Toward A More Scalable End-User Scripting Language

Alessandro Warth[†]
alex@vpri.org

Takashi Yamamiya[†]
takashi@vpri.org

Yoshiki Ohshima[†]
yoshiki@vpri.org

Scott Wallace[†]
scott@vpri.org

[†] *Viewpoints Research Institute*
1209 Grand Central Ave.
Glendale, CA 91201

Abstract

End-user scripting languages are relatively easy to learn, but have limited expressive power. Tile-based scripting systems are particularly accessible to beginners, but usually are very limited in scope and usually lack extensibility, and for some tasks the tile idiom becomes cumbersome. Conventional programming languages used by computer professionals are far more powerful, but at the cost of additional complexity and limited environmental support, which place them out of the casual programmer's reach. This paper presents TileScript, an attempt to combine the accessibility of a tile-based programming interface with the leverage of a full textual programming language and with a simple means of extension, making it potentially an appealing tool for the novice programmer without sacrificing any expressiveness. All TileScript programs, whether built originally with tiles or textually, can always be edited both graphically via a drag-and-drop tile interface and textually, and the user can freely switch back and forth between tile and textual representations at any time. Additionally, TileScript's simple yet powerful extensibility mechanisms allow the language to be used to tackle problems that would normally be out of the scope of an end-user scripting language.

This material is based upon work supported by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

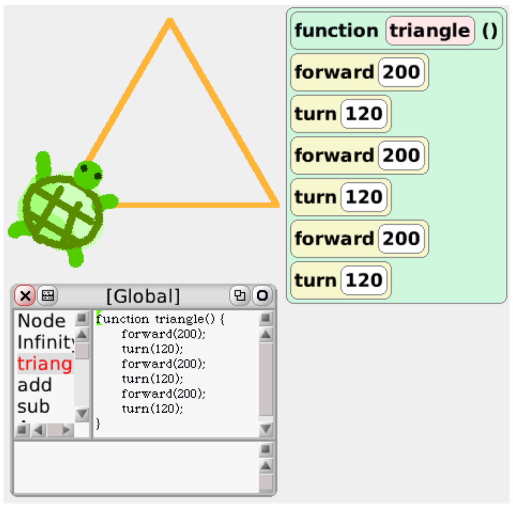
1. Introduction

Computers are present in virtually every aspect of our lives: middle-schoolers and grandmothers alike use them at home, at school, and at work. Sadly, computers are mostly used for activities like word processing, web browsing, e-mail, and games. To truly take advantage the power of their computers, *end-users* must be able to write programs. This makes end-user programming languages and environments an important area of research.

A number of end-user programing systems have been proposed. These systems have focused on various goals, and can be classified based on several criteria. One such criterion is the *steepness of the learning curve* — in other words, how easy it is for a new user to become productive. In systems like Viscuit [1] and StageCast [2], for example, simple graphical pattern-matching and rewrite rules allow end-users to compose simple animations quickly and easily, with very little training required.

Another criterion is the *ceiling height*, i.e., how well the language scales to more complex problems. A full-fledged programming language with many built-in features, where the user writes textual programs, will get the highest marks on this criterion.

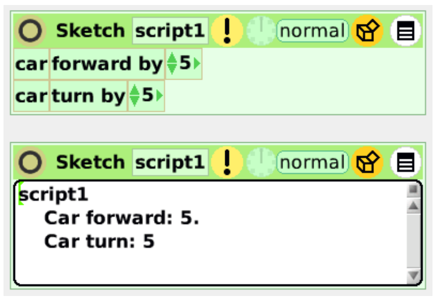
The tension between these criteria makes the task of designing an end-user programming system a difficult one. No system that we are aware of has both a gentle learning curve *and* a high ceiling. Existing systems tend to spread on the spectrum from gentle learning curve/low ceiling to steep learning curve/high ceiling. Viscuit, for example, has a very low learning curve, but expressions in Viscuit are represented solely iconically; Viscuit, in fact, does not even have the concept of numbers or arithmetic. In such systems, one quickly hits the rather low ceiling, after initially enjoy-



```
function triangle ()
forward 200
turn 120
forward 200
turn 120
forward 200
turn 120
```

```
function triangle() {
    forward(200);
    turn(120);
    forward(200);
    turn(120);
    forward(200);
    turn(120);
}
```

Figure 1. A screenshot from the TileScript system. The same code is shown as tiles and text-based code.



```
car forward by 5
car turn by 5
```

```
script1
    Car forward: 5.
    Car turn: 5
```

Figure 2. The same Etoys script in graphical representation and textual representation.

ing simple and easy programs.

