was invented ca. 1960 and used in the formal description of Algol-60. It is just a particular notation for grammars, in which

- Nonterminals are represented by names inside angle brackets, e.g., <*program*>, <*expression*>, <*S*>.
- Terminals are represented by themselves, e.g., `WHILE`, `(`, `3`. The empty string is written as <*empty*>.

BNF Example...

<*program*>	::=	`BEGIN` <*statement-seq*> `END`
<*statement-seq*>	::=	<*statement*>
<*statement-seq*>	::=	<*statement*> `;` <*statement-seq*>
<*statement*>	::=	<*while-statement*>
<*statement*>	::=	<*for-statement*>
<*statement*>	::=	<*empty*>
<*while-statement*>	::=	`WHILE` <*expression*> `DO` <*statement-seq*> `END`
<*expression*>	::=	<*factor*>
<*expression*>	::=	<*factor*> `AND` <*factor*>
<*expression*>	::=	<*factor*> `OR` <*factor*>
<*factor*>	::=	`(` <*expression*> `)`
<*factor*>	::=	<*variable*>
<*for-statement*>	::=	. . .
<*variable*>	::=	. . .

EBNF

EBNF is (any) extension of BNF, usually with these features:

• A vertical bar, |, represents a choice,

• Parentheses, (and), represent grouping,

• Square brackets, [and], represent an optional construct,

• Curly braces, { and }, represent zero or more repetitions,

• Nonterminals begin with upper-case letters.

• Non-alphabetic terminal symbols are quoted, at least when necessary to avoid confusion with the meta-symbols above.

EBNF EXAMPLE

Program	::=	`BEGIN` *Statement-seq* `END`
Statement-seq	::=	*Statement* ['`;`' *Statement-seq*]
Statement	::=	[*While-statement* | *For-statement*]
While-statement	::=	`WHILE` *Expression* `DO` *Statement-seq* `END`
Expression	::=	*Factor* { (`AND` | `OR`) *Factor* }
Factor	::=	'`(`' *Expression* '`)`' | *Variable*
For-statement	::=	. . .
Variable	::=	. . .

SYNTAX ANALYSIS (PARSING)

Parser **recognizes** syntactically legal programs (as defined by a grammar) and **rejects** illegal ones.

• Successful parse also captures **hierarchical** structure of programs (expressions, blocks, etc.).

• Convenient representation for further semantic checking (e.g., typechecking) and for code generation.

• Failed parse provides error feedback to the user indicating where and why the input was illegal.

Any context-free language can be parsed by a computer program, but only **some** can be parsed **efficiently**. Modern programming languages can usually be parsed efficiently.

Lexical Analysis

Programming language grammars usually take simple **tokens** rather than characters as terminals. Converting raw program text into token stream is job of the **lexical analyzer**, which

- Detects and identifies keywords and identifiers.
- Converts multi-character symbols into single tokens.
- Handles numeric and string literals.
- Removes whitespace and comments.

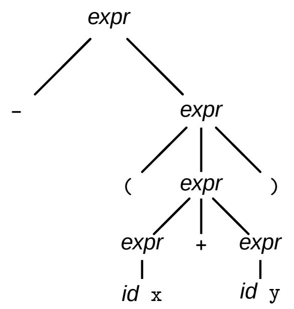
Parse Trees

Graphical representation of a derivation.

Given this grammar:

$$\textit{expr} \rightarrow \textit{expr} \texttt{ + } \textit{expr} \mid \textit{expr} \texttt{ * } \textit{expr} \mid \texttt{(}\textit{expr}\texttt{)} \mid \texttt{-}\textit{expr} \mid \textit{id}$$

Example tree for derivation of sentence `-(x + y)` :



Each application of a production corresponds to an **internal** node, labeled with a **non-terminal**.

Leaves are labeled with **terminals**, which can have **attributes** (in this case the specific identifier name).

The derived sentence is found by reading leaves (or "fringe") left-to-right.

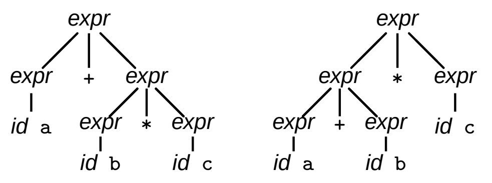
Ambiguity

A given **sentence** in $L(G)$ can have more than one parse tree. Grammars G for which this is true are called **ambiguous**.

