The CNS format

Documentation for version 1.0 (2009)

Updated 01 September 2009

Contents

- Introduction
  - Some terminology
- Player Variables
- States
  - Introduction to States
    - Life and power
    - Control
    - Game-time and state-time
    - Position, velocity and acceleration
    - Juggling
  - Basic Parts of a State
  - Details on StateDef
    - `type`
    - `movetype`
    - `physics`
    - `anim`
    - `velset`
    - `ctrl`
    - `poweradd`
    - `juggle`
    - `facep2`
    - `hitdefpersist`
    - `movehitpersist`
    - `hitcountpersist`
    - `sprpriority`
  - Details on State Controllers
    - Controller Format
    - Triggers
      - Trigger Logic
      - Trigger Persistency
      - Trigger redirection
    - Commonly-used controllers
- Common states (common1.cns)
- Special State Numbers
- Expressions
  - Data types
  - Arithmetic Operators
  - Precedence and associativity of operators
  - Expression syntax
  - Condition- and function-type triggers
  - Trigger redirection
  - bottom
    - Special Forms
  - Avoiding warnings
  - Expressions in trigger arguments
  - Expressions in state and state controller parameters
  - Organizing for efficiency

Introduction

The CNS file of a player serves two purposes:

i. It defines the variables of that player, such as walking speed, drawing scale factor, and so on.
ii. It contains the states of the player, which describe all the moves that the player can do. States are the building blocks that you can use to create simple as well as complicated moves.

Like many other character files, the CNS is a text file that you can edit with any text editor.

In the CNS file, a semicolon (;) is considered a "comment" character. Any text on the same line after the semicolon will be ignored by the program. The CNS is mostly case-insensitive, i.e. "MyName" is treated the same as "myname" and "mYnAMe". The one exception is the `command` trigger (covered in detail in the CMD documentation).

Some terminology

When we say "group", we mean any block of lines of text beginning with something that looks like `[groupname]`, and ending before the next group. For example, the group `Foo` consists of the first three lines in the following:

```
[Foo]
line1
line2

[Group 2]
more lines
```

Within a group, the parameters can appear in any order. So,

```
[SomeGroup]
value1 = 1234
value2 = "A string"
```

is equivalent to:

```
[SomeGroup]
value2 = "A string"
value1 = 1234
```

Player Variables

No full documentation yet. `See chars/kfm/kfm.cns` for comments on each variable.

Some important ones to note:

- In `[Size]`, you can use `xscale` and `yscale` to change the width and height of your character. This saves the trouble of scaling every single one of the sprites.
- Set up the speeds of the player in `[Velocity]`
- Set the player's downward acceleration -- `yaccel` in `[Movement]`

States

Animations and sounds are the most immediately noticeable manifestations of a MUGEN character, but the true "heart" of the character is its CNS file(s). The CNS file, also known as the character's "states" file,

contains the code that actually gives the character its functionality -- allowing it to perform moves when commanded, to hit its opponent, or to obey the law of gravity.

The structure of a CNS file is simple. Code is organized into logical groups called `[StateDef]`s. `[StateDef]` blocks are numbered, and a typical character may contain hundreds of them. Within each `[StateDef]` is a number of `[State]` blocks, also known as state controllers. Each state controller is a trigger-action pair: it specifies some action for the character to take, and the conditions under which that action should be performed.

At each instant ("tick") of game time, the character executes the code in some `[StateDef]` block. For instance, if the character is standing idly, then it is typically executing in the `[StateDef 0]` block. For brevity, we say that the character is "in state 0". At each tick that the character is in state 0, it will evaluate the state controllers in the `[StateDef 0]` block from top to bottom, checking the conditions and taking any actions that are specified. If the character needs to change to some other `[StateDef]` block, let's say to state 417 because it is performing a move, it can execute a `ChangeState` state controller to make the switch. The character will then start executing the code in `[StateDef 417]`, in exactly the same fashion.

Each MUGEN character additionally has three special states, numbered -1, -2, and -3. These are the only allowable states with negative numbers. State -1 generally contains state controllers that determine state transition rules based on user input (commands). State -2 contains other state controllers that need to be checked every tick. State -3 contains state controllers which are checked every tick unless the player is temporarily using another player's state data (for instance, when the player is being thrown).

To put it all together: for each tick of game-time, MUGEN makes a single pass through each of the special states, from top to bottom, in order of increasing state number (-3, -2, then -1). For each state controller encountered, its condition-type triggers are evaluated and, if they are satisfied, the controller is executed. Then processing proceeds to the next state controller in the state. A state transition (`ChangeState`) in any of the special states will update the player's current state number, and will abort processing the remainder of the state. After all the state controllers in the special states have been checked, the player's current state is processed, again from top to bottom. If a state transition is made out of the current state, the rest of the state controllers (if any) in the current state are skipped, and processing continues from the beginning of the new state. When the end of the current state is reached and no state transition is made, processing halts for this tick.

There is one exception to the above scenario. If the character is a "helper" character, i.e., spawned by the `Helper` state controller, that character will not have the special states -3 and -2. The helper character will not have the special state -1 either, unless it has keyboard input enabled. (This is controlled by the `Helper` state controller when the character is created.)

Introduction to States

Programming states is the hardest part of creating a character. It entails a lot of work, testing, and sometimes trial-and-error. In this section, we'll often refer the player being programmed, and to his opponent as well. Let us call the player whose states we are editing P1, and his opponent P2.

Do not be discouraged if you do not understand a lot of this document on your first reading. The best way to learn about states is to first play around with values in the CNS of a completed character, and see what effects they have on him or her. There's nothing to fear from "tweaking" the CNS; M.U.G.E.N is designed to detect syntax errors and report them.

Included with the M.U.G.E.N distribution package is a character named Kung Fu Man (KFM for short.) You can find him in the directory chars/kfm.

The CMD file contains declarations of command names and the definition of State -1, a special state which is used to control how the character responds to user input. See the CMD document for more information.

Below are some concepts that will be useful for you to know.

Life and power

A player's life bar is the yellow bar at the top of the screen on his side of the screen. When the life bar reaches zero, the player is knocked out. His power bar is the blue bar, and that increases with each attack he gives or takes. When the power bar reaches certain values, he can do super moves.

Control

When we say a player "has control", we mean that he is ready to walk or jump or attack. A player who does not have control will not respond to your input (from keyboard or joystick). For example, when P1 is in his stand state, he has control, and will walk forward if you press the forward button. A player will typically not have control when he is in an attack state, otherwise you could just walk away halfway through a punch.

There is an exception to the rule, however. Sometimes you can let the player respond to certain moves even if he has no control. That is called a "move interrupt", or a "move cancel". See the CMD documentation for details.

We will frequently refer to a player's "control flag". A "flag" is a value that is either true, or false. If we say the player's control flag is true, then it means he has control.