Systems like Scratch [3] offer graphical building blocks that the user can manipulate and combine with the mouse to construct programs. The blocks are symbolic representations of the elements of a program (such as numbers, symbols, and control structures) so the user has to learn what each block does and means, but as he learns more about these blocks, his productivity increases. Unfortunately, this gentle learning curve has some negative consequences; for example, Scratch intentionally omits the concept of inter-object reference to avoid confusion. This design choice makes it tricky to write a program where multiple objects work together.

At the other end of the spectrum, there are languages and systems with fully textual code. Many educational and end-user development environments are based on Java, Python and other conventional languages, which makes them very expressive. Also, text-based programming tends to be more efficient than tiles as programs grow in complexity. *Processing* [4] and J0 [5] are based on simplified versions of Java, and provide end-user-oriented development environments. However, users have to deal with syntax errors, and face a steeper learning curve.

1.1. The Goal

Our goal is to create a new end-user scripting language with a gentle learning curve and a high ceiling. Our system should support both tile scripting and textual coding, and the transition between these two representations should be as smooth as possible. Specifically, we aim to make it possible for the user to:

- convert a tile script into equivalent textual code, and vice-versa;
- extend the system by creating new kinds of tiles (abstractions), and optionally customize their appearance. (The meaning of new tiles should be described using tiles or textual code.)
- "pop the hood" of any part of the system, so that he can learn about (and perhaps even modify) it.

Although users will most likely get started using the tile scripting interface, all of the knowledge they acquire with the tiles will still be valid once they make the transition to text-based programming.

Our idea draws upon Squeak Etoys [6] [7]. Etoys, like Scratch, offers visual building blocks (called "tiles") which the user can combine by dragging-and-dropping to make a unit of program called a *script*.

Etoys tile scripts can be converted into textual code (represented in the system's base language, Squeak Smalltalk), as shown in Figure 2. Unfortunately, this conversion is only "one way"; once the user edits the code textually, there is no mechanism to convert the edited text back to tiles. This limitation exists because Smalltalk is a lot more expressive than the tile language, which makes it is possible for the user to type in code that does not have a corresponding graphical tile representation. Also, the Etoys object model is not the same as Smalltalk's, which means that the knowledge of the model the user acquires by using Etoys cannot be translated to Smalltalk programming. Lastly, because Etoys tiles are implemented in Smalltalk by the system's developers, the end-user is cannot extend the tile language with new tiles; he is limited to using the ones provided by the developers.

Figure 3 shows a comparison of the systems described in this section. In the figure, the vertical axis denotes the steepness of learning curve; the lower the oval is located, the easier to start using. The horizontal axis denotes the complexity of programs one can write *comfortably* in the system.

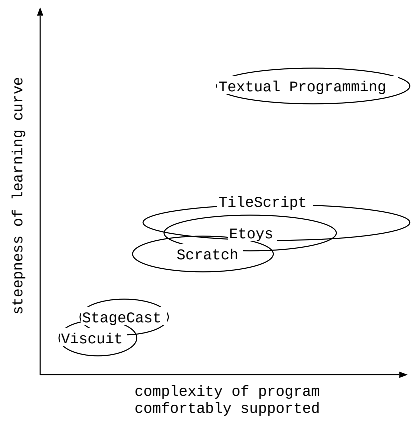


Figure 3. A classification of various end-user programming systems.

Our proposed system in the middle has similar steepness of learning curve as Etoys but trys to cover wider kind of programs.

1.2. Approach

We chose to use JavaScript [8] as the basis for *TileScript* both because of its simple object model and good reflective facilities, and because we had already produced our own implementation in Squeak.

Building our system on top of Squeak gave us access to the Morphic graphics framework [9], which facilitated the creation of our tile-based user interface and gave us access to a rich library of graphical objects.

We extended our JavaScript implementation with a simple macro system. TileScript's user-defined tiles are implemented as macros; since macros are also part of our base language, they can be written using tiles as well as textual code.

TileScript programs are stored as parse trees. We implemented conversions from textual code and tiles to parse trees, and from parse trees to textual code and tiles. Thus, tiles and textual code are views of the same model.

Lastly, we provided a mechanism that allows users to customize the visual appearance of new as well as existing tiles.

Figure 1 shows a screenshot from the system. The function that draws a triangle is shown in tiles on the right and in text at the bottom in a JavaScript object inspector.

The rest of this paper is organized as follows. Section 2 briefly describes our JavaScript implementation and our extensions to the language. Section 3 describes our extensible tile-based programming interface. In section 4, we discuss the findings from this experiment. Section 5 discusses related work. Section 6 discusses future work and concludes.

```
macro @if(cond, tbranch, fbranch) {
   if (cond)
      tbranch
   else
      fbranch
}
```

Figure 4. Definition of the `if` macro.

2. Our JavaScript Implementation

JavaScript is a language that features a dynamic, prototype-based object model. It has notably rich reflective features. While existing implementations widely used in common web-browsers and elsewhere provide these dynamic and reflective features, we decided to implement our own JavaScript on top of Squeak, which is another dynamic language. This approach allows us to change the language freely, to do deeper introspection of program execution, and to leverage the powerful Morphic GUI framework.

To capture the syntax structure of code, we have added a new macro system to the language. We use macros as the internal representation of tiles in our tile-scripting system. Since macro definitions are made in the same language, the end-user can write his own macros and hence define his own tiles, as we shall see below.

In the rest of this section, we briefly explain our base implementation of JavaScript and the macro system.

2.1. The Base Implementation

Our JavaScript parser and compiler are written in OMeta [10]. OMeta programs resemble EBNF grammars with interleaved semantic actions; we use a few different OMeta grammars to convert JavaScript programs to executable Smalltalk code.

A JavaScript object is a dictionary-like entity, so it is represented as a Dictionary in Squeak. JavaScript field accesses are converted to Squeak dictionary access-by-key operations, with delegation to the prototype object.

Our JavaScript implementation is written in 350 lines of OMeta, together with 750 lines of JavaScript library code written in JavaScript itself. This compactness is one of the keys in this project, as we would like to make the inner workings of our system fully accessible to the user.

2.2. The Macro System for Tiles

We have added a macro system to the base implementation described above. A macro definition begins with the `macro` keyword, followed by a macro name, which must

```
macro @repeat(k, body) {
   var n = k
   while (n-- > 0)
      body
}
```

Figure 5. Definition of the `@repeat` macro.

start with a `@`. For example, Figure 4 shows a macro version of the `if-then-else` statement, which can be used as follows:

`@if(5 > 6, alert("yes"), alert("no"))`

The example we are going to use in the rest of this paper is the `repeat` statement. Repeat is a simplified version of the `while` loop that is useful in turtle geometry and other end-user oriented programs. The macro definition of `repeat` is given in Figure 5. Note that upon expansion, the local variable `n` will get a unique internal name so that nested `repeats` will work.

Macro expansion happens at compile time, whereas at parse time, macro applications are kept in the parse tree; a macro instantiation corresponds to a tile instance, and the conversion from/to the graphical tile is done to/from the parse node that represents the macro instantiation. For example, when the user writes a code snippet with the macro, like:

```
@repeat(10, alert("hello"))
```

a `repeat` object is created and its two fields are initialized with the arguments (i.e., a parse tree for *10* and another parse tree for `alert("hello")`).

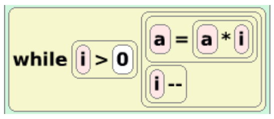
Notice that the bodies of the macros in our system are written in the end-user language, which simplifies the task of creating a new tile.

3. Tile Implementation

The macro system described in the previous section allows us to make any desired subset of a textual language available for viewing in graphical form. The textual code is parsed to create a parse tree. In this section, we describe the conversion to graphical tiles from the parse tree, as well as conversion in the opposite direction.

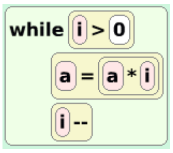
3.1. Conversion To Graphical Tiles

For each macro application, the parser creates an instance of the JavaScript "class" `Tile` that represents a node in the parse tree. There are different kinds of `Tile` objects to represent different types of syntax nodes; we provide pre-made `Tiles` for each basic JavaScript construct



```
while i > 0   a = a * i
              i--
```

Figure 6. The appearance of `while` statement with default look.



```
while i > 0
    a = a * i
    i--
```

Figure 7. The appearance of `while` statement with customized look.

such as `if`, `while`, etc. so that these can be also represented visually. These different `Tiles` delegate to a "superclass" called `GenericTile`, which defines common behavior for all tiles; the specific tile types specialize this inherited behavior as appropriate.

A method called `makeTile()` is implemented by the `Tiles`. In the TileScript implementation, this method escapes to underlying Smalltalk code that creates the graphical tile representation in the Morphic GUI framework. Since the tiles can be nested, `makeTile()` is also called recursively to create nested graphical tiles.

We could have used a single, universal, graphical representation for every macro, since all macros have the same structure (i.e., one parent and zero or more children nodes.) Figure 6 shows a hypothetical visual of the `while` statement in this generic way. The generic tile would be created with a list of sub-tiles, which would then be laid out in a simple manner.

However, we would like to provide better looking, better-suited, and more distinctive, tiles for the most commonly used basic language constructs. For example, the graphical representation of `while` actually looks like Figure 7.

The specialized look of such a tile is created by hand in Morphic. Morphic provides a direct manipulation interface for creating and copying "Morphs" ("Morph" is the basic graphic object in Morphic), for changing their size, color, border, etc., and most notably for embedding them into one another. By using this interface, we manually create a Morph structure that serves as the "template" for a particular tile.

The tile designer may embed as many "spacer" Morphs, and other cosmetic Morphs, as he wishes, to lay things out nicely and to create precisely the appearance he prefers. He then needs to designate (via a menu) a morph to represent each "hole" to be filled by tiles dropped by the user during drag-and-drop tile scripting; each "hole' Morph" is subsequently marked with a distinctive Morphic "property" so that the TileScript system can know which morphs are to be replaced. In the case of `while`, for example, there are two holes' to be marked, one for the boolean expression to be evaluated, and another for the statement(s) to be repeatedly executed.

In the implementation of `makeTile()` for a tile which has such a user-defined graphical tile template, the template is deeply copied and then the Morphs that represent the holes are replaced by the tiles that represent sub-trees.

The end-user can easily use the same mechanism to create customized tiles to suit his personal taste. For `repeat`, for example, the end-user would assemble Morphs and make a good-looking Morph with (more than) two submorphs. He would then identify two morphs (the iteration count and the body) via the UI. Once the user is happy with the look, TileScript stores the Morph as the new template for `repeat`.

Note that it is not strictly necessary to create a customized look for a user-defined tile; the generic tile can provide the same editing functionality.

When a macro node in the parse tree contains a non-macro (i.e., textual) node, a special kind of tile that behaves as a text field is instantiated. The layout algorithm of these tiles (including the text field tile) is simple at this point and resulting morphs tend to be sparse when the user mixes textual code and tiles.

Needless to say, the tile representation can be edited graphically. The user may obtain new tiles from any of the available templates, drag-and-drop them to construct a function, or delete tiles by dragging them out of structure, and he can type in expressions in the text fields.

Each of the basic constructs in textual code (such as `while`, `if`, or a function application), has its own predefined macro.

3.2. Conversion from Graphical Tiles

So far we have explained how to create a graphical representation from textual code that contains macros. To enable graphical editing by the user, we also need a way to convert the graphical representation back to a parse tree and to textual code.

If we look at a graphical tile (represented as a Morph) in the script, there are submorphs that are marked as particular sub-nodes in a tree. The converter recursively looks at the sub trees in the Morph and converts them to parse trees.

For a user-defined tile, the same macro definition can be used to do the conversion from tiles to the macro node. The macro specifies the name of tile, and the names (and numbers) of arguments. As long as the user marks the morphs that represent the sub-trees (provided as arguments) properly, the converter can visit the submorphs and convert them to partial parse nodes. Then, the parse node object for the user-defined tile itself is created with these sub-trees.

Each parse node knows how to convert itself to textual code, which makes it possible for parse trees to be rendered as text. In the current implementation, the indentation and layout in the original text is lost even if the user were only to write some textual code, convert it to the tiles, and then go back to textual code without modifying the tiles; we are planning to add more attributes to parse tree nodes so that properties such as comments, indentation levels, and positions in the original source code are preserved when possible.

3.3. Conversion from Base JavaScript to Macros

It is also useful to be able to convert arbitrary textual code written in the base JavaScript language (i.e., without macros) to the form with macros. To do this, we provide a macro definition for every JavaScript construct. The definition of `@if` above is an example of such a macro. By running a visitor that converts JavaScript constructs to macros, one can convert the textual JavaScript code to tiles.

4. Discussion

In this section, we discuss our findings and prospects for future work.

4.1. Extensibility

One of our goals was to provide fully bi-directional transformations between textually-written code and graphical tiles. We succeeded in doing this. However, we feel that our implementation is not as deeply extensible as it should be, for we can neither view nor modify the semantics and syntax of the language itself.

We had attempted to define a method called `eval()` in JavaScript for each kind of parse node; i.e, we tried to provide a meta-circular implementation of JavaScript which could be viewed and modified in the same system. However, a meta-circular definition has to have a fixed point, and we realized that the fixed point cannot be so *deep*; a function definition requires functions defined, a function call uses many function calls, getters and setters get and set values from objects, and `if` requires a `if` statement. These facilities cannot be written in the user-level language. In the other words, they have to be "primitives" and the user cannot modify them, for example, using tiles.

In this sense, we contend that the macro system provides better separation between the base language and the language that the end-user works with. For example, consider the `@if` macro. The user can still change the look of its associated tile, and even change its semantics. From the user's point of view, this is a very powerful concept even though he can neither change nor even see the definition of `if` in the base language,

We would like to revisit this issue in the future and try to define a language with a smaller core.

4.2. The Choice of Base Language

The choices of base language and end-user language have interesting trade-offs. Having a mainstream syntax helps to lower the learning curve in a practical sense, but from the standpoint of making text and tiles isomorphic, it can be problematical.

For instance, we would like to provide a model of the grammar that advanced end-users can access and understand. One of the authors made a kind of a visual grammar editor called LanguageGame [11]. We plan to experiment with an end-user grammar editor along the line of LanguageGame.

Had we decided to use a language with uniform syntax, like LISP or Scheme, as our base language, it would be fairly easy for end-users to understand the grammar, because it would be so simple. Javascript syntax has a much more complex grammar, and will certainly make the addition of extensible syntax to our system more difficult on the end-user.

4.3. The Type System

One of the biggest advantages of tile scripting is that it can be made type-safe without any additional complexity. The drag-and-drop interface can be made so that only type-conforming tiles may be combined, and the resulting script doesn't produce any run time errors.

TileScript does not currently have any type-checking mechanisms. Since TileScript allows textual coding and conversion to tiles, developing a reasonable type system will be an interesting challenge.

4.4. Formatting Textual Code

In the current implementation, parse tree nodes do not carry any information about the formatting of their corresponding textual code. Consider what happens when the user writes some code textually, converts it to tiles, and then makes some very small modification. If he converts the tiles back to textual code, all formatting information such as indentation and line breaks will be lost. In order to minimize such nasty surprises, TileScript parse tree nodes should contain as much formatting information as possible (these should be stored as properties), and the different conversions take them into account.

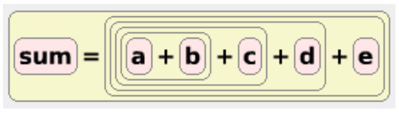


Figure 8. A nested tree from a flat expression.

4.5. Structure of Tiles

In our current implementation, the visual appearance of a tile script follows the structure of the parse tree. This is fine in most of the cases, but can be cumbersome for a chain of arithmetic operators. For example, if the user thinks about an expression:

```
sum = a + b + c + d + e
```

he does not need to think about operator precedence; it is better to simply think of the above expression as the "sum of five values". However, the in the tile representation, the user is inevitably forced to look at nested expressions as shown in Figure 8. A sophisticated tile scripting environment should allow the user to edit such expressions in less rigid ways.

5. Related Work

Scratch is a tile-based scripting language for kids. A pervasive simplicity and a well-polished user interface make Scratch a welcoming environment for first-time users. Naturally, this ease of use comes at a price. Unlike most other programming languages, Scratch provides no form of abstraction (e.g., users cannot create their own tiles or functions.) Scratch projects that go beyond a certain threshold of sophistication tend to contain unwieldy programs that can be difficult to write and to maintain.

