An Implementation of ACT ONE in MPS

Trong Duc Truong

Supervisor: Andreas Prinz

Agenda

- Background
- Objective
- Functional Requirements
- ACT ONE
- Term Rewriting Logic
- IDE
- Discussion
- Conclusion

Background

- DAT233-G - Concepts of Programming Languages
- Specification Language ACT ONE
- No dedicated tools
- We need a tool

Objective:

*Implement the **specification language ACT ONE** in **MPS**, fitted with features inline with the course **DAT233**.*

1. Our own implementation of the language **ACT ONE**.
2. A **term rewriting system** for it.
3. A corresponding **Integrated Development Environment (IDE)**.

Functional Requirements

Title	Must	Description
Implement Prefix Operations	YES	Allow ACT ONE to work with Prefix Operations.
Implement Infix Operations	NO	Allow ACT ONE to work with Infix Operations.
Let-notation	NO	Allow users to declare let-declarations.
If-Then-Else	NO	Allow users to declare if-then-else statements.
Allow specification extension	NO	Allow specification to extend another specification.
Implement Operator-Precedence	NO	Allow ACT ONE to work with Operations while handling Operator-Precedence logic.
Implement Associativeness	NO	Allow ACT ONE to work with Infix Operations with associativeness.
Implement Communativeness	NO	Allow ACT ONE to work with binary Operations while handling Communativeness.
UTF-8-compatible Syntax	YES	Allow the syntax of ACT ONE to be written in plain-text editors (UTF-8).
Implement Subsort	NO	Allow ACT ONE to work with subsorts.
Universal Sort	NO	Allow ACT ONE to work with universal sort, a sort that conforms with all sorts.
Parameterised Specification	NO	Make it possbile for ACT ONE to work with parameterised specifications.
Overloaded Operations Check	NO	Allow the IDE to see overloaded operations.
Duplicate Operations Check	NO	Allow the IDE to detect duplicate Operations.
Parameter Type Check	NO	Allow the IDE to detect wrongly typed parameters.
Parameter Number Check	NO	Allow the IDE to detect Operation-calls with wrong number of parameters.
Predefined Boolean	NO	Allow ACT ONE to use Boolean Specification by default.
Predefiend Integer	NO	Allow ACT ONE to use Integer Specification by default.
Predefined List	NO	Allow ACT ONE to use List Specification by default.
Step-Reduction	YES	Allow users to initiate n-Step-Reduction, where n is an abritrary, positive number.
Exhaustive Reduction	YES	Allow users to initiate Exhaustive-Reduction.
BFS Reduction	YES	Allow users to initiate Breadth-First Reduction.
DFS Reduction	YES	Allow users to initiate Depth-First Reduction.
Tuple Reduction	NO	Allow users to initiate double parameterised Reductions, e.g. Breadth-First 6-Step-Reduction.
Infinite-Reduction-Handler	NO	Allow the rewrite system to prematurely end a reduction that diverges towards infiity.

ACT ONE

Specification

*"An act of **identifying something precisely** or of stating a **precise requirement**."*

