Å Official Documentation

The official specification for Å

Contents

Abstract

This document is the official document for the Å programming language. It shows how a program is parsed from source code into a binary format and then how the binary is executed. Thus this document will also contain the semantics of the language and an explanation for the default implementation of the language. Besides explaining the semantics and syntax, this document will also explain how the source code is parsed. Additionally, how this parse result is operated on to convert it into a valid piece of Å code.

Anything written or discussed within this document is only safely assumed to be applicable for Å version 1.0. Therefore, code and other semantics may be outdated or even too new (or features not implemented yet). This document should, therefore, only act as documentation or as guidance for the previously specified version of Å.

Using the Virtual Machine

The virtual machine is first and foremost a command-line tool. It can - on Windows 10 - be run as an executable, wherein it will expect some arguments be given to it. The arguments given to the application will then be used to determine what the application will actually do.

The machine is capable of compiling source code into Å byte code. This byte code can then be executed by the machine directly after compilation (Such that it acts as an interpreter of the source code) or be executed later. Besides compiling and executing Å code, the machine is also capable of generating a readable text version of the generated byte code. Letting the user read the result of the compilation. To add to this, the compiler can also unparse the given Abstract Syntax tree that was created during the parsing of the source code. This can help in giving the programmer a bigger insight into the interpretation of the code.

Most arguments given to the machine can be executed in sequence (order of arguments is ignored). To which the use of the machine is simplified to simply giving it commands.

Argument	Description	Available	Version
`-ga`	Generate formatted bytecode from compilation	R|D	1.0
`-lc`	Log the compile time in seconds	R|D	1.0
`-le`	Log the execute time in seconds	R|D	1.0
`-silent`	Ignores program output	R|D	1.0
`-pause`	Will need user input to close	R|D	1.0
`-trace`	Trace virtual machine operations	D	1.0
`-test_regressive`	Run regression tests	D	1.0
`-unparse`	Unparse source after parsing	R|D	1.0
`-c "input file"`	Compile the input file	R|D	1.0
`-oae "output file"`	Output operation to specified file	R|D	1.0

Some command arguments (Marked with a 'D') may only work if the VM (The `C++` code) was compiled with the `_DEBUG` flag enabled. These operations may severely hinder the performance of the language and the VM and have, therefore, been disabled for public builds. In the case of `-test_regressive`, it cannot be guaranteed the end user has the files for running the regression tests. Thus to ensure no error reports of this, or unfortunate crashes, this feature is only available for the developers of Å.

As an example, if we wanted to compile and execute a file named "test.aa", with the code:

```
class Program {
  Program() {
    println("Hello World from class");
  }
};

int main() {
  Program p = new Program();
  0;
};

main();
```

It could be done with the command arguments:

```
call AAVM.exe -ga -unparse -c "test.aa" -oae "test.aab" -le -lc -pause
```

Which would give a formatted binary file ("test.aab") and an unparsed file alongside the generated binary. Additionally, we'd see the printed message in the console window, as well as the number 0.

Parsing

The parsing of Å source code is split into several different stages. The first stage is the lexical analysis. Here each character of the source code is examined and, in the given context, tokenised such that the parse tree has some workable data. The parse tree is constructed by first converting the tokens into parse nodes. This constructs a flattened tree (a simple vector), that, using the syntax rules of the language are expanded. Then the more detailed parsing takes places. Lastly, the whole parse tree is converted into an abstract syntax tree (AST). Afterwards, the parsing result (the AST) is handed off to the compiler.

The first parsing error - syntax error - recorded will stop the parsing of the code and immediately report the syntax error to the console. A short explanation of what failed (for example what was expected and what was found) is reported alongside a position (line and column) in code. If possible, a specific syntax error code is also given, hopefully, making it easier to correct or lookup solutions.