Example: given the grammar on the last slide, the sentence

`a + b * c`

has two parse trees:



We may think of the left tree as being the "correct" one, but nothing in the grammar says this.

To avoid the problems of ambiguity, we can:

- Rewrite grammar; or
- Use "disambiguating rules" when we implement parser for grammar.

Ambiguity in Arithmetic Expressions

To disambiguate a grammar like

$$E \rightarrow E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E \mid E \texttt{ / } E \mid \texttt{(}E\texttt{)} \mid \texttt{id}$$

we need to make choices about the desired order of operations.

For any expression of the form `X` op_1 `Y` op_2 `Z` we must define:

- **Precedence** - which operation (op_1 or op_2) is done first?
- **Associativity** - if op_1 and op_2 have the same precedence, then does `Y` "associate" with the operator on the left or on the right?

In other words, we need rules to tell us whether the expression is equivalent to `(X` op_1 `Y)` op_2 `Z` or to `X` op_1 `(Y` op_2 `Z)`.

The "usual" rules (based on common usage in written math) give `*` and `/` higher precedence than `+` and `-`, and make all the operators left-associative.

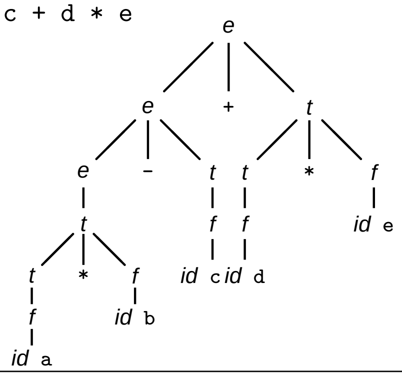
So, for example, `a - b - c * d` is equivalent to `(a - b) - (c * d)`. But this is a matter of **choice** when defining the language.

Rewriting Arithmetic Grammars

One way to enforce precedence/associativity is to build them into the grammar using extra non-terminals, e.g.:

$$\begin{array}{lll} \textit{factor} & \rightarrow & (\textit{expr}) \mid \texttt{id} \\ \textit{term} & \rightarrow & \textit{term} * \textit{factor} \mid \textit{term} \; / \; \textit{factor} \mid \textit{factor} \\ \textit{expr} & \rightarrow & \textit{expr} + \textit{term} \mid \textit{expr} - \textit{term} \mid \textit{term} \end{array}$$

Example: `a * b - c + d * e`



Limitations of Context-free Grammars

Context-free grammars are very useful for describing the structure of programming languages and identifying legal programs.

But there are many useful characteristics of legal programs that **cannot** be captured in a grammar (no matter how clever we are).

For example, in many programming languages, every variable in a legal program must be declared before it is used. But this property cannot be captured in a grammar.

So checking legality of programs typically requires more than syntax analysis. Most compilers use a secondary "semantic" analysis phase to check non-syntactic properties, such as type-correctness. Of course, sometimes illegal programs cannot be detected until runtime.

Parse Trees vs. Abstract Syntax Trees

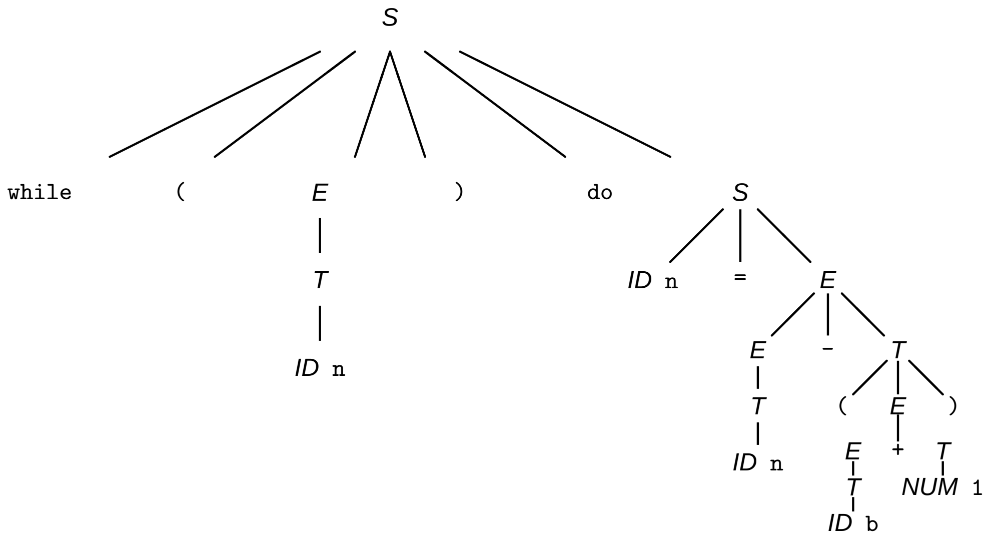
Parse trees reflect details of the **concrete** syntax of a program, which is typically designed for easy parsing.

