Progsbase: A Timeless, Translatable and Understandable Programming System

Martin Fagereng Johansen
2018-07-01
martinfjohansen@inductive.no
Inductive AS, Oslo, Norway

What is progsbase?

Progsbase is a programming system built from those building blocks that have proven themselves extensively. In other words, it is a programming language and related tooling, and it is built from the parts and constructs that have proven themselves for a long time and in a wide variety of productive automations.

What is interesting about this programming language, is that it has a number of features no other programming language has (as far the the author knows):

1. Timelessness: Programs written in progsbase have extensive longevity and are highly resistant to software rot.

2. Translatable: Programs written in progsbase can be translated between programming languages losslessly. Progsbase therefore also is in other programming languages, so it can be written using the tooling of existing programming languages.

3. Understandable: Progsbase is easy to understand and read: It includes those building blocks that have proven themselves extensively when it comes to intelligibility.

Before going into these features and why they are useful, let’s look at what the programming language contains.

Which Constructs

The following is an overview of the main parts of progsbase. Progsbase programs are limited to computations. This is, in fact, a rather small limitation: Computer programs normally consist of over 80% computations, and entire classes of computer programs are 100% computations. Examples of computations: Evaluating Mathematical formulas, sound, image and signal processing, audio and video codecs, data format conversions, graphics rendering, cryptography, compression, static program analysis, compiling, statistics, data search and numerical methods.

The highest level building blocks of progsbase programs are:

- Files and folders
- Dependencies
- A namespace
- A version

The next level is the contents of the files:

- Functions (procedures, subroutines, methods)
- Data Structures

The next level is the contents of functions:

- Lines of code (statements)
- Parameters
- Variable declarations
- A return type

The lines of code (statements) can be:

- Assignments
- Function calls
- Ifs
- Loops
- A Return
- Deallocations

Assignments can have expressions on either side of the assignment-sign. Expressions can be:

- Arithmetic expression
- Relational expression
- Function call
- Structural expression
- Allocation expression

Arithmetic expressions include the operators addition, subtraction, multiplication and division. It also includes exponentiation and modulus. Arithmetic expressions also include the unary operators absolute, logarithm (base 10) and natural logarithm, square root and exponentiation (e^x), sine, cosine, tangent, arc sine, arc cosine and arc tangent.

Arithmetic expressions operate on decimal floating point numbers with 15 digits precision and a two digit signed exponent.

Relational expressions include equals, not equals, more, less, more than or equal and less than or equal. The values are Boolean `yes` and `no` (usually called `true` or `false`.)

Structural expressions include the length operator for arrays and accessors for structures and arrays.

Allocation expressions include the allocation of arrays of a certain length and structures.

How does it work?

The core of progsabase is a data structure that holds a valid progsbase program. To see its role, let's look at a normal development process using progsbase:

A programmer programs a program in one of the supported languages, for example, Java version 5+. The programmer uses the Java compiler to check that the program is valid Java version 5+. If it is valid, the programmer passes the program to the progsbase analyzer to see if it is valid progsbase. All valid progsbase programs are valid Java programs, but not the other way around.

If the program is valid progsbase, the progsbase tooling reads the program into the progsbase data structure, mentioned above, which is the core of progsbase. Once in this format, the progsbase tooling can write the program out to one of the supported languages, which include all programming languages with a few exceptions, but these exceptions account for less that 1% of written programs.

The progsbase tooling consists of

1. compiler and analyzer – loads and analyzes a program to see if it is valid progsbase.

2. converter – converts a valid progsbase program to another programming language

3. program database – a database of progsbase programs that can be browsed and and downloaded in all supported programming languages.

Why does it work?

Let's look at why progsbase provide the three features listed at the beginning:

1. Timelessness

Progsbase provides longevity because it consists of the building blocks that have proven themselves through long time spans and in a wide variety of productive automations.

Much of the arithmetic and trigonometry was known to the ancients. The decimal number system was introduced into the western world in the 12th century by Leonardo of Pisa. Mathematical functions were studied in the 17th century, especially during the development of calculus. Logarithms and exponents were introduced in the 17th century by John Napier. Scientific notation was well known in the 17th century.

Imperative programming with sequences of operations, decimal arithmetic, variables, arrays and conditional jumps was introduced in the middle of the 19th century by Charles Babbage. Charles Babbage introduced the idea of computers being suited to automate a wide variety of mechanical tasks. This includes evaluating all mathematical formulas and doing operations on data encoded as numbers, including text, images, sound and geometrical data.

Those ideas listed above have been core building blocks of computer programs since the first computer was put into productive use in the 1940s and through to the time of writing this, the 2010s. They were established and well known at the start of this period and have continued to prove themselves throughout this period.

In programming research, a lot of ideas have been suggested, implemented and used. The problems of legacy software, software rot and lack of software reuse proves that some of these ideas were expedient or premature. Programs in old and established fields such as banking and insurance often become outdated and unusable in a few decades. As computers, operating systems and libraries are updated, software gradually stops working, sometimes even within a year after writing. Software reuse, even between suppliers and customers within the same field, let alone between companies in the same field, is severely underutilized. This is even though it was well known when the first computer was put into productive use, that their mechanical processes could be built from an existing set of well known building blocks.

2. Translatable

Progsbase programs are readily translatable into all programming languages (with only a few insignificant exceptions). The reason is that its building blocks have proven themselves for a long time and in a wide variety of fields. Programming language designers therefore must include them in new programming languages for them to be regarded as serious and useful by those utilizing computers in production.

This is the reason progsbase programs can be translated to most programming languages.

Programming languages are often famous for what differentiates them from other programming languages, and advocates of a language – or even a new version of a language – are often eager to use those differentiating features extensively. This gives programs written in those languages a

particular style that is unique for that language and that is not readily translatable to languages without those features.

Programs written in progsbase will have the same basic style no matter which language it is written in.

As the building blocks of progsbase are a part of other programming languages, those building blocks can be created with those languages. Progsbase is included in other languages as a restricted version of that language.

3. Understandable

A program is understandable if 1) its building blocks are easy to understand, 2) the building blocks fit together well and 3) it is easy to see how an automation is broken down into the building blocks.

Building blocks of a language might themselves be difficult to understand, but the complexity is amplified when the building blocks are combined. If the building blocks are combined in a single line of code, for example; such as in a formula or a declaration, if one has to remember a large amount of rather arbitrary detail, then the building blocks do not combine well. Examples include operator precedence in formulas in C and C++, declarations with a lot of generics, libraries with intrusive reflection, binary floating point formulas, undefined or implementation defined behavior, mixing code in different languages and implicitly changing types in a mixed expression.

The progsbase building blocks themselves are very well known. When combining them, there is basically no arbitrary detail required to be remembered to understand how they combine. It has also been known for a long time, since the works of Charles Babbage, that productive computations break down into many of the progsbase building blocks. (This, of course, in no way rules out the discovery of new kinds of building blocks that improves this process.)

What can it be used for?

The following are general use cases for progsbase.

You want your software to last a long time.

If you develop software that will be in productive use for a long time, from many years to many decades and beyond, progsbase ensures that the programs will stay functioning. A progsbase program is built from building blocks that will not get outdated or disappear from computers, the workforce skillset or human knowledge.

Examples of industries are banking, finance, telecom, insurance, manufacture and logistics.

You want software available in two languages when transitioning from one language to another.

If you are rebuilding some software, you can use progsbase to make code that is available in both the programming language of the software being replaced and the programming language of the new software.

You want to port existing code to gain the benefits of progsbase.

If you have existing code that is threatened to become legacy and where the average developer age is increasing, you can port the computational parts of the programs to progsbase. This way you also gain the possibility of translating the code to a newer language.

You want to teach or learn the most important and fundamental parts of programming.

If you are teaching or learning programming, learning progsbase first will give a basic understanding of programming that crosses language boundaries and is useful knowledge no matter which programming language you code in your job.

If you know progsbase you can participate in programming computations for any project.

You want your software to be widely reused internally.

