// **Syntax and Semantics Rewrite Rules** for the **K-Taint** in the K Framework.

module LANG-SYNTAX

```
  syntax DType ::= "int" | "void" | "float" | "char"

  syntax Decl ::= DType AExps ";"
            | DType Id "(" Params ")" Block

  syntax AExp ::= Int | Bool | String | Id | "%"Id
                | Id "++"
                | Id "--"
                | "&"Id
                | "*"AExp                                   [strict]
                 | "(" AExp ")"                             [bracket]
                > Id "[" AExp "]"
                | "&"Id "[" Id "]"
                | Id "[" "]"
                | AExp "(" AExps ")"                        [strict]
                 | "read" "(" ")"
                > left:
                 AExp "+" AExp                              [strict, left]
                |AExp "*" AExp                              [strict]
                | AExp "/" AExp                             [strict]
                | AExp "-" AExp                             [left]
                | AExp "MINUS" AExp                         [strict,left]
                | AExp "^" AExp
                | AExp "XOR" AExp                           [strict]
                 > AExp "=" AExp                            [strict(2)]

        syntax BExp ::= AExp "<=" AExp                      [strict]
                      | AExp "==" AExp                      [strict]
                      | AExp "<" AExp                       [strict]
                      | AExp ">=" AExp                      [strict]
                      | AExp ">" AExp                       [strict]
                      | AExp "!=" AExp                      [strict]
                      | "!" BExp
                      | BExp "&&" BExp                      [strict]
                      | BExp "||" BExp                      [strict]

        syntax Condt ::=  "if" "(" BExp ")" Block           [strict(1)]
                      | "if" "(" BExp ")" Block  "else" Block [strict(1)]
                      | "while" "(" BExp ")" Block          [strict(1)]
                      | "for" "(" Stmts BExp ";" AExp ")" Block
```

```
        syntax Param ::= DType AExp
        syntax Params ::= List{Param,","}
        syntax Ids  ::= List{Id,","}
        syntax AExps ::= List{AExp,","}          [strict]

        syntax Block ::= "{" "}"
                  | "{" Stmts "}"


        syntax Stmt ::= Decl | Block | "main" "(" ")" Block | Condt | Id "=" "&" Id ";"
                        | AExp ";"                                              [strict]
                        | "return" AExp ";"                                     [strict]
                         | "return" ";"
                         | "printf" "(" AExps ")" ";"                           [strict]
                        | "scanf" "(" AExp "," AExp ")" ";"

         syntax Stmts ::= Stmt
                        | Stmts Stmts                                           [right]

        rule D:DType E1:AExp, E2:AExp, Es:AExps; => D E1; D E2, Es;             [macro]
        rule D:DType X:Id = E; => D X; X = E;                                   [macro]
        rule  D:DType X:Id[ ]=V; => D X; X = V ;                                [macro]
        rule X:Id ++ => X = X + 1                                               [macro]
        rule X:Id -- => X = X - 1                                               [macro]
        rule &X:Id => X                                                         [macro]
        rule *X:Id => X                                                         [macro]
        rule scanf(E1:AExp,&X:Id); => X = read();                               [macro]

syntax Type ::= "taint" | "untaint"  | "zero"

endmodule     //end of the syntax module


module LANG

  imports  LANG-SYNTAX

        syntax K ::= Map "[" K "]"          [function,hook(MAP.lookup), klabel(Map:lookup)]
        syntax AExp ::= AExp "union" K                  [strict,left]
                        | AExp "union" AExp             [strict,left]
        syntax AExp ::= restore(K)
                        | approax(K)
                        | restoreenv(K)
                        | store(K)
```

```
syntax Val ::=  String | Type
                | a(Int,Int)
          |      lambda(Params,Stmts)
                | pointsTo(Id)

syntax Vals ::= List{Val, ","}
syntax AExp ::= Val
syntax KResult ::= Val
                 | Vals

configuration <T color="red">
                <k color="green"> $PGM:Stmts </k>
                <control color="cyan">
                        <fstack color="blue"> .List </fstack>
                </control>
                <tenv color="violet"> .Map </tenv>
                <gtenv color="pink"> .Map </gtenv>
                 <temp>.Map</temp>
                <lambda color="white"> .Map </lambda>
                <ptr-alias color="orange">
                        <alias color="yellow">.Map</alias>
                        <ptr color="white">.Map</ptr>
                </ptr-alias>
                <context> .Map </context>
                <in color="magenta" stream="stdin"> .List </in>
                <out color="brown" stream="stdout"> .List </out>
          </T>

rule <k> cont X:Id; => . ...</k> <context> Contxt => Contxt[X <- untaint] </context>
rule <k> D:DType X:Id; => . ...</k> <tenv> Env => Env[X <- untaint] </tenv>

rule <k> D:DType *X:Id; => . ...</k> <ptr> Var:Map => Var[X <- untaint] </ptr>
rule <k> D:DType X:Id[N:Int]; => . ...</k> <tenv> Env => Env[X <- untaint] </tenv>

rule <k> D:DType F(Xs) {S:Stmts} => . ...</k>
     <lambda>... .Map => F |-> lambda(Xs, S) ...</lambda>

rule <k> F:Id => V ...</k> <lambda>... F |-> V ...</lambda> [funclookup]

rule <k> A:Id[E:AExp] => V ...</k><tenv>... A |-> V ...</tenv> [arraylookup]

rule <k> main () {B} => B ...</k> <gtenv> Rho </gtenv>  [structural]
```

