Little Languages

and other programming paradigms

What is a language?

- "…a set of conventions for communicating an algorithm." - Horowitz
- But why just algorithms?
- HTML = hypertext markup language
- Tells browser what to do, but not exactly an *algorithm*
- In fact, browser has considerable smarts & retains considerable freedom.
- HTML is more like specifying ***input data***
  - to a generic webpage layout algorithm
  - to validators, style checkers, reformatters ...
  - to search engines and machine translation systems

LilyPond (`www.lilypond.org`)

c4

<<c4 d4 e4>>

{ f4 <<c4 d4 e4>> }

<< g2 \\ { f4 <<c4 d4 e4>> } >>

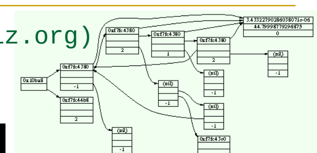
LilyPond (`www.lilypond.org`)

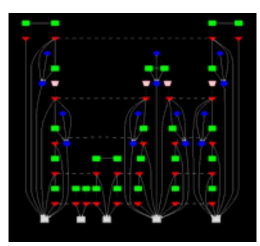
LilyPond (`www.lilypond.org`)

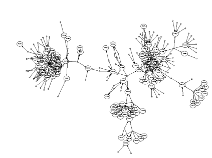
- Implemented in combo of C++, Scheme, LaTeX
- So it is built on top of another little language …
- Which is itself built on top of TeX
  - (an *extensible* little language: you can define new commands)
- Which is itself built in "literate Pascal" …

- Lilypond reminds me of MS BASIC: `play "c4.d8"`
  - Much better than TRS-80: `beep 278,12; beep 295, 4`
  - More generally, `play a$` where a$ is any string var – your program could *build* a$ at runtime!
- Other great thing about MS BASIC: `draw "u10r3d10l3"`

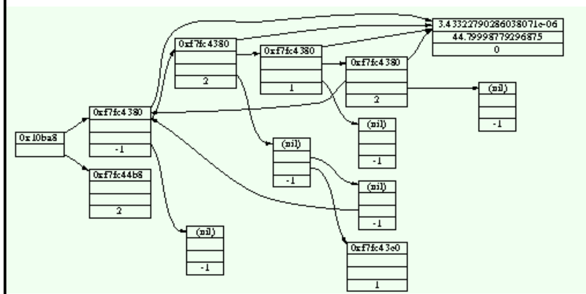
dot (`www.graphviz.org`)



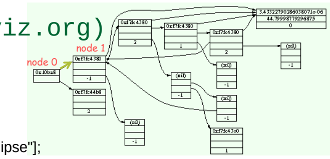




dot (www.graphviz.org)



dot (www.graphviz.org)




```
digraph g {
  graph [rankdir = "LR"];
  node [fontsize = "16“ shape = "ellipse"];
  edge [];

  "node0" [shape = "record“  label = "<f0> 0x10ba8 | <f1>"];
  "node1" [shape = "record“  label = "<f0> 0xf7fc4380 | <f1> | <f2> | -1"];
  …
  "node0":f0 -> "node1":f0 [id = 0];
  "node0":f1 -> "node2":f0 [id = 1];
  "node1":f0 -> "node3":f0 [id = 2];
  …
}
```

nodes

a little sub-language inside labels

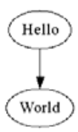
edges

What's the hard part? Making a nice layout!
Actually, it's NP-hard …

dot (www.graphviz.org)

Proof that it's really a language:

digraph G {Hello->World}



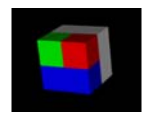
Running the compiler from the Unix shell (another language!)

echo "digraph G {Hello->World}" | **dot -Tpng** >hello.png

A little language for fractal cube graphics (embedded into Haskell)

```
u = 1.0 -- unit size
-- some basic coloured cubes to start with
redC   = XYZ .*. u $ shape red   Box{}
greenC = XYZ .*. u $ shape green Box{}
whiteC = XYZ .*. u $ shape white Box{}
```

```
((greenC .|. redC) .-. blueC) ./. whiteC
```



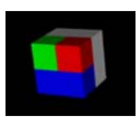
How is this defined?

