St Andrews Algol to Javascript compiler project

William Trend

March 2016

Abstract

This document describes the implementation of a compiler that accepts St Andrews Algol (S-algol) programs and produces analogous Javascript programs. It describes the implementation of a lexer, a parser, static analysis stages and code generation for such a compiler. It also provides an examination of the development in computer language design since 1979.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 13,759 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

1 Introduction

The over-arching aim of this project is to produce a Javascript compiler for S-algol which will preserve the language and provide a resource for people interested in its history. This report contains details of how to compile and execute the project, how the project was designed and how it was implemented.

A large proportion of the project requirements have been completed: the high-level features that have been implemented are listed in Figure 1. These features are verified by the test cases shown in Table 1 and examples of illegal program detection are detailed in Table 2. These tables can be found in Appendix A. The web-page that provides an interface for the project is available at
`https://wt6.host.cs.st-andrews.ac.uk/st-andrews-algol-compiler/` .

1. A lexer and parser for the full S-algol grammar
2. Code generation for most of the language features
   (a) Procedure declaration and application
   (b) Variable declaration, assignment and re-assignment
   (c) Mathematical operations
   (d) Logical operations
   (e) n-dimensional vectors
   (f) Structure declaration, instantiation and access.
   (g) Types annotations
   (h) All loop types
3. Checking for scope errors
4. Checking for type errors
5. A command-line interface for the compiler
6. A 'promotional' web interface for the language
7. A test suite for both individual parts of the compiler tool chain and end-to-end functionality.

Figure 1: Features Implemented

Some parts are missing from the final implementation due to technical and time constraints. These are listed in Figure 2.

1. Image and pixel manipulation
2. Blocking io
3. Some of the standard libraries
4. Some of the client libraries
5. Automatic semi-colon insertion

Figure 2: Missing features

2 Context survey

2.1 Historical Significance of S-algol

The St Andrews Algol language (S-algol) was designed as part of Ron Morrison's PHD thesis in 1979 [13]. It was subsequently used to teach programming to undergraduates at the university computer science department until 1999 when it was replaced by Java. The language was also used at the local Madras college.

S-algol is a member of a family of languages known as the 'ALGOLs'. The name of the family references their ability to easily abstract algorithms. In his thesis, Morrison highlights 5 features of programming languages that "roughly" distinguish the ALGOLs [13, p5-6]:

1. "Scope rules and block structure"
2. "Abstraction facility"
3. "Compile time type checking"
4. "Infinite store"

Given this description, it is clear that many modern languages could be described as being part of the ALGOL family. Indeed it seems that this is so much the case that it is no longer particularly useful to classify languages as ALGOLs. Certainly there are few modern languages that are not influenced heavily by the ALGOLs. Reynolds corroborates this by his definition of the ALGOL family in 1981 as "a conceptual universe for language design that one hopes will encompass languages far more general than its progenitor" [18].

In the introduction to his thesis, Morrison offers a thoughtful commentary on the problems of programming language design and the ways in which S-algol intends to solve them. In this spirit, this report shall provide a similar analysis regarding how modern languages have built on earlier languages such as S-algol.

The contrast between S-algol and modern languages were made clear during the development of this project since it is written in Typescript - a language released in 2012 [9]. Typescript has modern syntax and semantics and provides excellent reference as to how S-algol compares to the cutting edge of programming language design.

2.2 Javascript as a Tool for Preservation

Javascript was considered an appropriate language to use as a compile target for this compiler given its advantages in various capacities. These are detailed in this section.

Performance Since the main aim of this project is to be able to execute code written in S-algol, performance is not a priority. However, Javascript performance is known to be constantly improving: Mozilla has even managed to execute some subsets of the language at near native performance [16].

Portability The principal advantage of Javascript for this project was portability. It contrasts with compiled programming language such as C which are often architecture dependent. C may need to be re-compiled for each host operating system or architecture that it is to be run on. Javascript on the other hand can be distributed as source and executed on any platform. Furthermore, Javascript runs in a browser sandbox which means that it is encapsulated from the host operating system, preventing it from executing maliciously. Executing C often requires elevated privileges since it can perform operations that might be undesirable.

Ubiquity Access to the websites is the most basic function of most consumer electronic devices sold today. Since Javascript is the programming language of the web, almost every computing device can run it at least through a web browser. Some devices offer Javascript first-class citizenship as an application programming language. Native iOS apps for example may be written for the most part in Javascript using the native 'bridge' that makes it possible to execute "Javascript scripts from Objective-C or Swift code, to access values defined in or calculated in Javascript, and to make native objects, methods, or functions accessible to Javascript" [2]. The extent to which Javascript is embedded within today's technology means there is always likely to be some device that can run Javscript available.

Longevity The ubiquity of Javascript implies that the language will be still be usable for the foreseeable future. Its longevity is being aided by the availability of "compile-to-Javascript" languages. These languages use Javascript like "an assembly language" [8]. Erik Meijer suggests that in these cases, "the browser [may be able to] execute [Javascript], but no human should really care whats there." As such, it is unlikely that Javascript will be made redundant any time soon.

Precedence Javascript is used as a target language for other technological preservation projects. The "Javascript MESS" project is a movement to produce an emulator for 'hundreds' of virtual machine types in the browser specifically for preserving gaming platforms. In justification of MESS, Jason Scott states the

benefit of using Javascript: "by porting this program into the standardised and cross-platform Javascript language, it will be possible to turn computer history and experience into the same embeddable object [type] as movies, documents, and audio"[10] .

There is also precedent for writing compilers that target Javascript. Many compile-to-Javascript languages have been implemented. CoffeeScript is one of the most well-known compile-to-Javascript projects [3]. It does not offer vastly different programming paradigms to regular Javascript. Rather, the language offers 'syntactic sugar' to make writing succinct Javascript easier. The language has been validated by many companies including GitHub as a viable programming language by using it for some of their big projects [7].

2.3 Language specification

A major challenge of this project was something of an archaeological problem. The S-algol language was first designed and implemented nearly 40 years before the start of this project. This fact, coupled with the language's modest adoption outside of St Andrews, means that there is a limited set of documentation available. The sources available for reference were:

1. "S-algol Reference Manual" [14]
2. "ON THE DEVELOPMENT OF ALGOL" [13]
3. "Recursive Descent Compiling" [5]
4. "An introduction to programming with S-algol" [4]
5. A C implementation of the S-algol compiler.
6. An S-algol implementation of the S-algol compiler.

Within these sources 3 different grammar specifications are available. These, grammar specifications are roughly analogous but contain subtle differences.

"S-algol Reference Manual" defines a context-free grammar that is the main point-of-reference for this project.

"An introduction to programming with S-algol" defines a two-level grammar. This is a variation of a Van Wijngaarden grammar that precisely defines all possible strings accepted by the S-algol language including how types may be combined through operations [4].

The final formal specification, is found in Morrison's book, "Recursive Descent Compiling". This appears to be a subset of the context-free grammar found in "S-algol Reference Manual" that does not include the first-class pixel and image manipulation features.

The "S-algol Reference Manual" grammar was chosen as the main point of reference for this implementation since the document also contains a commentary on each syntax feature. As I became accustomed to the S-algol syntax, the

S-algol implementation of the compiler also became an increasingly useful resource for understanding the grammar since it shows exactly how the structures should be handled.

3 Requirements specification

Compiler implementation is a task that has provably infinite scope: the optimisation stage of a compiler alone is infinite since the best optimiser would be able to delete infinite loops in code thereby solving the halting problem - a problem that is known to be uncomputable. As such the scope of the project must be focused. To start to pare down the project into specific goals, it is useful to examine the motivations for the project.

The principal motivation is that of preservation. It is desired that the S-algol language should remain usable into the future to facilitate access to those who are interested in what programming was like at St Andrews in the late 20th Century. This leads to the premise that the requirements of the compiler project should be orientated towards making the S-algol language as accessible as possible. This aim happens to be in-line with those of compilers for most programming languages: the success of a programming language and its compiler is directly proportional to its popularity and this is driven in a large part by how easy it is to start using the language.

The homepages of several popular programming languages reveal how they promote accessibility.`http://coffeescript.org` a small IDE to try out the language and a detailed documentation of the language source code. `https://www.haskell.org` contains a list of the language features, a REPL to try out Haskell and links to news, documentation and downloads.

From these examples, the following concrete requirements are derived:

1. A play-ground to write and run the S-algol code.
2. A list of the S-algol features.
3. Links to the S-algol source-code.
4. Starting points for using S-algol in production.

To start breaking down the requirements of the core compiler, it is useful to examine the different processes in the compiler. Compilers often operate as a chain of different sections, each of which handle different parts of the implementation. This includes, the lexer, parser, code generator, optimiser and static analyser.

The lexer is a program that breaks a program string into logical tokens. Lexers are often trivial programs and as such this should be fully implemented.

The next phase is the grammatical verification of the tokens. This requires a parser. A parser accepts tokens that are in an order that is defined to be correct according to a grammar. Parsers often follow a fixed structure, built around the grammar. As such, once a parser has been designed for a basic

part of the language, it is a matter of graft rather than design to finish its implementation. This means that a fully complete parser for the grammar should be a requirement.

After these two phases have been implemented, the 'middle' section of the compiler tool-chain is open for infinite amounts of development. Many algorithms can be designed to operate on the parsed code tree: it may be checked for type safety, it may be optimised or it may be re-formatted. The flexibility of this section means that the requirements for this part of the compiler must be more carefully managed. Some of the requirements for this section are 'nice-to-have' rather than critical. This allows some features to take longer than expected. Features that are necessary for the compiler to output correct code are prioritised, next, the original features of the S-algol compiler and finally, features that can further assist the programmer.

The final phase of the compiler must be the generation of the code that is to be run. In this case, the code is Javascript. The work required for this section is bounded by the number of constructs available in the S-algol language. However, some constructs - such as complex io - may require significantly more work than the others. This means that development time for the code generator is not as predictable as for that of the parser and lexer.

It is also important to test the compiler's components and to provide an end-to-end testing of the entire toolchain.

This evaluation of the compiler's scope yields the following requirements.

1. A lexer that can correctly break an S-algol program into symbols and keywords.

2. A parser that accepts the full S-algol grammar.

3. Analysis phases sufficient for correct outputting of code. This includes correct recognition of constructs such as functions versus variable application.

4. Code generation for as much as possible of the S-algol language.

5. A testing suite to verify correctness of the compiler and parts of the compiler.

The final consideration that is made during the requirements specification is that of how input and output interaction should be managed in the Javascript generated from S-algol. This can be considered part of the code generation since the implementation details only need to be addressed at this point, however, it does pose a significant hurdle in this part of the compiler.

S-algol has three first-class forms of input and output. One is writing to stdin and reading from stdout. Another (which might be considered an extension of the first) is file reading and writing. Lastly, S-algol allows images to be manipulated and rendered to the screen. The problems associated with these modes are rooted in Javascript's evented io model versus S-algol's blocking io.

Solutions will be addressed in the design section of this document. It is important at the requirements stage, to recognise that they may take longer to implement than expected.

The choice of io affects which Javascript environment, the compiler is most compatible with. In browser environments, io is mostly orientated around the DOM but logging to the console is also possible and works in a similar way to writing to stdout. In the node Javascript environment, io also come from files and stdin. Since the focus is on browser compatibility. Io shall be prioritised in the following order.

1. stdin/stdout reading and writing.
2. Image rendering.
3. File reading and writing.

There are some technical corollaries to these requirements that must be accounted for early in the process. The first of these is the choice of development language. The requirement that it must be possible to test-drive the compiler in a browser means that it might be sensible to use only languages that are compatible with Javascript. This means that all compilation and execution can happen in the browser without need for a back-end. There are advantages to this approach: statically served code is cheaper to host; it is less likely to break since a back-end represents a single point of failure; and there are no security or performance considerations necessary (the Haskell online REPL rate limits test programs so that people do not exploit the service).

4 Software engineering process

4.1 Tool Usage

Before starting programming it was necessary to choose the tools that should be used. Some were straightforward, others required some experimentation.

GIT is used for version control. This version control software is arguably the currency of modern open source projects. Since one the targets of the project is to release the code for access by others, it makes sense to use GIT to manage the software versions.

NPM is used for package management. This is the most popular Javascript dependency manager and has the best support from node.

TypeScript is used as the principal development language. The TypeScript project homepage states that "TypeScript is a typed super-set of Javascript that compiles to plain Javascript" [12]. This means that any valid Javascript is valid TypeScript but not necessarily vice versa.

Typescript features allow a programmer to give variables and functions explicit types. It also allows class-based inheritance as a substitute for Javascript's prototype-based inheritance. These features permit better static checking of the code since TypeScript has enough information to check whether functions and objects are being used properly within their scope.

This compiler-assistance is particularly useful when implementing a compiler. This is because a common design pattern in compilers is to tie features of a language's grammar to classes within. Abstract and concrete syntax trees may be built up of the these classes. By explicitly defining the classes, there is a compiler warning if access to the incorrect field of a node in a syntax tree is attempted. In this project, there are over nine hundred classes that are dynamically generated to define the concrete syntax of S-algol. It would be impossible a human to remember all these classes and their field names; fortunately, the WebStorm IDE has excellent TypeScript support and will automatically suggest the field names. There is no performance penalty of TypeScript since all type information is discarded at compile time, leaving plain Javascript.

A further advantage of using TypeScript is that it is a very modern language and provides and excellent point of comparison with S-algol and the progress that programming languages have made since 1979.

Mocha is a Javascript test-runner comparable to JUnit. This allows automated testing with structures results. It also integrates well with continuous integration services.

Circle CI is a free continuous integration service that run test on the codebase every time it is pushed to GitHub so that I could verify that everything was working correctly with each change.

5 Design

The S-algol source of the original compiler provides design cues that provide part of the inspiration for the design of this compiler. However, the constraints on this compiler and the original mean that they have somewhat different requirements. For example, the computational constraints on this compiler are different to the original: Morrison's compiler is optimised for performance on constrained hardware; this implementation has no such constraints since computers of today are orders of magnitude faster than computers of the past.

The differing requirements have meant that some design features have been changed in this compiler from the original.

5.1 Recursive descent

One of the key design premises of Morrison's compiler is the usage of "recursive descent". Recursive descent is structured — according to Morrison — such that, "for every syntactic construct there is a procedure in the compiler which will

analyse it"[13, p18]. These procedures contain code that consumes appropriate terminals by removing them from the input and non-terminals by calling the procedure that represents them.

Selecting an appropriate parsing technique is important for correct implementation of a compiler because some parsing techniques are not sufficiently powerful to be able accept some classes grammars. "Modern Compiler Implementation in Java" describes these grammar classes using the diagram in Figure 3 [1, p66]. This shows how grammar classes may be compared. For example, a parsing technique that accepts LL(k) grammars can also parse LL(1) grammars since it is a subset but might not be able to parse an LR(0) grammar because this only intersects with LL(K).

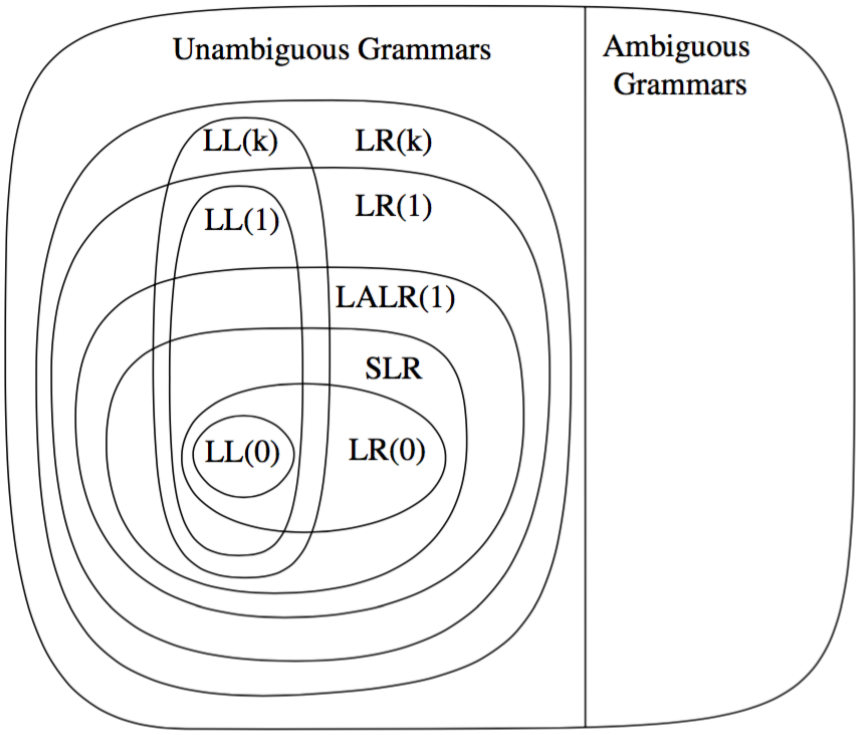

Figure 3: Grammar Classes

The S-algol grammar was a source of a design flaw in the parser section. Most of the S-algol grammar can be classified as LL(1) class but not quite all of it. During implementation, I decided to deviate from the recursive descent design in the parser section and uses a technique that automatically generates the recursive descent procedures. This implementation did not consider the fact that recursive descent can accept more than just LL(1) grammars. This caused

problems which are accounted for in the Parser section.

5.2 Single Pass vs Multi Pass

The original S-algol compiler was designed as a single-pass, recursive descent compiler. This design contrasts with multi-pass compilers.

A single-pass compiler performs compilation steps, lexing, parsing, type checking and code generation in one pass of the input code.

A multi-pass compiler does not perform all compiler stages in one pass. Instead, it abstracts compiler stages into separate passes and uses some intermediate representation of a program to communicate these stages. These representations are often trees of objects. Each pass traverses the intermediate representation appropriately and passes on the results to the next stage.

Morrison sums up the distinction between the two methods in his thesis: "The [single-pass] compiler is refined in layers rather than as separate passes" [13, p18]. He claims that single-pass implementation "makes the compiler faster but the real significance is that it makes the total code for the compiler smaller and easier to write." The performance argument was legitimate in 1979 but in 2016 it is less pertinant given orders of magnitude fast computing resources. The argument for a smaller and easier implementation also seems to be mitigated by the fact modern languages have more expressiveness for abstraction than S-algol and its contemporaries. This means it is easier nowadays to write more abstract programs and it is often better to use abstraction because it allows compilers to perform static analysis on code to pick up problems automatically.

On key design consideration is that single-pass compilers are less capable than multi-pass compilers. They lack the ability to check code in relation to code that they have not seen yet. For example, they cannot check that a function call is correct if the function is called before it is declared. This is why ALGOL languages often include "forward" declarations that inform the compiler in advance of a function's name and its type signature. Morrison points out that this is "awkward when recursive procedure definitions are involved" since a function must be available in scope so it can reference itself. The solution to this in S-algol is that "the identifier comes into scope after the parameter list has been specified allowing procedures to call themselves" [15, p24]. In multi-pass compilers, it is possible to infer forward declarations since they can do type checking after all the code has been seen.

A more subjective advantage of multi-pass compilers is that they offer a better structure for drawing abstraction between parts of a compiler. A type-checker might, for example, be an optional component that can be removed or replaced arbitrarily. In a single-pass compiler, each procedure might have to be modified to change such a feature. Similarly, if a different target language is chosen, it is simple to swap the component that deals with code generation for a different one.

Design patterns can also improve the abstraction of multi-pass compilers. Norman Neff describes how the visitor pattern can be used to abstract the traversal mechanism of a collection of referenced objects (such as a syntax tree)

from the operations on those objects. In the context of compiler design, Neff claims that "the visitor pattern gives the abstract syntax tree responsibility for defining a traversal sequence to be followed by all visitors" [17]. Neff points out however that the visitor pattern is limited to fixed orders of traversal: it is only useful if a programmer is doing many traversals using a fixed ordering such as depth-first or breadth-first traversal.

The visitor pattern gives the abstract syntax tree responsibility for defining a traversal sequence to be followed by all visitors. This pattern is used extensively by this project.

