# Table of Contents

## 1. Programming Languages

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sources</td>
<td>1</td>
</tr>
<tr>
<td>Schedule</td>
<td>2</td>
</tr>
<tr>
<td>What is a Programming Language?</td>
<td>3</td>
</tr>
<tr>
<td>What is a Programming Language? (II)</td>
<td>4</td>
</tr>
<tr>
<td>Themes Addressed in this Course</td>
<td>5</td>
</tr>
<tr>
<td>Themes Addressed in this Course ...</td>
<td>6</td>
</tr>
<tr>
<td>Generations of Programming Languages</td>
<td>7</td>
</tr>
<tr>
<td>How do Programming Languages Differ?</td>
<td>8</td>
</tr>
<tr>
<td>Programming Paradigms</td>
<td>9</td>
</tr>
<tr>
<td>Compilers and Interpreters</td>
<td>10</td>
</tr>
<tr>
<td>A Brief Chronology</td>
<td>11</td>
</tr>
<tr>
<td>Fortran</td>
<td>12</td>
</tr>
<tr>
<td>Fortran ...</td>
<td>13</td>
</tr>
<tr>
<td>“Hello World” in FORTRAN</td>
<td>14</td>
</tr>
<tr>
<td>ALGOL 60</td>
<td>15</td>
</tr>
<tr>
<td>ALGOL 60 ...</td>
<td>16</td>
</tr>
<tr>
<td>“Hello World” in BEALGOL</td>
<td>17</td>
</tr>
<tr>
<td>COBOL</td>
<td>18</td>
</tr>
<tr>
<td>“Hello World” in COBOL</td>
<td>19</td>
</tr>
<tr>
<td>4GLs</td>
<td>20</td>
</tr>
<tr>
<td>“Hello World” in RPG</td>
<td>21</td>
</tr>
<tr>
<td>“Hello World” in SQL</td>
<td>22</td>
</tr>
<tr>
<td>PL/1</td>
<td>23</td>
</tr>
<tr>
<td>“Hello World” in PL/1</td>
<td>24</td>
</tr>
<tr>
<td>Interactive Languages</td>
<td>25</td>
</tr>
<tr>
<td>Interactive Languages ...</td>
<td>26</td>
</tr>
<tr>
<td>Special-Purpose Languages</td>
<td>27</td>
</tr>
<tr>
<td>Special-Purpose Languages ...</td>
<td>28</td>
</tr>
<tr>
<td>Functional Languages</td>
<td>29</td>
</tr>
<tr>
<td>“Hello World” in Functional Languages</td>
<td>30</td>
</tr>
<tr>
<td>Prolog</td>
<td>31</td>
</tr>
<tr>
<td>“Hello World” in Prolog</td>
<td>32</td>
</tr>
<tr>
<td>Object-Oriented Languages</td>
<td>33</td>
</tr>
<tr>
<td>Object-Oriented Languages ...</td>
<td>34</td>
</tr>
<tr>
<td>Scripting Languages</td>
<td>35</td>
</tr>
<tr>
<td>Scripting Languages ...</td>
<td>36</td>
</tr>
<tr>
<td>Scripting Languages ...</td>
<td>37</td>
</tr>
<tr>
<td>What you should know!</td>
<td>38</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>39</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>40</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>41</td>
</tr>
</tbody>
</table>

## 2. Systems Programming

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is C?</td>
<td>42</td>
</tr>
<tr>
<td>C Features</td>
<td>43</td>
</tr>
<tr>
<td>“Hello World” in C</td>
<td>44</td>
</tr>
<tr>
<td>Symbols</td>
<td>45</td>
</tr>
<tr>
<td>Keywords</td>
<td>46</td>
</tr>
<tr>
<td>Operators (same as Java)</td>
<td>47</td>
</tr>
<tr>
<td>C Storage Classes</td>
<td>48</td>
</tr>
<tr>
<td>Memory Layout</td>
<td>49</td>
</tr>
<tr>
<td>Where is memory?</td>
<td>50</td>
</tr>
<tr>
<td>Declarations and Definitions</td>
<td>51</td>
</tr>
<tr>
<td>Header files</td>
<td>52</td>
</tr>
<tr>
<td>Including header files</td>
<td>53</td>
</tr>
<tr>
<td>Makefiles</td>
<td>54</td>
</tr>
<tr>
<td>C Arrays</td>
<td>55</td>
</tr>
<tr>
<td>Pointers</td>
<td>56</td>
</tr>
<tr>
<td>Strings</td>
<td>57</td>
</tr>
<tr>
<td>Pointer manipulation</td>
<td>58</td>
</tr>
<tr>
<td>Function Pointers</td>
<td>59</td>
</tr>
<tr>
<td>PS — S2003</td>
<td>60</td>
</tr>
</tbody>
</table>
3. Multiparadigm Programming

Essential C++ Texts
What is C++?
C++ vs C
“Hello World” in C++
C++ Design Goals
C++ Features
Java and C++ — Similarities and Extensions
Java Simplifications
New Keywords
Comments
References
References vs Pointers
C++ Classes
Constructors and destructors
Automatic and dynamic destruction
Orthodox Canonical Form

Why OCF?
Example: A String Class
A Simple String.h
Default Constructors
Destructors
Copy Constructors
A few remarks ...
Other Constructors
Assignment Operators
A few more remarks ...
Implicit Conversion
Operator Overloading
Overloadable Operators
Friends
Friends ...
What are Templates?
Function Templates
Class Templates
Using Class Templates
Standard Template Library
An STL Line Reverser
What we didn’t have time for ...
What you should know!
Can you answer these questions?

4. Stack-based Programming

What is PostScript?
Postscript variants
Syntax
Semantics
Object types
The operand stack
Stack and arithmetic operators
# 5. Functional Programming

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>References</td>
<td>148</td>
</tr>
<tr>
<td>A Bit of History</td>
<td>149</td>
</tr>
<tr>
<td>A Bit of History</td>
<td>150</td>
</tr>
<tr>
<td>Programming without State</td>
<td>151</td>
</tr>
<tr>
<td>Pure Functional Programming Languages</td>
<td>152</td>
</tr>
<tr>
<td>Key features of pure functional languages</td>
<td>153</td>
</tr>
<tr>
<td>What is Haskell?</td>
<td>154</td>
</tr>
<tr>
<td>“Hello World” in Hugs</td>
<td>155</td>
</tr>
<tr>
<td>Referential Transparency</td>
<td>156</td>
</tr>
<tr>
<td>Evaluation of Expressions</td>
<td>157</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tail Recursion</td>
<td>159</td>
</tr>
<tr>
<td>Tail Recursion ...</td>
<td>160</td>
</tr>
<tr>
<td>Equational Reasoning</td>
<td>161</td>
</tr>
<tr>
<td>Equational Reasoning ...</td>
<td>162</td>
</tr>
<tr>
<td>Pattern Matching</td>
<td>163</td>
</tr>
<tr>
<td>Lists</td>
<td>164</td>
</tr>
<tr>
<td>Using Lists</td>
<td>165</td>
</tr>
<tr>
<td>Higher Order Functions</td>
<td>166</td>
</tr>
<tr>
<td>Anonymous functions</td>
<td>167</td>
</tr>
<tr>
<td>Curried functions</td>
<td>168</td>
</tr>
<tr>
<td>Understanding Curried functions</td>
<td>169</td>
</tr>
<tr>
<td>Using Curried functions</td>
<td>170</td>
</tr>
<tr>
<td>Currying</td>
<td>171</td>
</tr>
<tr>
<td>Multiple Recursion</td>
<td>172</td>
</tr>
<tr>
<td>Lazy Evaluation</td>
<td>173</td>
</tr>
<tr>
<td>Lazy Lists</td>
<td>174</td>
</tr>
<tr>
<td>Programming lazy lists</td>
<td>175</td>
</tr>
<tr>
<td>Declarative Programming Style</td>
<td>176</td>
</tr>
<tr>
<td>What you should know!</td>
<td>177</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>178</td>
</tr>
</tbody>
</table>

## 6. Type Systems

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>References</td>
<td>179</td>
</tr>
<tr>
<td>What is a Type?</td>
<td>180</td>
</tr>
<tr>
<td>What is a Type?</td>
<td>181</td>
</tr>
<tr>
<td>Static and Dynamic Types</td>
<td>182</td>
</tr>
<tr>
<td>Static and Dynamic Typing</td>
<td>183</td>
</tr>
<tr>
<td>Kinds of Types</td>
<td>184</td>
</tr>
<tr>
<td>Type Completeness</td>
<td>185</td>
</tr>
<tr>
<td>Function Types</td>
<td>186</td>
</tr>
<tr>
<td>List Types</td>
<td>187</td>
</tr>
<tr>
<td>Tuple Types</td>
<td>188</td>
</tr>
<tr>
<td>User Data Types</td>
<td>189</td>
</tr>
<tr>
<td>References</td>
<td>190</td>
</tr>
<tr>
<td>7. Introduction to the Lambda Calculus</td>
<td>8. Fixed Points</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>References</td>
<td>Recursion</td>
</tr>
<tr>
<td>What is Computable?</td>
<td>Recursive functions as fixed points</td>
</tr>
<tr>
<td>Church’s Thesis</td>
<td>Fixed Points</td>
</tr>
<tr>
<td>Uncomputability</td>
<td>Fixed Point Theorem</td>
</tr>
<tr>
<td>What is a Function? (I)</td>
<td>How does Y work?</td>
</tr>
<tr>
<td>What is a Function? (II)</td>
<td>Using the Y Combinator</td>
</tr>
<tr>
<td>What is the Lambda Calculus?</td>
<td>Recursive Functions are Fixed Points</td>
</tr>
<tr>
<td>Parsing Lambda Expressions</td>
<td>Unfolding Recursive Lambda Expressions</td>
</tr>
<tr>
<td>What is the Lambda Calculus? ...</td>
<td>The Typed Lambda Calculus</td>
</tr>
<tr>
<td>Beta Reduction</td>
<td>The Polymorphic Lambda Calculus</td>
</tr>
<tr>
<td>Lambda expressions in Haskell</td>
<td>Hindley-Milner Polymorphism</td>
</tr>
<tr>
<td>Lambdas are anonymous functions</td>
<td>Polymorphism and self application</td>
</tr>
<tr>
<td>A Few Examples</td>
<td>Other Calculi</td>
</tr>
</tbody>
</table>
Can you answer these questions? 319

### 11. Applications of Logic Programming

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Solving a puzzle</td>
<td>321</td>
</tr>
<tr>
<td>A non-solution:</td>
<td>322</td>
</tr>
<tr>
<td>A non-solution ...</td>
<td>323</td>
</tr>
<tr>
<td>A first solution</td>
<td>324</td>
</tr>
<tr>
<td>A first solution ...</td>
<td>325</td>
</tr>
<tr>
<td>A second (non-)solution</td>
<td>326</td>
</tr>
<tr>
<td>A second (non-)solution ...</td>
<td>327</td>
</tr>
<tr>
<td>A third solution</td>
<td>328</td>
</tr>
<tr>
<td>A third solution ...</td>
<td>329</td>
</tr>
<tr>
<td>A third solution ...</td>
<td>330</td>
</tr>
<tr>
<td>A fourth solution</td>
<td>331</td>
</tr>
<tr>
<td>A fourth solution ...</td>
<td>332</td>
</tr>
<tr>
<td>A fourth solution ...</td>
<td>333</td>
</tr>
<tr>
<td>II. Symbolic Interpretation</td>
<td>334</td>
</tr>
<tr>
<td>Goal-directed interpretation</td>
<td>335</td>
</tr>
<tr>
<td>Definite Clause Grammars</td>
<td>336</td>
</tr>
<tr>
<td>Definite Clause Grammars ...</td>
<td>337</td>
</tr>
<tr>
<td>Example</td>
<td>338</td>
</tr>
<tr>
<td>How to use this?</td>
<td>339</td>
</tr>
<tr>
<td>How does it work?</td>
<td>340</td>
</tr>
<tr>
<td>Lexical analysis</td>
<td>341</td>
</tr>
<tr>
<td>Recognizing Tokens</td>
<td>342</td>
</tr>
<tr>
<td>Recognizing Numbers</td>
<td>343</td>
</tr>
<tr>
<td>Concrete Grammar</td>
<td>344</td>
</tr>
<tr>
<td>Parsing with DCGs</td>
<td>345</td>
</tr>
<tr>
<td>Representing Programs as Parse Trees</td>
<td>346</td>
</tr>
<tr>
<td>Testing</td>
<td>347</td>
</tr>
<tr>
<td>Interpretation as Proof</td>
<td>348</td>
</tr>
<tr>
<td>Building a Simple Interpreter</td>
<td>349</td>
</tr>
<tr>
<td>Running the Interpreter</td>
<td>350</td>
</tr>
<tr>
<td>Testing the interpreter</td>
<td>351</td>
</tr>
<tr>
<td>A top-level script</td>
<td>352</td>
</tr>
<tr>
<td>What you should know</td>
<td>353</td>
</tr>
<tr>
<td>Can you answer these questions?</td>
<td>354</td>
</tr>
</tbody>
</table>

### 12. TBA

### 13. Summary, Trends, Research ...

- C and C++ 357
- Functional Languages 358
- Lambda Calculus 359
- Type Systems 360
- Polymorphism 361
- Denotational Semantics 362
- Logic Programming 363
- Object-Oriented Languages 364
- Scripting Languages 365
# 1. Programming Languages

| Lecturer:     | Prof. Oscar Nierstrasz  
|              | Schützenmattstr. 14/103 |
| Tel:         | 031 631 4618             |
| Email:       | Oscar.Nierstrasz@iam.unibe.ch |
| Assistants:  | Gabriela Arévalo, Marc-Philippe Horvath |
| WWW:         | www.iam.unibe.ch/~scg/Teaching/ |
Sources

Text:

Other Sources:
## Schedule

1. 03-25 Introduction
2. 04-01 Systems programming
3. 04-08 Multi-paradigm programming
4. 04-15 Stack-based programming
5. 04-22 Functional programming
6. 04-29 Type systems
7. 05-06 Lambda calculus
8. 05-13 Fixed points
9. 05-20 Programming language semantics
10. 05-27 Logic programming
11. 06-03 Applications of logic programming
12. 06-10 Piccola — A Small Composition Language
13. 06-17 Summary, Trends, Research
14. 06-24 Final exam
What is a Programming Language?

- A formal language for describing computation?
- A “user interface” to a computer?
- Syntax + semantics?
- Compiler, or interpreter, or translator?
- A tool to support a programming paradigm?

“A programming language is a notational system for describing computation in a machine-readable and human-readable form.”

— Louden
What is a Programming Language? (II)

The thesis of this course:

A programming language is a tool for developing executable models for a class of problem domains.
Themes Addressed in this Course

Paradigms
- What computational paradigms are supported by modern, high-level programming languages?
- How well do these paradigms match classes of programming problems?

Abstraction
- How do different languages abstract away from the low-level details of the underlying hardware implementation?
- How do different languages support the specification of software abstractions needed for a specific task?

...
Themes Addressed in this Course ...

Types
- How do type systems help in the construction of flexible, reliable software?

Semantics
- How can one formalize the meaning of a programming language?
- How can semantics aid in the implementation of a programming language?
Generations of Programming Languages

1GL: machine codes
2GL: symbolic assemblers
3GL: (machine independent) imperative languages (FORTRAN, Pascal, C ...)
4GL: domain specific application generators

Each generation is at a higher level of abstraction
How do Programming Languages Differ?

Common Constructs:
- basic data types (numbers, etc.); variables; expressions; statements; keywords; control constructs; procedures; comments; errors ...

Uncommon Constructs:
- type declarations; special types (strings, arrays, matrices, ...); sequential execution; concurrency constructs; packages/modules; objects; general functions; generics; modifiable state; ...
Programming Paradigms

A programming language is a problem-solving tool.

<table>
<thead>
<tr>
<th>Style</th>
<th>Program description</th>
<th>Good for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imperative style:</td>
<td>program = algorithms + data</td>
<td>decomposition</td>
</tr>
<tr>
<td>Functional style:</td>
<td>program = functions ◯ functions</td>
<td>reasoning</td>
</tr>
<tr>
<td>Logic programming style:</td>
<td>program = facts + rules</td>
<td>searching</td>
</tr>
<tr>
<td>Object-oriented style:</td>
<td>program = objects + messages</td>
<td>encapsulation</td>
</tr>
</tbody>
</table>

Other styles and paradigms: blackboard, pipes and filters, constraints, lists, ...
Compilers and Interpreters

Compilers and interpreters have similar front-ends, but have different back-ends:

Details will differ, but the general scheme remains the same ...
# A Brief Chronology

**Early 1950s** “order codes” (primitive assemblers)

<table>
<thead>
<tr>
<th>Year</th>
<th>Language</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1957</td>
<td>FORTRAN</td>
<td>the first <em>high-level</em> programming language (3GL is invented)</td>
</tr>
<tr>
<td>1958</td>
<td>ALGOL</td>
<td>the first <em>modern, imperative</em> language</td>
</tr>
<tr>
<td>1960</td>
<td>LISP, COBOL</td>
<td></td>
</tr>
<tr>
<td>1962</td>
<td>APL, SIMULA</td>
<td>the birth of <em>OOP</em> (SIMULA)</td>
</tr>
<tr>
<td>1964</td>
<td>BASIC, PL/I</td>
<td></td>
</tr>
<tr>
<td>1966</td>
<td>ISWIM</td>
<td>first modern <em>functional</em> language (a proposal)</td>
</tr>
<tr>
<td>1970</td>
<td>Prolog</td>
<td><em>logic</em> programming is born</td>
</tr>
<tr>
<td>1972</td>
<td>C</td>
<td><em>the</em> systems programming language</td>
</tr>
<tr>
<td>1975</td>
<td>Pascal, Scheme</td>
<td>two teaching languages</td>
</tr>
<tr>
<td>Year</td>
<td>Language</td>
<td>Event</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>-------</td>
</tr>
<tr>
<td>1978</td>
<td>CSP</td>
<td>Concurrency matures</td>
</tr>
<tr>
<td>1978</td>
<td>FP</td>
<td>Backus’ proposal</td>
</tr>
<tr>
<td>1983</td>
<td>Smalltalk-80, Ada</td>
<td>OOP is reinvented</td>
</tr>
<tr>
<td>1984</td>
<td>Standard ML</td>
<td>FP becomes mainstream (?)</td>
</tr>
<tr>
<td>1986</td>
<td>C++, Eiffel</td>
<td>OOP is reinvented (again)</td>
</tr>
<tr>
<td>1988</td>
<td>CLOS, Oberon, Mathematica</td>
<td></td>
</tr>
<tr>
<td>1990</td>
<td>Haskell</td>
<td>FP is reinvented</td>
</tr>
<tr>
<td>1995</td>
<td>Java</td>
<td>OOP is reinvented for the internet</td>
</tr>
</tbody>
</table>
Fortran

History
John Backus (1953) sought to write programs in *conventional mathematical notation*, and generate code comparable to good assembly programs.

- No language design effort (made it up as they went along)
- Most effort spent on code generation and optimization
- FORTRAN I released April 1957; working by April 1958
- Current standards are FORTRAN 77 and FORTRAN 90

...
Fortran ...

Innovations

- Symbolic notation for subroutines and functions
- Assignments to variables of complex expressions
- DO loops
- Comments
- Input/output formats
- Machine-independence

Successes

- Easy to learn; high level
- Promoted by IBM; addressed large user base (scientific computing)
"Hello World" in FORTRAN

PROGRAM HELLO
DO 10, I=1,10
PRINT *, 'Hello World'
10 CONTINUE
STOP
END

All examples from the ACM "Hello World" project:
www2.latech.edu/~acm/HelloWorld.shtml
ALGOL 60

History

- Committee of PL experts formed in 1955 to design universal, machine-independent, algorithmic language
- First version (ALGOL 58) never implemented; criticisms led to ALGOL 60

...
ALGOL 60 ...

Innovations

- **BNF** (Backus-Naur Form) introduced to define syntax (led to syntax-directed compilers)
- First *block-structured* language; variables with local scope
- *Structured* control statements
- *Recursive* procedures
- Variable size arrays

Successes

- Highly influenced design of other PLs but never displaced FORTRAN
"Hello World" in BEALGOL

BEGIN
FILE F (KIND=REMOTE);
EBCDIC ARRAY E [0:11];
REPLACE E BY "HELLO WORLD!";
WHILE TRUE DO
  BEGIN
    WRITE (F, *, E);
  END;
END.
COBOL

History
- Designed by committee of US computer manufacturers
- Targeted business applications
- Intended to be readable by managers (!)

