Idiolect: A Reconfigurable Voice Coding Assistant

Breandan Considine
McGill University
bre@ndan.co

Nicholas Albion
Independent Developer
nalbion@yahoo.com

Xujie Si
University of Toronto
six@cs.utoronto.ca

***Abstract*—This paper presents Idiolect, an open source [1] IDE plugin for voice coding and a novel approach to building bots that allows for users to define custom commands on-the-fly. Unlike traditional chatbots, Idiolect does not pretend to be an omniscient virtual assistant but rather a reconfigurable voice programming system that empowers users to create their own commands and actions dynamically, without rebuilding or restarting the application. We offer an experience report describing the tool itself, illustrate some example use cases, and reflect on several lessons learned during the tool's development.**

***Index Terms*—speech recognition, voice programming, bots**

I. Introduction

Humans are able to quickly learn new words and phrases, and apply them in a variety of contexts. Chatbots, however, are often limited to a set of static commands and phrases. This creates a frustrating experience for users, who struggle to express their intent, as well as bot developers, who must anticipate user intent and make new capabilities discoverable. This rigidity is a common source of misalignment between user intent, author expectations and bot capabilities.

Voice coding allows users to quickly dictate new commands and behaviors. For example, one might say, "whenever I say *open sesame*, do the following action", thereafter, the system will perform the desired action when so instructed. Or, "whenever I say *redo thrice*, repeat the last action three times". Or, e.g., invoke a function in a scripting language, open a file, or perform other manually repetitive chores.

Idiolect provides a default lexicon of phrases, but does not impose them upon end-users. Instead, users may override the defaults with custom voice commands on-the-fly, which are incorporated into the lexicon without delay. By shifting the burden of adaptation onto the system, this frees users to express their intent in a more natural manner.

Primarily, Idiolect observes the following design principles: be (1) natural to use, (2) easy to configure, (3) as unobtrusive as possible. We believe that these principles are important for a system intended to be used by developers, who are busy people and capable of configuring the system themselves. We also support developers with visual and motor impairments, who may have difficulty typing, or prefer to use a voice interface.

In the following paper, we describe Idiolect, an IDE plugin originally developed at JetBrains in 2015 and recently updated to support many new features, including deep speech recognition and dynamically-reconfigurable voice commands, letting users verbally or programmatically express their desired intent.

[1] https://github.com/OpenASR/idiolect

II. Prior Work

Mary Shaw, during her 2022 SPLASH keynote [1] called for programming languages to address the needs of "vernacular developers". Jin Guo has also talked about the need for programming in "ordinary people's language". We take their proposals quite literally to mean that computers should be able to interpret spoken programs, and not just written ones.

Bots are essentially a realization of the once scorned [2], but now increasingly plausible idea natural language programming. Taken to its logical conclusion, the most natural form of expressing a program is therefor the human voice.

Early attempts to build voice programming systems can be traced back at least twenty years to Leopold and Amber's [3] (1997) work on keyboardless programming, later revisited by Arnold and Goldthwaite [4] (2001), Begel and Graham [5] (2005) and others. These systems allow users to write code by speaking into a microphone, however early voice programming systems were limited by a small vocabulary, and do not consider IDE integration or reconfigurability.

Another stream of work has explored teachable voice assistants and user-defined phrases (Chkroun & Azaria [6]). Though similar to our own, their work targets general-purpose voice user interfaces and does not directly consider voice programming. It also predates most of the recent progress on large language modeling, which we consider to be a transformative enabling technology for speech programming.

III. Speech Recognition

Today, automatic speech recognition (ASR), the translation of an audio waveform containing speech to text, is essentially a solved problem – one of the many pretrained ASR models would work well enough for our purposes. However, we also require realtime offline speech recognition capabilities built on an open source deep speech pipeline, which has only recently become possible for users running on commodity hardware.

Idiolect integrates with Vosk, a state-of-the-art deep speech system with realtime models for various languages, which provides an open source Java API. Initially, we distributed a default model for the English language inside the plugin, but at the request of JetBrains to reduce bandwidth, we instead prompt the user to select and download a pretrained model from the Vosk website upon first installing the plugin.