The expressiveness and abstraction available to multi-pass implementations justify its usage in this compiler.

5.3 Code Target

The original implementation of the S-algol compiler converts S-algol input to 'S-code'. This S-code is designed for execution on an S-code machine. This is similar to Java's execution model: Java code is compiled to bytecode and run on a JVM. The advantage of this approach is that virtual machines may be implemented for arbitrary hardware and operating system targets, allowing code in Java or S-algol to be run on any platform without changing the compiler implementation.

It would be possible to implement a similar approach for the Javascript compiler. It could could emit S-code and an S-code machine could be implemented in Javascript.

The advantage of this S-code based approach is precise control over execution. An S-code machine could execute instructions atomically and would have full access to the program state. As such, implementation of custom debuggers and other runtime tools would be possible. Furthermore, some of the more nuanced details of S-algol could be implemented in the machine rather than the compiler. Compilation of S-algol straight to Javascript, means this control is lost to the Javascript virtual machine. In this case, all code outputted by the S-algol compiler must precisely represent S-algol paradigms in terms of Javascript paradigms.

The advantage of compiling directly to Javascript is a smaller, simpler project that would allow the features that are already built in to Javascript virtual machines to be used such as debugging and state inspection. It is also likely to offer better performance than the S-code approach. This is because many features would have to be duplicated between the Javascript machine and the S-code machine. For example, they would both have to perform garbage collection, stack management and heap management.

Another key advantage of cutting out the S-code component is that the output code can work inline with any external Javascript or libraries. This is important because the Javascript ecosystem has a very established set of libraries and build tools available.

The simplicity of the direct-to-Javascript approach as opposed to attempting to implement the S-code machine is the main justification for this design

decision.

5.4 Overall Design

Given these decisions about individual aspects of the compiler, it is important to provide an account of how the components fit together.

The fundamental input to the compiler is S-algol code. This must be lexed into symbols. It then must be parsed into a concrete syntax tree which is an object-representation of the S-algol code.

Concrete syntax trees often have duplicated semantics, since there is usually more than one way of expressing something in code. This is true of S-algol, for example there are a couple of ways that loops may be defined or that arrays can be initialised. This means that it might make sense to convert the concrete syntax tree into an abstract syntax tree: while a concrete syntax tree directly represents the input program structure, an abstract syntax tree directly represents the semantics of the input program. This means that the abstract tree is necessarily at most the same size as the concrete tree but often considerably smaller. The advantage of a smaller tree is that static analysis stages and code generation can be made simpler with cases that must be handled.

The abstract syntax tree structure that is used for S-algol is defined in './src/sAlgolCompiler/AbstractSyntax.ts' and contains around 30 classes.

The output representation for the compiler should be Javascript. This could be output as strings by the code generation section. However, there is a standard set of classes called ESTREE which represent the Javascript syntax tree. Third party libraries such as escodegen accept ESTREE and produce well-formatted Javascript code. The advantage of producing ESTREE is that it is made up of classes and can be statically checked by TypeScript, strings cannot. Also, other third party tools such as Javascript minifiers can accept ESTREE, this could be useful for adapting the S-algol compiler in the future.

Given this overall structure, the diagram in Figure 4 can be constructed. It shows the stages of the compiler which must be implemented and the intermediate representations that are passed between stages.

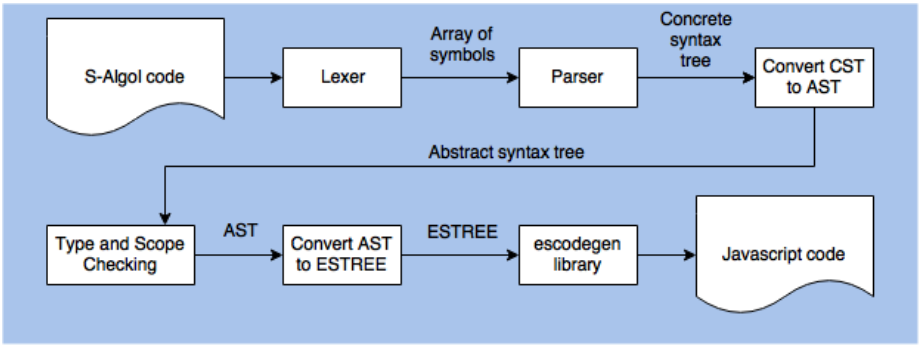

Figure 4: Project overview

6 Lexer

The role of a lexer in a compiler is to break up a continuous input string into lexical symbols that can be accepted by the parser. As such a lexer does not necessarily need to respect the grammar of a language. This assumption diverges from the original S-algol implementation. The single-pass nature of the original implementation means that the input string is lexed in the same recursive descent as all of the other language operations and, as such, the program knows which symbols should be expected to be lexed at any given point in the code. For example, if the program is currently executing the function that handles let declarations, it knows to expect the characters 'let' and nothing else; then the next string of characters will be a sequence of valid identifier characters. A marginal advantage of this method is that it might save a bit of computation since only a few possibilities need checking. However, it adds bulk to the parsing stage of the compiler and since the overall design of this new implementation is to emphasise separation of concerns over efficiency, it makes sense to implement a simple lexer and separate parser than to add a lexer to the already-complex parser.

Since the lexer should be able to lex input from any point within a program and does not have grammatical understanding of the input, there are some considerations that must be made to ensure the correct symbols are lexed. For example, when a lexer sees "let", it should produce a single LET symbol; but when a lexer sees 'foo', it should produce the three symbols F, O, O. As such, symbols can be broken into priority classes. It is common practice for a language to reserve a set of keywords that may not be used as identifiers. For example, let let = 1? is an invalid program because let cannot be used as an identifier.

6.1 Algorithm

The algorithm implemented by this lexer tries to recognise some class of symbol from the head of the input and if a match is found, it consumes that match from the head of the input and produces appropriate symbols. In this project, these symbols are implemented using an enum structure.

Each class of the language is represented by a regular expression. Listing 1 shows the regular expression that is used to recognise keywords. This class is checked first.

The expression is separable into three parts.

The initial character ^ asserts that the following regular expression should only match from the start of a line. This is important for efficiency and general functionality of the lexer: it is useless to recognise the string "; let a = 1;" as starting with a LET symbol because it does not.

The body of Listing 1 contains an enumeration of all the possible keywords. Full-stop literal characters are escaped since they are usually wild cards in regular expressions. An important consideration of the body of the regular expression is that the keywords are in reverse order of length. This means that if two keywords have the same beginning characters, the longest one will always

match. For example, the Javascript expressions " isnt ...". match(/^(is|isnt)/) will incorrectly match the head of the input as starting with an IS symbol whereas " isnt ...". match(/^(isnt|is)/) will correctly match the input string as starting with an ISNT symbol.

The final part of this regex is a negative lookahed (?![a-zA-Z0-9.]). This is a precaution that checks that the head of the input is not an identifier that starts with a keyword. If it is, no match is made by this regular expression. As such, this compiler accepts let letitia = 1? as a lexically and grammatically correct program that assigns 1 to variable letitia. Without the negative lookahead, this programme would not be acceptable.

Listing 1: Regular expression describing S-algol keywords

```
/^(read\.a\.line|structure|procedure|read\.name|read\.
    byte|default:|out\.byte|nullfile|forward|read\.32|#
    cpixel|maxreal|epsilon|read\.16|screen|maxint|repeat|#
    pixel|string|cursor|colour|rotate|out\.32|output|
    vector|out\.16|write|pixel|reads|readb|readr|image|
    limit|shift|scale|readi|begin|while|abort|false|peek|
    true|rand|onto|text|read|case|xnor|nand|from|copy|else
    |then|real|file|pntr|bool|isnt|end|xor|lwb|and|ror|upb
    |eof|int|off|nil|for|let|r\.w|i\.w|s\.w|s\.o|s\.i|div|
    rem|pic|nor|not|on|pi|do|is|by|if|to|of|at|or|in)(?![a
    -zA-Z0-9.])/
```

The next class of symbol is that of the punctuation symbols. The regular expression follows a similar structure. Here, order is important such that <= is recognised before <. Lookahead is not important because they cannot be part of an identifier since identifiers can only start with an alphabetic character.

Listing 2: Regular expression describing S-algol punctuation

```
/^(structure(|:=|::|++|!=|<=|>=|\*|;|:|~|{|}|@|=|!|#|$
    |%|&|\?|\+|-|\/|<|>|\[|\\|\]|\^|_|`|\||0|\(|\)
    |,|"|'|\.)/
```

The symbols that represent types are mostly just keywords, however, they may be prefixed by an arbitrary number of asterisks and c's. [*c]* matches such a prefix. It would be possible to write a regex that only lexes correct types ie not 'c*c*c*cint' but not 'ccccint' however, this check is done by the parser so it is an unnecessary complexity. Again, the type regex requires a negative lookahead check that it is not trailed by identifier characters such that the identifier of the declaration let introduction = 1? can be correctly lexed. The regular expression for types is shown in Listing 3.

Listing 3: Regular expression describing S-algol types

```
/^[*c]*(int|real|bool|string|pixel|pic|pntr|file|#pixel
    |#cpixel)(?![a-zA-Z0-9.])/
```

The final and most permissive regular expressions match identifiers and numbers. Listing 4 shows the expression that matches identifiers (which must start with alphabetic characters but may contain numbers and full stops).

Listing 4: Regular expression describing S-algol identifiers

```
/^[a-zA-Z][a-zA-Z0-9\.]*/
```

Listing 4 shows the expression that matches numbers including integers, floats and exponents. Examples of these are "1", "10.4" and "1.4e10".

Listing 5: Regular expression describing S-algol numbers

```
^(\+|-)?[0-9]+(\.[0-9]+)?(e[0-9]+)?
```

6.2 Comments

S-algol ignores strings that begin with exclamation marks until the next new line character. These may be used as comments. The lexer stage of the compiler removes these strings.

7 Parser and Meta-Compiler

During the implementation of the S-algol parser, I came across some problems that seemed like they would benefit from some automation.

