Lecture 2

Types, Functions, Quantifiers and Recursion

Perfect Symbols

- Identifiers
  - Case sensitive
  - must begin with a letter
  - letters,digits, underscore
- Symbols

  // comment

  ; separator between declarations and/or statements

Perfect Symbols

,	separator between expressions, between postconditions and between statements
^=	is defined as
:	of type
:-	such that
=	means equality, not assignment

Brackets

- Round
  - parenthesise expressions
    - 2*(4+7)
- Square
  - invoke indexing operator
  - enclose conditions in conditional constructs
- Curly
  - enclose parameter list in class constructors

Perfect symbols

- Reserved words
  - There are 208 reserved words
    - See Language reference manual 3.6 for a list
- Character literals
  - use single quote symbols

    `a`

    `\n` (new line)

Classes and types

- class
  - a set of allowed values which is not a proper subset of any other *Perfect* class nor a union of types
- type
  - a set of allowed values and may be:
    - a class
    - a class associated with a constraint
    - a union of types

Predefined classes

- anything
  - base class for all other classes
  - contains member function `toString`
- bool
  - class comprising the values `true`, `false`
- `byte`
  - class of 8 bit values

Predefined classes

- char
  - character set supported by the environment
- int
  - unbounded non-negative and negative whole numbers
- void
  - class comprising the single value null

Predefined classes

- rank
  - enumeration class comprising the values below, same and above in that order
- **Note**
- class anything is deferred
- all other classes are final

Predefined types

- `nat`
  - subset of int consisting of unbounded non-negative integers
- `string`
  - sequence of characters

Class bool

- Values
  - true, false
- Operators
  - &, |, ==>, <==, <==>, ~
- Functions
  - toString : string

Properties & Assertions

- `assert`
  - states a condition that must hold
  - generates a proof obligation
- `property`
  - an assertion
  - serves to:
    - verify that system will satisfy requirements
    - give hints to verifier for proving verification conditions
    - give hints to code generator

Example

```
property assert true & true = true;
property assert true & false = false;
property assert true | false = true;
property assert true ==> false = false;
property assert false ==> true = true;
property assert true <==> false = false;
property assert true <== false = true;
```

