Volume 6, Issue 5, May 2016 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

An Implementation of a Knowledge-based Programming Language with Dynamic Grammar

Jignesh V. Smart
G H Patel P G Department of Computer Science and Technology, Sardar Patel University,
Vallabh Vidyanagar, Gujarat, India

Abstract— There are so many programming languages, each designed to satisfy particular requirements. Duplication of work is rampant due to the difficulty of understanding and adapting existing code. Also, it is difficult to interface code developed using one software platform with code developed on another. The programming language called "One" has been designed to tackle these issues using a knowledgebase approach. One is designed to be an umbrella language that can be used to write computer programs in a more natural way and have them translated into code for any software development platform using appropriate knowledgebase. The language is fully extensible. While the grammar of most programming languages in common use is static, One language's knowledge-based approach allows a programmer to define one's own syntax. It also supports designing syntax that mimics any natural language. This paper provides a brief overview of the language and then describes its implementation.

Keywords— programming language, knowledgebase, programming language implementation, compiler, byte code, interpreter

I. MOTIVATION FOR THE WORK

Thousands of programming languages have been developed in the past decades [1]. Each is designed to satisfy particular requirements and has its own strengths and weaknesses. While code reuse remains an ideal, duplication of work is rampant due to the difficulty of understanding and adapting existing code. This results in high cost of development, delays in software projects and bugs. Also, often there is a requirement to interface code developed using one software platform with code developed on another. Such interfacing has never been easy. Various software development paradigms [2] have been developed to address these and other issues. Just to mention a few, programming languages like C [3], C++ [4], LISP [6], Prolog [12], Simula [13], F# [14], Self [15], Groovy [16], Ruby [17], Go [18], Dart [19], Haskell [20], Scala [21], Swift [22] have been developed that implement or can be used for implementing programming paradigms like logic programming [5], list programming [6], structured programming [7], object-oriented programming [8], service-oriented architecture [9], aspect-oriented programming [10], design by contract [11], etc. But there is a substantial scope for improvement. The programming language "One" has been designed to tackle these issues using a knowledgebase approach.

II. SALIENT FEATURES OF THE ONE LANGUAGE

- One is designed to be an umbrella language that can be used to write computer programs in a more natural way and have them translated into code for any software development platform using appropriate knowledgebase. Thus, One serves as a very high level programming language tier above the existing programming languages
- The language is fully extensible. While the grammar of most programming languages in common use is static, One language's knowledge-based approach allows a programmer to define one's own syntax. It also supports designing syntax that mimics any natural language, with mechanisms to handle ambiguity and context to substantial extent
- The One language does not have a fixed syntax or a fixed set of capabilities. New syntax and functionalities can be added by providing a knowledgebase (written in the One language itself) that can translate the code either into some existing programming language or into One code that uses the existing knowledgebases. Just like ready-made code libraries, One programmers can pick and choose knowledgebases as per their choice of syntax and requirement of functionality
- The One language aims to support natural-language-like syntax. To this end, it has support for taking advantage of the context of a programming language element as also for detecting and resolving ambiguities. It supports multiple meanings and shades of meaning (interpretation) for the same language element like word. It also has conflict management
- One programmers are expected to write their code in One, get the code translated to a high-level language of their choice using a translation knowledgebase and then execute the code using the facilities available for that language. The language supports multiple target languages. This is illustrated in Fig. 1

