S-algol in the Browser

Chance Carey

Supervised by Professor Alan Dearle

University of St Andrews

27th April 2020

Abstract

This report documents a compiler implementation and browser runtime for the S-algol programming language. The compiler runs fully in the browser, accepts S-algol input code, and generates dependency-free Javascript code. The browser aspect facilitates the editing and compilation of S-algol programs, and offers a runtime for the compiled Javascript code that accepts user input and displays output text and vector graphics.

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 14,287 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 report to be made available on the Web, for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis. I retain the copyright in this work, and ownership of any resulting intellectual property.

Table of Contents

1 Introduction

This project focuses on constructing a compiler implementation for the S-algol programming language, with the compiler targeting pure Javascript. An online environment is provided in order to edit and compile S-algol code, as well as to run the generated code with support for IO and the rendering of vector graphics.

The vast majority of the requirements that were set out have been achieved. Due to limitations of the browser environment in which the generated code is run, not all requirements were possible to implement. Despite these restrictions, the following requirements were fully implemented:

- Converting the grammar from a left recursive context sensitive grammar to a LL(1) context free grammar
- Creating a parser generator to generate a parser that performs lexical analysis and parsing simultaneously, and that is capable of doing so for the entirety of the language grammar
- Creating a semantic analysis engine for the complete type and scope analysis of all S-algol constructs when provided an Abstract Syntax Tree (AST) produced by the parser
- Creating a code generator that takes the AST and generates pure Javascript code
- Building a browser Integrated Development Environment (IDE) featuring an editor with syntax highlighting and keyword matching that is capable of running the compiler and executing the compiled output code, utilizing blocking input, and rendering text and vector graphic output
- Implementing a subset of the standard library and extending it to provide functions useful within the browser environment

Certain features were not fully implemented due to time constraints and limitations imposed by the browser:

- Raster graphics
- Debugging S-algol code at runtime

This report documents the design and implementation of the aforementioned features, as well as demonstrates their usage through running existing S-algol programs written in 1987 that generate complex vector graphics.

2 Context Survey

2.1 Historical Overview of S-algol

S-algol is a programming language that was designed and first implemented by Ron Morrison in his 1979 PhD thesis **On the Development of Algol**[1]. Morrison discusses the idea that most languages have large amounts of similarities and that it is by their syntax and data structures that they differ most significantly. The programming language S-algol was proposed as a member of the `ALGOL` series of languages, with a focus on design in the direction of simplicity and expressivity. Morrison's initial implementation of S-algol compiles to a bytestream labelled `S-code`, which is later executed on a purpose built stack machine.

Morrison's thesis discusses a traditional structure of building a compiler, through the use of three stages:

- A lexical analysis stage for converting some input into a sequence of basic symbols, or lexemes
- A syntax analysis stage for consuming these symbols and producing a syntax tree
- A code generation stage for consuming this syntax tree and producing machine code

In contrast to this traditional structure, Morrison develops an approach through the use of a recursive descent compiler. The term "recursive descent" is often used in the context of writing a parser (syntax analysis) through the use of parsing functions recursively calling other parsing functions on subsets of the input stream in order to construct a syntax tree. A recursive descent *compiler* operates in a vaguely similar way: procedures are recursively called to consume the sequence of lexemes given as input. It differs, however, in that instead of producing a syntax tree, the code within these functions would couple both parsing *and* code generation together, in an effort to completely avoid having to create and later consume a syntax tree, reducing the complexity of the compiler. Morrison describes the concept of adding "layers" to the recursive descent functions as opposed to having separate passes over the code. This approach is further documented in a book later authored by Morrison[2].

Morrison further describes the type system of S-algol, which is inductive and places a focus on having a complete rule set for the types to avoid any gaps that would later necessitate adding support for "special cases".

S-algol found usage largely as a teaching language at the University of St Andrews, as well as at a local school in St Andrews, Madras College. The language has, unfortunately, not made its way onto computer systems of the modern era. It is the hope of this project that a modern implementation targeting Javascript will serve to preserve S-algol.

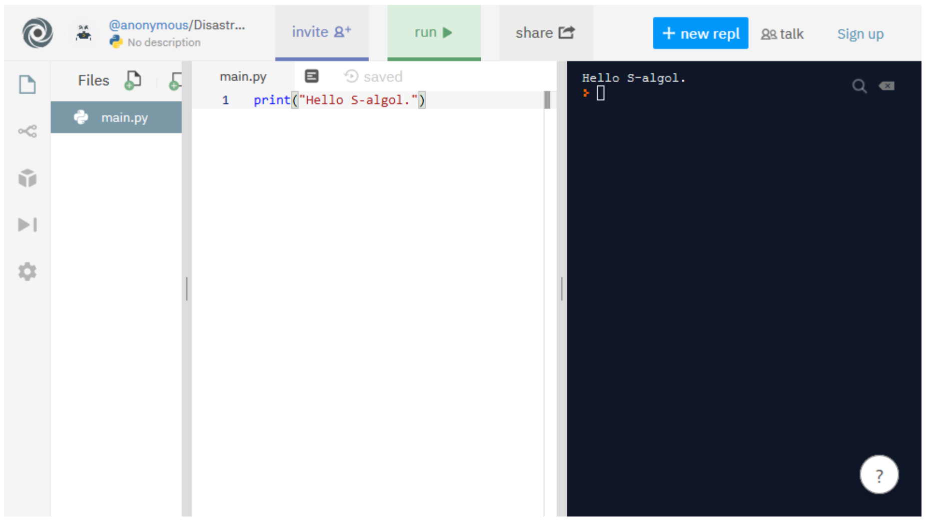
2.2 The Utility of Javascript

Javascript is one of the most ubiquitous languages in the world, owing to its status as the primary language of the client-side web[3]. Targeting such a widely available technology seems appropriate, then, when constructing a tool that aims to preserve an older technology. Javascript is also quick to execute, which mitigates a potential concern when working with a codebase of something likely to extend into the many thousands of lines, which tools as complicated as compilers tend to become. As Javascript has been, for a substantial period of time (up until the introduction of WebASM), the only way to script web applications client-side, it has also found itself quite often used as a compilation target for those wishing to use other programming languages for client-side web applications. An example of one such language is Typescript[4], an effort by Microsoft to develop a fully typed superset of Javascript for an improved developer experience, which compiles down to pure Javascript code. Languages of other paradigms such as the functional programming languages OCaml[5] and Elm[6] also target Javascript. This common use of Javascript as a compiler target lends justification to the decision to target it as the compiler backend of this project.

The barrier of entry to being able to use S-algol changes, then, from "anyone that runs a specific linux or windows kernel with compile tools on an x86/ARM cpu" to "anyone with a web browser on any platform".

2.3 Similar Languages and Environments

This project focuses on developing both a compiler for S-algol as well as a browser based Integrated Development Environment. There exist similar environments built for editing, compiling, and running various programming languages within the browser. One such environment is that of `repl.it`[7], an online website that permits the editing, compilation, and running of various programming languages all within a single browser tab.



```
print("Hello S-algol.")
```

```
Hello S-algol.
>
```

Figure 1: Screenshot of the `repl.it` interface

Figure 1 demonstrates the main elements that we aim to construct in this project - an editor, options to compile and run the program, and a way to view the output of that running program. One area where S-algol will differ to existing solutions such as `repl.it` is in the level of integration offered in the browser; most existing browser IDEs permit only text input and output. An element of this project is that it will support text input and output, however it will *also* permit the rendering of programmatically created graphics.

As mentioned above, S-algol found usage within the University of St Andrews and a local college as being a language often used for teaching. There exist other languages such as Scratch that serve this purpose.

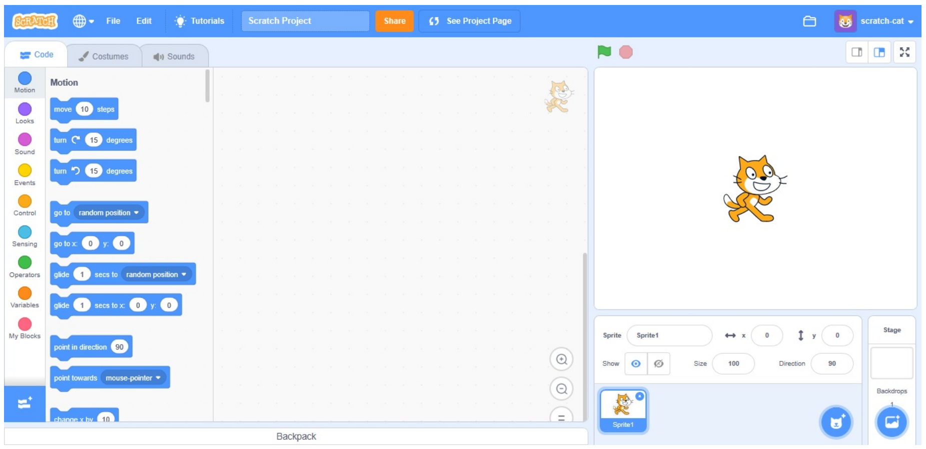


Figure 2: Screenshot of the Scratch programming environment

As shown in Figure 2, Scratch is a visual programming language intended for a younger audience that uses the composition of visual “blocks” to construct the logic of an application in order to display graphics. While S-algol is clearly not the first language to find usage in teaching, S-algol differs from the class of languages including Scratch in that S-algol is a traditional language that is edited textually. It does, however, retain the easy integration and usage of graphics which provides some appeal as a language used for teaching.

3 Requirements Specification

The S-algol language is large and complicated as it has many primitives that are relatively uncommon in programming languages, such as having vector and raster graphic functions built into the language itself. Compiler development is itself a wide field, with many possible approaches to each task in the compiler pipeline. This is to say: defining a set list of requirements is key to placing a bound on this unbounded problem area. These requirements are split into sets of primary and secondary concerns; this is done as the implementation of *all* primary requirements is necessary for the implementation of *any* of the secondary requirements.

The set of primary requirements begins with that of the manipulation of the S-algol grammar. The grammar will be copied from Morrison's original PhD thesis where it is specified, and transformed in multiple ways. The first transformation will be the conversion of the grammar, from left recursive and context sensitive, to LL(1) and context free, in order to aid with writing a parser generator. As the structure of the compiler will be a multi pass recursive descent design, having the grammar be context free is required for successful implementation, as knowing how to parse according to a context sensitive grammar by definition requires contextual knowledge, which will not be present in a multi pass compiler.