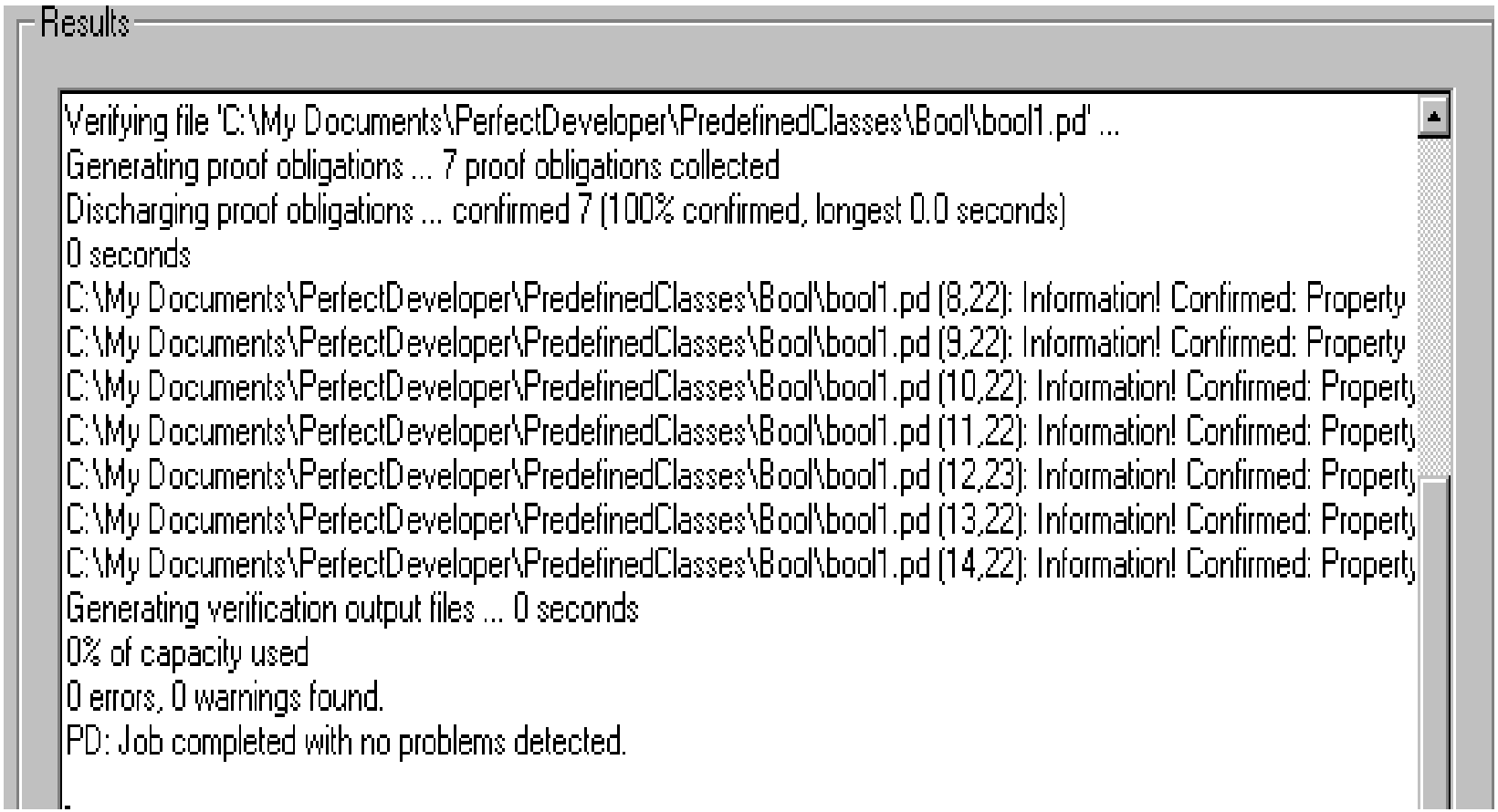
Using Verifier

- Click build -> verify

or

- Right click on the file name in PD and select verify

Using Verifier



Constants

To introduce a named constant use

```
const name : type
    ^= ?
```

Example

```
const b1 : bool
    ^= true;
```

Example

```
const b1 : bool
  ^= true;
const b2 : bool
  ^= false;
property assert b1 & b2 = false;
```

Class char

- char represents characters in some character set
- literals enclosed in back-quotes - `a`
- Methods
  - > successor
  - < predecessor
  - e.g. (>`A`) =`B`

Class char

+c = ordinal value of character c

e.g. +`A` = 65 (ASCII)

- Constructor
  - char{x} = character whose ordinal value is x
  - char{65} = `A` (ASCII)

Class char

- Functions
  - isLetter : bool
  - isDigit : bool
  - isPrintable : bool
  - digit : nat
    - returns numerical equivalent of digit
    - e.g. `1`.digit = 1
  - toString

Example

```
const c1 : char
 ^= `a`;
const c2:char
  ^= >c1;
property assert ~c1.isDigit;
property assert c2 > c1;
property assert c1.isLetter;
```

Class int

- Set of non-negative and negative whole numbers. No upper or lower bounds.
- Literals
  - decimal, binary or hexadecimal
- Binary Operators

  +, -, *, /, %, ^

  - Note: for both / and % the divisor must be > 0

Class int

- Unary operators
  - -, <, >
- Constructor
  - int{s}, where s is a string of digits only
- Functions
  - toString

Non-Member Function

```
function name(arg_list) : type
  ^= ?;
```

Example

```
function inc(a:int):int
    ^= a + 1;
```

Example

```
const x : int
 ^= 10;
function inc(a:int) : int
 ^= a+1;
function add(a:int, b:int):int
 ^= a+b;
property assert add(6,5) = 11;
property assert inc(x) = x + 1;
```

Enumeration class

A class that has a finite number of named, constant values

Declaration

class name

^= enum comma separated namedlist end

```
class TrafficLight
  ^= enum red, green, amber end
```

Enumeration class

To reference elements of the class use:

name@className

e.g. `red@TrafficLight`

Operators: < (predecessor) , >(successor)

```
< amber@TrafficLight = green@TrafficLight
```

Comparison also defined: <, =, >, ~~

Quantifiers over Types

Given a type T and predicate P quantifiers can be used to describe properties of the type that satisfy the given predicate P.

```
forall x : T  :- P(x)
exists x : T :- P(x)
```

Examples

- Every integer is even or odd

```
property assert forall x : int :- x%2 = 0 | x%2 = 1;
```

- The natural numbers include the number 10

```
property assert exists x : nat :- x = 10;
```

Subtypes

- An existing class offers the required functionality

  class *name* ^= *existing class name*

  e.g. class Length ^= int;

Subtypes

Constrained type

The quantifier `those` can be used to define a subtype of an existing type

```
class Mark ^= those x:int :- 0<=x<=100;

class Even  ^= those x:int :- x%2=0;
```

Declaring subtype constants

```
const a : Even
 ^= 4;
const m1 : Mark
 ^= 56;
```

Conditionals

- Guarded expression

  [booleanExp]: expression

  e.g. [a %2 = 0] : true

- Conditional expression

  ([b1] : exp_1,

  [b2]: exp_2,

  ..

