Clip: A LISP-like "Resyntaxing" of the C Programming Language

Marley Adair

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh

2025

Abstract

In this report, I discuss the design of the Clip language, a LISP-like programming language intended to be compiled to C, as well as an implementation of a compiler for that language. The Clip language is designed to provide an environment familiar to LISP programmers, and some LISP features that are not present in C, in a langauge that can also utilise C's power and compatibility. The produced implementation is able to convert basic Clip programs into functional C code that can be compiled and run using existing C compilers. This report covers the design an dimplementation process, considerations for the language, and outlines future plans for the second year of the project.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Marley Adair*)

Table of Contents

Chapter 1

Introduction

Despite being one of the oldest programming languages in existance, LISP still finds use today. Its simple syntax and grammar makes it ideal for functional programming and metaprogramming, which has caused it to be used as a common tool in data science. [1] However, the nature of its lack of distinction between code and data means that there is a performance cost to executing it, compared to compiled languages that can be converted to native instructions in advance, allowing for very efficient execution.

C is a low-level programming language known for its high performance, as well as many other things. It has been in common use for over 50 years [9], and has compilers available for an incredibly large range of hardware. Additionally, its widespread use has effected a large amount of research and development into optimising the output of compilers for a wide variety of tasks. Unlike LISP, C's complex grammar and imperative nature, while perhaps more intuative to the average person, makes it difficult to use in metaprogramming and functional programming based tasks.

This project presents Clip, a LISP-like programming language that is "compiled" to C code, allowing for the structure of LISP to be used with the efficiency of C. This report covers the design of Clip as a language, as well as the implementation of a compiler capable of converting Clip source files into C source files, which can then be compiled to native code and executed.

1.1 "Resyntaxing"

The project brief, and the title of this report, say that Clip is a "resyntaxing" of C. However, most of the report and work treats Clip as if it is simply a programming language that happens to have a compiler that targets C. So what does "resyntaxing" actually mean in this case?

As far as I can tell, "resyntaxing" isn't a common term, and has no standard definition. So instead, I'll present how I interpreted it, and what that means for the design of Clip.

I do not consider having a one-to-one mapping of syntactic structures between Clip and C a requirement. While that may be what "resyntaxing" literally suggests, it would be impractical to implement, and also likely not that useful. Instead, I am treating "resyntaxing" to mean that every piece of generated C code must correspond to something in the original Clip source. This means that Clip cannot output C code that is not related to the input. As a result, Clip is prevented from having any form of initialization code, or additional checks, or other features that a full programming language might have. This has the benefit of ensuring that additions added by Clip can't accidentally slow down the final code in unexpected ways.

Some of the future plans being considered would violate this constraint. How strictly Clip remains a "resyntaxing" will likely depend on the project's direction in part two, and discussions with my supervisor.

1.2 Motivation

C is a very widely used and supported language that has been optimised over its long life to become one of the most powerful ways of writing high-performance code. Its human-readable (at least compared to native instructions) syntax and grammar makes it ideal for programmers to quickly create programs in an intuative manner. However, there are tasks for which C is not well suited, which Clip aims to handle better.

1.2.1 ABI

Almost every platform's ABI is used by C. C has become the standard for binary naming and typing. By generating and compiling C code, Clip can easily interface with C libraries and C-style ABIs. It can also be used to generate such libraries, which can then be used by anything that can call C functions. This makes it easy to integrate with existing systems.

1.2.2 Metaprogramming

Metaprogramming is a technique where computer programs operate on other code as their data, either as a step of compilation in a single program, or as a full process that occurs before the final build. C allows for limited source generation during compilation via its preprocessor, which can perform operations such as conditionally including sections of code, embedding sorce files inside of others, and

some basic text replacement. However, anything beyond this must be handled by a separate program. Furthermore, C's syntax and grammar are designed to be read and written by computers, which makes performing operations on C source files a complex process, as the code must be parsed and transformed. The C standard does not provide a standard way of representing C code as data other than raw text, which means that various systems that operate on C code will likely be incompatible with each other.

Rust is an example of a programming language with much more capable metaprogramming tools, also in the form of macros. Unlike C, Rust macros can perform arbitrary transforms of the token stream, allowing much more flexibility, as well as the ability to interact with other systems for metaprogramming during compilation. This makes Rust macros incredibly powerful tools when used correctly. However, like C, Rust has a complex grammar intended for humans to write. While Rust does provide libraries to help with parsing and transforming the language, the resulting data structures are very complex due to the flexibility of the structure of Rust code, and the large number of types of syntax tree node present in the language.

Unlike those two languages, LISP is very friendly to metaprogramming by design. LISP code is represented using S-expressions, which are an explicit representation of the program's syntax tree. Furthermore, the language has a very limited number of node types, normally between 3 and 5, but this depends on the dialect. This makes it easy to operate on LISP programs, and LISP code will often make use of the ability to generate new executable S-expressions at runtime. While this is incredibly powerful for metaprogramming, the ease of which code can be changed makes it hard for compilers to optimise the resulting binary, and many LISPs are simply interpreted.

Clip aims to provide LISP's ease of metaprogramming at compile time instead of runtime, allowing for powerful macros while still being friendly to compiler optimisations. While it is not a LISP itself, it still uses LISP's S-expressions to represent its code, which gives it the benefit of being easy to perform transforms on the syntax tree. However, once compiled, its structure is set, allowing for the compiler to optimise without needing to consider the case that the code may be modified while the program is running.

1.2.3 List Processing and Higher Order Functions

While the current design and implementation of Clip do not contain anything to improve C's handling of these topics, they are planned to be considered in the future as additional goals for the language. See Section 8.3. Providing proper support for this will allow Clip to fulfil its role as a fast and efficient LISP replacement in many more areas. LISP has a lot of support for this as part of its design,

and if Clip is to be used in the same fields, it needs to be able to support the same use cases.

1.2.4 Optimisation

Optimising compiler output is an incredibly difficult task. The generation of efficient code is a large and active area of research. Due to its ubiquity in modern systems, a lot of research and development is done on optimising the output of C compilers.

By utilizing C as a code generation *target* instead, Clip can benefit from this research by using the C compiler to perform optimisation passes instead of needing to implement one itself. An additional benefit to this is that as C compilers improve over time, Clip will also gain the benefits of those developments, without needing to be updated.

1.3 Report Structure

This report covers the progress I made this year on the development of the design and implementation of Clip.

Chapter 2 discusses topics relevant to the report, and how they affect Clip. It covers the building blocks that make Clip possible, and some terminology from them that will be useful when reading the remainder of the report.

Chapter 3 discusses the syntax and grammar of Clip, and how the Clip compiler converts raw source code into an abstract internal representation.

Chapter 4 discusses semantics, how Clip assigns meaning to the nodes within its internal representation of the program, and how that meaning is converted into C code.

Chapter 5 discusses how Clip handles data types, how that interacts with C, and the problems that arise because of this.

Chapter 6 discusses design and implementation decisions that were important to the project or interesting, but that do not fall under any of the prior chapters.

Chapter 7 presents the interface of the implemented compiler.

Chapter 8 discusses future work for the project, and presents an outline of the work for year 2.

Chapter 9 discusses related work by other people.

Chapter 2

Background

2.1 LISP

LISP (or Lisp) was one of the first high-level programming languages created. It is based on a simple syntax made of nested lists of expressions, which may be terminals (symbols and literals) or other lists. Nowadays, there are many LISPs, each derived in some way from the original, keeping the same basic principles. A small example program in Common LISP is shown here:

```
(defun hello ()
    (print "Hello, world!"))

(hello)
```

LISP is notable for its "code is data" approach. In LISP, the structures above are first read as object literals, made of lists and values, before being evaluated by the interpreter. This makes it possible to write code that creates or modifies executable structures on the fly.

Clip is not a LISP. However, it does share many of the same goals. It also has a very similar syntax and grammar. As a result, it is often useful to talk about Clip's structure in the same way as LISP's.

2.1.1 Terminology

The terminology of LISP is useful also when talking about Clip. LISPs "objects" when represented as above are called S-expressions. In the case of Clip, they come in three types:

- Lists - Represented with parentheses surrounding zero or more items (other S-expressions.) For example, `(1 2 3)`
- Symbols - Represented by a word consisting of any characters that are not whitespace or otherwise special. For example, `hello`
- String Literals - Represented by text surrounded by double quotes. For example, `"Hello, world!"`

These three types of S-expression will be used when referring to expressions in Clip. Later chapters will go into more detal about how exactly each type of expression is defined, and their purpose in a program.

2.2 C

C is a programming language originally created by Dennis Ritchie. It has become one of the most used programming languages of all time, finding use in almost all areas of programming due to its power and flexibility. Unlike LISP, C is a compiled, statically typed language. This has many benefits in terms of efficiency, as compilation allows the language to be run directly as native code rather than through the extra layer of an interpreter, and its static typing allows for much stronger optimisations to be performed at compile time.

Clip targets C as its output language. As a result, many of these properties of C affect and restrain how Clip can work. However, their benefits apply to Clip too.

2.3 Compilers

A compiler is a program that converts a program written in a higher-level language and converts it to some lower-level representation, often executable binary code, or some form of object code or bytecode. Clip, by design, is a compiled language. Unlike traditional compiled languages, Clip's compiler outputs C code, another high-level language. However, its structure is still very similar to that of a traditional compiler.

