

Lecture 2: Compilers, Theory, & PL's

CSC 131
Fall, 2012

Kim Bruce

TA Hours

- Pomona:
 - Richard: Wednesday 8 to 10 p.m in main lab
- HMC Hot Air balloon lab: Beckman B 105:
 - Bridgette: Wednesday 8-10 p.m.
 - Jake: Thursday 8:30 - 10:30 p.m.

First Homework
Due Thursday!

Finish Grace

Sample Programs

- ComplexNumbers

Avoid Hoare's “Billion Dollar Mistake”

- No built-in **null**
- Accessing uninitialized variable is error
- Replace **null** by:
 - sentinel objects, or
 - error actions

6


Sentinel Objects

A real object, tailored for the situation, *e.g.*:

```
def emptyList = object {  
  method asString {"<emptyList>"}  
  method do(action) {}  
  method map(function) {self}  
  method size {0}  
}
```

list

*name for object
being defined*



Sentinel Objects

Simplifies code, eliminates testing for **null**

```
class aList.cons(value, tailList) {  
  method asString {"({head}:{tail})"}  
  method head {value}  
  method tail {tailList} no conditional code  
  method do(action) {  
    action.apply(head)  
    tail.do(action)  
  }  
  method map(function) {  
    aList.cons(function.apply(head),  
              tail.map(function))  
  }  
  method size {1 + tail.size}  
}
```

Error Actions

- Grace encourages the use of blocks to specify error actions or default values:

```
var x := table.at(key)ifAbsent{  
    return unknown(key)  
}
```

9

Grace Details

- No parens needed w/parameterless methods
- No parens if parameter bounded by "" or {}
- Must insert parens for most precedence
- Use blocks for code evaluated variable number of times
- import "file" as libname
 - provides usage of libname as an object w/all defs from library.
 - See mgcollections use

Other Features ...

- Pattern matching (like Haskell/ML)
- Dialects for easy customization

Theory Matters!

Theory, Dynamic Execution, & Static Checking

- Program, when started w/ input can:
 - Terminate normally.
 - Terminate w/ error message
 - Run forever
- Effect of program:
 - partial function $f: \text{string} \rightarrow \text{string} \cup \{\text{error}\}$
 - Ex: $g(x) = \text{if odd}(x) \text{ then } 1 \text{ else } g(x-1) + g(x-2)$

Computable Functions

- f is computable iff exists program computing it.
- Church-Turing thesis:
 - Computability independent of particular formal model
 - Church-Kleene used lambda calculus
 - Turing used Turing machines
 - Godel-Kleene used partial recursive functions
 - ...

Halting Problem

- Is there is a program that will determine for any other program whether or not it will halt?
- *More precisely:*
 - Is there a program that when provided with another program and its input, will determine correctly 100% of the time whether or not the given program will halt on its input? (In particular, this program always halts telling me yes or no.)

Halting Problem

- Proved undecidable -- i.e., no algo to solve it.
- Other undecidable problems:
 - Will program eventually divide by 0?
 - Will program eventually dereference a null pointer?
 - Will program touch a particular piece of memory?
 - Will program ever print out 0?

Totally Ridiculous!

- Could we have a language in which all programs halt, and yet get all "total" computable functions?
- Yes -- take only total functions from favorite language -- make those only legal programs.
- Def: Say a language is *good* iff there is a program (e.g., parser or type-checker) that effectively determines for each candidate program whether or not it is legal.

Abandon All Hope

- **Theorem:** Let L be a "good" language in which all programs halt for all input. Then there is an effectively computable total function that is not expressible in the language.
- No hope of finding a language with a type-checker which identifies all and only total effectively computable programs.

Virtual Machines

Abstraction

- *Dijkstra:* Originally we were obligated to write programs so that a computer could execute them. Now we write the programs and the computer has the obligation to understand and execute them.
- Progress in PL design marked by increasing support for abstraction.
- What are data types and how to construct new?
- What are ops and how do we construct new?