A meta-compiler is a software component used to assist the construction of compilers. The meta-compiler component of this compiler is perhaps the more exotic section of the project. Its core function is to parse the BNF grammar for S-algol and programmatically analyse it. This allows many of the arduous tasks involved in writing a compiler to be abstracted using code. Furthermore, it generates code that helps with the implementation of the other stages of the compiler. This section contains a more detailed break-down of the problems involved.

7.1 Motivation

It was mentioned in the design section of this report that the core design principal of the original S-algol parser was the use of a handwritten recursive descent structure. Inspired by this, the initial implementation of the Javascript compiler was a handwritten recursive descent parser that implemented a trivial subset of S-algol. This code was designed to accept a string of symbols from a separate lexer and produce an abstract syntax tree to represent the input code. It would also detect context-free errors in the grammar. Having implemented this experimental design, it became clear that this approach is problematic.

Firstly, the process of writing functions to represent syntax is extremely formulaic: the structure of the recursive descent is a direct mapping from the syntax notation to functions. The effect of this is that it is easy to be inconsistent

and cause confusion, especially regarding naming conventions and the design of syntax tree classes. It is not semantically obvious, for example, whether a recursive grammar fragment that represents a string of "bar"s (<foo> ::= bar <foo> | bar) should be represented as

```
public class foo {
        bars: string[];
}
```

or as

```
public class foo {
        bar: string;
        foo: foo;
}
```

"Modern Compiler Implementation in Java Second Edition" provides a good set of conventions for doing this mapping which settles the consistency issue. These are laid out in Figure 5. However, the implementation is arduous and hard to change once completed.

1. Trees are described by a grammar.
2. A tree is described by one or more abstract classes, each corresponding to a symbol in the grammar.
3. Each abstract class is extended by one or more subclasses, one for each grammar rule.
4. For each nontrivial symbol in the right-hand side of a rule, there will be one field in the corresponding class.
5. Every class will have a constructor function that initializes all the fields.
6. Data structures are initialized when they are created (by the constructor functions), and are never modified after that (until they are eventually discarded).

Figure 5: Conventions for representing tree data structures in Java. [1]

Another point of difficulty is the process of recognising which production should be applied in a given parsing situation. Take for example the simple S-algol program in Figure 6 that assigns "1" to the variable "a".

Listing 6: Simple Program

```
let a = 1?
```

The entry production for S-algol is <program>. The productions in Listing 7 make up the grammar subset that is required to parse the given section of S-algol.

Listing 7: Simple S-algol grammar subset

```
<program> ::=
    <sequence>?
<sequence> ::=
    <declaration>[;<sequence>] |
    <clause>[;<sequence>]
<declaration> ::=
    <let_decl> |
    <structure_decl> |
    <proc_decl> |
    <forward>
<let_decl> ::=
    let <identifier><init_op><clause>
```

The difficulty parsing this program lies in the fact that each production may represent multiple grammar structures. This means that every time a production is parsed, the input code must be inspected to work out which structure within a production should be used. For example in Listing 7, before a <declaration> is parsed, it must be programmatically intuited which of the four types of declaration is implied by the input code.

Listing 8 contains a fragment of a recursive descent compiler that might be able to handle the parsing of a let declaration.

Listing 8: Recursive descent fragment

```
var input = ['let', 'a', '=', '1'];

function program(): Program {
    return new Program(sequence());
}

function sequence(): Sequence {
    if (isDeclaration(input[0])) {
            return new Sequence(declaration(),
                sequence());

    } else if (isClause(input[0])) {
            return new Sequence(clause(), sequence())
                ;

    }
}
```

In Listing 8, the production recognition is abstracted by the functions, "isDeclaration" and "isClause". The implementation of the functions for this example seems fairly straight forward. Listing 9 shows a possible implementation that returns true if the input code begins with "let".

Listing 9: Possible 'isDeclaration' implementation

```
function isDeclaration(): boolean {
    return input[0] === "let";
}
```

This seems sufficient to parse the code in Listing 6, because we know that if the leading symbol is currently equal to "let", we are dealing with a declaration. But the real definition of "isDeclaration" should be as shown in Listing 10 because there are four different ways that a <declaration> production may begin.

Listing 10: Full 'isDeclaration' implementation

```
function isDeclaration(): boolean {
    return input[0] === "let" ||
        input[0] === "forward" ||
        input[0] === "procedure" ||
        input[0] === "let";
}
```

For a complete parser, it is necessary to write such recognition functions for every production. This ends up being time consuming especially for productions for which there is a lot of left recursion - where the first symbol is a non-terminal, which in turn points to many more non-terminals which have productions for which the first symbol is non-terminal.

The nature of these problems implies that they might be best solved algorithmically. Given the repetitive, time-consuming work that would be required, it seems obvious that at least the parser stage should be generated programmatically from the syntax notation. This approach has the following advantages:

- Implementation of parser generator meta-compiler would take equal or less time to hand writing parser.
- The generated representation of the concrete syntax would be consistent and changeable by simple changes to the meta-compiler.
- An object-based representation of the grammar syntax would allow production recognisers such as 'isDeclaration' to be generated programatically.

The integration of the meta-compiler can be expressed by the extension of the overall project design diagram shown in Figure 6.

7.2 Basic Implementation

The first requirement of the meta-compiler is to implement a small compiler chain for the BNF representation of the S-algol grammar. This is a simple implementation since the grammar has a small syntax. Non-terminals are surrounded in angle brackets; they are assigned using the "::=" operator; square brackets represent optional syntax; and "*" indicates possible repetition.

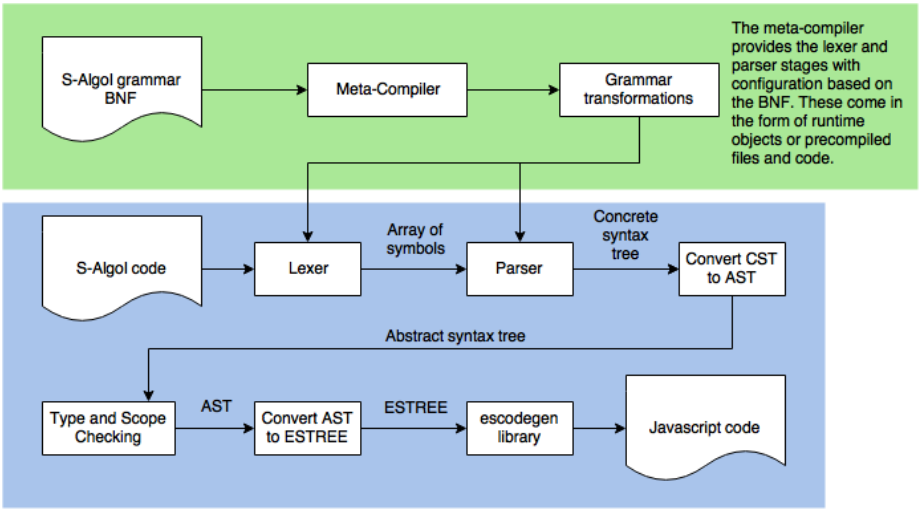

Figure 6: Project Overview With Meta-Compiler

The meta-compiler parses this syntax and produces a representation using objects: the production shown in Listing 11, for example, is converted into the object structure in Listing 12.

Given this object-representation of the grammar, it is possible to algorithmically solve the problems encountered whilst hand-writing the parser.

Listing 11: Simple production

```
<let_decl> ::= let <identifier><init_op><clause>
```

Listing 12: JSON representation of S-algol grammar fragment

```
var grammar = {
        ...
        "<let_decl>" : {
                productions: [
                        [
                            {value: "let"},
                            {value: "<identifier>"},
                            {value: "<init_op>"},
                            {value: "<clause>"}
                        ]
                ]
        }
        ...
}
```

7.3 Production Recogniser

The first problem tackled in the meta-compiler is that of production recognition. The meta-compiler provides an automated generation of functions such as "isDeclaration", shown in Figure 10. The task is to find a function that takes the current expected production (initially, always <program>) and the current first value in the input array then to output which production will parse the current input; just as "isDeclaration" does.

Within the meta-compiler, this function mapping is represented by a precomputed table. A subset example of this table is displayed in Figure 7. Usage of the table is simple:

1. Look up the current production in the left hand column.
2. Look up the first symbol in the input sequence in the centre column.
3. If there is no entry, throw a parser error.
4. For each element of the production in the right hand column:
   (a) If the symbol exists and is a non-terminal, use this non-terminal as the current production and return to step 1.
   (b) If the symbol exists and is a terminal, consume the first character in the input sequence and move to the next element in the grammar.
   (c) Otherwise, throw a parser error.

An example of this algorithm might be as follows. A program let a = 1? is assumed to be a grammar fragment of type <program>. The leading symbol, "let", is then looked up in the table for <program>. In this table, "let" maps to <sequence>. This identifies the beginning of the list to be of type <sequence >. It is then possible to look up "let" in the <sequence> table, to get < declaration>. This continues until the fragment is recognised as a <let_decl>. The <let_decl> production starts with the "let" non-terminal. This allows the parser to consume the "let" symbol and begin to parse the < identifier >.

The algorithm represented by this example shows how the recogniser table can be used to effectively configure a generic parser. Details of this parser will be discussed in a later section.

7.4 LL-ness of the S-algol Syntax

The basic meta-compiler design appears to describe an effective automatically generated parser implementation. Indeed, when planning the design, I anticipated this to be sufficient for a working parser implementation. However, this represented a misunderstanding of how expressive grammars can be.

Grammars can be classified based on the methods that are required to recognise them. LL(1) grammars are grammars that can be parsed simply by observing the first symbol of the input. This first symbol will always reveal how to parse the given section of the grammar.

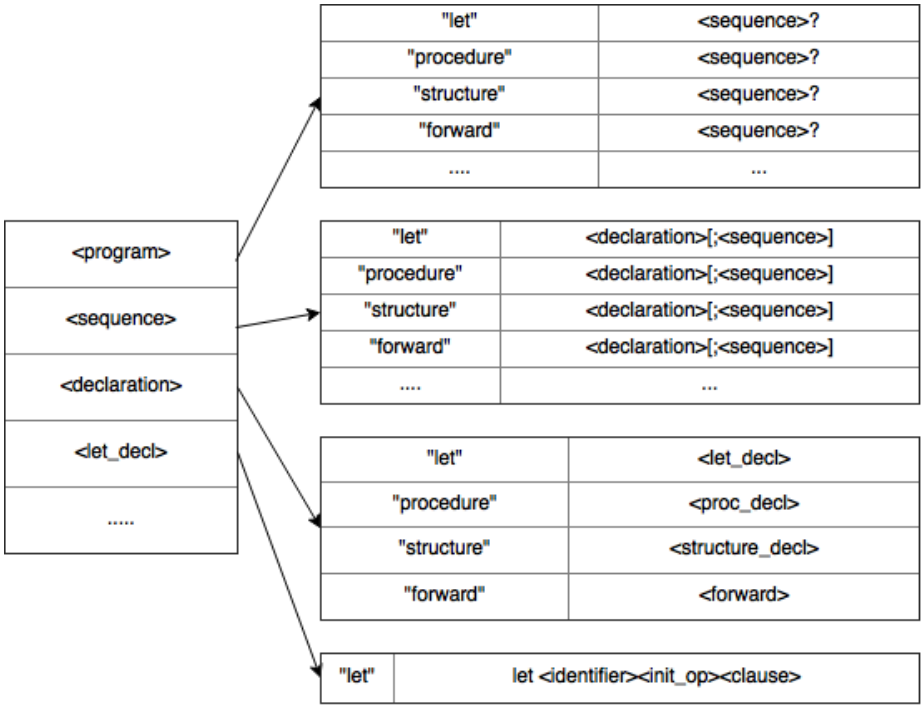

Figure 7: Recogniser table subset

The parser implementation approach described above can handle LL(1) grammars. Recursive descent can handle a strictly larger variety of grammars. This means that the S-algol syntax as documented by Morrison is not necessarily always LL(1).

A non-LL(1) language feature in S-algol is the clause production. Listing 13 shows the non-LL(1) fragment. Significantly, it is impossible to tell if an input is an "if, do" clause or an "if, then" clause by simply examining the first symbol - both of them begin with "if".

Listing 13: Fragment of the ¡clause¿ production

```
<clause> ::=
if<clause>do<clause> |
if<clause>then<clause>else<clause> |
...
```

Listing 13 produces a recogniser table containing the fragment shown in Figure 8. This clearly shows the conflict that would make the suggested parsing algorithm not work.

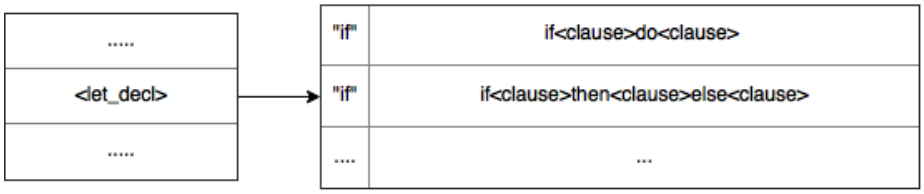

Figure 8: Clause recogniser table subset

Fortunately, it is possible to convert some non-LL(1) grammars into LL(1) grammars using factoring. Listing 14 shows the factored clause production. This effectively defers the decision as to whether an input sequence is an "if, do" clause or an "if, then" clause until they actually become distinct — after if<clause>.

Listing 14: LL(1) clause production

```
<if_tail> ::=
do<clause> |
then<clause>else<clause>

<clause> ::=
if<clause><if_tail> |
...
```

7.5 Left recursion

The other problem with generating an unambiguous S-algol recogniser table is that of left recursion. This occurs when there is a cycle in a grammar, appearing

on the left hand side of a production. A simple example of direct left recursion example is <expression> ::= <expression>()|<number>. This example produces an infinite loop and thereby a stack overflow in the naïve meta-compiler implementation described above. This is because to recognise an expression, it must be possible to recognise the first symbol of all of its productions. In this case, one of these is an expression so to recognise an expression, it must be possible to recognise an expression — that is circular.

The instance of this problem in the S-algol grammar lies in the <expression> syntax although it is not as obvious as the previous example. Figure 15 shows the expression production and the "exp" productions that establish operator precedence. By proxy of these "exp" productions, it is possible that an expression starts with an expression. This is called indirect left recursion. This causes the same infinite-loop problem with the meta-compiler.

Listing 15: Left recursion in expressions

```
<expression> ::= <exp1>[or<exp1>]*
<exp1> ::= <exp2>[and<exp2>]*
<exp2> ::= [~]<exp3>[<rel_op><exp3>]
<exp3> ::= <exp4>[<add_op><exp4>]*
<exp4> ::= <exp5>[<mult_op><exp5>]*
<exp5> ::= [<add_op>]<exp6>
<exp6> ::=
...
<expression>(<clause><bar><clause>) |
<expression>(<dereference>) |
...
```

This exact issue is addressed by Morrison in "Recursive Descent Compiling" [5, p84]. Morrison's solution was to defer the detection of opening brackets until a full expression had been parsed. Fortunately, there is a way of expressing this solution in a way that is compatible with the meta-compiler design. It can be done by lifting the bracketed section into a new production above <exp6>. This has the effect of parsing <exp6> before the brackets are parsed. It works because none of the other "exp" expression could yield a value that can be access like a string or an array. A side effect of the approach is that a wider set of inputs is accepted by this new grammar. This is acceptable however because the incorrect inputs may simply be caught during the type checking phase. Listing 16 shows the expression grammar with left recursion removed.

Listing 16: Non-Left-Recursive Expressions

```
<expression> ::= <exp1>[or<exp1>]*
<exp1> ::= <exp2>[and<exp2>]*
<exp2> ::= [~]<exp3>[<rel_op><exp3>]
<exp3> ::= <exp4>[<add_op><exp4>]*
```

```
<exp4> ::= <exp5>[<mult_op><exp5>]*
<exp5> ::= [<add_op>]<exp5a>
<exp5a> ::= <exp6>[(<clause><bar><clause>)][(<dereference
    >)]
<exp6> ::=
...
```

A similar solution is used to make the grammar non-left recursive for assignment clauses that may have an expression on the left hand side.

7.6 Class-Based Representation of S-algol Grammar

The advantage of using TypeScript over regular Javascript is that it has type-checking. This means that the TypeScript compiler can inform a programmer of problems such as incorrectly accessing an object or returning the wrong type of object from a function. Furthermore, this support can be integrated into text editors. WebStorm, for example, has automatic code suggestion which can auto-complete function and variable identifiers. This sort of support is not available in regular Javascript.

To take advantage of these features, classes must be defined so that TypeScript knows which fields and functions an object should have. To help implement of the rest of the compiler, the meta-compiler generates a class structure to represent the S-algol grammar. This makes any operations on the concrete syntax are type-safe. This is a marked advantage of implementing a custom meta-compiler rather than using a third-party library.

8 Code Generation

Despite many similarities, some parts of S-algol do not have clear equivalent implementations in Javascript. This means that code generation is less well bounded than the parser or lexer: each grammar structure must be examined and converted. This meant that most of the development time spent on this project was spent on this component. This section of the report details how these conversions are produced.

In many cases, where there are features missing from this compiler, it is because there was not time to implement a Javascript equivalent. In some case, such as the image manipulation constructs, a considerable amount of library code implementation would be required.

8.1 Clauses as Expressions

In S-algol, some clauses can act as expressions with return types. Conditional statements can yield a result which can be applied to a variable or argument. Listing 17 is an example of a trivial S-algol program that assigns 1 to variable a through a conditional statement. (Note that only “if, then, else” statements can yield values since “if, do” statements do not have an alternate value to yield).

An equivalent Javascript program might use the Javascript ternary operator as shown in listing 18.

Listing 17: S-algol conditional assignment

```
let a = if true then 1 else 2?
```

Listing 18: Javascript conditional assignment

```
var a = true ? 1 : 2;
```

This is a correct mapping however the Javascript ternary operator cannot not handle more complicated expressions. The code in listing 19 represents a marginally more complicated statement. Here, the result of the if expression is simply "34", however, two statements have yielded this number rather than just one in the previous expression.

Listing 19: More complicated S-algol conditional assignment

```
let a = if true then {let a = 34; a} else 2?
```

There is no idiomatic equivalent in Javascript: it is not possible to add a block statement to a ternary operation. However, it is possible to use a function instead. A function can be seen as an abstraction over a set of statements, just like a block statement except blocks are not reuseable, nor can they accept arguments. As such, the Javascript program in Listing 20 might be an appropriate equivalent to the S-algol in Listing 19. The function "temp" is used as an abstraction over the compiled S-algol block statement.

Listing 20: Javascript conditional assignment using a function

```
function temp() {
    var a = 34;
    return a;
}

var a = true ? temp() : 2;
```

Since this "temp" function will necessarily only be used once, it is better design to actually remove it from the namespace altogether to prevent collisions with other function names. This is possible with a some syntactic sugar available in Javascript shown in Listing 21. An anonymous function is declared and called in a single expression. Note the application of the anonymous function is triggered by the parentheses on line 4.

Listing 21: Javascript conditional assignment using an inline function

```
var a = true ? function() {
    var a = 34;
    return a;
}() : 2;
```

This ternary design scales to arbitrarily large conditional statements and is used as the implementation of conditionals in the Javascript compiler.

8.2 Scoping

There is an advantageous side effect to using Javascript functions to express S-algol blocks. This lies in Javascript's scoping paradigm. In the S-algol spec, Morrison describes the language's very simple scoping system: "the scope of an identifier starts immediately after the declaration and continues up to the next unmatched } or 'end"' [13]. As such, S-algol may have indefinitely-many scopes in a single program. In contrast, Javascript's two scopes are described in a Microsoft reference document as "global and local. A variable that is declared outside a function definition is a global variable, and its value is accessible and modifiable throughout your program. A variable that is declared inside a function definition is local...is created and destroyed every time the function is executed, and it cannot be accessed by any code outside the function" [11].

In this compiler implementation, scope checking is executed statically so it is not necessary in most cases to ensure that the scope is verified at run-time. However, there are some cases where the more-permissive Javascript scope will cause interference between variables. For example the S-algol programme in Listing 22 would be expected to print 5 then 1.

Listing 22: S-algol scoping

```
let a = 1;

if true then {
        let a = 5;
        write a;
}
write a?
```

However, a naïvely-compiled Javascript program in Listing 23 would print 5 then 5. This is because the if block is assigning variables in the global scope whereas in S-algol, the if block creates a new scope. (It is important to note that child blocks in S-algol do inherit scope from their parents such that if Line 4 were removed from Listing 22, the program would simply print 1 then 1.)

Listing 23: Javascript scoping

```
var a = 1;

if (true) {
        var a = 5;
        console.log(a);
}

console.log(a);
```

To coerce the Javscript program into acting more like S-algol, it is possible to take advantage of the function-local variable features of Javascript. The Javascript code in Listing 24 prints 5 then 1, just as was expected from the

S-algol program in Listing 22. Furthermore, the inner scope inherits scope from its parents just like S-algol scopes. As such, you can delete Line 4 from Listing 24 and the program will print 1 then 1.

Listing 24: S-algol-like Javascript scoping

```
var a = 1;

(function() {
        if (true) {
                var a = 5;
                console.log(a);
        }
})();

console.log(a);
```

Single-use anonymous functions provide convenient S-algol-like scoping and allow abstraction over block statements such that they can be used as expressions.

8.3 Input and Output

Morrison raises the problem of io in his PHD thesis. He addresses the fact that io interfaces for languages tend "to reflect the environment in which they were designed". As such he has optimistically tried to design S-algol to have "an extremely simple I/O system" so that "together with the abstraction facilities in the language, it will be powerful enough to handle any environment." Morrison adds wistfully that "This is perhaps a forlorn hope" [13, p16]. Indeed when it comes to Javascript, this goal is perhaps somewhat forlorn.

Javascript has specific paradigm of handling asynchronous operations that is different to other languages: it is evented. This means that when a piece of io is required, a callback function is attached to that io. When the io is ready (the data has been downloaded or the user input has been collected), the callback function is executed with the io data as an argument to the function. Within the Javascript run-time, there is a event loop which handles this: every iteration of the loop checks for available io and callbacks which have subscribed to that io. If matches are found, the callbacks are executed sequentially.

The implication of this is that it is not possible to have blocking io in Javascript. Listing 25 shows the S-algol blocking io and Listing 26 shows the equivalent using Javascript evented io.

Listing 25: S-algol blocking io

```
let a = readi;
write a + 1?
```

Listing 26: S-algol evented io

```
rl.on('line', function(line){
    console.log(line);
});
```

The differences between the models are just sufficient to make it very difficult to compile arbitrary uses of blocking io to Javascript. The same problem exists for file reading which works on the same principles in both languages.

There are a few possible solutions to this problem. The first might be to re-implement this compiler using a virtual machine as the target. The more precise control over execution that this affords would make it possible to manage asynchronous functionality. This is because the machine would be able to pause and play execution whilst waiting for io. This however would add considerable work to the implementation that might mean that development could not be completed.

Another possible fix might be to throw away the readi function from S-algol and only support Javascript-esque io. This is based on the premise that all the language features used in Listing 25 are also available in S-algol; most importantly, the passing of functions.

The code in Listing 27 is a valid S-algol program. the first line declares the existence of a function called "readLine" that accepts a function which accepts a string. The next line defines a function called handler that accepts a string and writes that string to stdout. The final line executes the function "readLine" with "handler" as its argument. The implementation of "readLine" can be written in Javascript as a library function as shown in Listing 28.

Listing 27: Javascript-esque io

```
forward readLine((string));

procedure handler(string input); write input;

readLine(handler)?
```

Listing 28: readLine implementation

```
var readLine = function (handler) {
    rl.on('line', handler);
}
```

This latter approach is easy to implement but it raises the question as to what makes S-algol, S-algol. Clearly, this approach breaks compatibility with S-algol programs that have been written using blocking io. However, it does allow a programmer to take advantage of the more-efficient io paradigm that is built into Javascript.

Given that the evented approach is simpler to implement, this compromise was selected, and this implementation of S-alogl does not support any of its original file reading or writing functionality.

8.4 Loops

Programming languages commonly have several different types of loop. S-algol has three types. Different loops generally only exist as syntactic sugar. Often, all loops can be translated to a while loop. As such, during the compilation of S-algol, all loops are represented by a single abstract syntax class. This class has three sections. The first is a clause that always executes at least once, the second is a boolean test that conditionally ends the execution, the third is a clause that only executes if the test is initially true and then executes until it becomes false. These three sections are the essential parts of an S-algol loop and allow all types to be compiled using the same code.

Listing 29: Loops as an expression

```
let x = 0;
while x < 10 do  {x:=x+1; x}
write x?
```

The code in Listing 30 shows Listing 29 as it is compiled into Javascript. The loop structure uses closure provided by functions in Javascript in the same way that conditional clauses do.

Listing 30: Javascript loop implementation

```
var x = 0;
function () {
    while (true) {
        var \$ret;
        if (!(x < 10)) {
            return \$ret;
        }
        \$ret = function () {
            x = x + 1;
            return x;
        }();
    }
}();
write(x);
```

8.5 Implementing 'Abort'

In S-algol, the abort keyword stops execution permanently. There are a few candidates for replication of abort in Javascript. In node Javascript, the environment object called "process" has a member function called "exit" which behaves like the POSIX "exit" function. It allows a node program to end abruptly with a return value. This does not unfortunately work in the browser. Javascript also has a GOTO-like syntax called a label that allows the execution flow to jump between line numbers, however, this does not allow exit from functions so

cannot be used to guarantee that a program is stopped from anywhere within the control flow.

This S-algol implementation uses the Javascript exception handling to allow halt execution. If an exception is thrown, execution is passed to the most-recently-entered try-catch block. To support the abort feature, the compiler wraps the compiled Javascript program in such a try-catch block and prints the appropriate output. This means that any abort clause may simply be compiled to a Javascript "throw" clause.

8.6 Vector Implementation

S-algol vectors are typed, multidimensional and of fixed length. These features are common to most array implementations. The S-algol vector differs from such implementations in a couple of ways. Firstly, they have customisable origins. Most languages use zero indexing of arrays, such that the first element of an array is referred to as element 0. S-algol allows the index of the first element of an array to be set programmatically. Listing 31 shows three arrays being assigned to variables named a, b and c. The arrays are each of type *int and each have a length of 3. The values contained in each are 1, 2 and 3. The difference between the arrays is that their first elements are indexed by different numbers. This number is prescribed by the expression after the '@' symbol. The first element of a has an index of 1, the first of b has an index of 100 and the first of c has an index of -60. The program will print three "1"s.

Listing 31: Array Initialisation

```
let a = @ 1 of int [1,2,3];
write a(1);

let b = @ 100 of int [1,2,3];
write b(100);

let c = @ 40 - 100 of int [1,2,3];
write c(-60)
?
```

This functionality provides more complexity for the compiler design. The S-algol vectors cannot be directly compiled to Javascript because it has no such built-in method for customising the index scheme. As such, the index "offset" must be stored such that it can be used to translate indices into Javascript's zero-indexed array scheme.

Another difference between the implementations is that Javascript does not implement fixed-length arrays and simply resizes the array as required by the program at runtime. In S-algol however, vectors are of fixed length and an exception should be raised if illegal access is attempted.

To support these features, an S-algol vectors is compiled into a Javascript object rather than an array. This object stores the length of the array, the lower

bound and the array values. It has a function "get" that allows the vector to be access and throws and exception if an illegal access is attempted. It has a function "set" that allows array values to be updated.

This approach supports multi-dimensional arrays" In the case of higher dimensional vectors, the Javascript objects store arrays of other Javascript objects that represent high dimensions.

From the perspective of usability, it is not clear why the decision of customisable indexing has been made. It is possible that it is to settle any argumentation over whether arrays should be 0-indexed or in fact 1-indexed. Indeed, S-algol was published at a similar time to Dijkstra's article "Why numbering should start at zero" [6] which presents an argument as to why arrays should be indexed from zero. Clearly, at this time zero-indexing was less of an assumption than it is today.

9 Static analysis

The lexer, parser and code generation phases are, for the most part, necessary and sufficient for an end-to-end compiler implementation. However, there is much further development possible in the static analysis of the code. Static analysis is the inspection of code without running it and can be used to report errors to a programmer that will cause the program not to execute or to execute in unexpected ways.

9.1 Implementation Structure of Static Analysis

Most static analysis requires one or more passes through the abstract syntax tree representation. Since the ordering of these passes is generally the same, it makes sense to use the same visitor pattern that was used in the meta-compiler to abstract the analysis of each node in the tree from the traversal mechanism. Using this pattern, a visitor object may be implemented with a set of functions called afterVisit <node> and beforeVisit <node> (where <node> is the name of an abstract syntax tree type). This allows a visitor some flexibility in whether it needs to traverse the tree in-order (left-to-right) or post-order (right-to-left). For example, to work out the actual return type of a procedure, it may be necessary to visit the procedure after its body has been visited.

More generic functions are also implemented such as "afterVisitNode" and "beforeVisitNode". These are called for every tree node visitation. The utility of these functions is demonstrated in Listing 32 which shows a traversal that touches all syntax nodes and treats them as the same type (since all nodes may have errors).

A further implementation detail that seemed appropriate was to use classes to represent errors. This means that meta-data can be stored within error messages such that they can be made informative. Furthermore, the instantiations of these errors are stored within the abstract syntax objects as an array. This means that if - for example - there is a type error on an operation, the error

will be stored in the node that represents that operation. It makes sense to keep errors tightly bound to syntax tree nodes in this way since it maintains an implicit ordering of errors (inline with the flow of the original program) and also allows for simple inspection of nodes for testing purposes.

Given this method of storing errors, a visitor class is implemented to traverse the tree and output error messages to std.err. Listing 32 shows the code that prints the errors. It also demonstrates the terseness of the code that can be written using the visitor pattern: most of the complexity of this feature is wrapped up in the visit function.

