Programming Language

• an abstract entity with some specification, including:
  • formal specification of tokens, i.e., legal keywords & identifiers
    ⇒ tool: Regular Grammars.
  • formal specification of acceptable sentences, or syntax
    ⇒ tool: Context Free Grammars.
  • (mainly informal) specification of meaning, or semantics.

The Birth of Programming Languages (PLs)

Prior to the 1950s

• programs had to be written in machine code, using absolute addressing.
• programs were hard to read and write, and the job didn’t attract many people.
• software started to cost way more than the hardware.

⇒ birth of assembly languages, with little impact on the design of high-level languages.
The Birth of High-Level PLs

Mid 1950s, 1960s
- announcement of the IBM 704 machine, with indexing and floating-point instructions in hardware
  - birth of the first high-level PL: Fortran. Language was actually implemented (compiler); variables, user-defined subprograms, limited control structures; no recursion, no dynamic data structures.
- growing interest in Artificial Intelligence (AI), from linguistics, psychology, and mathematics

History of Programming Languages

Late 1970s
- shift from procedure-oriented PL designs to data-oriented designs
  - first language with limited support for data abstraction: SIMULA 67, designed by Ole-Johan Dahl and Kristen Nygaard (Norwegian Computing Centre, Oslo).

Other Interesting/Influential PLs

Algol (ALGOrithmic Language):
- designed as a universal language for scientific computing
- hasn’t been used much, but had a great impact on many modern languages

COBOL (COmmon Business Language):
- designed for business applications, with sophisticated data structures
- has been used more than any other PL, but no impact on the design of other PLs

Early 1980s
- giving birth to object-oriented (OO) design
  - first language developing OO concepts: Smalltalk, designed by Adele Goldberg, Alan Kay & colleagues (Learning Research Group, Xerox PARC).
- support for OO programming is now part of most imperative, and even functional/logic PLs:
  - C++/C# (imperative), F# (functional), Prolog++ (logic)

Adele Goldberg
Other Interesting/Influential PLs

SQL (Structured Query Language):
• designed for managing and querying Relational Databases, that store data in the form of tables
• user describes what they want (not how to retrieve them from the database)

Example:

```
SELECT price, duration
FROM flights
WHERE flights.origin = 'toronto'
AND flights.destination = 'paris'
AND price < 1000.00
ORDER BY price;
```
Influences on PL Design

Imperative Paradigm:
- computer architecture (and instruction set).
- hardware and software costs.
- application.
  ⇒ mainly bottom-up,
  ⇒ closer to the needs of the machine/application.

Question: How about functional and logic paradigms?
**Scheme**

Data types: numbers, symbols, pairs, lists, ...

Syntax:

\[
<\text{expr}> ::= <\text{val}>
| (\text{if} <\text{expr}> <\text{expr}> <\text{expr}>)
| (\text{cond} \{\langle<\text{expr}> <\text{expr}>\rangle\} [\langle\text{else} <\text{expr}>\rangle])
| (\text{lambda} \{<\text{id}>\} <\text{expr}>)
| ...
\]

\[
<\text{defn}> ::= (\text{define} <\text{id}> <\text{expr}>)
\]

Example:

\[
> (\text{define} \text{max}
  (\text{lambda} (x y)
    (\text{if} (> x y) x y)))
> (\text{max} 3 4)
\]

**ML**

Datatypes: unit, bool, int, real, string, tuples, lists, ...

Syntax, Example:

\[
\text{fun} \text{mystery} \text{lst} =
\quad \text{let}
\quad \quad \text{fun} \text{mysteryHelper} [] \text{res} = \text{res}
\quad \quad | \text{mysteryHelper} (x::xs) \text{res} =
\quad \quad \quad \text{mysteryHelper} \text{xs} (x::\text{res})
\quad \text{in}
\quad \quad \text{mysteryHelper} \text{lst} []
\quad \text{end};
\]

\[
\text{mystery} : \text{'a list} \rightarrow \text{'a list}
\]

What does \text{mystery} do?

**Scheme**

- clear and simple syntax
- clear and simple semantics (for the core): \(\lambda\)-calculus
- uniform treatment of program and data
- implementations properly tail-recursive
- functions are values: created dynamically, stored, passed as parameters, returned as results, etc.
- static scoping
- dynamic typing
- pass-by-value

Anything in particular you liked/disliked about Scheme?

**ML**

- functional: recursion, higher-order functions
- type safe, static type checking, type inference
- built-in types, type synonyms
- user-defined (abstract) data types
  - enumerated, variant, recursive, mutually recursive types
- exception handling
- static scoping (dynamic scoping for exception handling)
- garbage collection, immutable data types, updatable references

Anything you liked/disliked about ML in particular?
### Prolog

**Syntax:**

- `<clause> ::= <pred> .
  | <pred> :- <pred> { , <pred> } .
- `<pred> ::= <pname>( <term> { , <term> } )
  | <const> | <var>
- `<term> ::= <functor>( <term> { , <term> } )
  | <const> | <var>
- `<const> ::= ...
- `<var> ::= ...

**Example:**

```
mylength([],0).
mylength([_|T],N) :- mylength(T,NT), N is NT+1.
```

### Deficiencies of Prolog as a Pure Logic PL

- Prolog matching is ordered: from top of the database, and from left end of a goal.
- Prolog allows explicit control of backtracking through the use of `Cut !`.
  - Pros and cons of using `Cut`?
- Closed-world assumption, and Negation by failure.

Anything you liked/disliked about Prolog?

### Comparison: Summing a List of Numbers

- **in a C-like language:**
  ```c
  sum = 0;
  for (i=0; i<len(lst); i++) {
    sum += lst[i];
  }
  return sum;
  ```

- **in Scheme:**
  ```scheme
  (define (sumlist lst)
    (+ (car lst) (sumlist (cdr lst))))
  ```

- **in Prolog:**
  ```prolog
  sumlist([H|Rest], Sum) :-
    sumlist(Rest, SumR),
    Sum is H + SumR.
  ```

**Question:** Do we know which one is closer to “how non-programmers reason about programming concepts”?  
[Pane & Myers’2006]

Giving birth to the idea of “More natural programming languages and environments”,
where the goal is:
To understand how people express concepts if not restricted by existing PLs.
There is evidence that:

- there are regularities in how people express step-by-step natural language procedures [Miller’74, 81; Biermann et al.’83]
- people improve the precision of their natural language expressions if they know the recipient has limited intelligence suggesting that such regularities can be incorporated into PL designs.

Despite such findings, these studies have had little impact on the design of new PLs. Why?

References