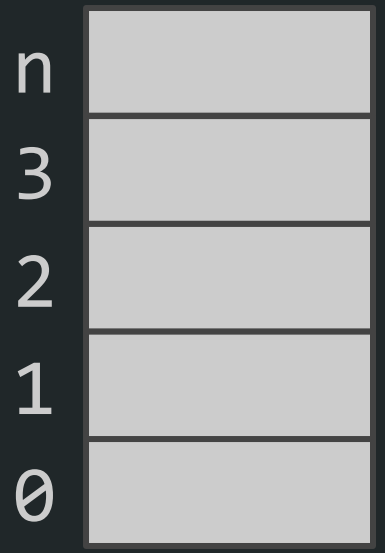
-Oxford Dictionary

Example: Stack

- Abstract Data Type
- LIFO

Example: Stack

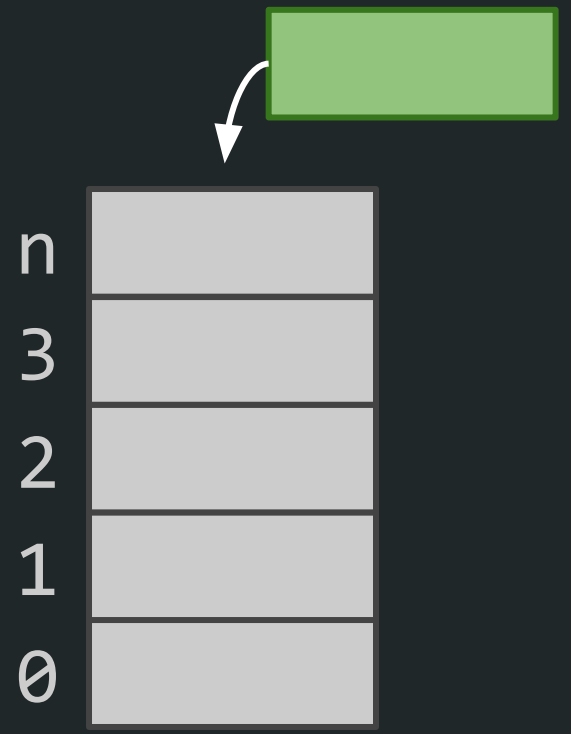
- Abstract Data Type
- LIFO



Init

Example: Stack

- Abstract Data Type
- LIFO



Push

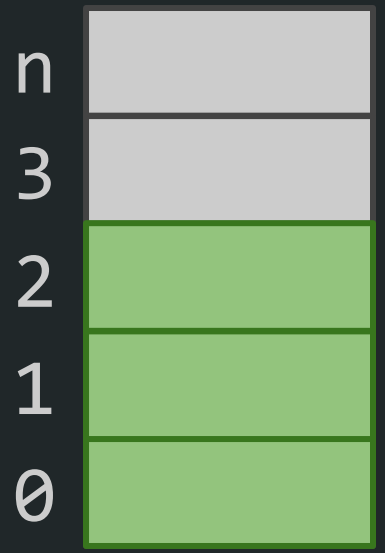
Example: Stack

- Abstract Data Type
- LIFO

Push

Example: Stack

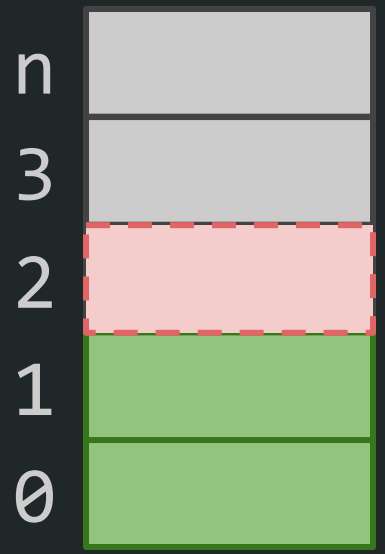
- Abstract Data Type
- LIFO



Push

Example: Stack

- Abstract Data Type
- LIFO



Pop

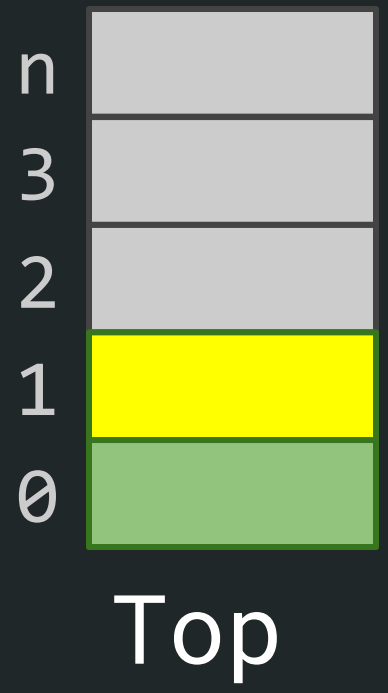
Example: Stack

- Abstract Data Type
- LIFO

Pop

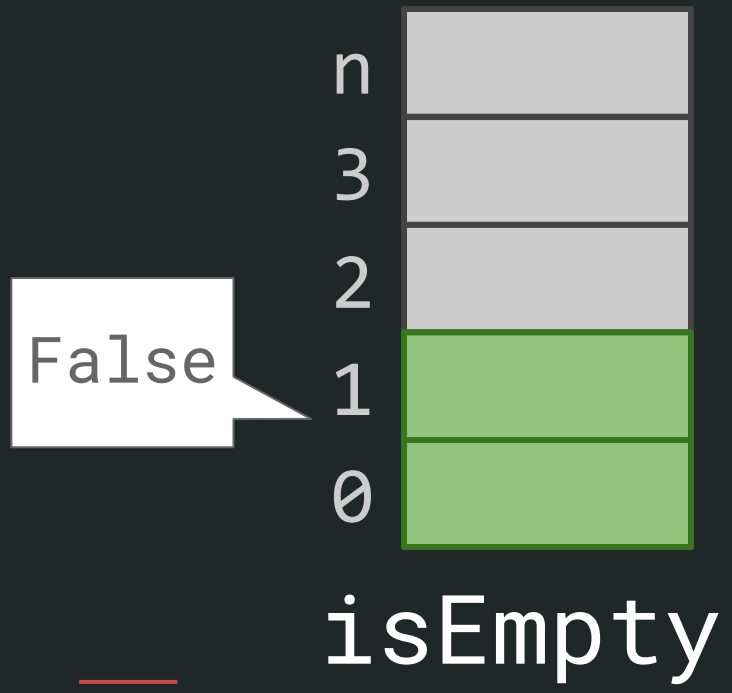
Example: Stack

- Abstract Data Type
- LIFO



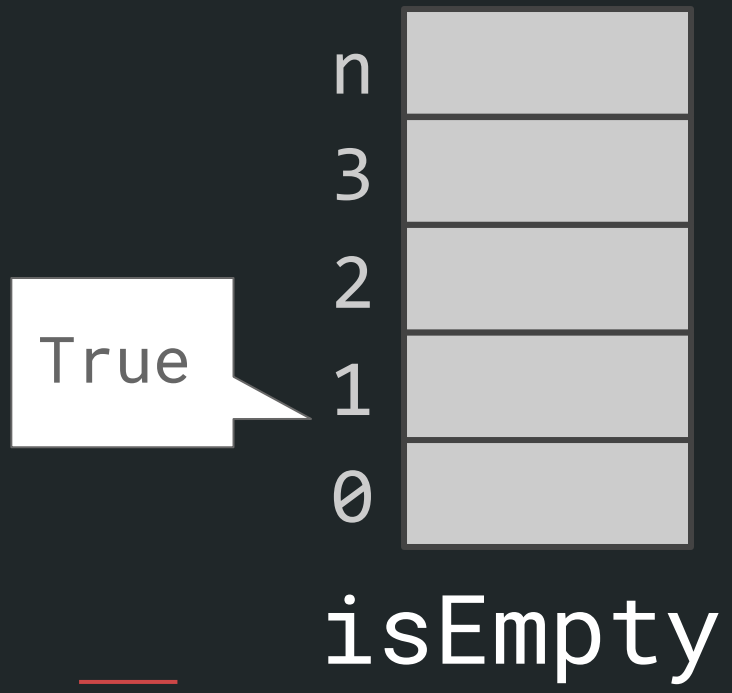
Example: Stack

- Abstract Data Type
- LIFO



Example: Stack

- Abstract Data Type
- LIFO



Specification: STACK

- **Init** : returns a new, empty stack
- **Push** : pushes an element to a given stack
- **Pop** : returns the given stack with the top-most element removed
- **Top** : returns the top-most element
- **isEmpty** : returns true if list is empty, false otherwise

Can we do better?

