2009

# Implementação de Linguagens de Script Para Jogos

## MAC 499 – Trabalho de Formatura Supervisionado

Monografia sobre Linguagens de Script para Jogos - Versão 2

Aluno: Daniel Barbosa Stein – NUSP: 5382462
Orientador: Flávio Soares Corrêa da Silva
11/28/2009

## Conteúdo

## A. Introdução

Jogos são aplicações multímedia com o objetivo de divertir os usuários. Pode ser um RPG, ou um jogo de primeira pessoa, ou até de plataforma. Existem inúmeros tipos de jogos.

Muitos destes jogos são baseados em *engines* específicas para eles. São chamadas de *Engines* de Jogos. Define-se *engine* de jogo (ou motor de jogo) como um programa de computador e (ou) conjunto de bibliotecas com o objetivo de simplificar e abstrair o desenvolvimento de jogos ou outras aplicações com gráficos em tempo real, para videogames e (ou) computadores rodando sistemas operacionais.

Como funcionalidade tipicamente fornecida em uma engine, temos: motor gráfico (que renderiza gráficos 2D e 3D), um motor de física (que controla a física ou serve apenas para detecção de colisão), suporte a animação, sons, IA (inteligência artificial), networking (rede), gerência de memória, gerência de arquivos, gerência de linha de execução, suporte a gráfos de cenas e entidades e suporte a linha linguagem de script. Um motor de jogo proporciona uma abstração do hardware, permitindo que o desenvolvedor crie jogos sem a necessidade de conhecer a arquitetura da plataforma-alvo. Muitas *engines* são desenvolvidas a partir de uma API existente, como OpenGL, DirectX, OpenAL e SDL, ou até mesmo de outra *engine*.

Com o uso de uma *engine* de jogo, o desenvolvimento de um jogo é frequentemente agilizado, além de que é possivel criar diferentes jogos usando o mesmo motor.

O objetivo deste trabalho é descrever uma linguagem de script e definir seu formato interno para a implementação da mesma.

## B. Motivação

Em desenvolvimento de jogos, muitas vezes algumas funcionalidades necessitam de um certo nível de customização. Como solução inicial, podemos gerar uma instância para cada caso que precisarmos, mas tal idéia se torna impraticável o quanto mais customizável a funcionalidade é. Além de tal problema, o desenvolvedor do sistema teria que abrir o código-fonte da aplicação para que a funcionalidade possa ser customizada.

Para resolvermos tais problemas, pode-se disponibilizar uma linguagem de scripts que proporciona ao desenvolvedor programar a funcionalidade desejada sem necessitar de criar no código original uma instância específica.

Em jogos, existem inúmeros casos onde isto ocorre, logo o uso de linguagens de script em tal área é de grande importância. Linguagens de script podem ser utilizadas em jogos desde a definição dos parâmetros de um sistema de partículas até IA de um objeto.

# C. Linguagem

## I. Introdução

Em jogos, uma linguagem de scripts normalmente tem um uso específico para que sua performance seja máxima. Por isto, antes de iniciar a implementação da mesma, precisa-se definir onde e como ela será utilizada.

Conforme seu uso, a linguagem pode ser mais integrada ao módulo que a utilizará, assim gerando linguagens bem específicas. Em vários casos, a linguagem gerada é otimizada especialmente para seu local de uso. Um bom exemplo é uma linguagem simples de configuração, que conterá comandos bem específicos ao seu uso.

Em linguagens de script, o código não executará diretamente sobre o processador, e sim em uma máquina virtual especialmente desenvolvida para cada linguagem.

Uma máquina virtual é basicamente um processador implementado em software, que executa uma dada linguagem. A máquina virtual de uma linguagem pode também ser chamada de Interpretador. Cada linguagem de scripts tem sua máquina virtual específica, que precisa ser definida conforme a linguagem de scripts é definida.

O que torna uma linguagem de scripts para jogos diferente de uma linguagem de scripts geral é o fato de que a linguagem de scripts para jogos é utilizada para controlar um motor de jogo, ou algum componente deste, assim contendo comandos e funcionalidades únicas. Além disso, a linguagem de script para jogos é integrada no componente alvo para o qual foi desenvolvida, e sua funcionalidade é limitada ou especializada para este componente.

Como passo inicial da criação da linguagem de exemplo, chamada SimpleScript, inicialmente definiremos como a linguagem será.

## II. Lista de Comandos

Como tipo mais simples de linguagem, temos a “Lista de Comandos”. Nesta linguagem, apenas temos uma lista sequencial de comandos que executarão em ordem. Esta linguagem básica será utilizada para definir as bases da linguagem “SimpleScript”.

Neste tipo simples de linguagem (lista de comandos), todos os comandos são customizados especificamente para seu módulo

de uso. Uma linguagem do tipo “lista de comandos” é bem simples e de uso bem limitado e pontual. Normalmente utilizada em configuradores e processadores específicos por não possuir nenhuma lógica de fluxo (como if) e nem variáveis.

Inicialmente, é necessário definir a estrutura básica de um comando. Esta estrutura será a base de todos os comandos suportados pela linguagem.

O formato do comando será similar ao da linguagem C e cada parâmetro será separado por uma vírgula (,) entre si: comando(parâmetros);

O próximo passo é definir quais tipos de dados pode-se ter como parâmetros. Como esta é uma linguagem simples, e não existem variáveis, a linguagem suportará dois tipos de dados: texto (string) e números. Como número, será utilizado o tipo float do C.

O tipo *float* é um número de ponto flutuante, representado por uma parte inteira e casas decimais. A forma deste número é <parte inteira>.<parte decimal>, sendo que a parte inteira e a decimal são separadas por um ponto.

Textos serão definidos como qualquer dado entre aspas (“) e números quaisquer outros dados que inicie em um número (e que contenham apenas números e no máximo um ponto). Outros tipos de dados serão considerados erro de digitação e serão ignorados pela linguagem (retornando erro ao desenvolvedor).