Game-time and state-time

M.U.G.E.N keeps track of the time that has passed in the game. Every time the game is updated (this includes updating the players, checking collisions, and drawing to the screen), we say game-time has increased by one. The time a player has spent in a state is known as the "state-time". The state-time starts at 0 at the beginning of a state, and increases by one tick for every tick of game-time.

Position, velocity and acceleration

Those of you with basic knowledge of math should understand these concepts. M.U.G.E.N uses the following coordinate system. The greater the x-position, the farther right the player is. The less the x-position, the closer he is to the left. A y-position of zero is at ground level. As the player's y-position gets larger he moves downwards. For example, a negative y-position means he is in the air. Similarly, when we say a player has positive x-velocity, it means he is moving forward, and if he as negative x-velocity, he is moving backwards. A player with positive y-velocity is moving downward, and a negative y-velocity means he is moving up. A positive x-acceleration means the player's x-velocity is increasing, a negative x-acceleration means his x-velocity is decreasing. Likewise for y-acceleration.

Juggling

M.U.G.E.N allows for certain moves to "juggle", that is, to hit opponents who have been knocked into the air, or are lying down on the ground. The juggling system works this way: each person starts with a set number of juggle "points" on the first hit that makes them fall, typically 15.

Some quick terminology: when we say a player is "falling", then we mean he does not recover control in the air, and will fall onto the ground.

If a player is hit while he is in the falling in the air or lying down on the ground, then his juggle points will decrease by an amount depending on the attack. When an attack requires more juggle points than the opponent has left, then the attack will miss. Any move that causes the opponent to fall immediately subtracts its juggle points on the first hit.

For example, an attack that requires 7 juggle points could theoretically be used to juggle the opponent twice (assuming you started with 15 points), leaving the opponent with 1 point left. Subsequent such attacks will miss.

The reason for this juggle system is to prevent infinite combos in the air.

Basic Parts of a State

Note: This section assumes you have at least browsed the documentation of AIR files, and understand the concepts of animation, as know the meaning of key words and phrases such as action and element of an action.

Here is a short example state for P1:

```
[Statedef 200]
type = S
physics = S
movetype = I
ctrl = 0
anim = 200
velset = 0

[State 200, 1]
type = ChangeState
trigger1 = AnimTime = 0
value = 0
ctrl = 1
```

This state plays back the Action 200 of P1's animation, and returns P1 to his standing state after the animation has ended. In this case, assume Action 200 has a finite looptime. That is, Action 200 does not have any elements with time equal to -1.

At this point, you do not need to worry about the details. Let us begin by knowing what a state consists of.

All states must have a single Statedef section and one or more State sections.

Statedef contains the starting information of a state, such as what kind of state it is (standing, crouching, in the air) and what kind of move he is doing (attacking, idling.)

Each State section is referred to as a state controller, or a controller for short. Controllers tell the program what to do to P1, and when to do it. There are many kinds of controllers, each with its own function. For example, there are controllers to change the players position or velocity, define the effects of attacks, create projectiles, switch between animation Actions, change states, and so on. Each controller must have at least one trigger. A trigger is an event that causes the controller to be activated. Examples are: trigger at the start of the state, trigger at the end of the animation (as seen in the example State above), trigger on an element of an animation Action, trigger when P2 is within a certain range of P1, and so on.

Details on StateDef

Every state must begin with exactly one StateDef group, also known as a Statedef section. A StateDef group must look like this (put in one or more parameters where the dots are):

```
[Statedef state_number]
. state_parameters
.
.
```

Replace `state_number` with the number of the state you are programming. With the exception of the special group numbers (see Appendix A) you are allowed to use any state number you choose. To avoid choosing a special group number, do not choose numbers from 0-199, and from 5000-5999.

The lines that follow should include the following basic parameters:

- type
- movetype
- physics
- anim

There are also additional optional parameters that may be included:

- velset

- ctrl
- poweradd
- juggle
- facep2
- hitdefpersist
- movehitpersist
- hitcountpersist
- sprpriority

type

This is the state type of P1 in that state. It defines if he is standing, crouching, in the air, or lying down. The corresponding values are "S", "C" , "A" and "L" respectively (without the quotation marks). To leave the type unchanged from the previous state, use a value of "U". If this line is omitted, it assumes the type is "S". You will most commonly use "S", "C" and "A". For example, a crouching state type would require the line:

```
type = C
```

The type is used to determine several factors, most importantly, how P1 will react to being hit. For example, being in a "stand"-type state, P1 will react as if he is standing on the ground. If the type was "air", then P1 would react to the hit accordingly.

movetype

This is the type of move P1 is doing: "A" for attack, "I" for idle and "H" for being hit. To leave the movetype unchanged from the previous state, use a value of "U". The value is assumed to be "I" if this line is omitted. "A" and "H" should be self-explanatory. "I" is used for states where P1 is neither attacking, nor being hit. For example, an attack state should have the line:

```
movetype = A
```

You need to specify the movetype so the program will know how to process the state. Incorrectly specifying the movetype may cause P1 to act incorrectly.

physics

You need to specify what physics to use in that state. Valid values are "S" for stand, "C" for crouch, "A" for air, and "N" for none. To leave the physics unchanged from the previous state, use a value of "U". If omitted, the value of "N" is assumed. The kind of physics is used to determine how P1 behaves.

- For "S" physics, P1 will experience friction with the ground. The value for the friction coefficient is set in the Player Variables.
- For "C" physics, P1 will experience friction, just like in the "S" state.
- For "A" physics, P1 will accelerate downwards, and if his y-position is greater than 0 (ie. he touches the ground) he will immediately go into his landing state.
- If you use "N" P1 will not use any of these pre-programmed physics.

Do not confuse "physics" with the state "type". They are usually the same, but you are given the choice if you want more control. For instance, you may choose to use "N" (no physics), and specify your own acceleration and landing detection for an aerial state.

anim

This parameter changes the Animation Action of P1. Specify the action number as the value. If you do not want P1 to change animation at the start of the state, omit this parameter.

So to have a state with number 400, where the player is doing a crouching attack with Action 400, the typical parameters would be:

```
[Statedef 400]
type = c
movetype = a
physics = c
anim = 400
```

velset

You can use `velset` to set P1's velocity at the beginning of the state. The format is a number pair, representing the x velocity and the y velocity respectively. Omitting this line will leave P1's velocity unchanged. For example,

```
velset = 4,-8
```

makes P1 start moving diagonally up and forwards.

