Types, Part I

Abstracting Types from language

Programming Languages

CS 214

©David Meyer All Rights Reserved.

Exam Topics

~60 questions

operator associativity

operator precedence

week 4 set operators: 5 questions

7 questions on history:

- first Language
- first implemented
- Implemented Multitasking

What makes a:

- Functional Language
- Object Oriented Language
- Procedural Language

Classes

Lists and their operators

Block Structure (begin and end)

Multi-Dimensional Arrays

Generic Types

24 questions on identifying the

- formalisms of different
- implementations of branching, selection and looping

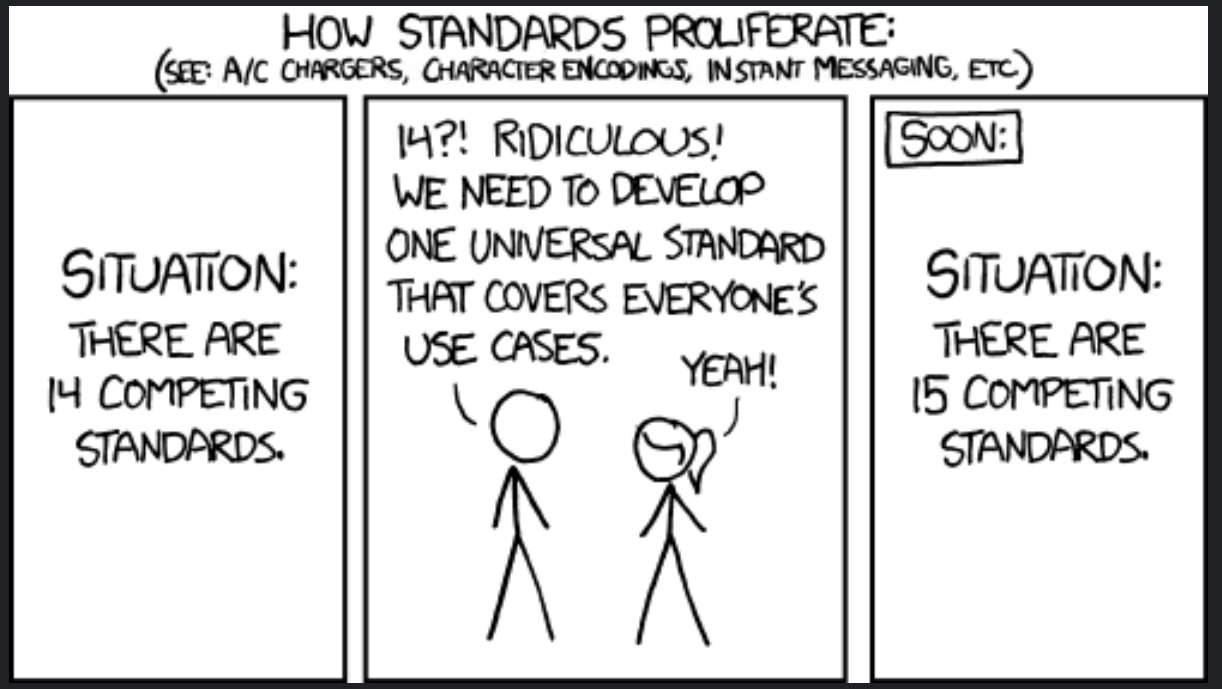
prove syntax ambiguity

create a parse tree

BNF

©David Meyer All Rights Reserved.

“Language Independent Description”



©David Meyer All Rights Reserved.

Types

A *type* is a set *V* of values, and a set *O* of operations onto *V*.

Examples from C++:

- The *int* type:

  V = {INT_MIN, ... -1, 0, 1, ..., INT_MAX-1, INT_MAX}

  O = {<<, >>, +, -, *, /, %, =, ++, –, ...}

- The *char* type:

  V = {NUL, ..., '0', ..., '9', ..., 'A', ..., 'Z', ..., 'a', ..., 'z', DEL}

  O = {<<, >>, =, ++, –, isupper(), islower(), toupper(), ... }

- The *string* type:

  V = {"", "A", "B", "C", ..., "AA", "AB", "AC", ..., "AAA", ...}

  O = {<<, >>, +, +=, [], find(), substr(), ... }

©David Meyer All Rights Reserved.

Fundamental Types

Let's assume the existence of some basic (primitive) types:

Name	V	C++	Ada	Smalltalk	Lisp
bool	false, true	*bool*	*boolean*	*Boolean*	*boole*
char	the set of chars	*char*	*character*	*Character*	*character*
int	the integers	*int*	*integer*	*Integer*	*integer*
real	the reals	*double*	*float*	*Float*	*real*

Such "primitive" types require little memory:

- *bool*—a single bit
- *char*—1-2 bytes
- *int*—a single word
- *real*—1-2 words

©David Meyer All Rights Reserved.