Users may optionally configure a built-in TTS voice from the host operating system and a cloud-based speech recognition or synthesis service, with the caveat that web speech requires uninterrupted internet connectivity and introduces an

Fig. 1. Idiolect's architectural overview.

additional 300-500ms of overhead latency depending on the user's proximity to the datacenter and other load factors.

Once a spoken utterance is decoded as text, Idiolect must determine the relevant actions and entities needed to fulfill the user's intent. Furthermore, it may need to consider the IDE context, i.e., the current editor state and command history, to resolve potentially ambiguous commands. For example, the command, "open plugin menu" could refer to multiple different menus, depending on when and how it was invoked.

The IntelliJ Platform has over 10^3 possible actions. These actions are bound to keyboard shortcuts, menu items, and toolbar buttons. The user can also bind a voice command directly to an action, presuming the user already knows the action's identifier. Idiolect's default grammar was manually curated from the IDE action list, using the CamelCase identifier to generate a suitable description for intent recognition.

In keeping with the principle of configurability, we allow users to define custom grammars and bind utterances to actions using a context-free grammar. By providing an extensible DSL, users can bind their own command patterns using either a simple configuration file, or programmatically via the plugin API to handle more complex usage scenarios.

Idiolect dispatches utterances to a series of recognizers using a turn-based priority queue, in which each recognizer is given a single turn to match or pass on each utterance. Once a command is recognized, the command is consumed and dispatched to no subsequent recognizer thereafter. This prevents the single command from triggering multiple actions.

The plugin first attempts to resolve an utterance using an exact lexical match against a finite language, i.e., a lexicon of predefined commands. Highest priority are those which control the plugin itself, enabling and disabling speech recognition.

User-defined commands are the next highest priority. These can be matched lexically, or more generally as a sentence in a context-free language. Nonterminal parameters allow one to define more complex instructions, such as "open the ⟨filename⟩[in ⟨project_name⟩]" or "jump to the ⟨nth⟩line".

The recognizer of last resort is a large language model. Open-ended commands which are unrecognizable to all of the previous methods are dispatched to a service that matches the intent against a set of available actions using a prompt, e.g., "Out of these actions: ..., which one most likely fulfills the command ...?", whose response is then dynamically invoked.

IV. Barriers and Pathways to Usability

In our experience as plugin users and maintainers, usability challenges typically arise in a few key areas. In the following section, we will discuss some of those obstacles encountered while developing Idiolect and our efforts to overcome them.

A. User Onboarding

Several users reported confusion when first installing our plugin. To address this issue, we added a wizard that guides users through installation. Upon first installing the plugin, a user is greeted and prompted to download the Vosk model for recognizing their preferred natural language, defaulting to the system locale. Once unpacked, the model is stored in the plugin configuration directory, `/.idiolect`. The user is next prompted to configure the properties file, and bind a few voice commands. This is a one-time process, and users may reconfigure the plugin via the settings menu at any time.

B. Plugin Observability

Failures arise in many stages of the intent recognition pipeline. Short of fixing the error directly, often the best way to address failure is by improving observability, so that users can diagnose an error themselves and avoid it in the future.

Recognition failure is a common issue, often caused by transcription errors in the ASR model, due to, e.g., noise in the audio signal, stopwords, or poor recognition accuracy. To address this issue, we add a visual cue at the corner of the IDE that reports the phrase transcribed in realtime, as well as the action (if any) that was triggered on intent recognition.

C. Command Discoverability

We draw a distinction between capability discovery and intent recognition. Users previously familiar with the IDE capabilities may wish to invoke or bind a specific action directly, but may not know the specific action identifier. The first and foremost way to improve discovery is through documentation, however keeping documentation in sync with capabilities can be challenging, and users are uninclined to read verbose documentation. To address this issue, we auto-generate documentation for each action by preprocessing the action list and annotating each action with a natural language description from the IntelliJ Platform source code.