There is an exception to this. Even if you have `velset = 0`, attacking P2 in the corner will push P1 away.

ctrl

This parameter will set P1's control. A value of "0" sets the flag to false, "1" sets it to true. If omitted, P1's control flag is left unchanged. For example, to give P1 control, use

```
ctrl = 1
```

poweradd

When included, the `poweradd` parameter adds to the player's power bar. The value is a number, and can be positive or negative. This parameter is typically used in attack moves, where you want the player to gain power just by performing the attack. For example, to add 40 power, type

```
poweradd = 40
```

juggle

The `juggle` parameter is useful only for attacks. It specifies how many points of juggling the move requires. If omitted for an attack, that attack will juggle if the previous attacking state successfully juggled. You should include the `juggle` parameter for all attacks. If an attack spans more than one state, include the `juggle` parameter only in the first state of that attack. See also Juggling.

facep2

When you include the line `facep2 = 1`, the player will be turned, if necessary, to face the opponent at the beginning of the state. `facep2` has the default value of "0" if omitted.

hitdefpersist

If set to 1, any HitDefs which are active at the time of a state transition to this state will remain active. If set to 0, the default, any such HitDefs will be disabled when the state transition is made.

movehitpersist

If set to 1, the move hit information from the previous state (whether the attack hit or missed, guarded, etc; see "Move*" triggers in trigger docs) will be carried over into this state. If set to 0 (the default), this information will be reset upon entry into this state.

hitcountpersist

If set to 1, the hit counter (how many hits this attack has done) will be carried over from the previous state to this state. If set to 0 (the default), the hit counter will be reset upon state transition. This parameter does not affect the combo counter which is displayed on the screen. See the `HitCount` and `UniqHitCount` trigger documentation for how to check the hit counter.

sprpriority

If this parameter is present, the player's sprite layering priority will be set to the value specified. If omitted, the sprite priority will be left unchanged. `common1.cns` (the CNS file that is inherited by every player) defines the sprite priority of standing or crouching players to be 0, and jumping players to be 1. For most attack states, you will want to set `sprpriority = 2`, so that the attacker appears in front.

See `SprPriority` in the sctrls documentation for how to change sprite priority using a controller.

Details on State Controllers

Controller Format

All states must have at least one state controller, otherwise it will cause an error. State controller groups have the following format:

```
[State state_number, some_number]
type = controller_type
trigger1 = condition_exp
<universal optional parameters>
<additional parameters depending on controller>
```

The state_number must be the same number of the state from the `StateDef`. *some_number* can be any number you choose; it is the number that is reported when an error is found, so you know which controller needs to be fixed.

The universal (applicable to all state controllers) optional parameters are the `ignorehitpause` and `persistency` parameters. If `ignorehitpause` is set to 1, MUGEN will check this state controller even if the character is paused by a hit. Otherwise, this state controller will not be checked during a hit pause. The default is 0, which is recommended for all but exceptional situations. For an explanation of the `persistency` parameter, see Trigger Persistency.

controller_type is the name of the controller you are using. Each type of controller has a different effect, and requires different parameters. See sctrls.txt for a full list of state controllers.

The order of the controllers is significant. Controllers listed first are the ones checked and, if necessary, activated first.

Here is an example of a controller that gives P1 control at the start of the state (the same effect as putting `ctrl = 1` as a parameter in the `StateDef`):

```
[State 300, 1] ;State 300. 1 is just an arbitrary number.
type = CtrlSet ;Changes the control flag.
trigger1 = Time = 0
value = 1
```

In this example, the `CtrlSet` type lets you change the control flag of P1. The line that reads `trigger1 = Time = 0` means that this controller is activated when the state-time is 0, i.e., at the start of that state. The line `value = 1` says that we want to set the value of the control flag to 1, which means true. If we want to make P1 start the state with no control, then we just need to change the last line to `value = 0`.

Let's look another example. This controller moves P1 forwards by 10 pixels twice: on the second and third element of his current Animation Action. Don't worry if you don't know what parameters go with which controller types. You can learn more about them from the state controller documentation (sctrls).

```
[State 300, 2]
type = PosAdd ;Adds to P1's position
trigger1 = AnimElem = 2 ;Trigger on 2nd element.
trigger2 = AnimElem = 3 ;Trigger on 3rd element.
x = 10
```

As you see above, each controller must have at least one trigger. A trigger is a condition that causes the controller to be activated. This example has two triggers, and the controller is activated when *either one* is true.

Triggers

Trigger Logic

The first trigger should always be `trigger1`, and subsequent triggers should be `trigger2`, then `trigger3` and so on. The logic for deciding if a controller should be activated is:

1. Are all conditions of `trigger1` true? If so, then yes, activate the controller.
2. Otherwise, repeat the test for `trigger2`, and so on, until no more triggers are found.

This can be thought of as "OR" logic.

Be careful; skipping numbers will cause some triggers to be ignored. For example, if you have triggers `trigger1`, `trigger2` and `trigger4` without a `trigger3`, then `trigger4` will be ignored.

Now what if you want more than one condition to be met before the controller is activated? Here is an commonly-used example for testing if a player in the air has reached the ground. The triggers used are:

```
trigger1 = Vel Y > 0 ; True if Y-velocity is > 0 (going down)
trigger1 = Pos Y > 0 ; True if Y-position is > 0 (below ground)
```

At this point, you may be confused by the format of the trigger. Do not worry about it for now. We will get to it soon.

As you can see above, both the triggers have the same number. When several triggers have the same number, it implements "AND" logic. That is, the controller is activated if every one of the triggers with the same number is true, but not if one or more of them is false.

You can combine both ideas. For example:

```
trigger1 = Vel Y > 0 ; True if Y-velocity is > 0 (going down)
trigger1 = Pos Y > 0 ; True if Y-position is > 0 (below ground)
trigger2 = Time = 5  ; True if state-time is 5
```

The controller for this would be activated if the player landed on the ground (y-velocity and y-Position are both > 0), OR if his state time was 5.

Here is a summary:

- Triggers with the same number activate the controller only if all of them are true.
- Triggers with different numbers activate the controller if any one or more of them are true.

The format of a trigger is:

`trigger?` = *condition_exp*

`condition_exp` is an arithmetic expression to be checked for equality to 0. If condition_exp is 0, then the trigger is false. If *condition_exp* is nonzero, then the trigger is true. The *condition_exp* is usually a simple relational expression as in the examples above, but can be as simple or as complicated as required.

It is possible to use logical operators between expressions. For instance, this is equivalent to the previous example above.

```
trigger1 = ((Vel Y > 0) && (Pos Y > 0)) || Time = 5
```

See Expressions for a detailed explanation of arithmetic expressions.

A useful shortcut you might use is `triggerall`. It specifies a condition that must be true for all triggers. For instance, consider:

```
triggerall = Vel X = 0
trigger1 = Pos Y > -2
trigger2 = AnimElem = 3
trigger3 = Time = [2,9]
```

For any of `trigger1` to `trigger3` to be checked, the `triggerall` condition must be true too. In this case, as long as the x-velocity is not 0, then the state controller will not be activated. You can have more than one `triggerall` condition if you need. Note that at least one trigger1 must be present, even if you specify `triggerall`.

