Chapter 8: The Language Design Space

Aarne Ranta

Slides for the book "Implementing Programming Languages. An Introduction to Compilers and Interpreters", College Publications, 2012.

How simple can a language be?

Two minimal Turing complete languages: Lambda calculus, Brainfuck.

Criteria for a good programming language

Domain-specific languages

Approaching natural language

Concepts and tools for Assignment 6

Models of computation

In the 1930's, before electronic computers were built, mathematicians developed **models of computation**:

- **Turing Machine** (Alan Turing), similar to imperative programming.
- **Lambda Calculus** (Alonzo Church), similar to functional programming.
- **Recursive Functions** (Stephen Kleene), also similar to functional programming.

These models are equivalent: they cover exactly the same programs.

Turing-complete = equivalent to these models

They correspond to different styles, **programming paradigms**

The halting problem

Turing proved that a machine cannot solve all problems.

In particular, the **halting problem**: to decides for any given program and input if the program terminates with that input.

All general-purpose programming languages used today are Turing-complete. Hence its halting problem is undecidable.

Pure lambda calculus as a programming language*

A minimal Turing-complete language

The minimal definition needs just three constructs: variables, applications, and abstractions:

```
Exp ::= Ident | Exp Exp | "\" Ident "->" Exp
```

This language is called the **=pure lambda calculus**.

Everything else can be defined: integers, booleans, etc.

Church numerals

Church numerals: integers in pure lambda calculus

```
0 = \f -> \x -> x
1 = \f -> \x -> f x
2 = \f -> \x -> f (f x)
3 = \f -> \x -> f (f (f x))
...
```

A number n is a higher-order function that applies any function f, to any argument x, n times.

Addition:

```
PLUS = \m -> \n -> \f -> \x -> n f (m f x)
```

gives a function that applies f first m times and then n times.

Examples of addition

(Using operational semantics for more details!)

```
PLUS 2 3
= (\m -> \n -> \f -> \x -> n f (m f x))
      (\f -> \x -> f (f x)) (\f -> \x -> f (f (f x)))
= \f -> \x -> (\f -> \x -> f (f (f x)))
      f ((\f -> \x -> f (f x)) f x)
= \f -> \x -> (\f -> \x -> f (f (f x))) f (f (f x))
= \f -> \x -> f (f (f (f (f x))))
= 5
```

Multiplication : add n to 0 m times.

```
MULT = \m -> \n -> m (PLUS n) 0
```

Booleans and control structures

Church booleans:

```
TRUE  = \x -> \y -> x
FALSE = \x -> \y -> y
```

`TRUE` chooses the first argument, `FALSE` the second. (Notice that `FALSE` = `0`)

Conditionals (the first argument is expected to be a Boolean):

```
IFTHENELSE = \b -> \x -> \y -> b x y
```

The boolean connectives (are they lazy?):

```
AND = \a -> \b -> IFTHENELSE a b FALSE
OR  = \a -> \b -> IFTHENELSE a TRUE b
```

Recursion

To be fully expressive, we need recursion.

We cannot just write (for e.g. the factorial $n!$),

```
fact n = if x == 0 then 1 else n * fact (n - 1)
```

because the pure lambda calculus has no definitions (the ones above were just shorthands, where the "defined" constant does not appear).

Solution: **fix-point combinator**, also known as the **Y combinator**:

```
Y = \g -> (\x -> g (x x)) (\x -> g (x x))
```

This function has the property (exercise!)

```
Y g = g (Y g)
```

which means that `Y` iterates `g` infinitely many times.

Following the idea

```
fact = \n -> if x == 0 then 1 else n * fact (n - 1)
```

we define

```
FACT = Y (\f -> \n -> IFTHENELSE (ISZERO n) 1 (MULT n (f (PRED n))))
```

where we need `ISZERO` (equal to 0) and `PRED` (predecessor, i.e. $n - 1$)

```
ISZERO = \n -> n (\x -> FALSE) TRUE
PRED = \n -> \f -> \x -> n (\g -> \h -> h (g f)) (\u -> x) (\u -> u)
```

(Exercise: verify that `PRED 1` is 0).

Another Turing-complete language*

BF, **Brainfuck**, designed by Urban Müller based on the theoretical language **P"** by Corrado Böhm.