Another common scenario is when the user is unfamiliar with the IDE capabilities, and the action they wish to perform is not supported by the IDE directly. User awareness of IDE functionality is outside the scope of this plugin, however, certain actions users may intend to perform are compound actions or have no associated binding. For example "delete the method named ⟨foo⟩" requires first resolving the method, selecting the body of the text and deleting it. Such actions can be defined, but require implementing a handcrafted recognition handler, and are generally quite brittle. We hope to improve scripting support for such actions in the future.

D. API Extensibility

Idiolect tries to anticipate users actions and expose default action bindings. Some functionality, however, is best left implemeneted by downstream developers, who are better-equipped to understand their users' needs. In addition to end-user configurability, Idiolect is designed to be extensible by other IntelliJ Platform plugins, which can define additional commands, recognition handlers and custom actions. We offer a simple message-passing API for plugins to communicate with Idiolect, and a DSL for defining custom grammars.

E. Error Recovery

Another common usability barrier is when transcription is accurate, but the utterance does not correspond to an actionable command, possibly due to stopwords, verbal fillers or extraneous text which cannot be directly parsed. In short, if a given phrase, e.g., "open uh foo java" is received, we attempt to repair the utterance. At the user's discretion, or in case of ambiguity when the phrase has multiple plausible alternatives, we can either visually or verbally prompt the user to choose from a set of actionable phrases, e.g., "Did you mean ⟨a⟩open file foo.java, ⟨b⟩open folder foo/java, or ⟨c⟩something else?"

To correct recognition errors, we apply Considine et al.'s [7] work on Tidyparse, which supports recognition and parsing of context-free and mildly context-sensitive grammars, and computing language edit distance. Tidyparse implements a novel approach to error correction based on the theory of context-free language reachability, conjunctive grammars and Levenshtein automata. We use a SAT solver to find the smallest edit transforming a string outside the language to a string inside the language. Only when the utterance is one or two tokens away from a known command do we attempt a repair.

VoskAPI is also capable of returning a list of alternate utterances alongside a confidence score for each. We use this list to determine if any recognized utterance is sufficiently close to an actionable command in the Idiolect grammar.

F. Build Reproducibility

One of the challenges of managing a multiplatform audio project is ensuring reproducible builds. The IntelliJ Platform does not allow shipping OS-specific plugins, which requires building a single artifact. In the past, updates would result in breaking changes on Windows, macOS, or Linux, which are difficult to debug or test from a development environment and only identified by user acceptance testing on those platforms.

To streamline the development process and mitigate the risk of platform-specific regressions in production, we set up an automated pipeline for testing and deploying the plugin. We use GitHub Actions to build the plugin for each release, and when a new commit is tagged with a release number and merged, a changelog is automatically generated from the intervening commit messages, then the plugin is signed and automatically uploaded to the JetBrains plugin repository. This allows us to quickly iterate on the plugin, test the plugin on multiple platforms, and to release new versions with assurance.

V. Evaluation

We conducted a preliminary experiment to evaluate recognition performance on synthetic speech. Using a set of English TTS voices (both male and female) included in macOS Ventura 13.1 with IntelliJ IDEA 2023.1, VoskAPI 0.3.45 and Idiolect 1.3.3, we synthesized 100 utterances from the predefined command list using the `say` command, then transcoded a 16000 kHz PCM signed 16-bit little-endian audio recording using `ffmpeg`, and finally transcribe the audio using VoskAPI to simulate a user-generated utterance. We report the average word error rate (WER) of three ASR models trained on US English speakers: `vosk-model-en-us-0.22` (0.30) `vosk-model-small-en-us-0.15` (0.28), and `vosk-model-en-us-0.22-lgraph` (0.35). Results on a per-voice and per-model basis are shown in Fig. 2 below.

	Eddy	Flo	Fred	Reed	Rocko	Samantha	Sandy	Shelly
…-en-us-0.22	0.581	0.635	0.888	0.627	0.981	0.293	0.492	0.528
…-en-us-0.22-lgraph	0.631	0.659	0.877	0.672	0.977	0.346	0.544	0.569
…-small-en-us-0.22	0.793	0.736	0.905	0.760	0.998	0.271	0.725	0.661

Fig. 2. WER for synthetic ASR using 100 commands from the default lexicon across three US English models and seven TTS voices (lower is better).