Trigger Persistency

In the case where you do not want the trigger to activate every single time the condition is true, you will need to add a `persistent` parameter. Let us begin with an example:

```
[State 310, 1]
type = PosAdd
trigger1 = Vel Y > 1
x = 10
```

This state controller moves P1 forwards by 10 pixels for every tick of game time where P1's y-velocity is greater than 1. That is, the controller is being activated everytime the trigger condition is true. If we want the controller to be activated only once, we will need to add a line:

```
[State 310, 1]
type = PosAdd
trigger1 = Vel Y > 1
persistent = 0       ;<-- Added this line
x = 10
```

`persistent` has a default value of 1, meaning that the controller is activated everytime the trigger is true. Setting `persistent` to 0 allows the controller to be activated only once during that state. This holds true until P1 leaves that state. If P1 returns to that state later, the controller can be activated once again.

The `persistent` parameter can also take values other than 0 and 1:

```
[State 310, 1]
type = PosAdd
trigger1 = Vel Y > 1
persistent = 2       ;<-- Modified this line
x = 10
```

In this case, setting `persistent` to 2 means the controller will be activated once of every two times the trigger is true. Setting `persistent` to 3 activates the controller every 3rd time, and so on.

Trigger redirection

One might wish to check the statetime of the player's target, or the player's parent (if the player is a helper), etc. This is possible using so-called trigger redirection. See Expressions for details.

Commonly-used controllers

The `null` controller will be useful for debugging. A `null` controller basically does nothing. You can use it to temporarily turn off certain controllers, instead of commenting out the entire section. For example, you might want to disable this:

```
[State 300, 1] ;Controller that accelerates P1 forwards
type = VelAdd
trigger1 = Time >= 0
x = .8
```

Simply comment out the type and put in `null`:

```
[State 300, 1] ;Controller that accelerates P1 forwards
type = null ;VelAdd
trigger1 = Time >= 0
x = .8
```

Later, when you want to reenable the controller, just change the type back to what it used to be.

Now let us look back at the example:

```
[Statedef 200]
type = S
physics = S
movetype = I
ctrl = 0
anim = 200
velset = 0

[State 200, 1]
type = ChangeState
trigger1 = AnimTime = 0
value = 0
ctrl = 1
```

`[State 200, 1]` is a `ChangeState` controller. As the name implies, it changes P1's state number (i.e., the `[StateDef]` block from which P1 executes). The `value` parameter should have the number of the state to change to. The optional `ctrl` parameter can be set P1's control flag as he changes states.

Now let's make this an attack state. First of all, the animation action needs attack collision boxes. A quick review from the AIR documentation: `Clsn1` is for attack and `Clsn2` is where the player can be hit. So P1 will hit P2 if any one of P1's `Clsn1` boxes intersects with any of P2's `Clsn2` boxes.

As an example, let's assume the animation action in P1's AIR file looks like this:

```
[Begin Action 200]
200,0, 0,0, 3
200,1, 0,0, 4
200,2, 0,0, 4
200,3, 0,0, 3
```

After defining the bounding boxes, it looks like:

```
[Begin Action 200]
Clsn2: 1
  Clsn2[0] = -10,0, 10,-80
200,0, 0,0, 3
Clsn1: 1
  Clsn1[0] =  10,-70, 40,-60
Clsn2: 2
  Clsn2[0] = -10,  0, 10,-80
  Clsn2[1] =  10,-70, 40,-60
200,1, 0,0, 4
Clsn2Default: 1 ;Use this box for the last two frames
  Clsn2[0] = -10,0, 10,-80
200,2, 0,0, 4
200,3, 0,0, 3
```

As you can see, each element has a Clsn2 box defined for it (the last two elements are using the same boxes). The second element is the only one with a Clsn1 box.

Note: It is all right to define Clsn1 boxes for any elements in an Animation Action, but if you put a Clsn1 box in the very first element, the attack will be instantaneous, and become unblockable. Therefore, it is recommended that you define Clsn1 boxes only for elements after the first one.

Now we are ready to set up the state in the CNS. We will explain the changes below.

```
[Statedef 200]
type = S
physics = S
movetype = A  ;<-- changed from "I" to "A"
ctrl = 0
anim = 200
velset = 0

[State 200, 1] ;<-- Added this state controller
type = HitDef
trigger1 = AnimElem = 2
attr = S, NA
animtype  = Light
damage    = 10
guardflag = MA
pausetime = 12,12
sparkxy = 0,-55
hitsound   = 5,0
guardsound = 6,0
ground.type = High
ground.slidetime = 12
ground.hittime  = 15
ground.velocity = -5
air.velocity = -2.5,-3.5

[State 200, 2]
type = ChangeState
trigger1 = AnimTime = 0
value = 0
ctrl = 1
```

The movetype parameter in the StateDef is set to A for "attack". Remember to do this for all attack states. As before, P1 changes back to his standing state after his animation is over.

That HitDef controller looks like a monster! Do not worry, we will go through it slowly.

```
type = HitDef
trigger1 = AnimElem = 2
```

This specifies the controller type as HitDef, which stands for "Hit Definition". It is triggered on the second element of animation. Any Clsn2 box from the time the trigger was activated will take on this hit definition.

If, for example, you had a Clsn1 in both the second and third element of animation, triggering a single HitDef at the second element makes it apply to both elements of animation. So P1 will hit at most once: if the second element hits, the third will miss. If the second element misses, the third can still hit. To make the attack hit twice, you must trigger a HitDef for each of the two elements.

```
attr = S, NA
```

This is the attribute of the attack. It is used to determine if the attack can hit P2. In this case, it is a Standing Normal Attack.

attr has the format attr = arg1, arg2, where:

- arg1 is either S, C or A. Similar to statetype for the StateDef, this says whether the attack is a standing, crouching, or aerial attack.
- arg2 is a 2-character string. The first character is either N for "normal", S for "special", or H for "hyper" (or "super", as it is commonly known). The second character must be either A for "attack" (a normal hit

attack), T for "throw", or P for projectile.

```
animtype = Light
```

This refers to the type of animation that P2 will go into when hit by the attack. Choose from light, medium, hard or back. The first three should be self-explanatory. Back is the animation where P2 is knocked off her feet.

```
damage = 10
```

This is the damage that P2 takes when hit, and it does no damage if guarded. If we changed that line to damage = 10, 1, then it would do 1 point of damage if guarded.

```
guardflag = MA
```

guardflag determines how P2 may guard the attack. Here, it may be guarded high(standing), low (crouching) and in the air. The argument must be a string of characters that includes any of the following: H for "high", L for "low" or A for air. M (mid) is equivalent to saying HL.

```
pausetime = 12,12
```

This is the time that each player will pause on the hit. The first argument is the time to freeze P1, measured in game-ticks. The second is the time to make P2 shake before recoiling from the hit.

```
sparkxy = 0,-55
```

This is where to make the hit/guard spark. The arguments must be in the form x, y. x is relative to the front of P2. A negative x makes a spark deeper inside P2. y is relative to P1. A negative y makes a spark higher up.

```
hitsound = 5,0
```