Why did the programmer quit his job?

Because he didn’t get arrays!

Non-Fundamental Types

New, non-fundamental types can be built from existing types:

- *Sequence* types (arrays, vectors, lists, etc.) store multiple values of the same type.
- Aggregate types (records, structs, classes, etc.) store multiple values of arbitrary types.

A *type constructor* is a formal mechanism for modeling a new type that is independent of any particular language, using the mathematics of *sets*.

A type constructor has 3 components:

- The syntax used to denote that constructor;
- The set of elements produced by that constructor; and
- The set of operations associated with that constructor.

©David Meyer All Rights Reserved.

Set Constructor I: *Kleene Closure*

Kleene Closure is a formalism for modeling *sequence types*.

– The Kleene Closure of a set A is denoted A*.

– The Kleene Closure of a set is the set of all tuples that can be formed using elements of that set.

Example: The Kleene Closure of bool – bool* – is the infinite set:

```
{ (), (false), (true), (false, false), (false, true),
 (true, false), (true, true), (false, false, false), … }
```

– For a tuple t ∈ A*, the operations include:

null(A*) → bool

```
null(())          → true
null((false))     → false
null((true))      → false
```

first(A*) → A

```
first((true, false))   → true
first((false, true))   → false
```

rest(A*) → A*

```
rest((true, true, false)) → (true,false)
rest((false, true, true)) → (true, true)
```


©David Meyer All Rights Reserved.

Kleene Closure Examples

If *char* is the set of ASCII characters, what is *char** ?

– The infinite set of all tuples formed from ASCII characters.
(in C, the set of all character strings).

The C/C++ notation: "Hello"
is just a different syntax for: ('H', 'e', 'l', 'l', 'o')

Thus, *int** denotes a sequence (array, list, ...) of integers;

```
int intStaticArray[32];
int * intDynamicArray = new int[n],
vector<int> intVec;
list<int> intList;
```

*real** denotes a sequence (array, list, ...) of reals;
and so on.

©David Meyer All Rights Reserved.

Sequence Operations

Sequence operations can be built via *null()*, *first()*, and *rest()*

- An output operation can be defined like this

(p

```
void print(ostream& out, int * a) {
  if ( !null(a) ) {
    out << first(a) << ' ';
    print(out, rest(a));
  }
};
```

- A subscript operation can be defined like this

(p

```
char& operator[](char * a, int i){
  if (i <= 0)
    return first(a);
  else
    return operator[](rest(a), i-1);
};
```

In Lisp:

first is called *car*

rest is called *cdr*.

©David Meyer All Rights Reserved.

Speaking of bears, what do you call an earless bear?

A b.

Set Constructor II: *Product*

Product is a formalism for modeling *aggregate types*.

- The product of two sets A and B is denoted $A \times B$.
- $A \times B$ consists of all ordered pairs (a, b): $a \in A, b \in B$.

  $A \times B \times C$ consists of all ordered triples (a, b, c): $a \in A, b \in B, c \in C$.

  $A \times B \times \ldots \times N$ consists of all ordered n-tuples $(a, b, \ldots, n)$:
  $a \in A, b \in B, \ldots, n \in N$.

  Example: the set *bool* $\times$ *char* has 256 elements:
  { ..., (true, 'A'), (false, 'A'), (true, 'B'), (false, 'B'), ..., }.
- Operations associated with product are the *projection* operations:
  - *first*, applied to an n-tuple $(s_1, s_2, \ldots, s_n)$ returns s_1.
  - *second*, applied to an n-tuple $(s_1, s_2, \ldots, s_n)$ returns s_2.
  - *nth*, applied to an n-tuple $(s_1, s_2, \ldots, s_n)$ returns s_n.

©David Mertz, All Rights Reserved

Product Example: C++ structs

```
struct Student
{
    int id;
    double gpa;
    char gender;
};
Student aStudent;
```

Formally, a Student consists of:

$$\text{int} \times \text{real} \times \text{char}$$

Formally, a particular Student:

```
aStudent.id = 12345;
aStudent.gpa = 3.75;
aStudent.gender = 'F';
```

is the 3-tuple: (12345, 3.75, 'F').

The C++ “dot-operator” is a projection operation:

```
cout << aStudent.id          // extract id
     << aStudent.gpa         // extract gpa
     << aStudent.gender      // extract gender
     << endl;
```


©David Meyer All Rights Reserved.

