5. <ha\_stmt> -> ha ( <boolexpr> ) <stmt> [ lol <stmt> ]

M\_ha(ha (<boolexpr>) <stmt>, s) 🡪

if M\_b(<boolexpr>, s) == error

return error;

if M\_b(<boolexpr>, s)

if M\_stmt(<stmt>, s) == error

return error;

return s = M\_stmt(<stmt>, s)

M\_ha(ha (<boolexpr>) <stmt1> lol <stmt2>, s) -->

if M\_b(<boolexpr>, s) == error

return error;

if M\_b(<boolexpr>, s)

if M\_stmt(<stmt1>, s) == error

return error;

return s = M\_stmt(<stmt1>, s)

else

if M\_stmt(<stmt2>, s) == error

return error;

return s = M\_stmt(<stmt2>, s)

6. <during\_stmt> -> during ( <boolexpr> ) <stmt>

M\_during(during (<boolexpr>) <stmt>, s) 🡪

If M\_bool(<boolexpr>, s) == error

return error;

if M\_bool(<boolexpr>, s)

if M\_stmt(<stmt>, s) == error

return error;

return s = M\_stmt(<stmt>, s)

7. <expr> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;`

M\_expr($$ <ident> { (\* | - | = | \*\*\*) <ident> } `;`, s) 🡪

If M\_ident(<ident>, s) == error

Return error;

If M\_ident(<ident>,s)

If M\_symbols((\* | - | = | \*\*\*), s) == error

Return error;

If M\_ident(<ident>,s)

If M\_symbols((\* | - | = | \*\*\*), s)

If M\_ident(<ident>,s) == error

Return error;

If M\_ident(<ident>,s)

If M\_symbols((\* | - | = | \*\*\*), s)

If M\_ident(<ident>,s)

If M\_ident(<ident>,s)

If M\_semi(; s) == error

Return error;

If M\_ident(<ident>,s)

If M\_symbols((\* | - | = | \*\*\*), s)

If M\_ident(<ident>,s)

If M\_ident(<ident>,s)

If M\_semi(; s)

Return s = M\_semi(;, s)

8. <expr> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;`

<ident> -> id | oat\_lit | ger\_lit | char\_lit | string\_lit | <bool\_value>

M\_expr($$ <ident> { (\* | - | = | \*\*\*) <ident> } `;`, s) 🡪

If M\_ident(<ident>, s) == error

Return error;

If M\_ident(<ident>,s)

If M\_symbols((\* | - | = | \*\*\*), s) == error

Return error;

If M\_ident(<ident>,s)

If M\_symbols((\* | - | = | \*\*\*), s)

If M\_ident(<ident>,s) == error

Return error;

If M\_ident(<ident>,s)

If M\_symbols((\* | - | = | \*\*\*), s)

If M\_ident(<ident>,s)

If M\_ident(<ident>,s)

If M\_semi(; s) == error

Return error;

If M\_ident(<ident1>,s)

If M\_symbols((\* | - | = | \*\*\*), s)

If M\_ident(<ident2>,s)

If M\_ident(<ident>,s)

**Return s = M\_ident(<bool\_value>, s)**

If M\_semi(; s)

Return s = M\_semi(;, s)

9. id = identifier, oat\_lit = float\_lit, ger\_lit = int\_lit | string\_lit = string\_lit |

<bool\_value> = true | false

\*\* optimal for scenario

1. **String + String does concatenation**

Syntax: <eq1> -> <assignment> -> `$`<data\_type> <ident>`;`

<data\_type> -> ger | oat | string | car | bool

<eq1> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;` \*\*\* = exponentiation

<ident> -> id | oat\_lit | ger\_lit | char\_lit | string\_lit | <bool\_value>

Semantics: <assignment.value> 🡨 data\_type.value ident.value1

If (data\_type.value == string) return string

Else if (data\_type.value == ger) return ger

Else if (data\_type.value == oat) return oat

Else if (data\_type.value == car) return car

Else return bool

<eq1.value > 🡨 ident.value2 + ident.value3

If (ident.value1 == identvalue2 && ident.value3 = string\_lit) return string\_lit