A lexical analyzer and recursive descent parser will be created. Due to the complexity of the grammar, it is likely that the use of a parser generator will make the development of the parser significantly less strenuous and easier to modify. Similarly, integration of the lexical analyzer into the parser will reduce code complexity by reducing the overall amount of code required.

S-algol is a strongly, mostly statically typed language. Semantic analysis is therefore a key element of the compiler pipeline, and the implementation of it will also be required to drive certain decisions in code generation as there will be some loss of information when converting the grammar from a context sensitive to context free form.

The final element of the compilation pipeline is, of course, code generation. Similar to the semantic analysis stage, an approach likely to be successful for this step will be to walk the Abstract Syntax Tree generated by the parser, and return Javascript code for each node.

The last requirement that is necessary to permit basic functionality will be the construction of a browser REPL-like environment to facilitate editing of S-algol programs, compilation, and running the generated programs.

After completing the set of primary requirements, a set of secondary requirements will be implemented that will serve to augment the completeness of this implementation of S-algol. The initial secondary requirement is that there be sufficient implementation of the standard library to be able to run a small set of existing S-algol programs. This is necessary in order to provide evidence of compatibility with programs written targeting the original S-algol compiler built in 1979. Following this, support will be added for the manipulation of certain graphics (vector and/or raster) using the S-algol language primitives, and being able to render these outputs to the browser environment. Finally, it would be useful to implement some sort of introspective behavior into the runtime to permit the debugging of S-algol code through the use of breakpoints or similar tools.

These primary requirements serve, then, as steps towards achieving a minimal working state of an S-algol implementation in the browser, and the secondary requirements serve to augment the experience and provide further completeness of the implementation.

4 Software Engineering Process

4.1 Development Tools

When developing any moderately complicated software project, the choice of tooling used is often key to how little friction there is during development.

The key to any form of ambitious development is being able to work on and break different parts of the program independently before merging together changes into a working copy, which is accomplished through the use of version control system. For this, `git`, a modern VCS, was chosen. All project files were checked in to the git repository, including build artifacts. Build artifacts were included as it permits quickly going back and running previous versions of the project without having to recompile each time, a useful property when running for example `git bisect` to iterate through the code history to locate where a bug was introduced. Additionally, all *reference* documentation (specifications and source codes of previous S-algol compilers) were checked into the repository as a form of backup and safe keeping as the documents are not easily available. Various git features were used during development such as feature branches, where different branches would track efforts on various parts of the program, a feature used when concurrently working on structure support at the same time as vector graphic support, before merging the feature branches into the master branch when they were completed. This separation of working code branches ensures that the master branch is always a working copy, and that multiple features that may cause errors during their development can be worked on concurrently.

As this project revolves around the browser, a large portion of the tooling used relates to the Javascript ecosystem. A view library called React[8] was used to simplify the development and operation of the UI through its declarative structure. Using React also permits the quick integration of other components such as Monaco[9], an editor that enables the use of custom syntax highlighting. To build the React app, the build tool `create-react-app`[10] was used to abstract away the complex build components and configuration. Finally, the helper utility `JS-Beautify`[11] was included and added as a post-processing step of the compiler pipeline in order to make the compiler output more readable, as the code generator cares little for concepts such as indentation and line breaks.

Typescript was used for the entirety of the development of this project. Typescript is a superset of Javascript that permits the use of strong static typing. This allows the Typescript compiler to catch entire classes of error at compile time instead of occurring as bugs during runtime (type errors, possible null errors, syntax errors, and so on). This further speeds up the development of various systems through the autocompletion and type hinting of fields enabled through the static nature of the language. This was of key utility as the parser creates a fully typed AST; having completion suggestion for the fields saved uncertainty when working with the hundreds of types of AST node classes.

The utility provided by these tools and libraries cannot be overstated; while modern development pipelines are perhaps overly complex, the quality of life of tools such as React and Typescript reduce development time and frustration by an order of magnitude.

4.2 Staged Development

This project followed the iterative and incremental[12] model of development. As described in the requirements section, the development of features followed a naturally incremental curve as implementing successive requirements often required all requirements before that to be implemented; for example, one cannot implement a semantic analyzer without having a parser produce a syntax tree for which it should analyze. The main stages of the development focused, then, on the three separate elements of the compiler: parsing, semantic analysis, and code generation. This multi stage approach required working fully through each stage before moving on to the next. Further, these components, while reliant on the outputs of the previous components, are themselves quite independent, and so they were neatly split into the folders of `parser`, `analyzer`, and `codegen`. These split folders helped to keep the codebase clean and maintained separation of concerns, as exporting between components is a deliberate process. Additionally, having each element present a static public interface permits the theoretical swap-out of any component - for example in the future the `codegen` component could perhaps be rewritten to target another language such as `WebASM` without having to modify the parser at all.

5 Design

5.1 Application Structure

The application is structured with two main elements. The first element is that of the compiler, which accepts an input string, and returns either an error or an output string of Javascript code. The second element is the browser environment in which the compiler is run, which also contains the editor as well as the runtime which permits IO to occur and graphics to be displayed.

The editor component displayed contains the S-algol code that is to be run. The editor enables syntax highlighting of the S-algol code according to the language grammar, and also permits keyword autocompletion as a quality of life feature.

The compiler is a multi-pass design, split into three distinct stages. These stages are parsing, semantic analysis, and code generation. As the grammar for S-algol is complicated, with more than 80 productions and over 150 branches, a parser generator is employed to create the large amount of code required for a fully typed parser. S-algol has some language features such as the optional use of whitespace instead of semicolons, with whitespace otherwise being discarded, that prevents the use of a pre-existing parser generator (and few target Typescript in any case). Thus, a parser generator was devised that takes the S-algol grammar and produces a parser that, given input S-algol code, produces an Abstract Syntax Tree (AST).

The semantic analysis stage is responsible for type and scope checking. Each unique type in the S-algol program (variable type, structure, procedure, etc) is recorded and scoped correctly, and all operations and procedure calls, structure creations, and indexing, are type checked during this stage. A minimal amount of type information is written to the AST during this stage.

The final step in the compiler pipeline is code generation. The AST is walked recursively to produce pure Javascript code that can then be run in the browser.

5.2 Approaches to Compiler Construction

Compiler construction is a wide field, and there are countless ways to write a compiler for the same input language targeting the same output language. The original S-algol implementation is a single-pass compiler utilizing a recursive descent approach; this approach involves the compiler parsing grammar and concurrently performing semantic analysis and code generation.

The benefits of a single-pass compiler are, as described in Morrison's original thesis, that the tight coupling of the various responsibilities of a compiler allows for reduced amounts of code and the reduced complexity of such code. Additionally, single-pass compilers discard the entire concept of producing a parse tree, as the input that is parsed is then immediately converted into code or stored for later type analysis.

The drawback of this approach, however, is that the tight coupling causes the modification of any of these components to be something of a difficult task. The alternative to this approach is through the use of a multi pass, or multi stage, compiler. This separates the frontend stages, responsible for parsing and semantic analysis, and the backend stages, responsible for code generation. The interaction between these components is then strictly defined, and this process permits far easier modification or even replacement of any of the components in question. As this project is focused on providing a modern implementation of the S-algol language, opting for the multi-pass compiler will enable easier future preservation if, for example, the code generator had to be changed due to a backwards incompatible change to the Javascript specification. Indeed, the entire code generation module could simply be replaced with one that targeted a different language such as x86 assembly for native execution, or WebASM if a more efficient browser-based approach was to be required.

5.3 Context Sensitivity in Grammars

The S-algol language defined in Morrison's thesis is described using a context sensitive grammar. Context sensitive grammars are those that have ambiguities in the grammar that are resolved during parsing through some knowledge of the input that has been previously parsed. In the case of S-algol, context sensitivity occurs through knowing how to parse expressions that are suffixed with brackets and a clause list - such a construction could either be a structure creation, procedure call, or structure or vector indexing expression. For example, `a(b)` could be creating the structure `a` with a field value `b`, calling the procedure `a` with the parameter `b`, or fetching the field `b` of a variable `a` that is a structure. In Morrison's single-pass recursive descent compiler, handling such context sensitivity is possible as the semantic analysis stage occurs in lockstep with the parser, and so type information is recorded at parse time, permitting future parsing of context sensitive constructions.

For a multi-pass compiler where the parser performs no semantic introspection into what it is parsing, context sensitive parsing becomes impossible. The remedy to this is to convert the productions of the grammar that are context *sensitive* into context *free* forms - this is done, mainly, through the reduction of ambiguous clauses. The drawback to this is that the AST then does not distinguish between semantically different types of productions, as they are syntactically identical and are parsed as such by the context free parser. This becomes a concern during code generation, as the code generator has to produce different code depending on the semantics of the S-algol input and not simply its syntax. By annotating the AST with relevant type information during semantic analysis, this problem is resolved.

5.4 Parser Variations

Given an LL(1) context free grammar, there are multiple possible ways to parse any input conforming to that grammar. One approach is to use a parser combinator, in which parsing occurs through a “trial and error” process by attempting each possible way of parsing a set of input lexemes. The strength of this approach is that it is simple to implement as one doesn’t have to consider and switch based on lookahead tokens. The drawback, however, is that error reporting is *severely* impeded as the trial and error nature of a combinator style parser means that discovering exactly where an error occurred is not possible, and as a result all errors appear to occur at the root node and the very first token of input.

A popular parser design for LL(1) CFGs is that of recursive descent. Recursive descent parsers consist of parsing functions that consume terminal tokens (for example a string), and that call other parsing functions for non terminal tokens. This permits exact error reporting as errors occur wherever the consumption of a terminal token fails, however it is more complicated than parser combinators as lookaheads have to be used to determine how to parse sequential tokens. This becomes an issue for grammars as large as S-algol, as some productions have upwards of 70 (in the case of the `clause` production) lookahead tokens. In addition, simply creating the type structure for a grammar with approaching 100 productions would be a highly tedious and error-prone task.