// Rules for Conditional construct and Loops

rule <k> if (B:Type) {S} => B ~> {S} ~> restore(Mu) ...</k><context>Mu</context>

rule <k> if(B:Type) {S1} else {S2} => if(B) {S1} else {} ~> restore(Mu) ~> restoreenv(Rho)~> if(B){} else {S2} ~> restore(Mu) ...</k> <context> Mu </context> <tenv> Rho </tenv>

rule <k> if(B) {S} else {} => B ~> {S} ...</k>
rule <k> if(B) {} else {S} => B ~> {S} ...</k>

rule <k> while (B:Type) {S} => B ~> {S} ~> restore(Mu) ...</k><context>Mu</context>
rule <k>for(S1:StmtsB;A:AExp){S}=>S1~>B~>{A;S}~>restore(Mu)... </k><context>Mu </context>

rule <k> B ~> {S} => {S} ...</k><context>Pc |-> T => Pc |-> (T union B:Type)</context>
rule <k> B ~> {{ } S} => {} ~> {S} ...</k><context>Pc |-> T => Pc |-> (T union B:Type)</context>

rule <k> {X1[E] = X2;} => {X1 = X2;} ...</k>
rule <k> {X1[E] = X2; S} => {X1 = X2; S} ...</k>
rule <k> {*X1= X2;} => {X1 = X2;} ...</k>
rule <k> {*X1 = X2; S} => {X1 = X2; S} ...</k>

rule <k>{ X = Y; } => X = ((Y union Pc) union Rho[X]) ; ...</k><context>Pc |-> V</context> <tenv>Rho</tenv>

rule <k>{ X = Y; S} => X = ((Y union Pc) union Rho[X]) ; {S} ...</k><context>Pc |-> V</context> <tenv>Rho</tenv>

rule <k> {if (BExp) {S}}=> if (BExp) {S} ...</k>
rule <k> {if (BExp) {S1} S2}=> if (BExp){S1} {S2} ...</k>

rule <k> {if (BExp) {S1} else {S2}}=> if (BExp) {S1} else {S2} ...</k>
rule <k> {if (BExp) {S1} else {S2} S3}=> if (BExp) {S1} else {S2} {S3} ...</k>

rule <k> {while (BExp) {S}}=> while (BExp) {S} ...</k>
rule <k> {while (BExp) {S1} S2}=> while (BExp){S1} [S2} ...</k>

rule <k> {for (Stmts; BExp ; AExp;) {S}}=> for (Stmts; BExp ; AExp;) {S} ...</k>
rule <k> {for (Stmts; BExp ; AExp;) {S1} S2}=> for (Stmts; BExp ; AExp;) {S1} {S2}...</k>

```
syntax AExp ::= Type

rule _:Int => untaint
rule N:Int => untaint when N =/=K 0
rule N:Int => zero when N ==K 0
```