Creating an Illusion



Pure Translators

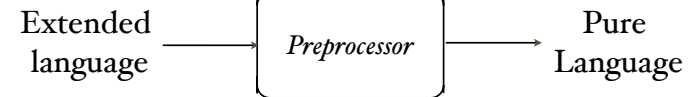
Assembler



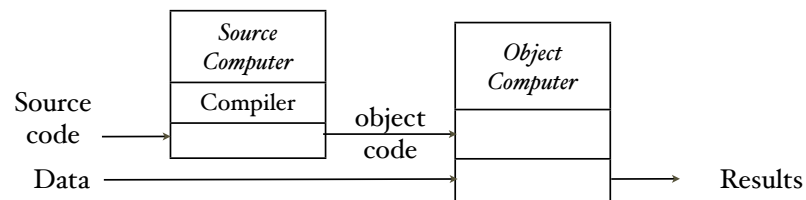
Compiler



Preprocessor

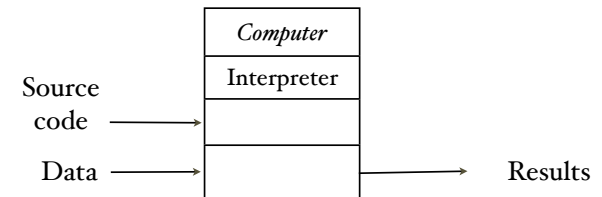


Execution w/Compiler



Virtual Machine

Interpreter



Virtual Machine

Virtual Machine of Language

- Virtual machine defined by language
- Machine language is set of instructions supported by language
- Layers of VM's
 - bare Intel Chip \Rightarrow Mac OS \Rightarrow Java \Rightarrow Application
- Describe language by VM it defines

VM of Language

- Problems:
 - Different implementors may have different conceptions of virtual machine
 - Different computers may provide different facilities and operations
 - Implementors may make different choices as to how to simulate elements of virtual computer
- May lead to different semantics, even on same computer.

VM Problems

- How ensure different implementations give same semantics?
- Sometimes VM's are explicit
 - Pascal P-code & P-machine
 - Modula-2 M-code
 - Java VM & JVM

More Detail: Interpreters

- Simulate virtual machine:

REPEAT

 Get next statement

 Determine action(s) to be executed

 Call routine to perform action

UNTIL done

More Detail: Compiler

- Translate from one VM to another
 - Translate all units of program into object code
 - Link into single relocatable machine code
 - Load into memory
 - Begin execution

Compiler vs Interpreter

<i>Compiler</i>	<i>Interpreter</i>
Only translate each statement once	Translate only if executed.
Speed of execution	Error messages tied to source. More supportive environment. <i>No longer as true</i>
Only object code in memory when executing. May take more space because of expansion	Must have interpreter in memory while executing (though source may be more compact)

Lack of Purity

- Rarely pure compiler or interpreter
 - Typically compile source into form easier to interpret.
 - Ex. Remove white space & comments, build symbol table, or parse each line and store in more compact form (e.g. tree)
- Java originally hybrid
 - Compile into JVMIL and then interpreted
 - Now compile to JVMIL & use just-in-time compiler on JVMIL

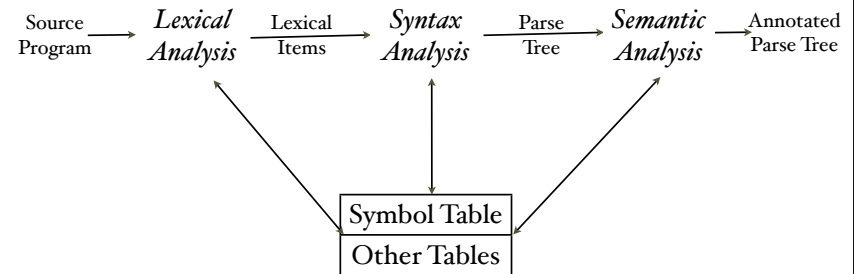
Compiler Structure

- Analysis:
 - Break into lexical items, build parse tree, annotate parse tree (e.g. via type checking)
- Synthesis:
 - generate simple intermediate code, optimization (look at instructions in context), code generation, linking and loading.

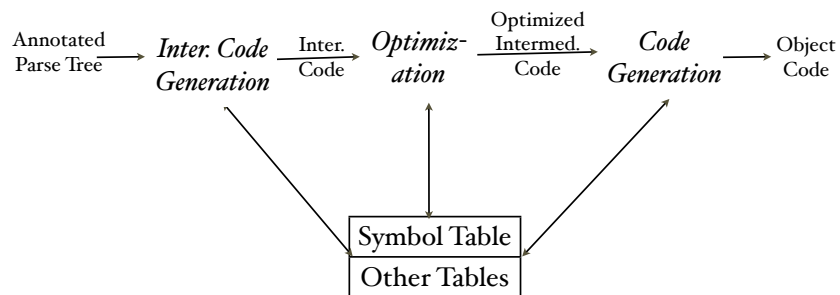
Symbol Table

- Symbol table:
 - Contains all id names,
 - kind of id (vble, array name, proc name, formal parameter),
 - type of value,
 - where visible, etc.