Else return float

Predicate: ident.value1 == identvalue2

Ident.value3 == string\_lit to add two strings together

1. **String \* Natural repeats the Natural**

Syntax: <assignment> -> `$`<data\_type> <ident>`;`

<eq1> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;` \*\*\* = exponentiation

<ident> -> id | oat\_lit | ger\_lit | char\_lit | string\_lit | <bool\_value>

<data\_type> -> ger | oat | string | car | bool

Semantic: assignment.value 🡨 data\_type.value ident.value1

If (data \_type.value == string) return string \*\*

Else if (data\_type.value == oat) return oat

Else if (data \_type.value == ger) return ger

Else if (data\_type.value == car) return car

Else return bool

Eq1.value 🡨 ident.value2 = ident.value3

If (ident.value2 == ident.value1 && ident.value3 == string\_lit) return string\_lit \*\*

Else If (ident.value2 == ident.value1 && ident.value3 == ger\_lit) return error

Else If (ident.value2 == ident.value1 && ident.value3 == oat\_lit) return error

Else If (ident.value2 == ident.value1 && ident.value3 == car\_lit) return error

Else If (ident.value2 == ident.value1 && ident.value3 == bool\_value) return error

Else return error

Eq1.value2 🡨 ident.value2 \*\*\* ident.value4

If (ident.value4 == ger\_lit) return string\_lit (ger\_lit times)

Else return error

Predicate: ident.value4 == ger\_lit

Ident.value1 == id

Ident.value1 == ident.value2

1. Assign bool to natural is allowed

Syntax: <assignment> -> `$`<data\_type> <ident>`;`

<eq1> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;` \*\*\* = exponentiation

<data\_type> -> ger | oat | string | car | bool

<ident> -> id | oat\_lit | ger\_lit | char\_lit | string\_lit | <bool\_value>

Semantics:

Assignment.value 🡨 data\_type.value ident.value1

If (data\_type.value == ger) return ger \*\*

else if (data\_type.value == oat) return oat

else if (data\_type.value == car) return car

else if (data\_type.value == string) return string

else return bool

eq1.value 🡨 ident.value2 = ident.value3

if (ident.value2 == ident.value1 && ident.value3 == bool\_value) return bool\_value \*\*

else if (ident.value2 == ident.value1 && ident.value3 == ger\_lit) return ger\_lit

else if (ident.value2 == ident.value1 && ident.value3 == oat\_lit) return error

else if (ident.value2 == ident.value1 && ident.value3 == string) return error

else if (ident.value2 == ident.value1 && ident.value3 == car\_lit) return car\_lit

Predicate: ident.value1 == id

Ident.value1 == ident.value2

Ident.value3 == bool\_value

Data\_type.value == ger

1. Assign natural to bool is allowed

Syntax: <assignment> -> `$`<data\_type> <ident>`;`

<eq1> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;` \*\*\* = exponentiation

<data\_type> -> ger | oat | string | car | bool

<ident> -> id | oat\_lit | ger\_lit | car\_lit | string\_lit | <bool\_value>

Assignment.value 🡨 data\_type.value ident.value1

If (data\_type.value == bool) return bool \*\*

Else if (data\_type.value == ger) return ger

Else if (data\_type.value == oat) return oat

Else if (data\_type.value == car) return car

Else if (data\_type.value == string) return string

Else return error

Eq1.value 🡨 ident.value2 = ident.value3

If (ident.value2 == ident.value1 && ident.value3 == ger\_lit) return ger\_lit \*\*

else if (ident.value2 == ident.value1 && ident.value3 == oat\_lit) return error

else if (ident.value2 == ident.value1 && ident.value3 == string\_lit) return error

else if (ident.value2 == ident.value1 && ident.value3 == car\_lit) return error

else return bool

Predicate: ident.value1 == id

Ident.value1 == ident.value2

Data\_type.value == bool

Ident.value3 == get\_lit

1. Assign char to natural is allowed

Syntax: <assignment> -> `$`<data\_type> <ident>`;`

<eq1> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;` \*\*\* = exponentiation

