UNIVERSITY GARDENS HIGH SCHOOL
SAN JUAN, PUERTO RICO

Domain Specific Language for differential equations with Scott-Strachey semantics

By
Benjamin Philippe Applegate
Jan-Paul V. Ramos Dávila

ABSTRACT

Domain Specific Languages give the user a rigorous environment for solving problems in a specific domain, in this case differential equations. There are many automatic differential programs that utilize operational semantic algorithms with the chain rule to apply differentiation. Scott-Strachey semantics (also known as denotational semantics) employ notations of differentiation familiar from real analysis. These denotational semantics are important in computing and pure mathematics because they formulate the meaning of programming languages by constructing denotations (mathematical objects) that describe the meanings of expressions from the language. Our DSL with its own Expression Sublanguage for the calculus creates an intuitive environment for any user, but at the same time these expressions can be notated mathematically using denotational semantics to express our differentiation algorithms in a purely mathematical way. These denotations can be used to find extensional equality from our DSL to another language, taking that two functions are equal if and only if they map the same inputs to the same outputs, it has to be proven whether the expressions in question denote extensionally equivalent functions.

TABLE OF CONTENTS

Introduction

Domain Specific Languages give the user a rigorous environment for solving problems in a specific domain, in this case differential equations. We utilize TypeScript to build the language, although the DSL is independent from the TypeScript infrastructure. This compiles to JavaScript where the user interacts with a website GUI to program in. The implementation of a GUI provides simplicity for the user when using the DSL.

A DSL for differential equations serves as a unique way to apply automatic differentiation and a visual interface for calculating the derivatives of certain functions described in the program. These procedures rely on the chain rule to obtain the desired derivatives, defining derivation as an operator, with a GUI that provides the user with a comfortable environment for solving the differentials. Our DSL is categorized as an external DSL, which provides us with an entirely stand-alone language, and involves the stages of lexical analysis, parsing, interpretation, compilation and code generation. This provides us with our own syntax and semantics. The automatic differentiation is extremely useful in scientific computing and in machine learning, however the relation between this automatic differentiation to the classical mathematical notation of the derivative isn't clear. To study the details of the differential algorithms, the use of operational and Scott-Strachey semantics is important.

Operational semantics employ popular implementation techniques for differential algorithms, whereas Scott-Strachey semantics employ notations of differentiation familiar from real analysis. The specialty of applying Scott-Strachey (also known as denotational) semantics to our DSL is the formulation of the language using mathematical objects (denotations) that describe the meaning of the expressions in the language. These mathematical objects would

represent the domain of differential equations. Denotational semantics for differential equations provide a useful translation of the language into mathematical notation, which can then be translated into byte-code and another language. The understanding of our DSL with denotational semantics provides a clear mathematical representation of our objects. For example, the expressions `(2+2)`, `(2*2)`, and `004` are different in notation, but syntactically they all denote the same abstract object `4`. The denotational definition for such expressions would show that

```
meaning[[(2+2)]] = meaning[[(2*2)]] = meaning[[004]] = meaning[[4]] = 4
```

Differentiation functions play a prominent role in denotational semantics, modeling the bindings in stores and environments as well as control abstractions in the DSL. An example for this would be the factorial function: `fact(n) = if n = 0 then 1 else n*fact(n-1)` (in the DSL's notation: `fun fact n: if == n 0: 1 else * n fact - n 1`) where a mathematical object that can be viewed as a set of ordered pairs, `{<0,1>,<1,1>,<2,2>,<3,6>,<4,24>,<5,120>,<6,720>,...}`, and a denotational semantics should confirm this relationship. The denotational specification for our DSL consists of four components, one specifying the syntactic world, one describing the semantic domains, and two defining the functions that map the syntactic objects to the semantics objects.

While going through all the indicated steps for developing our DSL and the GUI using TypeScript, algorithms using the chain rule are implemented to solve derivatives and their antiderivatives. This DSL fulfills the purpose of creating an environment in which a user can explore the entirety of our language with ease and be able to successfully solve any differentials. For the implementers, denotational semantics of the program are created, providing a rigorous mathematical definition for the DSL's grammar. This rigorous mathematical description can be used for language translation and a wider implementation of our differentiable algorithms, giving

the implementer the opportunity to find algorithms which notably different arguments with ours, but denotationally they are the same function. The denotations serve to prove extensional equality between our mathematical objects and other implementor's language's objects. Throughout the paper, Domain Specific Language will be referred to as DSL, the syntax [number-number] refers to the starting and ending lines of code the text is referencing, and various terminology is shortened after mentioned for the first time.

