TDDA69 Data and Program Structure

Concepts and models of programming languages

Cyrille Berger

LINKÖPING UNIVERSITY

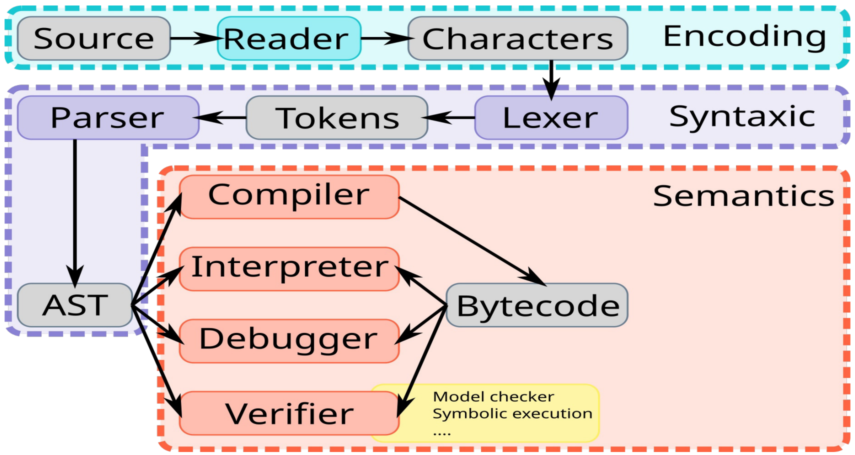
Lectures

Lecture content

- Defining a programming language
- Classification of programming languages
- The different programming paradigms
- Why different paradigms?
- Introduction to KL

Defining a programming language

Defining a programming language (1/2)



Defining a programming language (2/2)

- Syntax

```
SELECT * FROM table WHERE id=1
SELECT * FROM table WHERE
```

- Semantics
  - What operation does a SELECT do?

Formal Grammar

Defined by *G=(N, Σ, P, S)*:

- A finite set *N* of nonterminal symbols
- A finite set *Σ* of terminal symbols, disjoint from N
- A finite set *P* of production rules, each of the form $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
- A start symbol $S \in N$

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<integer> ::= ['-'] <digit> {<digit>}

Context-Free Grammar

Defined by *G=(N, Σ, P, S)*:

- A finite set *N* of nonterminal symbols
- A finite set *Σ* of terminal symbols, disjoint from N
- A finite set *P* of production rules, each of the form $N \rightarrow (\Sigma \cup N)^*$
- A start symbol $S \in N$

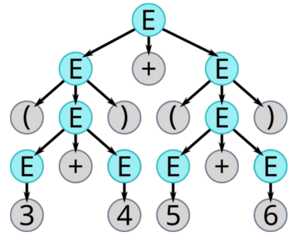
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<integer> ::= ['-'] <digit> {<digit>}

LR(k) Parser

- LR parser has a stack and input
- It uses two operations: *shift* and *reduce*
- It builds a *parse tree*

LR(k) Parser - Example

- Grammar:
  E↦int
  E↦(E)
  E↦E+E
- Stack:
  E
- Input:
- Next action:



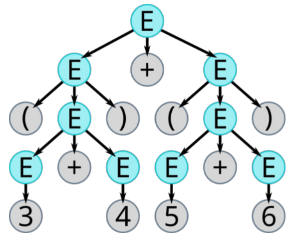
LR(1) and Chomsky Normal Form Grammar

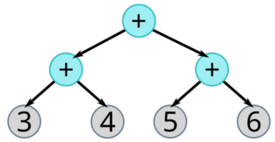
- A Chomsky Normal Form Grammar is a CFG such as, the production rules have the form

$$A \rightarrow BC \qquad A, B, C \in N^3$$
$$A \rightarrow a \qquad A \in N, a \in \Sigma$$

- Any CFG can be transformed into a CFN
- Most generated parser use a variant of LR(1) which is defined from a CNF

