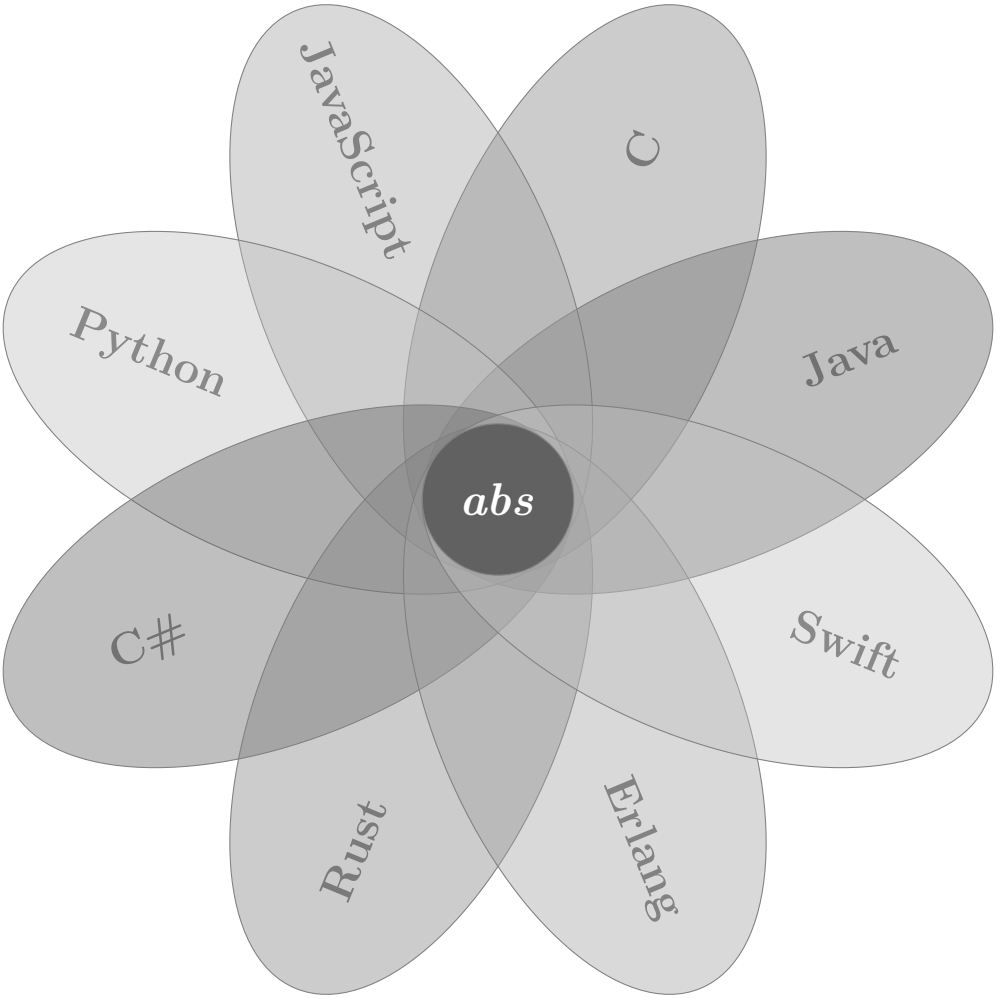
ABStract Language Specifcation

Author
John Dunlap

Acknowledgements
Daniel Pratt - Randall Fairman - Ryan Connolly

DRAFT Revision 33 - 2023.05.16



Contents

1 Introduction

The goal of the *ABStract Programming Language*, hereafter referred to as *abs*, is to cleanly share logic between wholly incompatible programming languages without introducing any dependencies beyond the logic itself.

To achieve this goal, *abs* has been designed as a transpiled programming language which enables unidirectional source-to-source transformation from *abs* into other programming languages, hereafter referred to as targets. The initial list of supported targets is: Java, C#, JavaScript, Erlang, Rust, C, Python, and Swift. By embracing dissimilar targets from day one, *abs* benefits from a broader range of perspectives and ensures that its design is naturally bounded by commonalities between targets. This is why the *abs* logo is a Venn diagram.

Commonalities between targets include, but are not necessarily restricted to: Variables, data types, functions, recursion, loops, if/else, arithmetic/logical operations, function call stack, and memory allocation.

abs supports three different kinds of functions: Native, pure, and impure. Native functions are *abs* method contracts which define a name, return type, and argument list but which do not have a body. These methods are assumed to be implemented in the target language and available at runtime. Pure functions are functions which cannot call native functions either directly or indirectly. This is enforced by raising a compiler error if a pure function calls a function which is not also a pure function. Impure functions are functions which call functions of all types. Sharing state outside of method contracts is strictly forbidden and, consequently, *abs* does not support global variables, singletons, static variables, or similar mechanisms. *abs* does not guarantee the size of its numeric data types. At first glance, this may seem like inhibits portability but it is the position of this document that the opposite is true because this allows *abs* to support different hardware platforms which have dissimilar CPU registers. This is similar to the approach taken by C which is, arguably, the most portable language ever created.

However, there are also aspects of the *abs* design with significantly differ from that of its supported targets. For example, *abs* introduces support for multiple entry points, removes support IO of any kind, and prevents runtime errors at compile time. Multiple entry points are necessary because native code, written in a target language, must be able to directly call *abs* functions. Support for IO is not provided because *abs* is dependent on the target to provide this functionality through native methods.