Most modern compilers work in passes. Each pass transforms the input in some way, until it is eventually converted to its target form. The Clip compiler uses three passes, a lexer, a parser, and a code generator. These three passes are about the minimum required for a modern compiler. Most modern compielrs have additional passes that handle tasks such as optimisation. As the Clip compiler outputs C code instead of a low-level binary however, such tasks can be left to the C compiler instead. The design and implementation of these passes are covered in this report.

Chapter 3

Syntax, Grammar, and Parsing

The main goals of Clip's design are fairly similar to that of any LISP's: have a simple syntax and grammar to facilitate metaprogramming, and provide a familiar environment for users of LISPs and other LISP-like languages. As Clip is not a LISP, it does not concern itself with maintaining any form of compatibility with an existing language. It was instead designed to use a syntax as simple as possible. Clip has only three types of expression, symbols, strings, and lists. This is similar to many LISPs. Unlike some LISPs, Clip does not make any explicit distinction between symbols and numeric literals. Instead it opts to parse them all as symbols, and to define any symbol matching a numeric form as a constant number as part of code generation. This allows for lexing and parsing, but also any code emission, to be incredibly simple (compared to most other languages.)

The current design of Clip assumes that all lists start with a symbol, which represents the function that the list invokes when run. This is under consideration for being changed in the future, in order to allow for more intuative use of constructs like first class functions, or the use of lists to represent data values instead of code.

The full grammar of Clip in extended Backus–Naur form is available in Appendix A.

3.1 Implementation

The parsing of Clip is done in two passes: a lexer that converts the raw program text to a stream of tokens, and a parser which assembles those into program trees.

3.1.1 Lexing

Lexing is the process of converting the input code, in its original textual form, into a stream of "tokens" which each have a semantic or grammatical meaning. The simple nature of Clip's syntax means that there are only four types of tokens

that the lexer outputs, representing "(", ")", symbols, and strings. The lexer converts an input buffer of text into these tokens simply by matching the head of the input code against patterns representing the valid forms of those tokens, either the single characters of "(" and ")", or the longer paterns that represent valid symbols and string literals. The lexer also matches blocks of whitespace at the head of the input, however it does not generate a token for them. Once a match is found, the lexer returns the appropriate token, and moves the input head to after the pattern. In the case of string literals, some extra processing is run to handle backslash escapes, so that the token object contains the actual contents of the represented string, instead of exactly what was written in the source file.

As an example in practice, here is a simple "hello world" program, colour coded to show how the lexer splits it into tokens. Note how the whitespace is not a part of any token.

```
(: main)
(defn main
    (puts "Hello, world!"))
```

Figure 3.1: A simple Clip program, split into tokens. Yellow tokens are symbols, cyan tokens are string literals, and red and green tokens are open and close parentheses.

3.1.2 Parsing

Parsing is the process of converting the stream of tokens produced by the lexer into an object that is structured like the actual program. There are several common algorhithms for doing this. Clip uses a recursive descent parser. Clip's explicit tree-based syntax makes this an ideal algorithm, as the S-expressions Clip uses are able to contain other S-expressions to an arbitrary depth. The actual parsing algorithm in the case of Clip is incredibly simple: A token is read from the stream; if it is a symbol or string literal, an equivalent node is output; if it is a "(", then the parser repeatedly calls itself until a ")" is matched, storing the list of outputs it receives, then itself returning them as a list.

A Clip program may consist of multiple root-level S-expressions. Therefore, the parser is called repeatedly on the token stream until it is exhausted, and the resulting list of expressions is used as the parsed program.

Consider again the example program in Figure 3.1. By performing this algorithm, it is converted into the following two trees.

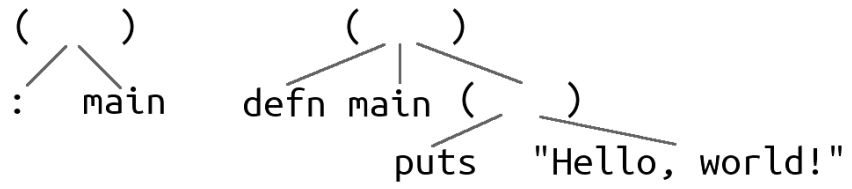


Figure 3.2: A simple Clip program, parsed into its "abstract syntax tree" form.

Chapter 4

Semantics and Code Translation/Generation

After parsing, a Clip program has been converted into a list of trees, which is the representation used when actually creating the final output. However, those trees have no actual meaning yet. Like LISP, the first item in each list used in code is a function of some kind that specifies how the rest of the list should be evaluated. In the case of Clip, this "evaluation" returns C code in some form, that is used by the levels of the tree above it, until an entire tree is evaluated into a section of a C program. In Clip, symbols actually work in a similar way, each returning some piece of C source when evaluated. These functions do not (necessarily) represent actual functions in C.

Examples of functions a symbol at the start of a list could represent are:

- Actual C functions: `(puts "Hello, world!")` $\Longrightarrow$ `puts("Hello, world!")`
- Operators: `(+ 1 2 3)` $\Longrightarrow$ `1 + 2 + 3`
- Control flow: `(while (> i 0) (set i (- i 1)))` $\Longrightarrow$ `while (i > 0) {i = i - 1;}`
- Metaprogramming: `(deftype string (pointer (const char)))` $\Longrightarrow$ *[Nothing]*

When the code generator encounters a node to evaluate, it needs to figure out what that node should generate. For string literals, this is simple; a Clip string literal always represents its equivalent string literal in C. For symbols and lists however, the generator needs to look up what that specific symbol or list represents. That lookup is done using the current scope.

4.1 Scopes

Like most modern programming languages, Clip has scopes. A scope provides information on the meaning of symbols, and acts to isolate definitions in one part of the code from another, allowing multiple sections to use the smae names without conflict. Like the language itself, scopes exist in a tree structure, with a root "global" scope that contains definitions for all the built in symbols. When a new scope is created in the code, for example by defining a function body, or creating a block, it inherits its parent's definitions. However, any new definition made within that scope is forgotten once it ends, meaning a scope cannot cause conflicts outside of its area of use. While C also uses scopes in a similar way, scopes in Clip do not necessarily map directly onto scopes in the resulting C program.

While in languages such as C, scopes store user definitions of things such as functions and variables, and maybe even types, scopes in Clip define everything that occurs during the code generation phase of the compiler. The global scope in Clip contains definitions for how to generate everything in the language. This contains both traditional definitions such as variables and functions, as well as things like operators, control flow, and metaprogramming. Some of these definitions simply produce some output C code, such as operators, however others such as definitions need to change the current scope itself, adding or editing entries for when they are used later in that scope, or in its children.

Consider a symbol like `def` to show the power of this. `def` is used to define a variable in the current scope. A simple use would be something like `(def int i)`, which creates a variable called `i` that stores an integer. This does produce C code as output, likely something similar to `int i;`. However, it also alters the current scope, adding an entry that tells the code generator that the symbol `i` refers to the C variable `i`. This may not seem important in this case, where the variable name in both languages is the same, however there are some times when this cannot be the case. Clip offers far more choice in symbol names that C offers in identifier names. While C's identifiers are restricted to alphanumeric characters and underscores, Clip allows for any character that is not whitespace, parenthesis, or double quotes to be used in a symbol. As such, there are cases where the C variable must take on a different name that the Clip one. This necessitates remembering what each Clip variable is called in C, and this information is stored in the scope the variable is defined in.

4.1.1 Shadowing

Shadowing refers to redefining a symbol to represent something new, either in the same scope as its previous definition, or in a child scope. Clip and C both support shadowing. In C, a variable can be defined in an inner scope with the same name

as a variable in a parent scope, and the new definition is used for the lifespan of that scope and all its children. After the scope ends, the original definition takes over again.

Clip also allows this. However, unlike C, Clip allows a variable (or other symbol) to be redefined in the same scope as its previous definition. This approach to shadowing as pros and cons. It allows more freedom in veriable definition. It can also be used to prevent a variable from being accessed after some point in the code. However, it could also cause confusion by having a single name in a single scope refer to multiple types. If a programmer accidentally redefines a variable, it could affect their program in unexpected ways.

There are many programming languages that use both these systems. Which one Clip uses is not particularly important to its design. However, the fact that Clip is able to deviate from C in this regard shows the utility of having a scope system that is independent of C's.

4.2 Generation Environment

As talked about in the section above, scope is passed down recursively during the generation process. While scopes cover most of the information required to generate code, there is a further piece of information that helps with code generation: the current context.

Unlike Clip and its single non-terminal grammatical structure, C has much more varied syntax. For example, C uses inflix operators (e.g. `1 + 2`), versus Clip's prefix operators (e.g. `(+ 1 2)`). Where multiple inflix operators occur together, C needs to determine what the resulting evaluation tree looks like. It does this via a precedence table based on common mathematical notation. In order to have an evaluation different to this, some of the subexpressions must be wrapped in parentheses.

When generating C code, the Clip compiler needs to determine if a subexpression should be parenthesised. This requires having knowledge of the subexpression and the expression which it is a part of at the same time, so that the precedence of the relevant operators can be compared. To solve this, the code generator passes to each subexpression the minimum precedence that it can safely generate before causing the C code to represent the incorrect tree. The inner expression can then check to see if it satisfies that requirement, and if it does not, it generates parentheses around its output.