Definidas as bases do comando, os dados necessários para a definição básica de um objeto “comando” estão disponíveis. Internamente, o objeto comando conterá os seguintes dados:

- Enumerador com id do comando (indicando qual comando o objeto está representando).
- Número de parâmetros lidos.
- Número de parâmetros do tipo texto lidos.
- Número de parâmetros do tipo número lidos.
- Lista com a assinatura dos parâmetros (define os tipos e a ordem em que os parâmetros foram lidos).
- Lista de parâmetros do tipo texto (string).
- Lista de parâmetros do tipo número.

A lista de assinatura dos parâmetros será utilizada durante a interpretação do código. Ela serve para definir a ordem em que os parâmetros foram lidos. Como exemplo, se os parâmetros do comando forem (“texto”,42,”a”), a lista de assinatura será: texto, número, texto.

Na linguagem “SimpleScript”, o script será pré-processado ao ser lido. Logo, com a base do comando definida, o próximo objetivo é definir o compilador e o interpretador principais. O compilador lerá os arquivos-fonte e os transformará em um código

interno temporário para que sua execução seja mais eficiente pelo interpretador. O interpretador por sua vez processará o código interno diretamente, seguindo as regras definidas da linguagem.

Primeiro, será criado o compilador. Nesta linguagem, ele será relativamente simples, pois apenas temos dois tipos básicos de dados, e não teremos variáveis. O compilador será chamado de "SimpleCompiler".

Usando o formato definido anteriormente para um comando, o compilador processará linha a linha do código-fonte, definindo qual é o comando atual, e quais são seus parâmetros. Em caso de erro, o tipo do comando será marcado como erro. Cada linha gerará uma instância do objeto comando com os dados apropriados.

Em seguida, o compilador inserirá o comando no final de uma lista ligada caso o comando lido seja válido (sem erros). Esta lista ligada definirá a ordem de execução dos comandos. Assim finalizado, o compilador terá gerado uma lista ligada de objetos do tipo comando com os dados lidos do arquivo-fonte.

O segundo passo é criar o interpretador. Dependendo do uso da linguagem, este pode ser mais complexo. Para a SimpleScript, será criado um interpretador simples. Ele apenas executará os comandos em ordem. Os passos que o interpretador executa são:

- Obtém a cabeça da lista ligada dos comandos que se deseja executar.
- Enquanto a lista não for vazia, interpreta o comando atual e passa para o próximo na lista.
- Retorna execução para a *engine*.

Cada vez que executarmos um arquivo de script, ele será executado totalmente. Também se pode ter apenas um script por arquivo, e somente um pode ser lido por vez.

## III. Variáveis

Variáveis são objetos que podem armazenar dados de um tipo definido na memória. São opcionais, mas muito úteis para que o desenvolvedor possa salvar temporariamente dados para uso posterior. Uma linguagem pode ter os seguintes tipos de tipagem de variáveis: Fraca ou Forte e Dinâmica ou Estática.

A tipagem forte implica que cada variável sempre é interpretada de uma mesma forma independendo do seu contexto. Um exemplo disto é a linguagem Java onde não podemos atribuir

um valor de número ponto-flutuante a uma variável do tipo número inteiro (*int*).

A tipagem fraca define que uma dada variável é interpretada diferentemente conforme o contexto. Como exemplo, temos a linguagem C onde podemos somar duas letras (variáveis do tipo caractere, *char*) e atribuir a uma variável do tipo número ponto-flutuante (*float*).

A tipagem estática força que as variáveis a serem utilizadas são declaradas antes do uso. Se uma variável não for definida anteriormente, ocorrerá um erro. As linguagens C e Java seguem este modelo.

A tipagem dinâmica não exige que o desenvolvedor declare as variáveis que for utilizar, logo os tipos das variáveis são definidos dinamicamente (conforme forem utilizadas). As linguagems Ruby e PHP usam este sistema.

No caso da linguagem SimpleScript, ela será de tipagem estática e forte. Mas conforme as necessidades, pode-se utilizar outra combinação de tipagens para melhor se adaptar ao uso pretendido.

Uma variável na linguagem SimpleScript, será basicamente representada por um objeto “variável” que conterá os seguintes campos:

- nome da variável
- id do tipo
- dado da variável

No campo “dado da variável” tem-se o dado referente á variável, do tipo da variável. Por exemplo, se a variável é do tipo texto, o dado da variável será texto.

Os tipos que a variável suportará podem ser definidos conforme a necessidade. Na linguagem de exemplo, apenas serão definidos alguns tipos básicos: texto (tipo string), número inteiro (tipo int), número ponto-flutuante (tipo float) e booleano (tipo bool). O id do tipo de um objeto variável será um destes tipos acima definidos.

Um ponto importante sobre variáveis é o escopo. Escopo basicamente é onde a variável está disponível para acesso. Conforme a linguagem for evoluindo, a idéia de escopo se mostrará necessária. Por agora, apenas teremos variáveis globais.

Variáveis globais são acessíveis em todo o código. Para isto, a linguagem internamente contém uma lista de variáveis que é sempre verificada quando uma variável é requisitada. Esta lista de variáveis é nomeada de lista de variáveis globais.

Os comandos utilizarão uma função de procura de variáveis para localizar as variáveis desejadas. Esta função será chamada de localiza_variável. Neste caso, a função apenas procurará na lista de

variáveis globais (pois apenas elas existem neste ponto da linguagem).

Para a linguagem SimpleScript, serão definidos alguns comandos específicos para o tratamento de variáveis: definevarglobal, setvar e modifyvar.

O definevarglobal cria uma variável conforme o tipo escolhido e tal variável será criada na lista de variáveis globais. Caso a variável exista, ele falha. O segundo parâmetro define qual tipo a variável deve assumir. Caso o tipo indicado não seja suportado, a criação da variável falhará. O setvar muda o valor de uma variável diretamente. O modifyvar executa operações com uma variável.