Goal: to create a Turing-complete language with the smallest possible compiler. Müller's compiler was 240 bytes in size.

BF has

- an array of bytes, initially set to zeros (30,000 bytes in the original definition)
- a byte pointer, initially pointing to the beginning of the array
- eight commands,
  - moving the pointer
  - changing the value at the pointer
  - reading and writing a byte
  - jumps backward and forward in code

The BF commands

>	increment the pointer
<	decrement the pointer
+	increment the byte at the pointer
-	decrement the byte at the pointer
.	output the byte at the pointer
,	input a byte and store it in the byte at the pointer
[	jump forward past the matching] if the byte at the pointer is 0
]	jump backward to the matching [unless the byte at the pointer is 0

All other characters are treated as comments.

Example BF programs

`char.bf`, displaying the ASCII character set (from 0 to 255):

```
.+[.+]
```

`hello.bf`, printing "Hello":

```
++++++++++                  Set counter 10 for iteration
[>+++++++>++++++++++<<-] Set up 7 and 10 on array and iterate
>++.                        Print 'H'
>+.                         Print 'e'
+++++++.                    Print 'l'
.                           Print 'l'
+++.                        Print 'o'
```

A BF compiler

Here defined via translation to C:

>	`++p;`
<	`--p;`
+	`++*p;`
-	`--*p;`
.	`putchar(*p);`
,	`*p = getchar();`
[	`while (*p) {`
]	}

The code is within a `main ()` function, initialized with

```
char a[30000];
char *p = a;
```

Criteria for a good programming language

Turing completeness might not be enough!

Other reasonable criteria:

- **Orthogonality**: small set of non-overlapping language constructs.
- **Succinctness**: short expressions of ideas.
- **Efficiency**: code that runs fast and in small space.
- **Clarity**: programs that are easy to understand.
- **Safety**: guard against fatal errors.

Criteria not always compatible: there are trade-offs.

Lambda calculus and BF satisfy orthogonality, but hardly the other criteria.

Rich languages such as Haskell and C++ have low orthogonality but are good for most other criteria.

In practice, different languages are good for different applications.

Even BF can be good - for reasoning about computability!

(There may also be languages that aren't good for any applications. And even good languages can be implemented in bad ways, let alone used in bad ways.)

Some trends

Toward more **structured programming** (from GOTOs to `while` loops to recursion).

Toward richer **type systems** (from bit strings to numeric types to structures to algebraic data types to dependent types).

Toward more **abstraction** (from character arrays to strings, from arrays to vectors and lists, from unlimited access to abstract data types).

Toward more **generality** (from cut-and-paste to macros to functions to polymorphic functions to first-class modules).

Toward more **streamlined syntax** (from positions and line numbers, keywords used as identifiers, `begin` and `end` markers, limited-size identifiers, etc, to a "C-like" syntax that can be processed with standard tools and defined in pure BNF).

Domain-specific languages

As different languages are good for different purposes, why not turn the perspective and create the best language for each purpose?

More or less equivalent names:

- **special-purpose languages**
- **minilanguages**
- **domain-specific languages**
- **DSL**'s

Examples

- Lex for lexers, Yacc for parsers;
- BNFC for compiler front ends;
- XML for structuring documents;
- `make` for specifying compilation commands;
- `bash` (a Unix shell) for working on files and directories;
- PostScript for printing documents;
- JavaScript for dynamic web pages.

Design questions for DSL's

- Imperative or declarative?
- Interpreted or compiled?
- Portable or platform-dependent?
- Statically or dynamically checked?
- Turing-complete or limited?
- Language or library?

Turing completeness

PostScript and JavaScript are Turing-complete DSL's

The price to pay:

- halting problem is undecidable
- no complexity guarantees for programs

E.g. BNFC is not Turing-complete: it can just defines LALR(1) grammars with linear parsing complexity (or, with a suitable back-end, context-free grammars with cubic complexity).

Embedded languages*

Embedded language = minilanguage that is a fragment of a larger **host language**

Advantages:

- It inherits the implementation of the host language.
- No extra training is needed for those who already know the host language.
- An unlimited access to "language extensions" via using the host language.

Disadvantages:

- One cannot reason about the embedded language independently of the host language.
- Unlimited access to host language can compromise safety, efficiency, etc.
- May be difficult to interface with other languages than the host language.
- Training programmers previously unfamiliar with the host language can have a large overhead.