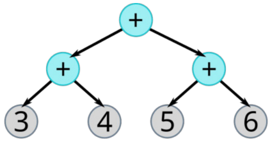
Abstract Syntax Tree





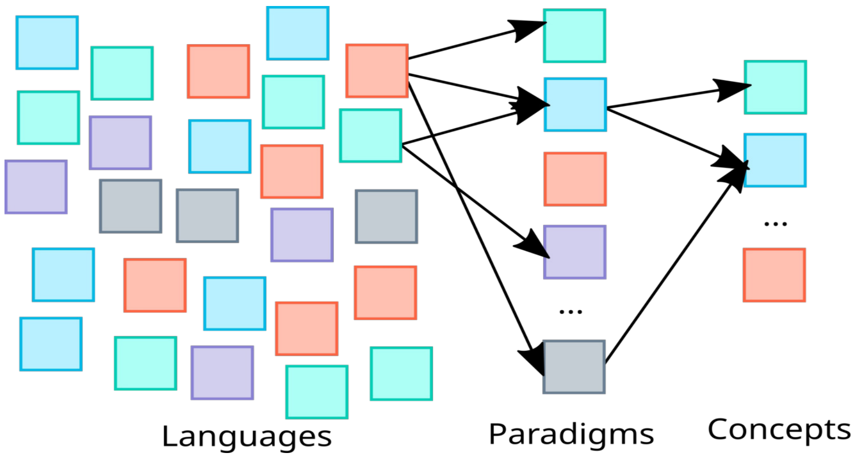
Semantic

- The grammar defines what is valid or not
  - (3+4)+(5+6)
  - 3+)(+6)
- The semantic defines what is the expected result
  - 18

