JOURNAL OF HUMANITIES AND LEADERSHIP STUDIES: JOURNAL OF FACULTY OF ARTS, DENNIS OSADEBAY UNIVERSITY, ASABA, DELTA STATE
Volume 1, Number 2, November, 2025, ISSN: 2971-5741

The Interface of English and Computing Languages: An Innovative Union

Joyce Emojorho

Department of English and Literary Studies, Faculty of Arts
Dennis Osadebay University, Anwai Road, Asaba, Delta State
(J*oyce.emojorho@dou.edu.ng*)

Abstract

This paper examines the functional, pedagogical, and semiotic relationship between English (a natural language) and computing languages (formal systems). The work responds to a persistent misconception: that the presence of English lexical items in programming implies a linguistic merger between natural and formal languages. Using representative examples from Python, SQL, HTML, and contemporary natural language processing (NLP) pipelines, and drawing on Saussurean and Peircean semiotics as well as contemporary semiotic thought, the study demonstrates that English primarily serves as a human-centered interpretive bridge, improving readability, learnability, and human–computer interaction, while programming languages retain formal autonomy. A qualitative, document and example-based methodology is used to analyze lexical, syntactic, and semantic relationships. Key findings show that English improves accessibility but does not alter computational syntax or machine-level semantics. Visual and non-English programming paradigms evidence the language-agnostic character of computation. NLP models computationally represent English for task performance without merging English into programming syntax. Implications touch pedagogy, multilingual computing, AI-assisted programming, and the semiotic theory of digital artefacts.

Keywords: English Language, Programming Languages, Semiotics, NLP, Human–Computer Interaction, Readability

Introduction

Digital computation has transformed communication, culture, and cognition; at its core sits an interplay between human natural languages and machine-oriented formal languages. English; owing to its global reach in science, business, and technology, frequently appears as the surface vocabulary (keywords, identifiers, documentation) of many mainstream programming languages. At the visual level a novice might read if, while, print, or select as recognizably English-like tokens; yet the computational reality of these tokens is governed by formal grammars, parsers, and execution semantics rather than by natural-language meaning.

This study interrogates the nature of that intersection. Is the presence of English lexical items evidence of a linguistic integration, or is it a pragmatic affordance to humans? The working hypothesis is the latter: English functions as a human-centred accessibility tool rather than becoming part of the programming language's formal semantics. To substantiate this claim the paper combines examples from popular programming languages, evidence from visual and non-English programming systems, and insights from NLP research that treats English as data to be modeled rather than as a fused hybrid with programming syntax.

Because the relationship is both functional and semiotic, the research draws on structuralist and semiotic theory to clarify how signs operate across human and machine interpreters. The study thereby contributes to theoretical clarity and practical recommendations for teaching, multilingual computing, and AI-driven programming interfaces.

Literature Review

Programming languages as formal systems and readability concerns

Programming languages are artificial systems created to specify computational procedures; they are implemented through compilers or interpreters and characterized by precise syntactic and semantic rules. While many high-level languages use English-like keywords to increase human readability, the computational effect of these tokens is strictly operational and unambiguous within the language grammar. The mainstream description of programming languages highlights their engineered nature and the distinction between human-facing syntax and machine-executable semantics.

Empirical work on vocabulary and naming in code also suggests linguistic patterns e.g. Zipf-like distributions in identifier usage, yet these observations are about human practices (naming, convention) rather than about computation requiring English. Studies on readability and "vocabulary" in code argue that lexical choices influence comprehension, maintenance, and collaboration. Readability metrics form part of modern language design discussions.

Visual programming environments (e.g., Scratch and Blockly) and regionally localized programming tools illustrate that computation does not inherently depend on English vocabulary. Block-based systems convert drag-and-drop constructs into underlying formal representations; their success in teaching computational thinking underscores that symbolic structure and control flow can be represented without linear English text. Likewise, programming languages and environments have been localized or created with non-English keywords, demonstrating that the choice of surface forms is a matter of human usability, not computational necessity.

