ZOMBIE

Introduction

ZOMBIE is a programming language designed for Necromancers, particularly evil ones. (Actually, what other sorts are there?) ZOMBIE is an acronym, and stands for Zombie-Oriented Machine-Being Interface Engine.

Design Principles

- The language should allow the necromancer to animate dead bodies, summon and control spirits, and solve any computable problem.
- There should be sensible guarantees against overwriting system memory, monopolising CPU cycles, and releasing malicious entities into the world.
- The language must be inherently evil.

Language Concepts

Entities

Entities are the main structural units in a ZOMBIE program. They come in several types:

- **Undead**
  Undead are spirits of the dead, unable to fully depart the physical realm for various reasons. These include:
  1. Enslaved undead, such as **zombies**, who are animated by necromancers.
  2. Restless undead, such as **ghosts**, who remain behind either because of an unconsecrated death or to avenge an injustice.
  3. Free-willed undead, such as **vampires**, who deliberately pervert their own death to remain active and wreak evil.
- **Demons**
  Demons are free-willed malicious entities who usually inhabit other planes of existence. They can be summoned to the material world and bound to perform tasks, but care must be taken on two fronts:
  1. Demons are notoriously perverse literalists and will twist the intent of commands to the utmost while keeping to a strict literal interpetation.
  2. If the protective wards used to hold a demon under control are imperfect in any way the demon will turn on its summoner.
- **Djinn**
  Djinn are free-willed entities of capricious nature. They can often be commanded by a person who controls an object which is bound to them in an unbreakable connection. Woe betide if you lose control of the object, however, since djinn harbour great grudges against those who dare to command them. Some of the most powerful djinn can grant wishes.

The safest entities to use are enslaved undead (thus the focus and name of the programming language). Other entities may be used though, to achieve more complex computing algorithms, but in such cases extra precautions need to be taken.

- Zombies may be *declared* and then *animated*. An animated zombie can be expected to do whatever it is commanded to do, straight away.
- Ghosts may be declared and then *disturbed*. A disturbed, and hence restless, ghost will *eventually* do what is asked of it.
- Vampires may be declared. They do not require animating or disturbing, and will do what is asked of them, but not necessarily in the order requested.
- Demons may be declared. They do not require animating or disturbing. A demon will do what is asked of it, if the proper precautions are taken, but may summon other demons to help it. This may or may not be a good thing.
- Djinn may be declared. They do not require animating or disturbing. A djinn may or may not do what is asked of it.

Threading

ZOMBIE runs in a multithreaded environment. Several entities may be animated at once, and each may perform multiple tasks at once. The relative speeds and orders in which the entities perform tasks is undefined, and should not be relied on by the programmer.

Entities and their tasks may be **active** or **inactive**. An active entity is one which has been animated, disturbed, or is free-willed. A zombie or ghost that has been summoned and bound is inactive. An active task is one that has been animated; an inactive task is one that has been bound, or that the entity has completed.

All the active entities in a ZOMBIE program process their active tasks.

- Zombies process their active tasks in sequence, beginning from the first task defined, as quickly as they can. They perform each task exactly once.
- Ghosts process their active tasks in sequence, beginning from the first task defined, but they may wait for an undefined time before beginning and between each task. They eventually perform each task exactly once.
- Vampires process their active tasks in random order, as quickly as they can. They perform each task exactly once, and complete one task before beginning the next.
- Demons process their active tasks in random order, as quickly as they can. They may decide to perform tasks multiple times before becoming inactive, but will perform each task at least once. They may perform multiple tasks at the same time. They may also summon additional demons exactly like themselves.
- Djinn process their active tasks in random order, as quickly as they can. They may decide to perform each task multiple times, or not at all, before becoming inactive. They may perform multiple tasks at the same time.

Comments

Syntax Elements

ZOMBIE programs are in the form of a list of **entity declarations**. Entities may be declared in any order, and declarations are completed before any entities are activated.

Note that syntax errors in ZOMBIE programs can be extremely dangerous, especially if demons or djinn are invoked. These spirits may escape the CPU and wreak havoc in the outside world without the correct binding commands. The language author recommends all computers running ZOMBIE code be surrounded by correctly inscribed pentagrams or other protective seals.

Data Values

Data values are free format arithmetic or string expressions, conforming to standard rules. Typing is weak. String and numeric values are converted implicitly as required, in a "sensible" manner.

Undead can remember exactly one data value, either numeric or string. If multiple copies of the same entity are invoked, they all remember the same data value - if one copy remembers a new value, the value remembered by all copies changes to the new value.

Entity Declaration

Entities are the basic elements of a ZOMBIE program. Valid ZOMBIE programs must declare at least one entity.

Entities are declared with the following structure:

```
entity-name is [a|an] entity-type { entity-statements }
```

Entity-type is one of the following:

- `zombie|enslaved undead`
- `ghost|restless undead`

- `vampire|free-willed undead`
- `demon`
- `djinn`

Entity-name is any well-formed identifier string.

Entity-statements is a list of any valid statements, which may include entity declarations.

Entity Declaration Statements

Some statements delineate matched pairs, which may be nested to any level. All the following combinations are properly matched pairs:

- `summon` / `animate`
- `summon` / `bind`
- `summon` / `disturb`
- `task` / `animate`
- `task` / `bind`

The following statements are used to declare entities:

- `animate`
  This statement concludes a `summon` or `task`. When concluding a `summon`, it concludes the entity summoning and, if a zombie, makes the entity active. It does not make other entities active. When concluding a `task`, it ends the command definition and marks that command as an active command for new entities.
- `bind`
  This statement concludes a `summon` or `task`. When concluding a `summon`, it concludes the entity summoning, leaving the entity inactive if it is a zombie or ghost. When concluding a `task`, it ends the command definition and marks that command as an inactive command for new entities.
- `task` *`task-name`*
  This statement defines a new task named *task-name*. The task statements appear after this statement and before the next matching `animate` or `bind` statement.
- `disturb`
  This statement concludes a `summon` and, if a ghost, makes the entity active. It does not make other entities active.
- `summon`
  This statement marks the beginning of an entity's task definitions.

Task Statements

These instruct the entity to do various tasks. They can be considered to be written and parsed in reverse-reverse-Polish notation. i.e. each statement has a temporary data stack, which can hold both data and entities. Values are added to the data stack beginning at the end of the statement, and working back to the beginning. For example, the statement

```
remember Zombie1 moan Zombie1 moan Zombie2
```

works like this:

1. Put Zombie2 on the stack.
2. `moan` causes the top stack element, Zombie2, to be replaced by its remembered data value.
3. Put Zombie1 on the stack.
4. `moan` causes the top stack element, Zombie1, to be replaced by its remembered data value.
5. Put Zombie1 on the stack.
6. `remember` causes the top stack element, Zombie1, to remember the sum of all elements in the stack.

The following task statements are defined in ZOMBIE. They all instruct the entity on top of the data stack to do something. If the top value on the data stack is not an entity, the calling entity is the one that does the command.

- `animate [`*`entity-name`*`]`
  Activates a new copy of the named entity, if it is an inactive zombie.

- `banish [entity-name]`
  Immediately deactivates the entity.
- `disturb [entity-name]`
  Activates a new copy of the named entity, if it is an inactive ghost.
- `forget [entity-name]`
  Instructs the entity to forget its remembered data value.
- `invoke [entity-name]`
  Invokes a new copy of the named entity.
- `moan [entity-name]`
  Instructs the named entity to moan its remembered data value, and to keep remembering it.
- `remember [entity-name]`
  Instructs the entity to remember the sum of the values in the statement stack. Since a zombie can only remember one thing at a time, this causes it to forget any previously remembered value.
- `say [entity-name] text`
  Print the *text* to the standard output. (It doesn't matter what entity does this, as the result is the same.)

Flow Control

- `shamble ... until variable`
  Causes the entity to repeat the statements between `shamble` and `until` until the variable evaluates to true.
- `shamble ... around`
  Causes the entity to repeat the statements between `shamble` and `around` in an infinite loop.
- `stumble`
  Causes the current task to become inactive immediately.
- `taste variable good ... bad ... spit`
  If the variable evaluates to true, causes the entity to perform the statements between `good` and `bad`, otherwise perform the statements between `bad` and `spit`.

Operators

- `remembering [entity-name] variable`
  Boolean operator that evaluates to true if the entity is currently remembering a data value equal to the given variable, false otherwise.
- `rend`
  This operator pops the top two value off the statement stack, divides the second value by the top value, and puts the result back on the statement stack.
- `turn`
  This operator replaces the top value of the statement stack with its negative.

Sample Programs

Hello World

```
HelloWorld is a zombie
summon
        task SayHello
                say "Hello World!"
        animate
animate
```

Fibonacci Numbers

```
Zombie1 is a zombie
summon
        remember 1
bind

Zombie2 is a zombie
```

```
summon
        remember 1
bind

FibonacciZombie is a zombie
summon
        remember 0
        task SayFibonaccis
                shamble
                        say moan Zombie1
                        say moan Zombie2
                        remember Zombie1 moan Zombie1 moan Zombie2
                        remember Zombie2 moan Zombie1 moan Zombie2
                        remember moan 2
                until remembering 100
        animate
animate
```

Resources

Last updated: Wednesday, 08 February, 2006; 00:40:42 PST.