Two primary trends are evident. First, model size has a weak, but observable effect on recognition performance, consistent with prior work on large-vocabulary continuous speech recognition. Though we only evaluated three of the smallest English-langauge models from the Vosk dataset, based on plentiful evidence in the scaling literature, we expect this trend to hold for larger models across natural languages. Out of the three models tested, we note that the `vosk-model-small-en-us-0.22` has the lowest overall WER across all voices. Second, is the presence of gender bias in the MacOS/Vosk TTS/ASR pipeline. Depsite the apparent intelligibility of both sets of voices, WER is substantially lower on female voices (Flo, et al.) than their corresponding male counterparts (Eddy, et al.). The cause of this discrepancy is not immediately clear, but merits further investigation.

We also evaluated the plugin using a variety of downstream metrics, such as user downloads of the plugin over a five-year timespan, bug reports, GitHub Insights, and feedback from the JetBrains Plugin Marketplace to help determine user satisfaction and identify areas for improvement.

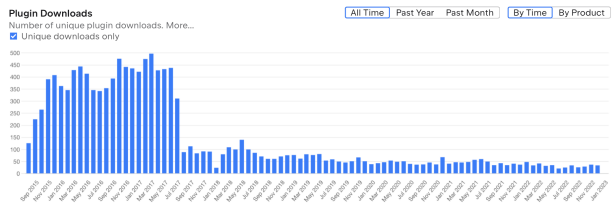

Fig. 3. Plugin downloads stalled circa 2017 due to IDE incompatibility.

Downloads of the plugin stalled since its initial release in 2015. Originally based on the CMUSphinx library that

was later abandoned by its maintainers, and lacking regular updates, our plugin became incompatible with more recent versions of the IntelliJ Platform. Following a five-year hiatus, we performed an intervention, rewrote the plugin from scratch based on a new speech recognition engine, implemented the usability improvements described in Sec. IV, and republished the plugin under its current name. Our assessment is ongoing.

VI. Threats to Validity

While voice programming may serve a niche purpose for developers, it has not been fully validated as a general programming mechanism. Our experience suggests frequent vocalizations may disturb programmers within earshot and we anticipate its usefulness in colocated workspaces will thus be limited, but do foresee some genuine use-cases, e.g., providing assistance to programmers with visual and motor disabilities.

We also acknowledge that our system is limited by its own natural language understanding capabilities. Speech is a comparatively inefficient medium for conducting programming tasks and users may be hesitant to invest time reconfiguring the system if intent recognition accuracy is low. Furthermore, the IDE plugin is currently limited to the set of actions that can be easily expressed using the IntelliJ Platform API.

Finally, we acknowledge that synthetic voices are, while increasingly lifelike, an imperfect proxy for human speech. The voices available on macOS are not a representative sample of human dialects and accents, nor we have not yet conducted a systematic human evaluation. The lexicon used to generate synthetic speech is also limited to the set of commands supported by the IDE, and does not capture how the plugin would perform in a more general-purpose setting.

VII. Future Work

In the future, we plan to conduct a thorough user study to better understand the use cases for the plugin. In particular, we hope to understand the development habits of visually and motor-impaired users to improve accessibility, as well as broader support for other natural languages and dialects.

We plan to conduct an evaluation of the plugin's intent-recognition capability and improve support for personalization, such as speaker adaptation and user-specific language models.

We also aim to improve the interaction mechanism by adding audiovisual feedback and modal voice commands. We would like to provide a more user-friendly configuration interface, possibly using an embedded DSL for defining dialog trees with a choose-your-own-adventure style modal dialog navigator. Incorporating modal logic à la Hazel [8], by adding a visual backpack and typing constraints would be an interesting direction to take and one we hope to explore.