Advances in NLP, particularly with Transformer-based architectures like BERT, treat natural language (including English) as input data to be represented, encoded, and transformed. Landmark models (BERT, GPT-family) achieve sophisticated performance on tasks by learning statistical and structural patterns, but crucially they do so as formal systems that operate on tokens and vectors, not by merging programming syntax with natural language grammar. The literature on NLP for programming (program synthesis, code generation, program understanding) further clarifies that "natural-language programming" is a layer of interface and specification: systems translate human descriptions into formal code representations via models and synthesis techniques. Surveys in NLP for programming map the space of tasks (code generation, summarization, translation between natural language and code) and emphasize that the relationship is one of representation and transformation rather than linguistic fusion.

Semiotics offers a conceptual vocabulary to understand how signs operate across human and machine interpreters. Saussure's signifier-signified duality foregrounds arbitrariness and the socially constituted nature of signs; read in this light, programming tokens have significance only within the formal system's conventions and the developer community's interpretive practices. Peirce's triadic sign; object, representamen, interpretant, helps chart how code functions: a token (representamen) refers to an operation or value (object) and is processed by the machine (interpretant) while also being read by human developers (secondary interpretants). Contemporary semioticians (e.g., Eco, Barthes) extend these insights to artefact-centred and cultural readings of technological sign systems. Applying these frameworks illuminates why English keywords serve as human-oriented signifiers without altering machine-internal referential semantics.

Theoretical Framework

The analysis draws on three interlocking theoretical lenses; Saussurean Structuralism, Peircean Semiotics and Contemporary semiotic and HCI thought (Eco & Barthes). Saussurean Structuralism will aim to distinguish between form (signifier) and meaning (signified) across natural and formal languages, highlighting the social nature of linguistic signs and the arbitrary relation between signifier and signified. Peircean Semiotics model code as triadic signs where the machine, as interpretant, enacts an operational meaning separate from human interpretants. This helps to illustrate how the same token participates in distinct sign processes (machine execution vs. human comprehension). Contemporary semiotic and HCI thought (Eco & Barthes) is to situate code within cultural and interpretive practices and to underscore the role of readability and usercentered design in choosing surface forms. These approaches emphasize that signs acquire layered meanings depending on context and interpretive communities.

Together these frameworks make it possible to track sign-level differences (syntactic tokens) versus interpretive-level functions (readability, pedagogy, interface design).

Methodology

A qualitative, interpretive methodology was adopted. The study selected representative programming constructs; small, readable examples from Python, SQL, and HTML because they are commonly cited as instances where English-like keywords are visible. The research also sampled evidence from visual programming (Scratch, Blockly) and from the NLP literature (BERT, program-synthesis surveys) to triangulate findings.

Data sources included canonical language documentation and descriptive overviews (programming language definitions and modern-language analyses); official project pages and descriptions for educational/visual languages (Scratch, Blockly); foundational NLP papers (BERT) and recent surveys on NLP for programming and program synthesis; semiotic and theoretical texts (Saussure, Peirce, Eco) as framing evidence.

Analytic Steps

Lexical sampling: extract typical keyword examples (e.g., if, while, print, SELECT) and analyze their syntactic role.

Comparative analysis: compare textual code with block-based examples and with localized/non-English language variants.

Semiotic mapping: apply Saussurean and Peircean categories to each example, tracking signifier–signified relations and triadic sign functions.

Cross-domain synthesis: integrate findings with current NLP research on program synthesis and code modeling.