Specification: STACK

- **Init** : returns a new, empty stack
- **Push** : pushes an element to a given stack
- **Pop** : returns the given stack with the top-most element removed
- **Top** : returns the top-most element
- **isEmpty** : returns true if list is empty, false otherwise

In ACT ONE

```
type STACK is
    sorts
        Stack
        Int

    constructors
        init:            -> Stack;
        push: Int, Stack -> Stack;

    functions
        vars
            s    : Stack;
            d, e : Int;

    func top: Stack -> Int;
        top(push(d,s)) = d;

    func pop: Stack -> Stack;
        pop(push(d,s)) = s;

    func isempty: Stack(D) -> Bool;
        isempty(init) = true;
        isempty(push(d,s)) = false;

endtype
```

Specification: STACK

- **Init** : returns a new, empty stack
- **Push** : pushes an element to a given stack
- **Pop** : returns the given stack with the top-most element removed
- **Top** : returns the top-most element
- **isEmpty** : returns true if list is empty, false otherwise

In ACT ONE

```
type STACK is
    sorts
        Stack
        Int

    constructors
        init:              -> Stack;
        push: Int, Stack -> Stack;

    functions
        vars
            s     : Stack;
            d, e : Int;

    func top: Stack -> Int;
        top(push(d,s)) = d;

    func pop: Stack -> Stack;
        pop(push(d,s)) = s;

    func isempty: Stack(D) -> Bool;
        isempty(init) = true;
        isempty(push(d,s)) = false;

endtype
```

Our version of ACT ONE

```
Type STACK is
Imports
  ref: BOOLEAN
  ref: INTEGER
Sorts
  new Stack
Operators
  ctor: init : -> Stack
  ctor: push : Int Stack -> Stack
  oper: top : Stack -> Int
  oper: pop : Stack -> Stack
  oper: isEmpty : Stack -> Bool
Variables
  var s : Stack
  var d : Int
  var e : Int
Axioms
  axiom: top(push(:d, :s)) = :d
  axiom: pop(push(:d, :s)) = :s
  axiom: isEmpty(init()) = True()
  axiom: isEmpty(push(:d, :s)) = False()
```

Original ACT ONE

```
type STACK is
    sorts
        Stack
        Int

    constructors
        init:            -> Stack;
        push: Int, Stack -> Stack;

    functions
        vars
            s    : Stack;
            d, e : Int;

    func top: Stack -> Int;
        top(push(d,s)) = d;

    func pop: Stack -> Stack;
        pop(push(d,s)) = s;

    func isempty: Stack(D) -> Bool;
        isempty(init) = true;
        isempty(push(d,s)) = false;

endtype
```

Our version of ACT ONE

```
Type STACK is
Imports
  ref: BOOLEAN
  ref: INTEGER
Sorts
  new Stack
Operators
  ctor: init : -> Stack
  ctor: push : Int Stack -> Stack
  oper: top : Stack -> Int
  oper: pop : Stack -> Stack
  oper: isEmpty : Stack -> Bool
Variables
  var s : Stack
  var d : Int
  var e : Int
Axioms
  axiom: top(push(:d, :s)) = :d
  axiom: pop(push(:d, :s)) = :s
  axiom: isEmpty(init()) = True()
  axiom: isEmpty(push(:d, :s)) = False()
```

Original ACT ONE

```
type STACK is
    sorts
        Stack
        Int

    constructors
        init:              -> Stack;
        push: Int, Stack -> Stack;

    functions
        vars
            s     : Stack;
            d, e : Int;

    func top: Stack -> Int;
        top(push(d,s)) = d;

    func pop: Stack -> Stack;
        pop(push(d,s)) = s;

    func isempty: Stack(D) -> Bool;
        isempty(init) = true;
        isempty(push(d,s)) = false;

endtype
```

Our version of ACT ONE

```
Type STACK is
Imports
  ref: BOOLEAN
  ref: INTEGER
Sorts
  new Stack
Operators
  ctor: init : -> Stack
  ctor: push : Int Stack -> Stack
  oper: top : Stack -> Int
  oper: pop : Stack -> Stack
  oper: isEmpty : Stack -> Bool
Variables
  var s : Stack
  var d : Int
  var e : Int
Axioms
  axiom: top(push(:d, :s)) = :d
  axiom: pop(push(:d, :s)) = :s
  axiom: isEmpty(init()) = True()
  axiom: isEmpty(push(:d, :s)) = False()
```

Original ACT ONE

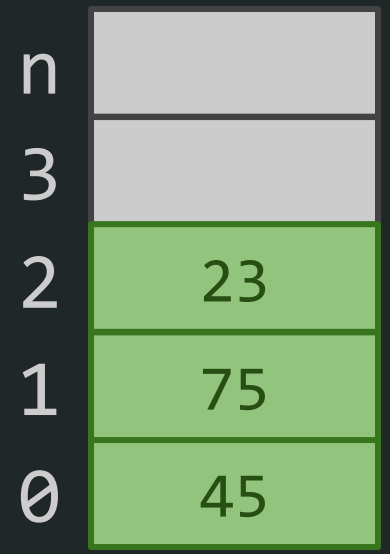
```
type STACK is
    sorts
        Stack
        Int

    constructors
        init:              -> Stack;
        push: Int, Stack -> Stack;

    functions
        vars
            s     : Stack;
            d, e : Int;

    func top: Stack -> Int;
        top(push(d,s)) = d;

    func pop: Stack -> Stack;
        pop(push(d,s)) = s;

    func isempty: Stack(D) -> Bool;
        isempty(init) = true;
        isempty(push(d,s)) = false;

endtype
```

What are these?