1.1 Why Another Language?

The world is full of weird and wonderful programming languages which aim to solve problems in every way you can imagine and likely a few ways you can't. At the time this was written, there were 694 programming languages listed on Wikipedia[1]. Every programming language has its pros and cons but, at this point, it's very unusual to encounter a problem which cannot be solved with at least one of them. The enescapable question which must be answered by every language designer is this: Why does a world with so many programming languages need another one? While it is certainly true that there are very few tasks, if any, which cannot be solved with an existing programming language, it is important to remember that this has always been the case. For example, there are no solvable problems which cannot be solved with assembly language and yet more than 600 programming languages have been created since assembly language was created. Just because it's possible to solve a problem with a particular language does not necessarily mean that it's the best tool for the job, even if it's the only available tool. Programming languages are created for a variety of reasons including expressiveness, domain specific needs, performance, parallelism, simplification, compatibility, etc. The need driving the creation of *abs* is broad compatibility. Certainly, you could write the same function multiple times, once for each of your deployment environments but it would be more efficient and preferable to write that function once and compile it separately for each of your deployment environments.

[1] `https://en.wikipedia.org/wiki/List_of_programming_languages`.

1.2 Design Descisions

1.2.1 Memory Management

Within the initial list of targets, there are four distinct approaches to memory management:

Approach	Targets
Garbage collected heap memory	Java, C#, Erlang, Swift, Python, JavaScript
Manually managed heap memory	C
Smart pointer managed heap memory	Rust
Automatically managed stack memory	All

Automatic memory management is a fundamental design goal of *abs*. Unfortunately, not all targets support it. C defers heap management to the programmer, Rust requires the explicit use of smart pointers, and other targets rely on garbage collectors to manage the heap. This inconsistency forces *abs* to include memory management in its design such that it it is possible to support targets which do not have a garbage collector.

It is not feasible to implement a portable garbage collector for targets which do not already have one. The simplest solution to this problem is to add compile time constraints which which have no affect in targets other than C while allowing *abs* to implement a stack managed heap when targeting C.

1.2.2 Types

1.2.3 Error Handling

1.2.4 Null References

To null or not to null? That is the question. In 1965, Tony Hoare[2] changed computer science forever by introducing null references in *ALGOL W*. In 2009, having observed his invention for 44 years, he had this to say:

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years." - Tony Hoare

Did Mr. Hoare fail to solve the problem of unsafe references? Null references are clearly unsafe.

TODO: This doesn't read well and comes across as being very clumsily written Let's take a step back and look at what Mr. Hoare's goal was when he invented null references. He said, *"My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler."* If unsafe references existed prior to null references then it seems reasonable to infer that reference errors were the underlying problem he was attempting to solve.

[2] `https://en.wikipedia.org/wiki/Tony_Hoare`.

It seems as if the problems Mr. Hoare attributes to null references, in the quote above, are the very similar to the problems he originally set out to solve.

Is the answer simply that Mr. Hoare failed to solve the problem of unsafe references? Not exactly. Reference errors existed both before and after the invention of null references so, in that sense, you could say that he failed. However, in fairness to Mr. Hoare, he succeeded in reducing the number of invalid references from many to one. This is no small feat and represents a major step forward in the field of computer computer science because it replaces complicated address range checks with a single equality comparison. This equality comparison allows higher level languages to safely report reference errors, gracefully fail, and even recover from reference errors without crashing the entire program. Unfortunately, this has led to the unfair perception that null is the root of the problem and it isn't. It doesn't matter if the address of an invalid reference is zero or some other value. If a program attempts to dereference an invalid reference, there will be an error. Period. If a compiler allows a program to dereference an invalid reference be it a null reference or not then the fault lies with the compiler. When these errors occur, null isn't an antagonist; It is a messenger.

Should *abs* support null? From a pragmatic perspective, the answer must be: Yes. Due to *abs* being a transpiled language, many of its compilation targets will support null and refusing to support null would complicate the interaction between native and non-native methods. Null references exist; That is a fact. Whether or not they *should* exist isn't relevant. What happens if *abs* does not support null and a native method returns null or passes null into an *abs* entry point? It's inconcievable that this would never happen in practice and, consequently, it must be accounted for in the design of the language. There are other ways to approach the problem but it is the position of this document that supporting null references is, by far, the simplest.

1.3 Why have cross platform languages failed in the past?

TODO: Define what is meant by "failed". What would success look like for a cross platform language? Did Java really fail? It's one of the most popular languages ever created. Most modern programming languages are capable of implementing any other programming language. However, just because language A can be used to implement language B doesn't necessarily mean that the resulting implementation of language B will perform well enough to be useful outside an academic setting.

The Java Programming Language[3] boasts *"Write once. Run anywhere"*. While this was mostly true when Java was originally created, it has become less and less true over time. The problem is that Java programs can only be run on platforms which are supported by the JVM[4] and the number of available platforms, in combination with the increasing complexity of the JVM, has made it very difficult for Java to support all of them. To make matters worse, even if a platform is supported by the JVM, the platform may prohibit the installation of the JVM, may provide an API or SDK which is not written in Java, or the JVM may be too resource intensive to be viable on the target platform. At this point, no language runtime has achieved the level of dominance necessary to be available on every platform, it seems unlikely that any languge runtime will achieve this level of dominance, and even if a language runtime were to achieve this level of dominance, it is unlikely that it could maintain this level of dominance indefinitely. Given the apparent impossibility of this dilemma, *abs* has chosen to abandon the concept of a language runtime entirely. Every platform which currently exists, or which is likely to exist, will support at least one programming language so there is no need to provide a language runtime in the first place. It is, therefore, the goal of the *abs* compiler to leverage whatever language, API, or SDK is available on the target platform. This goal has lead to the ideal: *"Write once. Compile anywhere."* **TODO: Expand on the idea that Java's downfall was too many runtime dependencies.**

[3] `https://en.wikipedia.org/wiki/Java_(programming_language)`
[4] `https://en.wikipedia.org/wiki/Java_virtual_machine`

1.4 Why aren't cross platform languages the norm?

TODOCross platform languages are hard. Different languages have different capabilities, strengths, and weaknesses. Too many features; Every supported feature complicates adding support for new platforms. Designing a language which is functionally divergent from the platforms on which it needs to run. Too much complexity. Failure to constrain language semantics to the overlapping capabilities of the platforms which must be supported. Similar or identical problems are solved in different ways on different platforms. Attempting to solve the same problem in a portable way may

1.5 Problems

The following are problems which are not, either in whole or in part, solved by existing languages.

