Referências Bibliográficas

[Ada 1995] International Organization for Standardization. **Ada 95 Reference Manual. The Language. The Standard Libraries**, Jan. 1995. ANSI/ISO/IEC-8652:1995. 2.2

[Beazley 1996] BEAZLEY, D. M.. **SWIG: an easy to use tool for integrating scripting languages with C and C++**. In: Association, U., editor, 4TH ANNUAL TCL/TK WORKSHOP '96, p. 129–139, Berkeley, CA, USA, July 1996. USENIX. 2.5

[Benton 1999] BENTON, N.; KENNEDY, A.. **Interlanguage working without tears: blending SML with Java**. In: ICFP '99: PROCEEDINGS OF THE FOURTH ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, p. 126–137, New York, NY, USA, 1999. ACM Press. 2.2

[Box 2002] BOX, D.; SELLS, C.. **Essential .NET, Volume I: The Common Language Runtime**. Addison-Wesley, Boston, MA, USA, 2002. 2.2

[Chapman 1997] CHAPMAN, B.; HAINES, M.; MEHROTRA, P.; ZIMA, H. ; ROSENDALE, J. V.. **Opus: A coordination language for multidisciplinary applications**. Scientific Programming, 6(4):345–362, Winter 1997. 2.3

[Collin 1997] COLLIN, S.; COLNET, D. ; ZENDRA, O.. **Type inference for late binding: The SmallEiffel compiler**. In: JOINT MODULAR LANGUAGES CONFERENCE, JMLC'97, volumen 1204 de **Lecture Notes in Computer Sciences**, p. 67–81. Springer Verlag, 1997. 2.5

[Conway 1958] CONWAY, M. E.. **Proposal for an UNCOL**. Communications of the ACM, 1(10):5–8, 1958. 2.4

[Dvorak 2005] DVORAK, Z.. **Gimplification improvements**. In: GCC DEVELOPERS' SUMMIT, p. 47–56, Ottawa, Canada, June 2005. 2.4

[ECMA 2005] Ecma International. **C++/CLI Language Specification**, Dec. 2005. Standard ECMA-372. 2.2

[Ewing 2006] EWING, G.. **Pyrex - a language for writing Python extension modules**, 2006. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`. 2.5

[Finne 1998] FINNE, S.; LEIJEN, D.; MEIJER, E. ; JONES, S. P.. **H/Direct: a binary foreign language interface for Haskell**. In: ICFP '98: PROCEEDINGS OF THE THIRD ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, p. 153–162, New York, NY, USA, 1998. ACM Press. 1

[Gelernter 1985] GELERNTER, D.. **Generative communication in Linda**. ACM Transactions on Programming Languages and Systems, 7(1):80–112, 1985. 2.3

[Gosling 2000] GOSLING, J.; JOY, B.; STEELE, G. ; BRACHA, G.. **The Java Language Specification**. Addison-Wesley, Boston, MA, USA, 2nd edition, 2000. 1.1

[Hugunin 1997] HUGUNIN, J.. **Python and Java - the best of both worlds**. In: PROCEEDINGS OF THE 6TH INTERNATIONAL PYTHON CONFERENCE, p. 11–20, San Jose, CA, USA, Oct. 1997. 2.2

[ISO 2006] International Organization for Standardization. **C# Language Specification**, June 2006. ISO/IEC 23270:2003. 2.2

[Ierusalimschy 2006] IERUSALIMSCHY, R.. **Programming in Lua**. Lua.org, 2nd edition, Mar. 2006. 1.1, 5

[Jones 1993] JONES, S. P.; HALL, C. V.; HAMMOND, K.; PARTAIN, W. ; WADLER, P.. **The Glasgow Haskell Compiler: a technical overview**. In: PROC. UK JOINT FRAMEWORK FOR INFORMATION TECHNOLOGY (JFIT) TECHNICAL CONFERENCE, p. 249–257, Keele, Staffordshire, UK, Mar. 1993. 2.1

[Jones 1999] JONES, S. L. P.; RAMSEY, N. ; REIG, F.. **C--: A portable assembly language that supports garbage collection**. In: PPDP '99: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE PPDP'99 ON PRINCIPLES AND PRACTICE OF DECLARATIVE PROGRAMMING, p. 1–28, London, UK, 1999. Springer-Verlag. 2.4

[KDE 2006] KDE.ORG. **KDE developer's corner - Language bindings**, Oct. 2006. `http://developer.kde.org/language-bindings/`. 2.5