Our own Squeak Etoys, another easy-to-learn tile-based scripting language, supports user-defined procedures, or *scripts*. Although Etoys scripts cannot return values (which makes them less powerful than functions), they provide an important form of abstraction that in turn enables users to tackle moderately sophisticated projects without "feeling like a dog standing on its hind legs". This is clearly a step in the right direction, but we can do better.

TileScript is our attempt to create an end-user programming language that is just as welcoming to new programmers as Scratch and Etoys, but which scales to arbitrarily complex tasks like a conventional programming language.

Squeak's Universal Tiles (UniTiles) was an earlier attempt at creating a more scalable end-user scripting system. UniTiles was isomorphic to Smalltalk, which made it just as expressive as—but no easier to learn than—Smalltalk. Additionally, because Smalltalk does not support macros, a UniTiles implementation of our `repeat` example would require its `body` argument to be passed in as a block, which can be confusing to beginners. TileScript's macro construct and its associated user interface component make it easy for end-users to define their own control structures.

Like TileScript, IMP [12] allowed programmers to define their own control structures using a macro-like mechanism. The main difference between these two approaches is that IMP was based on syntax extension: each new control structure was accompanied by a BNF description of its intended syntax. The analog of syntax extension in TileScript is our tile-creation GUI, which allows end-users to design the look-and-feel of the tile associated with a particular macro. We believe this approach to provide a better fit with our intended users, who are not expert programmers.

6. Conclusions

In this experiment, we added a small macro system to JavaScript to capture both the syntactic structure and the semantics of textual code into parse tree nodes that are used as internal representations of the code. We also described conversions between arbitrary tile scripts and arbitrary textual code, which work in either direction.

With these facilities, we managed to build a system in which new kinds of tiles in the tile-scripting system can be defined in a way accessible to end-users. All of the constructs of our language are available to the end-user in both tiles and text, which allows the end-user to explore all aspects of the system in a fairly deep manner.

This project is still in its infancy. As discussed in the previous section, we aim to allow even deeper exploration by defining more parts of the system in itself, and also to make the system more practical for real end-users to use effectively.

Acknowledgement

We would like to express thanks for the valuable insights that Alan Kay, Kim Rose, Ted Kaehler, Bert Freudenberg, Ian Piumarta, and other colleagues have given us.

References

[1] Y. Harada and R. Potter, "Fuzzy rewriting: soft program semantics for children," in *In Proceedings of Human Centric Computing Languages and Environment*, October 2003, pp. 39–46.

[2] D. C. Smith, A. Cypher, and L. Tesler, "Programming by example: novice programming comes of age," in *Communications of the ACM*, vol. 43, no. 3, March 2000, pp. 75–81.

[3] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick, "Scratch: A Sneak Preview," in *In proceedings of Second Conference on Creating, Connecting and Collaborating through Computing (C5)*, 2004, pp. 104–109.

[4] C. Reas and B. Fry, *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, 2007.

[5] A. Jonas, D. Lee, and A. Myers, "J0: A Java Extension for Beginning (and Advanced) Programmers," `http://www.cs.cornell.edu/Projects/j0/`.

[6] A. Kay, K. Rose, D. Ingalls, T. Kaehler, J. Maloney, and S. Wallace, "Etoys & SimStories," February 1997, ImagiLearning Internal Document.

[7] B.J. Allen-Conn and K. Rose, *Powerful Ideas in the Classroom*. Viewpoints Research Institute, 2003.

[8] "ECMAScript Language Specification," 3rd edition (December 1999).

[9] J. Maloney, *Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2002, ch. 2: An Introduction to Morphic: The Squeak User Interface Framework, pp. 39–68.

[10] A. Warth and I. Piumarta, "OMeta: an Object-Oriented Language for Pattern Matching," in *In proceedings of Dynamic Language Symposium*, 2007, (to appear).

[11] T. Yamamiya, "Languagegame - an interactive parser generator," pp. 110–117, 2003.

[12] E. T. Irons, "Experience with an extensible language," in *Communications of the ACM*, vol. 13, 1970, pp. 31–40.