Justification

Domain Specific Languages (DSL) are created for the purpose of solving problems in one domain. A domain is the area the programming language focuses on, and most of the time a DSL will only be capable of solving problems in this domain. In contrast with a general purpose programming language, which is capable of solving problems in multiple domains, the design and implementation of a DSL are very specific to the problems that it solves, therefore both the user and the authors have an easier time understanding and using the language. The declared syntax and grammar of a DSL with a differential equation domain is designed and implemented for the most optimized user experience for solving differential equations. The user-interface aspect of the DSL provides a clear workspace for the user, with various options tailored to the use of differential equations, that a general purpose language wouldn't provide. DSL sacrifices the customization and provides a rigorous environment to solve differential equations optimised and comfortable. The classification of a DSL is either external or internal (also known as embedded). The use of an internal DSL, using the infrastructure of a base language to build these domain specific semantics on top of it, implemented as a form of library, provides the option for domain-specific constructs need not to be strictly obeyed, and transformations are not required. Our implementation of an external DSL is an entirely stand-alone language, providing stages of lexical analysis, parsing, interpretation, compilation, and code generation. This gives us the opportunity to have our own syntax and semantics.

Scott-Strachey semantics serve as an approach of formalizing the meaning of our domain specific programming language by constructing mathematical objects called denotations. These denotations describe the meanings of expressions from our language, and are useful for the

translation of one programming language to another. By translating the differential equations DSL into a process calculus, the author has the mathematical notation for the semantics of the language. This can be used to translate it into byte-code and implement it in another programming language. By giving a mathematical understanding to the differential equations DSL, the algorithms implemented for the domain can be understood in a broader context, independent from a computer, and applied by theoretical computer scientists and pure mathematicians. The use of denotations provides a comparison between the DSL semantics and other semantics. Two functions are equal if and only if they map the same inputs to the same outputs, in which we prove whether the expression in question denotes extensionally equivalent functions. Two functions can have notably different arguments, but the denotationally be same function. A mathematical representation of the semantics can provide an answer to various questions in computing, ranging from machine learning to cryptography. For example, in reactive programming, when can we say that two different expressions nevertheless denote the same event stream.

The automatic differentiation in the external DSL provides not just an opportunity to search for differentiation algorithms, but provides special semantics to analyse with denotations. Most importantly, it serves to prove extensional equality, and answers when can we say that two function-valued expressions actually denote “the same” function, and thus can substitute each other.

Literary Revision

I. Mathematical Background: Calculus, Linear Algebra, and Differential Equations

Calculus has a long history, starting with its original name or infinitesimal calculus. The name of infinitesimal calculus comes from the study of continuous change, a similar way geometry studies a shape and algebra studies arithmetic (O'Connor & Robertson, 1996). Isaac Newton and Gottfried Wilhelm Leibniz each developed the theory for infinitesimal calculus in the 17th century, completely independent of one another and not even knowing another mathematician was developing the ideas. The first study of infinitesimals comes from Ancient Greece, in which we can see Democritus utilize the concept in geometrical razionalizations in cones, which actually prevented him from accepting the idea of infinitesimals (O'Connor & Robertson, 1996). Calculus originally comes from "small pebble", because small pebbles were used for calculations, but now it's just used as a method of computation (O'Connor & Robertson, 1996). Because of the various types of calculation complexity, related theories are given their own sub-areas (O'Connor & Robertson, 1996), such as propositional calculus, lambda calculus (mainly used in computation) (Edalat, 1997), and process calculus.

An important part of Calculus are foundations (O'Connor & Robertson, 1996). These refer to rigorous developments of Calculus from axioms and definitions. In modern mathematics, these foundations are part of real analysis. Applications of Calculus relate to problems in all types of sciences. Lagrange brought the whole range of Calculus into real analysis. We can find the applications of Calculus in various other fields of mathematics, such as Linear Algebra. Similarly to linear differential equations, linear algebra focuses on linear equations and linear maps (Chen, 2011), and their representations in vector spaces through matrices. The concept of

matrices in Linear Algebra is important as it is dependent on partial differentiation and the solvability of an equation by sections. As Calculus is related to real analysis, Linear Algebra refers to functional analysis, as it is an application to spaces of functions (Chen, 2011).

An important aspect of Linear Algebra are endomorphisms and square matrices. A linear endomorphism relates to a linear map that maps a vector space to itself. Linear endomorphisms and square matrices have properties that make their study extremely dependent on linear algebra, such as geometric transformations, coordinate changes, and quadratic forms. Eigenvalues and Eigenvectors are an example of linear endomorphisms, and are strongly related to computation. An example of this is Google's PageRank, which uses eigenvalues and eigenvectors as a way of forming their search engine. They form a link matrix representing the relative importance of the links in and out of each page. Although efficient, this requires a lot of computer power, which requires constant mathematical optimization processes.

Processes in numerical analysis that are the epicentre of mathematics in computation include convexity, interpolation, spline interpolation, and Lagrange Interpolating Polynomials. A Lagrange polynomial is used for polynomial interpolation. This concept relates to the interpolation of a given data set by the polynomial of lowest possible degree that passes through the points of the dataset. For a given set of points (x_j, y_j) the Lagrange polynomial is the polynomial of lowest degree that assumes at each x_j the corresponding value y_j, so that the functions coincide at each point (Chen, 2011).

As discussed beforehand, the intersection between Calculus and Linear Algebra depends on the study of differential equations. A way of understanding these concepts is by starting at ordinary differential equations, or as we will be referring to them in this paper ODEs. ODEs are differential equations containing one or more functions of one independent variable and the

derivatives (Bluman & Dridi, 2012). It is important to note that there can only be one independent variable, as the term ordinary is used in contrast with the term partial differential equations, or PDEs, which may be with respect to more than one independent variable. These equations are characterized by their linearity, which is defined by a linear polynomial in the unknown function and its derivatives (Bluman & Dridi, 2012). We can write an equation in the form $a_0(x)\,y + a_1(x)\,y' + a_2(x)\,y'' + \ldots + a_n(x)\,y^{(n)} + b(x) = 0$, where *a_n(x)* and *b_n(x)* are arbitrary differentiable functions that do not need to be linear, and *y',...,y^(n)* are the successive derivatives of the unknown functions *y* of the variable *x*.

We can classify linear systems of ODEs in types for various functionalities by knowing First-Order ODEs and Higher-Order ODEs. This type of First-Order ODEs may not be confused with first order logic utilized in logical gates (Jamieson & Brown, 2020). They can be characterized even further by taking a look at Linear Equations, Separable Equations, Exact Equations, Bernoulli Differential Equations, Substitutions, Intervals of Validity, Modeling with First Order Differential Equations, Equilibrium Solutions, and Euler's Method (Hale, Lunel, Verduyn & Lunel, 1993). Secondary Order derivatives are just characterized by them being the second derivative.

In regard to Linear Algebra, we utilize these matrix based processes for solving PDEs. PDEs are characterized by their multiple independent variables, and solving for each variable in relation to the other as a constant by halves, and ending up with a partial differential. Matrices become useful here for their nature in linear equations. Solution techniques include the separation of variables, Laplace Transformation, and Similarity Solution. Mathematical models that are derived to apply include Parabolic equations, Poisson's Equation, and the Wave Equation (Hale, Lunel, Verduyn & Lunel, 1993).

Some numerical analysis include the Trapezium Rule, Simpson's Rule, and Gaussian integration, which are all techniques for approximating the definite integral. Applications such as MATLAB and the open source alternative Scilab utilize numerical algorithms in terms of data types to apply these rules in our everyday lives.

II. Applications of Mathematical Concepts in Computation and Modern Computation and Future of Mathematics in Computational Methods

Mathematics and computation have always been inextricably tied, and there are very interesting overlaps. Over time, powerful fields of mathematics have inspired entirely new ways to program, such as dependent types (Xi, 1998), which can be used to formally prove mathematical aspects of your programs, such as guaranteed termination and a covering of all possible inputs. Other ways include statistical programming (Poulson, 2019) and array programming (GitLab Unfiltered, 2020), which is based on vectors. A fairly new field of mathematics that has given rise to a powerful new way to write programs is category theory (Wells, Charles, 2012), which is currently being implemented as typeclasses in languages such as Haskell and the dependently typed Idris.

Dependent types are a very intriguing aspect of programming, as it's a way programming has given back to mathematics, in the sense that it started out as a way to tell the program what kind of data a variable held, but turned into its own constructionist theory of mathematics, and an entire alternative foundation of mathematics. This is because types in dependent type theory correspond to logical propositions, and values to proofs of the proposition of their type. This means that to prove a proposition true, you have to *construct* a value of its type, using a program. The advantage this has over classical theories for computers is that these proofs can be completely automated and verified by the computer, as to reduce human error. This has the

potential of making larger mathematical proofs more solid that otherwise might be very hard to check for human mathematicians. A benefit for computer science is that this means functions can be checked to be total, essentially always terminating and covering all cases (Xi, 1998.) This doesn't eliminate all bugs, but can make a computer system safe from very costly common errors such as null pointer dereferencing, division by zero, and out of bounds array indexing.

Dependent type theory maps all the logical tools into type tools. Logical conjunction (the and operator) corresponds to a pair of types, disjunction (the or operator) corresponds to the either type, which has a value of either of two types, implication corresponds to functions, and falsity is simply a function taking a value of a false type and returning a value from the empty type, which is of course impossible. But the two logical tools that distinguish dependent type theory from polymorphic type theory are universal and existential quantification, which correspond to the dependent product and sum, respectively. The dependent product $\prod_{x:A} B(x)$ is a function that takes a value *x* of type *A*, and returns a value of type *B(x)*, a function taking *x* and returning a type. This is what gives the ability to construct higher level types, such as the type of numbers greater than another number, being a dependent function taking two numbers and returning a proof that the first is bigger than the second. This can be used to implement functions with constraints, such as requiring a natural number subtraction function requiring the second number to not be greater than the first, or division requiring that the divisor not be zero.

Dependent sums on the other hand are a pair $(a, b) : \sum_{x:A} B(x)$, where the type of *b* depends on the value of *a*, which can be used for example to package a value with a proof about it.

Category theory is a relatively new field of mathematics, and also an alternative foundation, which has had a great influence on the functional programming paradigm. This

works pretending the entire language is a category, with types being objects in said category, and the arrows between them being functions, similar to how the Set category works. The part that makes this useful in programming is that then category theory's functors and monads as typeclasses, which are somewhat similar to the programming concept of interfaces. These typeclasses can then also be used to implement interesting mathematical combinators (Meijer, E., Hughes, J. (Ed.), Fokkinga, M. M., & Paterson, R., 1991) that can be used to reduce explicit recursion, greatly simplifying many programs that would otherwise use loops or recursion.

Problem

General purpose programming languages written at high level operational semantics depend on a machine. The state changes in operational semantics are defined by coded algorithms for a virtual machine, whereas in denotational semantics, they are defined by rigorous mathematical functions. The machine dependency is gone, providing a mathematical understanding of objects, and the language itself becomes mathematics (lambda calculus). A mathematical understanding of objects helps solve common problems in computation with the use of denotational semantics. The DSL searches to optimise the user experience for automatic differentiation, while a mathematical translation for the grammar serves as an implementer to answer: When can we say that two function-valued expressions actually denote "the same" function, and thus can substitute each other?

Hypothesis

Designing a domain specific language for differential equations with Scott-Strachey semantics results more expressive than general purpose programming languages with operational semantics by constructing mathematical objects (denotations) that describe the meanings of expressions from the language. The user gains more expressive power and a mathematical understanding of the objects in the program's domain (representation of what the program does). This also results in denotations which can be used by any implementer to translate to their own language and apply to a broader mathematical context.

.

Methodology

I. Differential Equations DSL

a. Decision & Analysis

We decided to make automatic differentiation the domain of the language. With automatic differentiation we can provide an innovative solution towards solving differentials. The conclusion is that it will be syntactically original (made to be as best as possible for the DSL's purpose), concerning itself only whether or not the sentence is valid for the grammar of the language. The choice for automatic differentiation brings the ability to semantically break down the differentiation algorithms into their meaning mathematically. This helps us identify whether or not the sentence has a valid meaning. The entire language is written in TypeScript (TS), which compiles to JavaScript (JS), and the grammar is made with a parser generator for JS.

b. Design

i. **(Source Code A)**

The Abstract Syntax Tree (AST) serves as the DSL's tree representation of the abstract syntactic structure of the source code. Each node denotes a construct occurring in the source code. It is *abstract* in the sense that it doesn't represent every detail appearing in the real syntax, but rather just the structural or content-related details. This tree will be the main data-structure that a compiler or interpreter uses to process the program. The lines [1-35] describe the types that structure the DSL's AST. The main type is the Instr type, which describes the structure of all of the different instructions. The lines [37-61] describe the types that structure the Expression Sublanguage's AST. The main type is the bottom Final type.

ii. **(Source Code B)**

The grammar lets us transform a program written in the DSL into its corresponding AST form. Only programs that are syntactically valid can be transformed in this way.

<Add formal grammar>

The lines [1-91] are the parser for the DSL, and the lines [93-151] are the parser for the Expression Sublanguage.

iii. **(Source Code C)**

The compiler is the language processor that reads the source program and turns it into JavaScript which the browser can run.

c. Implementation

The DSL is used in a website that has an input box for input short programs, that are then immediately run. Any result the DSL returns is printed above the input box (in the log,) and has a button that fetches the value so the user can refer to it again in the input box. (This feature is called the result history.)

d. Deployment

Whenever the code is modified on GitHub, the website is updated. Also, whenever a milestone is reached, a GitHub release is made.

II. Scott-Strachey semantics

<Denotational semantics Appendix to be added when differentiation chain-rule algorithm is finished>

Data Analysis

<Denotational semantics of the program and verifying that the program runs as hypothesized. Appendix to images of working website and Denotational Semantics conclusion.>

Conclusion

<How effective was the program and all its components.>

Projections

<Projections for future updates on the software and applications on the denotational semantics.>

Acknowledgements

<Final Acknowledgements when finished.>

References

Abadi, M., & Plotkin, G. D. (2020). A simple differentiable programming language. Proceedings of the ACM on Programming Languages, 4(POPL), 1-28. doi:10.1145/3371106

Arora, S., & Barak, B. (2016). Computational complexity A Modern Approach. New York, New York: Cambridge University Press.

Bluman, G., & Dridi, R. (2012). New solutions for ordinary differential equations. Journal of Symbolic Computation, 47(1), 76-88. doi:10.1016/j.jsc.2011.08.018

Chen, L. (Director). (2011). *Differential Equations and exp (At) | MIT 18.06SC Linear Algebra, Fall 2011* [Video file]. Retrieved from https://www.youtube.com/watch?v=dZfdKXxhnTM

Dong, W. (n.d.). Second order estimates for complex Hessian equations with gradient terms on both sides. *Journal of Differential Equations, Volume 271*. doi:https://doi.org/10.1016/j.jde.2020.08.030

Duan, B., Hsieh, C., Yu, C., Yua, J., & Huang, J. (n.d.). A survey on HHL algorithm: From theory to application in quantum machine learning. *Physics Letters A, 384*(24). doi:https://doi.org/10.1016/j.physleta.2020.126595

Edalat, A. (1997). Domains for computation in mathematics, physics and exact real arithmetic. *Bulletin of Symbolic Logic*, *3*(4), 401-452.

GitLab Unfiltered. (2020, March 5). Array programming: A short J tutorial. Retrieved from https://www.youtube.com/watch?v=gPS6qEkZ-nw

Hale, J. K., Lunel, S. M. V., Verduyn, L. S., & Lunel, S. M. V. (1993). *Introduction to functional differential equations* (Vol. 99). Springer Science & Business Media.

Hwang, C., & Cheng, Y. C. (2006). A numerical algorithm for stability testing of fractional delay systems. *Automatica*, *42*(5), 825-831.

Jamieson, M., & Brown, N. (2020). High level programming abstractions for leveraging hierarchical memories with micro-core architectures. *Journal of Parallel and Distributed Computing, 138*, 128-138. doi:10.1016/j.jpdc.2019.11.011

Jha, R., Samuel, A., Pawar, A., & Kiruthika, M. (2013). A Domain-Specific Language for Discrete Mathematics. International Journal of Computer Applications, 70(15), 6-19. doi:10.5120/12036-7257

Meijer, E. (2016). *The Lost Art of Denotational Semantics,* [Video file]. Retrieved from https://www.youtube.com/watch?v=pQyH0p-XJzE

Meijer, E., Hughes, J. (Ed.), Fokkinga, M. M., & Paterson, R. (1991). *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire.* 124-144. https://doi.org/10.1007/3540543961_7

O'Connor, & Robertson. (1996, February). Calculus history. Retrieved from https://mathshistory.st-andrews.ac.uk/HistTopics/The_rise_of_calculus/

Poulson, B. (2019, June 6). R Programming Tutorial - Learn the Basics of Statistical Computing. Retrieved from https://www.youtube.com/watch?v=_V8eKsto3Ug

Valero-Lara, P., Catalán, S., Martorell, X., Usui, T., & Labarta, J. (2020). SLASs: A fully automatic auto-tuned linear algebra library based on OpenMP extensions implemented in OmpSs (LASs Library). *Journal of Parallel and Distributed Computing, 138*, 153-171. doi:10.1016/j.jpdc.2019.12.002

Vartanian, A. (2018, April 26). It doesn't have to be Turing complete to be useful – Increment: Programming Languages. Retrieved from https://increment.com/programming-languages/turing-incomplete-advantages/

Wells, Charles. (2012). Category Theory for Computing Science. Theory and Applications of Categories. 22.

Winther, G. (Director). (2020). *An Oscillating System (more ODEs!)* [Video file]. Retrieved from https://www.youtube.com/watch?v=NMkGNzUjTT4

Xi, H. (1998). *Dependent Types in Practical Programming* (Unpublished master's thesis). Carnegie Mellon University.

Yang, X. S. (2014). *Introduction to computational mathematics*. World Scientific Publishing Company.

Appendix

Source Code A

types.ts - Types

```
enum Instr_Type {
  op, name, ref, ls, obj, prop, num, str, expr, // Basic
  if, for, while, local, var, fun, anon // Structures
}

enum Op {
  "==", "!=", "<=", ">=", "<", ">", // Comparison
  "+", "~", "-", "*", "/", // Arithmetic
  "^", "%%", "%", "'", // Extra math
  "@", // List indexing
  "&", "|", "!", // Boolean
  "??", "?" // Special interaction
}

type Block = Instrs[];

type Instrs = Instr[];

type Instr =
  {type: Instr_Type.op, data: Op} |
  {type: Instr_Type.name, data: string} |
  {type: Instr_Type.ref, data: string} |
  {type: Instr_Type.ls, items: Instrs} |
  {type: Instr_Type.obj, pairs: {key: string, value: Instrs}[]} |
  {type: Instr_Type.prop, data: string} |
  {type: Instr_Type.num, data: number} |
  {type: Instr_Type.str, data: string} |
  {type: Instr_Type.expr, data: Expr} |
  {type: Instr_Type.if, branches: {cond: Instrs, body: Block |
null}[]} |
  {type: Instr_Type.for, var: string, iter: Instrs, body: Block} |
  {type: Instr_Type.while, cond: Instrs, body: Block} |
  {type: Instr_Type.local, var: string, def: Instrs, deriv_n: number}
|
  {type: Instr_Type.var, var: string, def: Instrs, deriv_n: number} |
```

```
34.   {type: Instr_Type.fun, fun: string, args: string[], body: Block} |
35.   {type: Instr_Type.anon, args: string[], body: Block};
36.
37. enum Expr_Top_Type {single, eq}
38.
39. enum Expr_Type {
40.   parens, call, const, main_vars, vars, num
41. }
42.
43. enum Term_Op {"+", "-"}
44.
45. enum Prod_Op {"*", "/"}
46.
47. enum Main_Var {x, y}
48.
49. type Expr =
50.   {type: Expr_Top_Type.single, expr: Terms[]} |
51.   {type: Expr_Top_Type.eq, left: Terms[], right: Terms[]};
52. type Terms = {op: Term_Op, prod: Prods}[];
53. type Prods = {op: Prod_Op, prod: Exps}[];
54. type Exps = Final[];
55.
56. type Final =
57.   {type: Expr_Type.parens, data: Terms[]} |
58.   {type: Expr_Type.call, name: string, args: Terms[][]} |
59.   {type: Expr_Type.const, data: number} |
60.   {type: Expr_Type.main_vars, data: Main_Var} |
61.   {type: Expr_Type.vars, data: string} |
62.   {type: Expr_Type.num, data: number};
63.
64. declare global {
65.   interface Window {
66.     vars: {[name: string]: any};
67.     funs: {[name: string]: (st: any[], out: any[]) => void};
68.     res_hist: any[];
69.   }
70. }
71.
72. export {Instr_Type, Op, Expr_Top_Type, Expr_Type, Term_Op, Prod_Op,
Main_Var, Block, Instrs, Instr};
```

Source Code B

parser.pegjs - Grammar

```
  1. Block = _ instr_chunks:(Instrs _ (";" _ Instrs _)*)
  2.   {return [instr_chunks[0], ...instr_chunks[2].map(instr =>
instr[2])]}
  3.
  4. Instrs = _ instrs:(Instr _)*
  5.   {return instrs.map(instr => instr[0]).reverse()}
  6.
  7. Instr
  8.   = Keywd
  9.   / Op
 10.   / Name
 11.   / Ref
 12.   / Ls
 13.   / Obj
 14.   / Prop
 15.   / Num
 16.   / Str
 17.   / Super_Str
 18.   / Expr
 19.
 20. Keywd = If / For / While / Local / Var / Fun / Anon
 21.
 22. If = "if" _ cond:Instrs _ ":" _ if_branch:Block _
elif_branches:("elif" _ Instrs _ ":" _ Block)* _ else_branch:("else" _
Block)? "end"?
 23.   {return {type: types.Instr_Type.if, branches: [
 24.     {cond, body: if_branch},
 25.     ...elif_branches.map(branch => ({cond: branch[2], body:
branch[6]})),
 26.     ...(else_branch ? [{cond: null, body: else_branch[2]}] : [])
 27.   ]}}
 28.
 29. For = "for" _ var_:Name _ iter:Instrs _ ":" _ body:Block "end"?
 30.   {return {type: types.Instr_Type.for, var: var_.data, iter, body}}
 31.
 32. While = "while" _ cond:Instrs _ ":" _ body:Block "end"?
 33.   {return {type: types.Instr_Type.while, cond, body}}
 34.
 35. Local = var_:Name deriv:"'"* _ ":=" _ def:Instrs
 36.   {return {type: types.Instr_Type.local, var: var_.data, def,
deriv_n: deriv.length}}
```

```
 37.
 38. Var = var_:Name deriv:"'"* _ "=" _ def:Instrs
 39.   {return {type: types.Instr_Type.var, var: var_.data, def, deriv_n:
deriv.length}}
 40.
 41. Fun = "fun" _ fun:Name _ args:(Name _)* ":" _ body:Block _ "end"?
 42.   {return {type: types.Instr_Type.fun, fun: fun.data, args:
args.map(arg => arg[0].data), body}}
 43.
 44. Anon = "anon" _ args:(Name _)* ":" _ body:Block _ "end"?
 45.   {return {type: types.Instr_Type.anon, args: args.map(arg =>
arg[0].data), body}}
 46.
 47. Op
 48.   = (
 49.     "==" / "!=" / "<=" / ">=" / "<" / ">" /
 50.     "+" / "~" / "-" / "*" / "/" /
 51.     "^" / "%%" / "%" / "'" /
 52.     "@" /
 53.     "&" / "|" / "!" /
 54.     "??" / "?")
 55.     {return {type: types.Instr_Type.op, data: types.Op[text()]}}
 56.
 57. Name = ! ("if"/"else"/"elif"/"end"/"for"/"while"/"var"/"fun"/"anon")
[A-Za-z_][A-Za-z0-9_]*
 58.   {return {type: types.Instr_Type.name, data: text()}}
 59.
 60. Ref = "`" ref:Name
 61.   {return {type: types.Instr_Type.ref, data: ref.data}}
 62.
 63. Ls = "[" instrs:Instrs "]"?
 64.   {return {type: types.Instr_Type.ls, items: instrs}}
 65.
 66. Obj = "#[" _ pairs:(":" Name Instrs)* "]"? {return {
 67.   type: types.Instr_Type.obj,
 68.   pairs: pairs.map(
 69.     pair => {return {key: pair[1].data, value: pair[2]}}
 70.   )
 71. }}
 72.
 73. Prop = "." prop:Name
 74.   {return {type: types.Instr_Type.prop, data: prop.data}}
 75.
 76. Num = [0-9]+ ("." [0-9]+)?
 77.   {return {type: types.Instr_Type.num, data: parseFloat(text())}}
 78.
```

```
 79. Str = '"' ([^"\\] / ("\\" .))* '"' {return {
 80.   type: types.Instr_Type.str,
 81.   data: text().slice(1, -1) // .replace(/\\n/g,
"\n").replace(/\\(.)/g, "$1")
 82. }}
 83.
 84. Super_Str = "\\" .* {return {
 85.   type: types.Instr_Type.str,
 86.   data: text().slice(1).replace(/\\/g, "\\\\").replace(/"/g, '\\"')
 87. }}
 88.
 89. _ = ([ \t\n\r] / Cmmnt)*
 90.
 91. Cmmnt = "#(" [^)]* ")"
 92.
 93. Expr = "{" _ eq:Eq _ "}"
 94.   {return {type: types.Instr_Type.expr, data: eq}}
 95.
 96. Eq = left:Terms right:(_ "=" _ Terms)? _
 97.   {return !right
 98.     ? {type: types.Expr_Top_Type.single, single: left}
 99.     : {type: types.Expr_Top_Type.eq, left, right}}
100.
101. Terms = prod:Prods prods:(_ [+\-] _ Prods)* _
102.   {return [
103.     {op: types.Term_Op["+"], prod},
104.     ...prods.map(([, op, , prod]) => {
105.       op: op == "+" ? types.Term_Op["+"] : types.Term_Op["-"],
106.       prod
107.     })
108.   ]}
109.
110. Prods = exp:Exps exps:(_ [*/]? _ Exps)* _
111.   {return [
112.     {op: types.Prod_Op["*"], exp},
113.     ...exps.map(([, op, , exp]) => {
114.       op: op == "*" ? types.Prod_Op["*"] : types.Prod_Op["/"],
115.       exp
116.     })
117.   ]}
118.
119. Exps = final:Final finals:(_ "^" _ Final)* _
120.   {return [
121.     final, ...finals.map(([, , , final]) => final)
122.   ]}
123.
```

```
Final
  = "~"? (
    "(" _ Terms _ ")" /
    Math_Call /
    Math_Const /
    Main_Vars /
    Vars /
    Num)

Main_Vars = "x" / "y"

Vars = [a-wz]

Math_Call
  = Math_Fun_1 _ "(" _ Terms _ ")"
  / Math_Fun_2 _ "(" _ Terms _ "," _ Terms _ ")"
  / Math_Fun_3 _ "(" _ Terms _ "," _ Terms _ "," _ Terms _ ")"

Math_Fun_1 = (
  "abs" / "sqrt" / "cbrt" / "ln" /
  "cosh" / "sinh" / "tanh" / "coth" / "sech" / "csch" /
  "cos" / "sin" / "tan" / "cot" / "sec" / "csc" /
  "arccosh" / "arcsinh" / "arctanh" / "arccoth" / "arcsech" /
"arccsch" /
  "arccos" / "arcsin" / "arctan" / "arccot" / "arcsec" / "arccsc")
Math_Fun_2 = "root" / "log"
Math_Fun_3 = "sum" / "prod"

Math_Const = "pi" / "π" / "tau" / "τ" / "e"
```