An alternative approach to constructing a recursive descent parser by hand is to make use of a parser generator. A parser generator consumes a grammar (in, for example, EBNF) and outputs a complete parser for that grammar. This was the approach taken for the compiler implementation in this project through the use of an S-algol specific purpose built parser generator. A slightly modified version of the S-algol grammar is created, and the parser generator consumes this, and produces a fully typed Typescript recursive descent parser. The generated parser is then used to consume a string of S-algol code and produce a fully typed Abstract Syntax Tree for later use.

5.5 Semantic Analysis

Semantic analysis is the process of walking an Abstract Syntax Tree and performing scope and type analysis. Scope analysis is ensuring that no variable is incorrectly referenced from another scope, and type analysis ensures that structures are created with correct field types, procedures are called with correct types and return the correct type, variables are not reassigned to different types, and so on for all type rules defined in the language specification.

In this project, scope checking and type checking are performed concurrently. Scope checking can be seen as a function of type checking; during type checking the recorded types are scoped correctly as S-algol permits type shadowing. As a result of this, scope checking naturally occurs when looking up types in each scope in the hierarchy.

5.6 Code Generation

Code generation is the process of consuming the AST and producing Javascript for each node in the AST. This is performed through functions recursively walking the AST and returning Javascript.

If this compiler were targeting a strongly typed language such as C, code generation would be far more involved as the AST would need to be fully type annotated during semantic analysis as when generating C code, types would have to be declared for all variables, structures, procedures, etc. Additionally, elements like memory management would be significantly more complicated and it is likely that some form of garbage collector (GC) would need to be added to the program to prevent memory leaks. Fortunately, targeting Javascript means that these concerns are not realized; Javascript is a dynamically typed language, and memory management is performed automatically. Javascript is also a highly flexible language which permits the vast majority of S-algol constructs to be expressed directly in Javascript without having to perform any form of tree rewriting or employ the use of further intermediate representations to modify the structure of the program.

6 Implementation

6.1 Browser Based Integrated Developer Environment

With an overall goal of being able to run S-algol in the browser, an important consideration was what being "able to run" actually *looks like*. Many popular compilers such as GCC and Clang are command line programs; the developer uses an editor of their choice to write the source file, and then a terminal application to actually run the compiler command. To build an integrated developer environment in the browser, many of those elements have to be recreated - for example the web application would have to include an editor, some way to indicate that the code in that editor should be compiled, and some way to run the compiled program and view the output.

React was used to build the user interface. While the interface is not overly complicated, using a view library like React is still preferred over using just DOM manipulation as it, for the most part, removes the entire class of errors that comes with performing manual DOM manipulation. Instead, the view is defined and provided with data, and the DOM is kept up to date as the data is mutated. Using a frontend library like React also allows for the easy integration of other libraries, for example text editors or canvas graphics displays. React applications with complicated state often employ utilities like a router to manage navigation (such as React Router[13]), as well as some form of state management (such as Redux[14]). As the interface of this project is straightforward and not overly complex, such utilities were not required.

A rough mockup was then created to set out the layout of how the browser IDE would look.

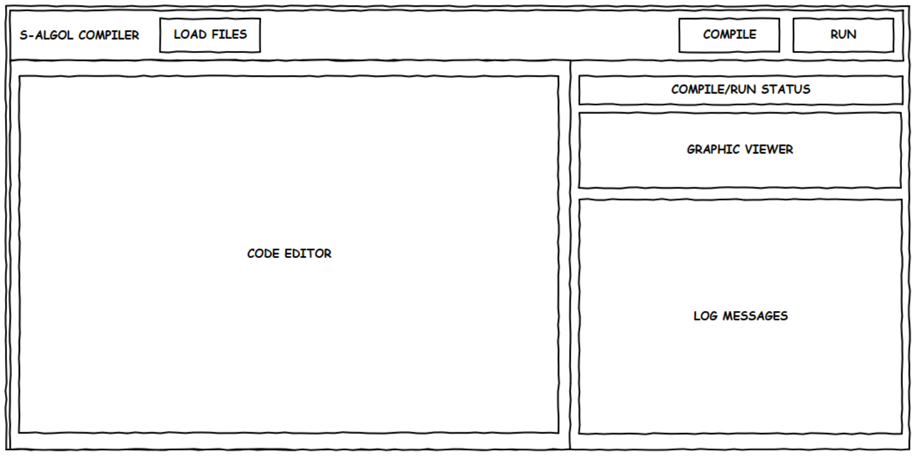
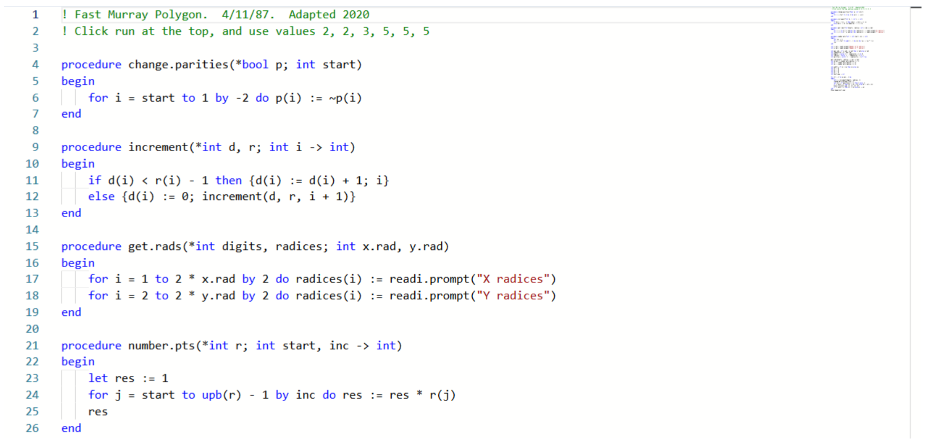


Figure 3: Mockup of the browser application layout

As shown in Figure 3, the layout is split into roughly three main sections. A menu bar rests at the top and permits actions such as loading from a few preset files, and buttons to begin the compilation and execution processes. The left section is where the code editor is located, and the right section is where all of the program output should be: compilation progress indicators (including error reporting), log output, and graphics output. As the layout is fairly straightforward, a simple CSS Grid layout for each of the three sections was sufficient.

An existing library was used to implement the code editor. Microsoft maintains the open source IDE Visual Studio Code[15], and the editor component of this is available as a standalone library that can be included in other React based applications, called Monaco[9]. While a simple HTML `textarea` with a monospace font would suffice for a barebones code editor, integrating an existing solution such as this permits some of the niceties that developers expect in modern applications, such as syntax highlighting and keyword suggestion. Monaco comes with syntax highlighting presets for popular languages preloaded, however S-algol is not included in this list. To address this, a custom highlighting preset was built using extracted keywords, types, and operators from the S-algol grammar, as well as additional rules such as for comment highlighting. This highlighting preset is registered with Monaco at runtime, and permits full syntax highlighting. A rendered example is included in Figure 4.

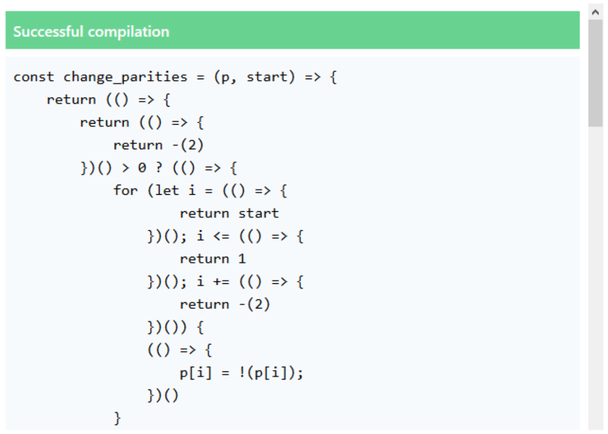


```
! Fast Murray Polygon.  4/11/87.  Adapted 2020
! Click run at the top, and use values 2, 2, 3, 5, 5, 5

procedure change.parities(*bool p; int start)
begin
    for i = start to 1 by -2 do p(i) := ~p(i)
end

procedure increment(*int d, r; int i -> int)
begin
    if d(i) < r(i) - 1 then {d(i) := d(i) + 1; i}
    else {d(i) := 0; increment(d, r, i + 1)}
end

procedure get.rads(*int digits, radices; int x.rad, y.rad)
begin
    for i = 1 to 2 * x.rad by 2 do radices(i) := readi.prompt("X radices")
    for i = 2 to 2 * y.rad by 2 do radices(i) := readi.prompt("Y radices")
end

procedure number.pts(*int r; int start, inc -> int)
begin
    let res := 1
    for j = start to upb(r) - 1 by inc do res := res * r(j)
    res
end
```

Figure 4: The Monaco editor displaying an S-algol program

Along the top of the application is a "menu bar" of sorts. It contains an indicator that the application is an S-algol compiler, as well as buttons that preload certain S-algol programs, and buttons to compile and run the programs. When a button is clicked to load an S-algol program, the program is loaded into the editor, and then also immediately compiled. When a compilation is started (automatically when loading a program, or by clicking the compile button), the program takes the current text in the editor, runs it through the parser to generate the Abstract

Syntax Tree, performs semantic analysis, and then generates and displays the compiled Javascript in the code view on the right panel. As the code generator creates code with no regards to formatting (and so it is often just one long line), a code “beautifier” is bundled with the application (JS-Beautify[11]) and the produced Javascript is run through this prior to being displayed, so that it is more legible. This run panel (including a message that the compilation was successful) is included in Figure 5.