<data\_type> -> ger | oat | string | car | bool

<ident> -> id | oat\_lit | ger\_lit | car\_lit | string\_lit | <bool\_value>

Assignment.value 🡨 data\_type.value ident.value1

If (data\_type.value == ger) return ger \*\*

Else if (data\_type.value == oat) return oat

Else if (data\_type.value == string) return string

Else if (data\_type.value == car) return car

Else return bool

Eq1.value 🡨 ident.value2 = ident.value3

If (ident.value1 == ident.value2 && ident.value3 == car\_lit) return car\_lit \*\*

Else if (ident.value1 == ident.value2 && ident.value3 == ger\_lit) return ger\_lit

Else if (ident.value1 == ident.value2 && ident.value3 == oat\_lit) return ger\_lit

Else if (ident.value1 == ident.value2 && ident.value3 == string\_lit) return string\_lit

Else error

Predicate: ident.value1 == id

Ident.value1 == ident.value2

Ident.value3 == car\_lit

Data\_type.value == ger

1. Assign natural to char is allowed

Syntax: <assignment> -> `$`<data\_type> <ident>`;`

<eq1> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;` \*\*\* = exponentiation

<data\_type> -> ger | oat | string | car | bool

<ident> -> id | oat\_lit | ger\_lit | car\_lit | string\_lit | <bool\_value>

Semantic:

Assignment.value 🡨 data\_type.value ident.value1

If (data\_type.value == car) return car \*\*

Else if (data\_type.value == ger) return ger

Else if (data\_type.value == oat) return oat

Else if (data\_type.value == string) return string

Else error

Eq1.value 🡨 ident.value2 = ident.value3

If (ident.value2 == ident.value1 && ident.value3 == ger\_lit) return ger\_lit \*\*

Else if (ident.value2 == ident.value1 && ident.value3 == oat\_lit) return error

Else if (ident.value2 == ident.value1 && ident.value3 == string) return error

Else if (ident.value2 == ident.value1 && ident.value3 == bool) return error

Else return car\_lit

Predicate: data\_type.value == car

Ident.value1 == id

Ident.value1 == ident.value2

Ident.value3 == ger\_lit

1. Assign natural to real is allowed

Syntax: <assignment> -> `$`<data\_type> <ident>`;`

<eq1> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;` \*\*\* = exponentiation

<data\_type> -> ger | oat | string | car | bool

<ident> -> id | oat\_lit | ger\_lit | car\_lit | string\_lit | <bool\_value>

Semantic: assignment.value 🡨 data\_type.value ident.value1

If (data\_type.value == oat) return oat \*\*

Else if (data\_type.value == ger) return error

Else if (data\_type.value == string) return string

Else if (data\_type.value == car) return car

else if (data\_type.value == bool) return bool

else return erro

eq1.value 🡨 ident.value2 = ident.value3

if (ident.value1 == ident.value2 && ident.value3 == ger\_lit) return ger\_lit

else if (ident.value1 == ident.value2 && ident.value3 == oat\_lit) return oat\_lit

else if (ident.value1 == ident.value2 && ident.value3 == string\_lit) return error

else if (ident.value1 == ident.value2 && ident.value3 == car\_lit) return error

else if (ident.value1 == ident.value2 && ident.value3 == string\_lit) return error

else error

Predicate: ident.value1 == id

Ident.value1 == ident.value2

Data\_type.value == oat

Ident.value3 == ger\_lit

1. No other types are allowed to be assigned to others outside of their own

Syntax: <assignment> -> `$`<data\_type> <ident>`;`

<eq1> -> `$$` <ident> { (\* | - | = | \*\*\*) <ident> } `;` \*\*\* = exponentiation

<data\_type> -> ger | oat | string | car | bool

<ident> -> id | oat\_lit | ger\_lit | car\_lit | string\_lit | <bool\_value>

Assignment.value 🡨 data\_type.value ident.value1

Eq1.value 🡨 ident.value2 = ident.value3

If (data\_type.value == string) return string

Else if (data\_type.value == oat) return oat

Else if (data\_type.value == ger) return ger

Else if (data\_type.value == string) return string

Else if (data\_type.value == bool) return bool

Else error

Eq1.value 🡨 ident.value2 = ident.value3

If (data\_type.value == string)