1.5.1 Multiple Platforms

One of the principle challenges in modern software development is supporting the ever increasing number of platforms and smart devices. Many of these devices, despite supporting similar use cases, technically have very little in common with one another. These devices have different form factors, different operating systems, different hardware, different language support, different libraries, and different distrubution mechanisms. Consequently, supporting multiple devices is more expensive than supporting a single device and every additional supported device increases costs and extends deadlines.

1.5.2 Limited Reusability

The word "abstract" can mean different things. In the context of this document, it is a programming language. In conversation, it can mean something is difficult to understand or it can refer to an idea. In the context of an object oriented programming language, it refers to classes which are intended to address the overlap between two or more similar problems. Code which is useful in multiple problems is included and code which is only useful for a single problem is excluded. This allows an unknown number of similar problems to be partially solved while fully solving a single problem. This is a valuable capability which simplifies the creation of reusable software. However, it does have limitations. Although many languages support the concept of an abstract class, in one form or another, an abstract class in languge A is, generally speaking, wholly incompatible with language B. Consequently, an abstract class isn't truly abstract because it must be written separately in each language which requires it. This inability to share abstract classes between languages and platforms is one of the reasons that supporting two or more platforms more expensive than supporting a single platform. **TODO: Limited reusability is a problem which applies to more than just abstract classes. For example, a function written in language A cannot be reused in language B**

1.5.3 Premature Obsolescence

Software applications degrade over time, for a variety of reasons, and, while some degrade more quickly than others, in the absence of extraordinary intervention, they all, eventually, reach the end of their useful service life. This degredation is, unfortunately, unavoidable because it is caused by the very technical implementation details which make the application functional. Over time, technical implementation details become increasingly intertwined with business logic and, past a certain point, it becomes impractical to separate them. Many software applications continue like this until it becomes necessary to retire their underlying technology which, in turn, forces the organization to retire the application as well. This is unfortunate because organizations invest decades of time, money, knowledge, and effort into building software applications which still have value at the point that they must be retired. Replacing these software applications is difficult, time consuming, and expensive.

1.5.4 Conflict Of Interest

The interests of corporations, which provide free software development tooling, are not always aligned with the interests of those who use their tooling. If a corporation spends a large amount of money developing

tools which they subsequently give away for free, there's very rarely an altruistic explanation. Common explanations include, but are not limited to, a corporation wants you to use their hardware, their operating system, or their platform and, consequently, makes it as easy as possible for you to do so. Whatever their reason happens to be, the use of their tools is usually a mutually beneficial relationship. However, it should be self evident that a corporation which offers free tools for building applications on their own platform cannot be relied upon also provide free tools for building applications on their competitors platforms. They may do it, under some circumstances, particularly if they are not the market leader, but they may discontinue support for their competitor's platforms, at any time and without warning, if it becomes too expensive or if they achieve market dominance. It is in the best interest of a corporation for application X to exclusively support their platform. By contrast, it is in the best interest of the consumer to for application X to support as many platforms as possible because this gives them the freedom to choose which platform works best for them.

1.5.5 Monolingual applications are expensive

TODO: Fill this out

1.6 Solution

It is the position of this document that there can be no solution to the problems outlined above except the creation of a new programming language.

1.7 Philosophy

The design of the *abs* language is guided by the principles found below. Some of these principles may be familiar to the discerning eye. This is because principles deemed to have value, in the context of *abs*, have been adopted from other projects.

- **Do one thing and do it well:** Don't try to be all things to all people. The goal is portable logic. Nothing more. Nothing less.
- **Compilation errors are better than runtime errors:** Errors encountered by a developer are better than errors encountered by a user. Minimize runtime errors by maximizing compilation errors.
- **Portability is more valuable than performance:** Without portability, there is no reason for *abs* to exist. Performance matters but portability should never be sacrificed in pursuit of it.
- **Pragmatism in all things:** Ideas are only useful insofar as they help us solve the problem at hand. If a good idea detracts from our ability to solve a problem then that idea must inform our solution rather than embody it.
- **Simple is better than complex:** Portability increases as complexity decreases. *The best part is no part* - Elon Musk
- **Explicit is better than implicit:** Make intentions and behaviors clear, rather than relying on implicit or obscure mechanisms.
- **Readability counts:** *Any fool can write code that a computer can understand. Good programmers write code that humans can understand* - Martin Fowler

1.8 Security Considerations

1.8.1 Native Code Injection

Native code injection attacks are very similar to SQL injection attacks. When accepting user created code, string concatenaction is extremely dangerous because the user could attempt to escape the compiler's sandbox by embedding native code in a string constant, which would be ignored by *abs* but not by the compiler of

the native language, with the goal of accessing native resources which would not otherwise be available. One possible solution to this problem would be to store all string constants in a text file which gets automatically loaded at runtime by the code which is emitted by the compiler.

1.8.2 Denial Of Service

If abs code is accepted from users then, in the absence of some kind of timeout, code which loops or iterates infinitely could be used to execute a denial of service attack.