O comando definevarglobal executa os seguintes passos:

- Verifica se a variável do parâmetro 1 existe (utilizando a função de apoio localiza_variável). Se existir, retorna falha e não cria a variável.
- Verifica o tipo definido pelo parâmetro 2. Se o tipo não for suportado, retorna falha e não cria a variável.
- Cria um objeto do tipo variável, sendo que o nome da variável é o primeiro parâmetro e o tipo de dado é o tipo definido. O dado da varíavel é alocado na memória conforme o tipo escolhido. Caso a alocação falhe, o objeto variável é removido e um erro é retornado.
- O objeto variável é adicionado à lista de variáveis globais.

O comando setvar funciona com este esquema:

- Localiza a variável cujo nome está no primeiro parâmetro. Caso não for encontrada, retorna erro.
- Verifica se o tipo do dado do segundo parâmetro é compatível com o tipo da variável. Se não for, retorna erro.
- Atualiza o dado da variável.

O comando modifyvar é o mais avançado dos três, e segue uma seqüência lógica especial:

- Localiza a variável cujo nome está no primeiro parâmetro. Caso não for encontrada, retorna erro.
- Verifica se o tipo do dado do terceiro parâmetro é compatível com o tipo da variável. Se não for, retorna erro.
- Verifica se a operação definida no segundo parâmetro é válida para o tipo da variável localizada. Se não for, retorna erro.
- Executa a operação sobre a variável.

As operações que o comando modifyvar pode executar podem ser facilmente definidas. Como base, podemos ter as seguintes operações disponíveis para os números: *add* (variável somada com valor), *subtract* (variável subraída do valor), *divide* (variável dividida por valor) e *multiply* (variável multiplicada por valor). Operações adicionais podem ser definidas, como invsub (valor subtraído da variável), pow (variável elevada ao valor), entre outras.

É possível expandir para textos, adicionando operações específicas para texto, como *append* (adiciona valor depois do final do texto) e *prepend* (adiciona valor antes do inicio do texto).

Eis um exemplo dos três comandos executando:

```
...
definevarglobal([v],”int”);
setvar([v],41);
modifyvar([v],”add”,1);
…
```

Neste exemplo, a variável v é criada, definida como tipo int. Em seguida, seu valor é colocado como 41, para depois ser adicionado de 1.

Cada variável quando criada também terá uma propriedade extra: ela conterá um valor base, inicial. Por exemplo, textos conterão um texto de tamanho 0. Números terão o valor nulo (0 para inteiro, e 0.0 para ponto flutuante). Booleanos serão definidos como falso. Esta prática é opcional, mas é adotada para que automaticamente cada variável ao ser criada contenha um valor conhecido.

## IV. Bloco de Comandos

Como foi mostrado no tipo de linguagem anterior, o programador era limitado a apenas executar uma única lista de comandos por arquivo.

Como foi definido na linguagem do tipo “lista de comandos”, o script gerado é salvo em uma lista ligada. Pode-se pensar em criar várias listas: a idéia de existirem vários blocos de *script* em um único arquivo. Para isto precisa-se definir como separar os blocos de *script* entre si.

Na Linguagem SimpleScript, é definido um bloco de *script* como todos os comandos entre uma linha neste formato: [nome do bloco] ou o final do arquivo. Em exemplo do formato:

```
[bloco1]
Comandos
[bloco2]
Comandos
...
[blocoN]
Comandos
```

Agora precisamos alterar alguns aspectos do compilador e do interpretador para ler e interpretar este novo formato. Criamos um novo objeto, o “bloco”. Ele conterá o nome do bloco e a lista ligada com os comandos. Tem-se também uma lista ligada de objetos do tipo “bloco”.

O compilador separará os comandos em blocos seguindo a regra definida acima, gerando a lista de blocos, com cada bloco tendo sua lista de comandos interna e seu nome.

No interpretador, será necessário criar uma função que localiza um bloco, além de que o interpretador agora necessita de um bloco inicial para executar. O nome do bloco inicial pode ser definido pelo desenvolvedor, e no caso da SimpleScript, será escolhido *main*, seguindo a tradição da linguagem C. O formato interno do interpretador continuará o mesmo neste estágio.

Agora o programador pode ter vários blocos de script em um único arquivo, mas pode apenas executar um bloco por vez. Seria o mesmo que utilizar vários arquivos a na versão anteirior da linguagem. Por isto, criamos dois comandos não específicos da linguagem: “runscript” e “jumptoscript”.

Iniciaremos pelo comando “jumptoscript”. Este comando pula a execução do local onde for executado para o início de um outro bloco de script. Ao ser interpretado, ele executa as seguintes funções:

- Verifica se o bloco definido pelo parâmetro (um string) é válido (se está na lista de blocos).
- Se o bloco não for encontrado, retorna (pode retornar um erro).
- Se o bloco for válido, obtém a lista ligada dos comandos.
- Atualiza o estado do interpretador para a cabeça da lista ligada do bloco obtido.

A atualização do estado do interpretador mencionada é simplesmente a mudança do comando atual a ser executado pelo interpretador para um outro comando (no caso acima, para o primeiro ítem de um bloco). Conforme a linguagem for evoluindo,

mais passos poderão ser necessários durante esta atualização de estado.

Ao executar estes passos e retornar ao interpretador, este na próxima iteração executará o primeiro comando do novo bloco. Nenhuma alteração adicional será necessária no interpretador.

O próximo comando a ser implementado será o “runscript”. Diferentemente do jumptoscript, o interpretador não apenas pulará para o início do bloco indicado, mas assim que este bloco terminar, ele retornará ao ponto onde o runscript foi executado. Este comando é mais complicado do que o anterior, e terá consigo um novo conceito: a pilha de retorno. A Pilha de Retorno é nada mais do que uma lista que guarda informações sobre pontos de retorno, ou seja, locais em uma lista de comandos, além de dados extras que se mostrem necessários.

Como a linguagem ainda é simples, uma célula da pilha de retorno apenas conterá um ponto de retorno. Este ponto de retorno para a linguagem SimpleScript é um ponteiro para o objeto comando com tipo “runscript” que foi executado. Definiremos a célula da pilha de retorno como o objeto “ponto_retorno”.