For processing a language, we usually want a **simpler**, more **abstract** view of the program. (No firm rules about AST design: matter of taste, convenience.)

Simple concrete grammar:

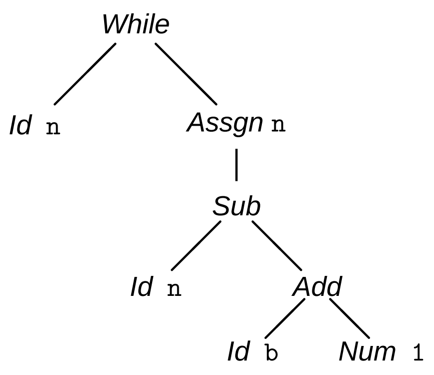
$$\begin{array}{l} S \rightarrow \texttt{while} \; \texttt{'('} \; E \; \texttt{')'} \; \texttt{do} \; S \mid ID \; \texttt{'='} \; E \\ E \rightarrow E \; \texttt{'+'} \; T \mid E \; \texttt{'-'} \; T \mid T \\ T \rightarrow ID \mid NUM \mid \texttt{'('} \; E \; \texttt{')'} \end{array}$$

`while (n) do n = n - (b + 1)`



Parse Trees vs. Abstract Syntax Trees (2)

Possible abstract syntax tree for `while (n) do n = n - (b + 1)`



Note that tree nodes may have **attributes** (such as the name of an *Id*) and/or **sub-trees**.

Tree Grammars

AST's obey a **tree grammar**. Rules have form

label : kind $\rightarrow$ *(attr*$_1$ *. . . attr*$_m$*) kind*$_1$ *. . . kind*$_n$

where the LHS classifies the possible node **labels** into **kind**s, and the RHS describes the label's atomic **attributes** (if any, in parentheses) and the kinds of its subtrees (if any).

Example:

While : Stmt $\rightarrow$ *Exp Stmt*
Assgn : Stmt $\rightarrow$ *(string) Exp*
Add : Exp $\rightarrow$ *Exp Exp*
Sub : Exp $\rightarrow$ *Exp Exp*
Id : Exp $\rightarrow$ *(string)*
Num : Exp $\rightarrow$ *(int)*

Abstract Syntax Captures the Essence

Concrete syntax is important for usability, but fundamentally superficial. The same abstract syntax can be used to represent many different concrete syntaxes.

Examples:

- C-like:

```
while (n) do n = n - (b + 1);
```

- Fortran-like:

```
do while(n .NE. 0)
        n = n - (b + 1)
end do
```

Concrete Syntax Examples (2)

- COBOL-like:

```
      PERFORM 100-LOOP-BODY
        WITH TEST BEFORE
        WHILE N IS NOT EQUAL TO 0
100_LOOP-BODY.
      ADD B TO 1 GIVING T
      SUBTRACT T FROM N GIVING N
```

- Use Chinese keywords in place of `while` and `do`.
- Use a graphical notation.

AST's in Java

AST's have recursive structure and irregular shape and size, so it makes sense to store them as **heap** data structures using one record for each tree node.

In Java, heap records are **objects**. We define **classes** corresponding to the various kinds and a subclass for each label, e.g.

```
abstract class Stmt { }
class While extends Stmt {
  Exp test;  Stmt body;
}
class Assgn extends Stmt {
  String lhs; Exp rhs;
}
abstract class Exp { }
class Add extends Exp {
  Exp left; Exp right;
}
class Num extends Exp {
  int value;
}
```

Nodes are created by constructor calls, e.g., `Exp e = new Num(42)`.