1.8.3 Types

The principal hurdle of building a portable type system is ensuring consistency and compatibility across multiple target languages. Transpilation involves converting code from one language to another, each with its own type system and semantic rules. Mapping types accurately between the source language and target languages can be complex, as different languages may have different type representations, behavior, and limitations. Maintaining semantic integrity and ensuring that the type system behaves consistently across targets requires careful consideration and meticulous design. Another challenge is dealing with language-specific features and constructs that may not have direct equivalents in all target languages. Balancing the expressiveness of the type system while accommodating the constraints and limitations of the target languages is crucial. Furthermore, error reporting and debugging can be challenging when dealing with type-related issues in a transpiled language, as the error messages need to be meaningful and provide insights that align with the target language's conventions and error reporting mechanisms. Overall, creating a portable type system in a transpiled language necessitates a deep understanding of both the source and target languages, as well as careful planning and implementation to address the inherent challenges of cross-language compatibility and semantic consistency.

1.8.4 Boolean truth

TODO: Describe the pros and cons of "truthiness" vs actual boolean types.

1.8.5 Error handling

Errors are inevitable and, consequently, it is necessary to both detect and react to them.
How do we implement exceptions in a language which doesn't natively support them(Like C)?
This must be taken into account in the architecture of every programming language. The are many ways to approach this problem but two popular approaches are returning error codes from functions in the case of an error or throwing an exception and rolling up the call stack until the exception is handled. In this section we will be exploring the pros and cons of the two approaches.

- Method calls can fail for multiple reasons.
- Calling code may need to know what those reasons are.
- Failing early is not necessarily better.
- In some circumstances, it may be preferable for execution to continue after an error.
- In some circumstances, it may be preferable for execution to halt after an error.

1.9 Versioning

TODO: It must possible to lexicographically compare versions with string operators to determine which version is newer. Older versions should always be lexicographically lesser than newer versions.

2 Specification

2.1 Definition Of Terms

2.1.1 Base Language

Any language which is supported as an *abs* compilation target.

2.1.2 Pure function

A pure function is any function which does not either directly or indirectly invoke a native method. This results in functions which do not have external dependencies.

2.1.3 Impure function

An impure function is not declared as native or pure and is able to call all types of functions.

2.1.4 Native function

Native functions do not have a body because they are assumed to have been implemented in the target language and to be available at runtime.

2.2 Literals

2.2.1 Number Literal

Number literals are comprised of the characters 0 through 9 and may not contain a decimal point. If a number literal is terminated with the letter L, case sensitive, then it is of type Long. Ohterwise, it is of type Integer. Number literals may use underscores as separators to make large numbers easier to read. Number literals which begin with 0x are hexidecimal. In hexadeciam numbers, underscores are not permitted until after the 0x. Leading zeros are ignored in all bases. Number literals may be prefixed with a minus sign or a plus sign to indicate the sign of the literal.

2.2.2 Decimal Literal

TODO: Decimal literals must contain a decimal point.

2.2.3 String Literal

String literals may contain any UTF-8 character. Double quote characters must be escaped with a backslash to avoid prematurely terminating the String literal. String literals may also contain interpolated variables. An interpolated variable must begin with a dollar sign. Dollar signs which are a part of the string literal must be escaped with a backslash character. If necessary, after the dollar sign, the name of the interpolated variable may be enclosed in curly braces to prevent characters from the string literal being included in the interpolated variable name.

2.2.4 Boolean Literal

Legal values are **true** and **false**.

2.2.5 Null Literal

null is the only legal value.

2.2.6 Paradigm Literal

Legal values are **oop**, **procedural**, and **functional**.

2.3 Reserved Words

It is illegal for any namespace, function, or variable name to coincide with a reserved word. Reserved words are case insenstive.

Word	Description
coalesce	Built-in method which accepts two or more parameters and returns its first non-null parameter or null if all parameters evaluate to null. The first parameter must be nullable. The last parameter may be nullable or non-nullable. If there are intermediate parameters, all of them must be nullable. Parameters are evaluated from left to right. All parameters must be the same type. The nullability of the return is inherited from the last parameter. It is illegal for any of the parameters to be a null literal.
nullable	Variable modifier which permits null assignment. In the absence of this modifier, the compiler will emit an error when an assignment operation may result in null assignment. This modifier may also be used on the function return type or function parameters. **TODO: This should be revisited it would be simpler to follow the .NET ? approach to determining nullability**
pure	Namespace modifier which prevents methods within the namespace from interacting with other namespaces which haven't also been declared as pure. This modifier is mutually exclusive with the native modifier. Pure namespaces may not interact with impure namespaces but impure namespaces may interact with pure namespaces. This increases the portability of pure namespaces because the *abs* compiler is able to verify, at compile time, that pure namespaces do not have any native dependencies.
native	Namespace modifier which indicates that methods declared within the namespace must be implemented separately in each target language. Method declarations within a native namespace should be terminated with a semicolon as it is illegal for them to have a body. This modifier is mutually exclusive with the pure modifier. It is illegal for a native method to be declared as returning a value which is non-nullable or a type which, either directly or indirectly, containes one or more properties which are non-nullable because the return value of a native method cannot be known until run time and, therefore, cannot be checked for null at compile time.