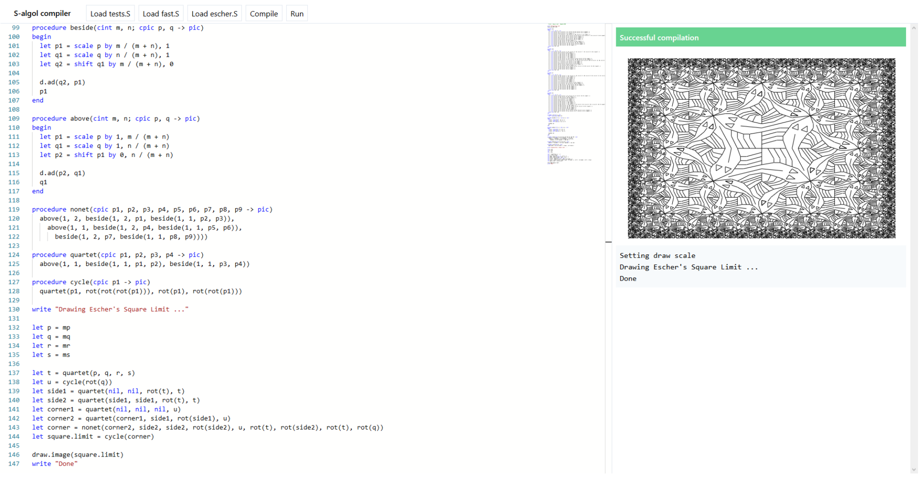


Successful compilation

```
const change_parities = (p, start) => {
    return (() => {
        return (() => {
            return -(2)
        })() > 0 ? (() => {
            for (let i = (() => {
                    return start
                })(); i <= (() => {
                    return 1
                })(); i += (() => {
                    return -(2)
                })()) {
                (() => {
                    p[i] = !(p[i]);
                })()
            }
```

Figure 5: S-algol generated into Javascript and displayed

When the user clicks the run button, much of the same process is repeated up to generating Javascript. Once the Javascript is generated, the application calls `eval` on the code to execute it in the same context as the application. Unfortunately, `eval` operates as a black box - code running within it cannot be interacted with - and while it works well for running the generated S-algol program, allowing it to interact with the main React application is difficult as React has a tightly defined lifecycle which makes interoperation with other Javascript code less than straightforward. This is important for things like IO, where the running S-algol code needs to interact with the React application to do things such as display output or display graphics. To work around this, the React application assigns various methods and parameters to the globally shared `window` object. One of the main properties assigned is that of an array being assigned to `window.olog`. This array is where the output of the S-algol program should be written, as when it is updated, React updates the output log segment of the application. Similarly, `window.setImg` is assigned to a method within the React component that accepts an “image” object (the exact structure of which is discussed in further detail later in this report). When this method is called, React updates the properties of the graphics display Canvas component, causing it to re-render the image. Figure 6 shows an example of the output of an S-algol program writing to the log as well as creating an image - that of the Escher Square Limit, included in the `escher.S` file.



```
procedure beside(cint m, n; cpic p, q -> pic)
begin
  let p1 = scale p by m / (m + n), 1
  let q1 = scale q by n / (m + n), 1
  let q2 = shift q1 by m / (m + n), 0

  d.ad(q2, p1)
  p1
end

procedure above(cint m, n; cpic p, q -> pic)
begin
  let p1 = scale p by 1, m / (m + n)
  let q1 = scale q by 1, n / (m + n)
  let p2 = shift p1 by 0, n / (m + n)

  d.ad(p2, q1)
  q1
end

procedure nonet(cpic p1, p2, p3, p4, p5, p6, p7, p8, p9 -> pic)
  above(1, 2, beside(1, 2, p1, beside(1, 1, p2, p3)),
    above(1, 1, beside(1, 2, p4, beside(1, 1, p5, p6)),
      beside(1, 2, p7, beside(1, 1, p8, p9))))

procedure quartet(cpic p1, p2, p3, p4 -> pic)
  above(1, 1, beside(1, 1, p1, p2), beside(1, 1, p3, p4))

procedure cycle(cpic p1 -> pic)
  quartet(p1, rot(rot(rot(p1))), rot(p1), rot(rot(p1)))

write "Drawing Escher's Square Limit ..."

let p = mp
let q = mq
let r = mr
let s = ms

let t = quartet(p, q, r, s)
let u = cycle(rot(q))
let side1 = quartet(nil, nil, rot(t), t)
let side2 = quartet(side1, side1, rot(t), t)
let corner1 = quartet(nil, nil, nil, u)
let corner2 = quartet(corner1, side1, rot(side1), u)
let corner = nonet(corner2, side2, side2, rot(side2), u, rot(t), rot(side2), rot(t), rot(q))
let square.limit = cycle(corner)

draw.image(square.limit)
write "Done"
```

```
Setting draw scale
Drawing Escher's Square Limit ...
Done
```

Figure 6: Rendered graphic of Escher's Square Limit

This environment permits an effective modern development flow of S-algol programs through offering a modern editor replete with syntax highlighting and keyword suggestion, immediate compilation results (with error reporting), and quick image and text output.

6.2 Grammar Refactoring

A significant portion of the time spent implementing this project was focused around refactoring the grammar. As mentioned in the design, the goal was to convert the grammar into an LL(1) form for easier parsing due to only having to do single lexeme lookaheads and having no ambiguity. This process began with copying the original grammar from the S-algol thesis document and reformatting it to use EBNF notation.

During this process it was noted that there were some minor differences between the official grammar and the grammar that existing S-algol code seems to use - specifically, the S-algol compiler (`comp.S`, that is written in S-algol) uses `forward` constructs and sometimes *does not* specify the procedure type, whereas this is required in the grammar. Thus, changes were made to the grammar so that it matched all original S-algol files (it was primarily tested against the compiler file).

The main alterations that had to be made to the grammar were those that would convert it to that of a context free LL(1) grammar. The original grammar was context sensitive, left recursive, and certain productions required upwards of 3 lexeme lookaheads to parse correctly. The primary reason for converting the grammar to LL(1) was to simplify the parser that would be written against the grammar; LL(1) recursive descent parsers are logically simple, which is an important factor when the intention is to write a parser generator for that grammar. Converting a context sensitive grammar to a context free grammar is usually not possible without some loss of information in the parse tree, and naturally causes further complexity in the later stages of the compiler. Unfortunately, context sensitive grammars are often not feasible to implement for multi stage compilers as the parser stage does not by itself record the necessary information required to implement the context sensitive logic. An example of such context sensitive logic (as well as of left recursion) related to the expression parsing productions is as follows:

```
exp6 = ...
| expression "(" clause | clause ")" // Depth selection
| expression "(" clause_list ")"     // Structure creation
| expression "(" clause_list ")"     // Procedure calls
```

As the `expression` production can also directly lead to `exp6` through left derivation, we immediately see there is possible left recursion. As the second element of each branch is identical, we also have ambiguity. The second and third branches are also identical - the original compiler is able to know which branch to parse due to the context of the expression, knowing if the expression resolves to a structure or procedure. The grammar was refactored to reduce left recursion, eliminate identical productions in order to become context free, and common terms were factored in order for the grammar to become LL(1). The previous example of an `exp6` production thus becomes:

```
exp5 = exp6 ["(" clause [clause_follow] ")"]
clause_follow = "|" clause | ("," clause)+
exp6 = ...
```

After the grammar had been converted to an LL(1) context free format written in EBNF, some additional changes were still in order. When manually writing a parser for a production like `"to" clause "by" clause`, a human would know to store the different clauses in separate fields, likely giving them different labels. For example, an AST node for that production may be a class with fields `clauseTo` and `clauseBy` for the two non terminals of the production (the clauses). The grammar itself is not labelled with `clauseTo` and `clauseBy` - it is simply a manual operation to choose names when writing the parser. As the parser is created automatically by a parser generator, such manual operations (naming fields in the AST nodes to correspond to the non terminals) are not possible. Instead, the parser generator needs to know, given *just* the grammar, how to name the fields. To permit this type of behavior in a parser generator, the grammar was extended from EBNF into a more verbose form that permitted the naming of elements of a production; the previous example becomes instead `"to" clause:to "by" clause:by`, with the two clauses being stored as `to` and `by`. For similar reasons, any production with grouped elements such as `exp4 [add_op exp4]*` had the groups factored out into separate productions so as to simplify the generated data structures.

With these changes made, the grammar had been modified from a left recursive, ambiguous, far lookahead, context sensitive form, into an unambiguous, LL(1), context free grammar with a simplified syntax that would permit straightforward parser generation. The grammar increased in number of productions from ~88 to ~98, but actually decreased in complexity due to the reduced usage of complex EBNF functions like nested groups.

6.3 Parser Generators

Parser generators are an established field of software, with Yacc[16] and ANTLR[17] being among the most well known. There are, however, very few parser generators that are capable of targeting (producing a parser in the language) Typescript. Of those few that do target Typescript, the only ones that are publicly available are all of the PEG variety, which are, as commented on in the design section of this report, unsuitable. ANTLR is one of many parser generators that are capable of targeting Javascript, however this was still insufficient as the requisite typing information would be lost when converting the parser generator to Typescript. Indeed, even if this project were written in pure Javascript, using an existing parser generator such as ANTLR may be less than suitable as using ANTLR requires bundling both the parser generator as well as a large amount of code in the form of the parser runtime. Existing parser generators would also be hard to extend for handling some of the more unusual syntax of S-algol such as the optional line delimiting semicolons.

The remaining avenue was, then, to write a custom parser generator. This would permit consuming the custom grammar syntax and producing a parser that was capable of generating fully typed Typescript, a key concern when dealing with a syntax with as many complex productions as S-algol contains.

The parser generator begins by parsing the S-algol grammar. This is performed using a very straightforward hand written recursive descent parser, as the syntax of the grammar was designed specifically to be bare bones and easy to parse. A small amount of semantic analysis is performed during the parsing of the grammar to ensure that the grammar is unambiguous, LL(1), and that all non terminals referenced in a production do exist. Similar to how the parsers generated by the parser generator function, the parsing of this is performed by creating lexemes and consuming them in the same step through the use of regular expressions for each type of lexeme. When creating the AST of the grammar, tokens (the terminals and non terminals of a productions) are checked to see if they are suffixed with `:` which indicate a renaming operation should be performed for that token, and is used during parsing to ensure there are no collisions of non terminals in a production (such a case would lead to undefined behavior in the parser generator).

After parsing the grammar, the parser generator then proceeds to a “lookahead fill” step. In order to create a parser for an LL(1) grammar, it needs to know, at every step in the parsing process, which branch it should take given the next lexeme. To do this, it must know the possible lexemes that could be accepted by each element (terminal or non terminal) in the production. This process is performed by recursively traversing each production and each element a single time until the base case of each production is found (the set of all terminals that could be accepted by the production). Minor error checking is performed here, for example ensuring that there are no duplicates in the set, as this would indicate ambiguity in the grammar. Complex cases are also dealt with, for example in a production such as:

```
expr = "a" "b"? "c"
```

The parser must know, at the point just after accepting the `"a"` terminal, that both terminals `"b"` *and* `"c"` could be accepted, as `"b"` is an optional term. After completing this step, the parser generator has sufficient knowledge to build a complete LL(1) parser.