Lexical Analysis

The lexical process is rather simple and is using a two-pass system. In the first pass, most if not all words and special characters are tokenized. Each found token saves the result in a data structure containing the associated text and the position in code. The token types are shown in the table below[1]:

#	Token	Description	Regular Expression
0	invalid	Invalid Token	
1	whitespace	Whitespace character	$\epsilon \vert \Lambda$
2	identifier	Identifier - name or word	$(\_\vert[a-Z])^{+}([0-9]\vert[a-Z]\vert\_)^{*}$
3	keyword	Word with specific meaning	$identifier \in \mathsf{keywordlist}$
4	separator	Separating character	$;\vert,\vert(\vert)\vert\{\vert\}\vert[\vert]$
5	OP	Operator character	$+\vert-\vert*\vert/\vert\%\vert=\vert!\vert?\vert<\vert>\vert\&\vert\vert\vert\$\vert\#\vert\wedge$
6	intlit	Integer Literal	$[1-9]^{+}[0-9]^{*}$
7	floatlit	Floating point literal (32-bit)	$[0-9]^{+}.[0-9]^{+}\mathsf{f}$
8	doublelit	Double literal (64-bit floating point)	$[0-9]^{+}.[0-9]^{+}$
9	charlit	Character literal	$'([0-9]\vert a-Z]\vert\_\vert\epsilon)'$
10	stringlit	String literal	$"([0-9]\vert a-Z]\vert\_\vert\epsilon)^{*}"$
11	stringOP	String operator	
12	comment	Comment token	$//([0-9]\vert[a-Z]\vert\_\vert\epsilon\vert\Lambda)^{*}\backslash\mathsf{n}$
13	accessor	Access token	$.\vert:$
14	quote	Quotation token	$'\vert"$

Some of these tokens may depend on their context (For example, the keyword-identifier relation). The given regular expressions can only be used to tokenise a piece of Å source code. It should not be taken as a complete set of regular expressions defining the language.

After the initial lexical run, the second pass starts. Here tokens that can be merged are merged. For instance, two operator tokens making up a valid operator if merged. A full list of these operators can be found in appendix 6. In this pass, string literals and char literals are also found, taking all the contents between two quote tokens and putting them into a string or char literal, depending on the length of the contents. This is part of why whitespace is tokenised, as that's how string literals keep track of the whitespace characters.

All registered identifiers are matched against keywordlist. If a match is found with any word from the above, the identifier token is "upgraded" to be a keyword token. Some keywords may be contextualised, meaning their given meaning may only apply if in the correct context. This is not decided by the lexical analysis but either by the lexical to parse tree conversion.

[1] The ϵ character represents the whitespace character ' ' and Λ represents the empty word - no character.

Parse tree (PT)

Creating the parse tree is done using a three pass algorithm. The first pass takes the result of the lexical analysis as input. It then converts the result into an array (`std::vector`) with a tree node representation of the result. This will create a flattened tree - no parents or children set. It's also here the parser determines if a token of type `OP` is a binary or a unary operation.

The lexical token type `OP` is by default designated as a unary operator, unless any literal type, end parenthesis, end index (`]`), or an identifier is preceding it. It will then be treated as a binary operator. *This may change when `++` and `--` is implemented.*

Abstract Syntax Tree (AST)

...

Simplifying the AST

...

Static Analysis

...

Type declarations

...

Function Matching

...

Building Inheritance Trees

...

Control path verification

...

Type System

...

Static Type Environment

...

Dynamic Type Environment

...

Typechecking

...

Internals of the Virtual Machine

...

Executing bytecode

...

The operand and call stacks

...

The Ref Value Environment

...

The Garbage Collector

Structure of an Å-binary file

...

Compiler

...

Compiling the AST

...

Verifying Å Bytecode

...

Appendix

The following pages are the appendices of this specification and contain full overview over the grammar, semantics, type rules, compile errors, runtime errors, and instruction list.

Appendix 1: Complete Overview of Grammar

The following is a BNF grammar - extended with '*' (Kleene Star)[2] and '+' (Kleene Plus). The ϵ character symbolises whitespace.

$$\begin{aligned}
Statement \quad &::= Expr\texttt{;} \\
&\mid\ Id\ \texttt{=}\ Expr\texttt{;}
\end{aligned}$$

$$\begin{aligned}
Expr \quad &:= \texttt{(}Expr\texttt{)} \\
&\mid\ Expr\ BinaryOp\ Expr \\
&\mid\ PreUnaryOp\ Expr \\
&\mid\ Expr\ PostUnaryOp \\
&\mid\ IntLit \\
&\mid\ FloatLit \\
&\mid\ DoubleLit \\
&\mid\ BoolLit \\
&\mid\ NullLit \\
&\mid\ StringLit \\
&\mid\ CharLit \\
&\mid\ Id
\end{aligned}$$