Analysis



Synthesis



Portable Compilers

- Separate front and back-ends that share same intermediate code (GNU compilers)
 - Write single front end for each language
 - Write single back end for each operating system - architecture combination.
 - Mix and match to build complete compilers
- JVM starting to play that role too

Formal Syntax

Formal Syntax

- Syntax:
 - Readable, writable, easy to translate, unambiguous, ...
- Formal Grammars:
 - Backus & Naur, Chomsky
 - First used in ALGOL 60 Report - formal description
 - Generative description of language.
- Language is set of strings. (E.g. all legal C++ programs)

Example

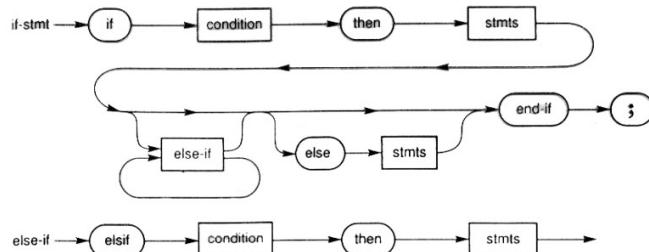
```
<exp>    ⇒ <term> | <exp> <addop> <term>
<term>   ⇒ <factor> | <term> <multop> <factor>
<factor> ⇒ <id> | <literal> | (<exp>)
<id>     ⇒ a | b | c | d
<literal> ⇒ <digit> | <digit> <literal>
<digit>  ⇒ 0 | 1 | 2 | ... | 9
<addop>  ⇒ + | - | or
<multop> ⇒ * | / | div | mod | and
```

Extended BNF

- Extended BNF handy:
 - item enclosed in square brackets is optional
 - <conditional> ⇒ if <expression> then <statement>
[else <statement>]
 - item enclosed in curly brackets means zero or more occurrences
 - <literal> ⇒ <digit> { <digit> }

Syntax Diagrams

- Syntax diagrams - alternative to BNF.
 - Syntax diagrams are never directly recursive, use "loops" instead.



Ambiguity

```
<statement> ⇒ <unconditional> | <conditional>  
  
<unconditional> ⇒ <assignment> | <for loop> |  
                    "{" { <statement> } "  
  
<conditional> ⇒ if (<expression>) <statement> |  
                if (<expression>) <statement>  
                    else <statement>
```

- *How do you parse:*

```
if (exp1)  
    if (exp2)  
        stat1;  
    else  
        stat2;
```

Resolving Ambiguity

- Pascal, C, C++, and Java rule:
 - else attached to nearest then.
 - to get other form, use { ... }
- Modula-2 and Algol 68
 - No "{", only "}" (except write as "end")
- Not a problem in LISP/Racket/ML/Haskell conditional *expressions*
- Ambiguity in general is undecidable

Chomsky Hierarchy

- Chomsky developed mathematical theory of programming languages:
 - type 0: recursively enumerable
 - type 1: context-sensitive
 - type 2: context-free
 - type 3: regular
- BNF = context-free, recognized by pda

Beyond Context-Free

- Not all aspects of PLs are context-free
 - Declare before use, goto target exist
- Formal description of syntax allows:
 - programmer to generate syntactically correct programs
 - parser to recognize syntactically correct programs
- Parser-generators: LEX, YACC, ANTLR, etc.
 - formal spec of syntax allows automatic creation of recognizers

Specifying Semantics: Lambda Calculus

Defining Functions

- In math and LISP:
 - $f(n) = n * n$
 - (define (f n) (* n n))
 - (define f (lambda (n) (* n n)))
- In lambda calculus
 - $\lambda n. n * n$
 - $((\lambda n. n * n) 12) \Rightarrow 144$

Pure Lambda Calculus

- Terms of pure lambda calculus
 - $M ::= v \mid (M M) \mid \lambda v. M$
 - Impure versions add constants, but not necessary!
 - Turing-complete
- Left associative: $M N P = (M N) P$.
- Computation based on substituting actual parameter for formal parameters

Free Variables

- Substitution easy to mess up!
- Def: If M is a term, then $FV(M)$, the collection of free variables of M , is defined as follows:
 - $FV(x) = \{x\}$
 - $FV(M N) = FV(M) \cup FV(N)$
 - $FV(\lambda v. M) = FV(M) - \{v\}$