Compiles into VRML (Virtual Reality Modeling Language)

slide thanks to Claus Reinke (modified)

A little language for fractal cube graphics (embedded into Haskell)

```
((greenC .|. redC) .-. blueC) ./. whiteC
```

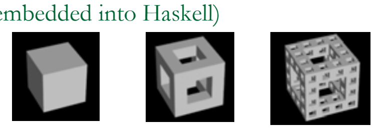


```
-- the cube combinators, rescaling to unit size;
-- a left of b, a on top of b, a before b
a .|. b = X .*. 0.5 $
     (X .+. (-0.5*u) $ a) .||. (X .+. (0.5*u)  $ b)
a .-. b = Y .*. 0.5 $
     (Y .+. (0.5*u)  $ a) .||. (Y .+. (-0.5*u) $ b)
a ./. b = Z .*. 0.5 $
     (Z .+. (0.5*u)  $ a) .||. (Z .+. (-0.5*u) $ b)
```

Compiles into VRML (Virtual Reality Modeling Language)

slide thanks to Claus Reinke (modified)

A little language for fractal cube graphics (embedded into Haskell)



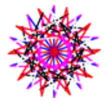
Needs recursion

```
rcube 0 = Cache "rcube0" $ shape white Box{}
rcube n = Cache ("rcube"++(show n)) $
      (s1 ./. s2) ./.  (s2 ./. s1)
  where
    s2  = (s11 .-. invisible) .-. (invisible .-. s11)
    s1  = (s12 .-. s11) .-. (s11 .-. s12)
    s11 = (white .|. invisible) .|. (invisible .|. white)
    s12 = (white .|. white) .|. (white .|. white)
    white = rcube (n-1)
```


slide thanks to Claus Reinke (modified)

Logo: A little(?) language for little people

- Created by Seymour Papert in 1968
  - Papert was first to see how computers could change learning
  - Had worked with the great Jean Piaget, studying children's minds
  - (Also, with Marvin Minsky, founded the MIT AI Lab and invented the first neural networks)
- Logo – a dialect of LISP
  - Fewer parentheses
  - Focus on graphics
  - Physical metaphor – robot turtles; kids could pretend to be turtles
  - Easy for kids to get started programming



Logo: A little(?) language for little people

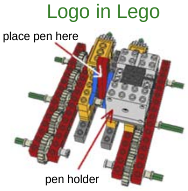
- Turtle talk (controlling a cursor with position, orientation, and drawing pen):
  - **forward *d*, backward *d***
  - **turnright *a*, turnleft *a***

slide thanks to Claus Reinke (modified)

Logo: A little(?) language for little people

- Turtle talk (controlling a cursor with position, orientation, and drawing pen):
  - **forward *d*, backward *d***
  - **turnright *a*, turnleft *a***
  - **pendown, penup**
  - **turnup *a*, turndown *a***
  - **spinright *a*, spinleft *a***
- Control structures:
  - **repeat *n cmds*, ifelse *c cmds cmds*,**
  - **to *procname params cmds*, procname**

Logo in Lego



slide thanks to Claus Reinke (modified)

Little languages: More examples (quick survey)

More little (or not so little) languages

(Do these describe algorithms or data?)