In the case of operators, this is not technically required. It would be valid for Clip

to simply parenthesise every expression. However, there are other cases where this is more useful. For example, using a different structure when an expression is a full statement, versus a subexpression in some larger construct, such as choosing between the `? :` operator and an `if else` statement. This distinction between "statements" and "expressions" made by C and not Clip also poses a major hurdle in code generation.

4.3 Expressions and Statements

In Clip, expressions that make up the body of functions, where actual code execution occurs, evaluate to some value which can then be used by their parent expression. In C, while some code (which C calls "expressions") do this, other code (which C calls "statements") do not. Any C expression can be used as a statement, however the reverse is not the case. As a result, Clip expressions that use the results of expressions that can only be represented in C using statements need to be restructured. A common example of this is control flow. Most control flow in C is available only as a statement. This means that Clip expressions which use such constructs must be split into multiple C statements, where the "result" of one must be saved to some temporary variable where it can be used in the next.

4.4 Implementing Code Generation in Practice

There were two attempts at this during the project. The first approach used a naïve recursive system, that ran into problems related to what was discussed in the previous section. The second approach is more complicated and was designed with the problems in mind, however was not completed within the time of the project's part one. Finishing that implementation will be the first major step of year two of the project.

4.4.1 The Naïve Approach

This approach, which was implemented during the first year of the project, recusively evaluates each S-expression to a string of C code with no extra data. Each node concatinates the result of its children in some way, which eventually produces a final full C program. While this works for some programs, the requirement for each node's C code to be located together prohibits splitting statements and talked about in the previous section. At first having some extra return value to indicate what should be a previous statement was attempted, however this was not viable or maintainable at the scale required.

4.4.2 The New Approach

Instead, a new system was designed to handle this. This system allows for a node to specify a list of statements that need to be run before the final expression is evaluated. This is recursive; those statements can also have requirements. Furthermore, a node can place the statements required by its children into a control flow block. This is important, as some nodes may evaluate their children repeatedly (`while`) or conditionally (`when`). If these statements have side effects, it is important they are run only when that value should be calculated.

Although I have begun implementing this system, I was unable to finish it before the deadline for this year. The current build of Clip, therefore, still uses the naïve system, and simply fails if a construct like the ones this system is designed to handle occurs. However, the design of the new system is largely complete. The practicalities of implementing it may cause some shift in the design, which will be documented in part two of the project next year.

Chapter 5

Data Types

As a "resyntaxing" of C, Clip uses the same data types as C does. Like C, and unlike most LISPs, Clip is statically typed, and requires explicit typing of variables and functions. The Clip compiler does not itself check type correctness, at least in its current implementation. This is instead left to the C compiler. However, there are some operations that Clip must handle types itself for. When splitting statements to allow for complex control flow in expressions, as talked about in Section 4.3, Clip must know the types of intermediate results, in order to store them in temporary variables. To do this, Clip must implement type inference.

5.1 Type Inference

Type Inference is the process of determining what type an expression evaluates to, without it being explicitly stated. In order to accomplish this, Clip adds another return value to each expression, indicating the C type that it produces, if any. How complicated this is to determine for an expression depends on what C that expression generates.

For some expressions, such as variables and function calls, determining type is trivial; it is explicitly given in the symbol's definition. Other expressions, such as conditionals and blocks, have a return type determined by their final included expression. Then there are operators.

5.1.1 The Problem with Operators

C defines the resulting type of numeric operator as a function of the types of the operator's arguments. This function is designed to give fairly intuative results for the most part; if the arguments are the same type, that is also the output type. If one of the arguments is floating point, and the other is an integer type, the result is floating point. If both types are integers with the same signedness but different widths, the larger width type is used. These all seem sensible descisions, which give reasonable output types. However, if the argument types are two dif-

ferent integer types, with opposing signedness, C defines the return type as follows:

> "If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
>
> Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type."
>
> (ISO/IEC 9899 C Specification)

Here, the returned type is dependent on if one type can fully represent the other. C, however, does not define the range of many of its integer types, instead defining minimum values for them, and stating they must be non-strictly increasing. This means that an operator, for example, that takes an `unsigned short` and an `int` may either return an `int` or an `unsigned int`.

This is a problem, as those two types have very different behaviours in some situations, and having a type change due to some difference later in an expression that causes Clip to split a statement into multiple C statements would be very unintuative and difficult to debug. Since Clip is just outputting C code, and knows nothing of the C compiler itself, the descision about what type to use here must be made by the C compiler. C requires types to be explicit. While they cannot be determined by other C code, they can be changed by the preprocessor, which allows text replacement. Importantly, it allows conditional text replacement based on compile-type constants. C also provides an operator to determine the width of a type, called `sizeof`, as a compile-type constant. Unfortunately, this operator cannot be use dby the preprocessor. In fact, the preprocessor is prohibited from inferring information about type sizes in any way. This means there is no way to determine what type certain intermediate values should be.

The current implementation of Clip assumes that the integer types *strictly* increase in width. This is the behaviour used by most modern C copmilers. Further research on this problem will be a consideration for year two of the project, however it seems unlikely that a solution exists based on the current design. It may be that the design is changed in the future to allow Clip to determine some properties of the C compiler it is targetting.

5.2 The Never Type

The ability to include control flow operations within expressions in Clip creates some interesting situations. Consider the following code:

```
(if flag
    1
else
    (break)
)
```

The intent of this code seems clear: if `flag` is set, return 1, otherwise break the current loop. However, this creates an interesting issue with types. `if` must require the types of both branches to be compatible, as it must have a static type to return from itself. The first branch is clearly an `int`, but what is the type of `break`? It doesn't make sense to assign a type to `break` at all, as it can never return anyway; it always triggers a branch to somewhere else.

To handle this, Clip borrows a trick from Rust: have a type to represent expressions that can never return a value. Clip calls this type `never`. `never` can be implicitly cast to any other type, allowing it to be used as a branch in a conditional where all types must match.

Chapter 6

Design and Implementation Decisions

6.1 Compiler Programming Language

As a language cannot start off bootstrapped, a decision had to be made as to what the original compiler should be written in. For this project, I went with Rust. Rust is a modern programming language known for its speed and safety. When developing a complex piece of software, such as a compiler, Rust provides useful compile-time checks for common mistakes in memory management. This is revy useful for catching potential problems early.

Another reason to pick Rust is its type system. When developing Clip, it will naturally be required to have some way of representing S-expressions, as they are the main structural part of a Clip program. Rusts "enums" are a useful feature for this purpose. The act like what most other languages that have a similar feature call "tagged unions", and allow for the creation of a type that is actually any one of some set of types. This is useful for handling S-expressions, which can be three types (symbols, lists, and strings.)

6.2 The : symbol

Something in Clip that may be unintuative to both LISP and C programmers is the : symbol. This symbol is sued to define the signature of a function, but not its body. It is required that every function has a signature defined using : before its body is defined.

This furfils two purposes:

1. It allows the programmer to specify the types of the function's parameters cleanly without creating any ambiguity as to where parameters end and

body code begins. If this was done alongside the function body, either they would have to be part of the same list, with an unclear divide, or they would have to each be contained in nested lists, which unnecessarily increases the depth of the code.

2. It allows circular dependencies in functions. Functions must know what types the values inside them are when they are declared. If two functions wish to call each other, they both must know each others types. If type signatures are defined alongside the body, this cannot be done, as one function must be declared first. : allows for the functions' types to be defined before either body, allowing this circular calling.

Chapter 7

Compiler Interface

The currently implemented interface of the compiler is incredibly basic. Once run, the compiler reads Clip code from standard in, and writes C code to standard out. This is not the intended final interface.

Due to time constraints, priority was given to implementing the compiler's functionality, rather than providing a clean interface. Designing and providing a proper interface with relevant settings will be done during part 2 of the project.

The current simple interface is however useful for the creation of debugging tools, such as the live viewer tool. This program allows the user to view the changes to the outputted C code in real time while they write in Clip.

Chapter 8

Future Work

Throughout this report, I have commented on many features and fixes I was not able to implement within the time of the project. These are all planned for year two instead. After any minor issues are resolved, I can start looking at major next steps.

The first major task of the second year of the project will be finishing the new code generator implementation discussed in Section 4.4.2. This will allow the creation of much neater and more complex code. Once that has been done, there are several areas in which the design and implementation of Clip could be improved.

8.1 Metaprogramming tools

For as much as Clip's design is made to allow for easy metaprogramming, Clip itself is lacking in tooling to use it. Providing systems to manipulate Clip code within Clip will both help with showing the power of Clip as a language, and help with using Clip to write more complex programs. The major challenges this involves are creating a representation of Clip code within Clip, and evaluating Clip code at compile time so its result can be used in the final program.

8.2 More Types

In order to benefit most from the LISP-like design of Clip, I am considering the posibility of implementing additional types as part of a Clip standard library. These would be used for common LISP tasks, such as those relating to list processing and first order function handling. If Clip is to be useful in data science, having access to such tools would have a major benefit, as it would make the implementation and use of common algoritms much easier.

8.3 First Order Functions

While C has some support for first order functions in the form of function pointers, I think having more support would aid Clip in its uses in many situations. LISP uses first order functions a lot, and having more support in Clip for structures such as closures would be useful for situations such as in data science where LISP is used for its good support for various algorithms.

8.4 Bootstrapping

A potentially final goal of Clip would be to bootstrap the compiler, rewriting it entirely in Clip. This is likely a stretch goal; the Clip compiler has taken a significant amount of work to produce already, and recreating that in a language with less tools available will be tough. However, doing this would show Clip's power as a programming language well.