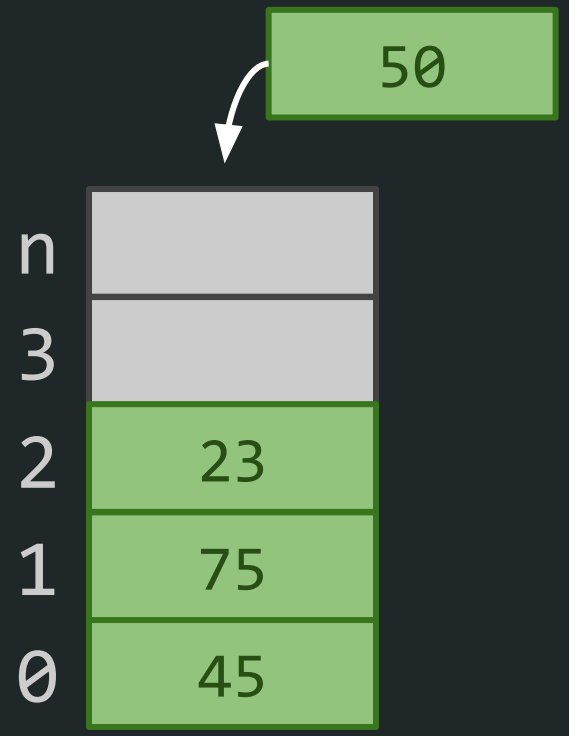
```
Type STACK is
Imports
  ref: BOOLEAN
  ref: INTEGER
Sorts
  new Stack
Operators
  ctor: init : -> Stack
  ctor: push : Int Stack -> Stack
  oper: top : Stack -> Int
  oper: pop : Stack -> Stack
  oper: isEmpty : Stack -> Bool
Variables
  var s : Stack
  var d : Int
  var e : Int
Axioms
  axiom: top(push(:d, :s)) = :d
  axiom: pop(push(:d, :s)) = :s
  axiom: isEmpty(init()) = True()
  axiom: isEmpty(push(:d, :s)) = False()
```

Example: Stack

This is Stack S

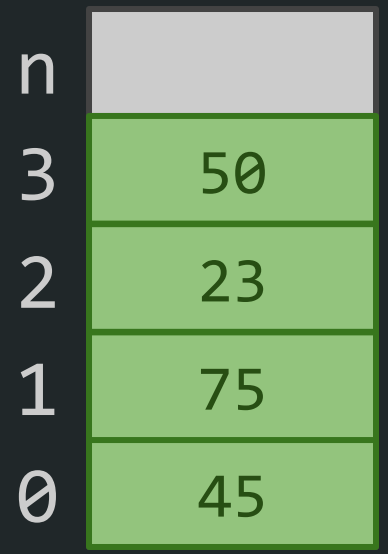


Example: Stack

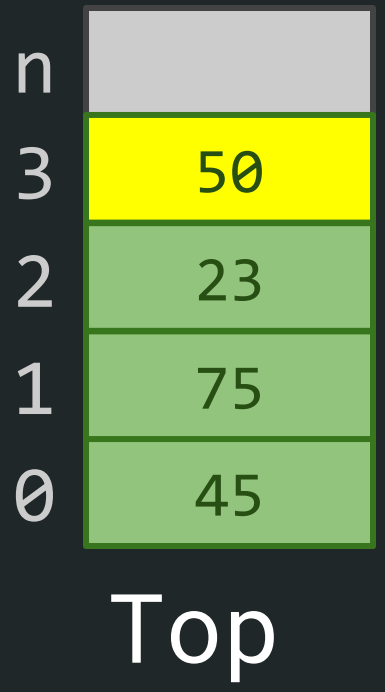


Push

Example: Stack



Example: Stack



Axioms

```
Top(Push(50,s)) = 50
```

Axioms

```
Top(Push(50,s)) = 50
```

```
Top(Push(x,s)) = x
```

Axioms

```
Type STACK is
Imports
  ref: BOOLEAN
  ref: INTEGER
Sorts
  new Stack
Operators
  ctor: init : -> Stack
  ctor: push : Int Stack -> Stack
  oper: top : Stack -> Int
  oper: pop : Stack -> Stack
  oper: isEmpty : Stack -> Bool
Variables
  var s : Stack
  var d : Int
  var e : Int
Axioms
  axiom: top(push(:d, :s)) = :d
  axiom: pop(push(:d, :s)) = :s
  axiom: isEmpty(init()) = True()
  axiom: isEmpty(push(:d, :s)) = False()
```

Term Rewriting Logic

Terms

Examples:

- Top(Push(50,Init()))
- Init()
- isEmpty(Push(50,Init()))

Rewriting

```
Top(Push(50,Push(40,Init())))
```

Rewriting

```
Top(Push(50,Push(40,Init())))
```

```
Variables
  var s : Stack
  var d : Int
  var e : Int
Axioms
  axiom: top(push(:d, :s)) = :d
  axiom: pop(push(:d, :s)) = :s
  axiom: isEmpty(init()) = True()
  axiom: isEmpty(push(:d, :s)) = False()
```