This is the sound to play on hit (from fight.snd). The included fight.snd lets you choose from 5,0 (light hit sound) through to 5,4 (painful whack). To play a sound from the player's own SND file, precede the first number with an S. For example, hitsound = S1,0.

```
guardsound = 6,0
```

This is the sound to play on guard (from fight.snd). Right now all we have is 6,0. To play a sound from the player's own SND file, precede the first number with an S.

```
ground.type = High
```

This is the kind of attack for ground attacks (it also defaults to air attacks if you do not specify an air.type parameter). In this case, it is a high attack. Choose from High for attacks that make P2's head snap backwards, Low for attacks that look like they hit in the stomach, Trip for low sweep attacks, or None to not do anything to P2. High and Low attacks are the same on P2 if the AnimType is Back.

```
ground.slidetime = 12
```

This is the time in game-ticks that P2 will slide back for after being hit (this time does not include the pausetime for P2). Applicable only to hits that keep P2 on the ground.

```
ground.hittime = 15
```

Time that P2 stays in the hit state after being hit. Applicable only to hits that keep P2 on the ground.

```
ground.velocity = -5
```

Initial x-velocity to give P2 after being hit, if P2 is in a standing or crouching state on the ground. You can specify a y-velocity as the second argument if you want P2 to be knocked into the air, eg. `ground.velocity = -3, -2`.

```
air.velocity = -2.5,-3.5
```

Initial velocity to give P2 if P2 is hit in the air.

There are more things that you can control in a `HitDef`. See the sctrls documentation for details.

Common states (common1.cns)

If you look at a player's DEF file, you will see the line:

```
stcommon = common1.cns  ;Common states
```

Every player shares some common states, which are the basic parts of the game engine. These common states are found in data/common1.cns. Some examples are states for running and getting hit. A full list is available in Special State Numbers.

If there is a common state that you would like to override for a certain player, all you need to do is make a state in that player's CNS with the same number as the one you would like to override. Then, when the player changes to that certain state number, he will enter that new state, instead of the one in common1.cns.

You should remember that when overriding certain states that have special properties coded inside M.U.G.E.N, the new states you make will still have the same special properties as the ones you overrode. For example, the run state (state 100) sets the player's velocity to whatever values you specified in his player variables. If you override state 100, the new state will still have the property of setting that player's velocity.

A common example is overriding the running state. M.U.G.E.N's default behaviour for the running state is to have the player continue moving forward at a constant speed, until you let go of the forward key. At that point he returns to the stand state.

Now, let's say we want that player (let us call him P1) to instead hop forward, just like the default double-tap back hop. You can make a state in P1's CNS:

```
; RUN_FWD (overridden to dash-type)
[Statedef 100]
type    = S    ;Running is on the ground
physics = N    ;We'll define our own physics
anim = 100     ;Anim action 100
ctrl = 0       ;No control for duration of dash

[State 100, 1] ;To start dashing forwards
type = VelSet
trigger1 = Time = [0,5]
x = 6

[State 100, 2] ;Friction after initial dash
type = VelMul
trigger1 = Time > 5
x = .85

[State 100, 3] ;
type = ChangeState
trigger1 = AnimTime = 0
value = 0
ctrl = 1
```

Here, we assume that Action 100 has a finite looptime. The velocity in `run.fwd` under `[Velocity]` of the player variables is not really ignored, but `[State 100,1]` overrides that detail by setting the x-velocity to 6.

Special State Numbers

Unless you plan to override a common state, avoid choosing state numbers in the range 0-199 and 5000-5999. Here is a list of states in common1.cns.

Number	Description
0	Stand
10	Stand to crouch
11	Crouching
12	Crouch to stand
20	Walk
40	Jump start
45	Air jump start
50	Jump up
52	Jump land
100	Run forward
105	Hop backwards
106	Hop backwards (land)
120	Guard (start)
130	Stand guard (guarding)
131	Crouch guard (guarding)
132	Air guard (guarding)
140	Guard (end)
150	Stand guard hit (shaking)
151	Stand guard hit (knocked back)
152	Crouch guard hit (shaking)
153	Crouch guard hit (knocked back)
154	Air guard hit (shaking)
155	Air guard hit (knocked away)
170	Lose (time over)
175	Draw game (time over)
190	Pre-intro
191	Intro (override this state to give character an intro)
5000	Stand get-hit (shaking)
5001	Stand get-hit (knocked back)
5010	Crouch get-hit (shaking)
5011	Crouch get-hit (knocked back)
5020	Air get-hit (shaking)
5030	Air get-hit (knocked away)
5035	Air get-hit (transition)
5040	Air get-hit (recovering in air, not falling)
5050	Air get-hit (falling)
5070	Tripped get-hit (shaking)
5071	Tripped get-hit (knocked away)
5080	Downed get-hit (shaking)
5081	Downed get-hit (knocked back)

Number	Description
5100	Downed get-hit (hit ground from fall)
5101	Downed get-hit (bounce off ground)
5110	Downed get-hit (lying down)
5120	Downed get-hit (getting up)
5150	Downed get-hit (lying defeated)
5200	Air get-hit (fall recovery on ground; still falling)
5201	Air get-hit (fall recovery on ground)
5210	Air get-hit (fall recovery in air)
5500	Continue screen animation
5900	Initialize (at the start of the round)

Expressions

MUGEN supports arithmetic expressions in most triggers and state controllers. This allows much more flexible and customizable behavior than would be possible using only fixed values. This section gives a description of expression concepts and syntax.

Data types

MUGEN uses three data types: 32-bit integers, 32-bit floats, and a special null value, "bottom". Integers represent whole numbers between -2^31 and 2^31-1, or about -2 billion to 2 billion. Floats are single-precision floating-point numbers. That is, they are numbers with a "decimal part" and about 7 significant figures of precision. Floats can represent small fractions or very large numbers.

When you write numbers in an expression, MUGEN deduces the data type from the presence of a decimal point. Thus "7" is always an int, for example. If you wanted 7 as a float, then you would write "7.0".

"bottom" is a special data type that zeros out any expression that it appears in (with a few very limited exceptions). Its presence signifies some kind of error condition. You should try to code in such a way that bottom is never produced. For details, you can see the dedicated section on bottom.

The behavior of arithmetic expressions depends heavily on the underlying data types used to represent numbers. Also, state controllers may expect their input to be given as a certain type, and will give errors if the wrong type is supplied.

"Type promotion" occurs when values of different data types need to be combined in some way (e.g., by addition). Generally this means that an integer will be changed to a float, with possible loss of precision in the process. In the sections below we will note any relevant type promotion scenarios.

Arithmetic Operators

Arithmetic operators allow you to perform basic operations like addition, multiplication, division, etc. MUGEN has a selection of operators that should be familiar to most programmers. They are as follows:

- +

  Adds two numbers. If x and y are both ints, then x + y is also an int. If both are floats, then x + y is a float. If one is a float, then the other is promoted to float, and then the two floats are added.

- -

  Subtraction of numbers. Type promotion works the same as addition.

