Making MinNo: Lexers, Parsers, and A Whole Mess of Parenthesis

By Logan davis

Abstract:

MinNo is acompiled language targeting the Arduino platform. This paper, as part of my Plan of Concentration, is an overview of the technologies and techniques used in making MinNo's compiler as well as a collection of opinions of the Racket language and ecosystem (MinNo's implementation language).

Contents:

Racket: A LISP for Language Design

For the lexer, parser, translator, and various tools of MinNo, I chose to use Racket. Formerly known as PLT scheme[CITE], Racket is one of the many opinionated and highly idiosyncratic members of the LISP family of languages. Where CLISP emphasizes maturity[CITE], Clojure focuses on functional constructions[CITE], and Scheme strives towards simplicity[CITE], Racket has built itself around specificity.

Originally, Racket was an education centric platform. It's homegrown IDE, ***DR.Racket*** allows for you to turn parts of the language on and off to slowly introduce new coders to extended functionality in small steps. Ultimately, prior to my time using the language, they still wanted to make an educational language but they wanted it to also be useful to the wider community. Racket made a concerted effort to change it's strengths.

Now Racket markets itself as a "language platform." The entire ecosystem is built around the idea of embedding Domain Specific Languages (DSLs) into programs to simplify and better manage sub tasks of larger programs[CITE]. To achieve this focus, the developers of the language have included built-in lexers[CITE], parsers[CITE], and an extensive macro system[CITE].

With Racket's ability to handling syntax parsing and my previous

experience with it, it came as a natural choice to use when implementing a compiler for MinNo.

The Lexer Itself: The Matt Might Method

In creating this compiler, by far the most helpful resource was Matt Might's blog posts about his compiler course. The lessons are brief, but give an excellent overview of what one needs to do to make a compiler. I most heavily leaned on his course syllabus and class notes for MinNo's lexer.

A lexer is the section of a language compiler (or interpreter) that takes a source file and breaks it down into the languages smallest pieces (tokens). For example: if we were making a compiler for a basic calculator, the lexer would need to recognize "=", "+", "*", "-", "/", and any integer or float. The lexer would need to return these tokens into some form that allows the parser (a later step) to easily recognize them and construct larger grammar objects in the language.

Regular Expressions and Their Use

The most typical way to construct a lexer for non-trivial syntax structures is to strap together a collection of regular expressions into some class or function that consumes a source file and returns a collection of tokens.

Racket, being a language platform, has a builtin lexer & regex library. Matt Might thinks highly of it and uses it in his own compiler course. An example of how to construct a lexer in Racket, using our basic calculator language, may look something like the following:

```
(define calc-lex
  (lexer
    [#\+   '('ADD-OP lexeme)]
    [#\-   '('SUB-OP lexeme)]
    [#\*   '('MULT-OP lexeme)]
    [#\/   '('DIV-OP lexeme)]
```

```
    [(+ (char-range #\0 #\9)) ('INT lexeme)]
    [(: (+ (char-range #\0 #\9))
        "."
        (+ (char-range #\0 #\9))) (‘FLOAT lexeme)]))
```

Racket prefaces any character with a "#" to mean a *character literal*, so "#+" matches "+" in a source file. **lexeme** is a reserved word by the Racket lexer to reference the currently matched token. The last two entries in the lexer construct are examples of *s-expression regexes*. + recognizes 1 or more of the following regex. **char-range** is to match a range of character either in letters or in numbers (#\a to #\z or #\0 to #\9 for example). **:** concatenates all following expressions into one single pattern. Longer regular expressions are typically defined as lexer abbreviations to look a little more clean. With those abbreviations defined the lexer would look like this:

```
(define-lex-abbrev int? (+ (char-range #\0 #\9)))
(define-lex-abbrev float? (: (+ (char-range #\0 #\9))
                             "."
                             (+ (char-range #\0 #\9))))
(define calc-lex
  (lexer
    [#\+   '('ADD-OP lexeme)]
    [#\-   '('SUB-OP lexeme)]
    [#\*   '('MULT-OP lexeme)]
    [#\/   '('DIV-OP lexeme)]
    [int? ('INT lexeme)]
    [float? (‘FLOAT lexeme)]))
```

Given a source file containing "5 + 7.0 / 3" would produce a token stream of:

```
'('INT "5") '('ADD-OP "+") '('FLOAT "7.0") '('DIV-OP "/")
```

```
'('INT "3")
```

The resulting lexer would be used to listen to an input-port and produce a token stream for the parser to consume as needed.

Token Construction and MetaData

The lexer can do more than just recognize tokens. It can help provide invaluable debugging information about the source file it is lexing. For parsing libraries like the one I used (which I will get to next), it expects the lexer to append the location of tokens in the source file being compiled to use when reporting a parsing error. The lexer can also be a place to catch stylistic warnings (such as rustc's enforcing of *snake_case*).