Rewriting

`Top(Push(50,Push(40,Init()))) = 50`

```
Variables
  var s : Stack
  var d : Int
  var e : Int
Axioms
  axiom: top(push(:d, :s)) = :d
  axiom: pop(push(:d, :s)) = :s
  axiom: isEmpty(init()) = True()
  axiom: isEmpty(push(:d, :s)) = False()
```

Rewriting - BFS & DFS

2 ways to evaluate terms:

- Inside-out = DFS
- Outside-in = BFS

```
Top(Push(50,Push(40,Init())))
```

```
Top(Push(50,Push(40,Init())))
```

Rewriting - BFS & DFS

2 ways to evaluate terms:

- Inside-out = DFS
- Outside-in = BFS

```
Top(Push(50,Push(40,Init())))
```

```
Top(Push(50,Push(40,Init())))
```

Rewriting - BFS & DFS

2 ways to evaluate terms:

- Inside-out = DFS
- Outside-in = BFS

```
Top(Push(50,Push(40,Init())))
```

```
Top(Push(50,Push(40,Init())))
```

Rewriting - BFS & DFS

2 ways to evaluate terms:

- Inside-out = DFS
- Outside-in = BFS

```
Top(Push(50,Push(40,Init())))
```

```
Top(Push(50,Push(40,Init())))
```

Rewriting - BFS & DFS

2 ways to evaluate terms:

- Inside-out = DFS
- Outside-in = BFS

```
Top(Push(50,Push(40,Init())))
```

```
Top(Push(50,Push(40,Init())))
```

Rewriting - BFS & DFS

2 ways to evaluate terms:

- Inside-out = DFS
- Outside-in = BFS

```
Top(Push(50,Push(40,Init())))
```

```
Top(Push(50,Push(40,Init())))
```