Chapter 9

Related Work

C and LISP are both very old and popular languages. As a result, there have been many similar and related projects by other people.

9.1 LISP extensions

LISP as a language has had a huge amount of variants created ofer its lifespan. Some of these have similar ideas to Clip.

9.1.1 Typed Racket

Typed Racket is a variant of Racket LISP that introduces types and type checks. [10] It uses gradual typing [3], and is designed to be an extension of Racket LISP. As a result, it needs to introduce more syntax to handle these extra features, instead of that syntax being naturally integrated with the rest of its design. It also uses its type system as a way to perform more static checking, not as a tool for optimisation.

Compared to this, Clip was designed with static typing in mind, so has more integration of types in its base syntax. Clip also uses static typing as a way to easily compile to C, allowing it to gain C's optimisation benefits from knowing value types at compile time.

9.1.2 Common Lisp

Although Common Lisp is dynamically typed, it does provide some forms of static type checking when compiled. [4] This allows for the compiler to produce some optimisations that would not otherwise be possible. For this to occur, the source code must contain type annotations, which Common Lisp doesn't require. Furthermore, if type constraints are violated, the compiler simply emits a warning

and continues to build.

9.2 Compilation to C

Clip is far from alone in treating C as a compile target. The benefits of using C this way also apply to many other languages, to the point where C has become known as a "portable assembly". [5]

Many languages, such as Haxe [8], Mercury [7], and Vala [**?**], target C for compilation. They cite various reasons such as portability and speed [6] for this choice.

9.2.1 C–

C– is a programming language that is intended to be used as a compiler target. It is designed to mitigate some issues that occur when compiling to C, while keeping the benefits of protability and speed. [5] This shows significantly improved results in some cases. [**?**]

While C– helps in many cases, especially for languages that are not related to C themselves, it is not a better choice for Clip. Clip by design is semantically similar to C, and therefore there is no significant cost caused by the translation of code between styles, or the implementation of features not present in C such as garbage collection.

9.3 Compiling LISP to C: The Embeddable Common Lisp

Embeddable Common Lisp is a tool for compiling (a subset of) Common Lisp programs to C. [2] Much like Clip, it aimed to get efficiency and portability out of this target. However, Common Lisp is a dynamically typed language, and Embeddable Common Lisp handles this by performing runtime type checking, and some level of interpretation. This reduces its performance benefits, versus a tool like Clip which is designed for targetting C.

Bibliography

[1] The top programming languages in ai. *APRO*.

[2] Guiseppe Attardi. The embeddable common lisp. In *Papers of the fourth international conference on LISP users and vendors*, pages 30–41, 1994.

[3] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound gradual typing: Only mostly dead. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[4] Martin Cracauer. Static type checking in the programmable programming language (lisp), 2019.

[5] Simon Peyton Jones, Thomas Nordin, and Dino Oliva. C–: A portable assembly language. *Lecture notes in computer science*, pages 1–19, 1998.

[6] The Mercury Project. The mercury project - motivation.

[7] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1-3):17–64, 1996.

[8] Stepan Stepasyuk and Yavor Paunov. Evaluating the haxe programming language-performance comparison between haxe and platform-specific languages. 2015.

[9] Tiobe. Tiobe index for april 2020. 2020.

[10] Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. The typed racket guide, 2014.

Appendix A

Clip's Grammar

CHARACTER and WHITESPACE are defined as their names suggest. Clip accepts all valid utf-8 as input, so the actual definitions are incredibly long.

```
SYMBOL_CHARACTER = CHARACTER - "(" - ")" - '"' - WHITESPACE;
SYMBOL = SYMBOL_CHARACTER, { SYMBOL_CHARACTER };

HEX_DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
          | "8" | "9" | "a" | "b" | "c" | "d" | "e" | "f"
          | "A" | "B" | "C" | "D" | "E" | "F";
STRING_LITERAL_PART = CHARACTER - "\" - '"'
                    | '\"'
                    | "\\"
                    | "\0"
                    | "\t"
                    | "\n"
                    | "\r"
                    | "\'"
                    | "\x",  HEX_DIGIT, HEX_DIGIT;
STRING_LITERAL = '"', { STRING_LITERAL_PART }, '"';

LIST = "(", { WHITESPACE }, SYMBOL, { WHITESPACE },
       { S_EXPR, { WHITESPACE } }, ")";

S_EXPR = SYMBOL | STRING_LITERAL | LIST;
PROGRAM = { WHITESPACE }, { S_EXPR, { WHITESPACE } };
```

Improving Clip, A Lisp-like language that transpiles to C

Marley Adair

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2024

Abstract

This MInf part 2 project report discusses my enhancements in the design and implementation of the Clip programming language, a Lisp-like programming language intended to be compiled to C. The Clip language is designed to provide an environment familiar to Lisp programmers, and some Lisp features that are not present in C, in a langauge that can also utilise C's power and compatibility. The goal of this project was to improve the design of the Clip language to add more features in order to assist with the creation of complex programs. These features include C features that were missing from the original Clip design, as well as additional features from higher-level languages that provide ease of use when working with large or complex codebases. As well as design, the compiler implementation produced in the previous year was extended to include support for the design changes.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Marley Adair*)

Table of Contents

Chapter 1

Introduction

1.1 Clip

In the first year of this project, I designed and produced a basic reference implementation of a programming language called Clip, which aimed to provide a programming environment similar to those of Lisp languages. The language was designed to be transpiled to C, in order to gain the benefits of the long history of research that has gone into optimising C compilation. The design of the language was based on several common Lisps, such as Scheme, Clojure, and Racket. Syntactically, the language was minimal, containing only four types of token (symbols, strings, `(`, and `)`) and structurally, the language was designed to be similar to C, in order to make the best use of C's optimisations. This allowed for the creation of a subset of C programs, in a manner similar to writing out the program's abstract syntax tree as it would be undrestood by the C compiler. An example of such a program, compatible with the year 1 version of Clip, is shown below in Figure 1.1.

```
(include-c stdio.h)
(: main)
(defn main
    (printf "Hello, world!\n"))
```

(a) Clip code

```
#include <stdio.h>
void main();
void main() {
printf("Hello, world!\x0a");
}
```

(b) Equivalent C code

Figure 1.1: A simple Clip program compatible with the year 1 version of Clip, and its transpiled output.

1.2 Summary of Project

The design and implementation was extended to allow for more complex features. The simple grammar of the language allows for quick and flexible development of language features. These features go beyond what the C language is capable of,

showing that the project is able to expand on its target language. Furthermore, the addition of metaprogramming tools is powerful in allowing users to alter and define their environment to further extend the language's capabilities to meet their needs.

1.3 Report Structure

Chapter 2 of the report provides a brief overview of the progress achieved in the first year of the project, and summarised how the Clip implementation worked at that time.

Chapter 3 covers some concepts and prior work from other people that will be relevant later in the report.

Chapter 4 lists the improvements made to the design and implamantation of Clip during this year of the project. It is broken down into sections on the design and implementation of the following:

- Comments
- Complex expression forms
- Custom data types
- Parameterized typing
- Namespacing, libraries, and nulti-file programs
- The standard libraries of both Clip and C
- Closures
- Metaprogramming and macros

Chapter 5 evaluates the progress made on the project, and provides potential future directions should anyone build upon the work here in the future.

Chapter 2

Prior Work

This chapter gives an overview of the design an implementation of Clip done in the first year of the project. [1] For a more detailed coverage, please read the first-year report.

2.1 Design

The year one Clip defined a tool that converts code from the Clip language to the C programming language. The Clip language was designed based on Lisp. It is composed of three types of structure: symbols, strings, and lists. Clip source code is a series of s-expressions, which are text representations of the structures above. These were made as follows:

Symbols: Any string of non-whitespace, non-parenthesis, and non-quote characters. A symbol represents a name that can be bound to a meaning or value, such as a variable or function. Examples of symbols are:

```
hello    10    +    <*u*>
```

Strings: A quoted string of text. Can contain escape sequences. Strings are used when representing literal text, either for use as text within the program, or as filenames for libraries. Examples of strings are:

```
"hello"    ""    "a\nb\nc"    "\""
```

Lists: A parenthesised sequence of expressions, including other lists. The first expression must be a symbol. Lists are how program structure is defined. Examples of lists are:

```
(foo)    (a b c 10 "hi")    (x (y (1 0)) z)
```

The entire grammar of the language is composed of these expressions. Typically, the root expressions that make up a Clip source file are all lists, such as in the example in Figure 1.1. The root expressions mostly exist to provide definitions

(done here by the lists beginning with `:` and `defn`) while their contents contain the actual code.

Given such a source file, Clip will output a block of C code. This code must then be compiled with a C compiler (not provided) in order to produce an executable binary that can actually be run. Since C code is cross-platform, this means that Clip code is too, making the language compatible with most systems, assuming you have an appropriate C compiler.

2.2 Implementation

The actual implementation of Clip produced in year one runs in several stages.

2.2.1 Lexing

The first step is lexing. This is the process of converting a Clip program into a list of tokens, which are units of meaning that can't be subdivided. This works by scanning the input text stream and matching valid token patters to it. This step is also used to discard whitespace. An example of the results of this step is shown in Figure 2.1

```
(: main)
(defn main
    (puts "Hello, world!"))
```

Figure 2.1: A simple Clip program, split into tokens. Figure is ripped directly from the year 1 report.