[Liang 1999] LIANG, S.. **Java Native Interface: Programmer's Guide and Reference**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 2.5, 3.1.3

[Lindholm 1999] LINDHOLM, T.; YELLIN, F.. **The Java Virtual Machine Specification**. Addison-Wesley, 2^{nd} edition, April 1999. 2.2

[MacEachern 2006] MACEACHERN, D.; ORWANT, J.. **perlembed(1)**. Perl 5 Porters, 5.8.8 edition, Jan. 2006. `http://perldoc.perl.org/perlembed.html`. 3.4.5, 4.1.2

[Manzur 2006] MANZUR, A.; CELES, W.. **toLua++ reference manual**, Apr. 2006. `http://www.codenix.com/~tolua/tolua++.html`. 2.5

[Marquess 2006] MARQUESS, P.. **perlcall(1)**. Perl 5 Porters, 5.8.8 edition, Jan. 2006. `http://perldoc.perl.org/perlcall.html`. 3.1.5

[Metzner 1979] METZNER, J. R.. **A graded bibliography on macro systems and extensible languages**. SIGPLAN Not., 14(1):57–64, 1979. 2.4

[Moura 2004] DE MOURA, A. L.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Coroutines in Lua**. Journal of Universal Computer Science, 10(7):910–925, July 2004. 2.5

[Niemeyer 2006] NIEMEYER, G.. **Lunatic Python**, July 2006. `http://labix.org/lunatic-python`. 2.4

[OMG 2002] Object Management Group, Inc., Framingham, MA, USA. **The Common Object Request Broker: Architecture and Specification, Version 3.0**, July 2002. 2.3

[Okamoto 2006b] OKAMOTO, J.. **perlguts(1)**. Perl 5 Porters, 5.8.8 edition, Jan. 2006. `http://perldoc.perl.org/perlguts.html`. 3.1.5

[Okamoto 2006a] OKAMOTO, J.; ROEHRICH, D.. **perlapi(1)**. Perl 5 Porters, 5.8.8 edition, Jan. 2006. `http://perldoc.perl.org/perlapi.html`. 3.4.5, 4.2.3

[Ousterhout 1990] OUSTERHOUT, J. K.. **Tcl: An embeddable command language**. In: PROCEEDINGS OF THE USENIX WINTER 1990 TECHNICAL CONFERENCE, p. 133–146, Berkeley, CA, 1990. USENIX Association. 2.6.2

[Ousterhout 1994] OUSTERHOUT, J. K.. **Tcl and the Tk Toolkit**. Addison Wesley, 1994. 2.6.2

[Ousterhout 1998] OUSTERHOUT, J. K.. **Scripting: Higher-level programming for the 21st century**. IEEE Computer, 31(3):23–30, 1998. 1, 2.6

[Peterson 2001] PETERSON, P.; MARTINS, J. R. R. A. ; ALONSO, J. J.. **Fortran to Python interface generator with an application to aerospace engineering**. In: PROCEEDINGS OF THE 9TH INTERNATIONAL PYTHON CONFERENCE, Long Beach, CA, USA, Mar. 2001. 2.4

[Randal 2004] RANDAL, A.; SUGALSKI, D. ; TOETSCH, L.. **Perl 6 and Parrot Essentials**. O'Reilly Media, Inc., 2nd edition, 2004. 2.2

[Roehrich 2006] ROEHRICH, D.. **perlxs(1)**. Perl 5 Porters, 5.8.8 edition, Jan. 2006. `http://perldoc.perl.org/perlxs.html`. 19

[Stepanian 2005] STEPANIAN, L.; BROWN, A. D.; KIELSTRA, A.; KOBLENTS, G. ; STOODLEY, K.. **Inlining Java native calls at runtime**. In: VEE '05: PROCEEDINGS OF THE 1ST ACM/USENIX INTERNATIONAL CONFERENCE ON VIRTUAL EXECUTION ENVIRONMENTS, p. 121–131, New York, NY, USA, 2005. ACM Press. 3.1.3

[Sun 2003] Sun Microsystems. **Java Native Interface 5.0 Specification**, 5.0 edition, 2003. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/`. 3.2.3

[Syme 2006] SYME, D.; MARGETSON, J.. **Microsoft F#**, 2006. `http://research.microsoft.com/fsharp/`. 2.2