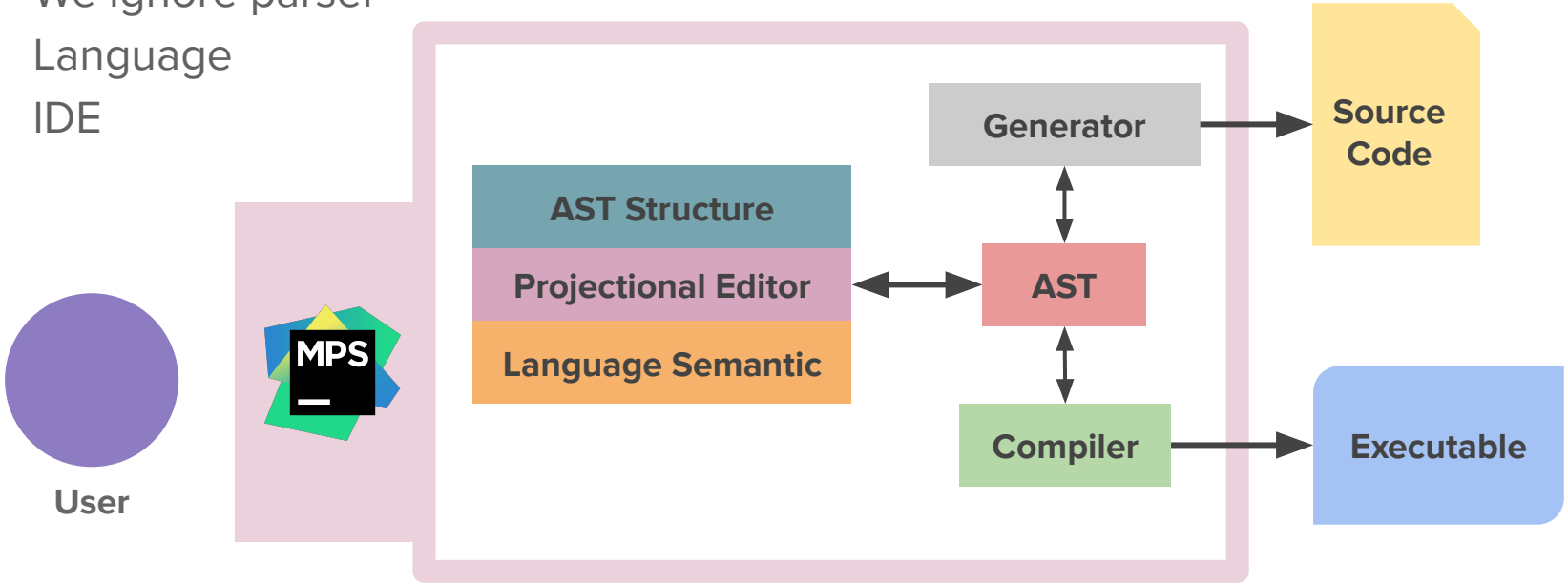
IDE

MPS - Meta-Programming System

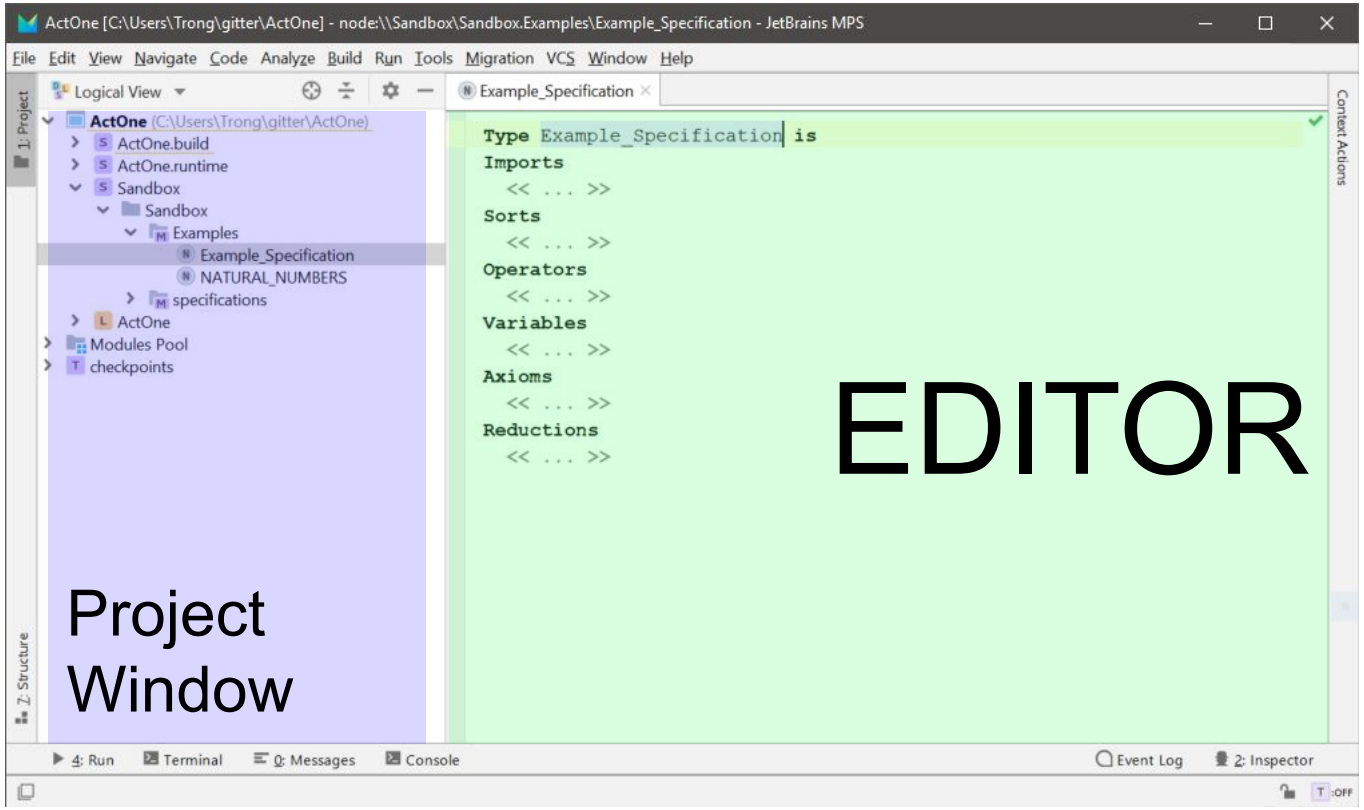
- JetBrains
- Language workbench
- Domain Specific Languages

How MPS works

- We ignore parser
- Language
- IDE



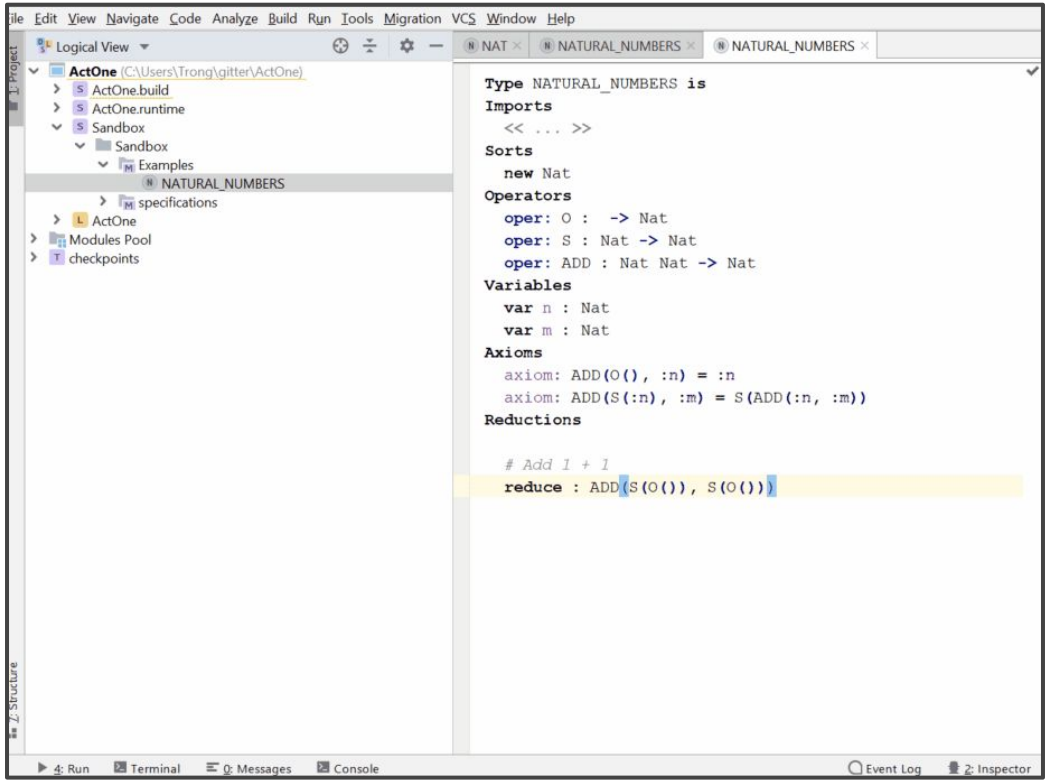
Our result


ActOne [C:\Users\Trong\gitter\ActOne] - node:\\Sandbox\Sandbox.Examples\Example_Specification - JetBrains MPS
File Edit View Navigate Code Analyze Build Run Tools Migration VCS Window Help
Logical View
Example_Specification
ActOne (C:\Users\Trong\gitter\ActOne)
ActOne.build
ActOne.runtime
Sandbox
Sandbox
Examples
Example_Specification
NATURAL_NUMBERS
specifications
ActOne
Modules Pool
checkpoints
Type Example_Specification is
Imports
<< ... >>
Sorts
<< ... >>
Operators
<< ... >>
Variables
<< ... >>
Axioms
<< ... >>
Reductions
<< ... >>
EDITOR
Project Window
4: Run
Terminal
0: Messages
Console
Event Log
2: Inspector
Context Actions
1: Project
7: Structure

