

BNF Specification for π -ADL.NET

Notation: Here we use the single quote “'” to denote a comment line. Non terminal symbols start with small letters, and terminal symbols start with capital letters. Other than that, the standard BNF notation is followed.

```
' --- Program
program := [explicitBehaviourDeclaration | abstractionDeclaration]+

' --- Types
typeDeclaration := valueTypeDeclaration | namedExplicitBehaviourDeclaration
valueTypeDeclaration := typeName ":" valueType ";"
typeName := [typeName "::"] identifier
inputConnection := "in"
outputConnection := "out"
identifier := Letter LettersNumbersUnderscores 'regex for LettersNumbersUnderscores: (A-Za-z) (A-Za-z0-9_)*
type := valueType | BehaviourType
valueType := baseType | constructedType | connectionType
baseType := Void | Float | String | Boolean | Integer
float := [sign] unsignedFloat
unsignedFloat := UnsignedInteger ["." UnsignedInteger]
sign := "+" | "-"
string := DoubleQuote AnythingExceptDoubleQuote DoubleQuote
boolean := "true" | "false"
integer := [sign] UnsignedInteger
value := string | float | boolean | integer | "" | tupleValue | viewValue
constructedType := tuple | view
tuple := "tuple" "[" valueType ("," valueType)* "]"
tupleValue := "tuple" "(" (value ";" )* value ")"
view := "view" "[" identifier ":" valueType ("," identifier ":" valueType)* "]"
viewValue := "view" "(" (identifier ":" value ";" )* identifier ":" value ")"
union := "union" "[" valueType ("," valueType)* "]"
unionValue := "union" "(" valueType ":" value ")"
connectionType := Connection "[" [valueType] "]"
behaviourTypeDeclarationPrefix := identifier " names "

' --- Behaviours
behaviourDeclaration := explicitBehaviourDeclaration | implicitBehaviourDeclaration
explicitBehaviourDeclaration := namedExplicitBehaviourDeclaration | unnamedExplicitBehaviourDeclaration
namedExplicitBehaviourDeclaration := behaviourTypeDeclarationPrefix "behaviour" "{" behaviourBody "}"
[renamingClause]
unnamedExplicitBehaviourDeclaration := "behaviour" "{" behaviourBody "}" [renamingClause]
implicitBehaviourDeclaration := "{" behaviourBody "}" [renamingClause]
behaviourBody := declarations (statementBlock)* terminalBlock

' --- Statements
declarations := (restrictStatement | connectionDeclaration | valueTypeDeclaration)*
argument := [restrictStatement | connectionDeclaration | valueTypeDeclaration]
restrictStatement := "restrict" connectionDeclaration
connectionDeclaration := typeName ":" connectionType ";"
prefix := outputPrefix | inputPrefix | silentPrefix | matchPrefix | pseudoApplication
outputPrefix := "via " typeName | outputConnection " send " (typeName | value) ";"
inputPrefix := "via " typeName | inputConnection " receive " typeName ";"
silentPrefix := "unobservable;";
matchPrefix := "if (" logicalExpression ") do " (done | unobservable | prefix | assignment | "{" statementBlock "}")
whilePrefix := "while (" logicalExpression ") do " (done | unobservable | prefix | assignment | "{" statementBlock "}")
done := "done ;" / generates opcode ret
unobservable := "unobservable ;";
renamingClause := "where {" [connectionRename]* lastConnectionRename "}"
connectionRename := lastConnectionRename " ;"
```

```

lastConnectionRename := identifier "renames" identifier 'identifier must be a connectionType
projection := "project" constructedType "as" typeName ("," typeName)* ";"

' --- Expression
expression := logicalExpression
logicalOp := relOp | andOp | orOp | notOp
notOp := "!" 'precedence 1
relOp := "> | >= | < | <= | == | !=" 'precedence 2
typeEqOp := "#=" 'precedence 2
andOp := "&&" 'precedence 3
orOp := "||" 'precedence 4
logicalExpression := orFactor [ orOp logicalExpression ]
orFactor := andFactor [ andOp orFactor ]
andFactor := ([ notOp ] relFactor | notOp "(" relFactor ")") [eqOp relFactor]
notFactor := (" logicalExpression") | boolean )
relFactor := (" logicalExpression") | polymorph typeEqOp polymorph | (arithmeticExpression relOp
arithmeticExpression) | (string eqOp string) | boolean

' --- Arithmetic
arithmeticOp := addOp | multiplyOp
addOp := "+ | -"
multiplyOp := "*" | "/" | "%"
arithmeticExpression := addFactor [arithmeticExpressionRHS]
arithmeticExpressionRHS = addOp addFactor [arithmeticExpressionRHS]
addFactor = term [addFactorRHS]
addFactorRHS = multiplyOp term [ addFactorRHS ]
term := [addOp] (bool | string | integer | float | "(" arithmeticExpression | logicalExpression ")")

' --- Assignment
assignment := identifier "=" expression ";" 'valid for basic and constructed types

' --- Blocks
behaviourHandle := (enclosedBlock | BehaviourType)
enclosedBlock := "{" block "}"
block := (chooseBlock | composeBlock | replicateBlock | statementBlock | selectBlock)
terminalBlock := (chooseBlock | composeBlock | replicateBlock)
chooseBlock := "choose {" block [" or " block]+ "}"
composeBlock := "compose {" block [" and " block]+ "}"
replicateBlock := "replicate " behaviourHandle
statementBlock := (prefix | assignment | projection)*
selectBlock := "select " union "{" caseBlock "}"
caseBlock := (" case " valueType " do " statementBlock)+

' --- Comment
comment := "/" AnyTypeOfCharacters Newline

' --- Abstraction
abstractionDeclaration := "value " identifier " is abstraction ( " argument " ) " enclosedAbstractionBody
enclosedAbstractionBody := "{" behaviourBody "}"
pseudoApplication := "via " Abstraction " send " (typeName | value) [renamingClause] ";"
dynamicPseudoApplication := "via dynamic(" string ") send " (typeName | value) [renamingClause] ";"

```