- The "units" program
  - You have: (1e-14 lightyears + 100 feet) / s
  - You want: furlongs per half fortnight
  - Answer: 376067.02 (other calculators are similar …)
- Regular expressions: pattern matching
  - b(c|de)*f – does it match bdedecf? overlap with (bd)*ef?
- Makefiles: running commands under certain conditions
  - Automatically determines order to run them (with parallelization)
- Lex and yacc: specify the format of another language!
  - Compiles into code for tokenizing and parsing that language
- Awk: process each line of a structured file
  - $2==$3 { sum += $0; print $0, sum }
  - Pattern { actions to perform on any line that matches pattern }

Protocols

- Programming languages are mainly used to deliver monologues
- But sometimes you talk to an application …
  - … and it talks back! Also in a structured language.
  - Compiler error messages? Not a great example.
- There are a lot of text-based protocols
- HTTP is one (and FTP before it)
  - You say to cs.jhu.edu: GET /holy/grail HTTP/1.0
  - cs.jhu.edu replies: 404 Not Found

Conversing with the sendmail daemon

- 220 blaze.cs.jhu.edu ESMTP Sendmail 8.12.9/8.12.9; Tue, 31 Jan 2006 11:06:02 -0500 (EST)
- helo emu.cs.jhu.edu
- 250 blaze.cs.jhu.edu Hello emu.cs.jhu.edu [128.220.13.179], pleased to meet you
- expn cs325-staff
- 250-2.1.5 Jason Eisner <jason@...>
- 250 2.1.5 Jason Smith <jrs026@...>
- quit
- 221 2.0.0 blaze.cs.jhu.edu closing connection
- Connection closed by foreign host.

Officially, what is a "little language"?

"A programming language tailored for a specific application domain: It is not general purpose, but rather captures precisely the semantics of the domain, no more and no less."

"The *ultimate abstraction* of an application domain; a language that you can teach to an intended user in less than a day."

"Hence, a clean notation for thinking about problems in the domain, and communicating them to other humans and to automatic solvers."

A user immersed in a domain *already knows* the domain semantics! All we need to do is provide a notation to express that semantics.

slide thanks to Tim Sheard (modified)

Some Application Domains

- Hardware description
- Silicon layout
- Text/pattern-matching
- Graphics and animation
- Computer music
- Distributed/Parallel comp.
- Databases
- Logic
- Security
- Scheduling
- Modeling
- Simulation
- Graphical user interfaces
- Lexing and parsing
- Symbolic computing
- Attribute grammars
- CAD/CAM
- Robotics

How many papers have you seen with a title such as: *"XXX: A Language for YYY"?*

slide thanks to Tim Sheard

Popular domain-specific languages

- Lex and Yacc (for program lexing and parsing)
- PERL (for text/file manipulation/scripting)
- VHDL (for hardware description)
- TeX and LaTex (for document layout)
- HTML/SGML (for document "markup")
- Postscript (for low-level graphics)
- Open GL (for high-level 3D graphics)
- Tcl/Tk (for GUI scripting)
- Macromedia Director (for multimedia design)
- Prolog (for logic)
- Mathematica/Maple (for symbolic computation)
- AutoLisp/AutoCAD (for CAD)
- Emacs Lisp (for editing)
- Excel Macro Language (for things nature never intended)

slide thanks to Tim Sheard

More domain-specific languages

- Stock market
  - composing contracts involving options
  - composing price history patterns
- Hardware specification languages (BlueSpec, Hawk, Lava,..)
- FRP (functional reactive programming)
  - Fran (animation), Frob (robotics), Fvision (computer vision)
  - FRP-based user interface libraries (FranTk, Frappe, Fruit,..)
  - Lula (stage lighting)
- VRML (virtual reality); XML (data interchange); HTML/CGI (web)
- SQL (database query language)
- Graphics (G-calculus, Pan, ..)
- Music (both sound and scores; Haskore, Elody,..)
- Parser combinators, pretty-printing combinators, strategy combinators for rewriting, GUI combinators (Fudgets, Haggis, ..)
- Attribute grammars
- Monads (a language "pattern")
- Coloured Petri Nets

slide thanks to Claus Reinke (modified)

Why user-centered languages matter

- Most programmers are not really programmers
  - They're teachers, engineers, secretaries, accountants, managers, lighting designers …
- Such programmers outnumber "professional" programmers by about 20 to 1.
  (Based on estimates of employment in particular fields, and the expected use of computers in those fields).
- The Ratio is only going to worsen.

slide thanks to Tim Sheard (modified)

What users are like (even techies!)