- *

  Multiplication of two numbers. Type promotion works the same as addition.

- /

  Division of two numbers. If x and y are both ints, then x/y gives the integer quotient, i.e., the number of times that y goes evenly into x. For instance, 7/2 = 3. If x and y are both floats, then x/y returns a float. Our previous example would become 7.0/2.0 = 3.5. If only one of x or y is a float, then the other is promoted and float division is performed. Division by 0 will produce bottom.

- %

  The remainder or mod operator. If x and y are both ints, then x % y gives the remainder when x is divided by y. If one or both are floats, or if y is 0, then bottom is produced.

- **

  The exponentiation operator. If x and y are both non-negative ints, then x**y gives an int representing x raised to the power of y. (We define 0**0 = 1). However, it is very easy to overflow the maximum possible value for an int when computing large powers. In these cases MAX_INT (the largest possible integer) will be returned, and a warning will be generated. If one of x or y is negative or is a float, then both arguments are promoted to float and x**y is computed as an exponentiation of real numbers. An invalid exponentiation like -1 ** .5 will produce bottom.

- !

  Logical NOT operator. !x evaluates to 0 (int) if x is nonzero, and 1 (int) if x is zero.

- &&

  The logical AND operator. x && y evaluates to 1 (int) if x and y are both nonzero, and to 0 (int) otherwise.

- ||

  The logical OR operator. x || y evaluates to 1 (int) if one or more of x and y is nonzero, and to 0 (int) otherwise.

- ^^

  The logical XOR operator. x ^^ y evaluates to 1 (int) if exactly one of x and y is nonzero, and to 0 (int) otherwise.

- ~

  The bitwise NOT operator. ~x inverts the bits of x's binary (two's complement) representation. Produces bottom if x is a float.

- &

  The bitwise AND operator. The nth bit of x&y is set if and only if the nth bits of both x and y are set. Produces bottom if x or y is a float.

-

  The bitwise OR operator. The nth bit of x|y is set if and only if the nth bit of either x or of y (or both) is set. Returns bottom if either x or y is a float.

- ^

  The bitwise XOR operator. The nth bit of x^y is set if and only if the nth bit of exactly one of x and y is set. Returns bottom if either of x or y is a float.

- =

  The equality operator. If x and y are both ints or both floats, x = y evaluates to 1 (int) if x and y are equal, and 0 otherwise. If exactly one of x or y is a float, then they are both converted to floats before testing for equality.

- :=

  The assignment operator. An unredirected variable name (var(n) or fvar(n) for suitable values of n) must appear on the left-hand side. If the left-hand side contains an integer variable, then the right-hand side is truncated to an integer before assignment. If the left-hand side contains a float variable, then the right-hand side is converted to float if necessary before assignment. In both cases, the value of the expression is the value that is assigned to the variable.

- !=

  The inequality operator. If x and y are both ints or both floats, x != y evaluates to 1 (int) if x and y are not equal, and 0 otherwise. If exactly one of x or y is a float, then they are both converted to floats before testing for equality.

- <

  The less-than operator. If x and y are both ints or both floats, x < y evaluates to 1 (int) if x < y, and 0 otherwise. If exactly one of x or y is a float, then they are both converted to floats before testing for equality.

- <=

  Similar to <, with the exception that if x = y, then x <= y returns 1 (int).

- >

  The greater-than operator. If x and y are both ints or both floats, x > y evaluates to 1 (int) if x > y, and 0 (int) otherwise. If exactly one of x or y is a float, then they are both converted to floats before testing for equality.

- >=

  Similar to >, with the exception that if x = y, then x >= y returns 1 (int).

- =[,] !=[,] =[,) !=[,) =(,] !=(,] =(,) !=(,)

  Interval operators. These take three arguments, x, y, and z. If any of x, y, or z is a float, they are all converted to floats. After conversion if necessary, x = [y,z] is equivalent to (x >= y) && (x <= z). Similarly, x = (y,z) is equivalent to (x > y) && (x < z). The half-open intervals have the obvious meaning.

  The negated interval operators work as follows: x != [y,z] is equivalent (after conversion if necessary) to (x < y) || (x > z). x != (y,z) is equivalent to (x <= y) || (x >= z). The half-open intervals again have the obvious meaning.

  You can view the interval operators as producing the appropriate open, closed, or half-open intervals on the ints or the floats. The = symbol means set membership in this context.

  Some restrictions apply on where intervals may be placed in an expression. See the section on syntax for details.

Precedence and associativity of operators

If you consider an expression like 3+2*5, the result is different depending if you evaluate the * first (yielding 13) or if you evaluate the + first (yielding 25). To disambiguate expressions such as this, operators are assigned distinct precedence levels. In this case, the precedence of * is higher than the precedence of +, so the * is evaluated first, then the + is applied to the result. So the correct answer is 13.

If two operators share the same precedence, then the expression is evaluated from left to right, except for the unary operators and the assignment operator, which associate right to left. For instance, * and / share the same precedence, so 5.0*5/6 evaluates to 25.0/6, which evaluates to 4.166667. On the other hand, 5/6*5.0 evaluates to 0*5.0, which evaluates to 0.0. In contrast, because unary operators associate right to left, -!0 is grouped as -(!0), which evaluates to -(1), which then evaluates to -1.

If part of an expression is grouped in parentheses (), then that part of the expression is evaluated first. For instance, in the expression (3+2)*5, the + is evaluated first, giving 5*5, which then evaluates to 25. If parentheses are nested, the innermost parentheses are evaluated first.

Operator precedence is basically the same as in C. The complete list of operator precedence, from highest to lowest, is as follows:

Operator(s)	Precedence Level
! ~ - (unary operators)	Highest
**	
* / %	
+ -	
> >= < <=	
= != intervals	
:=	
&	
^	
|	
&&	
^^	
||	Lowest

Programmers are encouraged to parenthesize as necessary to maintain clarity. Otherwise, bugs due to subtle misunderstanding of operator precedence are almost assured.

Expression syntax

Basically, any normal arithmetic expression is allowable. In addition, since the relational operators (>, <=, etc.) are viewed as returning ints, it is possible to operate on their return values, giving some unusual-looking expressions like

```
1.0 = (2 = (1 > 0) + !(0 < 1))
```

The 1 > 0 term evaluates to 1, and the 0 < 1 term evaluates to 0. Hence !(0 < 1) evaluates to 1, so the expression simplifies to

```
1.0 = (2 = 1 + 1)
```

Since 2 = 1 + 1, the term in parentheses evaluates to 1, so the expression further simplifies (after type conversion) to

```
1.0 = 1.0
```

which evaluates to 1 (int), since the equality holds.

A notable restriction in expression syntax is that interval operators are only allowed to appear on the rightmost side of an expression. If part of an expression is enclosed in parentheses, then that part is

considered a subexpression, and an interval is allowed to appear on the right side of that subexpression. So the following is a well-formed expression, which evaluates to 0:

```
(1 = [0,2]) = (0,1)
```

But the following is not well-formed:

```
1 = [0,2] = (0,1)
```

In addition, no operator symbols other than = or != may appear before an interval. So an expression like 5 > [0,2], or 4 + [1,4), is not allowed.

In comma-separated parameter lists, such as the arguments to some function-type triggers or parameters to state controllers, each expression in the list is considered a separate subexpression, and therefore intervals may appear at the end of those subexpressions.

Condition- and function-type triggers

For historical reasons, two distinct constructs are both called "triggers." The first is what might be more properly called a condition-type trigger, and the second is what might be more properly called a function-type trigger. For instance, in the CNS, a typical state controller might look like

```
[State 1234, 5]
type = ChangeState
trigger1 = time = 0
value = 0
```

The entire line "trigger1 = time = 0" is a condition-type trigger. If the expression "time = 0" evaluates to a nonzero value, then the ChangeState controller is executed. If the expression "time = 0" evaluates to zero, then the ChangeState controller is not executed. Thus whether the condition is zero or nonzero affects whether the controller is triggered.

On the other hand, the word "time" appearing in the expression is a function-type trigger. It returns a value, namely, the amount of time that the player has been in state 1234. Note that a function-type trigger doesn't "trigger" anything. It just gives a value that can be acted on within the expression.

To further illustrate the difference, let us consider a different state controller:

```
[State 1234, 5]
type = VarSet
trigger1 = 1
v = 0
value = time + 5
```

Note that the condition-type trigger "trigger1 = 1" now contains no function-type triggers within it. Since the expression "1" always evaluates to 1, the controller will be triggered every frame. To determine what value to assign var0, the expression "time + 5" is evaluated. The function-type trigger "time" returns the player's statetime. Then 5 is added and the result is stored in var0.

A complete list of function-type triggers can be found in trigger.html.

In general, which of the two types of triggers is meant is clear from context. Where there is some ambiguity, the terms "condition-type trigger" and "function-type trigger" will be used.

Trigger redirection

In the above example, the time trigger returned the statetime of the player. But sometimes, one might wish to check the statetime of the player's target, or the player's parent (if the player is a helper), etc. This can be accomplished by preceding the trigger name by a keyword indicating whose information should be returned. This process is known as trigger redirection. For example,

5 + (parent, time)

returns 5 + the player's parent's statetime.

The complete list of redirection keywords is the following:

- parent

  Redirects the trigger to the player's parent. (Player must be a helper.)

- root

  Redirects the trigger to the root.

- helper

  Redirects the trigger to the first helper found. See the related trigger "NumHelper" in the trigger documentation.

- helper(ID)

  ID should be a well-formed expression that evaluates to a positive integer. The trigger is then redirected to a helper with the corresponding ID number.

- target

  Redirects the trigger to the first target found.

- target(ID)

  ID should be a well-formed expression that evaluates to a non- negative integer. The trigger is then redirected to a target with the corresponding targetID. The targetID is specified in the "ID" parameter of a HitDef controller.

- partner

  Redirects the trigger to the player's partner. Normal helpers and neutral players are not considered opponents. See the related trigger "numpartner" in the trigger documentation.

- enemy

  Redirects the trigger to the first opponent found. Normal helpers and neutral players are not considered opponents. See the related trigger "numenemy" in the trigger documentation.

- enemy(n)

  n should be a well-formed expression that evaluates to a non- negative integer. The trigger is redirected to the n'th opponent.

- enemyNear

  Redirects the trigger to the nearest opponent.

- enemyNear(n)

  n should be a well-formed expression that evaluates to a non- negative integer. The trigger is redirected to the n'th-nearest opponent.

- playerID(ID)

  n should be a well-formed expression that evaluates to a non- negative integer. The trigger is redirected to the player with unique ID equal to ID. See the "ID" and "PlayerExistID" triggers in the trigger documentation.

If the trigger is redirected to an invalid destination (for instance, if it is retargeted to a helper when none exist), then bottom is returned.

Note: recursive redirection (e.g., "root,target,time") is not supported.

bottom

There are several sources of unrecoverable error in expressions. For instance, one could attempt to divide by 0, evaluate the square root of a negative number, or attempt to redirect a trigger to a nonexistent destination. In these situations, bottom is used as a way to complete expression evaluation gracefully. If bottom appears anywhere in an expression (with two exceptions to be noted below: see Special Forms), then the whole expression's value becomes bottom. For instance, consider the expression:

```
5 + fvar(0) ** 0.5
```

If fvar(0) were equal to -1 at the time this was evaluated, then the expression would become 5 + bottom, yielding bottom.

Condition-type triggers consider bottom to be 0. Thus, an expression that generates an error will never cause a trigger to fire. So, for example, in

```
type = ChangeState
trigger1 = helper, statetype = A
value = 0
```

the ChangeState controller would never be executed if no helpers existed, because the expression "helper, statetype = A" would evaluate to bottom, which is 0.

Typically a warning is printed to MUGEN's debug console when bottom is generated. This is because presence of bottom indicates a possible error or ambiguity in logic. For instance, in the above ChangeState example, if no helpers exist, then the statement "helper, statetype = A" is vacuous, and it is not clear whether it should be considered to be true or false.

Special Forms

The IfElse and Cond triggers handle bottom in a special way. Both of these triggers take the form:

```
<trigger_name>(<exp_cond>,<exp_true>,<exp_false>)
```

<exp_cond> is a condition expression which, depending if it is nonzero, controls which of <exp_true> or <exp_false> gets returned. If bottom is produced in the expression that is not returned, then it does not propagate out to the rest of the expression. For instance, consider the expression `IfElse(time > 0, 1.0/time, 2))`. If time > 0, then the value of the expression is `1.0/time`, which is a valid float value. If time = 0, then the value of the expression is 2, even though the unused branch divides by 0 and thus produces bottom.

However, even though the above IfElse trigger never returns bottom, it still has an annoying feature: the division by 0 still generates a warning to the debug output. This is because IfElse evaluates all its arguments, even the unused ones. In contrast, Cond will only evaluate the arguments that it actually uses. Thus, if we rewrote our expression as `Cond(time > 0, 1.0/time, 2)`, the argument `1.0/time` would never get evaluated when time was 0, and thus no warning would be generated.

You may wonder when to use Cond versus IfElse. The answer is that you almost always want to use Cond, unless one of <exp_true> or <exp_false> has a side effect that you need. In other words, if you are doing a variable assignment within one of the branches that always needs to be executed, regardless of the value of <exp_cond>, then you need to use IfElse. Otherwise, you should use Cond, especially for the purposes of isolating bottom.

Avoiding warnings

Several common error conditions often produce a flood of warnings in characters. Here are some of the most common scenarios and how to avoid them.