The parser generator then begins the process of building the final parser. The generation process begins by writing out a basic prelude featuring the base class types of the S-algol grammar. These base class types are `Type`, `String`, `Id`, and `Number`. While `Type` may seem like it has the same purpose as `Id`, it differs through having a complex regex that matches permitted types only:

```
/[*c]*(?:int|real|bool|string|pixel|pic|pntr|file|#pixel|#cpixel)/
```

By contrast, `Id` accepts any word characters optionally containing periods, a `String` accepts any double quote contained series of characters (using single quotes to escape double quotes), and a `Number` accepts any integer or float with an optional exponent. Regular expressions are also stored for matching against whitespace, symbols, comments, and semicolons. As mentioned in the design section of this report, an interesting part of the S-algol grammar is that, similar to Javascript, the use of semicolons to indicate line breaks is *permitted* but not *required*. This optional case effects every element in the grammar that would require a semicolon, for example:

```
proc_decl = "procedure" identifier proc_decl_type? ";" clause
```

In this example, when the parser should expect a semicolon, it should also accept any sequence of newline tokens. This complicates parsing as it means that whitespace can not be automatically stripped when parsing a token, as the syntax is *sometimes* whitespace sensitive. This is not easily expressed in the grammar, as an underlying assumption of most EBNF style grammars is that whitespace is automatically discarded, and so it must be handled by the parser as a special case.

The key reason for creating a parser generator that generates Typescript code is for the ability to maintain full type information of each parsed element. In order to do this, the parser generates a unique class for *each* production branch, named after the production and the index of that branch of the production. A type alias is then created for the production itself, aliasing a union type of all of the branch classes. For example, for the production:

```
bounds_op = "upb" | "lwb"
```

The generated classes and types for this would be as follows:

```
class BoundsOp0 {...}
class BoundsOp1 {...}
type BoundsOp = BoundsOp0 | BoundsOp1
```

From this we also notice that the case of the productions have been changed from snake_case to PascalCase. This is performed through simply uppercasing the first letter of the variable as well as any letter following an underscore, and then removing all underscores. While not required, this change is performed in order to adhere to Typescript/Javascript guidelines of using PascalCase for exported members.

Each class contains a constructor that creates public class variables containing the position in the input of the start of that production, as well as all of the non terminals in that production. Production classes also contain a function that returns the lookaheads for that branch or production; while this could be inlined in places where the checks are performed, it was noted that doing so produced vastly more code due to unnecessary replication.

The generated parser begins with a `Parser` class that is initially populated by the included prelude. The prelude includes utility functions that form the fundamentals of any parser, such as `expect`, `consume`, and `check` methods that control the flow of the parser by attempting to create and consume lexemes. Many utility methods also implicitly skip whitespace, but have override parameters available to permit the discussed functionality of permitting newlines where the grammar specifies semicolons be used. After these functions from the prelude are included, a series of functions are generated, one for each production; these handle the actual recursive descent parsing logic. Each function contains the logic for each branch of its production, and returns one of the possible classes for a branch of that production. For example, considering the production:

```
range_el = clause:a "::" clause:b
```

A function generated from this may look similar to:

```
function createRangeEl() {
  const a = this.createClause();
  this.expect("::");
  const b = this.createClause();
  return new RangeEl0(a, b);
}
```

For branches with conditional logic (such as *option* or *many* qualifiers on tokens), the lookaheads are used to check how to continue to parse the logic. Error reporting is also automatically handled within the `expect` functions, and as the parser maintains a cursor position in the input string at all times, the error messages automatically indicate the location in the input where the error occurs.

Running the parser generator produces a single, large string, containing the code for the parser; this is then saved to a Typescript file (`parser.ts`) and run through an automatic formatter (Prettier[18]) as the generator does not concern itself with producing well formatted code. The produced file then exports the classes and types of each generated branch and production, as well as a class `Parser` which contains a public function that accepts input, parses it, and produces a fully typed AST node of the `program` production, which is the root (first) node of the grammar. Running the parser generator is something of a "oneshot" process, as the process of writing the output to a file is manual; while tedious during development, this is more than sufficient as the parser generator only needs to be run when the grammar itself is changed, which, after correcting a few initial bugs in the grammar and parser generator, was not performed a single time during semantic analysis and code generation. While the parser generator is logically complicated, at 600 lines of dense code, it was still substantially faster to develop and deploy than it would have been to hand write the parser in full; as the generated parser exceeds 4000 lines of code, it was also likely less error prone. The parser generator, and the produced parser, are not the most efficient (in terms of time and space) possible implementations of their concepts, though they are more than sufficient for this project due to not needing to parse tens or hundreds of thousands of lines of code. When tested on the largest piece of "real" S-algol code that could be located (the original compiler), at approximately 2500 lines of code, it was able to parse the file in under 100ms. The utility provided by having a parser produce a fully typed parse tree cannot be overstated, as it immediately catches an entire class of possible errors by having code editors, linters, and the Typescript compiler all catch any issues of attempting to access or use fields of the AST nodes improperly.

6.4 Type Analysis

The semantic analysis stage of the compiler takes the AST generated by the parser and performs both type checking and scope checking. While testing various implementation ideas of the type checker, it was noted that scope checking is very easily integrated into the same code as type checking, as the nature of type checking is that it records information on the types of variables *for each scope*, and thus having a separate process just for scope checking would be replicating most of the functionality of the type checking code.

The development of the semantic analysis stage began by identifying the elements of S-algol that have discrete types that would have to be recorded and checked. These identified types were structures, containing field names and associated types, procedures, with parameters and return types, and forwards. Procedure parameter types are themselves quite complex, as they can be either a regular base type, a procedure type, or a structure. Type classes were created for use in the analyzer with the properties of each type that would be encountered in the code; for example for variables, this would be the variable name, whether it was mutable or immutable, and the type of the variable. A decision was made to utilize strings to represent the types of variables, and then to maintain maps of strings to the type that they refer to (for example a map of strings to variable types). This seemed natural as types are naturally referred to by their ID within most languages including S-algol, for example a procedure may be `procedure double (int i -> int)`. Here, the ID `int` is used to represent the type.

The semantic analyzer operates through maintaining a stack of type "stores". Each type store is a series of maps that map strings to a complex Typescript class of that type (procedures, structures, and variables). When entering a new scope, the analyzer pushes a new store to the stack, and when leaving that scope, the analyzer pops the stack. In this context, a scope is any context where type information should be lost when it is left; for example types defined within procedures, if statements, for loops, or any other sequence blocks. This way of storing types leads to type lifetimes being properly managed, as types and variables are unable to escape the scope in which they are defined as that scope is popped from the stack at the appropriate place. This also has the fortunate property of permitting type shadowing, where a variable can be redeclared with a *different type* and mutability in a nested scope different to the one in which it was declared initially. The process of looking up types is also simple, as it is possible to simply iterate *up* through the stack, checking each scope until one is found in which the type is defined.

The semantic analyzer begins by creating the first type store on the type stack and seeding it with certain types from the standard library. While this could be performed by adding a prelude containing the forward declarations of each structure and procedure used, it is not possible for every type as some have type signatures such as `nonvoid` that cannot be expressed within S-algol. A large part of the logic of the type analyzer is that of comparing two types to see if they match according to the type rules defined in the language specification:

```
type      = nonvoid     | void
nonvoid   = literal     | image   | vector | "pic"
literal   = writeable   | "pixel" | "pntr" | "file"
writeable = ordered     | "bool"
ordered   = "number"    | "string"
image     = "#pixel"    | "#cpixel"
vector    = "*" nonvoid | "*c" nonvoid
```

The type system works through the composition of different type rules. It begins with types being either void, i.e. having no type at all, or nonvoid, from which all other types are derived down to "real" types, or those types that are quoted - similar to non terminals and terminals in a grammar. Thus, a function such as `write` that accepts arguments of type `writeable` accepts any `bool` types or any `ordered` types, and `ordered` types are `number` and `string`. Thus, writeable accepts `bool`, `number`, and `string`. This method also means that more advanced type rules such as those for vectors (suffixing a type with `*`) are automatically handled effectively. A function `expectType` takes the expected type and the actual type and performs type reconciliation, checking if the actual type is able to match the expected type. Further functions were written to verify that expressions that create structures have all fields matching and are of the same type, and a similar function exists for verifying the procedure arguments passed to a procedure call.

The type analyzer operates through depth-first walking of the AST `Program` node, exhaustively and recursively checking every property of every tree node. Methods exist for each type of AST node that consume the node and return a string which is the "type" of the node, although this return type is optional as not all nodes have types. Something like a procedure call expression would have the return type of the procedure, whereas something like a variable declaration statement would itself not have a type. Additionally, every function calls `getType$Node` of all properties of that node even if it does not use the result of that call, as doing so is required to exhaustively walk the AST and perform type validation on every node. For example, a sequence is a series of declarations and clauses, where the last clause in the sequence is the return value (and so type) of the sequence. The function `getTypeSequence` calls `getTypeClause` on each element of the sequence, but unless an error occurs in one of those calls, the result is discarded - only the result of the *final* call is stored and then returned. In this way, exhaustive type checking of any S-algol program is performed.

The actual "checks" of the semantic analysis system occur in `getType$Node` functions where relevant. An example of this is the `getTypeLetDecl`, which accepts the parse node of the following production:

```
let_decl = "let" identifier init_op clause
init_op  = "=" | ":="
```

The function `getTypeLetDecl` begins by checking that the left side identifier does not exist in the *current* type scope. As variable shadowing is permitted, redeclaring a variable in a nested scope is allowed, however if the variable has already been declared in the *current* scope, an error is raised about the identifier already being used within that scope. The function then fetches the type of the `clause` non terminal through calling `getTypeClause`, and this value is checked to make sure it is not `void` - if it is void, an error is raised about the clause being of void return type. Finally, a variable type class is inserted into the current type map with the identifier, clause return type, and whether or not the variable is a constant (depending on which `init_op` is used). Similar processes to this occur in every relevant AST `getType` function in much the same way. The type store stacks are pushed and popped for certain `getType` functions such as for sequences and for clauses within `for` loops and other control constructs. An interesting consequence of the grammar is that certain functions of the standard library such as `reads` are actually *language* constructs rather than usual procedure calls, and so despite registering `reads` as a type, the semantic analyzer handles each of these separate constructs and simply returns the name of the construct that is then later (through the use of the manually registered entries in the type map) resolved to the actual return type of the procedure.

This pattern of effectively tree walking the AST and returning types for each node proves an effective and straight forward one. The support of complex type inheritance like `ordered` permits the easy checking of elements such as in comparison operators, as once the return types for both clauses on either side of the comparison operators are returned, the function `expectType` is simply called with the clause type and the expected type being `ordered`. Through this, this modern S-algol compiler has full scope and type checking for variables, vectors, structures, procedures, dereferences, and all other language constructs.

6.5 Code Generation

The nature of using Javascript as the code generation target is that it is quite straightforward to express a variety of programming constructs that may not be possible in less flexible languages. This is of particular concern when the source language is S-algol, as S-algol has some features that are not often seen in modern languages, and which could be difficult to express, such as how the language permits the use of complex statements as expressions (having sequences as the right side of a variable declaration is one example of this). When targeting a language like C, to express that concept would require creating intermediate variables, new scopes, and possibly entire functions. When targeting Javascript, however, functions are first class variables that can be created and then immediately executed. For the following S-algol code:

```
let a := {let x := 3; x := x + 3; x}
```

When targeting C, generated code might look like:

```
int _a() {int x = 3; x = x + 3; return x;}
...
int a = _a();
```

Immediately we see that this is far more verbose than in S-algol, and as functions can only be declared at the top level, any variables used within the sequence would have to be passed in as functions. By contrast, in Javascript, the generated code would be as follows:

```
const a = (()=>{let x = 3; x = x + 3; return x;})();
```

The flexibility of Javascript permits certain tricks like this, and allows for the full expression of S-algol concepts without having to create and mutate any further intermediate representations of the code.

While the code generator is for the most part independent of the semantic analyzer, it does rely in part on some annotations of the AST that the semantic analyzer writes during its stage. The grammar is for the most part context free, however as noted earlier there is some loss of information that occurs when converting the grammar from context sensitive to context free: namely, whether the clauses in brackets following an expression are structure creation, structure indexing, or procedure calls. The semantic analyzer writes to a property of the AST node for expressions as a result of this - specifically, the property `v_type` indicates the type of the expression. This is then read by the code generator in order to know which code to generate to handle this case.

For the most part, the code generator operates in a very similar way to the semantic analyzer in how it walks the tree depth first, with a function `gen$Node` for each type of AST node that returns a string - except, for the code generator, the returned string is actually Javascript code that is the end result of the process and that will later be executed. As discussed earlier, almost the entirety of the S-algol grammar can be expressed directly in Javascript without having to mutate the structure of the code in any way, although certain tricks such as creating and calling functions immediately are required. Were the compiler to target a less flexible language, this structure of code generation (accepting an AST node and returning a Javascript string) would likely not be possible.

6.5.1 Expressing Language Constructs

Structures in S-algol are created using the syntax `structName(field1, field2, ...)` and indexed using `structVariable(field)`. This is expressed in Javscript by creating objects that map field names to their values. Lookups, then, become simple indexing into that map using regular Javascript indexing notation. For example, a structure type, initialization, and access in S-algol would look as follows:

```
structure s.type (int a)
let s.var := s.type(10)
let res := s.var(a)
```

When converted to Javascript, the above code would look similar to the following:

```
let s_var = {a: 10}
let res = s_var["a"]
```

For loops have a slightly more complicated translation into Javascript code. The grammar defines a for loop roughly as:

```
for $identifier = $clause1 to $clause2 by $clause3 ...
```

One notes that there is no way to modify how the for loop is limited aside from `$clause1 to $clause2`, and the limit is actually defined further in the specification:

> With a positive increment, the for loop terminates when the control constant is initialized to a value greater than the limit. With a negative increment, the for loop terminates when the control constant is initialized to a value less than the limit.

Thus depending on the value of the `by` clause, the Javascript equivalent would be *one of* the following:

```
for (let i = $clause1; i <= $clause2; i += $clause3) {...}
for (let i = $clause1; i > $clause2; i += $clause3) {...}
```

Note how, in the second case, the iterator variable `i` is still incremented - this is because in this situation the `by` clause `$clause3` would actually be negative, and so decrementing a negative number would increment `i`, which is not intended.

Which form of `for` loop to use cannot be determined at compile time as the `by` clause value is determined at runtime. Thus, it is necessary to remedy this uncertainty in the condition part of the loop:

```
for (let i = $clause1;
     $clause3 > 0 ? i <= $clause2 : i > $clause2;
     i += $clause3)
  {...}
```

The ternary statement in the condition determines if the condition should be that of an incrementing loop or that of a decrementing loop.

Case statements have very similar syntax to Javascript and can be mapped with minimal effort, including for default cases. The example case statement given in the reference is:

```
case next.car.colour of
1, 4    : "green"
2       : "blue"
default : "any"
```

Javascript permits fall-through switch cases and default cases, and so the above code simply becomes:

```
switch (next_car_colour) {
  case 1:
  case 4: {return "green";}
  case 2: {return "blue";}
  default: {return "any";}
}
```

As both S-algol and Javascript permit the case conditions (in the example above, the numbers) to be clauses, the same syntax can be used for any comparable type (strings, booleans, etc).

Procedures have nearly identical semantics in S-algol as they do in Javascript. For example, in S-algol, a procedure may be as follows:

```
procedure proc(int param -> int); {...}
```

In Javascript, the type information of the parameters and return type is discarded, and the function is stored as a constant variable arrow function:

```
const proc = (param) => {...}
```

The majority of operators in S-algol, both unary and binary, can be expressed in Javascript in similar ways. For example, `1 + 1` in S-algol would be `1 + 1` in Javascript as well. There are, however, differences for operators that in S-algol use language features that are not available in Javascript, for example binary *picture* operators such as `[1, 1] ^ [2, 2]` which draws a point between cartesian points `(1,1)` and `(2,2)` and returns an image. For operators such as these, additional code must be generated as Javascript has no language builtins for concepts such as pictures in S-algol. The previous S-algol code may become, for example, `connect(point(1, 1), point(2, 2))` in Javascript, where `connect` takes two images and connects the last point of the first image and the first point of the last image, and `point` creates an image with a single point.

Vectors in S-algol are similar to arrays in Javascript, though there is a substantial difference in that vectors have manually defined upper bounds and lower bounds. An example vector may be `vector 1 :: 10 of 1`, which creates a vector with a lower bound of `1`, upper bound of `10`, and fills it with the number `1`. As Javascript has no concept of boundedness, it becomes necessary to somehow persist the lower and upper bounds of vectors, as the values cannot simply be stored in an array as subsequent accesses would be offset by the lower bound and thus would fail. To remedy this, vectors are expressed in Javascript through the use of an object map with the key being the vector position and the value being the value of that vector position. Thus the above vector becomes: `{1: 1, 2: 1, 3: 1, ...}`. This permits lookup using the correct indexes, and using the language builtins `upb` and `lwb` to find the upper and lower bounds of the becomes an exercise of finding the maximum and minimum values of the keys.

6.5.2 A Standard Library

While the specification defines a standard library of sorts, not all of these functions could be implemented in this Javascript compiler due to limitations in the browser based environment. File operations, for example, are not possible, as browsers do not permit disk accesses due to security concerns. Other functions such as controlling the terminal cursor are not possible to implement due to not being able to implement a full VT52 terminal. Additionally, time constraints meant that it was infeasible to implement the full standard library and primitives of S-algol. To that end, a *subset* of the standard library was implemented and further extended with a set of functions to ease the use of S-algol in the browser based environment.

The standard library is built in two parts: the first is a prelude containing series of `forward` statements that are prefixed to the program input prior to semantic analysis. This permits the use of these standard library procedures within the input program, as without the forward statements, the semantic analyzer would raise errors about the procedures being undefined. The Javascript generated by the code generator then contains references and calls to these procedures, but not the definition. When the IDE is about to evaluate the Javascript code generated from the S-algol program, the actual definitions and Javascript functions are prefixed to the program, so that when run, the function calls to these standard library functions resolves properly. The purpose of this split flow - using a sort of function header system, and then defining in "native" code - is that the standard library functions interact with the browser in ways that are not possible in S-algol due to the browser API being Javascript based, and some concepts (such as lambda functions and first class functions) are not possible in S-algol.

A subset of the original standard library procedures are included:

`write(*args)` Write objects to the REPL output

> Writes strings to the output buffer `window.olog` as described previously

`reads(-> string)` Read string from the user

> Operates through calling `prompt`, a browser API method, that presents a blocking diagram requesting user input

`readi(-> int)` Read int from the user

> Operates in a similar manner as `reads`, with the output run through `parseNumber`

`upb(-> int)` Fetch the upper bound of a vector

> Fetches the keys of the vector array and returns the maximum value

`lwb(-> int)` Fetch the lower bound of a vector

> Fetches the keys of the vector array and returns the minimum value

In original S-algol programs, users are often prompted for input through a `write` command followed by a `read` command. This is not possible in this Javascript compiler due to the limitations of the Javascript thread being single threaded, so blocking IO occurs "all at once" with no time to `write` and have output displayed to user. As the `write` then `read` pattern is common, procedures were introduced to prompt the user in the `read` dialog box:

`reads.prompt(string -> string)` Prompt user for string input

> Similar to `reads`, but passes a string to `prompt` which is then displayed in the dialog

`readi.prompt(string -> int)` Prompt user for int input

> Similar to `readi`, but passes a string to `prompt` which is then displayed in the dialog

This implementation of an S-algol compiler and run environment also implements support for vector graphics. Vector graphics in S-algol use a variety of language primitives to mutate `#pixel` types that are not easily mapped to Javascript, and so vector support is built from scratch manually. Functionality is supported for creating blank images, drawing lines and adding points, shifting, scaling, rotating, and merging images. The representation of a `#pixel` type in Javascript is that of a 2d array of the form `[[x1, y1], [x2, y2]][]` - or, an array where each element is an array of two elements, with each sub-element being cartesian coordinates. When rendering to the canvas, the image handler iterates through all lines (elements) in the array and draws corresponding lines at those coordinates on the canvas. While the output canvas is of a set static size, some example S-algol programs - like that which renders the Murray Polygon, or Escher's Square Limit - are either far too large or far too small. To address this, a function `draw.scale(int, int)` is available that sets a multiplier on all cartesian coordinates in order to scale them to the canvas. Further, the procedure `draw.image(#pixel)` is used to draw the image to the canvas.