If you have software in your company that has applicability throughout the organization, you may want to code those parts in progsbase so it can be used all the way from back-end systems, front-end systems to spreadsheets and databases without reimplementation. Core company software can be developed and quality checked once for the entire company to use.

You want your software to be widely reused externally.

If you are a company selling software services via web APIs or software used via APIs, you may want to bundle software that customers and integrators can use when using your services or software through your API.

You are writing software in many languages and want to avoid repetition and foster reuse.

Several software development categories use several languages today. Examples are:

- Mobile app development in Java, C/C++ or C# as typical backend languages, and Objective C, Swift and JavaScript as front-end language.

- Enterprise software written in Java, C# and PL/SQL as typical backend languages, and JavaScript as front-end language.

You want to create a reference implementation.

If you are a researcher or algorithm developer, you might want to create a reference implementation along with a white paper. This way, library writers and users can either directly use the implementation or use it as the basis for creating an optimized version.

This reference implementation can be published on the progsbase public repository for easy inclusion in other software.

You want to have the option to execute a program on different computers.

If you are developing an Internet or network application, you may want to postpone the decision as to where some code is executed. Client side execution may speed up an application or take load off a server.

You want to create development tools that can be used in any programming language.

If you are a software development tool vendor, you may develop tooling such as refactoring or analysis that is applicable for any program written in progsbase.

You want to build programs on a solid, established and well-understood foundation.

Existing programming languages often contain building blocks that are esoteric, experimental and very difficult to understand. You don’t want business critical code to unnecessarily use such constructs, when existing, established and well understood constructs are sufficient.

Regarding recruitment of programmers, if your code is built with well-understood, simple yet powerful building blocks, programmer recruitment for those parts is easier and reduces the dependence on programmers knowledgeable of esoteric and experimental programming techniques.

You want quality assurance involved at function level.

If your business domain has functionality available as functions in a program, you can involve testers on the function level. They can test functions without developer tooling and test software directly in their browser, without knowing source control systems or how to set up and maintain complex build-environments.

You want to outsource a programming task.

If your software project has programming tasks that are functions that do computation, you can specify and outsource those parts. The required knowledge needed to participate in programming is the progsbase building-blocks that are well-established programming techniques.

You want to sell software that can be used in any other programming language.

If you are selling software libraries and want your software to be available in any programming language, you can develop your software using progsbase and create packages for each language for your customers to use.

You want your customers to reuse your software in their own solutions

It is very common for a customer of both hardware and software to want to reuse some of the software. A very good example of this is a factory that wants to reuse some of the software controlling the hardware in their own solutions. Other examples are companies that use software and build their own solutions that also use the software. In this case, it is much better to use the software directly than through some slow, cumbersome and limited interface, which is very common. Most often, the software is unavailable on any usable interface.

Technical challenges

The following is an overview of the main technical challenges that had to be solved for progsbase to work.

Decimal arithmetic in a binary arithmetic environment

Most computers today run binary floating point arithmetic natively. How can a programming language requiring decimal arithmetic run predictably in an environment with binary arithmetic? The solution lies in working with numbers the right way.

The number of digits of precision in common floating point arithmetic is not arbitrary. Through experience, about 15 digits of precision is usually sufficient. If more is needed, one can use an arbitrary precision library (something which will be readily available for progsbase.)

When working with limited precision, one can use the Calculus of Errors to calculate the number of digits of precision an output will have given the precision of the input and the operations performed. If a result is rounded to the calculated precision at the end of a series of calculations, the result will be the same as long as the precision is equal to or better than the original assumptions.

It is required by progsbase programs to operate the same when using 15 digits of decimal precision, when using IEEE binary 64 (also called double) or if the calculations were done with an arbitrary precision library with precision higher than 15 digits.

Order of evaluation

Different languages have different orders of evaluation. Some languages even allow for different orders of evaluation given that rules such as a defined precedence is followed.

A valid progsbase program is required to always get the same result as if the program was evaluated with a certain set of complete rules: The result of a formula must be the same as if the expression was evaluated left-first, bottom-first. In other words, the next operator or function to be evaluated is the left-most of the nodes in the syntax tree of the formula where an operator can be evaluated next.

Operators or functions with differing output

Certain operators, such as divide, have different outputs in different languages. This is only for special cases, such as division by zero. In progsbase, such situations must throw an invalid operator exception. Exceptions in progsbase cannot be caught, and the uppermost progsbase function has in effect failed. A progsbase program should be programmed so that an exception of this sort never occurs. Then, the differing outputs in such situations do not matter.

The same goes for out of bounds access of arrays, the use of uninitialized variables, all kinds of invalid inputs to operators and functions, overflow of arithmetic operators and failure to allocate or deallocate memory.

What is not included.

The following is a brief rationale for not including certain building blocks in progsbase.

Interrupts and CPU ports

Interrupts and CPU ports (in and out instructions) are the main fundamental mechanisms for extending the capabilities of a computer beyond computations. How a programming language should be built to best incorporate these is still an open question.

Binary floating point arithmetic

There is agreement on binary floating point arithmetic, named IEEE 754, but it is awkward together with decimal numbers. Numbers are written and read in decimal form. This causes too few to understand the finer detail of something supposed to be so central. The benefits of binary floating point numbers over decimal floating point numbers does not warrant the added complexity of their default use.

Integer arithmetic

Integer arithmetic has a number of issues: What should the bounds be? What happens if integers of different sizes are in the same expression? What happens if signed and unsigned integers are in the same expression? What happens if an integer calculation overflows? How does integer division work for negative numbers? How does modulus work for negative numbers? Should there be unsigned integers at all? Should mathematical operators be modular by default? What if integer

arithmetic is mixed with ordinary arithmetic? Should integer arithmetic be default? Language designers answer these questions differently, and how it should work has not yet solidified.

null, nil, undefined

If an array or a structure or even a number or a boolean does not have a value, a language may set it to null, nil or undefined. What happens when such a value is used? Language designers disagree; this has not yet solidified.

-0

Can zero be negative? Is it equal to positive zero? Does such a value even make sense? Language designers disagree.

NaN, -NaN

When an operation does not make sense, such as the square root of -1 or the logarithm of a negative number, some language designers have included a special value, the Not a Number value, or `NaN`. Which cases should cause a `NaN`? Does `NaN` equal `NaN`? What is `−NaN`? Should it even be included? Language designers disagree.

∞, -∞

Potential infinity is an important and valid concept. Actual infinity, however, is invalid. Some language designers have decided to include absolute infinity as a value when floating point overflows or a number is divided by zero. What is `∞ − ∞`? What is `∞ − 3`? ∞ to the power of `0`? `∞/∞`? Etc. This does not make sense and should not be included in a programming language.

break, continue, early return, goto

Three related ideas are break and continue in loops and early returns in functions. Even though they are quite popular and mostly included in programming languages, there are strong arguments against their use and they are not always included. A particularly strong argument against them is that code on the same level will not always all execute or execute in order. They are variants of the goto-statement and are deviations from the idea of structured programming. Progsbase fully embraces structured programming, therefore break, continue, early return and goto are not included.

Exceptions

Exceptions, understood as a special mechanism for returning something from a function, are controversial and the discussions have not solidified. There are strong arguments against exceptions. A particularly strong argument is that errors should be checked and acted upon, an idea progsbase endorses. Exceptions are also a deviation of structured programming, being a mechanism for transferring control from one function to a distant catch.

Object-oriented programming

Object-oriented programming was initially introduced for programming discrete event simulations. It does have merits and adherents, but is infamous for being overused and stretched beyond its scope. What object-oriented programming consists of and what should be included as a part of it is still an open question. Along with the question about its scope, the discussion has not yet solidified.

Reflection

Reflection certainly has some very elegant uses, but is so prone to misuse and so infamous for being misused that it, at this point, only can be regarded as an interesting experiment.

Templates or Generics

Templates or generics also have some interesting use cases, but are also infamous for complexity spiraling out of control: Difficulty of understanding, unfocused error messages and huge disagreements means this has not solidified yet.

About the Inventor and Author

Martin Fagereng Johansen has a PhD in Computer science from the University of Oslo and works as an inventor, businessman, developer and computer scientist at Inductive AS, Norway.