The State of MinNo's Lexer:

MinNo's lexer works on an architecture similar to that used by Matt Might to take advantage of Racket's input ports. The lexer consumes a file-port's output and gathers it in a buffer. This is recursively done until the source file-port is empty. The resulting collection of tokens are passed to MinNo's parser.

The lexer works as intended, though some more utility could easily find it's way into that section of the compiler. constructing global tables of meta data about tokens (to help warnings in the translation step), or stylistic warnings (like I previously meantioned) will all be looked at when revising the lexer.

BRAG: How I Avoided Losing My Life to Making a Parser

Parsing is a topic of exploration large enough to the spend an entire dissertation on. For MinNo, instead of spending the time making one from scratch, or learning the incredibly idiom-centric syntax recognizer in Racket I opted to use a popular Racket tool for grammar specification and

parsing: *brag*. [MEANTION THAT MATTHEW FLATT USES BRAG].

brag is a *Backus Naur form* (BNF) grammar parser implemented as a *Domain Specific Language* within Racket. Before I get into using *brag*, some time should be taken to overview BNF.

Examples of Use and an explanation of BNF:

BNF grammars work using *symbols* and *expressions*. They are organized as such:

SYMBOL ::= EXPRESSION

an expression can be any collection of other symbols or terminals (tokens). If a symbol is in an expression, it is considered a nonterminal expression and must be parsed further. It is worth noting that, in use, some artistic merit is used with BNF notation. For instance, *brag* does not use "::=" for it's equal-sign, it just uses ":" so rules look like:

```
SYMBOL: EXPRESSION
```

Which can be confusing when first looking at it. *brag* allows for a common shim in BNF structures that allows expressions to be regular expression-like statements to allow concise grammar rules. So, to continue with the calc grammar, a *brag* rule to recognize numbers may look like:

```
num: (INT | FLOAT)
```

This rule states a "num" is a integer or a float (as tokenized by our lexer).

When a grammar file is given to the *brag* compiler, it produces a *parse* function. This function will consume a list and then, using the generated LALR [CITE] parser, it will construct an abstract syntax tree (AST) abiding by the defined grammar.

The State of MinNo's Grammar:

A few different versions of MinNo's grammar were toyed around with. Originally, MinNo was going to be of a LISP grammar, but between the limitations a LISP interpreter would have and the shoe-horned-ness of compiling it, I decided against the project. Later iterations took notes from F#, Scala, Java, Python, and, Pyret [cITE CITE CITE]. The result of those revisions is what you see currently. I feel it is a nice balance between Scala and F#'s (subjectively) clear type notation, and Python's clear syntax. By no means does it reach in any of those directions as far as the language MinNo takes it from (the type system is not even a fraction as complicated as Scala's and the keywords are not as clean as Pythons). MinNo just takes some loose form from languages I have used and enjoyed in the past.

The grammar produces a syntax object which is handled by the translator.

The Translator: A Handler Model

Like syntax rules [LINK], Racket's syntax objects are highly specific to the crowd that is doing cutting edge language research using the platform. To maintain my sanity, I decided to roll my own template-based method instead. Given MinNo's rather strict syntax, predicting the grammar that can be used at any point is a programs AST is fairly straight forward.

Avoiding BNF Hell

A word of warning, this is that area where you start to understand terrible mistakes you made in constructing your grammar or defining your lexer. Those steps can be made to consume and parse some pretty kludgy grammar structures. If they consume kludge they will also produce it. A complicated AST makes for a complicated project. It is worth the hours of revising your BNF grammar and lexer to save days of debugging further down the road.

Handlers:

The way I have decided to tackle MinNo's AST is to define a collection of handlers for different syntax structures. There are handlers for let statements, block statements, function definitions, conditionals, returns, and so on. Each handler knows how to pass off any sub-syntax structure it contains (such as the statements in a block statement) by using a handler directory. The directory used to dispatch sections of the program to their respective handlers until the program is fully translated. When is left is a new AST closer to that of an Arduino Language program.

Unpacking the Handler

What the handler leaves us with is a list of lists that represent the program's AST; this is far from executable. This is where the ***Unpacker*** comes in. The Unpacker spelunks through the AST and does one of three things:

- Returns contained inner string,
- Hands off special cases to a correcter function,
- or calls the Unpacker on sub lists for further unpacking.

The case of something that can just be returns is an atomic object of the AST such as:

```
'(lit "1")
```

There is no more information within this item and nothing needs to be formatted differently. However, in the case of certain AST objects require special unpacking to deal with some of the specific formatting of the Arduino language. Function calls, for instance, require special handling of arguements to make sure they are properly wrapped in parenthesis and seperated by commas. Another would be variables declared in **PROGMEM**; the Unpacker will wrap each reference of them (as a pointer) in a Arduino built-in function to read from program memory. The compiler knows how to do these correcrtions using a directory similiar to that of the handler's

directory function.