// rules for ***Division*** operation.

```
rule taint / taint => taint
rule taint / untaint => taint
rule taint / zero => untaint
rule untaint / taint => taint
rule untaint / untaint => untaint
rule untaint / zero => untaint
rule zero / taint => untaint
rule zero / zero => untaint
rule zero / untaint => untaint
```

// rules for ***Multiplication*** operation.

```
rule taint * taint => taint
rule taint * untaint => taint
rule taint * zero => untaint
rule untaint * taint => taint
rule untaint * untaint => untaint
rule untaint * zero => untaint
rule zero * taint => untaint
rule zero * zero => untaint
rule zero * untaint => untaint
```

// rules for ***XOR*** operation.

```
rule X:Id ^ Y:Id =>  untaint  when X ==K Y
rule X:Id ^ Y:Id =>  X:Id XOR Y:Id when X =/=K Y
rule taint XOR taint => taint
rule taint XOR untaint => taint
rule taint XOR zero => taint
rule zero XOR taint => taint
rule zero XOR untaint => untaint
rule zero XOR zero => untaint
rule untaint XOR taint => taint
rule untaint XOR untaint => untaint
rule untaint XOR zero =>  untaint
```

// rules for ***Addition*** operation.

```
rule taint + taint => taint
rule taint + untaint => taint
rule taint + zero => taint
rule untaint + taint => taint
rule untaint + untaint => untaint
rule untaint + zero => untaint
rule zero + taint => taint
rule zero + untaint => untaint
rule zero + zero => zero
```

// rules for ***Subtraction*** operation.

```
rule X:Id - Y:Id => untaint when X ==K Y
rule X:Id - Y:Id => X:Id MINUS Y:Id when X =/=K Y
rule X:Id - N:Int => X:Id MINUS N:Int
rule I:Int - X:Id => I:Int MINUS X:Id
rule taint MINUS taint => taint
rule taint MINUS untaint => taint
rule taint MINUS zero => taint
rule untaint MINUS taint => taint
rule untaint MINUS untaint => untaint
rule untaint MINUS zero => untaint
rule zero MINUS taint => taint
rule zero MINUS untaint => untaint
rule zero MINUS zero => untaint
```

// Rules for **Less than equal to** (<=) Expressions.

```
syntax BExp ::= Type
rule untaint <= untaint => untaint
rule untaint <= taint => taint
rule untaint <= zero => untaint
rule taint <= untaint => taint
rule taint <= taint => taint
rule taint <= zero => taint
rule zero <= taint => taint
rule zero <= untaint => untaint
rule zero <= zero => untaint
```

// Rules for **Greater than equal to** (>=) Expressions.

```
rule untaint >= untaint => untaint
rule untaint >= taint => taint
rule untaint >= zero => untaint
rule taint >= untaint => taint
rule taint >= taint => taint
rule taint >= zero => taint
rule zero >= taint => taint
rule zero >= untaint => untaint
rule zero >= zero => untaint
```

// Rules for **Less than** (<) Expressions.

```
rule untaint < untaint => untaint
rule untaint < taint => taint
rule untaint < zero => untaint
rule taint < untaint => taint
rule taint < taint => taint
rule taint < zero => taint
rule zero < taint => taint
rule zero < untaint => untaint
rule zero < zero => untaint
```