If (ident.value1 == ident.value2 && ident.value3 == string\_lit) return string\_lit

Else error

Else if (data\_type.value == oat)

If (ident.value1 == ident.value2 && ident.value3 == oat\_lit) return oat\_lit

(g.) Else if (ident.value1 == ident.value2 && ident.value3 == int\_lit) return int\_lit

Else error

Else if (data\_type.value == ger)

If (ident.value1 == ident.value2 && ident.value3 == ger\_lit) return ger\_lit

(c.) Else if (ident.value1 == ident.value2 && ident.value3 == bool\_value)

return bool\_value

(e.) else if (ident.value1 == ident.value2 && ident.value3 == car\_lit) return car\_lit

else error

Else if (data\_type.value == car)

If (ident.value1 == ident.value2 && ident.value3 == car\_lit) return car\_lit

(f.) Else if (ident.value1 == ident.value2 && ident.value3 == ger\_lit) return ger\_lit

Else error

Else if (data\_type.value == bool)

If (ident.value1 == ident.value2 && ident.value3 == bool\_value) return bool\_value

(d.) Else if (ident.value1 == ident.value2 && ident.value3 == ger\_lit) return ger\_lit

Else error

Else error

Predicate: ident.value1 == id

Ident.value1 == ident.value2

Only possible assignments (ger 🡨 bool, oat 🡨 ger, car 🡨 ger, bool 🡨 ger, ger 🡨 car, ger 🡨 ger, oat 🡨 oat, ger 🡨 oat, car 🡨 car, string 🡨 string, bool 🡨 bool)

1. **Dividing by zero is an error**

Syntax: <eq2> -> `$$$` <ident> { (+ | / | %) <ident> } `;`

Semantic:

<eq2.value> 🡨 ident.value1 / ident.value2

<eq2.value> 🡨 if (ident.value1 == ger && ident.value2 == ger) return ger

Else if (ident.value1 == oat && ident.value2 == oat) return oat

Else return oat

Predicate:

Ident.value2 != 0

1. **Modulo operating by zero is an error**

Syntax: <eq2> -> `$$$` <ident> { (+ | / | %) <ident> } `;`

Semantic:

<eq2.value> 🡨 ident.value1 % ident.value2

<eq2.value> 🡨 if (ident.value1 == ger && ident.value2 == ger) return ger

Else if (ident.value1 == oat && ident.value2 == oat) return oat

Else return oat

Predicate:

Ident.value2 != 0

10.

Please refer to the number10 file in github.

11. Axiomatic Semantics (find the weakest preconditions)

1. a = 2 \* (b – 1) – 1 {a > 0}

*a = 2 \* (b – 1) – 1*

*{a > 0}*

*2 \* (b – 1) – 1 > 0*

*2 \* (b – 1) > 1*

*(b – 1) > ½*

*b > ½ + 2/2*

*b > 3/2*

1. if (x < y) x = x + 1

x = x + 1 {x < 0}

else x = 3 \* x

x = 3 \* x {x < 0}

{x < 0}

{x + 1 = 0}

x = x + 1

{x < 0}

{3 \* x < 0}

x = 3 \* x

{x < 0}

{x < -1} {x < -1}

x = x + 1 when it satisfies this

{x < 0} condition it satisfies both

{x < 0} conditions

x = 3 \* x

{x < 0}

1. y = a \* 2 \* (b – 1) – 1 x = y + 1 {y < -1}

if (x < y) {x < 0} x = y + 1

x = y + 1 x = 3 \* x {x < 0}

else {x < 0} x = 3 \* x

x = 3 \* x {x < 0}

{x < 0}

Y = a \* 2 \* (b -1) – 1

{y < -1}

X = y + 1

{x < 0}

X = 3 \* x

{x < 0}

1. a = 3 \* (2 \* b + a); a = 3 \* (2 \* b + a); a = 3 \* (2 \* b + a);

b = 2 \* a – 1 {2 \* a – 1 > 5} {a > 3}

{b > 5} b = 2 \* a – 1; b = 2 \* a – 1;

{b > 5} {b > 5}

{b + a > ½}

A = 3 \* (2 \* b + a);

{a > 3}

B = 2 \* a – 1;

{b > 5}