- Coercion of floats to ints

  If MUGEN expects an int somewhere (e.g., a var number), but a float is provided, then MUGEN will convert the int to a float. However, it will complain, because this may represent an error on your part. To get rid of the warning, indicate that you know what you are doing by supplying an explicit conversion to int with floor() or ceil().

- Nonexistent trigger redirection targets

  If you redirect to a target that doesn't exist, such as a helper when you don't have one, then bottom is produced and a warning is logged. This is because the expression is logically ambiguous. You can avoid the warning by checking for existence of the target before redirecting. So,

  ```
  trigger1 = helper(1234), time > 20
  ```

  would generate a warning if helper 1234 didn't exist, but

  ```
  trigger1 = numhelper(1234) > 0
  trigger1 = helper(1234), time > 20
  ```

  would not. This is because the first trigger1 line causes evaluation to be aborted before attempting the redirection.

  Alternatively, you can use the Cond trigger to isolate code that could potentially produce a warning. Suppose we wanted an expression that gives helper(1234)'s life, if it exists, or the player's life otherwise. Then we could write

  ```
  Cond(numhelper(1234) > 0, (helper(1234), life), life)
  ```

  (The extra parentheses around `helper(1234), life` are not required, but improve readability.)

- Missing required animations/sprites

  This is not an expressions problem. Add the animations/sprites!

Expressions in trigger arguments

Most function-type triggers either take no arguments or take arguments in a parameter list. For instance, the time trigger takes no arguments, whereas the ifelse trigger takes three arguments

ifelse(exp1,exp2,exp3)

where exp1, exp2, and exp3 are all valid expressions. In this kind of situation, exp1, exp2, and exp3 are all considered separate subexpressions, so intervals can appear on the rightmost end of each of those subexpressions. The order of evaluation of parameter lists is from left to right.

Due to irregular syntax, some old function-type triggers cannot take expressions as their arguments. Because of this, they cannot be integrated into expressions in the standard manner. For nonstandard triggers of this type, the triggers can only appear with certain sets of operators and arguments (which are outlined in trigger.doc). In particular, these triggers cannot take expressions as their arguments. For instance,

```
trigger1 = AnimElem = (1+1)
```

is an invalid expression.

Old-style function-type triggers appear only in "clauses" of the form "trigger, relational operator, argument". These clauses are treated as a single unit (specifically, a single nullary trigger) for the purposes of expression evaluation. This means, among other things, that the concept of operator precedence is not applicable to operators appearing within an old-style function-type trigger clause. For instance, in

```
trigger1 = AnimElem = 5 + 4
```

the expression is broken down into three units:

```
 AnimElem=5    +    4
|__________|  |_|  |_|
```

The "AnimElem=5" unit is treated as the name of a nullary trigger, hence the + operator does not have precedence over the = appearing within the name "AnimElem=5". In other words, this expression means something like "Execute the trigger called 'AnimElem=5', then add 4 to the result."

Some old-style function-type triggers have expression-capable replacements. These are as follows:

- AnimElem, superseded by AnimElemTime
- TimeMod, superseded by the % operator
- ProjHit, ProjContact, ProjGuarded; superseded by ProjHitTime, ProjContactTime, and ProjGuardedTime

For the complete list of irregular triggers, see trigger.html. Irregular triggers are marked with ***.

Expressions in state and state controller parameters

For the most part, any parameter to the statedef or a state controller can be an expression. The exceptions are parameters that are given as strings. For instance, hit attributes, guardflags, etc. cannot be specified as expressions. Also, the ignorehitpause and persistent parameters (in all controllers) are irregular in that they cannot take expressions.

State controller parameters are evaluated at the time the controller is triggered, and are not subsequently re-evaluated unless the controller is triggered again. Parameters that are given as comma-separated lists are evaluated from left to right. To achieve continual evaluation of controller parameters, the controller must be continually triggered.

In the case of certain controllers such as HitDef, it is not always wise to use continual triggering, since this decreases performance and also may lead to undesired behavior. In this case, the programmer may wish to try to delay triggering the HitDef as long as possible, so as to evaluate the HitDef parameters right before they are used.

Organizing for efficiency

Expression handling does not tax modern computers, so readability of your code is more important than micro-optimization of expressions. However, following certain good practices will increase efficiency without harming clarity.

MUGEN evaluates condition-type triggers for a state controller in the following order: First it evaluates triggeralls, from top to bottom. If any of the triggeralls evaluates to 0, the rest of the triggers are skipped and evaluation proceeds to the next controller. If all the triggeralls evaluate to nonzero, then the engine starts to evaluate trigger1's, from top to bottom. If any of these evaluate to 0, then evaluation skips to the first trigger2, and so on. If all the triggers in a block (besides triggerall) evaluate to nonzero, then the state controller parameters are evaluated and the controller is triggered.

In other words, the logical evaluation of triggers is short-circuited. In C-like notation, this setup might be denoted

```
triggerall,1 && triggerall,2 && ... && ((trigger1,1 && trigger1,2
  && ...) || (trigger2,1 && trigger2,2 && ...) || ... )
```

where (e.g.) trigger1,2 denotes the second trigger1 line; trigger2,1 denotes the first trigger2 line; etc. The logical evaluation of this trigger group would then be short-circuited as in C.

Because of this system, considerable efficiency gains can be attained by organizing expressions so that the condition-type triggers are as simple, and as few in number, as possible. The bulk of the "work" can be offloaded to the state controller parameters, which are only evaluated once at trigger time, instead of every frame that the player is in the state. For instance,

```
[State -1]
type = ChangeState
trigger1 = command = "a"
trigger1 = power < 1000
value = 3000

[State -1]
type = ChangeState
trigger1 = command = "a"
trigger1 = power >= 1000
value = 3001

[State -1]
type = ChangeState
trigger1 = command = "a"
trigger1 = power >= 2000
value = 3002
```

could be more compactly expressed as

```
[State -1]
type = ChangeState
trigger1 = command = "a"
value = 3000 + (power >= 1000) + (power >= 2000)
```

You can also help the engine out by placing triggeralls that are most likely to be false at the top of the triggerall block. Similarly, the trigger1 block should be the most likely block to trigger, but within the trigger1 block itself, the triggers that are most likely to evaluate to 0 should be placed highest. For state controllers with many triggers containing duplicated conditions, it may be better to break the controllers up into two separate blocks, each with its own set of triggeralls.

If you have a complex condition which is being used as the trigger condition for many consecutive state controllers, you may choose to save the value of the condition in a variable, then use that variable as a trigger to the subsequent controllers. For instance,

```
trigger1 = (command="abc" && command!="holddown" && power>=1000) ||
           (command="abc" && command!="holddown" && var(5)) ||
           ((command != "abc" || command = "holddown") && power>=2000)
```

could be written as (assuming var(0) is available):

```
trigger1 = (var(0):=(command="abc" && command !="holddown") && power>=
         1000) || (var(0) && var(5)) || (!var(0) && power>=2000)
```

Here, you must balance the readability gain of factoring out common subexpressions against the readability loss of using the := operator.