// Rules for **Greater than** (>) Expressions.

```
rule untaint > untaint => untaint
rule untaint > taint => taint
rule untaint > zero => untaint
rule taint > untaint => taint
rule taint > taint => taint
rule taint > zero => taint
rule zero > taint => taint
rule zero > untaint => untaint
rule zero > zero => untaint
```

// Rules for **equal to** (==) Expressions.

```
rule untaint == untaint => untaint
rule untaint == taint => taint
rule untaint == zero => untaint
rule taint == untaint => taint
rule taint == taint => taint
rule taint == zero => taint
```

rule zero == taint => taint
rule zero == untaint => untaint
rule zero == zero => untaint

// Rules for **logical AND** (&&) Expressions.

rule untaint && untaint => untaint
rule untaint && taint => taint
rule untaint && zero => untaint
rule taint && untaint => taint
rule taint && taint => taint
rule taint && zero => taint
rule zero && taint => taint
rule zero && untaint => untaint
rule zero && zero => untaint

// Rules for **logical OR** (||) Expressions.

rule untaint || untaint => untaint
rule untaint || taint => taint
rule untaint || zero => untaint
rule taint || untaint => taint
rule taint || taint => taint
rule taint || zero => taint
rule zero || taint => taint
rule zero || untaint => untaint
rule zero || zero => untaint

// Rules for **Negation** (!) Expressions.

rule ! taint => taint
rule ! untaint => untaint
rule ! zero => zero

// Rules for **Union** Expressions.