entry	Method modifier which declares a method as an entry point into an *abs* program. This is necessary because compile time null checking is not possible if a native method, which isn't written in *abs*, can invoke any *abs* method at any time because the provided arguments might be null at run time even if the method in question expects non-nullable parameters. Consequently, methods declared as an entry point may not directly or indirectly accept parameters which have not been declared as nullable. This forces explicit null checking to be performed prior to invoking functions which do not accept nullable parameters. Additionally, the *abs* compiler may obfuscate functions in its output which have not been declared as an entry point to discourage their use in native functions as doing so would create the possibility of a null pointer exception. This modifier may be used within a pure namespace. It is an error to use this modifier within a native namespace.
final	Variable modifier which indicates that the value of the variable cannot be changed.
in	Operator which evaluates to true if the value on its left side exists in the collection on its right side and evaluates to false otherwise. It is illegal for the value on its left side to be null.
void	Function modifier which indicates that a function does not return a value.
return	Operator which immediately halts execution of the function in which it is invoked. This operator must be invoked without an argument if the function was declared with a void return type and must be invoked with a single argument otherwise. If invoked with a single argument, the argument will be returned to the invoking function. It is an error for the type or nullability of the argument to differ from that of the return type which was declared in the method signature.

synchronized	null	not null	package				
assert	class	enum	use	this	finally		
final	abstract	double	super	transient	public	private	protected
for	do	true	false	while			
break	byte	case	virtual	switch	continue	throws	throw
new	catch	if	implements	try	else	oop	
functional	procedural	set	get	goto	integer	struct	
string	float	boolean	thread	base	default	alias	narrow

2.4 Grammar

2.4.1 Object Oriented

2.4.2 Procedural

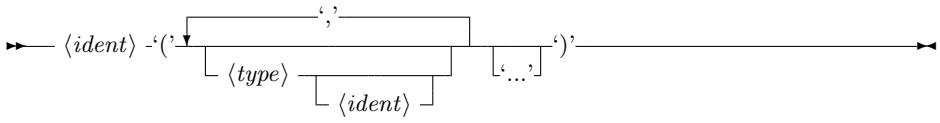
⟨*statement*⟩ ::= ⟨*ident*⟩ '`=`' ⟨*expr*⟩
| '`for`' ⟨*ident*⟩ '`=`' ⟨*expr*⟩ '`to`' ⟨*expr*⟩ '`do`' ⟨*statement*⟩
| '`{`' ⟨*stat-list*⟩ '`}`'
| ⟨*empty*⟩

⟨*stat-list*⟩ ::= ⟨*statement*⟩ '`;`' ⟨*stat-list*⟩ | ⟨*statement*⟩

Increase the two lengths

⟨*statement*⟩ ::= ⟨*ident*⟩ '`=`' ⟨*expr*⟩
| '`for`' ⟨*ident*⟩ '`=`' ⟨*expr*⟩ '`to`' ⟨*expr*⟩ '`do`' ⟨*statement*⟩
| '`{`' ⟨*stat-list*⟩ '`}`'
| ⟨*empty*⟩

⟨*stat-list*⟩ ::= ⟨*statement*⟩ '`;`' ⟨*stat-list*⟩ | ⟨*statement*⟩



1. **allow-functions-outside-class** - In languages which do not natively support OOP or where OOP is optional, it can be useful to declare functions outside class definitions. This feature could also be useful when transpiling other languages to *abs*. This feature is disabled by default but can be enabled by passing an option to the compiler.
2. **allow-variables-outside-class** - In languages which do not natively support OOP or where OOP is optional, it can be useful to declare variables outside class definitions. This feature could also be useful when transpiling other languages to *abs*. This feature is disabled by default but can be enabled by passing an option to the compiler.

2.5 Accessors

abs uses the same syntax to access attributes whether they have accessor methods or not. This allows accessor methods to be added or removed at a later date without breaking compatibility with code outside of the class definition.

2.6 Operators

Operator	Type	Description
+	Math	Numeric addition operator. Used for both integer and float types
+=	Math	Numeric addition-assignment operator. Used for both integer and float types
-	Math	Numeric subtraction operator. Used for both integer and float types
-=	Math	Numeric subtraction-assignment operator. Used for both integer and float types
*	Math	Numeric multipaction operator. Used for both integer and float types
*=	Math	Numeric multipaction-assignment operator. Used for both integer and float types
/	Math	Numeric division operator. Used for both integer and float types
/=	Math	Numeric division-assignment operator. Used for both integer and float types
%	Math	Numeric modulus operator. Used for both integer and float types

%=	Math	Numeric modulus-assignment operator. Used for both integer and float types
++	Math	Numeric increment operator. Used for both integer and float types
–	Math	Numeric decrement operator. Used for both integer and float types
==	Logical	Logical equality operator. This operator will evaluate to true if the expression on both sides evaluate to the same value. Otherwise, this operator will evaluate to false. This is used for integer, float, and string types.
<	Logical	Logical less-than operator. This is used for integer, float, and string types.
>	Logical	Logical greater-than operator. This is used for integer, float, and string types
<=	Logical	Logical less-than-or-equal operator. This is used for integer, float, and string types
>=	Logical	Logical greater-than-or-equal operator. This is used for integer, float, and string typesr
!	Logical	Logical NOT operator. This is a unary operator which evaluates to the opposite of what its expression
!=	Logical	Logical not-equal operator. This is a binary operator which evaluates to the opposite of what its argument evaluates to. This is used for integer, float, and string types
&&	Logical	Logical AND operator
||	Logical	Logical OR operator
in	Logical	Binary operator which evaluates to true if the value on its left exists within the collection on its right and evaluates to false otherwise
and	Bitwise	Bitwise AND operator
and=	Bitwise	Bitwise AND-assignment operator
nand	Bitwise	Bitwise NAND operator
nand=	Bitwise	Bitwise NAND-assignment operator
or	Bitwise	Bitwise OR operator
or=	Bitwise	Bitwise OR-assignment operator
xor	Bitwise	Bitwise XOR operator
xor=	Bitwise	Bitwise XOR-assignment operator
xnor	Bitwise	Bitwise XNOR operator
xnor=	Bitwise	Bitwise XNOR-assignment operator
nor	Bitwise	Bitwise NOR operator
nor=	Bitwise	Bitwise NOR-assignment operator
comp	Bitwise	Bitwise complement operator
comp=	Bitwise	Bitwise complement-assignment operator
lshift	Bitwise	left-shift operator
lshift=	Bitwise	left-shift-assignment operator
rshift	Bitwise	right-shift operator
rshift=	Bitwise	right-shift-assignment operator
?:	Misc	Ternary operator
=	Misc	The assignment operator assigns the computed value on the its right side to the variable on its left side