Example Usage

```
Type <no name> is
Imports
  << ... >>
Sorts
  << ... >>
Operators
  << ... >>
Variables
  << ... >>
Axioms
  << ... >>
Reductions
  << ... >>
```

Feature: Execution



Feature: Vertical Formatting

```
reduce : validate(mkPuzzle(ships(3, 2, 1, 0, 0, 0), horizontal(3, 0, 1, 3, 2, 1), vertical(2, 0, 2, 3, 1, 2),                    ))
                          mkBoard(mkLine(X(), W(), X(), X(), X(), X()), mkLine(W(), W(), W(), W(), W(), W()),     )
                                      mkLine(X(), W(), X(), X(), X(), X()), mkLine(W(), W(), W(), D(), W(), X()),
                                      mkLine(X(), W(), X(), X(), X(), X()), mkLine(X(), W(), W(), X(), X(), X())
```

Feature: Duplicate Name Check

```
  << ... >>
Sorts
  new Nat
  new Int
Operators
  << ... >>
```

Feature: Intentions

```
Reductions

  # Add 1 + 1
  reduce : ADD(S(O()), S(O()))
```

Feature: Operation Parameter Check

Error: Wrong number of parameters! Expected: 2, Actual: 1

```
reduce : ADD(S(O()))
```

Error: type Bool is not a subtype of Nat

```
reduce : ADD(True(), False())
```

Functional Requirements Revisited

Title	Must	Description
Implement Prefix Operations	YES	Allow ACT ONE to work with Prefix Operations.
Implement Infix Operations	NO	Allow ACT ONE to work with Infix Operations.
Let-notation	NO	Allow users to declare let-declarations.
If-Then-Else	NO	Allow users to declare if-then-else statements.
Allow specification extension	NO	Allow specification to extend another specification.
Implement Operator-Precedence	NO	Allow ACT ONE to work with Operations while handling Operator-Precedence logic.
Implement Associativeness	NO	Allow ACT ONE to work with Infix Operations with associativeness.
Implement Communativeness	NO	Allow ACT ONE to work with binary Operations while handling Communativeness.
UTF-8-compatible Syntax	YES	Allow the syntax of ACT ONE to be written in plain-text editors (UTF-8).
Implement Subsort	NO	Allow ACT ONE to work with subsorts.
Universal Sort	NO	Allow ACT ONE to work with universal sort, a sort that conforms with all sorts.
Parameterised Specification	NO	Make it possbile for ACT ONE to work with parameterised specifications.
Overloaded Operations Check	NO	Allow the IDE to see overloaded operations.
Duplicate Operations Check	NO	Allow the IDE to detect duplicate Operations.
Parameter Type Check	NO	Allow the IDE to detect wrongly typed parameters.
Parameter Number Check	NO	Allow the IDE to detect Operation-calls with wrong number of parameters.
Predefined Boolean	NO	Allow ACT ONE to use Boolean Specification by default.
Predefiend Integer	NO	Allow ACT ONE to use Integer Specification by default.
Predefined List	NO	Allow ACT ONE to use List Specification by default.
Step-Reduction	YES	Allow users to initiate n-Step-Reduction, where n is an abritrary, positive number.
Exhaustive Reduction	YES	Allow users to initiate Exhaustive-Reduction.
BFS Reduction	YES	Allow users to initiate Breadth-First Reduction.
DFS Reduction	YES	Allow users to initiate Depth-First Reduction.
Tuple Reduction	NO	Allow users to initiate double parameterised Reductions, e.g. Breadth-First 6-Step-Reduction.
Infinite-Reduction-Handler	NO	Allow the rewrite system to prematurely end a reduction that diverges towards infiity.

Functional Requirements Revisited

Title	Must	Description
Implement Prefix Operations	YES	Allow ACT ONE to work with Prefix Operations.
Implement Infix Operations	NO	Allow ACT ONE to work with Infix Operations.
Let-notation	NO	Allow users to declare let-declarations.
If-Then-Else	NO	Allow users to declare if-then-else statements.
Allow specification extension	NO	Allow specification to extend another specification.
Implement Operator-Precedence	NO	Allow ACT ONE to work with Operations while handling Operator-Precedence logic.
Implement Associativeness	NO	Allow ACT ONE to work with Infix Operations with associativeness.
Implement Communativeness	NO	Allow ACT ONE to work with binary Operations while handling Communativeness.
UTF-8-compatible Syntax	YES	Allow the syntax of ACT ONE to be written in plain-text editors (UTF-8).
Implement Subsort	NO	Allow ACT ONE to work with subsorts.
Universal Sort	NO	Allow ACT ONE to work with universal sort, a sort that conforms with all sorts.
Parameterised Specification	NO	Make it possbile for ACT ONE to work with parameterised specifications.
Overloaded Operations Check	NO	Allow the IDE to see overloaded operations.
Duplicate Operations Check	NO	Allow the IDE to detect duplicate Operations.
Parameter Type Check	NO	Allow the IDE to detect wrongly typed parameters.
Parameter Number Check	NO	Allow the IDE to detect Operation-calls with wrong number of parameters.
Predefined Boolean	NO	Allow ACT ONE to use Boolean Specification by default.
Predefiend Integer	NO	Allow ACT ONE to use Integer Specification by default.
Predefined List	NO	Allow ACT ONE to use List Specification by default.
Step-Reduction	YES	Allow users to initiate n-Step-Reduction, where n is an abritrary, positive number.
Exhaustive Reduction	YES	Allow users to initiate Exhaustive-Reduction.
BFS Reduction	YES	Allow users to initiate Breadth-First Reduction.
DFS Reduction	YES	Allow users to initiate Depth-First Reduction.
Tuple Reduction	NO	Allow users to initiate double parameterised Reductions, e.g. Breadth-First 6-Step-Reduction.
Infinite-Reduction-Handler	NO	Allow the rewrite system to prematurely end a reduction that diverges towards infiity.