[Tarditi 1992] TARDITI, D.; LEE, P. ; ACHARYA, A.. **No assembly required: compiling standard ML to C**. ACM Lett. Program. Lang. Syst., 1(2):161–177, 1992. 2.1

[Thomas 2002] THOMAS, R.. **Lua Technical Note 4 - a thin API for interlanguage working, or Lua in four easy calls**, Aug. 2002. `http://www.lua.org/notes/ltn004.html`. 4.1.2

[Thomas 2004] THOMAS, D.; HUNT, A.. **Programming Ruby: The Pragmatic Programmer's Guide**. Addison Wesley Longman, Inc., Boston, MA, USA, 2nd edition, 2004. 1.1, 3.2.2, 5

[Tolmach 1998] TOLMACH, A. P.; OLIVA, D.. **From ML to Ada: Strongly-typed language interoperability via source translation.** Journal of Functional Programming, 8(4):367–412, 1998. 2.1

[Wall 2000] WALL, L.; CHRISTIANSEN, T. ; ORWANT, J.. **Programming Perl**. O'Reilly, 3rd edition, July 2000. 1.1

[Welch 1995] WELCH, B. B.; JONES, K. ; HOBBS, J.. **Practical programming in Tcl and Tk**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995. 2.6.2

[Wilson 1992] WILSON, P. R.. **Uniprocessor garbage collection techniques**. In: PROC. INT. WORKSHOP ON MEMORY MANAGEMENT, número 637 em Lecture Notes in Computer Sciences, p. 1–42, Saint-Malo, France, 1992. Springer-Verlag. 3.2.2

[van Rossum 1999] VAN ROSSUM, G.. **Python/C API Reference Manual**. Corporation for National Research Initiatives (CNRI), Reston, VA, USA, 1.5.2 edition, Apr. 1999. `http://www.python.org/doc/1.5.2/api/api.html`. 4

[van Rossum 2006b] VAN ROSSUM, G.. **Python Reference Manual**. Python Software Foundation, 2.4.3 edition, Mar. 2006. `http://docs.python.org/ref/`. 1.1, 4.1.2

[van Rossum 2006a] VAN ROSSUM, G.. **Extending and Embedding the Python Interpreter**, 2.4.3 edition, March 2006. `http://docs.python.org/ext/ext.html`. 3.1.1, 3.2.1, 4.1.2

[van Rossum 2006c] VAN ROSSUM, G.. **Python/C API Reference Manual**. Python Software Foundation, 2.4.3 edition, Mar. 2006. `http://docs.python.org/api/api.html`. 3.1.1, 4

A
API de LibScript

A.1
Inicialização e Término

- `script_env* script_init(const char* namespace)`
  Inicializa LibScript e retorna um ponteiro para o ambiente virtual. O parâmetro `namespace` indica o nome a ser usado nas estruturas a serem criadas no espaço de nomes das máquinas virtuais para representar o ambiente virtual.
- `void script_done(script_env* env)`
  Encerra o ambiente virtual.

A.2
Registro de Funções

- `typedef script_err (*script_fn)(script_env*)`
  Tipo das funções C a serem registradas no ambiente virtual. Ao expor uma API existente para LibScript, a função tipicamente será uma função *wrapper* que carrega os parâmetros de entrada do ambiente, chama uma função do programa e envia os parâmetros de saída de volta ao ambiente.
- `script_err script_new_function(script_env* env, script_fn fn, const char* name)`
  Registra uma função no ambiente virtual.

A.3
Buffer de parâmetros

- `double script_get_double(script_env* env, int index)`
  `int script_get_int(script_env* env, int index)`
  `int script_get_bool(script_env* env, int index)`
  `const char* script_get_string(script_env* env, int index)`
  Obtêm dados do buffer. Estas funções devem ser chamadas ao início das funções *wrapper*. Para cada parâmetro de entrada, uma chamada deve ser realizada. Ao fim, pode-se invocar a macro `SCRIPT_CHECK_INPUTS(env)`,

que encerra a função retornando um código de erro caso alguma leitura com alguma destas funções não tenha encontrado um dado do tipo esperado (A API não realiza conversões automáticas entre strings e números). Em `script_get_string`, a string retornada pertence ao chamador, que passa a ser responsável por desalocá-la.

– `script_type script_get_type(script_env* env, int index)`
`int script_buffer_len(script_env* env)`
Estas funções permitem escrever funções em C que realizam verificação de tipo e número de parâmetros em tempo de execução. A função `script_get_type` obtém o tipo do elemento do buffer solicitado e `script_buffer_len` retorna o número de parâmetros no buffer.