After the unpacking phase, the resulting Arduino sketch will be returned as a full string that is written out to a file for uploading.

Targeting AVR Processors:

Making Sure Things Stayed Fast

A goal of MinNo was to more easily handle the boiler plate code I use when writing Arduino scripts. Given the Arduino's slow speed and limited resources, I needed to make sure that the generated code was not so slow as to entirely undermine the reason for using MinNo. Therefore I made the concious decision to make MinNo more of a sublte semantic departure from C rather than something entirely different. It allowed me to exaggrate certain points about Arduino programming that I find useful (like making immutability the default to make easy use of the architecture difference). Making it a shift in default behaviour, rather than an entire paradigm shift allows MinNo to be closely and consisely mapped to a C program with only minor runtime overhead.

Letting People Under The Hood: Why Translator Result Readability Matters

One day I want MinNo to be a mature enough platform to write the majority of my Arduino sketches in. But the likelihood of achieving that level of maturity while also wanting the graduate within 4 months of writing this seems minuscule (to put it lightly).

To make up for this, the MinNo compiler focuses on making (subjectively) readable C code. Often compilers will leverage language constructs (such as lambdas) to make code generation easier however this frequently generates code that is hard to read.[CITE] MinNo's template approach lets the generated code be written so it can follow consistent styling and formatting for readability.

The code being readable allows the current restrictions of MinNo to be bypassed by editing the resulting C script. If you really want some kind of manual memory allocation, MinNo generated C will be totally accepting of that even if MinNo itself isn't.

Disadvatages of My Approach:

A Very Insular Crowd:

Though I can say much to praise Racket, there are definitely some detractors. For instance, in trying to get into their syntax object I was assulted with numerous conversation-specific words, phrases, and frameworks. Though the documentation gave a great deal of links on where to read further, it was just layers upon layers of coversations that I had no sane introduction to.

Most of these topics were managable, but not without staring at docs and opening a REPL again and again. Racket is a small community with hyper-specific language to talk about what they do. Consequently there aren't very many resources availible from people other than the ones responsible for the documentation itself.

Something like ANTLR or YACC/Flex might have been more easily approachable in hind- sight just due to the breathe of material for learning it. [CITE THAT PAPER ABOUT LISP BEING FOR WEIRD PEOPLE]

Handler are Both Great and Terrible:

Handlers are a wildly simpile and easy framework to use in AST processing. It allows for very iterative design. Unfortunately, due to the way Racket does lexical parsing, it is very hard to make handlers that are sperated at the file level. There are plenty of pieces in MinNo's handler system that can totally be seperated out into another file for better organization, but this is not true for all of it. If a perticular handler needs to call the directory funtion on some sub-section of the AST is handling, it

needs to be in the same file as the directory. If you tried to sperate it, there would need to be a cyclical dependancy of the two files. Racket does not (and aboslutely should not) allow cyclical imports and therefore makes it impossible to seperate trampolining functions such as certain handlers and the handler directory.

What is Left to Be Done:

Targeting Closer to the Chip

Currently MinNo targets the Arduino programming language. A next step would be to target GCC compitable C and AVR assembly directly.

This would be more verbose code for sure, but the speed increases and system controllability would be greatly increased. A major benefit would also be the ability to expand support for more embedded platforms. Targeting Arduino restricts me to boards within Arduino or it's community's interest to support. Boards like the Beagle Bone and the Raspberry Pi are outside of this list. Targeting GCC or LLVM (when the support for AVR is mainlined) will allow faster resulting code and wider target platform support.

[TALK ABOUT THE LLVM]

Creating an Ecosystem & Library

MinNo can do plenty of common tasks on the Arduino, but it's current library of keywords and functions is sparse. In the interestof creating a useful language, a standard library needs to be properly fleshed out. String operations, mass pin manipulation, bit-shifting abilities, encoding/decoding function, and more. All of this needs to be included to make MinNo a robust platform for embedded development.

It also needs to take advantage of the thousands of programming hours that have already been poured into the platform. Interoperability with C code directly in MinNo files is on the short list of features to tackle next.

Their are far too many libraries for Arduino add-ons, such as sheilds, to reimlpement from scratch.

In managing the library, C code, and all of the chips to be targeted, better tooling needs to be developed. The rise of languages like Rust and Scala can be attributed solely to their innovation as languages, but that would be missing a key point that other languages (like D and Julia) neglected: tooling. SBT (Scala's build tool) and Cargo (Rust's version of SBT) helped root programmers in a central conversations of the respective languages. They allowed for immediately productive and understandable project structures. These common architectures allowed code to easily connect and communicate. MinNo needs to make a similar move into creating an entry point to allow coders to share what they make.