Discussion

- High Coupling

Discussion

- High Coupling
- Usability
  - Available assets
  - Feel
  - No derived MPS

```
Type <no name> is
Imports
  << ... >>
Sorts
  << ... >>
Operators
  << ... >>
Variables
  << ... >>
Axioms
  << ... >>
Reductions
  << ... >>
```

Discussion

- High Coupling
- Usability
  - Available assets
  - Feel
  - No derived MPS
- Correctness
  - Term rewriting logic
  - No-large scale testing

```
Type <no name> is
Imports
  << ... >>
Sorts
  << ... >>
Operators
  << ... >>
Variables
  << ... >>
Axioms
  << ... >>
Reductions
  << ... >>
```

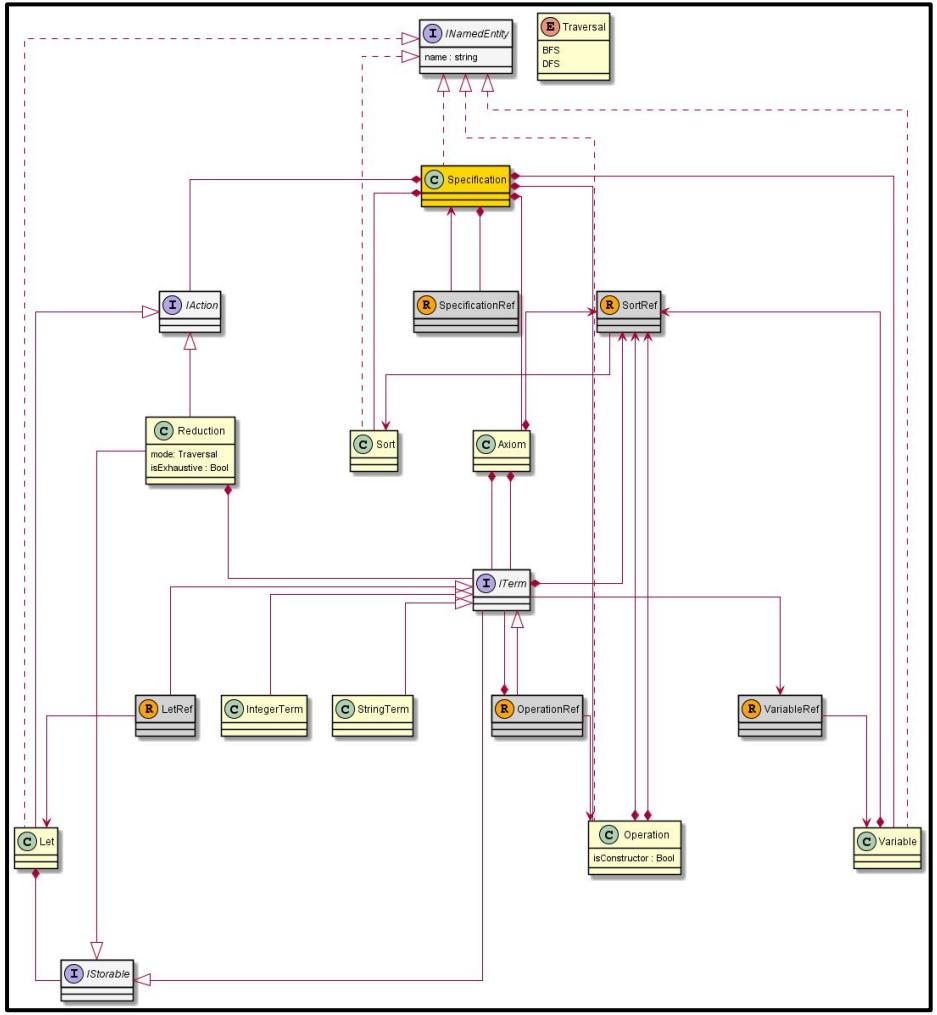
Conclusion

- Sufficient version of ACT ONE
- Working term rewriting system
- Usable IDE
- Befitting for DAT233

Questions?

- Need more elaboration on MPS?
- Want to see the product in action?
- Ask about Battleship!
- What does BFS and DFS stand for?
- Wonder why I chose to do this project?
- Something in the thesis that is unclear?

UML


INamedEntity
name : string
Traversal
BFS
DFS
Specification
IAction
SpecificationRef
SortRef
Reduction
mode: Traversal
isExhaustive : Bool
Sort
Axiom
ITerm
LetRef
IntegerTerm
StringTerm
OperationRef
VariableRef
Let
Operation
isConstructor : Bool
Variable
IStorable