Set Constructor III: *Function*

Function is a formalism for *subprograms* (type operations).

– The set of all functions from a set A to a set B is denoted (A) → B.

– A particular function *f* mapping A to B is denoted *f* (A) → B.

Examples:

– The set (char) → bool contains all functions that map char values into bool values, some C examples of which include:

```
isupper('A') → true        islower('A') → false
isalpha('A') → true        isdigit('A') → false
isalnum('A') → true        isspace('A') → false
```

– The set (char) → char contains all functions that map char values into char values, some C examples of which include:

```
tolower('A') → 'a'         toupper('a') → 'A'
```


©David Meyer All Rights Reserved.

Function and Product

What does this set contain? $(\text{int} \times \text{int}) \rightarrow \text{int}$

– All functions that map pairs of integers into an integer.

Examples?

```
+((2, 3)) → 5
*((2, 3)) → 6
```

```
-((2,3)) → -1
/((2,3)) → 0
```

Suppose we define an aggregate named *IntPair*:

```
struct IntPair {
   int a,
       b;
};
```

and then define a function named add():

```
int add(IntPair ip) {
  return ip.a + ip.b;
};
```

add() is a member of the set: $(\text{int} \times \text{int}) \rightarrow \text{int}$

The function constructor let us create new operations for a type.

©David Meyer All Rights Reserved.

Function Arity

Product serves to denote an aggregate or an argument-list.

What does this set contain? $(\text{int} \times \text{int}) \rightarrow \text{bool}$

- All functions that map pairs of integers to a boolean.

Examples? `==((2, 3))` → `false` `!=((2,3))` → `true`
`<((2, 3))` → `true` `>((2,3))` → `false`

Definition:

The number of operands an operation requires is its *arity*.

- Operations with 1 operand are *unary* operations, with *arity-1*.
- Operations with 2 operands are *binary* operations, with *arity-2*.
- Operations with 3 operand are *ternary* operations, with *arity-3*.
- ...

©David Meyer All Rights Reserved.

Example Ternary Operation

The C/C++ conditional expression has the form:

$$\texttt{<expr>}_0 \texttt{ ? } \texttt{<expr>}_1 \texttt{ : } \texttt{<expr>}_2$$

producing $\texttt{<expr>}_1$ if $\texttt{<expr>}_0$ is true, and
producing $\texttt{<expr>}_2$ if $\texttt{<expr>}_0$ is false.

Here is a simple *minimum()* function using it:

```
int minimum(int first, int second) {
  return (first < second) ? first : second;
};
```

The C/C++ conditional expression is a *ternary operation*;
the one above can be formally described by:

$$?:(\text{bool} \times \text{int} \times \text{int}) \rightarrow \text{int}$$

©David Meyer All Rights Reserved.

Example Ternary Operation: Pluralization

```
cout << "Enter the number of values you have to process: ”;
int n;
cin >> n;
      … read and process the n values
cout << "You entered " << n << "value"
      << (n == 1) ? "." : "s." // based on n, pluralize ‘value’
      << endl;
```

This ternary operation is from $(\text{bool} \times \text{char*} \times \text{char*}) \rightarrow \text{char*}$

Operator Positioning

Operators are also categorized by their position relative to their operands:

- *Infix* operators appear *between* their operands: `1 + 2`
- *Prefix* operators appear *before* their operands: `+ 1 2`
- *Postfix* operators appear *after* their operands: `1 2 +`

`* + 2 3 - 4 2` ≡ `(2 + 3) * (4 - 2)` ≡ `2 3 + 4 2 - *`

Prefix, infix, and postfix notation are different conventions for the same thing; a language may choose any of them:

C++ Expr	Category	Value	Lisp Expr	Category	Value
x < y	binary, infix	true, false	(< x y)	binary, prefix	true, false
++x	unary, prefix	x+1	(incf x)	unary, prefix	x+1
11 + 12	binary, infix	23	(+ 11 12)	binary, prefix	23
!*flag*	unary, prefix	neg. of *flag*	(not *flag*)	unary, prefix	neg. of *flag*
cout << x	binary, infix	cout	(princ x *str*)	binary, prefix	x
x++	unary, postfix	x	None		

©David Meyer All Rights Reserved.

Where did Noah keep the bees for 40 days and 40 nights?

In the ark hives.

Review

There are four fundamental types:

– *bool, char, int, real*

New types can be modeled using three set formalisms:

– *Kleene Closure* (sequence types): A*

Operations: null(), first(), rest()

– *Product* (aggregate types): $A \times B$

Operations: first(), second(), ...

– *Function* (type operations): $(A) \rightarrow B$

These *set constructors* provide a mechanism for modeling new types, independent of specific programming languages.