Source Code C

compiler.ts - Compiler

```
import types = require("./types");

let prelude =
`with(main.builtins) {

let st = [];
let out = [];
`;

let postlude = `

out = [...out, ...st];
out;
}`;

let builtins = [
  "print", "true", "false", "call", "len", "map", "filter", "reduce",
"times", "range", "srange", "enum",
  "pi", "e", "tau",
  "sin", "cos", "tan", "cot", "sec", "csc"];

function compile(ast: types.Block): string {
  return prelude + compile_rec(ast, 0, [], [], []) + postlude;
}

function compile_rec(
    ast: types.Block, indent: number,
    locals: string[], vars: string[], funs: string[]
  ): string {
  let compiled: string[][] = ast.map(instrs => instrs.map(instr => {
    switch(instr.type) {
      case types.Instr_Type.op:
        return `__ops["${types.Op[instr.data]}"](st, out);`;
      case types.Instr_Type.name:
        if(instr.data == "debug") return "debugger;";

        if(locals.includes(instr.data))
          return `st.push(${instr.data});`;
        else if(vars.includes(instr.data) || instr.data in
window.vars)
          return `st.push(window.vars.${instr.data});`;
```

```
 40.         else if(funs.includes(instr.data) || instr.data in
window.funs)
 41.           return `window.funs.${instr.data}(st, out);`;
 42.         else if(builtins.includes(instr.data))
 43.           return `__${instr.data}(st, out);`;
 44.         else
 45.           throw `${instr.data} is not a variable or function.`;
 46.       case types.Instr_Type.ref:
 47.         if(locals.includes(instr.data))
 48.           return `st.push(${instr.data});`;
 49.         else if(vars.includes(instr.data) || instr.data in
window.vars)
 50.           return `st.push(window.vars.${instr.data});`;
 51.         else if(funs.includes(instr.data) || instr.data in
window.funs)
 52.           return `st.push(window.funs.${instr.data});`;
 53.         else if(builtins.includes(instr.data))
 54.           return `st.push(__${instr.data});`;
 55.         else
 56.           throw `${instr.data} is not a variable or function.`;
 57.       case types.Instr_Type.ls:
 58.         return `st.push((() => {\n${tabs(indent + 1)}let st =
[];\n${compile_rec([instr.items], indent + 1, locals, vars,
funs)}\n${tabs(indent + 1)}return st.reverse();\n${tabs(indent)}})())`;
 59.       case types.Instr_Type.obj:
 60.         return `st.push({\n${instr.pairs.map(({key, value}) =>
`${tabs(indent + 1)}${key}: (() => {\n${compile_rec([value], indent + 2,
locals, vars, funs)}\n${tabs(indent + 2)}return st.pop();\n${tabs(indent +
1)}})()`).join(",\n")}\n${tabs(indent)}});`;
 61.       case types.Instr_Type.prop:
 62.         return `st.push(st.pop().${instr.data});`;
 63.       case types.Instr_Type.num:
 64.         return `st.push(${instr.data});`;
 65.       case types.Instr_Type.str:
 66.         return `st.push("${instr.data}");`;
 67.       case types.Instr_Type.if:
 68.         return instr.branches.map(branch =>
 69.           `${
 70.             branch.cond
 71.               ? `if((() => {\n${compile_rec([branch.cond], indent +
2, locals, vars, funs)}\n${tabs(indent + 2)}return
st.pop();\n${tabs(indent + 1)}})()) `
 72.               : ""
 73.           }{\n${compile_rec(branch.body, indent + 1, locals, vars,
funs)}\n${tabs(indent)}}`
 74.         ).join(" else ");
```

```
 75.       case types.Instr_Type.for:
 76.         return `for(const ${instr.var} of (() =>
{\n${compile_rec([instr.iter], indent + 2, locals, vars,
funs)}\n${tabs(indent + 2)}return st.pop();\n${tabs(indent + 1)}})())
{\n${compile_rec(instr.body, indent + 1, [...locals, instr.var], vars,
funs)}\n${tabs(indent)}}`;
 77.       case types.Instr_Type.while:
 78.         return `while((() => {\n${compile_rec([instr.cond], indent +
2, locals, vars, funs)}\n${tabs(indent + 2)}return
st.pop();\n${tabs(indent + 1)}})()) {\n${compile_rec(instr.body, indent +
1, locals, vars, funs)}\n${tabs(indent)}}`;
 79.
 80.       // Add deriv.
 81.       case types.Instr_Type.local:
 82.         if(!locals.includes(instr.var))
 83.           locals = [...locals, instr.var];
 84.         return `let ${instr.var} = (() =>
{\n${compile_rec([instr.def], indent + 1, locals, vars,
funs)}\n${tabs(indent + 1)}return st.pop();\n${tabs(indent)}})();`;
 85.       case types.Instr_Type.var:
 86.         if(locals.includes(instr.var))
 87.           return `${instr.var} = (() => {\n${compile_rec([instr.def],
indent + 1, locals, vars, funs)}\n${tabs(indent + 1)}return
st.pop();\n${tabs(indent)}})();`;
 88.         else {
 89.           if(!vars.includes(instr.var))
 90.             vars = [...vars, instr.var];
 91.           return `window.vars.${instr.var} = (() =>
{\n${compile_rec([instr.def], indent + 1, locals, vars,
funs)}\n${tabs(indent + 1)}return st.pop();\n${tabs(indent)}})();`;
 92.         }
 93.       case types.Instr_Type.fun:
 94.         funs = [...funs, instr.fun];
 95.         return `window.funs.${instr.fun} = (st, out) =>
{\n${instr.args.map(arg => `${tabs(indent + 1)}let ${arg} =
st.pop();\n`).join("")}${compile_rec(instr.body, indent + 1, [...locals,
...instr.args], vars, funs)}\n${tabs(indent)}}`
 96.       case types.Instr_Type.anon:
 97.         return `st.push((st, out) => {\n${instr.args.map(arg =>
`${tabs(indent + 1)}let ${arg} =
st.pop();\n`).join("")}${compile_rec(instr.body, indent + 1, [...locals,
...instr.args], vars, funs)}\n${tabs(indent)}});`
 98.       default:
 99.         return "NOT OP;";
100.     }
101.   }));
```

```
102.   return compiled.map(comp_instrs =>
103.     comp_instrs.map(instr =>
104.       tabs(indent) + instr
105.     ).join("\n")
106.   ).join("\n");
107. }
108.
109. function tabs(indent: number): string {
110.   return "\t".repeat(indent);
111. }
112.
113. export {compile};
```