isa	Misc	This operator evaluates to boolean true if the expression on its left is polymorphically and instance of the expression on its right. Otherwise, it evaluates to boolean false
new	Misc	This operator creates new instances of a class
throw	Misc	This operator throws a new exception
|	Misc	The pipe operator allows the output of one method to be chained into the input of another method, even if those methods do not belong to the same class. The method on the right must only accept a single parameter and the type of that parameter must match the return type of the function on the left.

2.7 Types

2.7.1 Supported

1. Integer(i32)
2. Long(int64)
3. Float(f32) - IEEE Standard 754 Single Precision
4. Double(f64) - IEEE Standard 754 Double Precision
5. Fixed - Fixed point precision
6. number<M> - M is the required digits to the right of the decimal point
7. number<N,M> - N is the number of required digits to the left of the decimal point and M is the required digits to the right of the decimal point
8. String
9. Byte - This is the only data type which is supported for binary operations.
10. Date - Date only. No timestamp.
11. DateTime - Date and timestamp.
12. Boolean - Used for all boolean operations.
13. Array<T>
14. Map<K,V>
15. User defined 2 dimensional data structures(classes).

2.7.2 Mapping

ABS types will be translated to target languages as follows:

	Perl	**Java**	**PHP**	**JS**	**C++**
Integer	scalar	Integer	dynamic	dynamic	int
Long	scalar	Long	dynamic	dynamic	long
Float	scalar	Float	dynamic	dynamic	float
Double	scalar	Double	dynamic	dynamic	double
String	scalar	String	dynamic	dynamic	std::stringstream
Byte	scalar	Byte	dynamic	dynamic	unsigned char
Date	???	Date	dynamic	dynamic	???
DateTime	???	Date	dynamic	dynamic	???
Boolean	scalar	Boolean	dynamic	dynamic	bool
Array	arrayref	ArrayList<T>	array	array	std::vector
Map	hashref	HashMap<K,V>	array	object	std::map

For the time being, the following languages have been omitted from this list:

1. **C** - Because it does not support OOP and because it does not support throw/catch exception semantics.
2. **Elixir** - Because it does not support OOP.

2.7.3 Narrowing

Narrow types like **uint8** will be be masked when necessary to provide "proper" narrow types on platforms which don't support them. Masking is necessary in the following situations:

1. When passing the value to an arbitrary function which is not known to tolerate "trash upper bits".
2. When storing the value to a storage location that does not automatically truncate the value.
3. Before specific operations, namely: shift right, divide, remainder, compare, conversion to a wider or floating point type

Most operations, namely AND, OR, XOR, NOT, shift-left, addition/subtraction/negation, multiplication and conversion to a narrower type, are not affected by "trash upper bits". The trash stays safely contained in those upper bits, which will be ignored in the end.

JavaScript and its floats are a bit trickier, because floats can throw away the bottom bits, which is bad. In fact, just multiplying two 32bit integers is non-trivial in JavaScript.
TODO: Reword this section because it is copied from the internet

2.8 Command Line Interface

The command line interface, hereafter referred to as the *cli*, houses all of the native code necessary for interacting with the *abs* compiler. It is important to differentiate between the *abs cli* and the *abs* compiler because the compiler is written in *abs* and the *cli* is not. This is a direct consequence of the *abs* language specification, which prohibits direct acccess to native functionality. While there is only one *abs* compiler, there could, theoretically, be as many as one *cli* per supported platform. Available *cli* functionality includes, but is not limited to, the ability to compile, refactor, reformat, and query *abs* code.

target directory All files created by the *abs* compiler will be placed in a directory called **target** within the project root directory. This makes it easy to to manually clean up after the compiler.

3 Road Map

3.1 Milestones

- Build a lexer in Java
- Build a parser in Java
- Build an interpreter. This interpreter will be used to execute the initial compilation targets, which will be written in *abs* rather than the host language.
- Compile to a single target
- Compile to multiple targets
- Rewrite the compiler in *abs*
- Self compilation

3.2 Command Line utility: absc

The *absc*(ABS Compiler) utility will be a combination of various native stubs and a C++ version of the abs compiler which can be installed through the package managers of popular linux distributions.

3.3 IDE Plugins

IDE plugins are central to the adoption strategy of the platform as a whole. The compiler itself should be embedded into IDE plugins for all popular IDEs. These would include but not be limited to:

- Jetbrains
- Eclipse
- Netbeans
- Code Blocks
- CodeMirror
- Ace
- VSCode
- Monaco

3.4 Compilation Targets

These are the language targets which will be implemented first.

- Haxe - Indirectly provides other targets including JavaScript and PHP
- Perl - Not provided by Haxe
- Elixir - Not provided by Haxe

Additionally, compilation targets for the following languages are planned for the future.