©David Meyer All Rights Reserved.

Practice Using Constructors

Give formal descriptions for:

- The *logical and* operation (&&):
  - –How many operands does it take? 2
  - –What types are its operands? bool, bool
  - –What type of value does it produce? bool
  - So && is a member of $(\text{bool} \times \text{bool}) \rightarrow \text{bool}$
- The C++/*STL substring* operation (*str*.substr(*i*,*n*)):
  - –How many operands does it take? 3
  - –What types are its operands? string, int, int
  - –What type of value does it produce? string
  - So substr() is a member of: $(\text{string} \times \text{int} \times \text{int}) \rightarrow \text{string}$
- For you: The *logical negation* operation (!):

©David Meyer All Rights Reserved.

Logical Negation (!)

How many operands does it have?	1
What types are they?	bool
What type of value does it produce?	bool
Then ! is a member of?	(bool) → bool

More Practice

- For you: this *C++ record*:

```
struct Student {
  int myID;
  string myName;
  bool iAmFullTime;
  double myGPA;
};
```


©David Meyer All Rights Reserved.

C++ Student Structure

As an aggregate type,
Student is a member of:

$$\text{int} \times \text{string} \times \text{bool} \times \text{real}$$

More Practice

- For you: this *C++ record:*

```
struct Student {
  int myID;
  string myName;
  bool iAmFullTime;
  double myGPA;
};
```

- For you: an *accessor* method:

```
struct Student {
  int myID;
  int getId() const;
  string myName;
  bool iAmFullTime;
  double myGPA;
};
```


©David Meyer All Rights Reserved.

C++ Accessor Method

As a method/function:

How many operands?	1
What are their types?	Student
What type of value does it return?	Int
Then getID() is a member of:	(Student) → int

More Practice

- For you: this *C++ record:*

```
struct Student {
   int myID;
   string myName;
   bool iAmFullTime;
   double myGPA;
};
```

- For you: an *accessor* method:

```
struct Student {
   int myID;
   int getId() const;
   string myName;
   bool iAmFullTime;
   double myGPA;
};
```

- How does this affect our *Student* description?

©David Meyer All Rights Reserved.

Adding an Accessor to Student

```
struct Student {
  int myID;
  int getId() const;
  string myName;
  bool iAmFullTime;
  double myGPA;
};
```

$$\text{int} \times (\text{Student}) \rightarrow \text{int} \times \text{string} \times \text{bool} \times \text{real}$$

Or perhaps more readably:

$$\begin{aligned} &\text{int} \times \\ &(\text{Student}) \rightarrow \text{int} \times \\ &\text{string} \times \\ &\text{bool} \times \\ &\text{real} \end{aligned}$$

Note: Since *string* is a sequence type, *char** could be used in its place.

More Practice (ii)

- For you: A “complete” class:

```
class Student {
 public:
  Student();
  Student(int, string, bool, double);
  int getID() const;
  string getName() const;
  bool getFullTime() const;
  double getGPA() const;
  void read(istream &);
  void print(ostream &) const;
 private:
  int    myID;
  string myName;
  bool   iAmFullTime;
  double myGPA;
};
```


©David Meyer All Rights Reserved.

Class Student

```
class Student {
 public:
  Student();
  Student(int, string, bool, double);
  int getID() const;
  string getName() const;
  bool getFullTime() const;
  double getGPA() const;
  void read(istream &);
  void print(ostream &) const;
 private:
  int    myID;
  string myName;
  bool   iAmFullTime;
  double myGPA;
};
```

$(\text{Student\&}) \to \text{Student} \times$

$(\text{Student\&} \times \text{int} \times \text{string} \times \text{bool} \times \text{real}) \to \text{Student} \times$

$(\text{Student}) \to \text{int} \times$

$(\text{Student}) \to \text{string} \times$

$(\text{Student}) \to \text{bool} \times$

$(\text{Student}) \to \text{real} \times$

$(\text{Student\&} \times \text{istream\&}) \to \oslash \times$

$(\text{Student} \times \text{otream\&}) \to \oslash \times$

$\text{int} \times$

$\text{string} \times$

$\text{bool} \times$

real

Summary

A type consists of *data* and *operations*.

There are four *primitive types*: *bool, char, int, real*

The set constructors:

- *Kleene closure* (to represent sequences)
- *Product* (to represent aggregates and function parameters)
- *Function* (to represent operations)

provide a formal way to model new (non-primitive) types, independent of any particular programming language:

→ Use the *product* and *Kleene closure* to represent the type's *data*

©David Meyer All Rights Reserved.