  []: exp_n

)

Conditionals

```
function even(a:int):bool
    ^=(
        [a % 2 = 0]:
            true,
        []:
            false
       );
```

- Verify function

```
property assert even(84);
property assert ~even(21);
```

let

Possible to introduce temporary values using the keyword `let`. For example,

```
function fourth(a:int):int
 ^=(
     let square ^= a*a;
     square*square
   );
```

- `Verify function`

```
property assert fourth(3) = 81;
```

Example

```
function div(a:int,b:int):int
 ^=(let t ^=
      ([b < 0]: -1*b,
       [b = 0]:  1,
       []:       b
      );
      assert t > 0;
      a/t
   );
```

Example

- Verify function div

```
property assert div(7,2) = 3;
property assert div(7,0) = 7;
property assert div(7,-3) = 2;
```

Comparison operators

- Equality (=)
  - Defined for all classes
  - Expressions of the same type
- Compare (~~)
  - Compare returns a value of the enumeration class rank whose members are below, same, and above.

Examples of ~~

```
const a : int
    ^= 10;
const b : int
    ^= 7;

 property assert a~~b = above@rank;
```

Definitions of

a > b <==> a~~b = above@rank

a = b <==> a~~b = same@rank

a < b <==> a~~b = below@rank

a >= b <==>

    a~~b = above@rank | same@rank

a <= b <==>

    a~~b = below@rank | same@rank

string

- Sequence of characters
- String literal denoted by double quote marks, e.g. “It is a string”
- Length of string

  #s
- Concatenation

  ++

string

- Comparison Operators

  <, >, <=, >=, =, ~=, ~~

Boolean Functions

s.begins(s1) returns true if s1 is a prefix of s; otherwise false

s.ends(s1) returns true if s1 is a suffix of s; otherwise false

string

`s.empty` returns `true` if `s1` is a suffix of `s`; otherwise `false`

`c in s` returns `true` if `s` contains character `c`; otherwise `false`

- Selector

`s[j]` equals the character at position `j`

where `0 <= j < #s`

string examples

```
const s1 : string
 ^= "Perfect Developer";

function len(s:string):nat
 ^= #s;

function concat(a:string,b:string):string
 ^= a++b;
```

string examples

```
property assert len(s1) = #s1;

property assert concat("Perfect
  ","Developer") = s1;

property assert "Hello" ~~ "hello" =
  below@rank;

property assert "Hello" < "hello";
```

string examples

```
property assert `P` in s1;

property assert s1.begins("Perfect");

property assert s1.ends("Developer");

property assert s1[0] = `P`;
```

Quantifiers over Ranges

- Format

  quantifier :: range :- predicate

```
forall j  :: a .. b :- P(j)
exists j  :: a .. b :- P(j)
```

Note: j is bound variable

Quantifier examples

A function that takes a string as argument and determines if all its characters are digits

```
function allDigits(a:string):bool
    ^= forall j :: 0..<#a :- `0`<= a[j] <= `9`;
```

A function that takes an integer as argument and determines if it is a prime number

```
function isPrime(a:nat):bool
^= (a >= 2) & (forall j :: 2..<a :- a % j ~= 0);
```

Testing

```
property assert isPrime(11);
property assert ~isPrime(12);

const s1 : string
 ^= "4536781";
property assert allDigits(s1);
```

Quantifying over strings

- Format

  quantifier :: string :- predicate

  `forall x :: s :- P(x)`

  forall x an element of s such that P(x)

  `exists x :: s :- P(x)`

  there exists x an element of s such that P(x)

  `those x :: s :- P(x)`

  those x an element of s such that P(x)

Examples

```
const s1 : string  ^=  "4536781";
const s2 : string ^= "ab2c5b";

function allDigits(a:string):bool
 ^= forall c :: a :- `0`<= c <= `9`;

function getDigits(a : string):string
 ^= those x :: a :- `0`<= x <= `9`;
```

Testing

```
property assert allDigits(s1);

property assert getDigits(s2) = "25";
```

Recursion

- A recursive definition is one defined in terms of itself
- Factorial n

  $0! = 1$

  $n! = n * (n-1)!$

- Fibonacci Numbers

  $f_0 = 1,\ f_1 = 1$

  $f_n = f_{n-1} + f_{n-2}$

fac(n)

```
function fac(n : nat): nat
    decrease n
    ^=
     ( [n = 0]:
          1,
        []:
         n * fac(n-1)
     );
```

Testing

```
property assert fac(0) = 1;

property assert fac(5) = 120;
```

sum(n)

```
function sum(n:nat):nat
  decrease n
  ^=
    ( [n = 0]:
        0,
      []:
        n + sum(n-1)
    );
```

Testing

```
function sum1(n:nat):nat
 ^= n*(n+1)/2;

property assert sum(6) = 21;
property assert sum(6) = sum1(6);
```