Listing 32: Error outputting visitor

```
export class ErrorOutputting extends SuperVisitor {
    foundErrors = false;

    afterVisitNode(node: A.AbstractSyntaxType) {
        if (node.errors && node.errors.length > 0) {
            this.foundErrors = true;
            for (let error of node.errors) {
                console.error(error.toString());
            }
        }
    }
}
// the visit function comes from a separate module
    that implements the abstract tree traversal of the
     visitor pattern
// the abstractSyntaxTree is an S-algol program in
    tree form
visit(abstractSyntaxTree, new ErrorOutputting());
```

9.2 Scope Checking

A basic type of static analysis is checking for scope irregularities. This might pick up problems such as a variable, procedure or structure being operated on, applied or initialised when it has not already been declared. let a = b + 1? is an example of a program that might trigger a scoping error. Here, the variable b is referred to despite the fact that it has not been yet been declared.

S-algol has a simple block-scoping mechanism. A begin keyword or { starts a scope that continues until the next unmatched end keyword or }. Any variable declared within a scope is available until the end of the scope and in all child scopes that exist within it.

Listing 33 represents a correctly scoped program. Variable a is declared and initialised in the outer scope, updated in a child scope and written to std.out. The program will print '2'.

Listing 33: Correctly scoped program

```
let a = 1;
if true do { a = 2 };
write a?
```

Listing 34 represents an incorrectly scoped program. Variable z is declared within a child scope and an access attempt is made in the parent scope. By the time the access attempt is made, that variable is no longer in scope.

Listing 34: Incorrectly scoped program

```
if true do { let z = 3 };
write z?
```

In this compiler implementation, scope-checking has a two fold advantage. Primarily, it helps a programmer catch bugs that they may not have noticed. However it is also a requirement for a correct implementation. Since Javascript is more permissive than S-algol, an incorrectly-scoped program may fail silently. For example, the incorrectly scoped program in listing 34 might be compiled to the Javascript code shown in Listing 35. This - when executed - will print '3' as a programmer might have expected from the original S-algol code. This, however, is incorrect usage of the language since the variable z should ordinarily have left the scope. Such usage should be discouraged since it might lead to incompatibility with other S-algol implementations.

Listing 35: Possible compilation of Listing 34

```
if (true) {
        var z = 3;
}
console.log(z);
```

The implementation of the scope checking algorithm is straight forward. Before traversal, an empty stack is initialised. The abstract syntax tree is then traversed in-order.

When the program is entered and at every entry to a scope, a dictionary object is pushed onto a stack. Every declaration of a variable, procedure or structure that is encountered is recorded in the top-most dictionary object of the stack using the identifier as the key and any meta-information as the value.

The identifier of every variable, procedure or structure access is checked against the objects in the stack from the top to the bottom. If no objects in the stack contain the identifier then it is not available in scope and there is a scope error.

Whenever a scope is exited, an object is popped from the stack. This causes all variables that were declared in that scope to be no longer visible to the scope checker.

9.3 Type Checking

Type checking finds problems in programs that are caused by incorrect passing of literals or identifiers. For example, a variable x may be declared as type a and operated on as though it were some incompatible type b. This represents a type checking error.

The implementation of type checking has a very similar mechanism to scope checking. This is because type checking also relies on keeping track of the current variable scope just as much as scope checking does. However, in the type checking implementation, more detail must be kept about the declarations. In this compiler implementation, the abstract syntax tree objects of declarations are kept as values of the objects in the scope stack that represent the current scope and all parent scopes. These abstract syntax tree objects contain all the relevant meta-data about a declaration. For a procedure, this would be its signature and return type; for a variable, its type; and for a structure, its signature. As such, when an application of one of these language features is encountered, it can be checked for correctness based on this meta-data.

To a large extent, type checking is an aid to the programmer rather than the compiler. In most cases, the compiler does not need to know about typing to correctly compile a program. In some cases however, it is necessary. Listing 9.3 shows such a case. This program uses S-algol's support for passing procedures as arguments to other procedures. The high-level procedure called doAdderThenMultiplyBy2 accepts a procedure and a number, executes the procedure on the number and multiplies the result by 2. However this syntax conflicts with another syntax feature, that is that procedures in S-algol may be called without use of parenthesise. As such, another understanding of this program might be that addOne is executed in line 9 and the result is passed into doAdderThenMultiplyBy2. Clearly the latter understanding would cause a type error and the former would not but since the two understandings of the program imply very different semantics, types must be analysed to produce correctly output code.

```
procedure addOne(int -> int); {
        a + 1
};

procedure doAdderThenMultiplyBy2((int -> int) adder; int
    x -> int); {
        adder(x) * 2;
};

let result = doAdderThenMultiplyBy2(addOne, 5);
write result?
```

9.4 Error Outputting

An important part of any programming language is the generation of helpful error messages for a programmer. It is often not enough to know simply that there has been a error; it is useful to know that there has been a certain type of error on a specific line with a specific operation. This helps a programmer fix the mistake and write correct code. Sometimes a compiler might even be able to offer advice as to fixes that a programmer can make. Often however, it is hard to infer the intent of a programmer from a piece of incorrect code.

This compiler provides three types of error message. The first is that of errors triggered during parsing. For example, if the input program does not have valid syntax. The second type is errors picked up by static analysis. For example, a type error. The third, is a runtime error. These may be triggered by an incorrectly accessed array. Runtime errors cannot be inferred at compile time because they require evaluation of the program which is a generally uncomputable problem.

Listing 36 shows some error messages that might be output by this compiler. S-Algol code that yields such errors can be found in Appendix A.

Listing 36: "Error Message examples"

```
[Parser Error] The input program is not complete.
[Parser Error] Line 0: 'semi_colon' is not a recognisable
    way of starting a <write_list> production.
[Error] Could not execute ADD on int and bool.
[Error] No function or variable named 'a' found in scope.
[Error] The vector 'a' only has 0 dimension(s). You tried
    to access dimension 1.
[Error] Could not apply 2 arguments to 'a' since it only
   usually takes 1.
[Error] Could not apply expression of type string to
   argument of type int as argument number 1 in the
   method a.
[Runtime Exception] Program attempted to access array
   index that is out of bounds.
```

A limitation in the implementation means that line numbers do not get passed from the concrete syntax tree to the abstract tree during conversion. Since type checking is performed on the abstract tree, line numbers are not available here. Given more time, it would be possible to make the static analysis error messages more informative by providing line numbers.

10 Evaluation and critical appraisal

Considering the ambition and scope of the project, I am pleased with the outcomes. The mitigating factor on the success of the project is that the overall

tool-chain does not work for all S-algol inputs. Unfortunately this manifests itself like an iceberg such that the small tip appears temperamental but the bulk of the project is of a high standard and robust: the lexer and parser stages and overall tool-chain are mature and work reliably; however, the code generation and static analysis do no fully conform to the S-algol specification. This makes the code overall seem buggy and non-functional because there are cases where the compiler cannot compile inputs.

Completion of the compiler however is only a question of more time rather than that of severe design flaws in the project. As mentioned in the testing section, this time would be best spent running the compiler on the repository of real-life code. Also, the implementation is sufficient that it can be useful to people interested in S-algol which was the fundamental requirement.

10.1 Use of the meta-compiler

The most ambitious aspect of this project is the meta-compiler implementation. It was not so much ambitious in its size but in its difficulty. I would argue however that it was worth implementing since it took around the same amount of time as it might have to hand-write a recursive descent parser. Furthermore, it allowed me to adjust the grammar flexibly by simply editing the BNF notation.

By writing a custom parser generator, it was possible to design a parser specifically for the Typescript programming language. This meant that the compiler could be made fairly robust. For example, the entire compiler tool-chain is written in a type-safe manner, this could not have been achieved without the custom parser generator producing Typescript classes. Implementing a parser generator also added some extra challenge to the project. It also provided an excellent insight to the nature of grammars and the difficulty in specifying them.

10.2 Incomplete Features

It is unfortunate that the full set of S-algol features could not be implemented by this project. This was due to both technical reasons and time constraints. The technical reasons are addressed in the implementation sections.

The time constraints could arguably be because of the scope of the project. Writing a compiler for a full language is by no means a simple task. The line-count command "wc -l ./src/**/**.ts" yields a result of nearly 10,000 lines of code — although around 20% of this is generated by the meta-compiler.

10.3 Tooling

One key limitation of this S-Algol implementation compared with other Javascript compilers that are available is that of tooling integration. To be a competitive compile-to-Javascript language, it is important to have integrations with build tools and IDEs. It would be possible to engineer these for this compiler. However, the core features of the language were a higher priority.

10.4 Comparison between S-algol and modern-day languages

In the context survey, it is identified that S-algol is part of the genus of modern programming languages. It is therefore appropriate to compare it with modern-day languages as a linguist might compare Latin to English. And, just as English and Latin resemble each other, so the basic components of S-algol are easily recognisable in all modern-day imperative languages.

A key difference in S-algol to a language such as Javascript is the use of first-class syntax to represent high level features. For example, image manipulation is built into the language at a syntax level whereas in Javascript, images are simply treated as regular objects. This represents a trend in programming languages as a whole: to move features from the syntax and implement them as libraries. This means that compiler design can be simpler and more portable across architectures. However, it does make it harder to target optimisation since it might be considered inelegant to target specific libraries within a compiler.

In S-algol, there are functions that are "hard-coded" into the grammar which in modern languages might also be considered part of standard libraries. This includes finding the bounds of an array and doing io operations.

The handling of clauses as expressions is not seen explicitly in modern languages. Most languages have some basic syntax to do similar things such as ternary operators. It is likely that this approach has been moved away from since it produces cluttered, confusing code.

Similarly, the lack of explicitly returned values is not very clear for programmers of modern day languages. An expression at the end of a block in S-algol is the returned value of that block. In Javascript for example, expressions must be prefixed by 'return'.

10.5 External Evaluation

During this project, I had three sources of external input. Throughout the implementation, Graham Kirby, my supervisor offered guidance. At the end, Kevin Hammond (original author of Glasgow Haskell Compiler) and Ryan Cavanaugh (Typescript contributor at Microsoft) also contributed some advice.