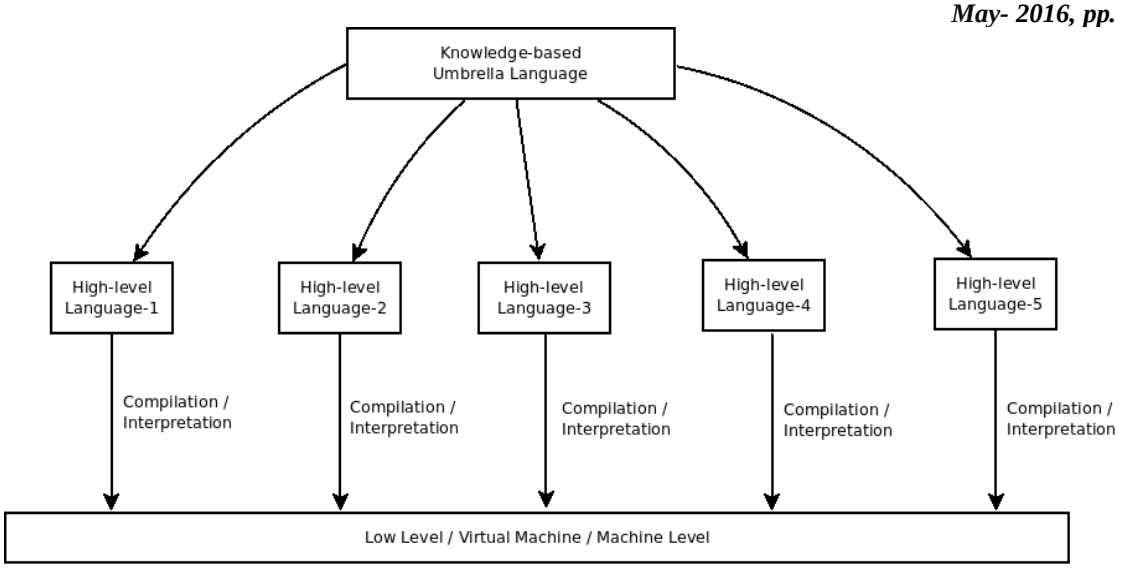

Fig. 1 The umbrella language concept

III. IMPLEMENTATION OVERVIEW

Currently, the One language has been implemented in Java. Fig. 2 provides an overview of the implementation. The major elements of the implementation are as follows:

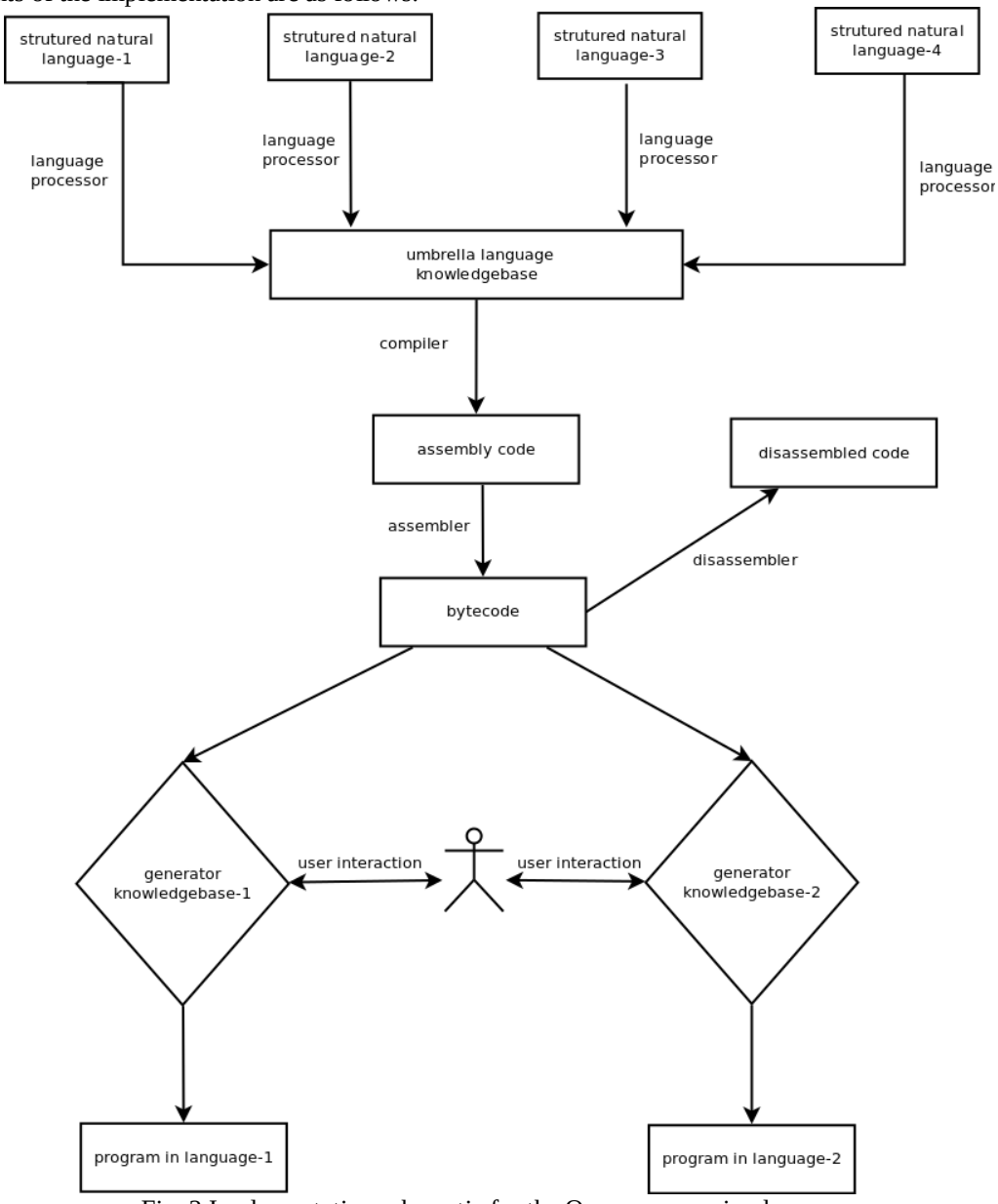

Fig. 2 Implementation schematic for the One programming language

- One is designed for collaborative development. It is expected that programmers around the world will create and maintain knowledgebases defining "structured natural language" syntax for the core as well as additional features of the One language. A structured natural language syntax resembles natural language syntax, but avoids the pitfalls of natural language processing. All structured natural languages are treated as dialects of the common One language, even when they differ drastically from one another. These knowledgebases may be called language processors for their respective languages. They are required to generate code in a common, standard One language syntax. User interaction may be required at this stage for disambiguation, context resolution, etc.
- This standard syntax is then compiled into the One assembly language, which serves as the intermediate representation for all One programs. This language follows many traditions for intermediate representations and ASTs (Abstract Syntax Trees).
- The assembly code is then converted into byte code. There is also provision of converting the byte code back to assembly code for understanding.
- The bytecode is translated into a high-level language program of the programmer's choice by using corresponding generator knowledgebase, if it is available. User interaction may be required at this stage for additional information needed by the generator or to elicit code generation options, if any, from the user.

IV. IMPORTANT CLASSES IN THE IMPLEMENTATION

While the Java implementation of the schematic in Fig. 2 consists of a large number of classes, some of the important classes are described below:

- **LexerenUS.java** This class is automatically generated using the lexical analyser generator tool flex from the lexical specification of the structured English dialect of One. It contains the scanner for the language, which scans a program in the language and returns tokens (words, punctuation, etc.) from it to the parser
- **Parser.java** This class defines the parser for the One language. Since the language does not have a fixed syntax, it is not possible to define static context-free grammar for it [23] [24]. Also most tools or algorithms commonly employed for parsing do not work for a dynamic grammar. Hence, a top-down parsing algorithm is designed to operate according to a dynamically changing list of parser rules. The set of parser rules is different for the One assembly language as well as different dialects of the One language
- **ParserRule.java** This class represents a parser rule for the parser. A parser rule consists of the syntax rule and the associated action to be taken when the rule matches. Thus, it can be seen that a Syntax-Directed Translation (SDT) approach is followed
- **LexerAssembly.java** This class contains the lexer for the One assembly language generated by the flex tool from the lexical specification of the One assembly language
- **Assembler.java** This class implements the assembler for the One assembly language
- **Disassembler.java** This class implements the disassembler for the byte code generated from the One assembly language
- **Execution.java** This class contains the code necessary to execute the byte code
- **VirtualMachine.java** This class contains the implementation of a virtual machine environment for executing the byte code
- **Memory.java** This class represents the simulated main memory of the virtual machine
- **Heap.java** This class represents the heap of the currently running code
- **Type.java** This class specifies the notion of data type in One
- **Symbol.java** This class defines the notion of a symbol in the One language
- **SymbolTable.java** This class defines a symbol table
- **KnowledgeBase.java** Objects of this class represent a knowledgebase
- **MountTable.java** A One program may make use of multiple knowledgebases, just like a high-level language program may make use of multiple libraries. The knowledgebases may be mounted and unmounted dynamically at run time, similar to the mounting and unmounting of file systems by the operating system. A knowledgebase must be mounted before it can be used. The mount table contains a table of currently mounted knowlegebases
- **Loader.java** This class is used to mount a knowledgebase by loading it from a file
- **SystemKnowledgeBase.java** This class contains a system knowledgebase representing the core infrastructure of the One language. The system knowledgebase is mounted automatically and cannot be unmounted

V. CONCLUSIONS

The One programming language is a knowledgebase-oriented programming language designed to sit on top of existing high-level languages. It makes programming easier and enables much greater reuse of code. It even permits the programmer to write programs in different dialects that resemble structured natural languages. It also enables the programmer to write a program in one very high level language and then to generate code for a variety of high-level programming languages and software development frameworks. This paper describes an implementation of the language.

ACKNOWLEDGEMENT

The work described in this paper is an extension of the work carried out in the research project "Design and Implementation of a New Programming Language" under the UGC Major Research Project scheme of the University Grants Commission, India between the years 2002 and 2005.

REFERENCES

[1] B. Kinnersley. (2016, May) *The language list*. [Online]. Available: http://people.ku.edu/nkinners/LangList/Extras/langlist.htm

[2] (2016) Programming Paradigm. [Online]. Available: https://en.wikipedia.org/wiki/Programming_paradigm

[3] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint, *The C programming language*. Prentice-Hall Englewood Cliffs, 1988, vol. 2.

[4] B. Stroustrup, *The C++ programming language*. Pearson Education, 2013.

[5] A. Church, "A set of postulates for the foundation of logic," *Annals of mathematics*, pp. 346–366, 1932.

[6] J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham et al., *LISP I Programmers Manual*, 1960.

[7] E. W. Dijkstra, *"Notes on structured programming,"* 1970. [Online]. Available: http://www.informatik.uni-bremen.de/agbkb/lehre/programmiersprachen/artikel/EWD-notes-structured.pdf

[8] D. Mandrioli and B. Meyer, Eds., *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1991, ch. Design by Contract, pp. 1–50.

[9] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, and R. Metz, *Reference Model for Service Oriented Architecture 1.0*, OASIS Std., Rev. 1.0, Oct 2006. [Online]. Available: http://docs.oasis-open.org/soa-rm/v1.0/

[10] G. Kiczales, J. Lamping, C. Lopes, J. Hugunin, E. Hilsdale, and C. Boyapati, *"Aspect-oriented programming,"* Oct. 15 2002, uS Patent 6,467,086. [Online]. Available: https://www.google.com/patents/US6467086

[11] B. Meyer, "Applying design by contract," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992. [Online]. Available: http://dx.doi.org/10.1109/2.161279

[12] A. Colmerauer and P. Roussel, "The birth of prolog," in *The Second ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL-II. New York, NY, USA: ACM, 1993, pp. 37–52. [Online]. Available: http://doi.acm.org/10.1145/154766.155362

[13] J. R. Holmevik, "Compiling simula: A historical study of technological genesis", *Annals of the History of Computing*, IEEE, vol. 16, no. 4, pp. 25–37, 1994.

[14] S.Somasegar, *"F# - a functional programming language,"* Oct 2007. [Online]. Available: http://blogs.msdn.com/b/somasegar/archive/2007/10/17/f-a-functional-programming-language.aspx

[15] D. Ungar and R. B. Smith, "Self," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 9–1–9–50. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238853

[16] V. Subramaniam, *Programming Groovy 2: Dynamic Productivity for the Java Developer*, D. H. Steinberg, Ed., 2008.

[17] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby 1.9: The Pragmatic Programmers' Guide*, 3rd ed. Pragmatic Bookshelf, Apr 2009.

[18] F. Schmager, N. Cameron, and J. Noble, "Gohotdraw: evaluating the go programming language with design patterns," in *Evaluation and Usability of Programming Languages and Tools*. ACM, 2010, p. 10.

[19] G. Bracha. (2011, 11) *A walk on the dart side: A quick tour of dart (transcript of the talk)*. Stanford University. [Online]. Available: http://blog.sethladd.com/2011/11/transcription-of-quick-tour-of-dart-by.html

[20] S. Thompson, *Haskell: the Craft of Functional Programming*, 3rd ed. Addison-Wesley, Jun 2011.

[21] J. D. Suereth, *Scala in Depth*. Manning Publications Co, 2012.

[22] Various, *The Swift Programming Language*. Apple Inc., Jun 2014.

[23] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[24] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