– `void script_put_double(script_env* env, int index, double value)`
`void script_put_int(script_env* env, int index, int value)`
`void script_put_bool(script_env* env, int index, int value)`
`void script_put_string(script_env* env, int index, const char* value)`
Inserem dados no buffer. Ao final de uma função, os valores de retorno devem ser passados com chamadas a estas funções e um código de erro `SCRIPT_OK` como retorno da função C.

– `void script_reset_buffer(script_env* env)`
Esvazia o buffer.

A.4 Executando Código

– `script_err script_run(script_env* env, const char* language, const char* code)`
Executa uma string de código em uma dada linguagem. Se necessário, o plugin apropriado é carregado e inicializado.

– `script_err script_run_file(script_env* env, const char* filename)`
Função de conveniência; carrega o texto de um arquivo e o executa com `script_run`. A linguagem é detectada a partir da extensão do arquivo.

– `script_err script_call(script_env* env, const char* fn)`
Requisita a execução de uma função em algum dos plugins cadastrados. Os parâmetros de entrada devem ser passados anteriormente com chamadas às funções `script_put_*`; valores de retorno podem ser obtidos com `script_get_*`. Inicialmente, a tabela de funções C do ambiente virtual

é consultada. Não havendo uma função definida em C, os plugins são consultados na seqüência em que foram inicializados implicitamente via `script_run` ou `script_run_file`: funções registradas na representação do ambiente virtual definido para a LibScript na máquina virtual da linguagem (isto é, no nome criado com `script_init`) são acessíveis via `script_call`.

– `script_err script_error(script_env* env)`
`const char* script_error_message(script_env* env)`
`void script_set_error_message(script_env* env, const char* message)`
Obtém o código e a mensagem de erro mais recentes do ambiente. Após uma chamada a script_error, o código de erro é zerado de volta para `SCRIPT_OK`. A mensagem de erro, por sua vez, não é zerada. A função `script_set_error_message` define um novo valor para a mensagem de erro do ambiente. Permite ao plugin propagar à aplicação as mensagens de erro da máquina virtual.

– `const char* script_get_namespace(script_env* env)`
Retorna o nome do namespace registrado com `script_init`.

A.5 API Exportada por Plugins

As chamadas aos plugins que implementam interfaces com as várias máquinas virtuais são realizadas internamente pela biblioteca principal, que espera encontrar as seguintes funções:

– `script_plugin_state script_plugin_init_`*linguagem*`(script_env* env)`
Responsável por inicializar o plugin. Durante a inicialização, o espaço de nomes do ambiente virtual deve ser exposto à máquina virtual de alguma forma apropriada para a linguagem (como uma tabela em Lua, ou um módulo em Python, ou ainda uma classe em Ruby). A rotina de inicialização pode retornar um handle que será passado de volta a ele nas chamadas subseqüentes. O estado da máquina virtual e o ponteiro para o ambiente LibScript devem ser armazenados de modo a ser posteriormente acessíveis a partir deste handle.

– `script_err script_plugin_run_`*linguagem*`(script_plugin_state st, char* text)`
Envia código para execução na máquina virtual. Esta função é utilizada internamente por `script_run` e `script_run_file`. Deve retornar

`SCRIPT_OK` em caso de sucesso, `SCRIPT_ERRLANGCOMP` para erros de compilação ou `SCRIPT_ERRLANGRUN` para erros de execução, preferencialmente definindo uma mensagem de erro com `script_set_error_message`.

– `script_err script_plugin_call_`*linguagem*`(script_plugin_state st, char* fn)`
Realiza a chamada de uma função que tenha sido definida nativamente no espaço de nomes do ambiente na máquina virtual do plugin. Ao chamar uma função no espaço de nomes, seja em C através de `script_call` ou executando código em algum dos plugins, LibScript irá utilizar esta função para tentar executar a função no contexto do plugin. Se a função não foi definida no plugin, o valor `SCRIPT_ERRFNUNDEF` deve ser retornado. Caso contrário, ela deve ser executada, com parâmetros de entrada obtidos através de `script_get_*` e valores de retorno enviados com `script_put_*`, e os valores `SCRIPT_OK` ou `SCRIPT_ERRLANGRUN` devem ser retornados, conforme apropriado.

– `void script_plugin_done_`*linguagem*`(script_plugin_state st)`
Responsável pelo encerramento do ambiente.