"Some people find it hard to understand why you can't simply add more and more graphical notation to a visual language.

For example, there have been many cases of people proposing (in private communication) all kinds and extensions to the language of statecharts.

These people could not understand why you can't just add a new kind of arrow that "means synchronization", or a new kind of box that "means separate-thread concurrency" ... It seemed to them that if you have boxes and lines and they mean things, you can add more and just say in a few words what they are intended to mean.

A good example of how difficult such additions can really be is the idea of having overlapping states in statecharts. ... [I]t took a lot of hard work to figure out a consistent syntax and semantics for such an extension. In fact, the result turned out to be too complex to justify implementation.

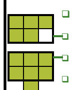
Nevertheless, people often ask why we don't allow overlapping ... It is very hard to convince them that it is not at all simple. One person kept asking this: "Why don't you just tell your system not to give me an error message when I draw these overlapping boxes?", as though the only thing that needs to be done is to remove the error message and you are in business!

- David Harel, Bernhard Rumpe, "Modeling Languages: Syntax, Semantics and All That Stuff; Part I: The Basic Stuff," 2000.

How would *you* build a new little language?

Designing a language …

- Useful
  - Easy for typical user to do what she needs; no "gotchas"; portable
- Elegant, learnable
  - Get everything by combining a few orthogonal concepts
  - Artificial limitations are bad: C/Pascal functions can't return arrays/records
  - Artificial extensions are bad: Perl has lots of magical special-case syntax
  - Is it good or bad to have lots of ways to do the same thing?



- Readable
  - syntax helps visualize the logical structure
- Supports abstraction
  - control abstractions: procedures, functions, etc.
  - data abstractions: interfaces, objects, modules
  - new programmer-defined abstractions?

slide thanks to James Montgomery (modified)

Leaving room for expansion

- Zawinski's Law:
  - "Every program attempts to expand until it can read mail.
  - Those programs which cannot so expand are replaced by ones which can."
- Similarly, every little language has users who start to want
  - arrays, pointers
  - loops
  - functions, local variables, recursion
  - library functions (random number generator, trig functions, …)
  - formatted I/O, filesystem access, web access, etc.

Leaving room for expansion: Options

- 1. Keep adding new syntax or library functions to your language
  - Spend rest of your life reinventing the wheel
- 2. Embed your language (from the start) in an existing real language
  - There are "host" languages *designed* to be extended: Lua, Tcl/Tk, …
    - Some general-purpose languages also support extension well enough
  - Your language automatically gets loops, local variables, etc.
  - It will look like the host language, with extra commands/operators
  - If you want to change the look a bit more, write a front-end preprocessor

  Example from before: Cube construction language was embedded into Haskell, with new operators .-., .|., ./. We used Haskell's recursion and local variables to construct complicated pictures.
- 3. *Don't* add to your language – keep it simple
  - User can work around limitations by *generating* code in your language
  - To loop *n* times, write a script to print *n* lines of code in your language
  - To generate random music, write a script to print MIDI or LilyPond

  Example from before: VRML doesn't have recursion, but we were able to use Haskell's recursion to generate a *long* VRML sequence.

Implementing your language (if not embedded)

How will the machine understand your language?