Primeiro, o interpretador será atualizado para suportar a pilha de retorno conforme descrita. Os passos do interpretador atualizado serão:

- Obtém a cabeça da lista ligada dos comandos que se deseja executar.
- Enquanto a pilha de retorno não estiver vazia, obter o topo da pilha (removendo o topo da pilha).
-- Atualizar a lista atual do interpretador com a cabeça extraída da pilha.
-- Enquanto a lista não for vazia, interpreta o comando atual e passa para o próximo na lista.
- Retorna execução para a engine.

O interpretador atualizado agora leva em consideração a pilha de retorno. Para finalizar, precisamos implementar o comando runscript. Ele funciona de forma semelhante ao comando jumptoscript, mas tem uma parte adicional: gravamos o ponto de retorno e colocamos na pilha de retorno:

- Verifica se o bloco definido pelo parâmetro (um string) é válido (se está na lista de blocos).
- Se o bloco não for encontrado, retorna (pode retornar um erro).
- Se o bloco for válido, obtém a lista ligada dos comandos.

- Cria um objeto do tipo “pilha_retorno” e grava o ponteiro da lista atual neste objeto (no caso, este ponteiro apontará para o comando runscript que acabou de ser executado).

- Coloca-se o objeto do tipo “pilha_retorno” no topo da pilha de retorno.

- Atualiza o estado do interpretador para a cabeça da lista ligada do bloco obtido.

Com o comando runscript definido, temos os blocos de scripts implementados. Comandos runscripts trabalham como chamadas de funções de linguagens como C, C++, Java, etc. O motivo de utilizarmos um comando específico para chamada de blocos deve-se ao fato de quando variáveis forem implementadas, podemos usar o valor delas para executar uma chamada de bloco.

Blocos podem também conter variáveis locais (que são limitadas a existirem apenas dentro do bloco). O objeto “bloco” terá adicionalmente uma lista de objetos “variável” que conterá as variáveis locais criadas durante a execução do bloco. Neste caso, é necessário tratá-las.

No caso de uma execução do jumptoscript, é preciso limpar a lista de variáveis locais (no caso de que não se deseja passar para o bloco seguinte as variáveis locais atuais). Como opção na linguagem SimpleScript, variáveis locais apenas existem no bloco que foram criadas.

Para o caso da execução do runscript, não pode-se simplesmente apagar as variáveis locais, pois assim que o bloco retornar, elas poderão ser necessárias. Assim, elas devem ser copiadas no ponto de retorno (e removidas do objeto “bloco” atual). O objeto “pilha_retorno” conterá também uma lista de variáveis locais, copiadas do bloco onde o comando runscript foi lido. Quando o estado do interpretador é atualizado, agora também as variáveis locais serão copiadas de volta.

Exemplo de código onde as variáveis locais serão necessárias após o retorno de um runscript:

```
[blocoX]
definevar([a],”int”);
setvar([a],2);
…
runscript(“blocoY”);
…
setvar([a],42);
…
```

```
[blocoY]
...
```

Com a adição de variáveis locais, tem-se um escopo específico para cada bloco. Por tal fato, a função de procura da variável deve ser alterada para procurar na lista das variáveis locais. A ordem de procura será definida como: local -> global. É mais interessante inicialmente executar a pesquisa para as variáveis locais. Também um novo comando é criado, chamado definevar. Este comando difere do definevarglobal, pois a variável será criada na lista de variáveis locais do bloco atual (ao invés de ser criada na lista de variáveis globais).

## V. Múltiplos Arquivos-Fonte

Mesmo com a implementação de blocos de script, muitas vezes o código gerado pode se tornar longo. Devido a isto, para melhor organização, preferencialmente separamos o código em vários arquivos. Linguagens como C, C++ e Java suportam múltiplos arquivos-fonte.

Uma solução clássica (adotada por compiladores de C, por exemplo) seria indicar quais arquivos-fonte necessitamos. No caso de um script de múltiplos arquivos, isto também funcionaria. Podemos usar esta estratégia para a linguagem de scripts, mas tomaremos um outro rumo a fim de simplificar o uso da linguagem.

Em C, por exemplo, define-se que um arquivo pode apenas ter acesso ás funções de outro arquivo se indicarmos o outro arquivo através de um pré-processador include (#include(arquivo)).

Para simplificar a linguagem, definiremos que todos os blocos têm acesso a todos os outros blocos. Isto elimina a necessidade do uso do #include como acontece na linguagem C.

Mas ainda existe a necessidade de se indicar quais arquivos queremos utilizar. Para resolver este problema, definimos um uso especial para o #include: o arquivo definido no #include será carregado no sistema de scripts.

Algumas modificações são necessárias para este sistema ser implementado. Primeiro, definimos o objeto “arquivo” que conterá uma lista de blocos e um nome. Este objeto será utilizado para guardar o caminho do arquivo-fonte lido e sua lista respectiva de blocos. Teremos também uma lista ligada de objetos do tipo “arquivo”. Esta lista define todo o universo de blocos disponíveis atualmente no sistema de scripts. Nomeamos a lista de Lista Geral de Arquivos.

O compilador precisa detectar e processar os pré-processadores segundo algumas regras. Definem-se regras para os pré-processadores:

- Toda linha iniciada por # será tida como pré-processador.
- Todo pré-processador é apenas válido se estiver antes de qualquer bloco de funções.
- O formato de um pré-processador segue a regra #nome(parâmetro), sendo que (parâmetro) é opcional.

No compilador, primeiro agora geramos uma instância do objeto do tipo “arquivo” e utilizamos a lista ligada de blocos desta instância quando lermos o arquivo. Também o caminho do arquivo é guardado neste objeto.

Definimos uma *flag* (marcador) que indica quando a primeira função foi lida (se for lida, marca como *true*), assim qualquer pré-processador detectado com esta *flag* igual a *true* será ignorado.