2.2.2 Parsing

The next step of transpilation is parsing. In this step, the list of tokens output by the lexer is converted into a list of expressions. Clip uses a recursive descent parser. Clip's explicit tree-based syntax makes this an ideal algorithm, as the expressions Clip uses are able to contain other expressions to an arbitrary depth. The parser simply checks the next token available; if it's a symbol or string token, then a symbol or string expression is output. If it's an open parenthesis token, a list is created, and populated with expressions created by recursively calling the parser until a matching close parenthesis is found. The result is a list of trees, demonstrated again on the example program in Figure 2.2.

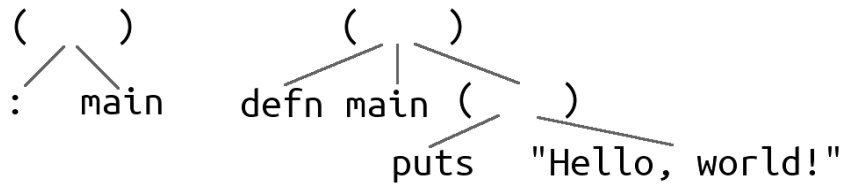


Figure 2.2: A simple Clip program, parsed into its "abstract syntax tree" form. Figure is ripped directly from the year 1 report.

2.2.3 Code Generation

Once parsing is complete, each expression is used to generate code in order. Every symbol is mapped to a pair of functions, one that generates code for when the

symbol is alone, and one that generates code for when the symbol is used to start a list. The second case will typically recursively call this generation process on the other expressions in the list.

In the example program, code generation is first invoked on `:`. This generates the C code `void main();`. It does not invoke code generation on the symbol `main`.

Next, code generation is invoked on `defn`. This outputs the C code `void main() {` and again does not invoke code generation on the symbol `main`. It does however invoke it on the next child, `puts`. This invocation generates the C code `puts(` and invokes code generation on the string, which outputs `"Hello, world!"`. Finally, as the control flow return back up through the tree, the program is closed off with `);}`, resulting in a valid "hello world" C program.

Chapter 3

Background and Related Works

The additional features added to Clip by this project are based on existing systems in other programming languages, typically that are higher level than C. As a result, many of the challenges regarding their implementation have been tackled before.

3.1 Namespaces

Namespaces are a common feature of programming languages. They allow for multiple pieces of code to be combined without worry that they contain common object names. In most modern programming languages, namespaces function by adding a common prefix to all names defined in a section of the program. These sections may be defined explicitly, such as in C++ or C#, or implicitly via separation of code into files or other delimiters, such as in Python or Rust. The prefix is then typically formed from the name of the namespace followed by a separation mark, commonly `.` (as seen in C# and Python) or `::` (as seen in C++ and Rust).

Namespaces are present in most modern languages that allow for reuse of code in the form of libraries, including some Lisps. C, however, does not have namespaces. The implementation of namespaces into a system that does not have them has been done before: C++, as part of its design goals, tries to maintain compatibility with C. While the implementation of this at a language level is trivial, simply treating all C names as part of the global namespace, compatability on the ABI level is more complex. The Itanum C++ ABI [2] is one of many attempts to solve this, detailing a method for providing binary names to C++ objects.

3.2 Parameterized Typing

As computer programming became more widespread and advanced, it became clear that the simple type systems used by languages such as C were not ideal for creating reusable code. For example, a common task in language and library design in the implementation of collections, such as lists, sets, and maps. These types find frequent use in a large variety of programs, resulting in them being present in the standard libraries of basically all modern language implementations. However, attempting to implement them in a simple static type system quickly runs into a problem: what type do they contain? For example, consider the common task of adding an item to a list; the intuative interface to expose to programmers would be a method, let's call it `add`, that takes a list and an item, and adds the item to the end of the list, either in place, or in the form of returning a new list. What should be the signature of this function? It must take an item to add to the list, and that parameter needs a type. In the case of a type system such as C's, A separate list type would be required for every possible type of item, and the code for methods such as `add` would need to be reduplicated.

Many staticly-typed languages solve this issue using *parameterized typing*. Parameterized typing refers to any method where types and functions can be written such that their signatures do not fully specify what types they contain, instead only putting restrictions on how those types interact. This allows for the same code to be used to handle multiple types, by only performing operations common to all of them. There are two common implementations of parameterized typing, *templates* and *generics*. [5]

3.2.1 Templates

Templates allow for type parameters to be specified, and functions and types to be written that contain the paramaterized types. When a compiler reads a template, it parses the code that uses the type parameters, but does not further compile it. [7] When a template type is used and its actual type defined, the specified "concrete" type is substituted into the parsed code, which is then finally compiled. This also allows template parameters to be objects other than types, but that is not important here.

An example of a language that uses templates is C++. [9]

Chapter 4

Language Improvements

This section of the report covers the changes made to the language design and specification in the second year of the project, as well as their implementation in the reference transpiler.

4.1 Comments

The only syntactic change to the language in this part of the project is the addition of comments. Comments are a feature of all major programming languages. They exist to aid programmers with reading and understanding existing code. [6]

Comments in Clip are started with a `;` character and continue to the end of the line. They can be be started anywhere that is not inside a symbol or string literal. This changes the set of valid symbols in Clip to exclude those starting with `;`, however a `;` located in the middle of a symbol is accepted.

This style of comment was chosen for its use in Common Lisp, and also in several other Lisps. Many programming languages, including some Lisps, support multi-line comments. Currently, Clip does not, however suppoer could be added in a similar way.

An example of a comment is shown in Figure 4.1. Such comments will be used in other examples throughout the report.

```
(: main -> int)
(defn main
    ; This is a comment!
    (return 0))
```

Figure 4.1: Clip code containing a comment.

4.1.1 Implementation

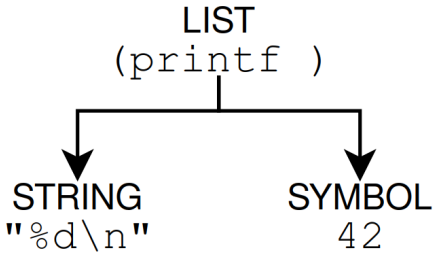
The implementation of comments occurs at the lexer level, as described in the report for part 1. When a token is requested from the lexer, the lexer first moves its pointer within the input source forwards over any whitespace. Comment parsing was added to this section - should a semicolon be encountered, the pointer is moved forwards until a newline character is found. By implementing comments here, they can easily be passed over while lexing whilst not affecting the behaviour of semicolons located within string literals.

4.2 Statements in Expressions

Code generation in the first year of the project was implemented as a traversal over the Clip expression tree that output code to a text stream at each node. For example, consider the following code and its expression tree:

```
(printf "%d\n" 42)
```

(a) Clip code



(b) Expression tree

The Clip transpiler would perform code generation in the following steps:

1. At the root "printf" node, since this represents a function call, output the string `printf(`.
2. Traverse to the first child.
3. This is a string literal, so output the equivalent literal as C (`"%d\x0a"`)
4. Return to the parent.
5. Another child is to come, so output `, `.
6. Traverse to the second child.
7. This symbol represents a number, so output the appropriate C literal (`42`)
8. Return to the parent.
9. No more children exist, so output `)`.
10. Return to the parent (in a larger program.)

This produces the C code `printf("%d\x0a", 42)`, which is the correct equivalent code for the expression. (In practice, this would be a statement and therefore end with a semicolon, however in Clip the statement environment and therefore its required syntax is handled by the parent node, which is not shown in this example.) While this works effectively for simple expressions like the code above, a problem occurs if one of the subtrees cannot be represented as a single expression in C.

4.2.1 The Structure of C Code

To explain this problem, first we must cover the structure of C code. A block of C code (part of the program that represents actually executable instructions)

consists of a list of statements. These statements may take the form of control flow, blocks, or expressions. [8] Control flow statements may contain expressions within them. Expressions can also contain sub expressions. This means a single statement may contain many expressions. Consider the following C:

```
if (x == 1)
```

This is an if statement, a type of control flow. It contains an expression in it, `x == 1`, that itself contains two sub expressions, `x` and `1`. Whilst C expressions can contain others almost arbitrarily, as long as types match, they cannot contain statements.

4.2.2 The Problem

Clip is not so structured. Its simple grammar allows for any structure to be placed inside any other. These combinations aren't all valid, however there are a lot that are whilst their equivalent C is not. The simplest such example is `do`, which evaluates a list of statements, returning the result of the final one. This fundamentally goes against the nature of C expressions, which cannot contain statements.

Consider the following Clip:

```
(+
    (do
        (foo)
        bar)
    (do
        (baz)
        quux))
```

This code adds the value of `bar` after `foo` has been called to the value of `quux` after `baz` has been called. The equivalent C code may look something like this:

```
foo();
baz();

bar + quux
```

Except this assumes that `baz` does not affect the value of `bar`, which may not be the case. The actual correct C code would be

```
bar_t temp;

foo();
temp = bar;
baz();

temp + quux
```

where `bar_t` is the type of `bar`. This is clearly very different structurally from the original Clip code, which is a problem in the part 1 method of code generation. Consider the initial example in this section, involving a call to printf. What if the second argument had not been a literal such as 42, but instead the expression above? By the time the compiler reached the expression, it would have already commited to outputting the string `printf("%d\x0a",` , meaning it could not insert the required definition and statements above.

(As a side note, the code actually could be represented as a C expression using the comma operator, in the form `(foo(), bar) + (baz(), quux)`. The code above was given only as an example of a case where a sequence point is required in an expression. There are many more complex expressions that cannot be written in C in such a manner, such as those containing looping control flow.)

4.2.3 The Solution

In order to allow for such code to be generated, the implementation was changed to recurse over the tree, with each node returning both code that represents the expression result, and code that contains any statements that need to be run prior. Most nodes also forward the required prior statements of any child node. Generating output like this means that no node commits anything before all of its descendants have been parsed. Nodes that represent contexts that can contain statements then place all the requested statements in order, instead of passing them up the tree. This is important in cases of control flow, where the required prior statements may have side-effects and should only be run conditionally.

Often, these statements work by assigning the desired result to a temporary variable, then returning that as their main code. Consider the following example of a conditional, where `some_t` is the type of `foo`'s parameter. (In practice, Clip would use a ternary conditional operator here, but assume that this is prohibited by one of the branches requiring multiple statements.)

```
(foo
    (if condition
        (bar)
    else
        (baz)))
```

(a) Clip code

```
some_t temp;
if (condition) {
    temp = bar();
} else {
    temp = baz();
}
foo(temp);
```

(b) Equivalent C code

4.2.3.1 Looping Control Flow

An additional consideration has to be made for looping control flow. If the same approach is used in, for example, a while loop, code such as the following could be generated:

```
int condition = required_prior_processing();
while (condition) {
    // some code
}
```

This is clearly an issue. `required_prior_processing` will only be called before the first iteration, instead of every loop. To avoid this, loops must be generated similar to the following:

```
while (1) {
    int condition = required_prior_processing();
    if (!condition) {
        break;
    }

    // some code
}
```

4.3 Sum and Product Types

The C programming language allows for the definition of custom types. These can be sum types, in the form of unions, or product types, in the form of structs. Such types allow programmers to group related data together, and manage it easier.

Clip implements such types in a manner equivalent to C, with syntax vaguely based on Racket's. Structs are defined with the `struct` node, which takes a name and a list of either (type, field name) pairs or `union` nodes. Named unions cannot be created in Clip. Instead, an equivalent type can be defined by using a named struct containing a single anonymous union. An example struct definition is shown in Figure 4.4

```
(struct my-struct
    int foo
    (union
        int bar
        float baz))
```

Figure 4.4: A simple struct definition that contains an anonymous union.

As well as being defined as a type, the struct name (`my-struct` in the example above) is also defined as a constructor that takes arguments representing the values of each field, in the order they are defined, and produces a new instance of the struct. If fewer arguments are provided than the struct has fields, then any field not represented is left uninitialized.

Further symbols are defined for field-level access. These take the form of [struct name]-[field name] and can be used to both set and get the relevant field's value. This naming pattern was chosen for its use by Racket.

4.4 Parameterized Typing

Clip uses templates to implement parameterized typing. `template` is a type of node that can be placed at the root level of a Clip file. It takes a list of template parameters, followed by any number of definitions. These definitions may use the template parameters in place of any type. Definitions placed inside of a template do not generate code when parsed; instead, the symbols that would be defined still are, and when they are used, the template parameters are substituted with the appropriate concrete types, and then the final code is generated and placed before the currently-being-parsed struct or function. Each parameter list is only generated once; templates store a map of already generated concretely-typed code to its used names in C, and reuse those definitions if the same parameter list occurs multiple times. Figure 4.5 shows an example of a simple function that uses a template to take and return multiple types.

```
(template (T)
    (: square T -> T)
    (defn square (x)
        (return (* x x))))
```

Figure 4.5: A function that squares a number, without requiring it be any specific type.

In order to call a function defined inside of a template, the template arguments must be passed to it first before its regular arguments. In the example shown above, squaring the integer 12 would be written as such: `(square int 12)`. Types defined in a template are similar, taking template arguments as their first arguments both when used as types and as constructors. Field access, however, does not require explicit typing, instead relying on the type of the argument to the access call. Clip does not implement any form of type inference in templated definitions outside of field access.

Currently, the arguments to Clip templates must be types. This is different from some languages, where they can be any expression. Extending Clip to allow for arbitrary expressions in templates is something that would not be too difficult based on the current implementation. The reasoning for only allowing types is that it is much easier to create a canonical form for types. This is useful when determining if two specializations of a template are the same, as they can be compared directly. Types, unlike expressions, also implement `Hash`, allowing them to be used efficiently as keys in a map.

A further reason to avoid arbitrary templates, at least as they would be in the current form of the language, is that it makes type inference much harder, should it be desired to be implemented for templates. It is rather unwieldy using the current system of explicit typing every type a templated object is referenced. Most

languages with parameterized typing will infer the types required if possible when invoking methods. As such, this is likely something Clip will have implemented in the future, and arbitrary expressions in templates complicates this process.

4.5 Namespaces and Libraries

As programs get more complex, it becomes infeasable to contain all their code in a single file. Additionally, several programs may wish to perform the same task. It would be useful for code that performs such tasks to not need to be duplicated in many places, but instead be shared across programs. Namespacing and libraries help with solving these problems.

Namespaces exist to collect related symbols and ensure that they don't conflict with symbol names from other parts of the program or libraries. In Clip, namespaces must span a full source file. The namespace of a file is set using the `namespace` node, which takes a single symbol for the name. Within the file, the namespace is not used; symbols are referred to by the names they are defined with only. However, the generated C code includes the namespace in all global C names, in order to avoid conflicts when the resulting object is linked into an executable.

4.5.1 Imports

Namespaces appear in the Clip source when a symbol defined in a namespaced file is accessed from outside that file. Such accesses are done via importing, which is how Clip handles both multi-file programs and libraries. When an `import` node is reached, the Clip transpiler opens the requested file and parses it as Clip. Unlike the main file, however, the only code generated is for type definitions and method signatures; no actual instructions are produced. The result is something similar to a C header file: code that provides information for the type checker to verify correctness and for the linker to connect afterwards, but not able to run on its own.

After the header-like code has been generated, the global symbols defined in the imported file are moved into the importing file's global scope. Here, they are prefixed by the namespace, followed by a colon. This ensures that the imported symbols don't overwrite any symbols defined in the main file or any other imported files, now that they are in a context where symbols from many files may be used together.

The result is a system that allows for effective multi-file programming. An example of both Clip and the resulting C is shown in Figure 4.6.

```
(namespace greeter)
(: greet (pointer (const char)))
(defn greet (name)
    (printf "Hello, %s!\n" name))
```

(a) Greeter.clip

```
(import "Greeter.clip")
(: main)
(defn main
    (greeter:greet "world"))
```

(b) Main.clip

```
void greeter3agreet(char const*);
void greeter3agreet
  (char const* name) {
    printf("Hello, %s!\x0a", name);
}
```

(c) Greeter.c

```
void greeter3agreet(char const*);
void main();
void main() {
    greeter3agreet("world");
}
```

(d) Main.c

Figure 4.6

4.5.1.1 Importing Templates

Due to templates not producing code until they are used, a template within an imported file is treated much the same as one defined in the currently-being-parsed file. The namespace is still prepended, however there is nothing to reference in the imported code's object file, so no code need be generated for the linker. In the case of a library that contains only templates, it is possible for a Clip file to be transpiled to an empty C file. This is similar to the idea of header-only libraries in C++, which are commonly used in similarly template-heavy situations. [9]

If two files that import the same template from the same library both specialize a template with the same arguments, both files seperately produce a type definition for the resulting type. In order for these types to be compatible, their definitions in C must match. As such, the Clip transpiler attempts to define types created from template specialization deterministically. However, this also leads to duplicate functions. Unlike with types, such a problem is currently unsolved in Clip.

4.5.2 Extern and Interfacing with Non-Clip Code

If a specifc symbol name is desired to be used in the resulting C code, the `extern-c` node can be used to bind it to a Clip name. This is useful for when you either want to write a library in Clip that exposes specific names so that other languages that are compatible with the relevant C ABI can use it, or for the case where you wish to use a function defined in a library written in a non-Clip language that is compatible with the C ABI. The node is named after the `extern "C"` declaration used in C++ to specify that a symbol has C linkage. [4]

4.6 The Standard Library

In addition to importing libraries local to the project being built, many programming languages also include a standard library that is available from anywhere and contains very common features that are likely to be useful across a wide range of tasks. This includes C's common implementations, which have various forms of libc available to them. In order for Clip to access the full power of C, it needs to also be able to call these libraries.

Additionally, Clip provides additional features on top of C, and so there is reason for it to have its own standard library beyond C's. In order for this to be implemented, some form of global search path for importing must be created, or passed as a compiler parameter in the manner of gcc's `-I` flag. This was not done for this project. Nevertheless, a very basic standard library concept was created for the purposes of providing features required in later sections of this report.