$$\begin{aligned}
Decl \quad &::= \texttt{var}\ Id = Expr\texttt{;} \\
&\mid\ \texttt{const var}\ Id = Expr\texttt{;} \\
&\mid\ TypeId\ Id = Expr\texttt{;} \\
&\mid\ \texttt{class}\ Id\ \{\ Decl^*\ \} \\
&\mid\ \texttt{class}\ Id\texttt{()}\ \{\ Decl^*\ \} \\
&\mid\ \texttt{class}\ Id\texttt{(}Param\texttt{)}\ \{\ Decl^*\ \} \\
&\mid\ TypeID\ Id\texttt{()}\ \{\ Statement^*\ \} \\
&\mid\ \texttt{void}\ Id\texttt{()}\ \{\ Statement^*\ \} \\
&\mid\ TypeID\ Id\texttt{(}Param\texttt{)}\ \{\ Statement^*\ \} \\
&\mid\ \texttt{void}\ Id\texttt{(}Param\texttt{)}\ \{\ Statement^*\ \}
\end{aligned}$$

$$Param \quad ::= TypeID\ Id \mid Param\texttt{,}\ Param$$

$$\begin{aligned}
BinaryOp \quad &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid < \mid > \mid <= \mid >= \mid \texttt{==} \mid \texttt{!=} \mid \texttt{+=} \mid \texttt{-=} \mid \Vert \\
&\mid\ \&\& \mid \vert \mid \& \mid << \mid >>
\end{aligned}$$

$$PreUnaryOp \quad ::= \texttt{-} \mid \texttt{!} \mid \texttt{\#}$$

$$PostUnaryOp \quad ::= \texttt{++} \mid \texttt{--}$$

$$Digit \quad ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$IntLit \quad ::= Digit^+$$

$$FloatLit \quad ::= Digit^+\texttt{.}Digit^+\texttt{f}$$

$$DoubleLit \quad ::= Digit^+\texttt{.}Digit^+$$

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

BoolLit ::= `true` | `false`

NullLit ::= `null`

Letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N
| O | P | Q | R | S | T | U | V | W | X | Y | Z | Å
| a | b | c | d | e | f | g | h | i | j | k | l | m | n
| o | p | q | r | s | t | u | v | w | x | y | z | å

Id ::= $Letter^{+} Digit^{*} Letter^{*}$ | _*Id*

Character ::= *Letter* | *Digit* | ϵ | _ | `+` | `-` | `*` | `/` | `%` | `#` | < | > | `=` | `|` | `&` | `$` | `€` | `@` | `£` | `!`
| `?` | `:` | `;` | `,` | `.` | `^` | `¨` | `~` | `(` | `)` | { | } | `[` | `]` | § | `\n` | `\t`

StringLit ::= ¨$Character^{*}$ ¨

CharLit ::= '*Character*'

TypeID ::= `string` | `bool` | `int` | `float` | `char` | `Any` | *Id*

Appendix 2: Complete Overview of Operational Semantics

...

$$\textsc{Rule}\ \frac{\mathit{top}}{\mathit{bottom}}$$

Appendix 3: Complete Overview of Typing Rules

...

Appendix 4: Complete Overview of Compile Errors

Appendix 5: Complete Overview of Instruction List

A complete and fully up to date list can be found here.

#	Opcode	Arguments	Stack before	Stack after	Description	Notes	Version
0	NOP		TBD	TBD	No operation		V1.0
1	PUSHC		TBD	TBD	No operation		V1.0
2	PUSHN		TBD	TBD	No operation		V1.0
3	PUSHV		TBD	TBD	No operation		V1.0
4	PUSHWS		TBD	TBD	No operation		V1.0
5	ADD		TBD	TBD	No operation		V1.0
6	SUB		TBD	TBD	No operation		V1.0
7	MUL		TBD	TBD	No operation		V1.0
8	DIV		TBD	TBD	No operation		V1.0
9	MOD		TBD	TBD	No operation		V1.0
10	NNEG		TBD	TBD	No operation		V1.0
11	CONCAT		TBD	TBD	No operation		V1.0
12	LEN		TBD	TBD	No operation		V1.0
13	INC		TBD	TBD	No operation		V1.0
14	DEC		TBD	TBD	No operation		V1.0
15	SETVAR		TBD	TBD	No operation		V1.0
16	GETVAR		TBD	TBD	No operation		V1.0
17	SETFIELD		TBD	TBD	No operation		V1.0
18	GETFIELD		TBD	TBD	No operation		V1.0
19	GETELEM		TBD	TBD	No operation		V1.0
20	SETELEM		TBD	TBD	No operation		V1.0
21	JMP		TBD	TBD	No operation		V1.0
22	JMPF		TBD	TBD	No operation		V1.0
23	JMPT		TBD	TBD	No operation		V1.0
24	LJMP		TBD	TBD	No operation	Not used	V1.0
25	CALL		TBD	TBD	No operation		V1.0
26	VCALL		TBD	TBD	No operation		V1.0
27	XCALL		TBD	TBD	No operation		V1.0
28	RET		TBD	TBD	No operation		V1.0
29	CMPE		TBD	TBD	No operation		V1.0
30	CMPNE		TBD	TBD	No operation		V1.0
31	LE		TBD	TBD	No operation		V1.0
32	GE		TBD	TBD	No operation		V1.0
33	GEQ		TBD	TBD	No operation		V1.0
34	LEQ		TBD	TBD	No operation		V1.0
35	LNEG		TBD	TBD	No operation		V1.0
36	LAND		TBD	TBD	No operation		V1.0
37	LOR		TBD	TBD	No operation		V1.0
38	BAND		TBD	TBD	No operation		V1.0
39	BOR		TBD	TBD	No operation		V1.0
40	TUPLECMP		TBD	TBD	No operation		V1.0
41	TUPLENEW		TBD	TBD	No operation		V1.0
42	TUPLEGET		TBD	TBD	No operation		V1.0
43	ALLOC		TBD	TBD	No operation	Not used	V1.0
44	ALLOCARRAY		TBD	TBD	No operation		V1.0
45	CTOR		TBD	TBD	No operation		V1.0
46	TRY		TBD	TBD	No operation		V1.0
47	THROW		TBD	TBD	No operation		V1.0
48	BRK		TBD	TBD	No operation		V1.0
49	POP		TBD	TBD	No operation		V1.0
50	CASTI2F		TBD	TBD	No operation		V1.0
51	CASTF2I		TBD	TBD	No operation		V1.0

52	CASTF2I		TBD	TBD	No operation		V1.0
53	CASTS2F		TBD	TBD	No operation		V1.0
54	CASTF2S		TBD	TBD	No operation		V1.0
55	CASTL2F		TBD	TBD	No operation		V1.0
56	CASTL2I		TBD	TBD	No operation		V1.0
57	CASTL2S		TBD	TBD	No operation		V1.0
58	CASTF2L		TBD	TBD	No operation		V1.0
59	CASTI2D		TBD	TBD	No operation		V1.0
60	CASTS2D		TBD	TBD	No operation		V1.0
61	CASTF2D		TBD	TBD	No operation		V1.0
62	CASTL2D		TBD	TBD	No operation		V1.0
63	CASTD2I		TBD	TBD	No operation		V1.0
64	CASTD2S		TBD	TBD	No operation		V1.0
65	CASTD2F		TBD	TBD	No operation		V1.0
66	CASTD2L		TBD	TBD	No operation		V1.0
67	WRAP		TBD	TBD	No operation		V1.0
68	UNWRAP		TBD	TBD	No operation		V1.0
69	BCKM		TBD	TBD	No operation		V1.0
70	BDOP		TBD	TBD	No operation		V1.0
71	EXTTAG		TBD	TBD	No operation		V1.0

Appendix 6: Complete operator list

...

Appendix 7: Complete keyword list

Keyword	Contextual	Description
`var`		Declares a variable - the type is decided statically by the type-checker.
`true`		Boolean value of true (1)
`false`		Boolean value of false (0)
`null`		Pointer to non-existing object
`void`		Function returns no value
`Any`		Variable or argument may be of any type
`if`		If-statement
`else`		
`for`		
`foreach`		
`while`		
`do`		
`match`		Pattern matching initializer
`case`		A case initialiser for a pattern in a pattern match
`break`		
`return`		
`continue`		
`yield`	X	
`when`	X	
`as`	X	
`is`	X	
`class`		
`new`		
`this`		
`base`		
`enum`		
`try`		
`catch`		
`throw`		
`using`		
`from`	X	
`namespace`		
`public`		
`private`		
`protected`		
`static`		
`abstract`		
`interface`		
`tagged`	X	
`sealed`		
`external`		
`internal`		
`virtual`		
`override`		
`operator`	X	