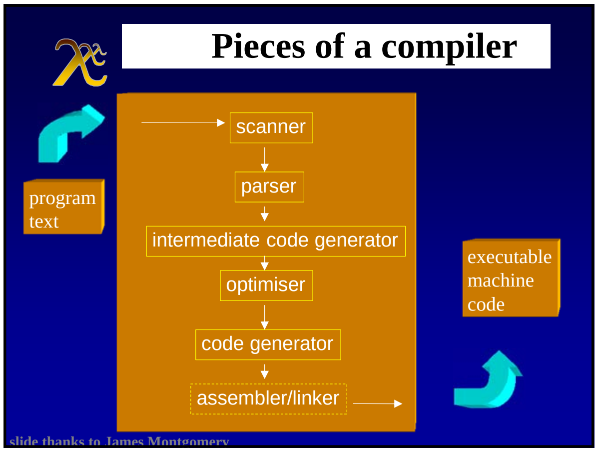
- Interpreter
  - Translates and executes your program, one line at a time
    - Lines 1-7 could define functions that are used in line 8
    - But line 8 is handled without knowledge of lines 9, 10, …
  - Starts producing output before it has seen the whole program
    - Helpful if the program is *very* long
    - Necessary if the user (a human or another program) wants to see output of line 7 before writing line 8
  - Examples
    - Interactive command-and-control languages: Unix shell, scripting languages,
    - Query languages: SQL, Prolog, …
    - Client-server protocols: HTTP, Dynagraph ("incrface"), …

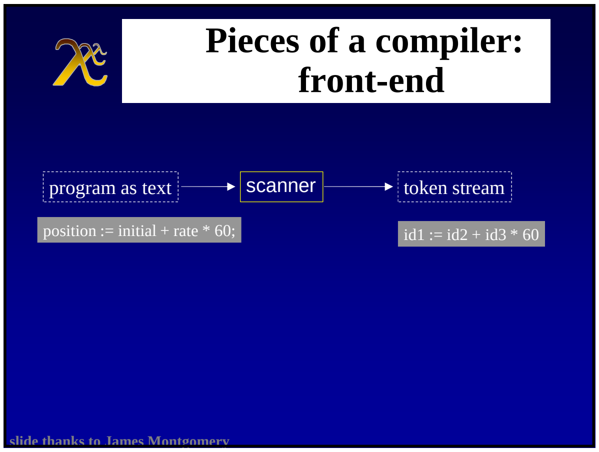
Implementing your language (if not embedded)

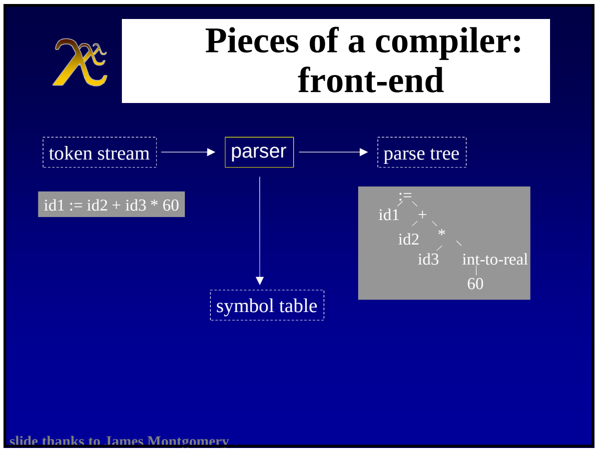
How will the machine understand your language?

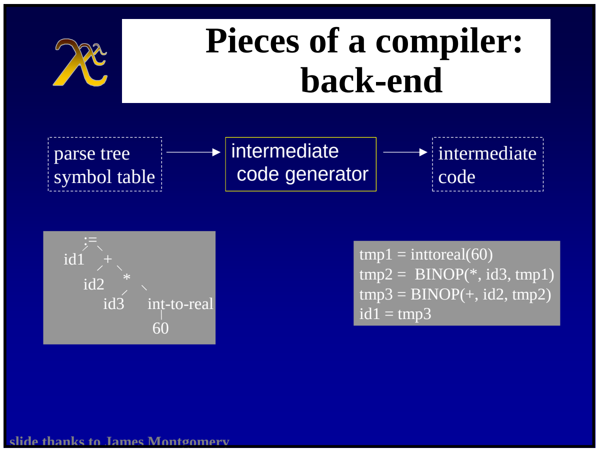
- Interpreter
- Compiler
  - Translates your *entire* program into a lower-level language
    - Can look at the whole program to understand line 8
    - Can rearrange or combine multiple lines for efficiency
    - Only has to translate it once
  - Lower-level language is then interpreted or compiled
    - Traditionally machine code, but could be VRML or C++ or …
  - Examples:
    - g++ compiles C++ into machine code, which is then interpreted by the chip
    - javac compiles Java into "Java bytecode," which is then interpreted by the Java Virtual Machine
    - dynac compiles Dyna into C++, which is then compiled by g++