Assim que o compilador detecta um pré-processador “include”, ele executará a função de compilação para o arquivo que estiver definido em seu parâmetro. Assim que o compilador ler todos os comandos, o objeto “arquivo” é inserido na Lista Geral de Arquivos.

A função de localização de blocos será alterada para localizar o bloco na Lista Geral de Arquivos, assim podendo localizar qualquer bloco lido e disponível no sistema.

## VI. Funções

Funções podem ser consideradas como uma versão avançada dos blocos de script. Elas contêm algumas funcionalidades adicionais: podem retornar um valor e possuem variáveis de parâmetro. Variáveis de parâmetro são um tipo especial de variáveis locais que são definidas pela função (sem serem criadas por uma execução do comando definevar), e são preenchidas antes da execução de um runscript. O objeto “bloco” será renomeado para “função” e os seguintes campos:

- id to tipo de retorno
- lista de base de variáveis de parâmetro
- lista atual de variáveis de parâmetro
- valor booleano para indicar se a função tem parâmetros

O id to tipo de retorno é especial. O enumerador que o define conterá os tipos de dados que uma função pode retornar,

além to tipo especial *void*. Este tipo indica que a função não poderá retornar nenhum valor. No caso da SimpleScript, uma função poderá retornar: nada (tipo void), número inteiro (tipo int), número ponto-flutuante (tipo float), texto (tipo string) ou booleano (tipo bool).

Para as variáveis de parâmetro, serão usadas duas listas idênticas. Uma (a base) apenas será usada para conter as definições dos parâmetros. A segunda conterá os valores atuais das variáveis de parâmetros quando a função for chamada. O valor booleano extra servirá como otimização para indicar se a função tem ou não variáveis de parâmetro.

Algumas alterações serão necessárias no sistema de variáveis e no interpretador para suportar as funções. Primeiro no sistema de variáveis, a função de localização de variáveis deverá procurar pelas variáveis primeiramente na lista de parâmetros atual. Assim, a função de pesquisa seguirá o seguinte novo formato: parâmetros -> locais -> globais.

No interpretador, o objeto “pilha_retorno” deverá conter as variáveis de parâmetro da função atual, e um booleano indicando se tem ou não parâmetros. A função de atualização do estado também deverá recolocar as variáveis de parâmetro (da mesma maneira que as variáveis locais).

Isto definido, é necessário a inserção de alguns comandos na linguagem para o funcionamento de funções. O primeiro comando a ser inserido é o prepareparameters. Este comando prepará os parâmetros da função que for chamada através do comando runscript. Sem ela, a função quando for executada, terá seus parâmetros definidos como o valor básico dos tipos. O comando populará uma série de listas especiais (semelhantes às listas definidas nos comandos) com os valores que forem utilizados como seus parâmetros. Estas listas especiais seguem o formato:

- Lista com a assinatura dos parâmetros (define os tipos e a ordem em que os parâmetros foram lidos).
- Lista de parâmetros do tipo texto (string).
- Lista de parâmetros do tipo número.

O comando runscript será alterado para fazer uso dos valores preparados (das listas) pelo comando prepareparameters. Caso a função contenha parâmetros, o comando runscript deve inicialmente gerar a lista atual de parâmetros (copiando da lista base). A lista assim gerada, o comando deve copiará os dados das listas especiais para a nova lista de parâmetros (texto em variável de texto, números em variáveis numéricas, etc) seguindo a lista de assinaturas. Em caso de erro (número de parâmetros menor, a assinatura é diferente do requerido), a linguagem pode tomar duas decisões:

- ignorar e executar a função, emitindo um aviso no sistema de log.
- emitir uma mensagem de erro e fechar o programa.

No 1º caso, as variáveis não alteradas terão o valor base. Neste caso, é garantida a execução ininterrupta do programa, mas pode causar problemas no futuro. Este é o sistema que será adotado na linguagem SimpleScript.

## VII. Controle de Fluxo

Controle de fluxo é uma das partes mais importantes de qualquer linguagem. É a habilidade de ajustar a maneira como um programa realiza suas tarefas. Sem nenhum controle de fluxo, o sistema é sempre forçado a executar uma mesma sequência de comandos, não importando nenhuma condição ou estado do sistema. É interessante uma linguagem ter controle de fluxo, pois assim o sistema pode reagir diferentemente conforme certas condições.

Um bom exemplo é ter um inimigo qualquer que faça suas ações de ataque através de um script. Sem nenhum controle de fluxo, ele é sempre forçado a executar o mesmo ataque independentemente de sua condição. Supomos que o desenvolvedor queira que o inimigo use alguma habilidade de cura no caso que ele esteja com pouca vida, além do ataque simples. Como não se tem controle do fluxo do script, isso é impossível. Controle de fluxo define a possibilidade de tomada de decisão.

No caso da linguagem SimpleScript, mesmo com funções e variáveis, o sistema sempre é forcado a seguir uma linha única de execução. Para resolver esta limitação, é necessário implementar um comando que proporcione um controle de fluxo, dando à linguagem uma certa opção de decisão. Este comando especial é o *if* (se). Ele terá o formato: if(“condição”,”verdadeiro”,”falso”);. Os 3 parâmetros são:

- condição: será um comando que retornará verdadeiro ou falso para o sistema.
- verdadeiro: comando a ser executado caso a condição retorne verdadeiro.
- falso: comando a ser executado caso a condição retorne falso.

Este novo comando contém um sistema diferente dos demais anteriormente definidos. Seus parâmetros são comandos

que precisam ser identificados e executados. Para isto, assim que o comando if é executado, identificamos o comando presente no parâmetro condição e o executamos. Conforme sua resposta, é escolhido qual dos dois comandos a frente serão identificados e executados.

A identificação é feita por uma função especial disponível no pré-processador (a mesma que é utilizada para identificar cada comando durante a leitura do código fonte).

Existem outros tipos de controles de fluxo como o *while* e *for*. É possível simular o comando while se forem criados dois comandos: *reset* e *return*. O comando *reset* não tem parâmetros, e tem a funçao de retornar ao primeiro comando do bloco ou função atual. O comando *return* executa o mesmo que o interpretador faz quando chega ao final de um bloco ou função: retorna para o último ponto da pilha de retorno (ou não faz nada caso não exista nenhum ponto de retorno). Eis um exemplo:

```
[exemplowhile]
if(“cond”,”return();”,”null()”);
…
reset();
```

O *for* não é útil ainda pois a linguagem não suporta vetores (uma lista seqüencial de variáveis do mesmo tipo) ou algum tipo de lista iterável.

Um comando que pode ser implementado é o comando *returnv*. É uma variação do comando *return* que possibilita o retorno de valores em funções. Possui um parâmetro do tipo variável que é usado para definir o dado a ser retornado. Ao ser executado, este comando verifica se o dado colocado é do mesmo tipo do retorno da função, e se for, copia o dado diretamente em uma variável especial disponível no interpretador (chamada de variável dado do retorno). Em seguida, executa a mesma rotina de retorno do comando return.

Para obter o dado depois, é utilizado o comando *getreturn*. Este comando lê a variável dado do retorno, e copia a mesma para a variável que for definida em seu parâmetro (fazendo as verificações de tipo necessárias). Eis um exemplo:

```
[geral]
definevar([result],”int”);
setvar([result],0);
prepareparameters(10,50);
runscript(“add”);
getreturn([result]);
```

```
[add](int)(int,[a],int,[b])
definevar([r],"int");
setvar([r],[a]);
modifyvar([r],"add",[b]);
returnv([r]);
```

## VIII. Structs

*Structs*, ou estruturas, são pacotes de variáveis (que podem ser de tipos diferentes). São definidas por um nome específico e podem ser criadas como uma variável. São úteis quando é necessário guardar vários dados conectados sem a necessidade de criar várias variáveis soltas (assim melhorando a organização). Um exemplo para ilustrar o fato é:

Existe um objeto no jogo que deseja-se guardar informações pelo sistema de script e este objeto tem três dados básicos: nome, posição X e posição Y. Para guardar estas informações, é necessário três variáveis, duas do tipo inteiro e uma do tipo texto, que serão criadas separadamente:

```
[objetoA]
definevarglobal([objA_nome],"string");
definevarglobal([objA_X],"int");
definevarglobal([objA_Y],"int");
```

Se existirem mais objetos, cada um deve ser criado desta forma, o que seria cada vez mais complicado e desorganizado. A solução para este problema de organização é o uso das *structs*. O objeto "estrutura" é simples, e é definido como:

- uma lista de variáveis
- um nome (identificando o tipo)

Diferentemente de uma variável comum, inicialmente é preciso que a *struct*, e sua base, sejam declaradas e definidas antes que possam ser usadas. Para isto, precisa-se definir como que será a declaração de uma *struct*. Para a linguagem SimpleScript, uma *struct* será definida seguindo este formato:

```
struct[nome](tipo,[nome1],...,tipo,[nomeN])
```

Esta definição deve apenas aparecer antes de qualquer definição de função ou bloco para ser válida. Cada objeto "estrutura"

criado será colocado em uma lista de definições de estruturas presente no interpretador.

O próximo passo é proporcionar ao sistema de script como utilizar as estruturas declaradas. Por simplicidade, estruturas apenas poderão ser criadas globalmente, logo apenas será necessário alterar o comando *definevarglobal*.

O objeto "variável" conterá um novo tipo de dado, o tipo "*struct*". Nos dados suportados na variável, também teremos um objeto "estrutura".

O comando *definevarglobal*, ao invés de falhar quando o tipo da variável não for um dos básicos, executará uma pesquisa na lista de estruturas para determinar se o tipo é uma estrutura ou não.

Caso o tipo for uma das estruturas declaradas, a variável a ser criada toma o tipo "*struct*" e a declaração da estrutura localizada é copiada para o objeto "estrutura" definido no objeto "variável". Assim, a variável agora é uma estrutura.

Com a variável agora disponível, é necessário que os dados da estrutura sejam acessíveis e alteráveis. Para isto, são criados alguns comandos específicos: getstructmember e setstructmember.

O comando getstructmember é um comando de três parâmetros:

```
getstructmember([estrutura],"campo",[variavel]);
```

O primeiro parâmetro é a variável que se deseja obter o valor de um dos membros. Deve ser uma estrutura. O segundo parâmetro é o campo o qual se deseja obter o dado. É o nome de uma das variáveis declaradas para a estrutura. O último parâmetro é a variável onde deseja-se guardar o dado obtido. Deve ser do mesmo tipo do dado da variável da estrutura que deseja-se ler. O comando executará os seguintes passos:

- procurar pela variável definida no primeiro parâmetro.
- se a variável existir, verificar se o tipo é *struct*.
- se o tipo for *struct*, verificar se contém o campo definido no segundo parâmetro.
- se o campo existir, verificar se o tipo do campo é do mesmo tipo da variável do terceiro parâmetro.
- se os tipos forem iguais, copiar o dado do campo para a variável do terceiro parâmetro.

O comando setstructmember também possui três parâmetros, mas difere do getstructmember no último:

```
setstructmember([estrutura],"campo",dado);
```

Os dois primeiros parâmetros são idênticos ao do comando getstructmember. O terceiro conterá apenas o dado que deseja-se copiar para um dos membros da estrutura. Os passos executados pelo setstructmember são similares ao do getstructmember:

- procurar pela variável definida no primeiro parâmetro.
- se a variável existir, verificar se o tipo é *struct*.
- se o tipo for *struct*, verificar se contém o campo definido no segundo parâmetro.
- se o campo existir, verificar se o tipo do campo é do mesmo tipo do dado do terceiro parâmetro.
- se os tipos forem iguais, copiar o dado do terceiro parâmetro no campo da estrutura.

Uma limitação presente nesta versão da linguagem SimpleScript é que não é possível ter-se uma estrutura como campo de outra estrutura.

## IX. Compilador

O compilador tem como objetivo transformar o código-fonte em um formato específico (pré-processado) para o interpretador com a finalidade de melhorar a performance da linguagem. O compilador pode identificar um comando a partir de um texto, separar os parâmetros dos comandos e verificar erros de script que possam estar presentes.

O modo como o compilador trata os erros depende da decisão do programador. Erros podem ser apenas declarados e a compilação continuar, ou a compilação pode ser forçada a parar e o programa fechado.

Na linguagem SimpleScript, o compilador executa os seguintes passos a obter um arquivo-fonte (em pseudo-código):

```
achoufuncao = falso;
funcaoatual = nernhuma;
para ( cada linha do arquivo fonte ) {
        se ( iniciar com "[" ) {
                achoufuncao = verdadeiro;
                le formato da função;
                se ( formato da função valido) {
                        cria nova função f;
                        funcaoatual = f;
                        insere na lista de funções;
                }
                senao erro = formato de funcao invalido;

        }
```

```
        senão {
            se ( primeiro caractere for # ) {
                se (achoufuncao) {
                    erro = declaracao de
                        pre-processador dentro de
                        funcao;
                }
                senao {
                    le formato de pre-processador;
                    executa pre-processador;
                }

            }
            senão se ( primeiros caracteres forem struct) {
                se (achoufuncao) {
                    erro = declaracao de struct dentro
                        de funcao;
                }
                senao {
                    le formato da struct;
                    se (formato valido) {
                        cria nova struct;
                        insere na lista de structs;
                    }
                    senao {
                        erro = formato errado na
                            funcao;
                    }
                }
            }
            senão {
                le formato do comando;
                se (formato invalido) {
                    erro = fomato de comando  invalido;
                }
                identifica comando;
                se (identificacao falhou) {
                    erro = comando invalido;
                }
                gera comando;
                insere na funcaoatual;
            }
        }
}
```

## X. Interpretador

O interpretador é a peça mais importante da linguagem. Ele processa os comandos em ordem e os executa, além de controlar a pilha de retorno e as variáveis. Ele é dividido em duas partes básicas:

- a função do *loop* de interpretação
- a função de execução dos comandos

A função de *loop* executa os comandos em ordem e a cada comando executa a função de execução dos comandos (para que cada comando seja processado e executado). Esta função executa os seguintes passos:

```
se (função selecionada atual for nula) retornar;
comando atual = cabeça da lista de comandos da função atual;
enquanto (verdade) {
        se (comando atual for inválido) {
                se (pilha de retorno estiver vazia) retornar;
                senão {
                        obter topo da pilha;
                        atualizar estado do interpretador
                                segundo dados do topo da pilha;
                }
        }
        senão {
                executar comando atual;
                avançar para o próximo comando;
        }
}
```

Dependo to objetivo da linguagem, é possível estender este sistema para incluir funcionalidades extras como retornar o controle à *engine* antes do término da linha de execução (por exemplo, um comando *wait* (para esperar X segundos) ou um comando *waitresponse* (para esperar uma ação do usuário, por exemplo)).

Outros comandos podem também parar a execução de uma lista de comandos no meio, não apenas os dois citados. Pode-se ter um comando que espere uma animação terminar ou uma colisão acontecer. Neste caso, o código pode ter alterado desta forma:

```
se (função selecionada atual for nula) retornar;
se (não deseja_retornar) comando atual = cabeça da lista de comandos da
                                                função atual;
deseja_retornar = falso;
enquanto (verdade) {
        se (comando atual for inválido) {
                se (pilha de retorno estiver vazia) retornar;
                senão {
                        obter topo da pilha;
                        atualizar estado do interpretador
                                segundo dados do topo da pilha;
                }
        }
        senão {
                executar comando atual;
                avançar para o próximo comando;
                se (deseja_retornar) retorna;
        }
}
```

Neste caso, um tratamento especial deve ser feito para que a espera seja executada da maneira desejada. Uma opção é um booleano que não deixe o sistema entrar no *loop* da linguagem (e que seja apenas desabilitado quando a espera terminar).

O interpretador também contém em seu objeto certas funções de ajuda para a criação das funções de execução dos comandos. Estas funções servem para obter um certo parâmetro, e verificar sai existência. Caso não possam obter o dado desejado (requisitamos um parâmetro numérico, mas pela assinatura, temos um parâmetro de texto, por exemplo), ele retorna um erro. Os comandos de obtenção de parâmetros avançam as listas de seus respectivos tipos automaticamente. Alguns destes comandos são:

- reverter todas as listas de parâmetros para a cabeça, incluindo a lista de assinaturas (parametros_cabeca).
- obtém um parâmetro texto de um objeto comando (obtem_texto).
- obtém um parâmetro numérico de um objeto comando (obtem_numero).
- obtém um parâmetro de variável de um objeto comando (obtem_variavel).
- avança na lista de assinatura de parâmetros de um comando (avanca_assinatura).

Como exemplo, será usado o comando *runscript* (necessita de um parâmetro de texto). Ele é um comando simples, que apenas utiliza um parâmetro. O pseudocódigo da função de execução dele será assim (para a identificação do parâmetro):

```
parametros_cabeca;
par = obtem_texto;
avanca_assinatura;
se (falha) retorna;
```

Todos os comandos seguem esta mesma idéia. Eles usam as funções de apoio do interpretador para ler os parâmetros necessários, e com eles obtidos e válidos, o comando é executado. Se novos tipos de parâmetros forem adicionados, é possível inserir novas funções de apoio para os parâmetros. Eis um exemplo do comando definevar:

```
parametros_cabeca;
var = obtem_variavel;
avanca_assinatura;
se (falha) retorna;
```

```
tipo = obtem_texto;
avanca_assinatura;
se (falha) retorna;
```

Outros comandos de apoio também estão disponíveis, como o localiza_variavel (que localiza uma variável no escopo atual, usando globais, locais e parâmetros).

# D. Extras

## I. Compilação

A cada leitura do código-fonte, o script é recompilado internamente para depois ser executado. Conforme o tamanho do script cresce, mais lenta é a esta operação. Como o compilador interno da linguagem já compila o script, uma opção é possível. Pode-se gerar um arquivo já pré-compilado para melhorar a performance de leitura, além de proteger o código-fonte.

Mesmo o compilador interno da linguagem sendo integrado diretamente na linguagem (pois ele apenas serviu até agora para melhorar a performance), é possível usá-lo para gerar um código compilado externo para uso posterior. O problema é definir o que precisa ser salvo no arquivo compilado para que o sistema possa ser recriado quando o arquivo compilado for lido. Conforme as funcionalidades definidas anteriormente, o arquivo compilado deve conter:

- todas as funções lidas (e sua estrutura)
- definições de estruturas

Conforme mais funcionalidades forem adicionadas, talvez seja necessário incrementar os dados salvos no arquivo compilado. Eis um formato simples de arquivo compilado para a linguagem SimpleScript:

Salvando a Linguagem
- salvar 3 chars, SSC
- salvar 1 número de ponto flutuante, define versão
- salvar inteiro indicando número de arquivos
- salvar arquivos (repetir para cada arquivo)
- salvar número de estruturas definidas
- salvar estruturas (repetir para cada estrutura)

Salvando Arquivos
- salvar número de funções
- salvar funções (repetir para cada função)

Salvando Funções
- salvar número de comandos
- salvar comandos (repetir para casa comando)
- salvar parâmetros (nome e tipo)
- salvar tipo de retorno

Salvando Comandos

- salvar id do comando
- salvar número de parâmetros totais
- salvar lista de assinatura de parâmetros
- salvar número de parâmetros de texto
- salvar lista de parâmetros de texto
- salvar número de parâmetros numéricos
- salvar lista de parâmetros numéricos
- salvar número de parâmetros de tipo variável
- salvar lista de parâmetros do tipo variável (apenas nome)

## II. Debug

Como qualquer linguagem de script, muitas vezes o desenvolvedor de depara com erros. Comandos escritos errado, variáveis com valores incorretos. Conforme o código cresce, erros podem acontecer, causando problemas no resultado desejado.

Para facilitar a localização destes erros, muitas vezes existem ferramentas de debug, que serve para mostrar o estado atual do sistema num determinado momento. Para uma linguagem de scripts também é possível, e recomendado, disponibilizar algum meio de executar o debug do código caso seja necessário.

Como primeira ferramenta de debug, temos que cada comando envie uma mensagem do que está fazendo assim que executado. Cada comando envia ao *log* uma mensagem específica (por exemplo, o comando *setvar* enviaria a mensagem “Colocado valor XXX na variável YYY”). O envio dessas mensagens pode ser ativado e desativado através de um valor *booleano*.

Outra ferramenta interessante é o comando *printvar*, que envia ao log uma mensagem contendo o nome e valor da variável colocada em seu parâmetro. Pode ser incluído na mensagem tipo da variável (local, global ou parâmetro) se for necessário.

## E. Conclusão

O desenvolvimento de uma linguagem de script é um processo longo e árduo. Utilizar uma solução pronta (uma linguagem como Lua) ou programas de ajuda (como flex e javacc) seria um processo simples e rápido do que desenvolver uma linguagem do zero.

Todo o processo necessitou de um bom conhecimento de algumas estruturas de dados, além de um entendimento de como uma linguagem funciona (variáveis, pilha de retorno, entre outros) para que o sistema funcionasse corretamente.

O desenvolvimento de uma linguagem requer também uma linguagem rápida e eficiente, além de ser interessante utilizar uma linguagem orientada a objetos (pois cada parte da linguagem pode ser facilmente representada por objetos).

Outro ponto interessante é também que o formato da linguagem que foi definido precisou ser bem pensado para que possa ser facilmente implementado e que seja simples para que a linguagem não fique complicada demais quando utilizada. Isso parcialmente se reflete internamente em como as estruturas internas foram definidas, especialmente no compilador.

# F. Parte Subjetiva

## I. Desafios e Frustrações

Durante a criação da linguagem, um dos principais desafios é definir o formato da linguagem para que este seja simples de ser utilizado e implementado. Decisões erradas podem gerar uma linguagem muito complicada de ser utilizada, e até gerar um compilador que seja mais complicado de ser implementado. E se não for bem pensado, o formato pode dificultar ainda a implementação de extensões de funcionalidade que possam no futuro ser necessárias.

Outro desafio foi desenvolver o funcionamento do interpretador para que este seja eficiente quando estiver executando o comando, mas possa ser flexível a ponto de suportar certas funcionalidades.

## II.Disciplinas Relevantes

As seguintes matérias são relevantes ao trabalho:

- MAC 122 – Princípios de Desenvolvimento de Algoritmos: muito importante, pois o desenvolvimento de uma linguagem necessita de vários algoritmos básicos, como os de pesquisa.
- MAC 323 – Estrutura de Dados: várias estruturas de dados são essenciais para a criação de uma linguagem, como listas ligadas e filas de prioridade.
- MAC 211 – Laboratório de Programação II: conceitos básicos de linguagens.
- MAC 338 – Análise de Algoritmos; especialmente importante pois muitos algoritmos utilizados na linguagem podem ser melhorados, assim subindo a performance da linguagem em diversos casos. Uma linguagem deve ser rápida, e a análise de como seu algoritmo interno funciona é fundamental para que possamos melhorá-lo.
- MAC 316 – Conceitos de Linguagens de Programação: boa base para a teoria de linguagens, especialmente de escopo. Muitas funcionalidades ficam melhor entendidas, como funções e escopo.

## III. Futuro

Pretendo estender a linguagem de forma a incluir funcionalidades adicionais como classes e utilizar a linguagem em uma *engine* de jogos.

## G. Referências


I. http://pt.wikipedia.org/wiki/Motor_de_jogo
II. http://www.dm.ufscar.br/~waldeck/curso/java/part26.html
III. Alex Varanese. Game Scripting Mastery, 2003.