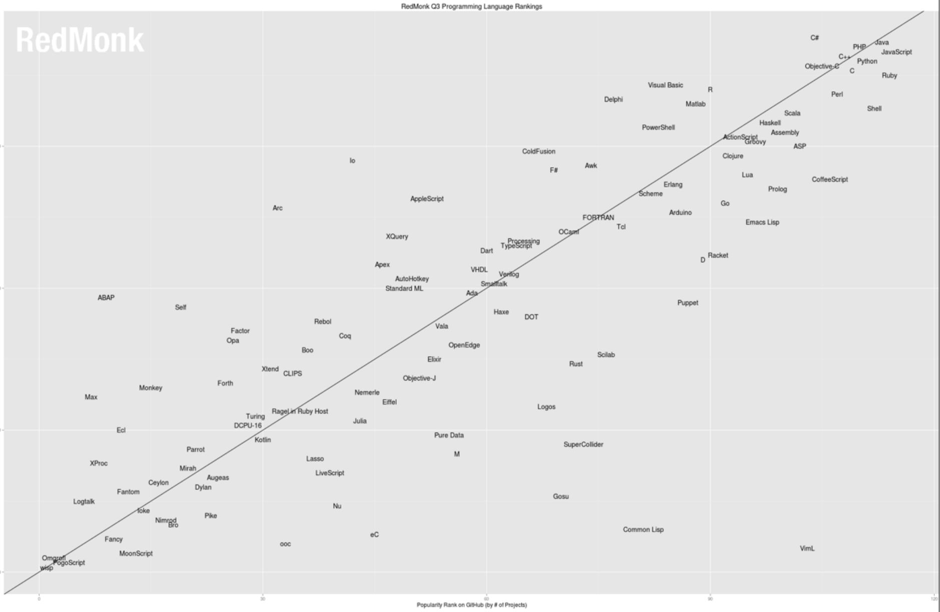
Lecture 8:
More Parsing & Types

CSC 131
Fall, 2012

Kim Bruce

Programming Language Ranking

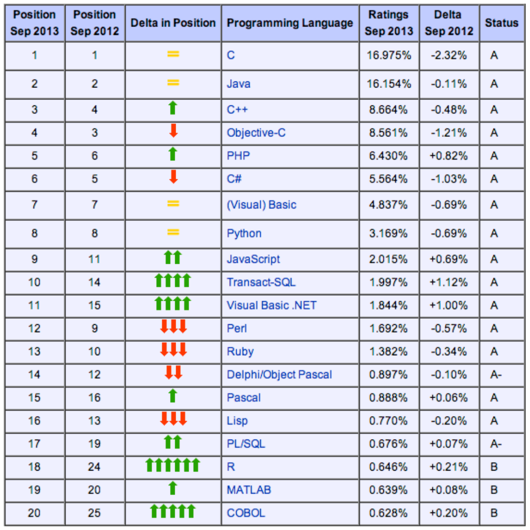


Top combined

1. Java *
2. JavaScript *
3. PHP *
4. Python *
5. Ruby *
6. C# *
7. C++ *
8. C *
9. Objective-C *
10. Shell *
11. Perl *
12. Scala
13. Assembly
14. Haskell
15. ASP
16. R
17. CoffeeScript
18. Groovy
19. Matlab
20. Visual Basic

http://redmonk.com/sogrady/2013/07/25/language-rankings-6-13/

Tiobe Index



Position Sep 2013	Position Sep 2012	Delta in Position	Programming Language	Ratings Sep 2013	Delta Sep 2012	Status
1	1	=	C	16.975%	-2.32%	A
2	2	=	Java	16.154%	-0.11%	A
3	4	↑	C++	8.664%	-0.48%	A
4	3	↓	Objective-C	8.561%	-1.21%	A
5	6	↑	PHP	6.430%	+0.82%	A
6	5	↓	C#	5.564%	-1.03%	A
7	7	=	(Visual) Basic	4.837%	-0.69%	A
8	8	=	Python	3.169%	-0.69%	A
9	11	↑↑	JavaScript	2.015%	+0.69%	A
10	14	↑↑↑↑	Transact-SQL	1.997%	+1.12%	A
11	15	↑↑↑↑	Visual Basic .NET	1.844%	+1.00%	A
12	9	↓↓↓	Perl	1.692%	-0.57%	A
13	10	↓↓↓	Ruby	1.382%	-0.34%	A
14	12	↓↓	Delphi/Object Pascal	0.897%	-0.10%	A-
15	16	↑	Pascal	0.888%	+0.06%	A
16	13	↓↓↓	Lisp	0.770%	-0.20%	A
17	19	↑↑	PL/SQL	0.676%	+0.07%	A-
18	24	↑↑↑↑↑↑	R	0.646%	+0.21%	B
19	20	↑	MATLAB	0.639%	+0.08%	B
20	25	↑↑↑↑↑	COBOL	0.628%	+0.20%	B

Scala #42
Haskell #46
ML #48
Go #49

Rewrite Grammar

```
      <exp> ::= <term> <termTail>                   (1)
 <termTail> ::= <addop> <term> <termTail>           (2)
              | ε                                   (3)
     <term> ::= <factor> <factorTail>               (4)
<factorTail> ::= <mulop> <factor> <factorTail>      (5)
              | ε                                   (6)
   <factor> ::= ( <exp> )                           (7)
              | NUM                                 (8)
              | ID                                  (9)
    <addop> ::= + | -                              (10)
    <mulop> ::= * | /                              (11)
```

No left recursion

How do we know which production to take?

Predictive Parsing

Goal: $a_1a_2\ldots a_n$

$S \to \alpha$

$\ldots$

$\to a_1a_2X\beta$

Want next terminal character derived to be a_3

a_3 in First(γ)

Need to apply a production $X ::= \gamma$ where

1) γ can eventually derive a string starting with a_3 or
2) If X can derive the empty string, then see if β can derive a string starting with a_3.

a_3 in Follow(X)

FIRST

- *Intuition*: $b \in \text{First}(X)$ iff there is a derivation $X \to^* b\omega$ for some ω.
- *Intuition:* A terminal $b \in \text{Follow}(X)$ iff there is a derivation $S \to^* vXb\omega$ for some v and ω.

First for Arithmetic

FIRST(<addop>) = { +, - }

FIRST(<mulop>) = { *, / }

FIRST(<factor>) = { (, NUM, ID }

FIRST(<term>) = { (, NUM, ID }

FIRST(<exp>) = { (, NUM, ID }

FIRST(<termTail>) = { +, -, ε }

FIRST(<factorTail>) = { *, /, ε }

Follow for Arithmetic

Only needed to calculate for <termTail>, <factorTail> !

FOLLOW(<exp>) = { EOF,) }

FOLLOW(<termTail>) = FOLLOW(<exp>) = { EOF,) }

FOLLOW(<term>) = FIRST(<termTail>) ∪
FOLLOW(<exp>) ∪ FOLLOW(<termTail>)
= { +, -, EOF,) }

FOLLOW(<factorTail>) = { +, -, EOF,) }

*FOLLOW(<factor>) = { *, /, +, -, EOF }*
FOLLOW(<addop>) = { (, NUM, ID }
FOLLOW(<mulop>) = { (, NUM, ID }

} *Not needed!*

Building Table

- Put X ::= α in entry (X,a) if either
  - a in First(α), or
  - e in First(α) and a in Follow(X)
- Consequence: X ::= α in entry (X,a) iff there is a derivation s.t. applying production can eventually lead to string starting with a.

Need Unambiguous

- No table entry should have more than one production to ensure unambiguous.
- Laws of predictive parsing:
  - If A ::= α_1 | …| α_n then for all $i \neq j$, First(α_i) ∩ First(α_j) = ∅.
  - If X →* ε, then First(X) ∩ Follow(X) = ∅.

See ParseArith.hs

Non-terminals	*ID*	*NUM*	*Addop*	*Mulop*	*(*	*)*	*EOF*
<exp>	1	1			1		
<termTail>			2			3	3
<term>	4	4			4		
<factTail>			6	5		6	6
<factor>	9	8			7		
<addop>			10				
<mulop>				11			

Read off from table which production to apply!

Alternatives to Recursive Descent Parsers

Table-Driven Stack-based Parser

- http://en.wikipedia.org/wiki/LL_parser
- Start with "S $" on stack and "input $" to be recognized.
- Use table to replace non-terminals on top of stack.
- If terminal on top of stack matches next input then erase both and proceed.
- Success if end up clearing stack and input
- Show with ID * (NUM + NUM)$

Another alternative

- LR(1) parsers -- bottom up, gives right-most derivation. Also stack-based.
- YACC is LR(1). ANTLR is LL(1).
- k in LL(k) and LR(k) indicates how many letters of look ahead are necessary -- e.g. length of strings in columns of table.
- Compiler writers are happiest with k=1 to avoid exponential blow-up of table. May have to rewrite grammars.

More Options

- Parser Combinators
  - Domain specific language for parsing.
  - Even easier to tie to grammar than recursive descent
  - Build into Haskell and Scala, definable elsewhere
    - Talk about when cover Scala

Parser Combinators in Scala

Syntax tree building code omitted

```
def multOp = ("*" | "/")
def addOp = ("+" | "-")
def factor = "(" ~> expr <~ ")" |  numericLit ^^ {...}
def term = factor ~ (factorTail*) ^^ {...}
def factorTail = multOp ~ factor ^^{...}
def expr  = term ~ (termTail*) ^^ {...}
def termTail = addOp ~ term ^^{...}
```

See Haskell Recursive Descent Parser, ParseArith.hs on web page

---

Types

Why (Static) Types?

- Increase readability
- Hide representation
- Detection of errors.
- Help disambiguate operators
- Compiler optimization. E.g. know where fields of record/struct are.
- Help ensure different components in separately compiled units will interoperate properly

Types & Constructors

- Built-in types - *primitive types (incl. size)*
- Aggregate types
- Mapping types
- Recursive types
- Sequence types - *files and strings (primitive?)*
- User-defined types

Aggregate Types

- Cartesian products (tuples)
- Records / Structs
- Union Types
  - C: typedef union {int i; float r;} utype
  - unsafe
  - Discriminated union safer
  - Haskell type defs safe

Discriminated Union: Ada

```
type geometric (Kind: (Triangle, Square) := Square) is
   record
      color : ColorType := Red ;
      case Kind of
         when Triangle =>
               pt1,pt2,pt3:Point;
         when Square =>
               upperleft : Point;
               length : INTEGER range 1..100;
      end case;
   end record;

ob1 : geometric -- default is Square
ob2 : geometric(Triangle) -- frozen, can't be changed
```

Kind is tag

Mappings

- Arrays
  - Static - *location & size frozen at compile time* (FORTRAN)
  - Semi-static - *size bound at compile time, location at invocation* (Pascal, C)
  - Dynamic - *size and location bound at creation* (ALGOL 60, Ada, Java)
  - Flex - *size and location can be changed any time* (Java vectors)
- Function Types - *update less efficient*
  - update f arg nuVal = fn x => if x = arg then nuVal else f x

Recursive Types

- In Haskell: data List = Nil | Cons (Integer, List)
- In C: struct list { int x; list *next; };
- Solutions to: list = { Nil } ∪ (int × list)
  - A. finite seqs of ints followed by Nil: e.g., (2,(5,Nil))
  - B. finite or infinite seqs: if finite then end w/ Nil
- Recursive eqn's always have a least solution
  - least fixed point!

Least Recursive Solutions

$$\begin{aligned}
list_0 &= \{Nil\} \\
list_1 &= \{Nil\} \cup (int \times list_0) \\
&= \{Nil\} \cup \{(n, Nil) | n \in int\} \\
list_2 &= \{Nil\} \cup (int \times list_1) \\
&= \{Nil\} \cup \{(n, Nil) | n \in int\} \cup \{(m, (n, Nil)) | m, n \in int\} \\
&\ldots \\
list &= \bigcup_n list_n
\end{aligned}$$

Some solutions inconsistent w/classical math!

User-Defined Types

- Named types
  - More readable
  - Easy to modify if localized
  - Factorization (why repeat same def?)
  - Added consistency checking if generative
- Enumeration types added to Java 5