Innovations
- Separate descriptions of environment, data, and processes

Successes
- Adopted as de facto standard by US DOD
- Stable standard for 25 years
- Still the most widely used PL for business applications (!)
"Hello World" in COBOL

000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.    HELLOWORLD.
000300 DATE-WRITTEN.  02/05/96        21:04.
000400* AUTHOR BRIAN COLLINS
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
001000 DATA DIVISION.
001100 FILE SECTION.
100000 PROCEDURE DIVISION.
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100400 DISPLAY " " LINE 1 POSITION 1 ERASE EOS.
100500 DISPLAY "HELLO, WORLD." LINE 15 POSITION 10.
100600 STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800 EXIT.
4GLs

“Problem-oriented” languages
- PLs for “non-programmers”
- Very High Level (VHL) languages for specific problem domains

Classes of 4GLs (no clear boundaries)
- Report Program Generator (RPG)
- Application generators
- Query languages
- Decision-support languages

Successes
- Highly popular, but generally ad hoc
“Hello World” in RPG

H
FSCREEN O F 80 80 CRT
C EXCPT
OSCREEN E 1
O 12 'HELLO WORLD!'
"Hello World" in SQL

CREATE TABLE HELLO (HELLO CHAR(12))
UPDATE HELLO
    SET HELLO = 'HELLO WORLD!'
SELECT * FROM HELLO
PL/1

History
- Designed by committee of IBM and users (early 1960s)
- Intended as (large) general-purpose language for broad classes of applications

Innovations
- Support for concurrency (but not synchronization)
- Exception-handling by on conditions

Successes
- Achieved both run-time efficiency and flexibility (at expense of complexity)
- First “complete” general purpose language
“Hello World” in PL/1

HELLO:   PROCEDURE OPTIONS (MAIN);

/* A PROGRAM TO OUTPUT HELLO WORLD */
FLAG = 0;

LOOP:   DO WHILE (FLAG = 0);
         PUT SKIP DATA('HELLO WORLD!');
      END LOOP;

END HELLO;
Interactive Languages

Made possible by advent of *time-sharing* systems (early 1960s through mid 1970s).

**BASIC**

- Developed at Dartmouth College in mid 1960s
- Minimal; easy to learn
- Incorporated basic O/S commands (NEW, LIST, DELETE, RUN, SAVE)

```
10 print "Hello World!"
20 goto 10
```

...
Interactive Languages ...

APL

❑ Developed by Ken Iverson for *concise* description of numerical algorithms
❑ Large, non-standard alphabet (52 characters in addition to alphanumerics)
❑ Primitive objects are *arrays* (lists, tables or matrices)
❑ *Operator-driven* (power comes from composing array operators)
❑ No operator precedence (statements parsed right to left)

'HELLO WORLD'
Special-Purpose Languages

SNOBOL

- First successful *string manipulation* language
- Influenced design of text editors more than other PLs
- String operations: *pattern-matching* and *substitution*
- Arrays and associative arrays (tables)
- Variable-length strings

```
OUTPUT = 'Hello World!'
END
...
```
Special-Purpose Languages …

Lisp

- Performs computations on symbolic expressions
- \textit{Symbolic expressions} are represented as \textit{lists}
- Small set of constructor-selector operations to create and manipulate lists
- \textit{Recursive} rather than iterative control
- No distinction between \textit{data} and \textit{programs}
- First PL to implement storage management by \textit{garbage collection}
- Affinity with \textit{lambda calculus}

(DEFUN HELLO-WORLD ()
    (PRINT (LIST 'HELLO 'WORLD)))
Functional Languages

ISWIM (If you See What I Mean)
- Peter Landin (1966) — paper proposal

FP
- John Backus (1978) — Turing award lecture

ML
- Edinburgh
- initially designed as meta-language for theorem proving
- Hindley-Milner type inference
- “non-pure” functional language (with assignments/side effects)

Miranda, Haskell
- “pure” functional languages with “lazy evaluation”
“Hello World” in Functional Languages

SML

    print("hello world!\n");

Haskell

    hello() = print "Hello World"
Prolog

History
- Originated at U. Marseilles (early 1970s), and compilers developed at Marseilles and Edinburgh (mid to late 1970s)

Innovations
- Theorem proving paradigm
- Programs as sets of clauses: facts, rules and questions
- Computation by “unification”

Successes
- Prototypical logic programming language
- Used in Japanese Fifth Generation Initiative
"Hello World" in Prolog

% HELLO WORLD. Works with Sbp (prolog)

hello :-
printstring("HELLO WORLD!!!!").

printstring([]).
printstring([H|T]) :- put(H), printstring(T).
Object-Oriented Languages

History

❑ **Simula** was developed by Nygaard and Dahl (early 1960s) in Oslo as a language for simulation programming, by adding *classes* and *inheritance* to ALGOL 60

```plaintext
Begin
  while 1 = 1 do begin
    outtext ("Hello World!");
    outimage;
  end;
End;
```

❑ **Smalltalk** was developed by Xerox PARC (early 1970s) to drive graphic workstations

```plaintext
Transcript show:'Hello World';cr
```

...
Object-Oriented Languages …

Innovations
- *Encapsulation* of data and operations (contrast ADTs)
- *Inheritance* to share behaviour and interfaces

Successes
- Smalltalk project pioneered OO *user interfaces*
- Large commercial impact since mid 1980s
- Countless new languages: C++, Objective C, Eiffel, Beta, Oberon, Self, Perl 5, Python, Java, Ada 95 …
Scripting Languages

History

- Countless “shell languages” and “command languages” for operating systems and configurable applications
- **Unix shell** (ca. 1971) developed as user shell and scripting tool
  
  ```plaintext
  echo "Hello, World!"
  ```

- **HyperTalk** (1987) was developed at Apple to script HyperCard stacks on OpenStack
  
  ```plaintext
  show message box
  put "Hello World!" into message box
  end OpenStack
  ```

...
Scripting Languages ...

- **TCL** (1990) developed as embedding language and scripting language for X windows applications (via Tk)
  
  ```
  puts "Hello World 
  ```

- **Perl** (~1990) became de facto web scripting language
  
  ```
  print "Hello, World!\n"
  ```

...
Scripting Languages …

Innovations

❑ Pipes and filters (Unix shell)
❑ Generalized embedding/command languages (TCL)

Successes

❑ Unix Shell, awk, emacs, HyperTalk, AppleTalk, TCL, Python, Perl, VisualBasic …
What you should know!

▷ What, exactly, is a programming language?
▷ How do compilers and interpreters differ?
▷ Why was FORTRAN developed?
▷ What were the main achievements of ALGOL 60?
▷ Why do we call C a “Third Generation Language”?
▷ What is a “Fourth Generation Language”? 
Can you answer these questions?

Why are there so many programming languages?
Why are FORTRAN and COBOL still important programming languages?
Which language should you use to implement a spelling checker?
A filter to translate upper-to-lower case?
A theorem prover?
An address database?
An expert system?
A game server for initiating chess games on the internet?
A user interface for a network chess client?
2. Systems Programming

Overview

- C Features
- Memory layout
- Declarations and definitions
- Working with Pointers

Reference:

What is C?

C was designed as a *general-purpose language* with a very *direct mapping* from data types and operators to machine instructions.

- **cpp** (C pre-processor) used for expanding macros and inclusion of declaration "header files"
- explicit *memory allocation* (no garbage collection)
- memory manipulation through *pointers*, pointer arithmetic and typecasting
- used as *portable*, high-level assembler
## C Features

Developed in 1972 by Dennis Ritchie and Brian Kernighan as a **systems language** for Unix on the PDP-11. A successor to B [Thompson, 1970], in turn derived from BCPL.

<table>
<thead>
<tr>
<th><strong>C preprocessor:</strong></th>
<th>file inclusion, conditional compilation, macros</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data types:</strong></td>
<td>char, short, int, long, double, float</td>
</tr>
<tr>
<td><strong>Type constructors:</strong></td>
<td>pointer, array, struct, union</td>
</tr>
<tr>
<td><strong>Basic operators:</strong></td>
<td>arithmetic, pointer manipulation, bit manipulation ...</td>
</tr>
<tr>
<td><strong>Control abstractions:</strong></td>
<td>if/else, while/for loops, switch, goto ...</td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td>call-by-value, side-effects through pointers</td>
</tr>
<tr>
<td><strong>Type operations:</strong></td>
<td>typedef, sizeof, explicit type-casting and coercion</td>
</tr>
</tbody>
</table>
"Hello World" in C

Pre-processor directive: include declarations for standard i/o library

A comment

Function definition: there is always a "main" function

#include <stdio.h>
/* My first C program! */
int main(void)
{
    printf("hello world!\n");
    return 0;
}

A string constant: an array of 14 (not 13!) chars
Symbols

C programs are built up from *symbols*:

<table>
<thead>
<tr>
<th></th>
<th>{ alphabetic or underscore } followed by { alphanumerics or underscores }</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names</td>
<td>main, IOStack, _store, x10</td>
</tr>
<tr>
<td>Keywords</td>
<td>const, int, if, ...</td>
</tr>
<tr>
<td>Constants</td>
<td>&quot;hello world&quot;, 'a', 10, 077, 0x1F, 1.23e10</td>
</tr>
<tr>
<td>Operators</td>
<td>+, &gt;&gt;, *, &amp;</td>
</tr>
<tr>
<td>Punctuation</td>
<td>{. }, .,</td>
</tr>
</tbody>
</table>
Keywords

C has a large number of reserved words:

<table>
<thead>
<tr>
<th>Control flow:</th>
<th>break, case, continue, default, do, else, for, goto, if, return, switch, while</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarations:</td>
<td>auto, char, const, double, extern, float, int, long, register, short, signed, static, struct, typedef, union, unsigned, void</td>
</tr>
<tr>
<td>Expressions:</td>
<td>sizeof</td>
</tr>
</tbody>
</table>
Operators (same as Java)

```java
int a, b, c;
double d;
float f;
a = b = c = 7; // assignment: a == 7; b == 7; c == 7
a = (b == 7); // equality test: a == 1 (7 == 7)
b = !a; // negation: b == 0 (!1)
a = (b>=0)&&(c<10); // logical AND: a == 1 ((0>=0)&&(7<10))
a *= (b += c++); // increment: a == 7; b == 7; c == 8
a = 11 / 4; // integer division: a == 2
b = 11 % 4; // remainder: b == 3
d = 11 / 4; // d == 2.0 (not 2.75!)
f = 11.0 / 4.0; // f == 2.75
a = b|c; // bitwise OR: a == 11 (03|010)
b = a^c; // bitwise XOR: b == 3 (013^010)
c = a&b; // bitwise AND: c == 3 (013&03)
b = a<<c; // left shift: b == 88 (11<<3)
a = (b++,c--); // comma operator: a == 3; b == 89; c == 2
b = (a>c)?a:c; // conditional operator: b == 3 ((3>2)?3:2)
```
C Storage Classes

You must explicitly manage storage space for data

<table>
<thead>
<tr>
<th>Static</th>
<th>static objects exist for the <em>entire life-time</em> of the process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>only live <em>during function invocation</em> on the “run-time stack”</td>
</tr>
<tr>
<td>Dynamic</td>
<td>dynamic objects live between calls to <em>malloc</em> and <em>free</em></td>
</tr>
<tr>
<td></td>
<td>their lifetimes typically <em>extend beyond</em> their <em>scope</em></td>
</tr>
</tbody>
</table>
Memory Layout

The address space consists of (at least):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text</strong></td>
<td>executable program text (not writable)</td>
</tr>
<tr>
<td><strong>Static</strong></td>
<td>static data</td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td>dynamically allocated global memory (grows upward)</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td>local memory for function calls (grows downward)</td>
</tr>
</tbody>
</table>
#include <stdio.h>

static int stat=0;
void dummy() { }

int main(void)
{
    int local=1;
    int *dynamic = (int*) malloc(sizeof(int),1);

    printf("Text is here: %u\n", (unsigned) dummy); /* function pointer */
    printf("Static is here: %u\n", (unsigned) &stat);
    printf("Heap is here: %u\n", (unsigned) dynamic);
    printf("Stack is here: %u\n", (unsigned) &local);
}
Declarations and Definitions

Variables and functions must be either declared or defined before they are used:

- A **declaration** of a variable (or function) announces that the variable (function) exists and is defined somewhere else.

```c
extern char *greeting;
void hello(void);
```

- A **definition** of a variable (or function) causes storage to be allocated.

```c
char *greeting = "hello world!\n";
void hello(void)
{
    printf(greeting);
}
```
Header files

C does not provide modules — instead one should break a program into header files containing declarations, and source files containing definitions that may be separately compiled.

**Hello.h**

```c
extern char *greeting;
void hello(void);
```

**Hello.c**

```c
#include <stdio.h>

char *greeting = "hello world!\n";

void hello(void)
{
    printf(greeting);
}
```
Including header files

Our main program may now include declarations of the separately compiled definitions:

```c
#include "hello.h"

int main(void)
{
    hello();
    return 0;
}
```

cc -c helloMain.c
compile to object code

cc -c hello.c
compile to object code

cc helloMain.o hello.o -o helloMain
link to executable
Makefiles

You could also compile everything together:

```
cc helloMain.c hello.c -o helloMain
```

Or you could use a makefile to manage dependencies:

```
helloMain : helloMain.c hello.h hello.o
  cc helloMain.c hello.o -o $@

...  
```

☞ "Read the manual"
C Arrays

Arrays are **fixed sequences of homogeneous elements**.

- **Type** `a[n];` defines a one-dimensional array `a` in a contiguous block of `(n*sizeof(Type))` bytes.
- `n` must be a compile-time **constant**.
- Arrays bounds run from **0 to n-1**.
- **Size cannot vary** at run-time.
- They can be initialized at compile time:
  ```c
  int eightPrimes[8] =
  { 2, 3, 5, 7, 11, 13, 17, 19 };
  ```
- But **no range-checking** is performed at run-time:
  ```c
  eightPrimes[8] = 0; /* disaster! */
  ```
## Pointers

A **pointer** holds the **address** of another variable:

```c
int i = 10;
int *ip = &i; /* assign the address of i to ip */
```

<table>
<thead>
<tr>
<th>Use them to access and update variables:</th>
<th>*ip = *ip + 1;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array variables behave like pointers to their first element</td>
<td>int *ep = eightPrimes;</td>
</tr>
<tr>
<td>Pointers can be treated like arrays:</td>
<td>ep[7] = 23;</td>
</tr>
<tr>
<td>But have different sizes:</td>
<td>sizeof(eightPrimes) == 32)</td>
</tr>
<tr>
<td></td>
<td>sizeof(ep) == 4)</td>
</tr>
<tr>
<td>You may increment and decrement pointers:</td>
<td>ep = ep+1;</td>
</tr>
<tr>
<td>Declare a pointer to an unknown data type as void*</td>
<td>void *vp = ep;</td>
</tr>
<tr>
<td>But typecast it properly before using it!</td>
<td>((int*)vp)[6] = 29;</td>
</tr>
</tbody>
</table>
Strings

A string is a pointer to a NULL-terminated (i.e., '\0') character array:

<table>
<thead>
<tr>
<th>char *cp;</th>
<th>uninitialized string (pointer to a char)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *hi = &quot;hello&quot;;</td>
<td>initialized string pointer</td>
</tr>
<tr>
<td>char hello[6] = &quot;hello&quot;;</td>
<td>initialized char array</td>
</tr>
<tr>
<td>cp = hello;</td>
<td>cp now points to hello[]</td>
</tr>
<tr>
<td>cp[1] = 'u';</td>
<td>cp and hello now point to &quot;hullo&quot;</td>
</tr>
<tr>
<td>cp[4] = NULL;</td>
<td>cp and hello now point to &quot;hull&quot;</td>
</tr>
</tbody>
</table>

What is sizeof(hi)? sizeof(hello)?
# Pointer manipulation

**Copy string s1 to buffer s2:**

```c
void strCopy(char s1[], char s2[])
{
    int i = 0;
    while (s1[i] != '\0') {
        /* Assume s1 is NULL-terminated! */
        s2[i] = s1[i];      /* assume s2 is big enough! */
        i++;
    }
    s2[i] = '\0';
}
```

**More idiomatically (!):**

```c
void strCopy2(char *s1, char *s2)
{
    while (*s2++ = *s1++);
    /* fails only when NULL is reached */
}
```
Function Pointers

int ascii(char c) { return((int) c); } /* cast */

void applyEach(char *s, int (*fptr)(char)) {
    char *cp;
    for (cp = s; *cp; cp++)
        printf("%c -> %d\n", *cp, fptr(*cp));
}

int main(int argc, char *argv[]) {
    int i;
    for (i=1; i<argc; i++)
        applyEach(argv[i], ascii);
    return 0;
}
Working with pointers

Problem: read an arbitrary file, and print out the lines in reverse order.

Approach:

- Check the file size
- Allocate enough memory
- Read in the file
- Starting from the end of the buffer
  - Convert each newline (\n) to a NULL (\0)
  - Printing out lines as you go
- Free the memory.
Argument processing

```c
int main(int argc, char* argv[]) {
    int i;
    if (argc<1) {
        fprintf(stderr, "Usage: lrev <file> ...
" );
        exit(-1);
    }
    for (i=1;i<argc;i++) {
        lrev(argv[i]);
    }
    return 0;
}
```
Using pointers for side effects

Return pointer to file contents or NULL (error code)
Set bytes to file size

```c
char* loadFile(char *path, int *bytes)
{
    FILE *input;
    struct stat fileStat; /* see below ... */
    char *buf;
    *bytes = 0; /* default return val */
    if (stat(path, &fileStat) < 0) { /* POSIX std */
        return NULL; /* error-checking vs exceptions */
    }
    *bytes = (int) fileStat.st_size;
    ...
```
Memory allocation

NB: Error-checking code left out here for readability ...

```c
...  buf = (char*) malloc(sizeof(char)*((*bytes)+1));
...  input = fopen(path, "r");
...
int n = fread(buf, sizeof(char), *bytes, input);
...
buf[*bytes] = '\0'; /* terminate buffer */
fclose(input);
return buf;
}
```
Pointer manipulation

void lrev(char *path)
{
    char *buf, *end;
    int bytes;
    buf = loadFile(path, &bytes);
    ... 
    end = buf + bytes - 1; /* last byte of buffer */
    if ((*end == '\n') && (end >= buf)) {
        *end = '\0';
    }
    ...

✉ What if bytes = 0?
Pointer manipulation ... 

/* walk backwards, converting lines to strings */

while (end >= buf) {
    while (*end != '\n' && (end >= buf))
        end--;
    if (*end == '\n' && (end >= buf))
        *end = '\0';
    puts(end+1);
}
free(buf);

✎ Is this algorithm correct? How would you prove it?
Built-In Data Types

The precision of built-in data types may depend on the machine architecture!

<table>
<thead>
<tr>
<th>Data type</th>
<th>No. of bits</th>
<th>Minimal value</th>
<th>Maximal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>signed short</td>
<td>16</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>signed int</td>
<td>16/32</td>
<td>-32768/32767</td>
<td>32767/2147483647</td>
</tr>
<tr>
<td>signed long</td>
<td>32</td>
<td>-2147483648</td>
<td>214748647</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16/32</td>
<td>0</td>
<td>65535/4294967295</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32</td>
<td>0</td>
<td>4294967295</td>
</tr>
</tbody>
</table>
## Built-In Data Types ...

<table>
<thead>
<tr>
<th>Data type</th>
<th>No. of bytes</th>
<th>Min. exponent</th>
<th>Max. exponent</th>
<th>Decimal accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4</td>
<td>-38</td>
<td>+38</td>
<td>6</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>-308</td>
<td>+308</td>
<td>15</td>
</tr>
<tr>
<td>long double</td>
<td>8 / 10</td>
<td>-308 / -4932</td>
<td>+308 / 4932</td>
<td>15 / 19</td>
</tr>
</tbody>
</table>
User Data Types

Data structures are defined as C “structs”.

In `/usr/include/sys/stat.h`:

```c
struct stat {
    dev_t    st_dev;   /* inode's device */
    ino_t    st_ino;   /* inode's number */
    mode_t   st_mode;  /* inode protection mode */
    nlink_t  st_nlink; /* number of hard links */
    uid_t    st_uid;   /* user ID of the file's owner */
    gid_t    st_gid;   /* group ID of the file's group */
    ...
    off_t    st_size;  /* file size, in bytes */
    int64_t  st_blocks; /* blocks allocated for file */
    ...
};
```
Typedefs

Type names can be assigned with the typedef command:

```c
typedef long long int64_t;
typedef int64_t quad_t;
typedef quad_t off_t;  /* file offset */
```
Observations

- C can be used as either a high-level or low-level language
  - generally used as a “portable assembler”

- C gives you complete freedom
  - requires great discipline to use correctly

- Pointers are the greatest source of errors
  - off-by-one errors
  - invalid assumptions
  - failure to check return values
Obfuscated C

A fine tradition since 1984 ...

#define iv 4
#define v ;(void
#define XI(xi)int xi[iv*'V'];
#define L(c,l,i)c(){d(l);m(i);}
#include <stdio.h>
int*cc,c,i,ix='\t',exit(),x='\n'+'\d';XI(VI)XI(xi)extern(*vi[]())(,(*
signal())();char*V,cm,D['x'],M='\n',I,*gets();L(MV,V,(c=\'d',ix))m(x){v
signal(X/'I',vi[x]);}d(x)char*x;v)write(i,x,i);}L(MC,V,M+I)xv(){c>=i?m(c/M/M+M):(d(&M),m(cm));}L(mi,V+cm,M)L(md,V,M)MM(){c=c%M%X;V-=cm;m(ix);}
LXX(){gets(D)||*(vi[iv]()c=atoi(D);while(c>=X){c=-X;d("m");}V="ivxIcdm"
+iv;m(ix);}LV(){c=c;while((i=cc[*D=getchar()]>-I)i?(c?(c<i&l(-c-c,
"%d"),l(i,"+%d")):l(i,"%d")):l(&l(M,""),l(*D,"%c")),c=i;c&l(X,"")",l
(-i,"%c"));m(iv-!(i&I));}L(ml,V,\'f')li(){m(cm+!isatty(i=I));}ii(){m(cm= ++I)v)pipe(VI);cc=xi+cm++;for(V="jWYmDEnX";*V;V++)xi[*V"']|=c,xi[*V++]
=c,c*=M,xi[*V"']|=xi[*V]=c>I;cc[-I]=ix v)close(*VI);cc[M]=M;}main(){(*vi());for(;v)write(VI[I],V,M));}l(xl,lx)char*lx;{v)printf(lx-xl)v
fflush(stdout);}L(xx,V+I,(c=X/cm,ix))int(*vi[]())={ii,li,LXX,LV,exit,l,
d,l,d,xv,MM,md,MC,ml,MV,xx,xx,xx,xx,xx,xx,MV,mi};
A C Puzzle

What does this program do?

```c
char f[] = "char f[] = %c%s%c;%cmain() {printf(f, 34, f, 34, 10, 10);}%c"
main() {printf(f, 34, f, 34, 10, 10);}
```
What you should know!