Example: parser combinators in Haskell

An alternative to using a grammar formalisms: write recursive-descent parsers directly in Haskell

Clearer and more succinct than raw coding without the combinators (Chapter 3)

The basic operations of sequencing (`...`), union (`|||`), and literals (`lit`).

The power to deal with arbitrary context-free grammars, and even beyond, because they allow recursive definitions of parsing functions.

The next slide is a complete parser combinator library.

```
-- read input [a], return value b and the rest of the input
type Parser a b = [a] -> [(b,[a])]

-- sequence: combine two parser
(...) :: Parser a b -> Parser a c -> Parser a (b,c)
(p ... q) s = [((x,y),r) | (x,t) <- p s, (y,r) <- q t]

-- union: two alternative parsers
(|||) :: Parser a b -> Parser a b -> Parser a b
(p ||| q) s = p s ++ q s

-- recognize a token that satisfies a predicate
satisfy :: (a -> Bool) -> Parser a a
satisfy b (c:cs) = [(c,cs) | b c]
satisfy _ _ = []

-- literal: recognize a given string (special case of satisfy)
lit :: (Eq a) => a -> Parser a a
lit a = satisfy (==a)

-- semantic action: apply a function to parse results
(***) :: Parser a b -> (b -> c) -> Parser a c
(p *** f) s = [(f x,r) | (x,r) <- p s]
```

Example

The if/while language Chapter 3, with the BNF grammar

```
SIf.    Stm ::= "if" "(" Exp ")" Stm ;
SWhile. Stm ::= "while" "(" Exp ")" Stm ;
SExp.   Stm ::= Exp ;
EInt.   Exp ::= Integer ;
```

With the parser combinators:

```
pStm :: Parser String Stm
pStm = lit "if" ... lit "(" ... pExp ... lit ")" ... pStm ***
       (\ (_,(_,(e,(_,s)))) -> SIf e s)
  |||
     lit "while" ... lit "(" ... pExp ... lit ")" ... pStm ***
       (\ (_,(_,(e,(_,s)))) -> SWhile e s)
  |||
     pExp ... lit ";" *** (\ (e,_) -> SExp e)

pExp :: Parser String Exp
pExp = satisfy (all isDigit) *** (\i -> EInt (read i))
```

The abstract syntax datatypes must be defined manually:

```
data Stm = SIf Exp Stm | SWhile Exp Stm | SExp Exp
data Exp = EInt Integer
```

Pros and cons

- more code to be written than in BNFC
+ the run-time code is smaller

+ more expressive power than in BNFC (e.g. the copy language!)
- the parsers can be inefficient (exponential), even loop (because of backtracking and left recursion)
- no automatic diagnostics as in LALR(1)

Thus, the extended trade-offs.

Case study: BNFC as a domain-specific language

The first version BNFC in 2002, targeting Haskell with Happy and Alex.

In 2003, ported to Java, C, and C++.

In 2006, a part of the "stable" Debian Linux distriburion.

Since then, it has changed only minimally, mostly to preserve compatibility with the targeted tools, whenever these tools have had backward-incompatible changes.

The goals of BNFC

To implement exactly the idea that a parser returns an abstract syntax tree built from the trees for the nonterminal items by using the rule label.

To help programmers with the following goals:

- to save writing code (conciseness)
- to keep the different compiler modules in sync,
- to use the grammar as reliable documentation,
- to guarantee the symmetry of parser and pretty printer
- to be sure of the complexity of parsing,
- to be able to port the same grammar to different host languages.

Conciseness

Short programs are more reliable:

> *The number of bugs per line is independent of programming language.*

(Eric S. Raymond, *The Art of Unix Programming*)

Code size needed for the task accomplished by BNFC (the CPP language definition):

format	**CPP.cf**	**Haskell**	**Java 1.5**	**C++**	**raw C++**
files	1	9	55	12	12
lines	63	999	3353	5382	9424
chars	1548	28516	92947	96587	203659
chars target/src	1	18	60	62	132

Design decisions in BNFC

- *Imperative or declarative?* Declarative. BNFC just defines the grammar, and uses separate algorithms that work on all grammars.
- *Interpreted or compiled?* Compiled. BNFC code is translated to host language code.
- *Portable or platform-dependent?* Portable. BNFC is designed to work with many host languages.
- *Statically checked?* Yes, some consistency checks are made before generating the host language code. But more checks would be desirable, for instance, the check for LALR conflicts.
- *Turing-complete or limited?* Limited. The grammars are limited to a subset of context-free grammars.
- *Language or library?* Language, with its own syntax, semantics, and compiler. And the need of some learning.

Declarativity

The most important lesson from BNFC: declarativity makes it both succinct, portable, and predictable.

Using Happy, CUP, or Bison directly would be none of these, because any host language code can be inserted in the semantic actions.

Using BNFC for implementing languages

It is easy to get started with a new language.

A prototype with a dozen rules can be ready to run in five minutes.

It is good to start with a set of code examples, usable as unit and regression tests.

The examples should meaningful programs that make something useful with the tasks for which your language is designed.

Compile your grammar often and re-run it on the example set!

The price to pay

The languag must be **well-behaved**, in the sense that

- lexing must be strictly finite state;
- parsing must be strictly LALR(1);
- white space cannot be given any meaning, but it is just ignored in lexing (except in string literals).

Surprisingly many legacy languages are not quite "well-behaved"

- Haskell violates all the three restrictions
- Java and C are largely well-behaved
- full C++ requires a more powerful parser than LR(k) for any k

Layout syntax (Haskell and Python) makes a language non-context-free!

BNFC however supports a restricted layout syntax.

Alpha convertibility

Alpha conversion = changing a variable name (in all occurrences).

This should not change the program behaviour.

Counterexample in Haskell:

```
eval e = case e of EAdd x y -> eval x + eval y
                   EMul x y -> eval x * eval y
```

If you rename `e` to `exp`, the code gets a syntax error!

Compiling natural language*

Natural language is the ultimate limit of bringing a language close to humans.

Automatic translation from Russian to English was one of the first computer applications in the late 1940's

- initial idea: Russian is encrypted English - one needs just break the code
- more difficult than expected - as hard as Artificial Intelligence in general!
- tools like Google translate succeed, but by compromising quality

Human-computer interaction (**HCI**)

- restricted language, high quality
- e.g. speech-based interaction with a car's navigation system

Case study: a query language*

Example interactions:

> *Is any even number prime?*
> Yes.
>
> *Which numbers greater than 100 and smaller than 150 are prime?*
> 101, 103, 107, 109, 113, 127, 131, 137, 139, 149.

Cf. **Wolfram Alpha**, "the computational knowledge engine".

A BNF grammar for a simple query language

```
-- general part

  QWhich.     Query    ::= "which" Kind "is" Property ;
  QWhether.   Query    ::= "is" Term Property ;
  TAll.       Term     ::= "every" Kind ;
  TAny.       Term     ::= "any" Kind ;
  PAnd.       Property ::= Property "and" Property ;
  POr.        Property ::= Property "or"  Property ;
  PNot.       Property ::= "not" Property ;
  KProperty.  Kind     ::= Property Kind ;

-- specific part

  KNumber.    Kind     ::= "number" ;
  TInteger.   Element  ::= Integer ;
  PEven.      Property ::= "even" ;
  POdd.       Property ::= "odd" ;
  PPrime.     Property ::= "prime" ;
  PDivisible. Property ::= "divisible" "by" Term ;
  PSmaller.   Property ::= "smaller" "than" Term ;
  PGreater.   Property ::= "greater" "than" Term ;
```

Bugs in the grammar

It only has singular forms of nouns:

- *which number is prime*, but not *which numbers are prime*

Properties placed before kinds:

- *greater than 3 number*, should be *number greater than 3*.

We could solve both issues by more categories and rules, but this would clutter the abstract syntax.

Grammatical Framework, GF*

A grammar formalism inspired by compiler construction but designed for natural language grammars.

GF has been applied to dozens of languages ranging from European languages like English and Dutch to Nepali, Swahili, and Thai.

GF is enables writing a translator by just writing a grammar.

GF grammar = abstract syntax + concrete syntaxes

Translation = parsing with one concrete syntax + linearization with another

Example: a compiler for arithmetic expressions

```
abstract Arithm = {
  cat Exp ;
  fun EInt : Int -> Exp ;
  fun EMul : Exp -> Exp -> Exp ;
  }
concrete ArithmJava of Arithm = {
  lincat Exp = Str ;
  lin EInt i = i.s ;
  lin EMul x y = x ++ "*" ++ y ;
  }
concrete ArithmJVM of Arithm = {
  lincat Exp = Str ;
  lin EInt i = "bipush" ++ i.s ;
  lin EMul x y = x ++ y ++ "imul" ;
  }
```

Using the grammar

Save the three modules in three `.gf` files. Then invoke `gf` with

```
gf ArithmJava.gf ArithmJVM.gf
```

In the shell that opens, you use a pipe to parse from Java and linearize to JVM:

```
> parse -lang=Java -cat=Exp "7 * 12" | linearize -lang=JVM
bipush 7 bipush 12 imul
```

Notice that the Java grammar is ambiguous: `7 * 12 * 9` has two parse trees. GF returns them both and produces two JVM expressions.

This ambiguity could be solved by using precedences. But in natural language, ambiguous grammars cannot be escaped.

How GF works

Separate abstract and concrete syntax: the BNF rule

```
EMul. Exp ::= Exp "*" Exp
```

says two things at the same time:

- `EMul` is a tree-building function that takes two `Exp` trees and forms an `Exp` tree.
- The tree is linearized to a sequence of tokens where `*` appears between the expressions.

In GF, they become two separate rules: a `fun` (function) rule and a `lin` (linearization) rule:

```
fun EMul : Exp -> Exp -> Exp
lin EMul x y = x ++ "*" ++ y
```

The rules are put into separate `abstract` and `concrete` modules.

Linearization types

A similar distinction applies to categories (`cat`) and their linearization types (`lincat`)

```
cat Exp ;
lincat Exp = Str ;
```

This means: expressions are linearized to strings.

But `lincats` can be different in different languages.

Example: in Java, expressions should be linearized to strings with precedences:

```
lincat Exp = {s : Str ; p : Prec} ;
```

This is a **record** with two **fields**: a string `s` and a precedence parameter `p`.

The full precedence example

```
concrete ArithmJava of Arithm = {
  param Prec = P0 | P1 ;
  lincat Exp = {s : Str ; p : Prec} ;
  lin EInt i = {s = i.s ; p = P1} ;
  lin EMul x y = {
    s = x.s ++ "*" ++
        case y.p of {P0 => parenth y.s ; P1 => y.s} ;
    p = P0
    } ;
  oper parenth : Str -> Str = \s -> "(" ++ s ++ ")" ;
  }
```

Here two precedence levels are enough.

Some history

GF and BNFC are genetically related.

GF was first released in 1998.

It was powerful enough for programming language implementation.

But not ideal for this, because the grammar format is *too* powerful and therefore potentially inefficient.

BNFC is, in a way, a special version of GF, which uses the much simpler BNF notation and converts the grammars to standard compiler tools.

Quiz: how to define the copy language in GF?

A GF grammar for queries*

Separate a base `Query` grammar and its `Math` extension.

Solve the bugs in the previous BNF grammars by using parameters.

Two concrete syntaxes: English and Haskell.

Abstract syntax, base part

```
abstract Query = {
flags startcat = Query ;
cat
  Query ;
  Kind ;
  Property ;
  Term ;
fun
  QWhich   : Kind -> Property -> Query ;     -- which numbers are prime
  QWhether : Term -> Property -> Query ;     -- is any number prime
  TAll : Kind -> Term ;                      -- all numbers
  TAny : Kind -> Term ;                      -- any number
  PAnd : Property -> Property -> Property ; -- even and prime
  POr  : Property -> Property -> Property ; -- even or odd
  PNot : Property -> Property ;              -- not prime
  KProperty : Property -> Kind -> Kind ;     -- even number
}
```

Abstract syntax, math part

The `MathQuery` module inherits all categories and functions of `Query` and adds some of its own.

```
abstract MathQuery = Query ** {
fun
  KNumber : Kind ;
  TInteger : Int -> Term ;
  PEven, POdd, PPrime : Property ;
  PDivisible : Term -> Property ;
  PSmaller, PGreater : Term -> Property ;
}
```

English concrete syntax

Use a number and fixity (prefix/postfix) parameter

```
concrete QueryEng of Query = {
lincat
  Query = Str ;
  Kind = Number => Str ;
  Property = {s : Str ; p : Fix} ;
  Term = {s : Str ; n : Number} ;
param
  Fix = Pre | Post ;
  Number = Sg | Pl ;
lin
  QWhich kind property = "which" ++ kind ! Pl ++ be ! Pl ++ property.s ;
  QWhether term property = be ! term.n ++ term.s ++ property.s ;
  TAll kind = {s = "all" ++ kind ! Pl ; n = Pl} ;
```

```
  TAny kind = {s = "any" ++ kind ! Sg ; n = Sg} ;
  PAnd p q = {s = p.s ++ "and" ++ q.s ; p = fix2 p.p q.p} ;
  POr p q = {s = p.s ++ "or" ++ q.s ; p = fix2 p.p q.p} ;
  PNot p = {s = "not" ++ p.s ; p = Post} ;
  KProperty property kind = \\n => case property.p of {
    Pre  => property.s ++ kind ! n ;
    Post => kind ! n ++ property.s
    } ;
oper
  be : Number => Str = table {Sg => "is" ; Pl => "are"} ;
  prefix : Str -> {s : Str ; p : Fix}  = \s -> {s = s ; p = Pre} ;
  postfix : Str -> {s : Str ; p : Fix}  = \s -> {s = s ; p = Post} ;
  fix2 : Fix -> Fix -> Fix = \x,y -> case x of {Post => Post ; _ => y} ;
}
```

```
concrete MathQueryEng of MathQuery = QueryEng ** {
lin
  KNumber = table {Sg => "number" ; Pl => "numbers"} ;
  TInteger i = {s = i.s ; n = Sg} ;
  PEven = prefix "even" ;
  POdd = prefix "odd" ;
  PPrime = prefix "prime" ;
  PDivisible term = postfix ("divisible by" ++ term.s) ;
  PSmaller term = postfix ("smaller than" ++ term.s) ;
  PGreater term = postfix ("greater than" ++ term.s) ;
}
```

The answering engine*

Denotational semantics: translate to sets, functions, etc:

$(QWhich\ kind\ prop)^* = \{x | x \in kind^*, prop^*(x)\}$

$(QWhether\ term\ prop)^* = term^*(prop^*)$

$(TAll\ kind)^* = \lambda p.(\forall x)(x \in kind^* \supset p(x))$

$(TAny\ kind)^* = \lambda p.(\exists x)(x \in kind^* \& p(x))$

$(TAnd\ p\ q)^* = \lambda x.p^*(x) \& q^*(x)$

$(TOr\ p\ q)^* = \lambda x.p^*(x) \vee q^*(x)$

$(TNot\ p)^* = \lambda x. \sim p^*(x)$

$(KProperty\ prop\ kind)^* = \{x | x \in kind^*, prop^*(x)\}$

$(TInteger\ i)^* = \lambda p.p^*(i)$

Denotational semantics as translation to Haskell

```
concrete QueryHs of Query = {
lincat
  Query, Kind, Property, Term, Element = Str ;
lin
  QWhich kind prop = "[x | x <-" ++ kind ++ "," ++ prop ++ "x" ++ "]" ;
  QWhether term prop = term ++ prop ;
  TAll kind = parenth ("\\p -> and [p x | x <-" ++ kind ++ "]") ;
  TAny kind = parenth ("\\p -> or  [p x | x <-" ++ kind ++ "]") ;
  PAnd p q = parenth ("\\x ->" ++ p ++ "x &&" ++ q ++ "x") ;
  POr p q = parenth ("\\x ->" ++ p ++ "x ||" ++ q ++ "x") ;
  PNot p = parenth ("\\x -> not" ++ parenth (p ++ "x")) ;
  KProperty prop kind = "[x | x <-" ++ kind ++ "," ++ prop ++ "x" ++ "]" ;
oper
  parenth : Str -> Str = \s -> "(" ++ s ++ ")" ;
}
```

```
concrete MathQueryHs of MathQuery = QueryHs ** {
lin
  KNumber = "[0 .. 1000]" ;
  TInteger i = parenth ("\\p -> p" ++ i.s) ;
  PEven = "even" ;
  POdd = "odd" ;
  PPrime =
    parenth ("\\x -> x > 1 && all (\\y -> mod x y /=0) [2..div x 2]") ;
  PDivisible e =
    parenth ("\\x ->" ++ e ++ parenth ("\\y -> mod x y == 0")) ;
  PSmaller e = parenth ("\\x ->" ++ e ++ parenth ("x<")) ;
  PGreater e = parenth ("\\x ->" ++ e ++ parenth ("x>")) ;
}
```