rule taint union untaint => taint
rule taint union taint => taint
rule taint union zero => taint
rule untaint union untaint => untaint
rule untaint union taint => taint
rule untaint union zero => untaint
rule zero union untaint => untaint
rule zero union taint => taint

```
        rule zero union zero => untaint


// Rules for Function Handling

  syntax KItem ::=  (Map,K,ControlCellFragment)
          rule <k> lambda(Xs,S)(Vs:Vals) ~> K => mkDecls(Xs,Vs) S return;</k>
                   <control>
                            <fstack> .List => ListItem((Env,K,C)) ...</fstack>
                                      C
                   </control>
          <tenv> Env => GEnv </tenv>
          <gtenv> GEnv </gtenv>
          rule <k> {return (AExp);} => return (AExp); ...</k>
          rule <k> return(V:Val); ~> _ => V ~> K </k>
                   <control>
                            <fstack> ListItem((Env,K,C)) => .List ...</fstack>
                           (_ => C)
                   </control>
                   <tenv> _ => Env </tenv>
          rule <k> return; =>. ~> K </k>
                   <control>
                            <fstack> ListItem((Env,K,C)) => .List ...</fstack>
                             (_ => C)
                   </control>
                   <gtenv> .Map => Rho </gtenv><tenv> Rho => Env </tenv>


          // Rules for Assignment and other basic operations

          rule <k> read() => taint ...</k> <in> ListItem(I:Int) => .List ...</in>  [read]
          rule <k> X:Id => V ...</k> <tenv>... X |-> V ...</tenv>
          rule <k> X:Id => V ...</k> <context>... X |-> V ...</context>
          rule <k> X:Id => V ...</k> <alias>... X |-> V ...</alias>

          rule <k> X:Id = V:Val => V ...</k><tenv>... X |-> (_ => V) ...</tenv>
          rule <k> A:Id[ E:AExp ]=V:Val => V ...</k> <tenv>... A |-> (_ => V) ...</tenv>  when
          V:Val ==K taint [assignment]

          rule <k> *P:Id => *pointsTo(X) ...</k> <alias>... P |-> pointsTo(X) ...</alias>

          rule <k> *pointsTo(X) => V ...</k><tenv>... X |-> V ...</tenv>
          rule <k> X:Id = pointsTo(Y) => Y ...</k> <alias>... X |-> (_ => pointsTo(Y)) ...</alias>
          rule <k> *X:Id = V:Val => V ...</k> <tenv>... X |-> V ...</tenv>
          rule <k> P:Id = &X:Id; =>. ...</k> <ptr>... P |-> (_ => V) ...</ptr> <alias>
          A:Map=>A[P<-pointsTo(X)]</alias> <tenv>... X |-> V ...</tenv>
```

```
        rule <k> P:Id = &X:Id; =>. ...</k> <alias> A:Map=>A[P<-pointsTo(X)]</alias>


        rule {} => .  [structural]

        rule S1:Stmts S2:Stmts => S1 ~> S2  [structural]
        rule _:Val; => .
        rule <k>  Rho =>. ...</k>  <tenv> _ => Rho </tenv>    [structural]

// Rules for Pointer Aliases

        rule <k> X:Id = V:Val; =>  X = V:Val;  P = V:Val; ...</k>
                <ptr-alias>
                        <alias>... P |-> PointsTo(X) ...</alias>
                        <ptr> Mu </ptr>
                </ptr-alias>
                <tenv> Rho </tenv>  when (X in keys(Rho)) // <gtenv> Mu </gtenv>

        rule <k> P = V:Val => R = V:Val ...</k>
                <ptr-alias>
                        <alias>... R |-> PointsTo(P) ...</alias>
                        <ptr> ... P |-> (_ => V) ... </ptr>
                </ptr-alias>

        rule <k> P:Id = &X:Id; => P = V:Val; ...</k>
                <ptr-alias>
                        <alias>A:Map => A[P <- PointsTo(X)] </alias>
                        <ptr> Mu </ptr>
                </ptr-alias> <tenv>... X |-> V ...</tenv>

        rule <k> P:Id = &X:Id; => P = V; ...</k>
                <ptr-alias>
                        <alias>A:Map => A[P <- PointsTo(X)] </alias>
                        <ptr>... X |-> V ...</ptr>
                </ptr-alias>

        rule <k> P = Q; => P = V:Val ...</k>
                <ptr-alias>
                        <alias>... P |-> (_ => Q |-> PointsTo(?S)) ...</alias>
                        <ptr>... Q |-> V:Val ...</ptr>
                </ptr-alias>

        rule <k> P = &Q; => P = V; ...</k>
                <ptr-alias>
                        <alias> A:Map => A[P <- PointsTo(Q)] </alias>
                        <ptr>...  Q |-> V ...</ptr>
```

```
            </ptr-alias>


        rule <k> P = *Q; => P = ?T:Type; ...</k>
            <ptr-alias>
                <alias>... P |-> (_ => Q |-> PointsTo(?S |-> PointsTo(?M)))  ...</alias>
                <ptr> Mu </ptr>
            </ptr-alias>  when (P in keys(Mu))


 syntax Stmts ::= mkDecls(Params,Vals)  [function]

        rule  mkDecls(( DType  X:Id,  Ps:Params), (V:Val,  Vs:Vals)) => DType X=V;
              mkDecls(Ps,Vs)
        rule  mkDecls(( DType* X:Id,  Ps:Params), (V:Val,  Vs:Vals)) => DType* X=V;
              mkDecls(Ps,Vs)
        rule  mkDecls(( DType  &X:Id, Ps:Params), (V:Val,  Vs:Vals)) => DType X=V;
              mkDecls(Ps,Vs)
        rule  mkDecls(( DType  X[]:AExp, Ps:Params), (V:Val,  Vs:Vals)) => DType X[]=V;
              mkDecls(Ps,Vs)
        rule mkDecls(.Params,.Vals) => {}


// Rules for restore context after exiting construct body

        rule <k> restore(Mu) =>. ...</k><context> _ => Mu </context>
        rule <k> restoreenv(Rho) =>. ...</k> <tenv> _ => Rho </tenv>
        rule <k> break => store(CurRho) ...</k><tenv>CurRho</tenv>
        rule <k> store(CurRho) => . ...</k><tenv>CurRho</tenv><temp> .Map => CurRho
        </temp>


endmodule               //end of the LANG module.
```