AST's in Python

In Python, heap records are again objects, and we define classes corresponding to the various labels, e.g.

```
class While:
    def __init__(self,test,body):
        self.test = test
        self.body = body
class Asgn:
    def __init__(self,lhs,rhs):
        self.lhs = lhs
        self.rhs = hrs
class Add:
    def __init__(self,left,right):
        self.left = left
        self.right = right
class Num:
    def __init__(self,num):
        self.num = num
```

Nodes are again created by invoking constructors, e.g. `e = Num(42)`. But note that the type relationships among the classes and fields are lost, since Python doesn't have static types.

AST's in OCaml

In Ocaml, we can use **algebraic datatypes** to define a suitable set of variants, e.g.:

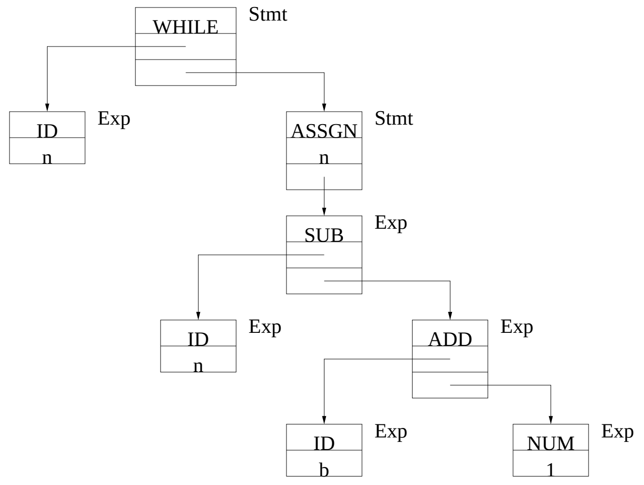
```
type stmt = While of exp * stmt
          | Assgn of string * exp
          | ...
and exp = Add of exp * exp
        | ...
        | Num of int
```

These declarations also define constructors, so we can say, e.g., `e:exp = Num(42)`.

Heap Structure

All these approaches generate roughly the **same** heap structures, e.g. for

`while (n) do n = n - (b + 1)`



External Representation of ASTs

Although ASTs are designed as an **internal** program representation, it can be useful to give them an **external** form too that can be read or written by other programs or by humans.

Any external representation of ASTs must accurately reflect the internal tree structure as well as the "fringe" of the tree. Can't use tree grammar to parse, since it is typically ambiguous!

One approach (deriving from the programming language LISP) is to use **parenthesized prefix notation** to represent trees.

Each node in the tree is represented by the expression

$(\quad \textit{label} \;\; \textit{attr}_1 \;\ldots\;\; \textit{attr}_m \;\; \textit{child}_1 \;\; \textit{child}_n \;\;)$

where *label* is the node label, the *attr*$_i$ are the label's attributes (if any), and the *child*$_i$ are the labels sub-trees (if any), each of which is itself a node expression. To make things more readable, we might use abbreviations for common labels, e.g., + for `Add`. We may also represent simple leaf nodes by their bare attributes, e.g., use `3` for `(Num 3)`, as long as no confusion can arise.

External AST Example

So the representation of our AST example could be

```
(While (Id n)
       (Assgn n (- (Id n)
                   (+ (Id b)
                      (Num 1)))))
```

where the indentation is optional, but makes the representation easier for humans to read. The BNF for this syntax can be given as:

```
<stmt> ::= (<While> <expr> <stmt>)
       |   (<Assgn> <name> <expr>)

<expr> ::= (+ <expr> <expr>)
       |   (- <expr> <expr>)
       |   <name>
       |   <int>
```

Concrete and abstract syntax are isomorphic.

Parsing Paranthesized Prefix Notation

Note that this BNF description contains essentially the same information as our AST internal datatype declarations (except for the descriptive field names).

Parsing this representation is easy! Why?

- Everything is either an atom (keyword, symbol, numeric literal, etc.) or a parenthesized list of atoms
- Each node in the AST corresponds to a list or atom.
- The first item in each list is always a symbol that identifies the node type and implies the kind and number of the remaining things in the list.
- The lexical analyzer just needs to identify the atoms and list delimiters.