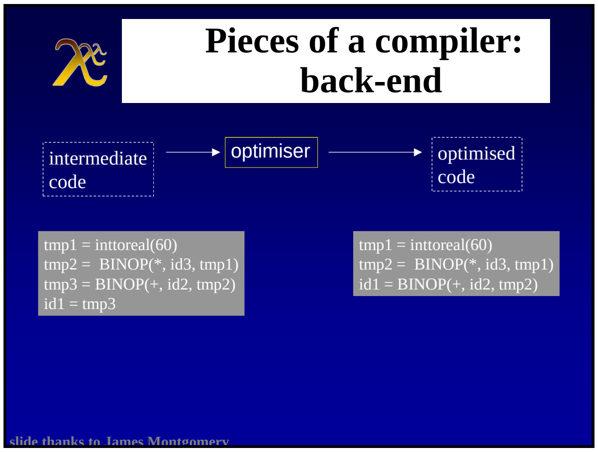
600.325/425 Declarative Methods - J. Eisner 31











Pieces of a compiler: back-end

optimized intermediate code → code generator → relocatable machine code

tmp1 = inttoreal(60)
tmp2 = BINOP(*, id3, tmp1)
id1 = BINOP(+, id2, tmp2)

movf id3, R3
mulf #60.0, R2
movf id2, R1
addf R2, R1
movf R1, id1

slide thanks to James Montgomery

Languages to help you build languages

A typical compiler pipeline:
- scanning (lexical analysis)
- parsing (syntax analysis)
- static analysis (types, scopes, ..)
- optimisation
- code generation

followed by a runtime system
- code execution
- memory management, ..

Regular expressions

BNF, or railroad diagrams

Inference rules

Attribute grammars

Control-flow graphs, data-flow graphs,..

Transformation Rules, rewriting

slide thanks to Claus Reinke (modified) 38

Languages to help you build languages

A typical compiler pipeline:

Regular expressions

Decent free tools have emerged for most of these steps
"Compiler construction kits"
(see course homepage)

- Very little languages may not need tools
  - As in your homework …
  - Just spend a couple hours writing a Perl script
- Tools are great for a more ambitious language
  - They free you up to focus on working the kinks out of the design – which still takes a lot of time

slide thanks to Claus Reinke (modified) 39

Oh yeah …

- So, Prof. Eisner, what are declarative methods??
  - A declarative program states only *what* is to be achieved
  - A procedural program describes explicitly *how* to achieve it
- Sorting in a declarative language
  - "Given array X, find an array Y such that
    - (1) Y is a permutation of X
    - (2) Y's elements are in increasing order"
  - Compiler is free to invent any sorting algorithm! (Hard?!)
  - You should be aware of when compiler will be efficient/inefficient
- Sorting in a procedural language
  - "Given array X, run through it from start to finish, swapping adjacent elements that are out of order …"
  - Longer and probably buggier
  - Never mentions conditions (1) and (2), except in comments

600.325/425 Declarative Methods - J. Eisner 40

Other ways to classify languages

- Declarative vs. procedural
- High-level vs. low-level (sort of the same thing)
- Domain-specific vs. general purpose
- Imperative vs. object-oriented vs. functional vs. logic
  - Ask me, or take 600.426 Programming Languages
- First-generation through sixth-generation
  - Browse web to learn history of programming languages

1st	Machine languages	4th	Application languages (4GLs)
2nd	Assembly languages	5th	AI techniques, inference
3rd	Procedural languages	6th	Neural networks (?), others….

table thanks to Grant Malcolm (modified)

600.325/425 Declarative Methods - J. Eisner 41