Currently, we only track download statistics, although to inform the development of these features and better address the needs of voice programmers, it would be helpful to collect minimal telemetry on user utterances and custom commands.

Recent progress in machine learning has enabled the use of large language models (LLMs) for a variety of tasks, including speech recognition, machine translation, and text generation. An LLM trained on a large corpus of natural language can be used to perform simple reasoning tasks, such as predicting the most likely intent from a given utterance. For example, the utterance, "I want to edit foo.java" is more likely to be the command, "open foo.java" than the command, "execute foo.java". While LLMs are currently served on the cloud, recent efforts to compress and do inference on commodity hardware are ongoing. We predict in the next few years these models will become available on the edge, and can be used to perform end-to-end speech-to-intent recognition in realtime.

Given their increasing reasoning capabilities, we would like to further explore the use of LLMs within the intent recognition pipeline. In particular, we could use language models to rerank the most likely intent corresponding to a user's utterance, conditioned on a previous historical commands, the UI state and other application-specific settings. Contextual-awareness would require more careful IDE integration and enable us to more accurately predict user intent.

VIII. Conclusion

In this work, we presented Idiolect, a plugin for the IntelliJ Platform that allows users to control the IDE and reconfigure the plugin using voice commands. This addresses a common usability issue in voice UX design, where users are unable to express their intent in a manner the system understands. We described the design of the plugin, and the challenges we faced while implementing it. We also presented the results of an early evaluation, and discussed avenues for future work.

On one end of the design spectrum are metaprogramming languages and software language engineering frameworks that allow users to define their own commands and build embedded domain specific languages (eDSLs). These systems can be powerful, but require a high upfront investment from the user, who must design a language, and then learn the language itself, all whilst doing their daily job as a software engineer. On the other end are finite languages, which are too inflexible.

In the middle of this language design spectrum however are *idiolects*, which give users the freedom to create conversational domain-specific languages according to their own idiomatic style: this is the design space which Idiolect occupies. By targeting IDEs, whose users are typically adept programmers, commands with more complex semantics can be expressed programmatically, then invoked on-the-fly, sans recompilation.

We believe the intersection between conversational agents, vernacular programming languages and programmable voice assistants to be fertile ground, and one that is relatively unexplored in the language design space. We hope our small contribution inspires others to explore these ideas and take steps towards making programming more fun and accessible.

IX. Acknowledgements

The first author wishes to thank his former colleagues Alexey Kudinkin and Yaroslav Lepenkin, who contributed to the plugin during the original JetBrains hackathon in 2015, and former manager Hadi Hariri for his advice and encouragement.

REFERENCES

[1] M. Shaw, "Myths and mythconceptions: What does it mean to be a programming language, anyhow?" *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 1–44, 2022.

[2] E. W. Dijkstra, "On the foolishness of" natural language programming"," *Program Construction, International Summer School*, pp. 51–53, 1979.

[3] J. L. Leopold and A. L. Ambler, "Keyboardless visual programming using voice, handwriting, and gesture," in *Proceedings. 1997 IEEE Symposium on Visual Languages (Cat. No. 97TB100180)*. IEEE, 1997, pp. 28–35.

[4] S. C. Arnold, L. Mark, and J. Goldthwaite, "Programming by voice, VocalProgramming," in *Proceedings of the fourth international ACM conference on Assistive technologies*, 2000, pp. 149–155.

[5] A. Begel and S. L. Graham, "Spoken programs," in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 2005, pp. 99–106.

[6] M. Chkroun and A. Azaria, "Lia: A virtual assistant that can be taught new commands by speech," *International Journal of Human–Computer Interaction*, vol. 35, no. 17, pp. 1596–1607, 2019.

[7] B. Considine, J. Guo, and X. Si, "Tidyparse: Real-Time Context-Free Error Correction," in *Workshop on Live Programming*, 2022. [Online]. Available: https://github.com/tidyparse/tidyparse

[8] C. Omar, D. Moon, A. Blinn, I. Voysey, N. Collins, and R. Chugh, "Filling typed holes with live GUIs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 511–525.