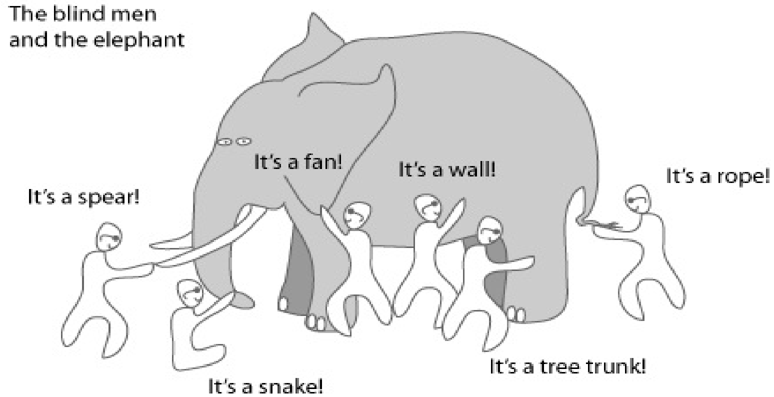


Classification of programming languages

Classification of programming languages



Limitation of a paradigm classification



The different programming paradigms

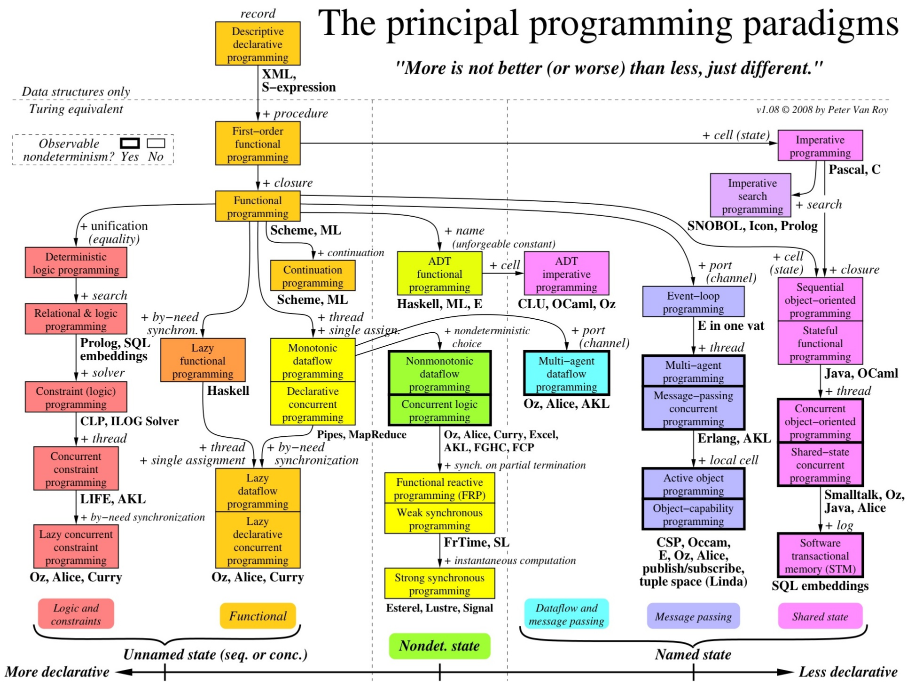
Creative extension principl

- No exceptions

```
function f1 -> x1()
{
  set x2 = f2()
  if(!x2) { ... }
  else x1 = ...
}
function f2 -> x2()
{
  set x3 = f3()
  if(!x3) { x2 = null }
  else x2 = ...
}
function f3 -> x3()
{
  set x4 = f4()
  if(!x4) { x3 = null }
  else x3 = ...
}
function f4 -> x4()
{
  if(valid) { f4 = ... }
  else { f4 = null }
}
```

- With exceptions

```
function f1 -> x1()
{
  set x2 = f2()
  x1 = ...
}
function f2 -> x2()
{
  set x3 = f3()
  x2 = ...
}
function f3 -> x3()
{
  set x4 = f4()
  x3 = ...
}
function f4 -> x4()
{
  if(valid) { f4 = ... }
  else { throw Exception() }
}
```

Declarative

- Expresses logic of computation without control flow:
  - What should be computed and not how it should be computed.
- Examples: XML/HTML, antlr4/yacc/ regular expressions, make/ants, SQL, …

Declarative - Examples

- **Hello world!**
- `SELECT name FROM student WHERE course eq 'TDDA69'`
- grammar Hello;
  r : 'hello' ID;
  ID : [a-z]+ ;
  WS : [' '\t\r\n]+ -> skip ;

Functional

- Computation are treated as mathematical function
  - without changing any internal state
- Examples: Lisp, Scheme, Haskell...

Functional - Examples

- `(print "Hello World")`
- `(take 25 (squares-of (integers)))`
  -> (1 4 9 16 25 36 49 64 ... 576 625)

Imperative

- Express how computation are executed
  - Describes computation in term of statements that change the internal state
- Examples: C/C++, Pascal, Java, Python, JavaScript...

Imperative - Examples

```
for(var i = 1; i < 26; ++i)
{
  var sq = i*i;
  console.log(sq)
}
```

```
#include <stdio.h>
int main()
{
  char ch;
  printf("Enter a character\n");
  scanf("%c", &ch);
  if (ch == 'a' || ch == 'A' || ch == 'e' || ch == 'E' || ch == 'i' || ch == 'I'
|| ch =='o' || ch=='O' || ch == 'u' || ch == 'U')
    printf("%c is a vowel.\n", ch);
  else
    printf("%c is not a vowel.\n", ch);
  return 0;
}
```


Object-Oriented

- Based on the concept of *objects*, which are *data structures* containing *fields* and *methods*
  - Programs are designed by making objects interact with each others
- Examples: C++, Java, C#, Python, Ruby, JavaScript...

Object-Oriented - Programming

```
#include <iostream>
class Character : public Symbol
{
public:
  Character(char _c) : m_c(_c) {}
  bool isVowel() const
  {
    return ch == 'a' || ch == 'A' || ch == 'e' || ch == 'E' || ch == 'i'
           || ch == 'I' || ch =='o' || ch=='O' || ch == 'u' || ch == 'U';
  }
private:
  char m_c;
};
int main()
{
  char c;
  std::cout << "Enter a character:\n";
  std::cin >> c;
  Character ch(c);
  if(ch.isVowel())
  {
    std::cout << c << " is a vowel.\n";
  } else {
    std::cout << c << " is not a vowel.\n";
  }
}
```


Logic and symbolic

- Logic
  - Based on Formal logic: expressing facts and rules
- Symbolic
  - A program can manipulate its own formulas and components as if they are data
- Example: prolog

Logic programming

```
likes(mary,food).
likes(mary,wine).
likes(john,wine).
likes(john,mary).
```

```
| ?- likes(mary,food).
 yes.
| ?- likes(john,wine).
 yes.
| ?- likes(john,food).
 no.
```