In S-algol, one adds points and lines to an image through the caret operator, for example:

```
[x1, y1] ^ [x2, y2] ^ [x3, y3]
```

The above would draw lines between `(x1,y1)<>(x2,y2)` and `(x2,y2)<>(x3,y3)`. When generating code to handle this, for each operator the left side is taken as an image (a `#pixel` type), and the right as one cartesian coordinate. An entry to the image array is then made by drawing a line between the second coordinate of the last entry of the image array, and the coordinate of the right side of the operator. When creating a new array, the first "line" in the array is simply a single point. Thus, the above would represented as so in Javascript:

```
[[[x1, y1], [x1, y1]],
 [[x1, y1], [x2, y2]],
 [[x2, y2], [x3, y3]]]
```

S-algol implements a variety of vector manipulation language constructs, such as:

```
shift $image by $x,$y
rotate $image by $r
```

Each returns a new image, and so they can be composed as, for example (where `p` is an image):

```
shift rotate p by -90 by 1, 0
```

The main primitives of `rotate`, `shift`, and `scale` are implemented. Rotation is performed through simple trigonometric mutations of each point in the array, scaling is performed through multiplying each coordinate by constant `x` and `y` quantities, and shifting is performed by incrementing each coordinate by constant `x` and `y` quantities. Through the implementation of these simple primitives, the compiler and browser environment are able to produce complex vector graphics such as the Murray Polygon and Escher Square Limit, as shown in Figure 7.

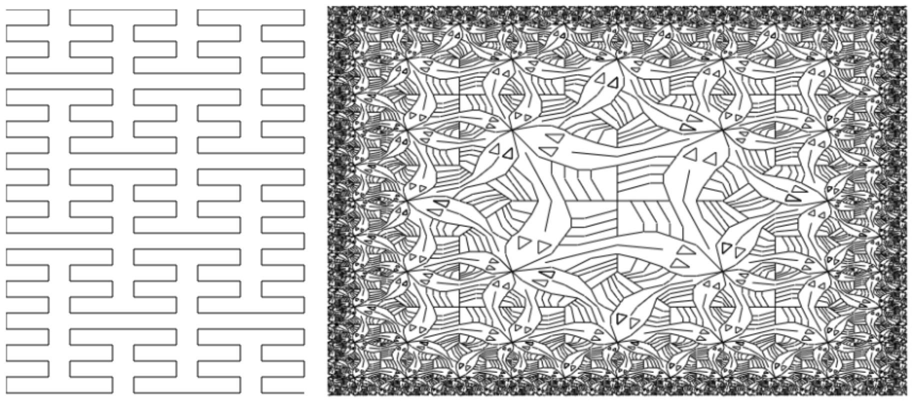


Figure 7: Rendered vector graphics

6.6 Blocking Input and Rendering Output

A significant hindrance in the execution of a S-algol program compiled to Javascript is that of concurrency. Tabs within browsers are single threaded, and the only way to execute Javascript code from within Javascript is through the use of the `eval` function, which runs the code on the thread of the tab running the code that calls `eval`. This means that, when running an S-algol program, it is the only thing that can be running at that time; as a consequence, the browser based IDE/REPL is effectively frozen as the S-algol program runs. Because of this, there is no way to interact with the user through the IDE/REPL UI while the S-algol program is running, as any methods to request user input would naturally be blocking, therefore blocking the main thread running the IDE UI, prevent user input .. and so on. This also has the limitation that introspection of the running S-algol code is not possible, and so there are no ways to catch things like infinite loops, as they will instead just freeze the tab until the browser automatically stops the process.

Output is dealt with through the S-algol program writing to an array which is displayed after the program has finished execution. Blocking input, however, is only possible through the use of the Javascript `prompt` function, which presents a dialog asking for user input, an example of which is shown in Figure 8.

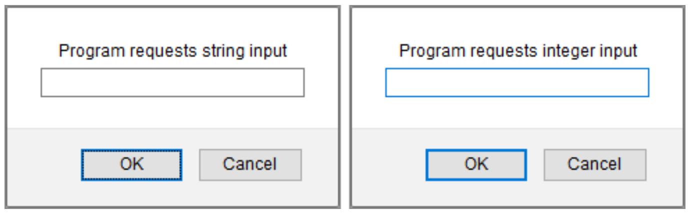


Figure 8: Input dialogs

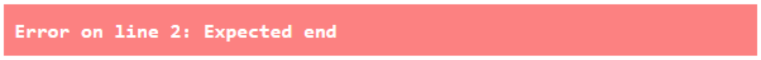
Textual output is handled exclusively through the `write` language feature, which accepts a clause list and displays the list to the user. As mentioned previously, this works through writing to a property `window.olog`, an array of strings, where each string is a “line” of output. When the S-algol program has finished executing, the React component re-renders and updates the area that displays the result of `window.olog`; a `<pre>` monospaced text area. While this form of output is quite rudimentary and does not seek to emulate a terminal (for example ANSI escape sequences are not implemented), coupled with the blocking input boxes, it is sufficient for basic IO.

Vector graphic display is handled through the use of a HTML `<canvas>` component. The S-algol function `draw.image` is made available that, when called, invokes a function property `window.setImg`, which stores the image array. When the S-algol program has finished executing, React passes this image array property to the canvas component. When the canvas component notices that the properties passed to it have changed, it clears the canvas and resets the “cursor” position. It then iterates through the array of coordinate pairs, drawing a line between each coordinate in each pair. This operation is extremely fast to perform even for more complex images (such as Escher’s Square Limit). When adding an artificial delay in between drawing lines, the process of drawing the lines one by one is visible.

Through these methods, it is possible to conduct blocking IO as well as render out vector graphics and text. This is not the same as in a true S-algol terminal, where the program interacts directly with the terminal and where text and graphics can be mixed, however given the constraints of the environment, these methods seem the only current solution.

6.7 Error Handling

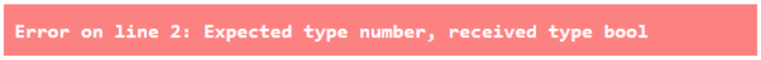
Possible errors in the compiler can occur at three separate points: during parsing, during semantic analysis, and during runtime. Errors in parsing and semantic analysis are raised through the use of Javascript exceptions, and then are caught by the IDE/REPL interface that called the compile methods. These exceptions are then displayed to the user, indicating both the position in the code that the error occurred at, as well as the error itself.



```
Error on line 2: Expected end
```

Figure 9: Example of a syntax error

Figure 9 demonstrates an error that occurs during parsing, where a sequence has been left without an `end` statement. The nature of the LL(1) parser is that it knows, in a production, what the next token should be; when that next token is missing or replaced with something else, it is possible to indicate exactly what was expected but not received.



```
Error on line 2: Expected type number, received type bool
```

Figure 10: Example of a type error

Figure 10 shows an example of an error raised during type analysis, where a procedure takes a parameter of type `number`, but has been called with a value of type `bool` instead. As S-algol is statically typed, it is possible to display the exact types of what was expected and what was received.

The final element of this project where errors can occur is in the runtime of the compiled S-algol program. Unfortunately, as mentioned previously, the compiled code is executed using the `eval` function; this operates something like a black box where introspection is impossible to do, and the same is true of error handling: errors that occur cannot be caught by the code that calls `eval`, and are instead pushed directly to the browser console. There is, regrettably, not much that can be done about this; the browser simply does not provide sufficient APIs to interact with code running in `eval`. Fortunately, the nature of a statically typed language is such that many possible errors occur and are caught *during compilation*, for which error handling *is* possible, and as shown above, performed correctly and displayed to the user.

6.8 Testing

The S-algol language is complicated. There are around 80 productions in the original grammar, and the language permits certain constructions that are quite unique, such as meshing together expressions and statements, and having graphic operations as language primitives. Other features, such as implicit returns for final statements in a sequence, are also unusual for modern languages. Testing is therefore clearly important, but complicated.

In order to test the parser and semantic analyzer, it was necessary to use input sufficiently complicated so that the probability of running into edge cases would increase. The largest existing S-algol program that was readily available was that of the original compiler that was written in S-algol - `comp.S`. The compiler uses nearly every language feature bar image manipulation, and creates and uses hundreds of procedures and types. During the development of this project, `comp.S` was used to benchmark the completeness of the parsing and semantic analysis stages; progress was fairly incremental as each addition to the parser and semantic analyzer would permit further execution in compiling `comp.S` before running into errors. This was an effective technique - it even drew attention to the apparent errors in the original grammar, as some elements of `comp.S` seemingly violated parts of the original grammar (declaring forwards without defining their types).

While effective for testing parsing and semantic analysis, using `comp.S` for testing code generation was unrealistic as the code in `comp.S` used features that were not possible to implement in the browser (file IO for example). It was also unsuitable for testing specific elements of the language, as it was not a purpose built test suite. A purpose built test suite was clearly in order, and so one was built. Available by default as one of the preloaded programs in the browser S-algol compiler, `tests.S` tests the majority of the language features one at a time. Control flow, IO, operators, structures, vectors, and procedures are all tested as part of this suite. Each test returns some output that is dependent on the test "working", which is then compared against the theoretical/expected output. These actual/expected results are then written to the output display for the user to verify:

```
test while, expect 20 -> 20
test vector access, expect 6 -> 6
test struct int, expect 10 -> 10
test struct bool, expect true -> true
test struct assign, expect 20 -> 20
test if, expect 10 -> 10
test if, expect 21 -> 21
test for, expect 6 -> 6
test case, expect 3 -> 3
...
```

Graphics were tested using the files `fast.S` for generating a Murray Polygon, and `escher.S` for generating an Escher Square Limit image; these are run and tested separately.