Analysis and Discussion

English as a readability and pedagogical affordance

Considering the canonical Python snippet:

```
if temperature > 30:

print ("Warning: High Temperature")
```

Here, *if* and *print* are English-derived tokens that signal conditional branching and output. From a human-readability perspective they leverage English vocabulary to lower cognitive friction for learners and maintainers. Nevertheless, the Python interpreter treats if as a reserved token defined by Python's grammar; its semantics are mechanically specified and executed irrespective of any English semantics beyond convention. This illustrates lexical borrowing (if, print) that facilitates human comprehension. The keywords function according to Python syntax; their semantic meaning is entirely formalized. The pattern recurs in SQL:

```
SELECT name, age FROM students WHERE age > 18;
```

SELECT and WHERE are English words that serve to structure human-readable queries; the database engine parses these tokens according to SQL's formal syntax and semantics. The presence of English here optimizes communication among developers, DBAs, and technical documentation writers, but the computational rule set is independent of English meaning. This demonstrate that English words clarify logic for humans without influencing the formal system. These observations align with scholarly descriptions of programming languages as engineered formal systems designed for precise execution.

```
class Student:

def __init__(self, name, age):

self.name = name

self.age = age

student = Student("Jane", 22)

print(student.name)
```

The tokens class, def, and print are English-like and semantically suggestive, helping learners relate the structure to natural-language concepts such as "define" and "print." The lexical familiarity improves comprehension and aids in teaching object-oriented design concepts without altering the underlying computational execution.

Visual and localized programming: proving language-agnostic computation

Visual and non-English programming languages confirm this functional distinction. Scratch and Blockly remove textual reliance, while region-specific programming languages demonstrate that English is not essential for computational logic.

Educational platforms such as Scratch and Google's Blockly have succeeded precisely because they separate the symbolic structure of computation from linear English text. Blocks capture control flow, state, and events via shape and composition; a program assembled in blocks can be translated into textual code in multiple surface languages. This demonstrates that the computational structure (the program's abstract syntax tree, execution semantics) is independent of the surface lexical choices developers or learners make. The practical pedagogical success of these platforms supports the claim that English is an accessibility medium rather than a computational necessity.

Natural Language Processing: modeling English, not merging it with code

Transformers and large pre-trained language models (e.g., BERT) show how English can be represented and manipulated by statistical and neural systems for tasks ranging from classification to generation. BERT's architecture and training demonstrate that natural-language processing is a formal procedure operating on tokenized inputs, embeddings, and learned transformations; it represents English patterns in vector spaces for downstream tasks. In the domain of code, program synthesis research builds models that translate natural-language descriptions into code or vice versa, but these processes are translational rather than integrative: the models map between two formal representations (a human-text representation and a programming-language representation). Surveys of NLP for programming detail this mapping, noting open challenges such as specification ambiguity, semantics-preserving generation, and cross-lingual code understanding.

in NLP-based programming and AI-assisted code generation, English functions as data to be modeled, not merged into program syntax. For example, consider a Python snippet;

```
summarizer = pipeline("summarization")

text = "English readability enhances programming pedagogy."

summary = summarizer(text, max_length=20)

print(summary)
```

The string "English readability enhances programming pedagogy." is in natural language. Tokens in English guide the model to perform summarization but do not influence Python syntax. English serves a representational role, enhancing human interpretation of the program's purpose while the computation remains language-agnostic.

Semiotic mapping: dual audiences, dual interpretants

From a semiotic perspective, programming languages are symbolic systems. Peircean semiotics identifies symbols (if, print) representing operations (object) interpreted by the machine (interpretant). Saussurean Structuralism highlights that the signifier–signified relationship is arbitrary within the formal system, with English aiding human interpretation rather than shaping computation.

Applying Saussure and Peirce clarifies why English tokens occupy a hybrid semiotic space. Saussure's emphasis on social convention explains why if or select function as meaningful signifiers for developer communities: their social meaning is constituted by usage and documentation. Peirce's triadic model, by contrast, helps separate machine execution from human interpretation: the same token participates in a machine-level interpretant (program execution) and a human-level interpretant (comprehension, maintenance). Eco and Barthes remind us that cultural and rhetorical dimensions—documentation style, naming conventions, comment practices—imbue code with additional interpretive layers that affect collaboration, pedagogy, and the socio-technical life of software.