- What is a header file for?
- What are declarations and definitions?
- What is the difference between a `char*` and a `char[]`?
- How do you allocate objects on the heap?
- Why should every C project have a makefile?
- What is `sizeof(“abcd”)`?
- How do you handle errors in C?
- How can you write functions with side-effects?
- What happens when you increment a pointer?
Can you answer these questions?

- Where can you find the system header files?
- What's the difference between c++ and ++c?
- How do malloc and free manage memory?
- How does malloc get more memory?
- What happens if you run: free("hello")?
- How do you write portable makefiles?
- What is sizeof(&main)?
- What trouble can you get into with typecasts?
- What trouble can you get into with pointers?
3. Multiparadigm Programming

Overview

- C++ vs C
- C++ vs Java
- References vs pointers
- C++ classes: Orthodox Canonical Form
- Templates and STL

References:

Essential C++ Texts

What is C++?

A “better C” that supports:

- Object-oriented programming (classes & inheritance)
- Generic programming (templates)
- Programming-in-the-large (namespaces, exceptions)
- Systems programming (thin abstractions)
- Reuse (large standard class library)
C++ vs C

Most C programs are also C++ programs.

Nevertheless, good C++ programs usually do not resemble C:

- avoid macros (use inline)
- avoid pointers (use references)
- avoid malloc and free (use new and delete)
- avoid arrays and char* (use vectors and strings) ...
- avoid structs (use classes)

C++ encourages a different style of programming:

- avoid procedural programming
  - model your domain with classes and templates
"Hello World" in C++

Include standard iostream classes

A C++ comment

```cpp
#include <iostream>
// My first C++ program!
int main(void)
{
    cout << "hello world!" << endl;
    return 0;
}
```

cout is an instance of ostream

operator overloading (two different argument types!)
C++ Design Goals

“C with Classes” designed by Bjarne Stroustrup in early 1980s:

- Originally a translator to C
  - Initially difficult to debug and inefficient

- Mostly *upward compatible* extension of C
  - “As close to C as possible, but no closer”
  - Stronger type-checking
  - Support for object-oriented programming

- Run-time efficiency
  - Language primitives close to machine instructions
  - *Minimal cost* for new features
# C++ Features

<table>
<thead>
<tr>
<th>Version</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C with Classes</strong></td>
<td>Classes as structs</td>
</tr>
<tr>
<td></td>
<td>Inheritance; virtual functions</td>
</tr>
<tr>
<td></td>
<td>Inline functions</td>
</tr>
<tr>
<td><strong>C++ 1.0 (1985)</strong></td>
<td>Strong typing; function prototypes</td>
</tr>
<tr>
<td></td>
<td>new and delete operators</td>
</tr>
<tr>
<td><strong>C++ 2.0</strong></td>
<td>Local classes; protected members</td>
</tr>
<tr>
<td></td>
<td>Multiple inheritance</td>
</tr>
<tr>
<td><strong>C++ 3.0</strong></td>
<td>Templates</td>
</tr>
<tr>
<td></td>
<td>Exception handling</td>
</tr>
<tr>
<td><strong>ANSI C++ (1998)</strong></td>
<td>Namespaces</td>
</tr>
<tr>
<td></td>
<td>RTTI</td>
</tr>
</tbody>
</table>
Java and C++ — Similarities and Extensions

Similarities:
- primitive data types (in Java, platform independent)
- syntax: control structures, exceptions ...
- classes, visibility declarations (public, private)
- multiple constructors, this, new
- types, type casting (safe in Java, not in C++)

Java Extensions:
- garbage collection
- standard abstract machine
- standard classes (came later to C++)
- packages (now C++ has namespaces)
- final classes
Java Simplifications

- no pointers — just references
- no functions — can declare static methods
- no global variables — use public static variables
- no destructors — garbage collection and finalize
- no linking — dynamic class loading
- no header files — can define interface
- no operator overloading — only method overloading
- no member initialization lists — call super constructor
- no preprocessor — static final constants and automatic inlining
- no multiple inheritance — implement multiple interfaces
- no structs, unions, enums — typically not needed
- no templates — but generics will likely be added ...
### New Keywords

In addition the keywords inherited from C, C++ adds:

<table>
<thead>
<tr>
<th>Exceptions</th>
<th>catch, throw, try</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declarations:</strong></td>
<td>bool, class, enum, explicit, export, friend, inline, mutable, namespace, operator, private, protected, public, template, typename, using, virtual, volatile, wchar_t</td>
</tr>
<tr>
<td><strong>Expressions:</strong></td>
<td>and, and_eq, bitand, bitor, compl, const_cast, delete, dynamic_cast, false, new, not, not_eq, or, or_eq, reinterpret_cast, static_cast, this, true, typeid, xor, xor_eq</td>
</tr>
</tbody>
</table>
Comments

Two styles:

/*
   * C-style comment pairs are generally used
   * for longer comments that span several lines.
   */

   // C++ comments are useful for short comments

Use // comments exclusively within functions so that any part can be commented out using comment pairs.
References

A reference is an alias for another variable:

```c
int i = 10;
int &ir = i;
ir = ir + 1;  // increment i
```

Once initialized, references cannot be changed.

References are especially useful in procedure calls to avoid the overhead of passing arguments by value, without the clutter of explicit pointer dereferencing:

```c
void refInc(int &n)
{
    n = n+1;  // increment the variable n refers to
}
```
References vs Pointers

References should be *preferred to pointers* except when:

- manipulating dynamically allocated objects
  - `new` returns an object pointer

- a variable must range over a *set* of objects
  - use a pointer to walk through the set
C++ Classes

C++ classes may be instantiated either *automatically* (on the stack):

```cpp
MyClass oVal;  // constructor called
               // destroyed when scope ends
```

or *dynamically* (in the heap)

```cpp
MyClass *oPtr;  // uninitialized pointer

oPtr = new MyClass;  // constructor called
                     // must be explicitly deleted
```
Constructors and destructors

Constructors can make use of *member initialization lists*:

```cpp
class MyClass {
private:
    string _name;
public:
    MyClass(string name) : _name(name) { // constructor
        cout << "create " << name << endl;
    }
    ~MyClass() { // destructor
        cout << "destroy " << _name << endl;
    }
};
```

C++ classes can specify cleanup actions in *destructors*
**Automatic and dynamic destruction**

```cpp
MyClass& start() { // returns a reference
    MyClass a("a"); // automatic
    MyClass *b = new MyClass("b"); // dynamic
    return *b; // returns a reference (!) to b
} // a goes out of scope

void finish(MyClass& b) {
    delete &b; // need pointer to b
}

finish(start());
```

create a
create b
destroy a
destroy b
Orthodox Canonical Form

Most of your classes should look like this:

```cpp
class myClass {
public:
    myClass(void); // default constructor
    myClass(const myClass& copy); // copy constructor
    ... // other constructors
    ~myClass(void); // destructor
    myClass& operator=(const myClass&); // assignment
    ... // other public member functions
private:
    ... 
};
```
Why OCF?

If you don’t define these four member functions, C++ will generate them:

- default constructor
  - will call default constructor for each data member
- destructor
  - will call destructor of each data member
- copy constructor
  - will shallow copy each data member
  - pointers will be copied, not the objects pointed to!
- assignment
  - will shallow copy each data member
**Example: A String Class**

We would like a String class that protects C-style strings:
- strings are indistinguishable from char pointers
- string updates may cause memory to be corrupted

**Strings should support:**
- creation and destruction
- initialization from char arrays
- copying
- safe indexing
- safe concatenation and updating
- output
- length, and other common operations ...
A Simple String.h

class String
{
    friend ostream& operator<<(ostream&, const String&);

public:
    String(void); // default constructor
    ~String(void); // destructor
    String(const String& copy); // copy constructor
    String(const char*s); // char* constructor
    String& operator=(const String&); // assignment

    inline int length(void) const { return ::strlen(_s); }
    char& operator[](const int n) throw(exception);
    String& operator+=(const String&) throw(exception); // concatenation

private:
    char * _s; // invariant: _s points to a null-terminated heap string
    void become(const char*) throw(exception); // internal copy function
};
Default Constructors

Every constructor should *establish the class invariant*:

```cpp
String::String(void)
{
    _s = new char[1];  // allocate a char array
    _s[0] = '\0';      // NULL terminate it!
}
```

The *default constructor* for a class is called when a new instance is declared without any initialization parameters:

- `String anEmptyString; // call String::String()`
- `String stringVector[10]; // call it ten times!`
Destructors

The String destructor must *explicitly* free any memory allocated by that object.

```cpp
String::~String (void)
{
    delete [] _s; // delete the char array
}
```

*Every new must be matched somewhere by a delete!*
- use `new` and `delete` for objects
- use `new[]` and `delete[]` for arrays!
Copy Constructors

Our String copy constructor must create a deep copy:

```cpp
String::String(const String& copy) {
    become(copy._s); // call helper
}

void String::become(const char* s) throw (exception) {
    _s = new char[::strlen(s) + 1];
    if (_s == 0) throw(logic_error("new failed"));
    ::strcpy(_s, s);
}
```
A few remarks ...

- If we do not define our own copy constructor, copies of Strings will *share the same representation*!
  - Modifying one will modify the other!
  - Destroying one will invalidate the other!

- If we do not declare `copy` as `const`, we will not be able to construct a copy of a `const` String!

- If we declare `copy` as `String` rather than `String&`, *a new copy will be made* before it is passed to the constructor!
  - Functions arguments are always **passed by value** in C++
  - The “value” of a pointer is a pointer!

- The abstraction boundary is a class, *not an object*. Within a class, all private members are visible (as is `copy._s`)
Other Constructors

Class constructors may have arbitrary arguments, as long as their signatures are unique and unambiguous:

```cpp
String::String(const char* s)
{
    become(s);
}
```

Since the argument is not modified, we can declare it as `const`. This will allow us to construct String instances from constant char arrays.
Assignment Operators

Assignment is different from the copy constructor because an instance already exists:

```cpp
String& String::operator=(const String& copy) {
    if (this != &copy) { // take care!
        delete [] _s;
        become(copy._s);
    }
    return *this; // NB: a reference, not a copy
}
```
A few more remarks ...

- Return `String&` rather than `void` so the result can be used in an expression.

- Return `String&` rather than `String` so the result won't be copied!

- This is a pseudo-variable whose value is a pointer to the current object.
  
  So *this is the value of the current object, which is returned by reference*
Implicit Conversion

When an argument of the “wrong” type is passed to a function, the C++ compiler looks for a constructor that will convert it to the “right” type:

```cpp
str = "hello world";
```

is implicitly converted to:

```cpp
str = String("hello world");
```
Operator Overloading

Not only assignment, but other useful operators can be “overloaded” provided their signatures are unique:

```cpp
char&
String::operator[](const int n) throw(exception) {
    if ((n<0) || (length()<=n)) {
        throw(logic_error("array index out of bounds"));
    }
    return _s[n];
}
```

NB: a non-const reference is returned, so can be used as an lvalue in an assignment.
Overloadable Operators

The following operators may be overloaded:

<table>
<thead>
<tr>
<th>Overloadable Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>++</td>
</tr>
<tr>
<td>+=</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
</tr>
</tbody>
</table>

**NB:** arity and precedence are fixed by C++
Friends

We would like to be able to write:

```c++
cout << String("TESTING ... ") << endl;
```

But:

- It can’t be a member function of `ostream`, since we can’t extend the standard library.
- It can’t be a member function of `String` since the target is `cout`.
- But it must have access to `String`’s private data

So ... we need a binary `function <<` that takes a `cout` and a `String` as arguments, and is a `friend` of `String`.
Friends ...

We declare:

```cpp
class String
{
    friend ostream&
        operator<<(ostream&, const String&);
    ...
};
```

And define:

```cpp
ostream&
operator<<(ostream& outStream, const String& s)
{
    return outStream << s._s;
}
```
What are Templates?

A template is a *generic specification* of a function or a class, *parameterized* by one or more types used within the function or class:

- functions that only assume basic operations of their arguments (comparison, assignment ...)
- “container classes” that do little else but hold instances of other classes

**Templates are essentially glorified macros**

- like macros, they are compiled only when instantiated (and so are defined exclusively in header files)
- unlike macros, templates are not expanded literally, but may be intelligently processed by the C++ compiler
The following declares a generic $\text{min}()$ function that will work for arbitrary, comparable elements:

```cpp
template <class Item>
inline const Item& min(const Item& a, const Item& b)
{
    return (a<b) ? a : b;
}
```

**Templates are automatically instantiated by need:**
```
cout << "min(3,5) = " << min(3,5) << endl;
// instantiates: inline const int& min(int&, int&);
```
**Class Templates**

Class templates are declared just like function templates:

```cpp
template <class First, class Second>
class pair {
public:
  First first;
  Second second;
  pair(const First& f, const Second& s) :
      first(f), second(s) {}
};
```
Using Class Templates

Template classes are instantiated by binding the formal parameter:

typedef pair<int, char*> MyPair;

MyPair myPair = MyPair(6, "I am not a number");

cout << myPair.first << " sez "
   << myPair.second << endl;

Typedefs are a convenient way to bind names to template instances.
Standard Template Library

STL is a general-purpose C++ library of generic algorithms and data structures.

1. **Containers** store collections of objects
   - vector, list, deque, set, multiset, map, multimap

2. **Iterators** traverse containers
   - random access, bidirectional, forward/backward ...

3. **Function Objects** encapsulate functions as objects
   - arithmetic, comparison, logical, and user-defined ...

4. **Algorithms** implement generic procedures
   - search, count, copy, random_shuffle, sort, ...

5. **Adaptors** provide an alternative interface to a component
   - stack, queue, reverse_iterator, ...
An STL Line Reverser

```cpp
#include <iostream>
#include <stack>          // STL stacks
#include <string>         // Standard strings

void rev(void)
{
    typedef stack<string> IOStack;  // instantiate the template
    IOStack ioStack;              // instantiate the template class
    string buf;

    while (getline(cin, buf)) {
        ioStack.push(buf);
    }
    while (ioStack.size() != 0) {
        cout << ioStack.top() << endl;
        ioStack.pop();
    }
}
```
What we didn’t have time for …

❑ virtual member functions, pure virtuals
❑ public, private and multiple inheritance
❑ default arguments, default initializers
❑ method overloading
❑ const declarations
❑ enumerations
❑ smart pointers
❑ static and dynamic casts
❑ template specialization
❑ namespaces
❑ RTTI

...
What you should know!

- What new features does C++ add to C?
- What does Java remove from C++?
- How should you use C and C++ commenting styles?
- How does a reference differ from a pointer?
- When should you use pointers in C++?
- Where do C++ objects live in memory?
- What is a member initialization list?
- Why does C++ need destructors?
- What is OCF and why is it important?
- What’s the difference between delete and delete[]?
- What is operator overloading?
- Why are templates like macros?
Can you answer these questions?

- Why doesn’t C++ support garbage collection?
- Why doesn’t Java support multiple inheritance?
- What trouble can you get into with references?
- Why doesn’t C++ just make deep copies by default?
- How can you declare a class without a default constructor?
- Why can objects of the same class access each others private members?
- Why are templates only defined in header files?
- How are templates compiled?
- What is the type of a template?
4. Stack-based Programming

Overview
- PostScript objects, types and stacks
- Arithmetic operators
- Graphics operators
- Procedures and variables
- Arrays and dictionaries

References:
- PostScript® Language Tutorial and Cookbook, Adobe Systems Incorporated, Addison-Wesley, 1985
What is PostScript?

PostScript “is a simple interpretive programming language ... to describe the appearance of text, graphical shapes, and sampled images on printed or displayed pages.”

- introduced in 1985 by Adobe
- display standard now supported by all major printer vendors
- simple, stack-based programming language
- minimal syntax
- large set of built-in operators
- PostScript programs are usually generated from applications, rather than hand-coded
Postscript variants

Level 1:
- the original 1985 PostScript

Level 2:
- additional support for dictionaries, memory management
  ...

Display PostScript:
- special support for screen display

Level 3:
- the current incarnation with “workflow” support
## Syntax

<table>
<thead>
<tr>
<th><strong>Comments:</strong></th>
<th>from &quot;%&quot; to next newline or formfeed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% This is a comment</td>
</tr>
<tr>
<td><strong>Numbers:</strong></td>
<td>signed integers, reals and radix numbers</td>
</tr>
<tr>
<td></td>
<td>123  -98  0  +17  - .002  34.5</td>
</tr>
<tr>
<td></td>
<td>123.6e10  1E-5  8#1777  16#FFE  2#1000</td>
</tr>
<tr>
<td><strong>Strings:</strong></td>
<td>text in <em>parentheses</em> or hexadecimal in <em>angle brackets</em> (Special characters are escaped: \n \t ( ) \ ... )</td>
</tr>
<tr>
<td><strong>Names:</strong></td>
<td>tokens consisting of “regular characters” but which aren’t numbers</td>
</tr>
<tr>
<td></td>
<td>abc Offset $$  23A  13-456  a.b</td>
</tr>
<tr>
<td></td>
<td>$MyDict  @pattern</td>
</tr>
<tr>
<td><strong>Literal names:</strong></td>
<td>start with <em>slash</em></td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------</td>
</tr>
<tr>
<td></td>
<td>/buffer /proc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Arrays:</strong></th>
<th>enclosed in <em>square brackets</em></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ 123 /abc (hello) ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Procedures:</strong></th>
<th>enclosed in <em>curly brackets</em></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{ add 2 div }</td>
</tr>
<tr>
<td></td>
<td>% add top two stack items and divide by 2</td>
</tr>
</tbody>
</table>
## Semantics

A PostScript program is a \textit{sequence of tokens}, representing \textit{typed objects}, that is interpreted to manipulate the \textit{display} and four \textit{stacks} that represent the execution state of a PostScript program:

<table>
<thead>
<tr>
<th>Stack Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Operand stack}:</td>
<td>holds (arbitrary) \textit{operands} and \textit{results} of PostScript operators</td>
</tr>
<tr>
<td>\textit{Dictionary stack}:</td>
<td>holds only \textit{dictionaries} where keys and values may be stored</td>
</tr>
<tr>
<td>\textit{Execution stack}:</td>
<td>holds \textit{executable objects} (e.g. procedures) in stages of execution</td>
</tr>
<tr>
<td>\textit{Graphics state stack}:</td>
<td>keeps track of current \textit{coordinates} etc.</td>
</tr>
</tbody>
</table>
Object types

Every object is either literal or executable:

Literal objects are pushed on the operand stack:
- integers, reals, string constants, literal names, arrays, procedures

Executable objects are interpreted:
- built-in operators
- names bound to procedures (in the current dictionary context)

Simple Object Types are copied by value
- boolean, fontID, integer, name, null, operator, real ...

Composite Object Types are copied by reference
- array, dictionary, string ...
The operand stack

Compute the average of 40 and 60:

\[
40 \quad 60 \quad \text{add} \quad 2 \quad \text{div}
\]

At the end, the result is left on the top of the operand stack.
# Stack and arithmetic operators

<table>
<thead>
<tr>
<th>Stack</th>
<th>Op</th>
<th>New Stack</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>num₁ num₂</td>
<td>add</td>
<td>sum</td>
<td>num₁ + num₂</td>
</tr>
<tr>
<td>num₁ num₂</td>
<td>sub</td>
<td>difference</td>
<td>num₁ - num₂</td>
</tr>
<tr>
<td>num₁ num₂</td>
<td>mul</td>
<td>product</td>
<td>num₁ * num₂</td>
</tr>
<tr>
<td>num₁ num₂</td>
<td>div</td>
<td>quotient</td>
<td>num₁ / num₂</td>
</tr>
<tr>
<td>int₁ int₂</td>
<td>idiv</td>
<td>quotient</td>
<td>integer divide</td>
</tr>
<tr>
<td>int₁ int₂</td>
<td>mod</td>
<td>remainder</td>
<td>int₁ mod int₂</td>
</tr>
<tr>
<td>num den</td>
<td>atan</td>
<td>angle</td>
<td>arctangent of num/den</td>
</tr>
<tr>
<td>any</td>
<td>pop</td>
<td></td>
<td>discard top element</td>
</tr>
<tr>
<td>any₁ any₂</td>
<td>exch</td>
<td>any₂ any₁</td>
<td>exchange top two elements</td>
</tr>
<tr>
<td>any</td>
<td>dup</td>
<td>any any</td>
<td>duplicate top element</td>
</tr>
<tr>
<td>any₁ ... anyₙ n</td>
<td>copy</td>
<td>any₁ ... anyₙ any₁ ... anyₙ</td>
<td>duplicate top n elements</td>
</tr>
<tr>
<td>any₀ ... anyₙ</td>
<td>index</td>
<td>any₀ anyₙ</td>
<td>duplicate n+1th element</td>
</tr>
</tbody>
</table>

and many others ...
Drawing a Box

“A path is a set of straight lines and curves that define a region to be filled or a trajectory that is to be drawn on the current page.”

```
newpath % clear the current drawing path
100 100 moveto % move to (100,100)
100 200 lineto % draw a line to (100,200)
200 200 lineto
200 100 lineto
100 100 lineto
10 setlinewidth % set width for drawing
stroke % draw along current path
showpage % and display current page
```
# Path construction operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newpath</td>
<td>Initialize current path to be empty</td>
</tr>
<tr>
<td>currentpoint</td>
<td>Return current coordinates</td>
</tr>
<tr>
<td>moveto</td>
<td>Set current point to ((x, y))</td>
</tr>
<tr>
<td>lineto</td>
<td>Append straight line to ((x, y))</td>
</tr>
<tr>
<td>rlineto</td>
<td>Relative lineto to ((x, y))</td>
</tr>
<tr>
<td>arc</td>
<td>Append counterclockwise arc</td>
</tr>
<tr>
<td>closepath</td>
<td>Connect subpath back to start</td>
</tr>
<tr>
<td>fill</td>
<td>Fill current path with current colour</td>
</tr>
<tr>
<td>stroke</td>
<td>Draw line along current path</td>
</tr>
<tr>
<td>showpage</td>
<td>Output and reset current page</td>
</tr>
</tbody>
</table>

**Others:** arcn, arcto, curveto, rcurveto, flattenpath, ...
Coordinates are measured in *points*:

- **72 points = 1 inch**
- **2.54 cm**

21 cm = 595 points

29.7 cm = 840 points

A4 paper

(0,0)

(595, 840)
“Hello World” in Postscript

Before you can print text, you must (1) look up the desired font, (2) scale it to the required size, and (3) set it to be the current font.

```
/Times-Roman findfont  % look up Times Roman font
18 scalefont          % scale it to 18 points
setfont              % set this to be the current font
100 500 moveto        % go to coordinate (100, 500)
(Hello world) show    % draw the string "Hello world"
showpage              % render the current page
```

Hello world
Character and font operators

<table>
<thead>
<tr>
<th>key</th>
<th>findfont</th>
<th>font</th>
<th>return font dict identified by key</th>
</tr>
</thead>
<tbody>
<tr>
<td>font scale</td>
<td>scalefont</td>
<td>font'</td>
<td>scale font by scale to produce font'</td>
</tr>
<tr>
<td>font</td>
<td>setfont</td>
<td>-</td>
<td>set font dictionary</td>
</tr>
<tr>
<td>-</td>
<td>currentfont</td>
<td>font</td>
<td>return current font</td>
</tr>
<tr>
<td>string</td>
<td>show</td>
<td>-</td>
<td>print string</td>
</tr>
<tr>
<td>string</td>
<td>stringwidth</td>
<td>$w_x$ $w_y$</td>
<td>width of string in current font</td>
</tr>
</tbody>
</table>

Others: definefont, makefont, FontDirectory, StandardEncoding ....
Procedures and Variables

Variables and procedures are defined by binding *names* to literal or executable objects.

```
<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
<th>def</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Define a general procedure to compute averages:**

```
/average { add 2 div } def
% bind the name "average" to "{ add 2 div }"
```

40 60 average

```
40
40
100
100
50
```
A Box procedure

Most PostScript programs consist of a prologue and a script.

% Prologue -- application specific procedures

/prox {
    % grey x y -> __
    newpath
    moveto % x y -> __
    0 150 rlineto % relative lineto
    150 0 rlineto
    0 -150 rlineto
    closepath % cleanly close path!
    setgray % grey -> __
    fill % colour in region
}

def

% Script -- usually generated

0 100 100 box
0.4 200 200 box
0.6 300 300 box
0 setgray
showpage
# Graphics state and coordinate operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num setlinewidth -</td>
<td>set line width</td>
</tr>
<tr>
<td>num setgray -</td>
<td>set colour to gray value (0 = black; 1 = white)</td>
</tr>
<tr>
<td>sx sy scale -</td>
<td>scale use space by sx and sy</td>
</tr>
<tr>
<td>angle rotate -</td>
<td>rotate user space by angle degrees</td>
</tr>
<tr>
<td>tx ty translate -</td>
<td>translate user space by (tx, ty)</td>
</tr>
<tr>
<td>- matrix matrix create identity matrix</td>
<td></td>
</tr>
<tr>
<td>matrix currentmatrix matrix fill matrix with CTM</td>
<td></td>
</tr>
<tr>
<td>matrix setmatrix -</td>
<td>replace CTM by matrix</td>
</tr>
<tr>
<td>- gsave -</td>
<td>save graphics state</td>
</tr>
<tr>
<td>- grestore -</td>
<td>restore graphics state</td>
</tr>
</tbody>
</table>

**gsave saves the current path, gray value, line width and user coordinate system**
A Fibonacci Graph

/fibInc { % m n \rightarrow n (m+n)
    exch % m n \rightarrow n m
    1 index % n m \rightarrow n m n
    add
} def
/x 0 def /y 0 def /dx 10 def
newpath
100 100 translate % make (100, 100) the origin
x y moveto % i.e., relative to (100, 100)
0 1 25 {
    /x x dx add def % increment x
    dup /y exch 100 idiv def % set y to 1/100 last fib value
    x y lineto % draw segment
    fibInc
} repeat
2 setlinewidth
stroke
showpage
Numbers and Strings

Numbers and other objects must be converted to strings before they can be printed:

<table>
<thead>
<tr>
<th>int</th>
<th>string</th>
<th>create string of capacity int</th>
</tr>
</thead>
<tbody>
<tr>
<td>any string</td>
<td>cvs</td>
<td>substring</td>
</tr>
</tbody>
</table>
Factorial

/LM 100 def % left margin
/FS 18 def % font size
/sBuf 20 string def % string buffer of length 20
/fact { % n -> n!
  dup 1 lt % -> n bool
  { pop 1 } % 0 -> 1
  { dup % n -> n n
    1 % -> n n 1
    sub % -> n (n-1)
    fact % -> n (n-1)! NB: recursive lookup
    mul % n!
  }
  ifelse
} def
/showInt { % n -> ___
  sBuf cvs show % convert an integer to a string and show it
} def
Factorial ...

/showFact {  % n -> __
  dup showInt  % show n
  (! = ) show
  fact showInt  % show n!
} def
/newline {  % __ -> __
  currentpoint exch pop  % get current y
  FS 2 add sub  % subtract offset
  LM exch moveto  % move to new x y
} def

/Times-Roman findfont FS scalefont setfont
LM 600 moveto
0 1 20 { showFact newline } for  % do from 0 to 20
showpage

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6.22702e+09
14! = 8.71783e+10
15! = 1.30767e+12
16! = 2.09228e+13
17! = 3.55687e+14
18! = 6.40237e+15
19! = 1.21645e+17
20! = 2.4329e+18
### Boolean, control and string operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eq</code></td>
<td>test equal</td>
</tr>
<tr>
<td><code>ne</code></td>
<td>test not equal</td>
</tr>
<tr>
<td><code>ge</code></td>
<td>test greater or equal</td>
</tr>
<tr>
<td><code>true</code></td>
<td>push boolean value <code>true</code></td>
</tr>
<tr>
<td><code>false</code></td>
<td>push boolean value <code>false</code></td>
</tr>
<tr>
<td><code>if</code></td>
<td>execute <code>proc</code> if <code>bool</code> is true</td>
</tr>
<tr>
<td><code>ifelse</code></td>
<td>execute <code>proc1</code> if <code>bool</code> is true else <code>proc2</code></td>
</tr>
<tr>
<td><code>for</code></td>
<td>execute <code>proc</code> with values <code>init</code> to <code>limit</code> by <code>steps of incr</code></td>
</tr>
<tr>
<td><code>repeat</code></td>
<td>execute <code>proc</code> <code>int</code> times</td>
</tr>
<tr>
<td><code>length</code></td>
<td>number of elements in <code>string</code></td>
</tr>
<tr>
<td><code>get</code></td>
<td>get element at position <code>index</code></td>
</tr>
<tr>
<td><code>put</code></td>
<td>put <code>int</code> into <code>string</code> at position <code>index</code></td>
</tr>
<tr>
<td><code>forall</code></td>
<td>execute <code>proc</code> for each element of <code>string</code></td>
</tr>
</tbody>
</table>
A simple formatter

/LM 100 def % left margin
/RM 250 def % right margin
/FS 18 def % font size
/showStr { % string -> __
  dup stringwidth pop % get (just) string’s width
currentpoint pop % current x position
  add % where printing would bring us
  RM gt { newline } if % newline if this would overflow RM
  show
} def
/newline { % __ -> __
  currentpoint exch pop % get current y
  FS 2 add sub % subtract offset
  LM exch moveto % move to new x y
} def
/format { { showStr ( ) show } forall } def % array -> __
/Times-Roman findfont FS scalefont setfont
LM 600 moveto
A simple formatter ...

[ (Now) (is) (the) (time) (for) (all) (good) (men) (to) (come) (to) (the) (aid) (of) (the) (party.) ] format showpage

Now is the time for all good men to come to the aid of the party.
# Array and dictionary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[- mark ]</code></td>
<td>start array construction</td>
</tr>
<tr>
<td><code>mark obj0 ... objn-1 ]</code></td>
<td>end array construction</td>
</tr>
<tr>
<td><code>int array</code></td>
<td>create array of length <code>n</code></td>
</tr>
<tr>
<td><code>array length</code></td>
<td>number of elements in array</td>
</tr>
<tr>
<td><code>array index get</code></td>
<td>get element at index position</td>
</tr>
<tr>
<td><code>array index any put</code></td>
<td>put element at index position</td>
</tr>
<tr>
<td><code>array proc forall</code></td>
<td>execute proc for each array element</td>
</tr>
<tr>
<td><code>int dict</code></td>
<td>create dictionary of capacity <code>int</code></td>
</tr>
<tr>
<td><code>dict length</code></td>
<td>number of key-value pairs</td>
</tr>
<tr>
<td><code>dict maxlength</code></td>
<td>capacity</td>
</tr>
<tr>
<td><code>dict begin</code></td>
<td>push <code>dict</code> on dict stack</td>
</tr>
<tr>
<td><code>- end</code></td>
<td>pop dict stack</td>
</tr>
</tbody>
</table>
Using Dictionaries — Arrowheads

/arrowdict 14 dict def % make a new dictionary
arrowdict begin
  /mtrx matrix def % allocate space for a matrix
end
/arrow {
  arrowdict begin % open the dictionary
    /headlength exch def % grab args
    /halfheadthickness exch 2 div def
    /halfthickness exch 2 div def
    /tipy exch def
    /tipx exch def
    /taily exch def
    /tailx exch def
    /dx tipx tailx sub def
    /dy tipy taily sub def
    /arrowlength dx dx mul dy dy mul add sqrt def
    /angle dy dx atan def
    /base arrowlength headlength sub def
}
/savematrix mtrx currentmatrix def % save the coordinate system
tailx taily translate % translate to start of arrow
angle rotate % rotate coordinates
0 halfthick neg moveto % draw as if starting from (0,0)
base halfthick neg lineto
base halfheadthick neg lineto
arrowlength 0 lineto
base halfheadthick lineto
base halfthick lineto
0 halfthick lineto
closepath
savematrix setmatrix % restore coordinate system
end
} def
Instantiating Arrows

newpath
    318 340 72 340 10 30 72 arrow
fill
newpath
    382 400 542 560 72 232 116 arrow
3 setlinewidth stroke
newpath
    400 300 400 90 90 200 200 3 sqrt mul 2 div arrow
.65 setgray fill
showpage
Encapsulated PostScript

EPSF is a standard format for importing and exporting PostScript files between applications.

%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 90 490 200 520
/Times-Roman findfont
  18 scalefont
  setfont
100 500 moveto
(Hello world) show
showpage

Hello world

(90, 490)

(200, 520)
What you should know!

- What kinds of stacks does PostScript manage?
- When does PostScript push values on the operand stack?
- What is a path, and how can it be displayed?
- How do you manipulate the coordinate system?
- Why would you define your own dictionaries?
- How do you compute a bounding box for your PostScript graphic?
Can you answer these questions?

✎ How would you program this graphic?

✎ When should you use translate instead of moveto?

✎ How could you use dictionaries to simulate object-oriented programming?
5. Functional Programming

Overview

- Functional vs. Imperative Programming
- Referential Transparency
- Recursion
- Pattern Matching
- Higher Order Functions
- Lazy Lists
References

- Simon Peyton Jones and John Hughes [editors], Report on the Programming Language Haskell 98 A Non-strict, Purely Functional Language, February 1999

[www.haskell.org](http://www.haskell.org)
# A Bit of History

<table>
<thead>
<tr>
<th><strong>Lambda Calculus</strong> (Church, 1932-33)</th>
<th>formal model of computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lisp</strong> (McCarthy, 1960)</td>
<td>symbolic computations with lists</td>
</tr>
<tr>
<td><strong>APL</strong> (Iverson, 1962)</td>
<td>algebraic programming with arrays</td>
</tr>
<tr>
<td><strong>ISWIM</strong> (Landin, 1966)</td>
<td><em>let</em> and <em>where</em> clauses</td>
</tr>
<tr>
<td></td>
<td>equational reasoning; birth of “pure” functional programming ...</td>
</tr>
</tbody>
</table>
# A Bit of History

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ML</strong></td>
<td>originally meta language for theorem proving</td>
</tr>
<tr>
<td>(Edinburgh, 1979)</td>
<td></td>
</tr>
<tr>
<td><strong>SASL, KRC, Miranda</strong></td>
<td>lazy evaluation</td>
</tr>
<tr>
<td>(Turner, 1976-85)</td>
<td></td>
</tr>
<tr>
<td><strong>Haskell</strong></td>
<td>“Grand Unification” of functional languages ...</td>
</tr>
<tr>
<td>(Hudak, Wadler, et al., 1988)</td>
<td></td>
</tr>
</tbody>
</table>
Programming without State

Imperative style:

\[
\begin{align*}
n &:= x; \\
a &:= 1; \\
\text{while } n > 0 \text{ do} \\
&\quad \text{begin } a := a \times n; \\
&\quad \quad n := n - 1; \\
\text{end};
\end{align*}
\]

Declarative (functional) style:

\[
\begin{align*}
\text{fac } n &= \;
\begin{align*}
\text{if } n &= 0 \\
\text{then } 1 \\
\text{else } n \times \text{fac } (n - 1)
\end{align*}
\end{align*}
\]

Programs in pure functional languages have \textbf{no explicit state.}
Programs are constructed entirely by composing expressions.
Pure Functional Programming Languages

Imperative Programming:
- Program = Algorithms + Data

Functional Programming:
- Program = Functions ◯ Functions

What is a Program?
A program (computation) is a transformation from input data to output data.
Key features of pure functional languages

1. All programs and procedures are functions
2. There are no variables or assignments — only input parameters
3. There are no loops — only recursive functions
4. The value of a function depends only on the values of its parameters
5. Functions are first-class values
**What is Haskell?**

Haskell is a *general purpose, purely functional* programming language incorporating many recent innovations in programming language design. Haskell provides *higher-order functions*, *non-strict semantics*, *static polymorphic typing*, *user-defined algebraic datatypes*, *pattern-matching*, *list comprehensions*, *a module system*, *a monadic I/O system*, and *a rich set of primitive datatypes*, including *lists, arrays, arbitrary and fixed precision integers, and floating-point numbers*. Haskell is both the culmination and solidification of many years of research on lazy functional languages.

— The Haskell 98 report
“Hello World” in Hugs

```hugs
hello() = print "Hello World"
```
Referential Transparency

A function has the property of referential transparency if its value depends only on the values of its parameters.

Does $f(x)+f(x)$ equal $2\cdot f(x)$? In C? In Haskell?

Referential transparency means that “equals can be replaced by equals”.

In a pure functional language, all functions are referentially transparent, and therefore always yield the same result no matter how often they are called.
Evaluation of Expressions

Expressions can be (formally) evaluated by substituting arguments for formal parameters in function bodies:

\[
\text{fac 4} \rightarrow \text{if 4 == 0 then 1 else 4 * fac (4-1)}
\]

\[
\rightarrow \begin{array}{l}
\quad 4 * \text{fac (4-1)} \\
\quad 4 * \text{if (4-1) == 0 then 1 else (4-1) * fac (4-1-1)} \\
\quad 4 * \text{if 3 == 0 then 1 else (4-1) * fac (4-1-1)} \\
\quad 4 * ((4-1) * \text{fac (4-1-1)}) \\
\quad 4 * ((4-1) * \text{if (4-1-1) == 0 then 1 else (4-1-1) * ...}) \\
\quad \ldots \\
\quad 4 * ((4-1) * ((4-1-1) * ((4-1-1-1) * 1))) \\
\quad \ldots \\
\quad 24
\end{array}
\]

Of course, real functional languages are not implemented by syntactic substitution ...
Tail Recursion

Recursive functions can be less efficient than loops because of the *high cost of procedure calls* on most hardware.

A *tail recursive function* calls itself *only* as its last operation, so the recursive call can be *optimized away* by a modern compiler since it needs only a single run-time stack frame:

\[
\begin{align*}
\text{fact } 5 & \rightarrow \text{fact } 5 \text{fact } 4 \rightarrow \text{fact } 5 \text{fact } 4 \text{fact } 3 \\
\text{sfac } 5 & \rightarrow \text{sfac } 4 \rightarrow \text{sfac } 3
\end{align*}
\]

...
Tail Recursion ...

A recursive function can be converted to a tail-recursive one by representing partial computations as explicit function parameters:

\[
\text{sfac } s \ n = \begin{cases} s & \text{if } n == 0 \\ s & \text{then } s \\ \text{else } \text{sfac } (s*n) \ (n-1) \end{cases}
\]

\[
\text{sfac} 1 \ 4 \Rightarrow \text{sfac} (1*4) \ (4-1) \\
\Rightarrow \text{sfac} 4 \ 3 \\
\Rightarrow \text{sfac} (4*3) \ (3-1) \\
\Rightarrow \text{sfac} 12 \ 2 \\
\Rightarrow \text{sfac} (12*2) \ (2-1) \\
\Rightarrow \text{sfac} 24 \ 1 \\
\Rightarrow ... \Rightarrow 24
\]
Equational Reasoning

Theorem:
For all \( n \geq 0 \), \( \text{fac } n = \text{sfac } 1 \ n \)

Proof of theorem:
\( n = 0 \): \( \text{fac } 0 = 1 = \text{sfac } 1 \ 0 \)
\( n > 0 \): Suppose
\[
\begin{align*}
\text{fac } (n-1) &= \text{sfac } 1 \ (n-1) \\
\text{fac } n &= n \times \text{fac } (n-1) \quad \text{— by def} \\
&= n \times \text{sfac } 1 \ (n-1) \\
&= \text{sfac } n \ (n-1) \quad \text{— by lemma} \\
&= \text{sfac } 1 \ n \quad \text{— by def}
\end{align*}
\]
...

© O. Nierstrasz — U. Berne  Functional Programming
Equational Reasoning ... 

Lemma:
For all \( n \geq 0 \), \( \text{sfac } s \ n = s \times \text{sfac } 1 \ n \)

Proof of lemma:
\( n = 0 \): \( \text{sfac } s \ 0 = s = s \times \text{sfac } 1 \ 0 \)
\( n > 0 \): Suppose:
\( \text{sfac } s \ (n-1) = s \times \text{sfac } 1 \ (n-1) \)
\( \text{sfac } s \ n = \text{sfac } (s \times n) \ (n-1) \)
\( = s \times n \times \text{sfac } 1 \ (n-1) \)
\( = s \times \text{sfac } n \ (n-1) \)
\( = s \times \text{sfac } 1 \ n \)
Pattern Matching

Haskell support multiple styles for specifying case-based function definitions:

Patterns:
\[
\begin{align*}
\text{fac'} \ 0 &= 1 \\
\text{fac'} \ n &= n \times \text{fac'} \ (n-1)
\end{align*}
\]

-- or: \( \text{fac'} \ (n+1) = (n+1) \times \text{fac'} \ n \)

 Guards:
\[
\begin{align*}
\text{fac'}' \ n \mid n &= 0 = 1 \\
\mid n &\geq 1 = n \times \text{fac'}' \ (n-1)
\end{align*}
\]
Lists

Lists are pairs of elements and lists of elements:

- \[ \] — stands for the empty list

- \( x:xs \) — stands for the list with \( x \) as the head and \( xs \) as the rest of the list

- \([1,2,3]\) — is syntactic sugar for \(1:2:3:[]\)

- \([1..n]\) — stands for \([1,2,3, \ldots, n]\)
Using Lists

Lists can be *deconstructed using* patterns:

\[
\begin{align*}
\text{head } (x:_\_) &= x \\
\text{len } [ ] &= 0 \\
\text{len } (x:xs) &= 1 + \text{len } xs \\
\text{prod } [ ] &= 1 \\
\text{prod } (x:xs) &= x \times \text{prod } xs \\
\text{fac'''} n &= \text{prod } [1..n]
\end{align*}
\]
Higher Order Functions

Higher-order functions treat other functions as *first-class values* that can be composed to produce new functions.

\[
\begin{align*}
\text{map } f \; [ ] & = [ ] \\
\text{map } f \; (x:xs) & = f \; x : \text{map } f \; xs
\end{align*}
\]

\[
\text{map fac } [1..5] \quad \Rightarrow \quad [1, 2, 6, 24, 120]
\]

**NB:** \text{map fac} is a new function that can be applied to lists:

\[
\begin{align*}
mfac & = \text{map fac} \\
mfac \; [1..3] & \quad \Rightarrow \quad [1, 2, 6]
\end{align*}
\]
Anonymous functions

Anonymous functions can be written as “lambda abstractions”. The function \( \lambda x \to x * x \) behaves exactly like \( \text{sqr} \):

\[
\text{sqr} \ x = x * x
\]

\[
\text{sqr} \ 10 \quad \Rightarrow 100
\]

\[
(\lambda x \to x * x) \ 10 \quad \Rightarrow 100
\]

Anonymous functions are first-class values:

\[
\text{map} \ (\lambda x \to x * x) \ [1..10]
\]

\[
\Rightarrow [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
\]
Curried functions

A curried function [named after the logician H.B. Curry] takes its arguments one at a time, allowing it to be treated as a higher-order function.

\[
\begin{align*}
\text{plus } x \; y &= x + y \quad -- \text{curried addition} \\
\text{plus } 1 \; 2 &= 3 \\
\text{plus'}(x,y) &= x + y \quad -- \text{normal addition} \\
\text{plus'}(1,2) &= 3
\end{align*}
\]
Understanding Curried functions

\[ \text{plus } x \ y = x + y \]

*is the same as:*

\[ \text{plus } x = \lambda y \rightarrow x + y \]

In other words, \text{plus} is a function of one argument that returns a function as its result.

\[ \text{plus } 5 \ 6 \]

*is the same as:*

\[ (\text{plus } 5) \ 6 \]

In other words, we invoke \((\text{plus } 5)\), obtaining a function,

\[ \lambda y \rightarrow 5 + y \]

which we then pass the argument 6, yielding 11.
Using Curried functions

Curried functions are useful because we can bind their argument *incrementally*

```haskell
inc = plus 1  -- bind first argument to 1
inc 2 \rightarrow 3

fac = sfac 1  -- binds first argument of
where sfac s n  -- a curried factorial
    | n == 0 = s
    | n >= 1 = sfac (s*n) (n-1)
```
Currying

The following (pre-defined) function takes a binary function as an argument and turns it into a curried function:

curry f a b = f (a, b)

plus(x,y) = x + y ---- not curried!
inc = (curry plus) 1

sfac(s, n) = if n == 0 ---- not curried
then s
else sfac (s*n, n-1)

fac = (curry sfac) 1 ---- bind first argument
Multiple Recursion

*Naive* recursion may result in *unnecessary* recalculations:

\[
\begin{align*}
\text{fib } 1 & = 1 \\
\text{fib } 2 & = 1 \\
\text{fib } (n+2) & = \text{fib } n + \text{fib } (n+1)
\end{align*}
\]

Efficiency can be regained by *explicitly passing* calculated values:

\[
\begin{align*}
\text{fib}' \ 1 & = 1 \\
\text{fib}' \ n & = a \quad \text{where} \quad (a,\_\,) = \text{fibPair } n \\
\text{fibPair } 1 & = (1,0) \\
\text{fibPair } (n+2) & = (a+b,a) \\
& \quad \text{where} \quad (a,b) = \text{fibPair } (n+1)
\end{align*}
\]

✍️ *How would you write a tail-recursive Fibonacci function?*
Lazy Evaluation

“Lazy”, or “normal-order” evaluation only evaluates expressions when they are actually needed. Clever implementation techniques (Wadsworth, 1971) allow replicated expressions to be shared, and thus avoid needless recalculations.

So:

\[
\text{sqr } n = n \times n
\]

\[
\text{sqr } (2+5) \Rightarrow (2+5) \times (2+5) \Rightarrow 7 \times 7 \Rightarrow 49
\]

Lazy evaluation allows some functions to be evaluated even if they are passed incorrect or non-terminating arguments:

\[
\text{ifTrue True x y } = x
\]

\[
\text{ifTrue False x y } = y
\]

\[
\text{ifTrue True 1 (5/0) } \Rightarrow 1
\]
Lazy Lists

Lazy lists are *infinite data structures* whose values are generated by need:

from n = n : from (n+1)

**from 10** $\Rightarrow$ [10,11,12,13,14,15,16,17,....]

take 0 _ = [ ]
take _ [ ] = [ ]
take (n+1) (x:xs) = x : take n xs

**take 5 (from 10)** $\Rightarrow$ [10, 11, 12, 13, 14]

*NB: The lazy list (from n) has the special syntax: [n..]*
Many sequences are naturally implemented as lazy lists. Note the top-down, declarative style:

```haskell
fibs = 1 : 1 : fibsFollowing 1 1
    where fibsFollowing a b =
            (a+b) : fibsFollowing b (a+b)
```

```haskell
take 10 fibs

⇒ [ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

How would you re-write `fibs` so that `(a+b)` only appears once?
Declarative Programming Style

primes = primesFrom 2
primesFrom n = p : primesFrom (p+1)
    where p = nextPrime n

nextPrime n
    | isPrime n   = n
    | otherwise   = nextPrime (n+1)

isPrime 2 = True
isPrime n = notDivisible primes n

notDivisible (k:ps) n
    | (k*k) > n       = True
    | (mod n k) == 0  = False
    | otherwise       = notDivisible ps n

take 100 primes \[ [2, 3, 5, 7, 11, 13, \ldots 523, 541] \]
What you should know!

- What is referential transparency? Why is it important?
- When is a function tail recursive? Why is this useful?
- What is a higher-order function? An anonymous function?
- What are curried functions? Why are they useful?
- How can you avoid recalculating values in a multiply recursive function?
- What is lazy evaluation?
- What are lazy lists?
Can you answer these questions?

✎ Why don’t pure functional languages provide loop constructs?

✎ When would you use patterns rather than guards to specify functions?

✎ Can you build a list that contains both numbers and functions?

✎ How would you simplify fibs so that (a+b) is only called once?

✎ What kinds of applications are well-suited to functional programming?
6. Type Systems

Overview

- What is a Type?
- Static vs. Dynamic Typing
- Kinds of Types
- Overloading
- User Data Types
- Polymorphic Types
References

What is a Type?

Type errors:

\[ 5 + [ ] \]

**ERROR: Type error in application**

*** expression : 5 + [ ]
*** term : 5
*** type : Int
*** does not match : [a]

A type is a set of values?

- int = \{ ... -2, -1, 0, 1, 2, 3, ... \}
- bool = \{ True, False \}
- Point = \{ [x=0,y=0], [x=1,y=0], [x=0,y=1] ... \}
What is a Type?

A type is a partial specification of behaviour?

- $n,m:\text{int} \Rightarrow n+m$ is valid, but $\text{not}(n)$ is an error

- $n:\text{int} \Rightarrow n := 1$ is valid, but $n := \text{"hello world"}$ is an error

What kinds of specifications are interesting? Useful?
Static and Dynamic Types

Values have **static types** defined by the programming language.

Variables and expressions have **dynamic types** determined by the values they assume at run-time.

Applet myApplet = new GameApplet();

- declared, static type is Applet
- actual dynamic type is GameApplet
Static and Dynamic Typing

A language is \textit{statically typed} if it is always possible to determine the (static) type of an expression \textit{based on the program text alone}.

A language is \textit{strongly typed} if it is possible to ensure that every expression is \textit{type consistent} based on the program text alone.

A language is \textit{dynamically typed} if only values have fixed type. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

Type consistency may be assured by (i) \textit{compile-time type-checking}, (ii) \textit{type inference}, or (iii) \textit{dynamic type-checking}.
Kinds of Types

All programming languages provide some set of built-in types.

- **Primitive types**: booleans, integers, floats, chars ...
- **Composite types**: functions, lists, tuples ...

Most strongly-typed modern languages provide for additional user-defined types.

- **User-defined types**: enumerations, recursive types, generic types, objects ...
Type Completeness

The Type Completeness Principle:

No operation should be arbitrarily restricted in the types of values involved. — Watt

First-class values can be evaluated, passed as arguments and used as components of composite values.

Functional languages attempt to make no class distinctions, whereas imperative languages typically treat functions (at best) as second-class values.
Function Types

Function types allow one to *deduce* the types of expressions without the need to evaluate them:

\[ \text{fact} :: \text{Int} \rightarrow \text{Int} \]

\[ 42 :: \text{Int} \quad \Rightarrow \quad \text{fact} \ 42 :: \text{Int} \]

**Curried types:**

\[ \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad = \quad \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]

and

\[ \text{plus} \ 5 \ 6 \quad = \quad ((\text{plus} \ 5) \ 6). \]

so:

\[ \text{plus} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \Rightarrow \quad \text{plus} \ 5 :: \text{Int} \rightarrow \text{Int} \]
List Types

A list of values of type \( a \) has the type \([a]\):

\[
[1] :: [\text{Int}]
\]

NB: All of the elements in a list must be of the same type!

\[\text{['a', 2, False]}\text{-- this is illegal! can't be typed!}\]
Tuple Types

If the expressions $x_1, x_2, ..., x_n$ have types $t_1, t_2, ..., t_n$ respectively, then the tuple $(x_1, x_2, ..., x_n)$ has the type $(t_1, t_2, ..., t_n)$:

$\begin{align*}
(1, [2], 3) &:: (\text{Int}, [\text{Int}], \text{Int}) \\
('a', \text{False}) &:: (\text{Char}, \text{Bool}) \\
((1,2),(3,4)) &:: ((\text{Int}, \text{Int}), (\text{Int}, \text{Int}))
\end{align*}$

The unit type is written () and has a single element which is also written as ().
User Data Types

New data types can be introduced by specifying (i) a `datatype name`, (ii) a set of `parameter types`, and (iii) a set of `constructors` for elements of the type:

```plaintext
data DatatypeName a1 ... an = constr1 | ... | constrm
```

where the constructors may be either:

1. **Named** constructors:
   ```plaintext
   Name type1 ... typek
   ```

2. **Binary** constructors (i.e., starting with ``":``):
   ```plaintext
   type1 CONOP type2
   ```
**Enumeration types**

User data types that do not hold any data can model enumerations:

```haskell
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

Functions over user data types must *deconstruct* the arguments, with one case for each constructor:

```haskell
whatShallIDo Sun = "relax"
whatShallIDo Sat = "go shopping"
whatShallIDo _ = "guess I'll have to go to work"
```
Union types

data Temp = Centigrade Float | Fahrenheit Float

freezing :: Temp -> Bool
freezing (Centigrade temp)= temp <= 0.0
freezing (Fahrenheit temp)= temp <= 32.0
Recursive Data Types

A recursive data type provides constructors over the type itself:

```haskell
data Tree a = Lf a | Tree a :^: Tree a
```

```haskell
mytree = (Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10
```

? :t mytree ➔ mytree :: Tree Int
Using recursive data types

leaves, leaves' :: Tree a -> [a]
leaves (Lf l) = [l]
leaves (l :^: r) = leaves l ++ leaves r

leaves' t = leavesAcc t []
  where leavesAcc (Lf l) = (l:)
        leavesAcc (l :^: r) = leavesAcc l . leavesAcc r

✎ What do these functions do?
✎ Which function should be more efficient? Why?
✎ What is (l:) and what does it do?
Monomorphism

Languages like Pascal and C have **monomorphic type systems**: every constant, variable, parameter and function result has a unique type.

- good for *type-checking*
- bad for writing *generic* code
  - it is impossible in Pascal to write a generic sort procedure
Polymorphism

A **polymorphic function** accepts *arguments of different types*:

```haskell
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```
Type Inference

We can infer the type of many expressions by simply examining their structure. Consider:

\[
\begin{align*}
\text{length } [ ] &= 0 \\
\text{length } (x:xs) &= 1 + \text{length } xs
\end{align*}
\]

Clearly:

\[
\text{length } :: a \rightarrow b
\]

Furthermore, \( b \) is obvious \( \text{int} \), and \( a \) is a list, so:

\[
\text{length } :: [c] \rightarrow \text{int}
\]

We cannot further refine the type, so we are done.
Composing polymorphic types

We can *deduce* the types of expressions using polymorphic functions by simply *binding type variables to concrete types*.

Consider:

\[
\begin{align*}
\text{length} & \quad :: \ [a] \rightarrow \text{Int} \\
\text{map} & \quad :: \ (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\end{align*}
\]

Then:

\[
\begin{align*}
\text{map length} & \quad :: \ [[a]] \rightarrow \text{[Int]} \\
[ \text{“Hello”, “World”} ] & \quad :: \ [[\text{Char}]] \\
\text{map length} \ [ \text{“Hello”, “World”} ] & \quad :: \ \text{[Int]}
\end{align*}
\]
Polymorphic Type Inference

Hindley-Milner Type Inference provides an effective algorithm for automatically determining the types of polymorphic functions.

\[
\begin{align*}
\text{map} & \quad : \quad [ \ ] & = & \quad [ \ ] \\
\text{map} & \quad : \quad (x:xs) & = & \quad f \ x : \ \text{map} \ f \ \text{xs} \\
\text{map} & \quad :: \quad X & \rightarrow & \quad Y & \rightarrow & \quad Z \\
\text{map} & \quad :: \quad (a \rightarrow b) & \rightarrow & \quad [ \ c \ ] & \rightarrow & \quad [ \ d \ ] \\
\text{map} & \quad :: \quad (a \rightarrow b) & \rightarrow & \quad [ \ a \ ] & \rightarrow & \quad [ \ b \ ]
\end{align*}
\]

The corresponding type system is used in many modern functional languages, including ML and Haskell.
Type Specialization

A polymorphic function may be explicitly assigned a more specific type:

\[ \text{idInt :: Int} \rightarrow \text{Int} \]
\[ \text{idInt} \ x = x \]

Note that the :t command can be used to find the type of a particular expression that is inferred by Haskell:

? :t \x -> [x]
\[ \x \rightarrow [x] :: a \rightarrow [a] \]

? :t (\x -> [x]) :: Char \rightarrow \text{String}
\[ \x \rightarrow [x] :: \text{Char} \rightarrow \text{String} \]
Kinds of Polymorphism

Polymorphism:

- Universal:
  - Parametric: polymorphic map function in Haskell; nil/void pointer type in Pascal/C
  - Inclusion: subtyping — graphic objects

- Ad Hoc:
  - Overloading: + applies to both integers and reals
  - Coercion: integer values can be used where reals are expected and v.v.
Coercion vs overloading

Coercion or overloading — how does one distinguish?
  3 + 4
  3.0 + 4
  3 + 4.0
  3.0 + 4.0

Are there several overloaded + functions, or just one, with values automatically coerced?
Overloading

Overloaded operators are introduced by means of type classes:

```haskell
class Eq a where
    (==), (=/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

A type class must be instantiated to be used:

```haskell
instance Eq Bool where
    True == True = True
    False == False = True
    _ == _ = False
```
Instantiating overloaded operators

For each overloaded instance a separate definition must be given ...

```
instance Eq Int where (==) = primEqInt
instance Eq Char where c == d = ord c == ord d
instance (Eq a, Eq b) => Eq (a,b) where
  (x,y) == (u,v) = x==u && y==v
instance Eq a => Eq [a] where
  [ ] == [ ] = True
  [ ] == (y:ys) = False
  (x:xs) == [ ] = False
  (x:xs) == (y:ys) = x==y && xs==ys
```
Equality for Data Types

Why not automatically provide equality for all types of values?

User data types:

```hs
data Set a = Set [a]
instance Eq a => Eq (Set a) where
  Set xs == Set ys = xs `subset` ys && ys `subset` xs
  where xs `subset` ys = all (`elem` ys) xs
```

✎ How would you define equality for the Tree data type?

NB: all (`elem` ys) xs tests that every x in xs is an element of ys
Equality for Functions

Functions:

? (1==) == (\x->1==x)

ERROR: Cannot derive instance in expression
*** Expression : (==) d148 ((==) {dict} 1) (\x->(==) {dict} 1 x)
*** Required instance : Eq (Int -> Bool)

Determining equality of functions is **undecidable** in general!
What you should know!

- How are the types of functions, lists and tuples specified?
- How can the type of an expression be inferred without evaluating it?
- What is a polymorphic function?
- How can the type of a polymorphic function be inferred?
- How does overloading differ from parametric polymorphism?
- How would you define == for tuples of length 3?
- How can you define your own data types?
- Why isn’t == pre-defined for all types?
Can you answer these questions?

- Can any set of values be considered a type?
- Why does Haskell sometimes fail to infer the type of an expression?
- What is the type of the predefined function `all`? How would you implement it?
- How would you define equality for the Tree data type?
Chapter 7: Introduction to the Lambda Calculus

Overview

- What is Computability? — Church’s Thesis
- Lambda Calculus — operational semantics
- The Church-Rosser Property
- Modelling basic programming constructs
References

What is Computable?

Computation is usually modelled as a mapping from inputs to outputs, carried out by a formal “machine,” or program, which processes its input in a sequence of steps.

An “effectively computable” function is one that can be computed in a finite amount of time using finite resources.
Church’s Thesis

Effectively computable functions [from positive integers to positive integers] are just those definable in the lambda calculus.

Or, equivalently:

It is not possible to build a machine that is more powerful than a Turing machine.

Church’s thesis cannot be proven because “effectively computable” is an intuitive notion, not a mathematical one. It can only be refuted by giving a counter-example — a machine that can solve a problem not computable by a Turing machine.

So far, all models of effectively computable functions have shown to be equivalent to Turing machines (or the lambda calculus).
Uncomputability

A problem that cannot be solved by any Turing machine in finite time (or any equivalent formalism) is called uncomputable.

Assuming Church’s thesis is true, an uncomputable problem cannot be solved by any real computer.

The Halting Problem:
Given an arbitrary Turing machine and its input tape, will the machine eventually halt?

The Halting Problem is provably uncomputable — which means that it cannot be solved in practice.
What is a Function? (I)

Extensional view:

A (total) function \( f: A \rightarrow B \) is a subset of \( A \times B \) (i.e., a relation) such that:

1. for each \( a \in A \), there exists some \((a, b) \in f\) (i.e., \( f(a) \) is defined), and

2. if \((a, b_1) \in f\) and \((a, b_2) \in f\), then \( b_1 = b_2 \) (i.e., \( f(a) \) is unique)
What is a Function? (II)

Intensional view:

A function \( f : A \to B \) is an abstraction \( \lambda x . e \), where \( x \) is a variable name, and \( e \) is an expression, such that when a value \( a \in A \) is substituted for \( x \) in \( e \), then this expression (i.e., \( f(a) \)) evaluates to some (unique) value \( b \in B \).
What is the Lambda Calculus?

The Lambda Calculus was invented by Alonzo Church [1932] as a mathematical formalism for expressing computation by functions.

Syntax:

\[ e ::= x \quad \text{a variable} \]
\[ | \quad \lambda x . e \quad \text{an abstraction (function)} \]
\[ | \quad e_1 e_2 \quad \text{a (function) application} \]

\( \lambda x . x \) — is a function taking an argument \( x \), and returning \( x \)
Parsing Lambda Expressions

Lambda extends as far as possible to the right

\[ \lambda f. x \ y \equiv \lambda f. (x \ y) \]

Application is left-associative

\[ x \ y \ z \equiv (x \ y) \ z \]

Multiple lambdas may be suppressed

\[ \lambda f \ g. x \equiv \lambda f . \lambda g. x \]
What is the Lambda Calculus? ...

(Operational) Semantics:

$\alpha$ conversion (renaming):
\[ \lambda x . e \leftrightarrow \lambda y . [y/x] e \]
where $y$ is not free in $e$

$\beta$ reduction (application):
\[ (\lambda x . e_1) e_2 \rightarrow [e_2/x] e_1 \]
avoiding name capture

$\eta$ reduction:
\[ \lambda x . e \; x \rightarrow e \]
if $x$ is not free in $e$

The lambda calculus can be viewed as the simplest possible pure functional programming language.
Beta Reduction

Beta reduction is the *computational engine* of the lambda calculus:

Define: \[ I \equiv \lambda x . x \]

Now consider:

\[ I \ I = (\lambda x . x)(\lambda x . x) \rightarrow [\lambda x . x / x]x \]
\[ = \lambda x . x \]
\[ = I \]

β reduction

substitution
Lambda expressions in Haskell

We can implement most lambda expressions directly in Haskell:

\[ i = \lambda x \rightarrow x \]

\[ i \ 5 \]

5
(2 reductions, 6 cells)

? i i 5

5
(3 reductions, 7 cells)
Lambdas are anonymous functions

A lambda abstraction is just an *anonymous function*.

Consider the Haskell function:

$$\text{compose } f \ g \ x = f(g(x))$$

The *value* of \(\text{compose}\) is the anonymous lambda abstraction:

$$\lambda f \ g \ x . \ f(\ g \ x)$$

*NB: This is the same as:*

$$\lambda f . \lambda g . \lambda x . \ f(\ g \ x)$$
A Few Examples

1. \((\lambda x.x) y\)
2. \((\lambda x.f x)\)
3. \((\lambda x.x) (\lambda x.x)\)
4. \((\lambda x.x y) z\)
5. \((\lambda x.y.x) t f\)
6. \((\lambda x.y.z.z x y) a b (\lambda x.y.x)\)
7. \((\lambda f.g.f.g) (\lambda x.x) (\lambda x.x) z\)
8. \((\lambda x.y.x y) y\)
9. \((\lambda x.y.x y) (\lambda x.x) (\lambda x.x)\)
10. \((\lambda x.y.x y) ((\lambda x.x) (\lambda x.x))\)
Free and Bound Variables

The variable x is \textit{bound} by \lambda in the expression: \lambda x.e

A variable that is not bound, is \textit{free}:

\[
\begin{align*}
fv(x) &= \{ x \} \\
fv(e_1 e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\lambda x . e) &= fv(e) - \{ x \}
\end{align*}
\]

An expression with \textit{no free variables} is \textit{closed}. (AKA a \textit{combinator}.) Otherwise it is \textit{open}.

For example, y is \textit{bound} and x is \textit{free} in the (open) expression: \lambda y . x y
“Hello World” in the Lambda Calculus

hello world

☞ Is this expression open? Closed?
Why macro expansion is wrong

Syntactic substitution will not work:

\[(\lambda x \cdot x y) y \rightarrow [y / x](\lambda y \cdot x y)\] \(\beta\) reduction

\[\neq (\lambda y \cdot y y)\] incorrect substitution!

Since \(y\) is already bound in \((\lambda y \cdot x y)\), we cannot directly substitute \(y\) for \(x\).
Substitution

We must define substitution carefully to avoid name capture:

\[
\begin{align*}
[e/x] x &= e \\
[e/x] y &= y & \text{if } x \neq y \\
[e/x] (e_1 e_2) &= ([e/x] e_1) ([e/x] e_2) \\
[e/x] (\lambda x . e_1) &= (\lambda x . e_1) \\
[e/x] (\lambda y . e_1) &= (\lambda y . [e/x] e_1) & \text{if } x \neq y \text{ and } y \notin \text{fv}(e) \\
[e/x] (\lambda y . e_1) &= (\lambda z . [e/x] [z/y] e_1) & \text{if } x \neq y \text{ and } z \notin \text{fv}(e) \cup \text{fv}(e_1)
\end{align*}
\]

Consider:

\[
(\lambda x . ((\lambda y . x) (\lambda x . x))) x y \rightarrow [y/x] (((\lambda y . x) (\lambda x . x))) x = (((\lambda z . y) (\lambda x . x)) y
\]
Alpha Conversion

Alpha conversions allow us to \textit{rename bound variables}.

A bound name $x$ in the lambda abstraction $(\lambda x. e)$ may be substituted by any other name $y$, as long as there are \textit{no free occurrences of $y$ in $e$}:

Consider:

$(\lambda x\ y\ .\ x\ y)\ y$ 
$
\rightarrow (\lambda x\ z\ .\ x\ z)\ y$ \hspace{1cm} $\alpha$ conversion
$
\rightarrow [y/x](\lambda z\ .\ x\ z)$ \hspace{1cm} $\beta$ reduction
$
\rightarrow (\lambda z\ .\ y\ z)$
$= y$ \hspace{1cm} $\eta$ reduction
Eta Reduction

Eta reductions allow one to remove “redundant lambdas”.

Suppose that $f$ is a closed expression (i.e., there are no free variables in $f$).

Then:

$$\left( \lambda x . f \, x \right) \, y \rightarrow f \, y \quad \text{β reduction}$$

So, $( \lambda x . f \, x )$ behaves the same as $f$!

Eta reduction says, whenever $x$ does not occur free in $f$, we can rewrite $( \lambda x . f \, x )$ as $f$. 
Normal Forms

A lambda expression is in normal form if it can no longer be reduced by beta or eta reduction rules.

Not all lambda expressions have normal forms!

\[ \Omega = (\lambda x . x x) (\lambda x . x x) \rightarrow [ (\lambda x . x x) / x ] (x x) \]
\[ = (\lambda x . x x) (\lambda x . x x) \quad \beta \text{ reduction} \]
\[ \rightarrow (\lambda x . x x) (\lambda x . x x) \quad \beta \text{ reduction} \]
\[ \rightarrow (\lambda x . x x) (\lambda x . x x) \quad \beta \text{ reduction} \]
\[ \rightarrow \ldots \]

Reduction of a lambda expression to a normal form is analogous to a Turing machine halting or a program terminating.
Evaluation Order

Most programming languages are strict, that is, all expressions passed to a function call are evaluated before control is passed to the function.

Most modern functional languages, on the other hand, use lazy evaluation, that is, expressions are only evaluated when they are needed.

Consider:

\[
\text{sqr } n = n \times n
\]

Applicative-order reduction:

\[
sqr (2+5) \Rightarrow sqr 7 \Rightarrow 7 \times 7 \Rightarrow 49
\]

Normal-order reduction:

\[
sqr (2+5) \Rightarrow (2+5) \times (2+5) \Rightarrow 7 \times (2+5) \Rightarrow 7 \times 7 \Rightarrow 49
\]
The Church-Rosser Property

“If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing normal-order and applicative order reduction), then all of these evaluation orders yield the same result.”

So, evaluation order “does not matter” in the lambda calculus.
Non-termination

However, applicative order reduction may not terminate, even if a normal form exists!

\[(\lambda x . y)((\lambda x . x x)(\lambda x . x x))\]

Applicative order reduction \quad Normal order reduction

\[\rightarrow (\lambda x . y)((\lambda x . x x)(\lambda x . x x)) \quad \rightarrow y\]

\[\rightarrow (\lambda x . y)((\lambda x . x x)(\lambda x . x x))\]

\[\rightarrow ...\]

Compare to the Haskell expression:

\[(\lambda x \rightarrow \lambda y \rightarrow x) \, 1 \, (5/0) \not\Rightarrow 1\]
Currying

Since a lambda abstraction only binds a single variable, functions with multiple parameters must be modelled as Curried higher-order functions.

As we have seen, to improve readability, multiple lambdas are suppressed, so:

$$\lambda x y . x = \lambda x . \lambda y . x$$
$$\lambda b x y . b x y = \lambda b . \lambda x . \lambda y . (b x) y$$
Representing Booleans

Many programming concepts can be directly expressed in the lambda calculus. \textit{Let us define:}

\[
\begin{align*}
\text{True} & \equiv \lambda x y . x \\
\text{False} & \equiv \lambda x y . y \\
\text{not} & \equiv \lambda b . b \text{False } \text{True} \\
\text{if } b \text{ then } x \text{ else } y & \equiv \lambda b x y . b x y
\end{align*}
\]

then:

\[
\begin{align*}
\text{not True} & = (\lambda b . b \text{False } \text{True })(\lambda x y . x ) \\
& \rightarrow (\lambda x y . x ) \text{False } \text{True} \\
& \rightarrow \text{False} \\
\text{if True then } x \text{ else } y & = (\lambda b x y . b x y ) (\lambda x y . x ) x y \\
& \rightarrow (\lambda x y . x ) x y \\
& \rightarrow x
\end{align*}
\]
Representing Tuples

Although tuples are not supported by the lambda calculus, they can easily be modelled as higher-order functions that “wrap” pairs of values.

n-tuples can be modelled by composing pairs ...

Define:

\[ \text{pair} \equiv (\lambda \, x \, y \, z. \, z \, x \, y) \]
\[ \text{first} \equiv (\lambda \, p. \, p \, \text{True}) \]
\[ \text{second} \equiv (\lambda \, p. \, p \, \text{False}) \]

then:

\[ (1, 2) = \text{pair} \, 1 \, 2 \]
\[ \rightarrow (\lambda \, z. \, z \, 1 \, 2) \]
\[ \text{first (pair} \, 1 \, 2) \rightarrow (\text{pair} \, 1 \, 2) \, \text{True} \]
\[ \rightarrow \text{True} \, 1 \, 2 \]
\[ \rightarrow 1 \]
Tuples as functions

In Haskell:

\[
\begin{align*}
t &= \lambda x \to \lambda y \to x \\
f &= \lambda x \to \lambda y \to y \\
pair &= \lambda x \to \lambda y \to \lambda z \to z \ x \ y \\
\text{first} &= \lambda p \to p \ t \\
\text{second} &= \lambda p \to p \ f \\
? \text{first} (\text{pair} \ 1 \ 2) &= 1 \\
? \text{first} (\text{second} (\text{pair} \ 1 \ (\text{pair} \ 2 \ 3))) &= 2
\end{align*}
\]
Representing Numbers

There is a “standard encoding” of natural numbers into the lambda calculus:

Define:

\[ 0 \equiv (\lambda x . x) \]
\[ \text{succ} \equiv (\lambda n . (\text{False}, n)) \]

then:

\[ 1 \equiv \text{succ} \ 0 \quad \rightarrow \quad (\text{False}, 0) \]
\[ 2 \equiv \text{succ} \ 1 \quad \rightarrow \quad (\text{False}, 1) \]
\[ 3 \equiv \text{succ} \ 2 \quad \rightarrow \quad (\text{False}, 2) \]
\[ 4 \equiv \text{succ} \ 3 \quad \rightarrow \quad (\text{False}, 3) \]

...
Working with numbers

We can define simple functions to work with our numbers.

Consider:

```markdown
iszero ≡ first
pred ≡ second
```

then:

```markdown
iszero 1 = first (False, 0) → False
iszero 0 = (λp.p True) (λx.x) → True
pred 1 = second (False, 0) → 0
```

What happens when we apply pred 0? What does this mean?
What you should know!

- Is it possible to write a Pascal compiler that will generate code just for programs that terminate?
- What are the alpha, beta and eta conversion rules?
- What is name capture? How does the lambda calculus avoid it?
- What is a normal form? How does one reach it?
- What are normal and applicative order evaluation?
- Why is normal order evaluation called lazy?
- How can Booleans, tuples and numbers be represented in the lambda calculus?
Can you answer these questions?

- How can name capture occur in a programming language?
- What happens if you try to program $\Omega$ in Haskell? Why?
- What do you get when you try to evaluate $(\text{pred } 0)$? What does this mean?
- How would you model negative integers in the lambda calculus? Fractions?
- Is it possible to model real numbers? Why, or why not?
8. Fixed Points

Overview

- Recursion and the Fixed-Point Combinator
- The typed lambda calculus
- The polymorphic lambda calculus
- A quick look at process calculi

References:

Recursion

Suppose we want to define *arithmetic operations* on our lambda-encoded numbers.

In Haskell we can program:

```haskell
plus n m
  | n == 0 = m
  | otherwise = plus (n-1) (m+1)
```

so we might try to “define”:

```
plus ≡ λ n m . iszero n m ( plus ( pred n ) ( succ m ) )
```

Unfortunately this is *not a definition*, since we are trying to *use plus before it is defined*. I.e., plus is free in the “definition”!
Recursive functions as fixed points

We can obtain a *closed expression* by *abstracting* over `plus`:

\[ rplus \equiv \lambda \text{plus} \ n \ m . \ \text{iszero} \ n \ m \ ( \ \text{plus} \ ( \ \text{pred} \ n \ ) \ ( \ \text{succ} \ m \ ) \ ) \]

`rplus` takes as its *argument* the actual `plus` function to use and returns as its result a definition of that function in terms of itself. In other words, if `fplus` is the function we want, then:

\[ rplus \ fplus \leftrightarrow fplus \]

I.e., we are searching for a *fixed point* of `rplus` ...
Fixed Points

A **fixed point** of a function $f$ is a value $p$ such that $f(p) = p$.

**Examples:**

- $\text{fact 1} = 1$
- $\text{fact 2} = 2$
- $\text{fib 0} = 0$
- $\text{fib 1} = 1$

Fixed points are not always “well-behaved”:

- $\text{succ n} = n + 1$

**What is a fixed point of $\text{succ}$?**
Fixed Point Theorem

Theorem:
Every lambda expression \( e \) has a fixed point \( p \) such that \( (e \; p) \leftrightarrow p \).

Proof: Let:
\[
Y = \lambda f . (\lambda x . f (x \; x)) \; (\lambda x . f (x \; x))
\]
Now consider:
\[
p \equiv Y \; e \rightarrow (\lambda x . e (x \; x)) \; (\lambda x . e (x \; x))
\]
\[
\rightarrow e ((\lambda x . e (x \; x)) \; (\lambda x . e (x \; x)))
\]
\[
= e \; p
\]

So, the “magical \( Y \) combinator” can always be used to find a fixed point of an arbitrary lambda expression.
How does \( Y \) work?

Recall the non-terminating expression

\[
\Omega \equiv (\lambda x . x x) (\lambda x . x x)
\]

\( \Omega \) loops endlessly \textit{without doing any productive work}.

Note that \((x x)\) represents the body of the “loop”.

We simply define \( Y \) to take an \textit{extra parameter \( f \)}, and \textit{put it into the loop}, passing it the body as an argument:

\[
Y \equiv \lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))
\]

\textit{So \( Y \) just inserts some productive work into the body of \( \Omega \).}
Using the Y Combinator

Consider:

\[ f \equiv \lambda x. \text{True} \]

then:

\[ Y \ f \rightarrow f \ (Y \ f) \quad \text{by FP theorem} \]

\[ = (\lambda x. \text{True}) \ (Y \ f) \]

\[ \rightarrow \text{True} \]

Consider:

\[ Y \ \text{succ} \rightarrow \text{succ} \ (Y \ \text{succ}) \quad \text{by FP theorem} \]

\[ \rightarrow (\text{False}, (Y \ \text{succ})) \]

What are succ and pred of \((\text{False}, (Y \ \text{succ}))\)? What does this represent?
Recursive Functions are Fixed Points

We seek a fixed point of:

\[ r_{plus} = \lambda \, \text{plus} \, n \, m \, . \, \text{iszero} \, n \, m \, ( \, \text{plus} \, ( \, \text{pred} \, n \, ) \, ( \, \text{succ} \, m \, ) \, ) \]

By the Fixed Point Theorem, we simply take:

\[ \text{plus} = Y \, r_{plus} \]

Since this guarantees that:

\[ r_{plus} \, \text{plus} \iff \text{plus} \]

as desired!
Unfolding Recursive Lambda Expressions

\[ \text{plus } 1 \ 1 = \ (Y \ rplus) \ 1 \ 1 \]

\[ \rightarrow rplus \ plus \ 1 \ 1 \quad \text{(NB: fp theorem)} \]

\[ \rightarrow \ iszero \ 1 \ 1 \ (\text{plus} \ (\text{pred} \ 1) \ (\text{succ} \ 1) \ ) \]

\[ \rightarrow \ False \ 1 \ (\text{plus} \ (\text{pred} \ 1) \ (\text{succ} \ 1) \ ) \]

\[ \rightarrow \ plus \ (\text{pred} \ 1) \ (\text{succ} \ 1) \]

\[ \rightarrow rplus \ plus \ (\text{pred} \ 1) \ (\text{succ} \ 1) \]

\[ \rightarrow \ iszero \ (\text{pred} \ 1) \ (\text{succ} \ 1) \]

\[ \quad \ (\text{plus} \ (\text{pred} \ (\text{pred} \ 1) \ ) \ (\text{succ} \ (\text{succ} \ 1) \ ) \ ) \]

\[ \rightarrow \ iszero \ 0 \ (\text{succ} \ 1) \ (...) \]

\[ \rightarrow \ True \ (\text{succ} \ 1) \ (...) \]

\[ \rightarrow \ succ \ 1 \]

\[ \rightarrow \ 2 \]
The Typed Lambda Calculus

There are many variants of the lambda calculus. The **typed lambda calculus** just decorates terms with type annotations:

**Syntax:** $e ::= x^{\tau} \mid e_1^{\tau_2 \to \tau_1} e_2^{\tau_2} \mid (\lambda x^{\tau_2}. e_1^{\tau_1})^{\tau_2 \to \tau_1}$

**Operational Semantics:**

- $\lambda x^{\tau_2}. e_1^{\tau_1} \equiv \lambda y^{\tau_2}. [y^{\tau_2}/x^{\tau_2}] e_1^{\tau_1}$  $y^{\tau_2}$ not free in $e_1^{\tau_1}$
- $(\lambda x^{\tau_2}. e_1^{\tau_1}) e_2^{\tau_2} \Rightarrow [e_2^{\tau_2}/x^{\tau_2}] e_1^{\tau_1}$ $x^{\tau_2}$ not free in $e_1^{\tau_1}$
- $\lambda x^{\tau_2}. (e_1^{\tau_1} x^{\tau_2}) \Rightarrow e_1^{\tau_1}$

**Example:**

$\text{True} \equiv (\lambda x^A. (\lambda y^B. x^A)^{B \to A})^{A \to (B \to A)}$
The Polymorphic Lambda Calculus

Polymorphic functions like "map" cannot be typed in the typed lambda calculus!

Need type variables to capture polymorphism:

$\beta$ reduction (ii): $(\lambda x^\nu . e_1^{\tau_1}) e_2^{\tau_2} \Rightarrow [\tau_2 / \nu ] [ e_2^{\tau_2}/x^\nu ] e_1^{\tau_1}$

Example:

$\text{True} \equiv (\lambda x^{\alpha} . (\lambda y^{\beta} . x^{\alpha})^{\beta \rightarrow \alpha})^{\alpha \rightarrow (\beta \rightarrow \alpha)}$

$\text{True}_{\alpha \rightarrow (\beta \rightarrow \alpha)}^{\beta \rightarrow \alpha} a^{A} b^{B} \rightarrow (\lambda y^{\beta} . a^{A})^{\beta \rightarrow A} b^{B}$

$\rightarrow a^{A}$
Hindley-Milner Polymorphism

Hindley-Milner polymorphism (i.e., that adopted by ML and Haskell) works by inferring the type annotations for a slightly restricted subcalculus: polymorphic functions.

If:
\[
\text{doubleLen ~len} \; \text{len'} \; \text{xs} \; \text{ys} = (\text{len} \; \text{xs}) + (\text{len'} \; \text{ys})
\]

then
\[
\text{doubleLen ~length} \; \text{length} \; \text{"aaa" [1,2,3]}
\]
is ok, but if
\[
\text{doubleLen'} \; \text{len} \; \text{xs} \; \text{ys} = (\text{len} \; \text{xs}) + (\text{len} \; \text{ys})
\]
then
\[
\text{doubleLen'} \; \text{length} \; \text{"aaa" [1,2,3]}
\]
is a type error since the argument \text{len} cannot be assigned a unique type!
Polymorphism and self application

Even the polymorphic lambda calculus is not powerful enough to express certain lambda terms.

Recall that both $\Omega$ and the Y combinator make use of “self application”:

$$\Omega = (\lambda x. \ x\ x) (\lambda x. \ x\ x)$$

What type annotation would you assign to $(\lambda x. \ x\ x)$?
Other Calculi

Many calculi have been developed to study the semantics of programming languages.

Object calculi: model inheritance and subtyping.
- lambda calculi with records

Process calculi: model concurrency and communication
- CSP, CCS, π calculus, CHAM, blue calculus

Distributed calculi: model location and failure
- ambients, join calculus
What you should know!

✎ Why isn’t it possible to express recursion directly in the lambda calculus?
✎ What is a fixed point? Why is it important?
✎ How does the typed lambda calculus keep track of the types of terms?
✎ How does a polymorphic function differ from an ordinary one?
Can you answer these questions?

- Are there more fixed-point operators other than $Y$?
- How can you be sure that unfolding a recursive expression will terminate?
- Would a process calculus be Church-Rosser?
9. Introduction to Denotational Semantics

Overview:

- Syntax and Semantics
- Approaches to Specifying Semantics
- Semantics of Expressions
- Semantics of Assignment
- Other Issues

References:

Defining Programming Languages

Three main characteristics of programming languages:

1. **Syntax**: What is the *appearance* and *structure* of its programs?

2. **Semantics**: What is the *meaning* of programs?
   The *static semantics* tells us which (syntactically valid) programs are semantically valid (i.e., which are *type correct*) and the *dynamic semantics* tells us how to interpret the meaning of valid programs.

3. **Pragmatics**: What is the *usability* of the language?
   How *easy is it to implement*? What kinds of applications does it suit?
Uses of Semantic Specifications

Semantic specifications are useful for language designers to communicate with implementors as well as with programmers.

A precise standard for a computer implementation:
How should the language be implemented on different machines?

User documentation: What is the meaning of a program, given a particular combination of language features?

A tool for design and analysis: How can the language definition be tuned so that it can be implemented efficiently?

Input to a compiler generator: How can a reference implementation be obtained from the specification?
Methods for Specifying Semantics

Operational Semantics:

- $[[\text{program}]] = \text{abstract machine program}$
- can be simple to implement
- hard to reason about

Denotational Semantics:

- $[[\text{program}]] = \text{mathematical denotation}$
  (typically, a function)
- facilitates reasoning
- not always easy to find suitable semantic domains

...
Methods for Specifying Semantics …

Axiomatic Semantics:
- $[[ \text{program} ]] = \text{set of properties}$
- good for proving theorems about programs
- somewhat distant from implementation

Structured Operational Semantics:
- $[[ \text{program} ]] = \text{transition system}$
  (defined using inference rules)
- good for concurrency and non-determinism
- hard to reason about equivalence
Concrete and Abstract Syntax

How to parse “4 * 2 + 1”?  

Abstract Syntax is compact but ambiguous:

Expr ::= Num | Expr Op Expr  
Op ::= + | - | * | /

Concrete Syntax is unambiguous but verbose:

Expr ::= Expr LowOp Term | Term  
Term ::= Term HighOp Factor | Factor  
Factor ::= Num | ( Expr )  
LowOp ::= + | -  
HighOp ::= * | /

Concrete syntax is needed for parsing; abstract syntax suffices for semantic specifications.
A Calculator Language

Abstract Syntax:

\[
\begin{align*}
\text{Prog} & ::= \ 'ON' \ \text{Stmt} \\
\text{Stmt} & ::= \ \text{Expr} \ 'TOTAL' \ \text{Stmt} \\
& \ | \ \text{Expr} \ 'TOTAL' \ 'OFF' \\
\text{Expr} & ::= \ \text{Expr}_1 \ '+' \ \text{Expr}_2 \\
& \ | \ \text{Expr}_1 \ '*' \ \text{Expr}_2 \\
& \ | \ 'IF' \ \text{Expr}_1 \ ',' \ \text{Expr}_2 \ ',' \ \text{Expr}_3 \\
& \ | \ 'LASTANSWER' \\
& \ | \ '(' \ \text{Expr} \ ')' \\
& \ | \ \text{Num}
\end{align*}
\]

The program “ON 4 * ( 3 + 2 ) TOTAL OFF” should print out 20 and stop.
Calculator Semantics

We need three semantic functions: one for programs, one for statements (expression sequences) and one for expressions.

The meaning of a program is the list of integers printed:

Programs:

\[ P : \text{Program} \rightarrow \text{Int} * \]

\[ P \lbrack \text{ON} \ S \rbrack = S \lbrack S \rbrack (0) \]

A statement may use and update LASTANSWER:

Statements:

\[ S :: \text{ExprSequence} \rightarrow \text{Int} \rightarrow \text{Int} * \]

\[ S \lbrack E \ \text{TOTAL} \ S \rbrack (n) = \text{let } n' = E \lbrack E \rbrack (n) \]

\[ \text{in cons}(n', S \lbrack S \rbrack (n')) \]

\[ S \lbrack E \ \text{TOTAL OFF} \rbrack (n) = [ E \lbrack E \rbrack (n) ] \]
Calculator Semantics...

Expressions:

\[ E : \text{Expression} \rightarrow \text{Int} \rightarrow \text{Int} \]

\[ E \left[ E_1 + E_2 \right] (n) = E \left[ E_1 \right] (n) + E \left[ E_2 \right] (n) \]

\[ E \left[ E_1 \times E_2 \right] (n) = E \left[ E_1 \right] (n) \times E \left[ E_2 \right] (n) \]

\[ E \left[ \text{IF} E_1, E_2, E_3 \right] (n) = \begin{cases} E \left[ E_2 \right] (n) & \text{if } E \left[ E_1 \right] (n) = 0 \\ \text{else} E \left[ E_3 \right] (n) & \end{cases} \]

\[ E \left[ \text{LASTANSWER} \right] (n) = n \]

\[ E \left[ (E) \right] (n) = E \left[ E \right] (n) \]

\[ E \left[ N \right] (n) = N \]
Semantic Domains

In order to define semantic mappings of programs and their features to their mathematical denotations, the semantic domains must be precisely defined:

```haskell
data Bool = True | False
(&&), (||) :: Bool -> Bool -> Bool
False && x = False
True && x = x
False || x = x
True || x = True

not :: Bool -> Bool
not True = False
not False = True
```
Data Structures for Abstract Syntax

We can represent programs in our calculator language as syntax trees:

```haskell
data Program = On ExprSequence
data ExprSequence = Total Expression ExprSequence
  | TotalOff Expression
data Expression = Plus Expression Expression
  | Times Expression Expression
  | If Expression Expression Expression
  | LastAnswer
  | Braced Expression
  | N Int
```
Representing Syntax

The test program “ON 4 * ( 3 + 2 ) TOTAL OFF” can be parsed as:

```
ON TOTAL OFF
```

And represented as:
```
test = On (TotalOff (Times (N 4)
  (Braced (Plus (N 3)
     (N 2))))))
```
Implementing the Calculator

We can implement our denotational semantics directly in a functional language like Haskell:

**Programs:**

\[ pp :: \text{Program} \rightarrow [\text{Int}] \]
\[ pp \ (\text{On} \ s) \quad = \ ss \ s \ 0 \]

**Statements:**

\[ ss :: \text{ExprSequence} \rightarrow \text{Int} \rightarrow [\text{Int}] \]
\[ ss \ (\text{Total} \ e \ s) \ n \quad = \ \text{let} \ n' = (\text{ee} \ e \ n) \]
\[ \text{in} \ n' : (ss \ s \ n') \]
\[ ss \ (\text{TotalOff} \ e) \ n \quad = \ (\text{ee} \ e \ n) : [ \ ] \]

...
Implementing the Calculator ...

Expressions:

\[ ee :: Expression \rightarrow Int \rightarrow Int \]

\[ ee \ (Plus \ e1 \ e2) \ n \ = \ (ee \ e1 \ n) \ + \ (ee \ e2 \ n) \]
\[ ee \ (Times \ e1 \ e2) \ n \ = \ (ee \ e1 \ n) \ * \ (ee \ e2 \ n) \]
\[ ee \ (If \ e1 \ e2 \ e3) \ n \]
\[ \quad | \quad (ee \ e1 \ n) \ == \ 0 \quad = \ (ee \ e2 \ n) \]
\[ \quad | \quad \text{otherwise} \quad = \ (ee \ e3 \ n) \]
\[ ee \ (LastAnswer) \ n \ = \ n \]
\[ ee \ (Braced \ e) \ n \ = \ (ee \ e \ n) \]
\[ ee \ (N \ num) \ n \ = \ num \]
A Language with Assignment

Prog ::= Cmd '.

Cmd ::= Cmd1 ';' Cmd2
| 'if' Bool 'then' Cmd1 'else' Cmd2
| Id '=' Exp

Exp ::= Exp1 '+' Exp2
| Id
| Num

Bool ::= Exp1 '=' Exp2
| 'not' Bool

Example:
“z := 1 ; if a = 0 then z := 3 else z := z + a .”

Input number initializes a; output is final value of z.
Representing abstract syntax trees

Data Structures:

\[
\begin{align*}
\text{data Program} & = \text{Dot Command} \\
\text{data Command} & = \text{CSeq Command Command} \\
& \quad | \quad \text{Assign Identifier Expression} \\
& \quad | \quad \text{If BooleanExpr Command Command} \\
\text{data Expression} & = \text{Plus Expression Expression} \\
& \quad | \quad \text{Id Identifier} \\
& \quad | \quad \text{Num Int} \\
\text{data BooleanExpr} & = \text{Equal Expression Expression} \\
& \quad | \quad \text{Not BooleanExpr} \\
\text{type Identifier} & = \text{Char}
\end{align*}
\]
An abstract syntax tree

Example:

“z := 1 ; if a = 0 then z := 3 else z := z + a .”

Is represented as:

\[
\text{Dot} \quad (\text{CSeq} (\text{Assign} 'z' (\text{Num} 1))
\quad (\text{If} (\text{Equal} (\text{Id} 'a') (\text{Num} 0))
\quad (\text{Assign} 'z' (\text{Num} 3))
\quad (\text{Assign} 'z' (\text{Plus} (\text{Id} 'z') (\text{Id} 'a'))))
\]
Modelling Environments

A store is a mapping from identifiers to values:

```haskell
type Store = Identifier -> Int
newstore :: Store
newstore id = 0

update :: Identifier -> Int -> Store -> Store
update id val store = store'
  where store' id'
    | id' == id = val
    | otherwise = store id'
```
**Functional updates**

*Example:*

\[
env1 = \text{update } 'a' \ 1 \ (\text{update } 'b' \ 2 \ (\text{newstore})) \\
env2 = \text{update } 'b' \ 3 \ env1
\]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>env1 'b'</td>
<td>2</td>
</tr>
<tr>
<td>env2 'b'</td>
<td>3</td>
</tr>
<tr>
<td>env2 'z'</td>
<td>0</td>
</tr>
</tbody>
</table>
Semantics of assignments

\[ \text{pp} :: \text{Program} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ \text{pp} \ (\text{Dot} \ c) \ n = (\text{cc} \ c \ (\text{update} \ 'a' \ n \ \text{newstore})) \ 'z' \]

\[ \text{cc} :: \text{Command} \rightarrow \text{Store} \rightarrow \text{Store} \]
\[ \text{cc} \ (\text{CSeq} \ c1 \ c2) \ s = \text{cc} \ c2 \ (\text{cc} \ c1 \ s) \]
\[ \text{cc} \ (\text{Assign} \ id \ e) \ s = \text{update} \ id \ (\text{ee} \ e \ s) \ s \]
\[ \text{cc} \ (\text{If} \ b \ c1 \ c2) \ s = \text{ifelse} \ (\text{bb} \ b \ s) \]
\[ \quad (\text{cc} \ c1 \ s) \ (\text{cc} \ c2 \ s) \]

...
Semantics of assignments …

\[
\begin{align*}
  \text{ee} & : \text{Expression} \rightarrow \text{Store} \rightarrow \text{Int} \\
  \text{ee} (\text{Plus} \ e_1 \ e_2) \ s & = (\text{ee} \ e_2 \ s) + (\text{ee} \ e_1 \ s) \\
  \text{ee} (\text{Id} \ \text{id}) \ s & = s \ \text{id} \\
  \text{ee} (\text{Num} \ n) \ s & = n \\
  \\
  \text{bb} & : \text{BooleanExpr} \rightarrow \text{Store} \rightarrow \text{Bool} \\
  \text{bb} (\text{Equal} \ e_1 \ e_2) \ s & = (\text{ee} \ e_1 \ s) \ == \ (\text{ee} \ e_2 \ s) \\
  \text{bb} (\text{Not} \ b) \ s & = \text{not} \ (\text{bb} \ b \ s) \\
  \\
  \text{ifelse} & : \text{Bool} \rightarrow \ a \rightarrow \ a \rightarrow \ a \\
  \text{ifelse} \ \text{True} \ x \ y & = x \\
  \text{ifelse} \ \text{False} \ x \ y & = y
\end{align*}
\]
Running the interpreter

src1 = "z := 1 ; if a = 0 then z := 3 else z := z + a ."
ast1 = Dot (CSeq
    (Assign 'z' (Num 1))
    (If (Equal (Id 'a') (Num 0))
        (Assign 'z' (Num 3))
        (Assign 'z' (Plus (Id 'z') (Id 'a')))))

pp ast1 10
⇒ 11
Practical Issues

Modelling:
- Errors and non-termination:
  - need a special “error” value in semantic domains
- Branching:
  - semantic domains in which “continuations” model “the rest of the program” make it easy to transfer control
- Interactive input
- Dynamic typing
- ...
Theoretical Issues

What are the denotations of lambda abstractions?
- need Scott’s theory of semantic domains

What is the semantics of recursive functions?
- need least fixed point theory

How to model concurrency and non-determinism?
- abandon standard semantic domains
- use “interleaving semantics”
- “true concurrency” requires other models ...
What you should know!

✎ What is the difference between syntax and semantics?
✎ What is the difference between abstract and concrete syntax?
✎ What is a semantic domain?
✎ How can you specify semantics as mappings from syntax to behaviour?
✎ How can assignments and updates be modelled with (pure) functions?
Can you answer these questions?

- Why are semantic functions typically higher-order?
- Does the calculator semantics specify strict or lazy evaluation?
- Does the implementation of the calculator semantics use strict or lazy evaluation?
- Why do commands and expressions have different semantic domains?
10. Logic Programming

Overview

- Facts and Rules
- Resolution and Unification
- Searching and Backtracking
- Recursion, Functions and Arithmetic
- Lists and other Structures
References

What is a Program?
A program is a database of facts (axioms) together with a set of inference rules for proving theorems from the axioms.

Imperative Programming:
- Program = Algorithms + Data

Logic Programming:
- Program = Facts + Rules

or
- Algorithms = Logic + Control
What is Prolog?

A Prolog program consists of facts, rules, and questions:

**Facts** are named relations between objects:

parent(charles, elizabeth).

% elizabeth is a parent of charles
female(elizabeth).
% elizabeth is female

**Rules** are relations (goals) that can be inferred from other relations (subgoals):

mother(X, M) :- parent(X,M), female(M).
% M is a mother of X
% if M is a parent of X and M is female
Prolog Questions

Questions are statements that can be answered using facts and rules:

?- parent(charles, elizabeth).
⇒ yes

?- mother(charles, M).
⇒ M = elizabeth
yes
Horn Clauses

Both rules and facts are instances of Horn clauses, of the form:

$$A_0 \text{ if } A_1 \text{ and } A_2 \text{ and } \ldots \text{ and } A_n$$

$A_0$ is the head of the Horn clause and “$A_1 \text{ and } A_2 \text{ and } \ldots \text{ and } A_n$” is the body.

Facts are just Horn clauses without a body:

- parent(charles, elizabeth) if True
- female(elizabeth) if True
- mother(X, M) if parent(X, M) and female(M)
Resolution and Unification

Questions (or goals) are answered by matching goals against facts or rules, unifying variables with terms, and backtracking when subgoals fail.

If a subgoal of a Horn clause matches the head of another Horn clause, resolution allows us to replace that subgoal by the body of the matching Horn clause.

Unification lets us bind variables to corresponding values in the matching Horn clause:

mother(charles, M)  
⇒  
parent(charles, M) and female(M)  
⇒  
{ M = elizabeth }  
True and female(elizabeth)  
⇒  
{ M = elizabeth }  
True and True
Prolog Databases

A Prolog database is a file of facts and rules to be “consulted” before asking questions:

female(anne).    parent(andrew, elizabeth).
female(diana).    parent(andrew, philip).
female(elizabeth). parent(anne, elizabeth).
family(anne, philip).
male(andrew).     parent(charles, elizabeth).
male(charles).    parent(charles, philip).
male(edward).     parent(edward, elizabeth).
male(harry).      parent(edward, philip).
male(philip).     parent(harry, charles).
male(william).    parent(harry, diana).
parent(william, charles).
parent(william, diana).
Simple queries

?- consult('royal').
⇒ yes

?- male(charles).
⇒ yes

?- male(anne).
⇒ no

?- male(mickey).
⇒ no

...
Queries with variables

You may accept or reject unified variables:

?- parent(charles, P).
▷ P = elizabeth <carriage return>
   yes

You may reject a binding to search for others:

?- male(X).
▷ X = andrew ;
   X = charles <carriage return>
   yes

Use anonymous variables if you don’t care:

?- parent(william, _).
▷ yes
Unification

Unification is the process of instantiating variables by *pattern matching*.

1. **A constant** unifies only with itself:
   
   ?- charles = charles.
   
   ✔ yes
   
   ?- charles = andrew.
   
   ✔ no

2. **An uninstantiated variable** unifies with anything:
   
   ?- parent(charles, elizabeth) = Y.
   
   ✔ Y = parent(charles,elizabeth) ?
   
   yes

...
Unification ...

3. A **structured term unifies** with another term only if it has the same function name and number of arguments, and the arguments can be unified recursively:

```
?- parent(charles, P) = parent(X, elizabeth).
▷ P = elizabeth,
   X = charles ?
   yes
```
Evaluation Order

In principle, any of the parameters in a query may be instantiated or not

?- mother(X, elizabeth).
  X = andrew ;
  X = anne ;
  X = charles ;
  X = edward ;
  no

?- mother(X, M).
  M = elizabeth,
  X = andrew ?
  yes
Closed World Assumption

Prolog adopts a *closed world assumption* — whatever cannot be proved to be true, is assumed to be false.

?- mother(elizabeth,M).
\[\Rightarrow \text{no}\]

?- male(mickey).
\[\Rightarrow \text{no}\]
Backtracking

Prolog applies resolution in linear fashion, replacing goals left to right, and considering database clauses top-to-bottom.

father(X, M) :- parent(X,M), male(M).

?- trace(father(charles,F)).

+ 1 1 Call: father(charles,_67) ?
+ 2 2 Call: parent(charles,_67) ?
+ 2 2 Exit: parent(charles,elizabeth) ?
+ 3 2 Call: male(elizabeth) ?
+ 3 2 Fail: male(elizabeth) ?
+ 2 2 Redo: parent(charles,elizabeth) ?
+ 2 2 Exit: parent(charles,philip) ?
+ 3 2 Call: male(philip) ?
+ 3 2 Exit: male(philip) ?
+ 1 1 Exit: father(charles,philip) ? ...
Comparison

The predicate = attempts to *unify* its two arguments:

?- X = charles.
\[ \Rightarrow X = \text{charles} \ ? \]

\[ \text{yes} \]

The predicate == tests if the terms instantiating its arguments are *literally identical*:

?- charles == charles.
\[ \Rightarrow \text{yes} \]

?- X == charles.
\[ \Rightarrow \text{no} \]

?- X = charles, male(charles) == male(X).
\[ \Rightarrow X = \text{charles} \ ? \]

\[ \text{yes} \]
Comparison ...

The predicate \( \neq \) tests if its arguments are *not* literally identical:

?- X = male(charles), Y = charles, X \( \neq \) male(Y).
\[ \Rightarrow \text{no} \]
Sharing Subgoals

Common subgoals can easily be factored out as relations:

\[
\text{Sibling (X, Y)} :- \text{mother (X, M), mother (Y, M), father (X, F), father (Y, F), X} \neq Y.
\]

\[
\text{Brother (X, B)} :- \text{Sibling (X, B), male (B).}
\]

\[
\text{Uncle (X, U)} :- \text{Parent (X, P), Brother (P, U).}
\]

\[
\text{Sister (X, S)} :- \text{Sibling (X, S), female (S).}
\]

\[
\text{Aunt (X, A)} :- \text{Parent (X, P), Sister (P, A).}
\]
Disjunctions

One may define *multiple rules* for the same predicate, just as with facts:

```prolog
isparent(C, P) :- mother(C, P).
isparent(C, P) :- father(C, P).
```

Disjunctions (“or”) can also be expressed using the “;” operator:

```prolog
isparent(C, P) :- mother(C, P); father(C, P).
```

Note that *same information* can be represented in *different* forms — we could have decided to express `mother/2` and `father/2` as facts, and `parent/2` as a rule. Ask:

- Which way is it easier to *express* and *maintain* facts?
- Which way makes it *faster* to *evaluate* queries?
Recursion

Recursive relations are defined in the obvious way:

\[
\begin{align*}
\text{ancestor}(X, A) &: \quad \text{parent}(X, A). \\
\text{ancestor}(X, A) &: \quad \text{parent}(X, P), \text{ancestor}(P, A).
\end{align*}
\]

?- trace(ancestor(X, philip)).

\begin{align*}
+ & \ 1 \ 1 \ \text{Call: } \text{ancestor}(_61,\text{philip}) \ ? \\
+ & \ 2 \ 2 \ \text{Call: } \text{parent}(_61,\text{philip}) \ ? \\
+ & \ 2 \ 2 \ \text{Exit: } \text{parent}(\text{andrew},\text{philip}) \ ? \\
+ & \ 1 \ 1 \ \text{Exit: } \text{ancestor}(\text{andrew},\text{philip}) \ ? \\
X & = \text{andrew} \ ? \\
\text{yes}
\end{align*}

Will ancestor/2 always terminate?
Recursion …

?- trace(ancestor(harry, philip)).

+ 1 1 Call: ancestor(harry,philip) ?
+ 2 2 Call: parent(harry,philip) ?
+ 2 2 Fail: parent(harry,philip) ?
+ 2 2 Call: parent(harry,_,316) ?
+ 2 2 Exit: parent(harry,charles) ?
+ 3 2 Call: ancestor(charles,philip) ?
+ 4 3 Call: parent(charles,philip) ?
+ 4 3 Exit: parent(charles,philip) ?
+ 3 2 Exit: ancestor(charles,philip) ?
+ 1 1 Exit: ancestor(harry,philip) ?

yes

What happens if you query ancestor(harry, harry)?
Evaluation Order

Evaluation of recursive queries is sensitive to the order of the rules in the database, and when the recursive call is made:

\[
\text{anc2}(X, A) :- \text{anc2}(P, A), \text{parent}(X, P).
\]
\[
\text{anc2}(X, A) :- \text{parent}(X, A).
\]

?- \text{trace(anc2(harry, X)).}

+ 1 1 Call: \text{anc2(harry, _67)} ?
  + 2 2 Call: \text{anc2(_325, _67)} ?
  + 3 3 Call: \text{anc2(_525, _67)} ?
  + 4 4 Call: \text{anc2(_725, _67)} ?
  + 5 5 Call: \text{anc2(_925, _67)} ?
  + 6 6 Call: \text{anc2(_1125, _67)} ?
  + 7 7 Call: \text{anc2(_1325, _67)} ? \text{abort}

{Execution aborted}
Failure

Searching can be controlled by **explicit failure**:

\[
\text{printall}(X) :- X, \text{print}(X), \text{nl}, \text{fail}.
\]
\[
\text{printall}(_).
\]

?- \text{printall(brother(_,_))}.
\[\Rightarrow\]
\[
\text{brother(\text{andrew},\text{charles})}
\]
\[
\text{brother(\text{andrew},\text{edward})}
\]
\[
\text{brother(\text{anne},\text{andrew})}
\]
\[
\text{brother(\text{anne},\text{charles})}
\]
\[
\text{brother(\text{anne},\text{edward})}
\]
\[
\text{brother(\text{charles},\text{andrew})}
\]
\[
\ldots
\]
**Cuts**

The *cut* operator (!) *commits* Prolog to a particular search path:

```prolog
parent(C,P) :- mother(C,P), !.
parent(C,P) :- father(C,P).
```

Cut says to Prolog:

“This is the right answer to this query. If later you are forced to backtrack, please do not consider any alternatives to this decision.”
Negation as failure

Negation can be implemented by a combination of cut and fail:

\[
\text{not}(X) :- X, !, \text{fail}. \quad \% \text{if } X \text{ succeeds, we fail} \\
\text{not}(_). \quad \% \text{if } X \text{ fails, we succeed}
\]
Changing the Database

The Prolog database can be modified dynamically by means of `assert` and `retract`:

```prolog
rename(X,Y) :- retract(male(X)),
            assert(male(Y)), rename(X,Y).
rename(X,Y) :- retract(female(X)),
              assert(female(Y)), rename(X,Y).
rename(X,Y) :- retract(parent(X,P)),
              assert(parent(Y,P)), rename(X,Y).
rename(X,Y) :- retract(parent(C,X)),
              assert(parent(C,Y)), rename(X,Y).
rename(_,_).
```
Changing the Database ...

?- male(charles); parent(charles, _).
⇒ yes
?- rename(charles, mickey).
⇒ yes
?- male(charles); parent(charles, _).
⇒ no

NB: With SICSTUS Prolog, such predicates must be declared dynamic:

:- dynamic male/1, female/1, parent/2.
Functions and Arithmetic

Functions are *relations* between *expressions* and *values*:

?- x is 5 + 6.
\[ \Rightarrow x = 11 \ ? \]

Is *syntactic sugar* for:

\[ \text{is}(X, +(5,6)) \]
Defining Functions

User-defined functions are written in a relational style:

\[
\text{fact}(0,1).
\]
\[
\text{fact}(N,F) :-
\begin{align*}
N & > 0, \\
N1 & \text{is } N - 1, \\
\text{fact}(N1,F1), \\
F & \text{is } N \times F1.
\end{align*}
\]

?- fact(10,F).
\rightarrow F = 3628800 ?
Lists

Lists are pairs of elements and lists:

<table>
<thead>
<tr>
<th>Formal object</th>
<th>Cons pair syntax</th>
<th>Element syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>.(a , [ ])</td>
<td>[ a</td>
<td>[ ] ]</td>
</tr>
<tr>
<td>.(a , .(b, [ ]))</td>
<td>[ a</td>
<td>[ b</td>
</tr>
<tr>
<td>.(a , .(b , .(c , [ ])))</td>
<td>[ a</td>
<td>[ b</td>
</tr>
<tr>
<td>.(a , b)</td>
<td>[ a</td>
<td>b ]</td>
</tr>
<tr>
<td>.(a , .(b , c))</td>
<td>[ a</td>
<td>[ b</td>
</tr>
</tbody>
</table>

Lists can be **deconstructed** using cons pair syntax:

?- [a,b,c] = [a|X].
⇒ X = [b,c]?
Pattern Matching with Lists

\[
in(X, [X \mid _]).
\]
\[
in(X, [\_ \mid L]) :- in(X, L).
\]

?- in(b, [a,b,c]).
\[\Rightarrow yes\]

?- in(X, [a,b,c]).
\[\Rightarrow X = a ? ;
X = b ? ;
X = c ? ;
no\]
Pattern Matching with Lists …

Prolog will automatically introduce new variables to represent unknown terms:

?- in(a, L).

\[ L = [ a \mid _A ] ? ; \]
\[ L = [ _A , a \mid _B ] ? ; \]
\[ L = [ _A , _B , a \mid _C ] ? ; \]
\[ L = [ _A , _B , _C , a \mid _D ] ? \]
yes
Inverse relations

A carefully designed relation can be used in many directions:

\[
\text{append}([\ ],L,L).
\]
\[
\text{append}([X|L1],L2,[X|L3]) :- \text{append}(L1,L2,L3).
\]

?- \text{append}([a],[b],X).
\(\Rightarrow\) X = [a,b]

?- \text{append}(X,Y,[a,b]).
\(\Rightarrow\) X = [ ] Y = [a,b] ;
\(\quad\) X = [a] Y = [b] ;
\(\quad\) X = [a,b] Y = [ ]
\(\Rightarrow\) yes
Exhaustive Searching

Searching for permutations:

```
perm([ ],[ ]).  
perm([C|S1],S2) :- perm(S1,P1),  
append(X,Y,P1), % split P1  
append(X,[C|Y],S2).
```

?- printall(perm([a,b,c,d],_)).  
\[perm([a,b,c,d],[a,b,c,d])\]  
\[perm([a,b,c,d],[b,a,c,d])\]  
\[perm([a,b,c,d],[b,c,a,d])\]  
\[perm([a,b,c,d],[b,c,d,a])\]  
\[perm([a,b,c,d],[a,c,b,d])\]  
...
Limits of declarative programming

A *declarative*, but hopelessly *inefficient* sort program:

\[
\begin{align*}
\text{ndsort}(L,S) & :\quad \text{perm}(L,S), \\
& \quad \text{issorted}(S).
\end{align*}
\]

\[
\begin{align*}
\text{issorted}([\_]). \\
\text{issorted}([\_\_]). \\
\text{issorted}([N,M|S]) :&\quad N \leq M, \\
& \quad \text{issorted}([M|S]).
\end{align*}
\]

*Of course, efficient solutions in Prolog do exist!*
What you should know!

- What are Horn clauses?
- What are resolution and unification?
- How does Prolog attempt to answer a query using facts and rules?
- When does Prolog assume that the answer to a query is false?
- When does Prolog backtrack? How does backtracking work?
- How are conjunction and disjunction represented?
- What is meant by “negation as failure”?
- How can you dynamically change the database?
Can you answer these questions?

❖ How can we view functions as relations?
❖ Is it possible to implement negation without either cut or fail?
❖ What happens if you use a predicate with the wrong number of arguments?
❖ What does Prolog reply when you ask `not(male(X)).`? What does this mean?
11. Applications of Logic Programming

Overview

- I. Search problems
  - SEND + MORE = MONEY
- II. Symbolic Interpretation
  - Definite Clause Grammars
  - Interpretation as Proof
  - An interpreter for the calculator language

Reference

I. Solving a puzzle

Find values for the letters so the following equation holds:

SEND
+MORE
-----
MONEY
A non-solution:

We would *like* to write:

```
soln0 :-
  A is 1000*S + 100*E + 10*N + D,
  B is 1000*M + 100*O + 10*R + E,
  C is 10000*M + 1000*O + 100*N + 10*E + Y,
  C is A+B,
  showAnswer(A,B,C).

showAnswer(A,B,C) :- writeln(['[A, '+', B, '+', C]].
writeln([]) :- nl.
writeln([X|L]) :- write(X), writeln(L).
```
A non-solution ...

?- soln0.

» evaluation_error: [goal(_1007 is 1000 * _1008 +
100 * _1009 + 10 * _1010 + _1011),
argument_index(2)]
[Execution aborted]

But this doesn’t work because “is” can only evaluate expressions over instantiated variables.

?- 5 is 1 + X.

» evaluation_error: [goal(5 is 1+_64),argument_index(2)]
[Execution aborted]
A first solution

So let's instantiate them first:

digit(0). digit(1). digit(2). digit(3). digit(4).
digits([]).
digits([D|L]) :- digit(D), digits(L).

% pick arbitrary digits:
soln1 :- digits([S,E,N,D,M,O,R,E,M,O,N,E,Y]),
        A is 1000*S + 100*E + 10*N + D,
        B is 1000*M + 100*O + 10*R + E,
        C is 10000*M + 1000*O + 100*N + 10*E + Y,
        C is A+B, % check if solution is found
        showAnswer(A,B,C).
A first solution ...

This is now correct, but yields a trivial solution!

\[ \text{soln1.} \]

\[ \Rightarrow 0 + 0 = 0 \]

\[ \text{yes} \]
A second (non-)solution

So let’s constrain S and M:

```prolog
soln2 :- digits([S,M]),
    not(S==0), not(M==0), % backtrack if 0
digits([N,D,M,O,R,E,M,O,N,E,Y]),
A is 1000*S + 100*E + 10*N + D,
B is 1000*M + 100*O + 10*R + E,
C is 10000*M + 1000*O + 100*N + 10*E + Y,
C is A+B,
showAnswer(A,B,C).
```
A second (non-)solution ...

Maybe it works. We'll never know ...

```
soln2.
⇒ [Execution aborted]
```

after 8 minutes still running ...

⚠️ What went wrong?
A third solution

Let's try to exercise more control by instantiating variables bottom-up:

```prolog
sum([],0).
sum([N|L], TOTAL) :- sum(L,SUBTOTAL),
    TOTAL is N + SUBTOTAL.
```

% Find D and C, where $\sum L$ is $D + 10*C$, digit(D)
```prolog
carrysum(L,D,C) :-
    sum(L,S), C is S/10, D is S - 10*C.
```

?- carrysum([5,6,7],D,C).
\[D = 8\]
\[C = 1\]
A third solution ...

We instantiate the final digits first, and use the carrysum to constrain the search space:

\[
\text{soln3 :- digits([D,E]), carrysum([D,E],Y,C1), digits([N,R]), carrysum([C1,N,R],E,C2), digit(O), carrysum([C2,E,O],N,C3), digits([S,M]), not(S==0), not(M==0), carrysum([C3,S,M],O,M)},
\]

A is 1000*S + 100*E + 10*N + D,
B is 1000*M + 100*O + 10*R + E,
C is A+B,
showAnswer(A,B,C).
A third solution ...

This is also correct, but uninteresting:

\[ \text{soln3.} \]

\[ 9000 + 1000 = 10000 \]

yes
A fourth solution

Let's try to make the variables unique:

% There are no duplicate elements in the argument list
unique([X|L]) :- not(in(X,L)), unique(L).
unique([]).

in(X, [X|_]).
in(X, [_|L]) :- in(X, L).

?- unique([a,b,c]).
⇒ yes
?- unique([a,b,a]).
⇒ no
A fourth solution ...

soln4 :- L1 = [D,E], digits(L1), unique(L1),
carrysum([D,E],Y,C1),
L2 = [N,R,Y|L1], digits([N,R]), unique(L2),
carrysum([C1,N,R],E,C2),
L3 = [0|L2], digit(O), unique(L3),
carrysum([C2,E,O],N,C3),
L4 = [S,M|L3], digits([S,M]),
    not(S==0), not(M==0), unique(L4),
carrysum([C3,S,M],O,M),
A is 1000*S + 100*E + 10*N + D,
B is 1000*M + 100*O + 10*R + E,
C is A+B,
showAnswer(A,B,C).
A fourth solution ...

This works (at last), in about 1 second on a G3 Powerbook.

\[ \text{soln4.} \]
\[ \Rightarrow 9567 + 1085 = 10652 \]
\[ \text{yes} \]
II. Symbolic Interpretation

Prolog is an ideal language for implementing small languages:

- Implement BNF using Definite Clause Grammars
- Implement semantic rules directly as Prolog rules
Goal-directed interpretation

Input string

List of tokens

Parse tree

Output value
Definite Clause Grammars

Definite clause grammars are an extension of context-free grammars.

A DCG rule in Prolog takes the general form:

\[ \text{head} \rightarrow \text{body}. \]

meaning “a possible form for head is body”.

The head specifies a non-terminal symbol, and the body specifies a sequence of terminals and non-terminals.
Definite Clause Grammars ...

- **Non-terminals** may be any Prolog *term* (other than a variable or number).

- A sequence of zero or more *terminal* symbols is written as a Prolog *list*. A sequence of ASCII characters can be written as a string.

- **Side conditions** containing Prolog goals may be written in { } brackets in the right-hand side of a grammar rule.
Example

This grammar parses an arithmetic expression (made up of digits and operators) and computes its value.

```
expr(Z) --> term(X), "+", expr(Y), \{Z is X + Y\}.
expr(Z) --> term(X), "-", expr(Y), \{Z is X - Y\}.
expr(X) --> term(X).

term(Z) --> number(X), "\*", term(Y), \{Z is X * Y\}.
term(Z) --> number(X), "/", term(Y), \{Z is X / Y\}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), \{C is -X\}.
number(X) --> [C], \{0'0=<C, C=<0'9, X is C - 0'0\}.
```
How to use this?

The query

\[
\text{?- expr(Z, "-2+3*5+1", []).}
\]

will compute \( Z = 14 \).
How does it work?

DCG rules are just syntactic sugar for normal Prolog rules.

\[
\text{expr}(Z) \rightarrow \text{term}(X), "+", \text{expr}(Y), \{Z \text{ is } X + Y\}.
\]

translates to:

\[
\text{expr}(Z, S0, S) :-
\begin{align*}
\text{term}(X, S0, S1), & \quad % \text{input and goal} \\
\text{'}C\text{'}(S1,43,S2), & \quad % \text{pass along state} \\
\text{expr}(Y, S2, S), & \\
Z \text{ is } X + Y .
\end{align*}
\]

'\text{'}C\text{'}\text{'} is a built-in predicate to recognize terminals.
Lexical analysis

We can use DCGs for both scanning and parsing.

Our lexer will convert an input atom into a list of tokens:

```prolog
lex(Atom, Tokens) :-
    name(Atom, String),
    scan(Tokens, String, []), !.

scan([T|Tokens]) -->
    whitespace0, token(T), scan(Tokens).
scan([]) --> whitespace0.
```
Recognizing Tokens

We will represent simple tokens by Prolog atoms:

token(on) --> "ON".
token(total) --> "TOTAL".
token(off) --> "OFF".
token(if) --> "IF".
token(last) --> "LASTANSWER".
token(',')) --> ",".
token('+') --> "+".
token('*') --> "*".
token('(') --> "(".
token(')') --> ")".
Recognizing Numbers

and a number \( N \) by the term \( \text{num}(N) \):

\[
\text{token}(\text{num}(N)) \rightarrow \text{digits}(DL), \{ \text{asnum}(DL, N, 0) \}.
\]

\[
\text{digits}([D|L]) \rightarrow \text{digit}(D), \text{digits}(L).
\]

\[
\text{digits}([D]) \rightarrow \text{digit}(D).
\]

\[
\text{digit}(D) \rightarrow [D], \{ "0" \leq D, D \leq "9" \}.
\]

اسلامة

\textit{How would you implement} \textit{asnum/3}?
Concrete Grammar

To parse a language, we need an unambiguous grammar!

\[
p ::= \text{`ON` } s \\
s ::= e \text{ `TOTAL` } s \\
| e \text{ `TOTAL` } \text{`OFF`} \\
e ::= \text{`IF` } e_1 , , , e_1 \\
| e_1 \\
e_1 ::= e_2 \text{`+`} e_1 \\
| e_2 \\
e_2 ::= e_3 \text{`*`} e_2 \\
| e_3 \\
e_3 ::= \text{`LASTANSWER`} \\
| \text{num} \\
| \text{`(`, } e_0 \text{`)'} \]
Parsing with DCGs

The concrete grammar is easily written as a DCG:

\[
\begin{align*}
\text{prog}(S) & : \text{[on]}, \text{stmt}(S). \\
\text{stmt}([E|S]) & : \text{expr}(E), \text{[total]}, \text{stmt}(S). \\
\text{stmt}([E]) & : \text{expr}(E), \text{[total, off]}. \\
\text{expr}(E) & : \text{e0}(E). \\
\text{e0}(\text{if}(\text{Bool}, \text{Then}, \text{Else})) & : \text{[if]}, \text{e1}(\text{Bool}), [','], \text{e1}(\text{Then}), [','], \text{e1}(\text{Else}). \\
\text{e0}(E) & : \text{e1}(E). \\
\text{e1}(\text{plus}(E1,E2)) & : \text{e2}(E1), ['+'], \text{e1}(E2). \\
\text{e1}(E) & : \text{e2}(E). \\
\text{e2}(\text{times}(E1,E2)) & : \text{e3}(E1), ['*'], \text{e2}(E2). \\
\text{e2}(E) & : \text{e3}(E). \\
\text{e3}(\text{last}) & : \text{[last]}. \\
\text{e3}(\text{num}(N)) & : \text{[num}(N)]. \\
\text{e3}(E) & : ['(', e0(E), ').'].
\end{align*}
\]
Representing Programs as Parse Trees

We have chosen to represent expressions as Prolog terms, and programs and statements as lists of terms:

```prolog
parse(Atom, Tree) :-
    lex(Atom, Tokens),
    prog(Tree, Tokens, []).
```

```prolog
parse(
    'ON (1+2)*(3+4) TOTAL LASTANSWER + 10 TOTAL OFF',
    [ times(plus(num(1),num(2)),
        plus(num(3),num(4))),
        plus(last,num(10))
    ])```

Testing

We exercise our parser with various test cases:

\[
\begin{align*}
\text{check}(\text{Goal}) & :- \text{Goal}, !. \\
\text{check}(\text{Goal}) & :- \\
& \quad \text{write('TEST FAILED: ')}, \\
& \quad \text{write(\text{Goal}), nl.}
\end{align*}
\]

\[
\begin{align*}
\text{parseTests} & :- \\
& \quad \text{check(parse('ON 0 TOTAL OFF', [\text{num}(0)])),} \\
& \quad \ldots
\end{align*}
\]
Interpretation as Proof

One can view the execution of a program as a step-by-step "proof" that the program reaches some terminating state, while producing output along the way.

- The program and its intermediate states are represented as structures (typically, as syntax trees).

- Inference rules express how one program state can be transformed to the next.
Building a Simple Interpreter

We define semantic predicates over the syntactic elements of our calculator language.

```
peval(S,L) :- seval(S, 0, L).

seval([E], Prev, [Val]) :- xeval(E, Prev, Val).
seval([E|S], Prev, [Val|L]) :- xeval(E, Prev, Val),
                             seval(S, Val, L).

xeval(num(N), _, N).
xeval(plus(E1,E2), Prev, V) :- xeval(E1, Prev, V1),
                              xeval(E2, Prev, V2),
                              V is V1+V2.
```

...
Running the Interpreter

The interpreter puts the parts together

eval(Expr, Val) :-
    parse(Expr, Tree),
    peval(Tree, Val).

eval('ON (1+2)*(3+4) TOTAL LASTANSWER + 10 TOTAL OFF', X).

⇒ X = [21, 31]
Testing the interpreter

We similarly define tests for the interpreter.

evalTests :-
    check(eval('ON 0 TOTAL OFF', [0])),
    check(eval('ON 5 + 7 TOTAL OFF', [12])),
    ...

A top-level script

Finally, we can package the interpreter as a ciao module, and invoke it from a script:

```
#!/bin/sh
exec ciao-shell $0 "$@" # -- mode: ciao; --
:- use_module(calc, [eval/2, test/0]).
main([]) :- test.
main(Argv) :- doForEach(Argv).
doForEach([]).
doForEach([Arg|Args]) :-
    write(Arg), nl,
    eval(Arg, Val),
    write(Val), nl,
    doForEach(Args).
```
What you should know!

- What are definite clause grammars?
- How are DCG specifications translated to Prolog?
- Why are abstract grammars inappropriate for parsing?
- Why are left-associative grammar rules problematic?
- How can we represent syntax trees in Prolog?
Can you answer these questions?

- What happens when we ask `digits([A,B,A])`?
- How many times will `soln2` backtrack before finding a solution?
- How would you check if the solution to the puzzle is unique?
- How would you generalize the puzzle solution to solve arbitrary additions?
- Why must DCG side conditions be put in `{ curly brackets }`?
- What exactly does the 'C' predicate do?
- Why do we need a separate lexer?
- How would you implement an interpreter for the assignment language we defined earlier?
12. Piccola — A Small Composition Language

Handouts will be distributed before the lecture.
13. Summary, Trends, Research ...

- Summary: functional, logic and object-oriented languages
- Research: ...

www.iam.unibe.ch/~scg
C and C++

Good for:
- systems programming
- portability

Bad for:
- learning (very steep learning curve)
- rapid application development
- maintenance

Trends:
- increased standardization
- generative programming
Functional Languages

Good for:
- equational reasoning
- declarative programming

Bad for:
- OOP
- explicit concurrency
- run-time efficiency (although constantly improving)

Trends:
- standardization: Haskell, “ML 2000”
- extensions (concurrency, objects): Facile, “ML 2000”, UFO ...
Lambda Calculus

Good for:
- simple, operational foundation for sequential programming languages

Bad for:
- programming

Trends:
- object calculi
- concurrent, distributed calculi (e.g., π calculus, “join” calculus ...)
Type Systems

Good for:
- catching static errors
- documenting interfaces
- formalizing and reasoning about domains of functions and objects

Bad for:
- reflection; self-modifying programs

Trends:
- automatic type inference
- reasoning about concurrency and other side effects
Polymorphism

Good for:
- parametric good for generic containers
- subtyping good for frameworks (generic clients)
- overloading syntactic convenience (classes in gopher, overloading in Java)
- coercion convenient, but may obscure meaning

Bad for:
- local reasoning
- optimization

Trends:
- combining subtyping, polymorphism and overloading
- exploring alternatives to subtyping ("matching")
Denotational Semantics

Good for:
- formally and unambiguously specifying languages
- sequential languages

Bad for:
- modelling concurrency and distribution

Trends:
- “Natural Semantics” (inference rules vs. equations)
- concurrent, distributed calculi
Logic Programming

Good for:
- searching (expert systems, graph & tree searching ...)
- symbolic interpretation

Bad for:
- debugging
- modularity

Trends:
- constraints
- concurrency
- modules
Object-Oriented Languages

**Good for:**
- domain modelling
- developing reusable frameworks

**Bad for:**
- learning (steep learning curve)
- understanding (hard to keep systems well-structured)
- semantics (no agreement)

**Trends:**
- component-based software development
- aspect-oriented programming
Scripting Languages

**Good for:**
- rapid prototyping
- high-level programming
- reflection; on-the-fly generation and evaluation of programs
- gluing components from different environments

**Bad for:**
- type-checking; reasoning about program correctness
- performance-critical applications

**Trends:**
- replacing programming as main development paradigm
- scriptable applications
- graphical “builders” instead of languages