Resolve Quick Reference

Data Types

Character: `'C'`
Integer: `-50, 25`
Boolean: `True(), False()`
Char_Str: `"Hello"`

Operators

Arithmetic

```
x+y  x-y  x*y  x/y  x mod y
```

Relational

```
x=y  x/=y  x<y  x<=y  x>y  x>=y
```

Boolean

```
x or y      x and y      not x
```

Statements

Modification

Assignment: `x := y;`
Swap: `x :=: y;`

Variable Definition

```
Var i: Integer;
```

Branching / Looping

```
If condition then
  statements
else
  statements
end;

While condition
  maintaining condition;
  decreasing expression;
do
  statements
end;
```

Operation Definition

Return value

```
Operation Add(evaluates x: Integer;
    evaluates y: Integer): Integer;
Procedure
  Add := x + y;   -- assign return value
end Add;
```

No return value

```
Operation Increment(updates x: Integer);
Procedure
  x := x + 1;
end Increment;
```

Recursive Procedure

```
Operation Flip(updates Q: Queue);
Recursive Procedure
    decreasing |Q|
    ...
end Flip;
```

Parameter Passing Modes

Caller's parameter may be changed:

- **updates** - parameter is changed to a value specified by the ensures clause
- **alters** - parameter is changed to an unspecified value
- **clears** - parameter reset to default type value on completion
- **replaces** - initial value of parameter not used; replaced with a value specified by ensures clause

Caller's parameter is unchanged:

- **preserves** - parameter is not changed at any time during call
- **restores** - if procedure alters parameter during execution, procedure will set parameter to its incoming value upon completion
- **evaluates** - parameter may be an expression

RESOLVE Math Theory Quick Reference

Math Types

Z:	-5, 0, 15
B:	`true, false`
str(Z):	< *-5, 10, -15*, … >
powerset(Z):	{ 10, 15, -5, … }
(others…)	

Operators

Arithmetic

`x+y  x-y  x*y  x/y  x mod y`

Relational

`x=y  x/=y  x<y  x<=y  x>y  x>=y`

Boolean

`x` **`or`** `y` `x` **`and`** `y` **`not`** `x`

String

`|S|`	length of S
`S1 o S2`	concatenation of S1 and S2
`<e>`	string containing item *e*
`Reverse(S)`	reverse of S
`Prt_Btwn(start, end, S)`	substring of S from start .. end (start and end are 1-based)
`DeString(S)`	if |S|=1, the first element *e* in S

Examples

Concept

```
Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
        uses String_Theory, Integer_Theory;
        requires Max_Depth > 0;

    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        constraint |S| <= Max_Depth;
        initialization ensures S = Empty_String;
    end;

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| < Max_Depth;
        ensures  S = <#E> o #S;

end;
```

Realization

```
Realization Array_Realiz for Stack_Template;
    uses Binary_Iterator_Theory;
    Type Stack is represented by Record
            Contents: Array 1..Max_Depth of Entry;
            Top: Integer;
        end;
        convention
            0 <= S.Top <= Max_Depth;
        correspondence
            Conc.S = Reverse(Iterated_Concatenation(1, S.Top,
                        lambda(i : Z).(<S.Contents(i)>)));
        initialization
            S.Top := 0;
        end;
    end;

    Procedure Push(alters E: Entry; updates S: Stack);
        S.Top := S.Top + 1;
        E :=: S.Contents[S.Top];
    end;
end;
```

Facility

```
Facility Stack_Demo;
  uses Stack_Template;

  Facility Integer_Stack_Fac is Stack_Template (Integer, 10)
      realized by Array_Realiz;

  Operation Main();
  Procedure
    Var S: Stack;
    Var I: Integer;

    I := 5;
    Push(I, S);

  end Main;

end Stack_Demo;
```

Updated 3/26/2015 by Stephen Schaub