These tests were run constantly during the implementation of this browser based S-algol compiler. This was an effective way of testing that any changes to the S-algol compiler pipeline (parsing, semantic analysis, code generation, and execution) caused no regression bugs, and through using test driven development, served as valuable indicators for showing how much of the language had been successfully implemented.

7 Evaluation and Critical Appraisal

Building a compiler and runtime is a complex task involving many moving parts. The task is made more difficult in proportion to the complexity of the language that it compiles and the standard library and features that are required for basic operation.

S-algol is a complex language. The syntax is long and complex, and the language primitives are complex - few modern languages include features like image manipulation as part of the syntax itself.

The modern implementation of S-algol described in this document represents a substantial success in spite of these complexities. The compiler is capable of parsing the full S-algol grammar, the semantic analyzer is complete, and the code generator is capable of generating Javascript equivalents for the vast majority of the grammar. The browser based Integrated Development Environment features a code editor with syntax highlighting for the S-algol grammar as well as keyword autocompletion, and the runtime permits blocking IO and for the rendering of vector graphics. Support for existing S-algol programs is such that programs written in 1987 that produce the Murray Polygon are capable of running with almost no changes at all (the only modification is using a different standard library call to render to the browser canvas).

The original requirements of this project are almost all met. The requirements that are not met - debugging running S-algol code as well as self hosting the original S-algol compiler - are due to the browser not providing sufficient APIs for the introspection of running Javascript, and a lack of real filesystem IO for self hosting the original S-algol compiler. Additionally, raster graphics could not be implemented in the time available for this project. Certain missing features such as raster graphics and elements of the standard library could be implemented given a longer project period, provided that the reason for their omission was not due to limitations of the browser environment.

There are a few decisions that were made during the design and subsequent implementation of this project that are of particular note. The usage of Typescript as the language in which the compiler and browser IDE components are written provided an excellent developer experience, as type hinting during editing and type checking during compile time vastly sped up development and reduced the cognitive overhead of having to remember the types of the complex objects unavoidable with such a complex task. The usage of a parser generator provided substantial and undeniable time savings; the small upfront investment into the relatively small generator saved having to write thousands of lines of tedious and error-prone parsing code.

Having learned a substantial amount about writing compilers, there are perhaps elements that could have been designed differently to reduce future issues. During semantic analysis, the internal representation of types was performed using strings to represent types and lookup maps for the complex type objects that they represented. While this is a simple approach, there are a number of edge cases that had to be handled due to, for example, variables going out of scope and yet still returning the type of the out of scope variable (the solution to this was to sanitize variables on exiting a scope). Returning the complex types initially may have led to slightly more design considerations, but overall would have been a simpler approach as less edge cases like the above would need to be handled.

The works documented in this practical compare favorably with the existing browser based IDEs like `repl.it` and the teaching programming language `scratch`. The implementation described in this report matches many of the features of `repl.it` - the editor, single click compiler integration, and blocking IO - while also providing certain features that `repl.it` does not provide, such as the ability to render out vector graphics. This implementation also provides the ease of use and visual display features that make `scratch` an effective language for teaching.

8 Conclusions

This project contains a modern compiler pipeline that is capable of taking original S-algol programs written over three decades ago and producing Javascript code that can run in a browser environment, interact with the user, and render textual and vector graphics.

The majority of the original S-algol specification is implemented in ths project. As described during the report, almost all of the syntax is implemented, and certain new IO and vector functions are introduced to allow operation and interaction within a modern browser. Regrettably, browser API constraints restrict the full implementation of the standard library as there are no ways to perform file IO, and time constraints restrict the complete implementation of all features such as raster graphics.

The strict separation of concerns of *all* aspects of this project, from the compiler pipeline stages to their integration within the browser environment, permit the easy and incremental improvement of *any* aspect of this project. In future projects and implementations, features such as raster graphics and a virtual filesystem could be implemented. As it stands currently, this implementation of S-algol in the browser returns to life a type of programming language that is not often seen in the modern era.

Testing Summary

Variable creation, update, access

```
let n := 13
write n
n := 20
write n
```

Result:

```
13
20
```

Vector creation, update, access

```
let v := vector 1 :: 10 of 1
write v(1) + v(5)
v(5) := 5
write v(1) + v(5)
```

Result:

```
2
6
```

Structure creation, update, access

```
structure sdef (bool b; int c)
let s := sdef(true, 10)
write s(b), ":", s(c)
s(b) := false
s(c) := 20
write s(b), ":", s(c)
```

Result:

```
true:10
false:20
```

While loop

```
let n := 13
while n < 20 do n := n + 1
write n
```

Result:

```
20
```

If statement

```
let n := 10
let b1 := if n = 10 then true else false
let b2 := if n = 20 then true else false
write b1, ":", b2
```

Result:

```
true:false
```

Incrementing for loop

```
let n := 0
for i = 1 to 10 by 2 do n := n + 1
write n
```

Result:

```
5
```

Decrementing for loop

```
let n := 0
for i = 10 to 1 by -2 do n := n + 1
write n
```

Result:

```
5
```

Case statement

```
procedure proc (string n -> int)
     case n of
     "a": 1
     "b": 2
     "c": 3
     default: 0
write proc("a"), proc("b"), proc("c"), proc("d")
```

Result:

```
1230
```

Procedure creation, call with single statement

```
procedure proc (-> int); 3
write proc
```

Result:

```
3
```

Procedure creation, call with multiple statements

```
procedure proc (-> int); {let n := 10; n := n + 10; n}
write proc
```

Result:

```
20
```

Attempting to access unknown structure field

```
structure sdef (int n)
let s := sdef(10)
s(nn)
```

Result:

```
Error: Unable to match field nn
```

Attempting to create structure with wrong field type

```
structure sdef (bool b)
let s := sdef("hello")
```

Result:

```
Error: Expected type bool actual bad type string
```

Attempting to create structure with wrong number of fields

```
structure sdef (bool b)
let s := sdef(true, 10)
```

Result:

```
Error: Expected 1 fields for structure, received 2 fields
```

Attempting to assign to constant variable

```
let n = true
n := false
```

Result:

```
Error: Unable to modify constant variable n type bool
```

Attempting to assign wrong type to variable

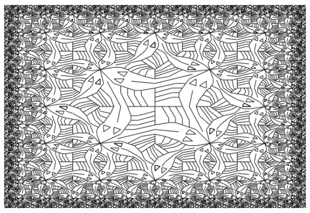
```
let n := true
n := "world"
```

Result:

```
Error: Expected type bool actual bad type string
```

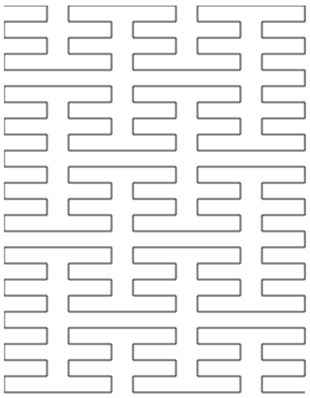
Running `escher.S`

File is included in the compiler.



Running `fast.S`

File is included in the compiler. Ran with arguments `2`, `2`, `3`, `5`, `5`, `5`.



User Manual

The project is "built" (compiled from Typescript to Javascript) using NodeJS. The version of NodeJS used was `v13.11.0`. Dependencies are handled through `yarn`, an alternative to `npm`. `yarn` can be installed by running:

```
npm install -g yarn
```

A lockfile of all dependencies used for building this project is included: `yarn.lock`. To install all dependencies, when in the root of the compiler (the directory containing `yarn.lock`), run the following:

```
yarn install
```

To then compile the project for deployment, run:

```
yarn build
```

This will produce a folder `build/` in the same directory containing the compiled project resources. To then use the compiler, open the file `index.html` in a web browser.

Note It may be required that something such as `nginx` or `apache` be used to make the project available, as not all browsers permit local filesystem references (so the main compiler Javascript file would not be loaded).

A pre-built version is included in `build/`. To prepare for a clean build, delete the folders `build/` and `node_modules/`.

To run this project in `develop` mode (where changes to the source files causes automatic recompilation and hot-reloading), perform the above steps, except instead of running `yarn build`, run `yarn start`. This should open a browser window directed to `http://localhost:3000`.

This project was tested using the browser Firefox 75.0 (64-bit).

References

[1] R. Morrison, “On the Development of Algol,” PhD thesis, University of St Andrews, 1979.

[2] A. J. T. Davie and R. Morrison, “Recursive Descent Compiling.” 1981.

[3] “Usage Statistics of Javascript as Client-Side Programming Language on Websites.” https://w3techs.com/technologies/details/cp-javascript (accessed Apr. 19, 2020).

[4] “Typescript.” https://www.typescriptlang.org (accessed Apr. 19, 2020).

[5] “BuckleScript.” https://bucklescript.github.io (accessed Apr. 22, 2020).

[6] “Elm.” https://elm-lang.org.

[7] “Repl.it.” https://repl.it.

[8] “React.” https://reactjs.org (accessed Apr. 19, 2020).

[9] “Monaco Editor.” https://microsoft.github.io/monaco-editor (accessed Apr. 19, 2020).

[10] “Create React App.” https://create-react-app.dev (accessed Apr. 19, 2020).

[11] “JS Beautify.” https://github.com/beautify-web/js-beautify (accessed Apr. 19, 2020).

[12] C. Larman and V. R. Basili, “Iterative and Incremental Development: A Brief History.” 2003.

[13] “React Router.” https://github.com/ReactTraining/react-router (accessed Apr. 19, 2020).

[14] “Redux.” https://github.com/reduxjs/react-redux (accessed Apr. 19, 2020).

[15] “Visual Studio Code.” http://code.visualstudio.com (accessed Apr. 19, 2020).

[16] “Yacc.” https://linux.die.net/man/1/yacc (accessed Apr. 19, 2020).

[17] “ANTLR.” https://www.antlr.org (accessed Apr. 19, 2020).

[18] “Prettier.” https://prettier.io (accessed Apr. 19, 2020).