- PHP
- Javascript
- Java
- C#
- C++
- C
- Rust
- Python
- Perl
- Lua
- XML Schema(This would only export data structures, not behavior)

3.5 Supported

- **Encoding** - *Abs* source files must be UTF-8 encoded and an illegal character exception must be thrown during compilation if an invalid UTF-8 character is encountered. This is critical because the compiler will be compiled to multiple platforms and each of them may operate in a different encoding by default. Therefore, for *abs* source code to be portable from one compiler to another, it must be written in a standard encoding.
- **Native classes** - Native classes function as an interface between *abs* code and the outside world. Libraries and APIs, implemented in the target languguage, can be invoked by *abs* through native classes. However, because the *abs* compiler has no control over, nor knowledge of, what happens in a native class, the return type of methods within a native class cannot contain *notnull* properties. Forcing these properties to be nullable allows the *abs* compiler to verify, for example, that a function is never called on a null value, that null is never passed to a function which requires a not-null parameter, and that null is never assigned to a not-null variable. The *abs* compiler does not support the mixing of native and concrete methods within the same class because, although it is theoretically possible, this would significantly complicate the addition of new compilation targets because the target would have to be able to both generate and parse the target language. It is, therefore, preferable that parsing the target language is not necessary when implementing new compilation targets. Additionally, to protect the compiler's ability to enforce notnull constraints, methods within a native class are not permitted to return native classes.
- **Keyword: entry** - Methods declared with the *entry* keyword will not be obfuscated. Additionally, the class/package name of classes which contain at least one *entry* method will not be obfuscated because they are necessary for invoking both the constructor and static methods. Obfuscated method names will have a randomly generated name in the generated source to inhibit direct invokation. This is necessary because direct invokation could circumvent the compiler's not-null protections. Additionally, methods which havve been declared with the entrypoint keyword cannot accept native classes as arguments. This is necessary to protect the compiler's ability to enforce *notnull* constraints.
- **Compilation targets are written in ABS itself.** This was, initially, a way to maximize code re-use during the development of the bootstrap compiler, which was written in Java. Implementing the compilation targets in ABS and running them in an interpreter during compilation, minimized the amount of code which would later have to be rewriteen when the bootstrap compiler was ported from Java to ABS.
- Abstract methods.
- Dot invokation on variables. Eligible functions will accept the data type of the variable as their fist parameter. Eligible functions will include sibling methods from the same class and static methods.
- Ternary operator. This can be expanded to if/else statements in languages which do not support the ternary operator.
- A package can implement another package by saying "implements PACKAGE;". Doing this will cause the compiler to issue an error if the current package does not implement all of the abstract methods in the referenced package.
- If, elseif, and else conditionals.
- Bitwise operators.
- For loops.
- While loops.
- For-Each loops.
- Switch-case statements. This can be expanded to if/else statements in languages like Perl which do not native support for switches.

- Keyword "not-null". This keyword may precede variable declarations, function arguments, and function return types. A compiler error shall be thrown if it is possible for a null value to exist at that junction. The compiler must be able to detect if it is possible for a nullable variable to contain null if a nullable variable's value is assigned to a non-nullable variable or if the value of a nullable value has been
- Immutable variables. Once initialized, a compiler error will be issued if an attempt is made to modify the variable. Variables prefixed with the keyword "immutable" are implicitly non-nullable and a compiler error will be issued if a variable is prefixed with both "immutable" and "not-null".
- Namespaces. There is a direct relationship between the file path of a source file and its namespace. It is not possible to declare a name space which is different than the file path of the file.
- Compile time null detection. A compiler error shall be thrown if a branch can be found in which a method is invoked on a null object reference.
  NOTE: This compile time null checking must be thread safe.
- Divide by zero is a checked exception.

3.6 Future

Automatic line wrapping in comments comment editing in most source editors is very annoying because it comments don't automatically line wrap when they get too long. This forces the developer to do it manually which results in a cascade of changes to ensure that each line is roughly the same lengh.

Concurrent loop Provide a special type of loop which processes a set of data in parallel instead of synchronously following the set order.

Possible concurrency model Make each module/package single threaded and all interactions between modules is done via message passing. This avoids the performance overhead of deep deep copies of function arguments which are necessary to avoid locks. **How would this work with modules which implement collections? Surely, we don't want all operations on collections to happen within the same thread?**

Automatic constructor invokation If a single parameter is passed to a method which does not accept that parameter type, look for other single parameter methods of the same name which expect a parameter which has a single parameter constructor of the type which was provided. If found, automatically invoke the constructor and pass the newly created instance to the method. If multiple options are found, issue a parse error.

AST: dot Support for printing the abstract syntax tree in the dot graphing language. This allows the abstract syntax tree to be graphically rendered.

AST: json Support for printing the abstract syntax tree as a json data structure. This allows other tools and programming languages to access the abstract syntax tree.

AST: xml Support for printing the abstract syntax tree as a xml data structure. This allows other tools and programming languages to access the abstract syntax tree.

AST: yaml Support for printing the abstract syntax tree as a yaml data structure. This allows other tools and programming languages to access the abstract syntax tree.

Interpreter Support for running *abs* in interpreted mode built directly into the compiler. For example, if the *abs* compiler is embedded in your application and users are typing *abs* code into your application, they will likely want to run it once they have typed it in. Depending on the compilation target, it may be possible to load the output code directly into the target language. However, that will not always be possible or desired. Having an interpreter built into the compiler would offer another avenue for executing user provided code.

Tree Shaking The *abs* compiler should only compile those functions which are directly or indirectly invoked from entrypoint methods. This will create a minimal output irrespective of how much *abs* exists within the project.