Multilingual and multicultural implications

The dominance of English in many mainstream tools raises equity and access concerns. Localization efforts and non-English programming initiatives show feasible alternatives for non-Anglophone learners and practitioners. For global collaboration, it is crucial that tooling, documentation, and educational resources be multilingual. Moreover, AI-driven interfaces (natural-language programming assistants and code-completion tools) must account for linguistic diversity to avoid further entrenching English-centric workflows. Current advances in NLP for programming point to opportunities and challenges in multilingual code modeling and localized developer tooling.

In NLP, English is computationally modeled. BERT and GPT process English to generate meaningful outputs, demonstrating functional integration. The language is the object of computation, not a merged linguistic system. This underscores that English serves human accessibility and interpretive guidance, while formal structures govern computational operations.

Findings

The study's qualitative analysis yields the following principal findings:

1. English enhances readability and comprehension in programming and NLP but does not integrate into syntax or semantics. Surface English improves human readability and learning but does not determine machine-level syntax or semantics. The lexical resemblance to English is a design choice for human usability, not a linguistic integration into the computational substrate.

2. Programming languages are formally autonomous. Their grammars, parsing rules, type systems, and runtime semantics are specified independently of natural-language meaning; replacement of surface keywords with non-English tokens or block-based representations preserves computational behavior.

3. NLP treats English as representational data. Transformer-based models (e.g., BERT) and program-synthesis systems process English via tokenization, embeddings, and learned mappings to/from code; these are computational transformations, not syntactic fusions.

4. Semiotic theory clarifies dual interpretants. Saussurean and Peircean frameworks show that tokens play distinct semiotic roles for humans and machines; the dual interpretant model explains why human-facing lexical choices facilitate collaboration and pedagogy without changing execution semantics.

5. Pedagogical and equity implications are substantial. Localization, block-based platforms, and AI-assisted code tools provide avenues to broaden access; however, current research in multilingual code modeling and no-code interfaces must be accelerated to reduce English-centric barriers.

Conclusion and Implications

The interaction between English and computing languages is best characterized as a pragmatic semiotic union oriented toward human interpretability rather than as a linguistic merger. English functions as an interpretive scaffold that improves readability, learnability, and collaboration; computation itself relies on formally specified grammars and semantics that are language-agnostic. For pedagogy, the findings recommend continued use of English-like keywords where appropriate for readability, paired with concerted efforts to develop multilingual resources, block-based introductory environments, and interfaces that reduce language barriers. For AI and programming tool design, the results suggest prioritizing models and interfaces that treat natural language as a distinct representational layer subject to translation and formalization rather than as part of program syntax. For semiotics and critical theory, the paper offers an analytical vocabulary for examining how code functions as a cultural artefact with layered meanings.

Recommendations for Future Research

1. Empirical readability studies that quantify how surface lexical choices affect comprehension across linguistic backgrounds.

2. Longitudinal studies of multilingual codebases and the impact of localization on collaboration.

3. Interdisciplinary investigations combining semiotics, HCI, and machine learning to design inclusive natural-language programming interfaces.

4. Ethnographic research into naming conventions, code comments, and documentation across developer communities to better understand socio-semiotic practices.

References

Saussure, F. de. (2011). *Course in General Linguistics* (H. Baskin, Trans.). Columbia University Press.

Thurlow, C., & Mroczek, K. (Eds.). (2011). *Digital Discourse: Language in the New Media.* Oxford University Press.

Buchanan, R. (2019). "Interaction Design and Semiotics." Design Issues, 35(3), 30–43.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of NAACL-HLT 2019*.

Zhu, Q., et al. (2024). A Survey on Natural Language Processing for Programming. *LREC 2024 Proceedings*.

Shin, J., et al. (2021). A Survey of Automatic Code Generation from Natural Language. *Journal/Conference Survey (2021)*.

"Programming language." *Wikipedia*. Overview of programming language concepts and readability considerations.

Chaudhary, P. (2025). Modern programming languages: features and readability. *IACIS 2025*.

Blockly — Google Developers. Blockly documentation and educational role.

Peirce, C. S. (1998). *The Essential Peirce: Selected Philosophical Writings*. Indiana University Press.