During implementation stages of the project Graham had suggested that the meta-compiler approach might be solving a bigger problem than was necessary. However, not being able to see a simpler solution to the parser stage, I implemented it nonetheless.

When I talked over the compiler implementation with Kevin, he agreed with the decision to use multi-pass rather than single-pass architecture. However, he did say that the meta-compiler implementation was somewhat over-complicated and that it would have been better to use an external library. He said that it might have been better to focus on completing later stages of the compiler rather than the parser. Although he did qualify this with the suggestion that a custom meta-compiler implementation might be appropriate if very specific

customisations were required. This did advantage my implementation to a certain extent since I was able to generate Typescript classes which would not have been possible if using a generic parser generator.

Ryan said that it was a very interesting project and requested to add it to the Typescript repository of "real-world" project examples. He said that the fact that I using the Typescript language in unusual ways (for example, dynamically generating classes through the meta-compiler) would mean that it would provide an interesting test case. This further reinforced my impression that the best way to finesse a compiler is to work with as many real-world examples as possible.

11 Conclusions

This project has provided a fascinating insight into the nature of compilers and computer languages. It is likely that had I implemented the parser stages in a less robust, quicker way, I might have been able to produce a more complete code generation stage and thereby be able to handle more S-algol inputs. However, the learning experience that has come from implementing the parser in detail has been more educationally rewarding than a finessed compiler implementation would have been; certainly I believe I have learnt more through doing this than I would have through running lots of test cases on the project and making tweaks to support esoteric features.

I am satisfied that I have produced a high-quality software engineering project which is extensible. When feedback has been complete for the project, I will release the source on GitHub and publish web interface to fulfil the requirement that it provide a resource for people who are interested by the language.

References

[1] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java.* Vol. 2nd ed. Cambridge University Press, 2002. ISBN: 9780521820608. URL: http://search.ebscohost.com/login.aspx?direct=true&db=edsebk&AN=112705&site=eds-live.

[2] Apple. *JSContext Class Reference.* https://developer.apple.com/library/ios/documentation/JavaScriptCore/Reference/JSContext_Ref/. accessed 17-2-2016. 2015.

[3] CoffeeScript. *List of languages that compile to JS.* https://github.com/trending/coffeescript. accessed 5-3-2016. 2012.

[4] A. Cole. *An introduction to programming with S-algol.* Cambridge University Press, 1982.

[5] A.J.T. Davie and R. Morrison. *Recursive Descent Compiling.* Ellis Horwood Publishers, 1981.

[6] Edsger W Dijkstra. "Why numbering should start at zero". In: (1982).

[7] GitHub. *Trending in open source*. https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js. accessed 5-3-2016. 2012.

[8] Scott Hanselman. *JavaScript is Assembly Language for the Web: Sematic Markup is Dead! Clean vs. Machine-coded HTML*. goo.gl/KT4Jom. accessed 17-2-2016. 2011.

[9] JSConf. *Introducing TypeScript*. http://2012.jsconf.eu/speaker/2012/10/05/introducing-typescript.html. accessed 17-2-2016. 2012.

[10] Javascript Mess. *Jason Scott*. http://www.archiveteam.org/index.php?title=Javascript_Mess. accessed 30-3-2016. 2015.

[11] Microsoft. *Variable Scope (JavaScript)*. https://msdn.microsoft.com/en-us/library/bzt2dkta(v=vs.94).aspx. accessed 30-3-2016.

[12] Microsoft. *Welcome to TypeScript*. http://www.typescriptlang.org. accessed 17-2-2016. 2012.

[13] R. Morrison. "ON THE DEVELOPMENT OF ALGOL". PhD thesis. University of St Andrews, Dec. 1979.

[14] R. Morrison. *S-algol Reference Manual*. University of St. Andrews, North Haugh, Fife, Scotland. KY16 9SS, Dec. 1979.

[15] Ronald Morrison. "On the development of algol". PhD thesis. Citeseer, 1979.

[16] Mozilla. *Gap between asm.js and native performance gets even narrower with float32 optimizations*. https://hacks.mozilla.org/2013/12/gap-between-asm-js-and-native-performance-gets-even-narrower-with-float32-optimizations/. accessed 17-2-2016. 2013.

[17] Norman Neff. "OO design in compiling an OO language". In: *ACM SIGCSE Bulletin* 31.1 (1999), pp. 326–330.

[18] John C. Reynolds. "ESSENCE OF ALGOL." In: 1981, pp. 345–372. URL: http://search.ebscohost.com/login.aspx?direct=true\&db=edselc\&AN=edselc.2-52.0-0019705872\&site=eds-live.

Appendices

A Testing summary

S-Algol Code	Output After Compilation and Evaluation
`write 2 = 2?`	`true`

`write 2?`	2
`write 1 + 1.1?`	2.1
`write "test"?`	test

`if true`
`then write 1`
`else write 2?`
	1

`if false`
`then write 1`
`else write 2?`
	2

`let a = 4;`
`a := a + 1;`
`write a?`
	5

`let a = 4;`
`repeat a := a + 1 while a < 10;`
`write a?`
	10

`let a = 4;`
`while a < 10 do a := a + 1;`
`write a?`
	10

`let a = 0;`
`repeat {`
`  write 1;`
`  a := a + 1`
`} while a < 2 do write 2?`
	1
	2
	1

`let a = 0;`
`begin`
`  let a = 4;`
`  write a`
`end;`
`write a?`
	4
	0

`let a = 0;`
`{`
`  let a = 4;`
`  write a`
`};`
`write a?`
	4
	0

`let a = @ 0 of int[1,2,3,4];` `write a(3)?`	4

`let a = @ 0 of *int[@ 2 of int`
`  [1,2,3,4]];`
`write a(0, 2)?`
	1

`procedure a(int b -> int); b;` `write(a(4))?`	4

`procedure a(real b -> real); b;` `write(a(4.12))?`	4.12

`let a = @ 0 of int [1,2,3,4];`
`procedure b(*int arr -> int); arr(2);`
`write(b(a))?`
	3

`let a := -1;` `write abs(a)?`	1

`let a = 1;`
`let b = if a > 0 then 3 else 4;`
`write b?`
	3

`let a = {let q = 1; q + 2};` `write a?`	3
`let x = 0; while x < 10 do x:=x+1; write 10?`	10

`procedure fibpair( int n -> *int );`
`if n = 1 then @1 of int[ 1,0 ] else`
`if n = 2 then @1 of int[ 1,1 ] else`
`if n rem 2 = 0 then`
`begin`
`  let fg = fibpair( n div 2 );`
`  let f = fg( 1 );`
`  let g = fg( 2 );`
`  let s = f * f;`
`  let t = g * g;`
`  @1 of int[ s + 2 * f * g,s + t ]`
`end else`
`begin`
`  let fg = fibpair( n - 1 );`
`  @1 of int[ fg( 1 ) + fg( 2 ),fg( 1 ) ]`
`end;`
`procedure fib( int n -> int );`
`if n = 0 then 0 else fibpair( n )( 1 );`
`for i = 0 to 5 do write i,fib( i )?`
	`0 0`
	`1 1`
	`2 1`
	`3 2`
	`4 3`

Table 1: S-Algol compilation test cases

S-Algol Code	Output After Compilation and Evaluation
`wr`	[Parser Error] The input program is not complete.
`write ;?`	`[Parser Error] Line 1\: 'semi_colon' is not a recognisable way of starting a <write_list> production.`

`let a = 1 + true?`	[Error] Could not execute ADD on int and bool.
`a(1)?`	[Error] No function or variable named 'a' found in scope. [Error] No function or variable named 'undefined' found in scope.

`{`
`  let a = 1`
`};`
`write a?`
	[Error] No function or variable named 'a' found in scope.

`let a = @ 0 of int [1, 2, 3];` `write a(1, 2)?`	[Error] The vector 'a' only has 1 dimension(s). You tried to access dimension 0.
`let a = @ 0 of int [1, 2.0, 3];` `write a(1, 2)?`	[Error] Vector err. [Error] The vector 'a' only has 1 dimension(s). You tried to access dimension 0.
`let a = 1; a(2)?`	[Error] The vector 'a' only has 0 dimension(s). You tried to access dimension 0.
`structure test(int a);` `let a = test("test")?`	[Error] Could not apply expression of type string to argument of type int as argument number 1 in the method 'test'.
`structure test(int a);` `let a = test(1, 2)?`	[Error] Dereference error..

`structure test(int a);` `let a = test("wrongargtype")?`	[Error] Could not apply expression of type string to argument of type int as argument number 1 in the method 'test'.

Table 2: S-Algol compilation test cases that correctly notice errors

B User manual

The project dependencies and build are handled by npm. The project is provided with compiled sources and dependencies but to install them from scratch, clean the build using 'rm −rf ./node_modules/ ./bin/ ./typings/' then run 'npm install'.

B.1 Regular Usage

The command line interface is accessible using the utility ./s, found in the root of the project. This accepts S-algol programs from stdin or quoted inline. Listing 93 shows two ways of interfacing with the compiler.

Listing 93: "Command Line Interface"

```
# print compiled code to stdout
./s -cs "write 1?"
echo "write 1?" | ./s -cs

# execute compiled code using node Javascript
echo "let a = if true then 1 else 2; write a?" | ./s -cs
    | node
./s -cs "let a = if true then 1 else 2; write a?" | node
```

B.2 Testing

To test the project run npm run test in the root directory.

B.3 Project Layout

The project sources are laid out in the following directories.

./src/ contains the project source.

./src/metaCompiler contains the source of code which handles compilation and manipulation of the S-algol grammar.

./src/web contains the source code for the web interface.

./src/sAlgolCompiler contains the source of the S-algol compiler.

./src/sAlgolCompiler/generatedFiles contains the files generated by the meta-compiler for use by the main compiler.

./src/sAlgolCompiler/generatedFileHelpers contains non-generated files that augment the generated files.

./src/sAlgolCompiler/sAlgolSources contains implementations of the S-algol standard library and some procedure declarations.

./src/sAlgolCompiler/visitors contains compilation phases that are implemented using the visitor pattern such as type checking.

./src/test contains mocha test files.