Transpile Haxe to ABS As there is already a lot of cross-platform code written in Haxe, it would make it easier for people to evaluate *abs* if existing Haxe code could be converted to *abs*. The easiest way to do this would be to add support for *abs* as a Haxe compilation target and contribute that code back to the Haxe project.

inotify Support for the Linux inotify Kernel API can be used to automatically run the compiler against ABS source files which have changed. This will accelerate development in target languages, like PHP, which support live reload. Ex: The developer modifies and saves an ABS source file, an inotify event triggers the compiler, an updated PHP file is written to disk, the developer reloads their webpage by hitting F5, and they can see their changes.

Obfuscation The command line flag –obfuscate will cause ABS to emit obfuscated code in the target language.

ABS Standard Library The standard library should be comprised of concrete methods which can be fully implemented in ABS and abstract methods which must be implemented by the user. An example of an abstract API might be functions for querying a database. The standard library should contain all of the abstract methods necessary to give ABS the functionality of a general purpose programming language. This would make it possible for these methods to be implemented for each compilation target as a separate open source project to simplify using ABS in other projects.

Code as data The ABS compiler should allow users to query their code as if it were a database. The usefulness of text-based search and replace tools are very limited because they are unabled to differentiate between comments, source code, and other meta data which is only availabe at compile time. A fundamental role of the ABS compiler is parsing an ABS source file into a data stream of tokens. Consequently, a heavyweight IDE(Integrated Development Environment) should not be necessary to achieve operations which can easily be implemented in the compiler itself by leveraging fundamental compiler functionality. This will make the language easier to use in the absence of an IDE and simplify the addition of ABS support to existing IDE's. All search operations should be reductive in nature(search results should be the intersection of provided search parameters) and should not be mutually exclusive. Supported operations should include:

- Find usages of a variable or method by name
- Find usages of a variable or method by type signature
- Find usages of a variable or method by type signature
- Rename a variable or method
- Wildcard search and replace

Source Code Formatting Conventions The ABS compiler should be exposed to the user as a command line tool which allows the white space within an ABS source file to be checked against rules provided to the command line tool and, optionally, to be reformatted to conform to to those rules. This will greatly simplify the enforcement of and compliance with project and or organizational code formatting standards.

License Enforcement The compiler should, optionally, be able to check for the absence of a license in a source file and, optionally, be able to add or update the license text within a file.

Copyright Enforcement The compiler should, optionally, be able to check for the absence of a copyright in a source file and, optionally, be able to add or update the copyright within a source file. This is most useful for updating dates within a copyright statement.

Pre-Processor Because reflection is not supported at runtime a fully type aware pre-processor would be useful for generating or expanding the source code during compilation.

Self Compilation Fundamentally, the ABS compiler is a text-to-text transform. As text processing is a fundamental operation in computer science and supported in every modern programming language, the ABS compiler should be re-written in ABS so that it can be compiled to different runtimes just like any other ABS program. The ABS compiler should be able to generate identical output regardless of whether it is compiled to Java, PHP, JavaScript, or C. This feature will primarily be used to embed the ABS compiler into IDE plugins which are written in a variety of different languages depending on the IDE.

Note: While the original ABS compiler will become obsolete, a native ABS compiler will always be necessary. As such, the last official act of the original ABS compiler will be to generate its replacement from ABS source.

DSL support TBD tokens should allow blocks of source to be deferred to a user provided parser, written in ABS, which will allow the compiler to verify the syntactic correctness of Domain-Specific-Languages(DSL). User provided parsers could, for example, be created for SQL dialects to provide compile time syntax checking of queries and IDE autocompletion. User provided parsers could also be made aware of the user's database schema if ABS metadata was generated from the user's database schema. Multi-line strings could potentially be implemented with a DSL as could embedded JSON or embedded CSV.

3.7 Unsupported

- Reflection at runtime is not supported by ABS because it would allow the user to circumvent the type system. This is especially true of a transpiled language, like ABS, because it has no control over the target runtime environment. If reflection at runtime is necessary, it must be provided to ABS code through an abstract method which is natively implemented in the target language.
- Access to the file system is not permitted by the ABS compiler for security reasons because the compiler assumes that ABS code may be from an untrusted source. If file system access is required, it must be provided to ABS code through an abstract method which is natively implemented in the target language.
- Access to the system shell is not permitted by the ABS compiler for security reasons because the compiler assumes that ABS code may be from an untrusted source. If shell access is required, it must be procided to ABS code through an abstract method which is natively implemented in the target language.
- Access to STDIO is not permitted because it may not exist in the target language and because it may be used for something in the target language.
- Access to STDERR is not permitted because it may not exist in the target language and because it may be used for something in the target language.
- Directly exiting an ABS program is not considered necessary or desirable as it allows user code to bypass cleanup routines which would otherwise have executed if every function in the stack had been permitted to return normally.

- Explicit namespace declaration. The namespace or package of an *abs* source file is inferred from its file system location so declaring one explicitly would be redundant.
- Typeless key/value structures. Allowing this kind of generic data structure would make IDE auto-completion nearly impossible because it would be impossible to infer the data type of its contents at compile time.
- All logical expressions must evaluate to a boolean value because this minimizes unexpected behaviors which can result from "truthy" logical expressions.
- Undefined variables are not supported because having multiple sentinel values is redundant and checking for both undefined and null makes code unnecessarily verbose and complicated for little or no benefit. Consequently, uninitialized variables will automatically be initialized to null.

4 To Do List

- Once the abs compiler is functional, go back through this document and rewrord everything to describe **exactly** what the abs compiler/language does and nothing more.

5 License

Current thought: Release the Java source code for the bootstrap compiler and the *abs* source code for the main compiler under the GPL**(TODO: Which version?)**. Compile the compiler into each supported target language and release the compiled source code under the BSD license. Copyright would be retained by the author(s) of the source code.