4.6.1 The C Standard Library

Addapting the C standard library for Clip is essentially a task of reproducing the header files. As C code is linked against libc anyway, nothing but signatures and type definitions need be converted. As a process, this would be easy to automate using a C parser, however this was not done for this project, as only certain minimal parts of the C standard library were actuall converted. The library was also given a namespace, unlike in C. Figure 4.7 shows a sample of the implementation of `malloc` and related memory functions as they appear in `stdlib.h`, compared with their C equivalent.

```
extern void *malloc (size_t __size)
__THROW __attribute_malloc__
__attribute_alloc_size__ ((1)) __wur;

extern void *calloc (size_t __nmemb,
size_t __size) __THROW
__attribute_malloc__
__attribute_alloc_size__ ((1, 2))
__wur;

extern void *realloc (void *__ptr,
size_t __size) __THROW
__attribute_warn_unused_result__
__attribute_alloc_size__ ((2));

extern void free (void *__ptr)
__THROW;
```

(a) C header

```
(namespace c)

(extern-c malloc malloc)
(: malloc size_t ->
  (pointer void))

(extern-c calloc calloc)
(: calloc size_t size_t ->
  (pointer void))

(extern-c realloc realloc)
(: realloc (pointer void)
  size_t -> (pointer void))

(extern-c free free)
(: free (pointer void))
```

(b) Equivalent Clip library

Figure 4.7: A small sample of stdlib.h converted to Clip.

4.6.2 The Clip Standard Library

For the purposes of this project, not much of a Clip standard library was designed. The main focus was on providing a list type similar in behaviour to C++'s `std::vector` or Rust's `std::vec::Vec`. Such a type has both the demonstration purpose of showing how Clip's can be used to easier create more complex data structures than are available in C, and the practical purpose of being required to represent expressions in the "Metaprogramming" section later in the report. The definition of the list type is given in Figure 4.8.

```
(namespace std)

(template (T)
    (struct list
        (pointer T) data
        size_t length
        size_t capacity))
```

Figure 4.8: Type definition of list

This implementation is based on Rust's. [3] The elements of the list are stored at `data`, which points to a contiguous section of `capacity` `T`s, of which only the first `length` are guaranteed to be initialized.

When an element is added to a list, `length` and `capacity` are first compared; if they are equal, `capacity` is doubled and `data` is reallocated to be the new, larger size. Then, length is incremented, and the new element is written to the end of the list. Removing an element from the list simply decrements `length`.

Such operations change the values stored in the list struct. As such, if the list is ever copied, the copies can become out of sync, and potentially end up containing dangling pointers. Therefore, Clip currently recommends list objects be kept on the heap, and only referenced to with pointers, to prevent accidental copying. However, this adds an additional layer of indirection when accessing a list, and requires a heap allocation every time a list is created, which is inefficient. There are two common ways non-garbage-collected languages deal with this issue, neither of which were deemed viable to implement in Clip within the project timeframe.

The first solution is used by C++. This solution relies on the language allowing the definition of custom behaviour for when a value is moved, copied, or dropped. In such a system, whenever a list object is copied, a copy of the memory pointed at by `data` is also made. This allows both lists to safely become out of sync, and makes them essentially passed by value. Additionally, when a list is dropped, it's allocated data is freed, meaning the programmer does not have to manually manage memory.

This approach has pros and cons. It's behaviour is intuative; the value of a list is its contents, and it can be copied and passed around like any other type. There is no synchronization between copies of the same list, which prevents issues such as values changing when not written to because another copy was modified. However, by defining a custom copy operator, copying the list object becomes an $O(n)$ operation. Since invocations of the copy operator are usually implicit, it can be hard to tell where it is occurring, making finding performance and memory problems when dealing with large lists difficult.

The other solution is used by Rust. This solution prohibits objects from being copied; when a value is assigned to storage, it is removed from any other location. This prevents the invisible performance costs of the other solution; copying a struct of fixed size is always $O(1)$. However, the inability to copy values is unintuative to programmers coming from many other languages, notably including C and Lisp. Additionally, implementing this as a language feature is difficult, as it requires the compiler to keep track of what variables are valid to use. This is difficult and often impossible to solve at compile time in the general case. A large part of Rust's design is dedicated to solving this problem, and it adds significant complexity to the language incompatible with the design principles of Clip.

4.7 Closures

Closures are first-class function objects that contain names bound to variables outside of the function's scope. They are an important part of functional programming, and are used commonly in Lisp, to the point where there is even a Lisp named after them (Clojure). Unlike a function pointer, a closure must also store data about the environment it was created in, as it may access variables present there. As a result, multiple closures created from the same function may have different behaviours.

Memory management regarding closures is complicated. The variables they refer to typically exist on the stack, and may go out of scope before the closure itself is dropped. This is a problem for languages that do not peform automatic memory management. While some solutions exist, they typically use techniques similar to those described in the section on lists above (See "The Clip Standard Library") which for reasons prior discussed have not been implemented in Clip. As such, the following design for closures in Clip presents a more limited form, in which closure objects may only be passed down-stack from where they are created. Whilst the current design of closures in Clip is complete and shown here, a full and correct implementation has not been completed.

Closures in Clip are typed by their signature; closures with different signatures are different types, but all closures with the same signature share a type. This is different to the implementations in some languages, where closures with the same signature but different base functions are different types. A closure is represented in Clip as a tuple of a void pointer and a function pointer to a function with the same signature as the closure, except for an additional void pointer parameter. When a closure is called, the function pointer is called with the void pointer passed as the argument to the added parameter. This void pointer points to the data that the closure closed over.

For the purposes of the closure type, there is no restriction on the type or format of the data pointed to by the void pointer; the function just has to understand it. However, for closures created via language features such as lambdas, it typically takes the form of an array of pointers located on the stack in the same frame that the closure was created in that contains the addresses of all variables referenced within the function. The addresses are used instead of the values themselves as closures are allowed to mutate variables in the enclosing scope.

Much like with templates, the type used by a closure (a two-field struct) is defined before the first type or function definition that uses it, with a map to ensure the definition is not repeated within the same C file. An example of how this looks in C, where a closure is created and immediately called, is shown in Figure 4.9.

```
(defn main
    (def int x)
    (def int y 5)
    (def (lambda int) f
        (lambda int i
            (set x (+ i y))))
    (call f 4)
    (c:printf "%d\n" x)) ; Prints 9
```

(a) Clip code snippet

```
struct _closure_int_void {
    void (*function)(void*,
        int);
    void *data;
};

void _closure_int_void_call
(struct _closure_int_void
closure, int a) {
    closure.function(
        closure.data, a);
}

void _lambda_700a581f
(void *_data, int i) {
    *(((int**)_data)[0]) = i
        + *(((int**)_data)[1]);
}

void main() {
    int x; // (def int x)

    int y = 5; // (def int y 5)

    // (def (closure int) f ...
    void *_lambda_data_700a581f[2];
    _lambda_data_700a581f[0] =
        (void*)&x;
    _lambda_data_700a581f[1] =
        (void*)&y;
    struct _closure_int_void f;
    f.function = &_lambda_700a581f;
    f.data = _lambda_data_700a581f;

    // (call f 4)
    _closure_int_void_call(f, 4);

    // (printf "%d\n" x)
    printf("%d\x0a", x);
}
```

(b) Equivalent C code

Figure 4.9: A simple closure that both reads from and writes to its enclosing scope.

The function used by the lambda is named randomly to avoid conflicts with other lambdas of the same type. In order to determine the size and elements of the

closure's data, the transpiler must keep track of what scope accesses are made during the parsing of the lambda. To implement this, a flag was added to scope objects that allows them to be marked as the root scope of a lambda. When such a scope is asked for a symbol, and is required to pass it up the tree due to not having it defined within itself, the request is recorded and the definition of the symbol changed to refer to a value within the data parameter.

4.8 Metaprogramming

Metaprogramming is a technique where programs operate on other code as their data, either at runtime, as a step of compilation in a single program, or as a full process that occurs before the final build. Lisp's design and philosophy of "code is data" makes it ideal for metaprogramming, as any code can be easily generated or manipulated. Unfortunately, such a system relies on Lisp being interpreted, as the ability to generate code on the fly means the code can only be known at runtime. As such, when bringing metaprogramming to Clip, different approaches need to be considered.

C by itself supports only very basic metaprogramming. It has a preprocessor capable of performing basic logic and text substitution tasks. This is sufficient for simple uses of metaprogramming, such as reducing heavily duplicated code, and conditionally including code based on the build environment. However it is incapable of performing logic based on the program itself, rendering it much less powerful than Lisp's macros, which can be arbitrary functions on the programs syntax tree.

Rust provides an example of a much more powerful macro system. In Rust, macros are functions that map between lists of tokens. [3] They can also call on the compiler to parse token lists, should they wish to operate on the syntax tree, and call on the lexer to parse strings, should they wish to generate raw source code. This allows for incredible flexibilty when creating macros. Unfortunately, it also comes with extra complexity; token stream manipulation is not intuative, as code may be very complex, and the Rust syntax tree has a very large number of node types. Clip, however, does not have these issues, and so aims to implement a similar system.

Unlike Rust's macros, Clip macros operate on the abstract syntax tree directly. This is due to how close Clip's syntax is to being a direct representation of the AST. A Clip macro is a function from an AST node (representing the macro call and any children) to an AST node (which the macro call will be replaced with) that is called when a matching symbol is encountered. Similarly to Rust, a Clip macro may query the transpiler about its state, or for information on the program (such as definitions.)

When implementing this, a problem quickly becomes apparent: the macro function needs to be evaluated at compile time, which means it must be itself compiled; howver it is possible the function is part of the same compilation unit as the code currently being built. Early designs of Clip's macro system attempted to resolve this by isolating the macro function and building it alone, however complications quickly arose in the task of determining its dependencies on other parts of the program. Instead, the current design once again follows Rust's

lead, and simply prohibits macros from being used within the same compilation unit as they are defined. This has the unfortunate consequence of requiring any program that uses macros to contain multiple compilation units, discouraging their use in smaller projects and quick scripts. I was unable to find a solution to this, and since Rust has the same issue, I do not believe an obvious answer exists.