Example query and its translation

```
> p -lang=Eng "which even numbers are prime" | l -lang=Hs

[x | x <- [x | x <- [0 .. 1000] , even x ] ,
  ( \x -> x > 1 && all (\y -> mod x y /=0) [2..div x 2] ) x ]
```

A minimalistic user interface

1. Parse English query, `p -lang=Eng`

2. Select just the first tree if ambiguous: `pt -number=1`

3. Linearize to Haskell expression: `l -lang=Hs`

4. Evaluate the Haskell expression: `ghc -e`

A `query` shell script, doing all this:

```
#!/bin/bash
ghc -e "$(echo "p -lang=Eng \"$1\" | pt -number=1 \
  | l -lang=Hs" | gf -run MathQueryEng.gf MathQueryHs.gf)"
```

User interaction at work

```
./query "is any even number prime"
True

./query "which numbers greater than 100 and smaller than 150 are prime"
[101,103,107,109,113,127,131,137,139,149]
```

The limits of grammars*

Many compilation phases have counterparts in machine translation:

- **Lexical analysis**: recognize and classify words.
- **Parsing**: build an abstract syntax tree.
- **Semantic analysis**: disambiguate; add information to tree.
- **Generation**: linearize to target language.

Lexical analysis, parsing, and generation are in both cases derived from grammars,

But what about semantic analysis?

The problem of ambiguity

In compilers: overload resolution, type annotations.

In natural language, ambiguity appears on all levels of lexicon and syntax.

Word sense disambiguation

One word may have several possible translations, corresponding to different senses of the word.

Example: English *drug*, French *médicament* (medical drug) or *drogue* (narcotic drug)

How to translate

the company produces drugs against malaria

is in most cases

la société produit des médicaments contre le paludisme

Why? Because substances used against malaria are medical drugs, not narcotic drugs.

Notice the similarity of this analysis to overload resolution in compilers.

Syntactic ambiguity

I ate a pizza with shrimps

I ate a pizza with friends

Preferred analyses:

I ate a (*pizza with shrimps*)

(*I ate a pizza*) *with friends*

The translation of *with* may depend on the analysis.

How to choose?

The problem accumulates

> *pizza with shrimps from waters without salt from countries in Asia*

has 42 analyses, and their number grows exponentially in the length of the sentence.

Parse error recovery

In compilers: the users are expected to read the language manual. Report a syntax error if the code cannot be parsed!

In natural language: no such manual exists, The system should be **robust**: it should recover from errors by using some back-up method.

(An alternative is **predictive parsing**, which helps the user to stick to the grammar by giving suggestions. But this is only viable in interactive systems, not in batch-oriented text parsers.)

Statistical language models

Derived from the cryptographic methods of the 1940's (Shannon).

Statistical language models: data about co-occurrences of words and sequences of words.

- disambiguation *drogues contre le paludisme* is less common than *médicaments contre le paludisme*
- error recovery by **smoothing**: if a sequence of three words *u v w* is not found, try combine *u v* and *v w*.

Precision/coverage trade-off

Google translate: must deal with any user input and therefore opt for coverage. Typically by statistical models.

Voice commands in a car: deal with limited domains of language, opt for precision. Typically by grammars.

No natural language system of today combines high coverage with high precision.

Hybrid systems

Combine grammar sith statistics.

The current trend in natural language translation.

Interestingly, also increasingly used in compilers

- for unsolvable problems such as optimization
- if rule-based solutions are too complex in practice

Exaample: **profiling** - the analysis on actual runs of programs.

Thus GCC can run a program to collect statistics and use the outcome in later compilations of the same program. For instance, how many times different branches were taken, which may be impossible to know from the program code alone.

A difference

In natural language processing, one may be happy with uncertain outcomes, and therefore rely on methods such as statistical models.

In compilers construction, one wants to be sure. Anything that affects the semantics of programs must be based on firm knowledge; statistics is just used as a possible source of extra gain such as speed improvement.