Symbolic programming

```
d( X, X, 1 ):- !.                  /* d(X) w.r.t. X is 1      */
d( C, X, 0 ):- atomic(C).     /* If C is a constant */
                                      /* then d(C)/dX is 0 */
d( U+V, X, R ):-                /* d(U+V)/dX = A+B where   */
   d( U, X, A ),                   /* A = d(U)/dX and         */
   d( V, X, B ),
   R = A + B.

...

d( sin(W), X, Z*cos(W) ):-      /* d(sin(W))/dX = Z*cos(W)   */
   d( W, X, Z).                         /* where Z = d(W)/dX          */
d( exp(W), X, Z*exp(W) ):-     /* d(exp(W))/dX = Z*exp(W)   */
   d( W, X, Z).                         /* where Z = d(W)/dX          */

...
```

```
?- d(cos(2*X+1), X, what)
 what = 2*sin(2*X+1)
```


Constraint Programming

- Constraint
  - A relation between two variables are stated in the form of a constraint (can be logic or numerical)
- Example: Oz (functional), Kaleidoscope (imperative), Prolog (logic)

Constraint Programming - Examples

```
local
 proc {MyScript Solution}
   X = {FD.int 1#10}
   Y = {FD.int 1#10}
   Z = {FD.int 1#10}
 in
   Solution = unit(x:X y:Y z:Z)
   X + Y =: Z
   X <: Y
   %% search strategy
   {FD.distribute naive Solution}
 end
in
 {Browse {SearchAll MyScript}}
end
```

Why different paradigms?

Is there a paradigm to rule them all?

- In theory you can program everything in C/C++ and imperative programming, or functional programming...
- But is that convenient?
- And is that safe?

Functional vs Imperative

- Double all the numbers in an array

```
var numbers = [1,2,3,4,5]
```

- Imperative:

```
var doubled = []
for(var i = 0; i < numbers.length; i++) {
  var newNumber = numbers[i] * 2
  doubled.push(newNumber)
}
```

- Functional:

```
var doubled = numbers.map(function(n) {
  return n * 2
})
```


Declarative vs Imperative

- Select all the dogs that belongs to a specific owner
- Declarative:

```
SELECT * from dogs INNER JOIN owners
         WHERE dogs.owner_id = owners.id
```

- Imperative:

```
var dogsWithOwners = []
var dog, owner

for(var dog in dogs) {
  for(var owner in owners) {
    if (owner && dog.owner_id == owner.id) {
      dogsWithOwners.push({ dog: dog, owner: owner })
    }
  }
}
```

Introduction to KL

LINKÖPING UNIVERSITY

Kernel Language

- Kernel language: is the minimal language that you need for a given paradigm
- Define a mapping between a full programming language into the kernel language

Kl

- Practical language

```
function sqrt(x) {
return(x*x); }
define b =
sqrt(sqrt(a));
```

- Kernel language

```
define sqrt = function
(x) -> y { y = mul(x,
x); }
define ___t1___ =
sqrt(a);
define b =
sqrt(___t1___);
```

- Practical language
  - Provides usefull abstractions for the programmer
  - Can be extended with linguistic abstractions
- Kernel language
  - Easy to understand and reason
  - Has a precise (formal) semantic

KL

- KL is a Kernel Language
- You will be developing its interpreter during the labs
- Written in RPython
- It is defined as:
  - syntax: inspired C/JavaScript
  - semantic: +procedure+closure+cell
  - In the future, based on the idea of adding concepts
- Mapping through a macro system (lecture 14)

Syntax

```
define constant_name = 10;
cell variable_name = 10;
variable_name = divide(constant_name, 2);
define f = function y -> (a, b)
{
  define _t1_ = equals(b, 0);
  define _t2_ = function () -> r { r = divide(a, b); };
  define _t3_ = function () { raise("Division by 0"); };
  y = cond(_t1_, _t2_, _t3_);
}
define _t4_ = function() { f(1, 0, 0); }
define _t5_ = function(except) { print(except); };
define _t6_ = function() { print("No problem!"); };
catch(_t4_, _t5_);
catch(_t4_, _t5_, _t6_);
```


Modules

```
import std 1.0;
std.print("Hello World");

import test 1.0;
test.case("Functions Call")
    .check(true);
```


Conclusion

- Definition of a programming language
- Defining the semantinc using programming paradigms classification
- KL