4.8.1 Macros: From the Developer's Side

Macros in Clip are implemented literally as functions from abstract syntax trees to abstract syntax trees, with an extra command to define that they are a macro and not a regular function. In order for this to be done, a representation of an abstract syntax tree needs to be made available in Clip. Such a type is included as part of the Clip standard library, and is defined as follows:

```
(struct expression
    int type ; Used as an enum
             ; 0 => Symbol
             ; 1 => String
             ; 2 => List
    (union
        (pointer (const char)) text ; Used by symbols
                                    ; and strings

        (pointer (std:list expression)) items ; Used by lists
    ))
```

As well as expressions, macro developer's have access to various other functions to query or affect the transpiler, such as asking for types of expressions, or throwing errors. The list of available functions has not been finalized during the development of this project. As for the actual macro code, it takes the following form:

```
(: my-macro std:macro:expression -> std:macro:expression)
(defn my-macro (input)
    ; Perform some operations that define a result
    (return output))

(macro my-macro)
```

4.8.2 Macros: From the Transpiler's Side

Whilst the language specification itself does not restrict how macros are implemented, the following was used for the implementation in this project. Unfortunately, said implementation was unable to be completed in a functional manner. Macros are compiled into functions with wrapper code around them. As they are used like libraries, a compilation unit containing macros must not also contain a main function. Once compiled, the object is stored as a binary executable indexed

by its contained macros' fully-namespaced names. This allows it to be found and run when the transpiler encounters a macro. The process for building and running a macro relies on executing the macro as a separate process to the transpiler, and communicating with it over a socket. It may seem more effective to compile macros to shared libraries that can be dynamically linked and invoked directly, however there are many benefits to the multi-process model.

For example, another process cannot alter the transpiler's memory. If the macro's code corrupts memory or crashes, the main process remains intact, allowing it to output a sane error message. It also allows the transpiler's code to be memory safe, which interacting with an arbitrary C function isn't. As the transpiler is implemented in Rust, whcih has a lot of complicated rules about valid memory states, unsafe memory operations are best avoided. Additionally, socket based communication is more standardized across modern operating systems than run-time linking of libraries is, meaning there is less need to develop platform specific code.

When a macro call is required, the Clip transpiler binds a socket on the loop-back interface on a random available port. It then launches the desired macro executable, communicating the port to it via the process arguments. If multiple macro binaries are required, they all share a server address. Once connected, the transpiler sends the AST for any macro call to the process over the socket. The macro process then may query the transpiler any number of times for program information as part of the process of evaluating the macro. Once either a new AST or an error has been determined as the result of the macro, it is sent back to the transpiler. The connection stays open in case another instance of the same macro is encountered; once the build is finished, the process is closed via a signal. Figure 4.10 shows the communications flow of this process.

4.8.2.1 Serialization of Expressions

In order to simply both encoding and decoding of expressions when transmitted between processes, the s-expression form is not used. Instead, they are binary encoded. The first byte indicates the node type (0 = Symbol, 1 = String, 2 = List), the next four bytes encode the length of the node, and then there are either length bytes send representing the text contents, for symbols and strings, or length more expressions sent, for lists.

When the macro returns the new expression tree, the format is the same, however the root expression may take on a forth type (represented with byte 4): Error. This is treated like a symbol or string, in that the contents are read as text. This text is used as the error message displayed by the transpiler; the line and column numbers are inserted automatically based on the position of the macro.

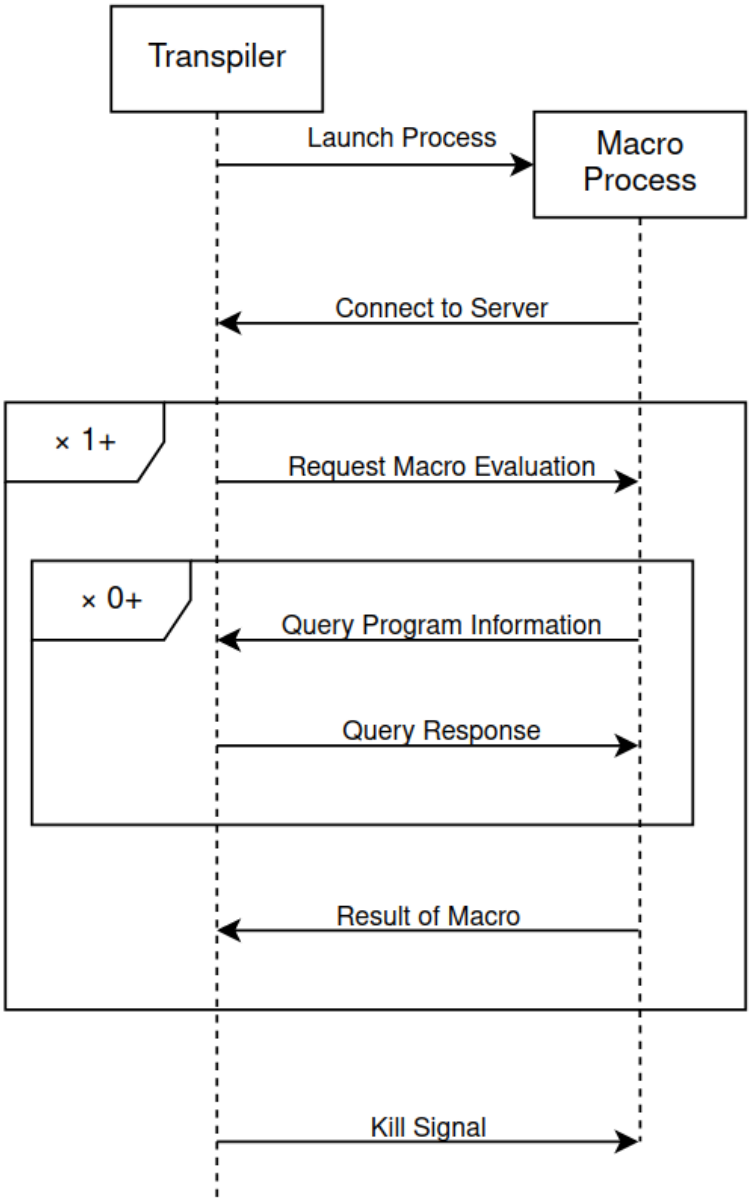


Figure 4.10: Flow of transpiler-macro communication

Chapter 5

Conclusions and Potential Future Work

5.1 Conclusions

This project has clearly shown the feasibility of developing a Lisp-like language on top of C. The design created here demonstrates the various ways that such a language can be created, both to utilize the power of the C language, and to extend it in ways that aid with the development of complex programs by implementing various features typically seen in higher-level languages.

The features included in this report cover the desired language features discussed in the first year of the project (the improved type system, closures, and metaprogramming tools) as well as some extras. This suggests the work this year made good progress towards extending the language further.

Unfortunately, the implementation part of the language has fallen behind the design, with some features only partially working. However, all the remaining problems are in the process of creating the transpiler as described, and not in any aspect of its design. As such the language itself as described is likely completely viable to implement.

5.2 Potential Future Work

Whilst this is the final year of the project, there are still improvements that could be made to the language. This section lists some possible considerations should this work be built upon in the future.

5.2.1 Implementation

Obviously, the creation of a fully-working implementation of the language specified in this document would be a great first step. While much of the implementation is complete, some aspects are not yet fully functional. Additionally, there are likely various edge cases within the existing system that do not function properly. Finding these, and perhaps designing a proper test suite for them, would be an important part of ensuring the language is viable for real-world use.

5.2.2 Custom Copy and Drop Operators

The current way lists function in Clip is not very intuative, and this problem will only get worse as more complex data structures are implemented. As discussed in the section on the Clip standard library, there are two ways this problem is commonly solved. The method used by Rust requires a significant overhaul in language design that goes against the principles of Clip, therefore the actual solution is likely to be the ability to define custom copy and drop operators like in C++. Designing and implementing this feature would go a long way towards making the language easier to use.

5.2.3 More Standard Library

The current Clip standard library essentially exists as a proof of concept. A full language's standard library should include far more common data types and operations. Expanding the library with various common data structures, functions, and macros will significantly aid the development of future Clip programs. Languages such as C++ have extensive standard libraries that cover large amounts of common data structures; it may be a good idea to attempt to implement analogous libraries in Clip.

5.2.4 Bootstrapping

A stretch goal from last year that still applies: bootstrap the compiler, rewriting it entirely in Clip. The additional tools now available as part of the language make this much easier than it previously was. It will still be tough, the Clip compiler has taken a significant amount of work to produce as it is. Completing

this, however, would undeniably show Clip's viability as a programming language.

Bibliography

[1] Marley Adair. Clip: A lisp-like "resyntaxing" of the c programming language. Master's thesis, University of Edinburgh, 2023.

[2] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI et al. *Itanium C++ ABI*.

[3] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. The Rust Foundation.

[4] Microsoft. *extern (C++)*.

[5] Microsoft. *Generics and Templates (C++/CLI)*.

[6] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)*, pages 83–92. Ieee, 2013.

[7] Bjarne Stroustrup. Parameterized types for c++. In *C++ Conference*, pages 1–18, 1988.

[8] Sergey Zubkov. *C language | Statements*.

[9] Sergey Zubkov. *C++ Reference | Templates*.