Opinions On The Ecosystem: Racket, Raco, and General House Keeping

Package Management & Tooling

With Racket's niche being a platform to build more languages, they needed to have some way to share the tools that programmers produce so no one has ot spend time reinventing the wheel. Therefore, Dr.Racket has a very intuitive graphical package manager built on top of their command line tool **raco**.

pkgs.racket-lang.org [LINK THIS] lists, with all the aesthetic mindfulness of Dr.Racket, the packages available though **raco**. It contains tags, links to package documentation, and checksums (if you are into that).

There is a common moment of anxiety in developement when a language or library releases an upgrde while you are using the now-out-of-date version. Some projects are stuck in a version lock (or rewrite) due to their reliance on libraries built in a previous version (looking at you, Python 2[CITE]). While writing the MinNo Compiler, Racket came out with not 1, but 2 language releases.

These were minor version changes (6.7 & 6.8), so I thought about not jumping versions, but after reading their change-logs, I knew I had to. 6.7 greatly increased performance of strings in the language.[CITE] 6.8 improves optimization of equality statements.[CITE] There make up the vast majority of operations and data handled by a compiler; it would be a crime to not utilize the upgrades.

Of course I had that moment of panic when I downloaded the newest version of Racket. The raw terror of some feature of the Lex library not working and requiring me to 1. revert to the previous version or 2. rewrite some not trivial part of the program. After installation was finish, migrating my dependancies was next on the list.

In the graphical version of the Racket Package Manager, there is a very handy migration tool from previous language installations. It just checks your previous installation of racket for downloaded libraries and will automatically grab them from raco for the new installation. Once BRAG was migrated and updated, I decided to test how much a had broken my code base.

```
module: identifier already imported from a different
source
whitespace
parser-tools/lex
brag/support
```

It seemed that one of the two libraries decided to now include the term "whitespace" which was already part of the other library, creating a naming conflict. After some very brief searching, some *import prefixing* (answer: http://stackoverflow.com/questions/17894875/overlapping-module-imports-in-racket) did the trick. After that, the upgrade was successful & the compiler felt much snappier then before (though I have no way of saying this objectively, since I never ran a numeric test).

One very surprising thing was that when I upgraded Racket, all of my preferences and settings carried over despite doing a "fresh" install. Binds, color schemes, and everything else. As someone who is color blind, not being bogged down for hours figuring out a color scheme that actually helps me was a really nice surprise.

Though I don't have a great deal of praise of Dr.Racket (I don't have deep criticism either, just a mild understanding that it is around and has a particular use), I have grown very accustomed to using it for Racket development.

All in all, it is not as comprehensive of a tooling infrastructure as **Cargo** or dynamically connected as something like **npm**/**bower** but there is something to be said for it's simplicity.

Feelings about Where the Language is Going

Racket is unfortunately held back by it's view as a tool for education and the lack of functional utility that it's only development environment, Dr.Racket, exudes. When talking to a fellow college student during an internship, I mentioned that I used Racket. He looked at me with some confusion. "Racket, we used that in high school for the intro programming course. It was terrible."[CITE] Racket was initially created as an education tool. There is a huge amount of resources poured into it to do as much. With the state of the default development environment, you are constantly affronted with what looks like software meant to educate you, not to do serious development. Being more open and accommodating with other setups for coding in Racket is absolutely required to attract & keep new developers.

It is a surprise that Racket hasn't broken out of their educational roots in regards to Dr.Racket like the rest of the language has. Though education is still a huge part of Racket, the language platform discussion has really taken off.[CITE] It's reputation for a systems scripting language is also getting some recognition as well.[CITE] The language from an inner

mechanics level has evolved greatly even since I started using it (Spring, 2013). But by far one of the most interesting footholds I have witnessed Racket take is in game scripting.[CITE] You can find talks about 3d rendering systems and game engines at quite a few different Racket conventions, this is not surprising.[CITE] But at GDC, one of the largest graphics and game conventions currently the company Naughty Dog has given multiple lectures about their use of LISP languages and most recently their utalization of Racket.[CITE] Something about their process just seems to root around using LISP (considering they went through the trouble of maintaining their own sub language for a number of years).[CITE] For the last new (award winning) titles they have released, Racket has been their choice in scripting languages. [CITE]

The movements into fields are far apart as language development and gaming scripting have done a great deal to lift Racket out of it's PLT reputation. If it could shed away from the last major reminder of those days, it's deep attachment to Dr.Racket, I think it could really be up there with Clojure as the new face of LISP.

Bibliography: