EDSAC FORTRAN II COMPILER USER MANUAL

Elliot Alexander

University of Southampton

Table of Contents

EDSAC FORTRAN II COMPILER USER MANUAL

This document serves as supporting documentation for my Dissertation project, entitled 'EDSAC FORTRAN II Compiler'. The aim of this document is not to provide technical documentation, rather a guide for the user attempting to compile running programs for EDSAC using the compiler. This will encompass the scope of the FORTRAN language used, as documentation on the specifics of FORTRAN II is sparse. Additionally, this document will encompass the installation and compilation procedures for anyone attempting to build the project from source, as well as some of the nuances of the language. In some cases, the language and syntax of FORTRAN II has been adopted to enable functionality on EDSAC. Early versions of FORTRAN were often machine specific, with researchers often making a variety of changes, adaptations and modifications depending on their specific use case. This led to the standardisation of FORTRAN '66. This is not an inherent problem; however, some nuances of early FORTRAN are unobtainable on the EDSAC.

Getting started

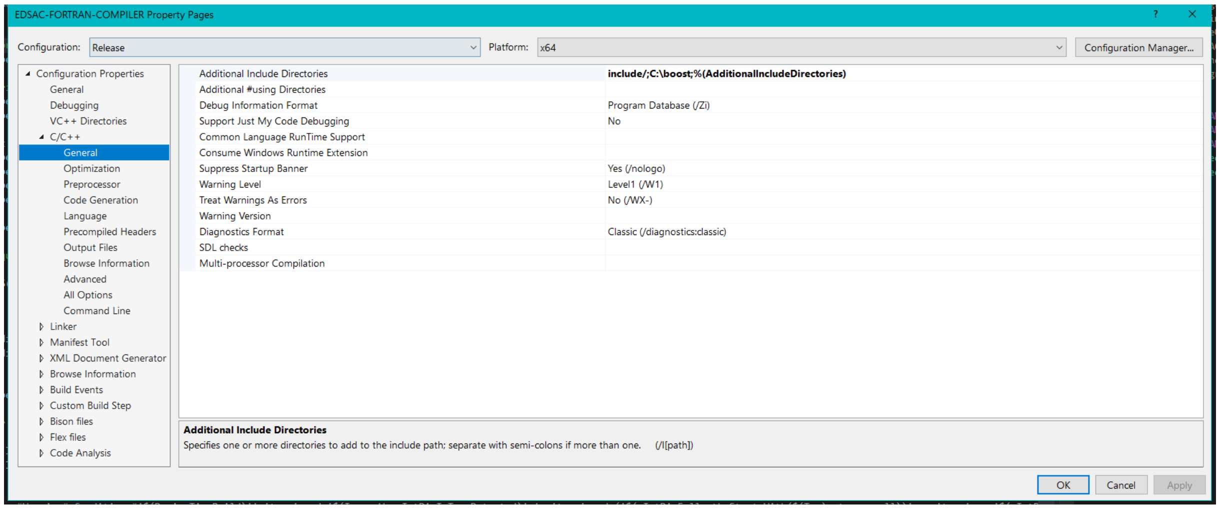
The compiler is distributed in two formats, both as a compiled binary for each platform, and as C++ Source. If you are not planning on making modifications to the compiler, it is advisable to use the pre-compiled binaries. The compiler has been tested on Windows 10, MacOS Mojave (10.14), and Ubuntu 18, although compatibility should remain with any modern Linux distro. The compiler is bundled as standalone binary and should require no supporting files on any platform. Should you wish to make modifications to the compiler, or build the compiler for another platform, the following section will be of use. If not, safely skip past the 'Compiling from Source' section.

Compiling from Source

The following section will briefly outline how to compile the compiler on each platform. The compiler is targeted at C++14 and relies on some features exclusively introduced in C++14. Hence, it is extremely unlikely that the compiler will run on earlier versions without significant modification.

Windows:
It should be noted from the outset that compilation for Windows is by some margin the most difficult of the three supported platforms, and a working knowledge of C++ and Visual Studio is advisable. Compilation on Windows is best achieved using Visual Studio Community 2017, with MSVC version 141. Flex and Bison versions used are 2.5.4 and 3.0 respectively. Additionally, Boost version 1.69 is used. In order to compile the project on Windows, both Boost and Flex/Bison must be in your PATH. The required Boost libraries are not prebuilt; hence Boost must be compiled in order to be used. Compiling Boost from source for MSVC can be a complicated process, the scope of which is outside this document, however the compiler does come bundled with project files for a Visual Studio build solution which relies on the path for

Boost being installed to C:/boost. If installation to another directory is required, the included path can be modified inside the Property Page of the bundled solution. Once open, see ‘Additional Include Directories’ in the Solution Property Page, where ;C:\boost; is replaced with ;<the absolute path>; of the Boost install directory.



Configuration: Release Platform: x64

Additional Include Directories	include/;C:\boost;%(AdditionalIncludeDirectories)
Additional #using Directories	
Debug Information Format	Program Database (/Zi)
Support Just My Code Debugging	No
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level1 (/W1)
Treat Warnings As Errors	No (/WX-)
Warning Version	
Diagnostics Format	Classic (/diagnostics:classic)
SDL checks	
Multi-processor Compilation	

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one.   (/I[path])

Figure 1 The property pages pane of the bundled Visual Studio project

Once Flex and Bison are added to the path, and Boost is installed and included with MSVC, the solution can be built. The bundled solution includes custom build rules which generate the Flex / Bison files before beginning the build process. This enables the entire build process to be contained within Visual Studio. While it is advisable to use the bundled build solution, the compiler does come bundled with the custom build rules required to set up a new solution using Flex and Bison. Alternatively, the Flex and Bison files can be compiled from the command line, using the following commands (Presuming both are already included in your PATH variable):

```
bison -d parse.y -o parse.tab.cc
flex -o lex.flex.cc
```

Ultimately, the end result of the compilation should be the parse.tab.cc and lex.flex.cc files being included in the project directory. Once this setup is complete, the project can be built inside Visual Studio with ‘Build -> Build Solution’.

MacOS Mojave

Compilation on MacOS is decidedly easier, with Flex and Bison being available via ‘Brew’. Note that at the time of writing, the most recent versions of Flex and Bison available are 2.6.4 and 3.3.2, both of which are compatible. Installation on MacOS for Flex and Bison is via:

```
Brew install flex
Brew install bison
```

Again, Boost is required. The bundled Makefile has a variety of configuration options, but the Boost directory is the most likely to be needed. By default, the Boost installation directory is configured as ‘`/usr/local/Cellar/boost/1.69.0/`. It is important to note that unlike on Windows, where simply adding the Boost directory to the include path is sufficient, G++

requires both the -I and -L flags to be configured for Boost. Hence, note that both the Link and Library flag parameters in the Makefile will need to be modified if Boost is installed in an alternative path. The compilation of Boost is again outside the scope of this document; however it is important to note that the included Makefile targets G++, and hence Boost should be compiled for GCC.

With both Boost and Flex/Bison installed, compilation on MacOS is provided through the included Makefile. The command `make macos` runs the build script for MacOS.

Linux Compilation

Compilation for Linux is largely the same as MacOS, utilising the same toolchain. Boost is required to be installed, and the installation directory configured in the Makefile. Flex and bison are both installable via:

```
sudo apt-get install flex
sudo apt-get install bison
```

Like MacOS, both the -I and -L flags are included, both to include and link the boost library to G++. The command in Make to execute the linux build path is:

```
Make linux
```

The boost install path must be set in the makefile. By default, this is configured as /usr/lib.

Software Versions

The software versions utilised in each compiled binary (this also serves as the test platform) are included at the foot of this document.

Segments

The compiler breaks down each program into a series of Segments, corresponding to `FUNCTION,` `SUBROUTINE` and `PROGRAM` blocks. These segments are then processed in their normal order, with the exception or PROGRAM blocks, which are processed last. This is to enable proper function and subroutine mapping prior to their calling instances. As segments are loaded, they are outputted in the format:

```
      +SUBROUTINE [4,10]{SUBROUTINEFACTORIALS(X)}.
       +PROGRAM [11,12]
```

Where the two integers indicate the starting and ending point of that block. In the case of SUBROUTINES and FUNCTION blocks, the name and arguments of the function are also outputted. This is to aid in ensuring that the program is parsed properly. Should a segment be improperly terminated, a warning will display prior to parsing of the values within it.

Command Line Output

Command line output is broken into three categories, [INFO], [WARN], and [ERROR]. Info messages are general broader debugging or compilation information, while Warn messages generally represent semantic oddities which may present a problem, for example, a line mapping referenced at the start of a program which has not yet been declared. Error messages are generally serious problems with compilation and are not advisable to ignore.

At the end of compilation, a summary of the messages outputted by each is outputted. The formatting of the messages, contained within "[ ]", is designed to make filtering and sorting the messages easier. The compiler can generate a lot of output (Upto hundreds of lines with all options enabled), so searching or filtering the output by a regular expression (for example: '^\[ERROR\]' will match only lines tagged as errors.

Command Line Options

The compiler has been designed from the ground up to offer a wide variety of debugging options, output, and warnings. This section will briefly examine each of these, as well as explaining their usefulness. Each argument is specified as "-<character>/--<string>", where the former is a character representing the argument, and the latter is the full expansion of the argument.

Some command arguments specify a string or number following them, and this is indicated in the heading of each. It should also be noted that some arguments are unused or have no effect on current versions of the compiler and are included for expansion purposes only. Such libraries are marked with a ***"Not Relevant"*** tag.

-f / --file <Argument>

Arguably the most important argument, this specifies a file input to the compiler. Valid file extensions are validated against a preset list of extensions. Files load from the current executable path and support relative paths. Each file successfully loaded is indicated alongside the loading of command arguments at the top of the compilers output.

```
+load: Tests/Statement tests/SUBROUTINE_test_1.f
```

-e / --library <Argument> *(Not relevant)*

This option is relates to the extensibility of the system for including subroutine libraries in the program. The compiler (in its current state) comes bundled with only two libraries – P6 and M20, where P6 is the only library utilised by the compiler. For future versions of the compiler, there is scope for adding a full suite of includable subroutines, including user configurable routines. This routine takes an argument string following it, in the format:

--library <Routine name>

Where Routine name is the name of the library to be included. For example:

--library P6

Adding this flag will force the P6 library to be included in the compilers output, offsetting the main program as such.

When a library is enabled, the following output will be posted.

```
:: Standard Library Initialisations ::

[INFO] Building bundled libraries.
[INFO] Skipping M20, remains disabled
[INFO] Offsetting P6 by 56
[INFO] Adding subroutine P6[32].
[INFO] Total library size: 32
```

In the current version of the compiler, there is **no need to specify this option.** All statements requiring subroutine libraries are capable of enabling their inclusion internally. The inclusion of this argument is intended for expansion purposes however, as well as for a documentary aspect.

-o / --output <Argument>

This option allows the user to specify a custom output file. Only one output file is generated, and by default the file **out.edsac** is generated if this option is not specified. Note that the value of this option also affects the output of Three Op Code to file, with the output file for Three Op Code being formatted as:

```
              toc_<output_file_name>
```

Where output_file_name is the option passed into this flag.

-h/--help

This option is reasonably self-explanatory, and outputs the help page for the compilers command line options. Note that this option is **blocking**, and compilation will not progress past the output of the help page.

-x / --allextensions

This option disables warnings relating to file type extensions, as well as some of the verification relating to if a file is accessible and regular. This option may be useful if encountering filesystem issues relating to files not being openable, or the compiler rejecting files because they're not accessible.

-l / --linemappings

This option enables outputting of line mapping offsets. When the compiler links three op code for EDSAC, each line reference is offset by a value relating to the size of the three op code before it. This option also enables output of line mappings as they are added into the internal representation, allowing a user to track references from FORTRAN II to EDSAC addresses as the output is built. This is a more advanced debugging feature, designed primarily to help diagnose bugs with the compiler itself. However, this may be useful for determining where individual statements, line labels or subroutines are located if attempting to modify compiled EDSAC code by hand. One potential use case here is compiling Subroutines for EDSAC, which will then be used in other code. This option may help an end user locate the start point of a subroutine, where it can be modified and injected into other EDSAC code.

-d / --dumpfiles

This option enables the dumping of all input files at the start of execution. This option may be useful for generating a recordable compiler output – i.e. The input program is included in the compiler output, so the output itself may be used as a standalone reference of what was compiled, and hence outputted. This option may also be useful for debugging encoding or file loading problems, in addition to generally providing a more useful overall output.

-t / --tokens

This is perhaps one of the more useful compiler outputs. This output in particular causes the compiler to output the identified type of each Statement it encounters. This is primarily useful in isolation for validating that the parsing stage of the compiler has succeeded, successfully identifying each individual token in the FORTRAN II program. This is also useful for recording in the compiler output the complete stream of statements not only included in the File (see "**-d**"), but the type of statement identified, as well as the Segments these files are contained within.

Additionally, this option is best paired with "**-p**", which enables complete output of all parsed values from each statement. In the event of a problem with a statement not parsing correctly, these two options will output the complete "end result" – i.e. all values loaded into the internal representation, of the parsing stage.

-p / --parsedvalues

The compiler functions by identifying each statement uniquely using a regular expression, determining its structure and type. There is then an individual implementation (on a statement by statement basis) which dissects each statement type. This option enables complete output from that individual dissection, outputting to the console each element of each statement processed. This is extremely useful for ensuring that the values loaded into the compilers internal representation are correct, in addition for debugging some common pitfalls with specific statements.

-h / --stops

This option enables the output of all operations on the symbol table during compilation, including the memory offset amounts of each variable, in addition to the base offset of the symbol table. This is typically the sum of the size of the complete compiled program, the base memory offset, and the size of any included libraries. This option adds a large amount of output to the tokenization section, whereas each statement is processed, it's variables and data addresses are initialised. This also allows the inspection of which addresses are used by specific statements, as well as what for.

This option is useful for debugging errors where data values are incorrectly referenced, or arguments are not processed correctly. This option is also best paired with "**-y**", which dumps the final symbol table to an output at the end of compilation. This allows the tracking of variables and symbol table entries throughout the compiler, showing the initialisation point in the context of the Statement being processed, as well as the offsetting and final location of each value.

-y / --stdump

This option enables the complete dump of the offset Symbol Table once compilation has completed. The symbol table output is broken into various function scopes, in addition to the main program scope, and each scope is divided into declared, undeclared, temporary and Common symbol tables. Note that in the current version of the compiler, the Common table is a placeholder and remains unused.

-c / --toc

This option enables output of the internal representation, or three op code, of the compiled EDSAC output. This output maps directly onto the addresses and instructions loaded into EDSAC, a feature which makes this representation both extremely useful for debugging, as the entire program structure (including data value) can effectively be read in plain text on the command line. Note that this is outputted in the format:

```
[x] <Instruction> <Address>
```

Where x is the memory address the instruction will be loaded into, and the instruction and address are outputted as strings. For example, the instruction A145F in address 91 will be printed as:

```
[91] ADD TO ACCUMULATOR 145
```

Also note that this option enables file output of the three-op code, in a file prefixed with toc_, followed by the name of the EDSAC output file. For example, if an output file name of **"hello_world.edsac"** is specified, and "**-c**" is enabled, the three op code will be output in a file named "**toc_hello_world.edsac**".

One caveat of the outputted three op code is the treatment of control characters. Specific instructions inside EDSAC are effectively invisible, processed by the Initial Orders but never actually loaded into memory. For example, a control code of "**GK**" appears in the inputted instructions, and appears as an instruction, however is never actually loaded into memory. In order to maintain a direct mapping of memory addresses and locations from the three op code to the addresses loaded into EDSAC, these control codes are "hidden". The majority of control codes are accompanied by a genuine instruction, and hence the control code value is nested within the short long bit of each instruction. This completes the output, and ensures that it maps correctly, however makes three op code sometimes difficult to read. For example, when defining a function, the opening instruction is always "GKA3F", which builds the return instruction from the value loaded into the accumulator. In this case, the control code "GK" precedes the rest of the instruction, so the control code cannot be hidden within the end of the instruction. A3F is a fixed value, so the three-op code representation of this function is defined as NO_OPERATION, and the complete control code is included in the address field. In this case, the three-op code of the output is:

```
        [88] NO OPERATION  GKA3F
```

This allows the control code to represent its true size in memory (i.e. The control code + "A3F" instruction is two EDSAC instructions, but only A3F is loaded into memory), without offsetting all later instructions by an incorrect amount. If "GK" and "A3F" had separate entries in the internal representation, "GK" would not be loaded into memory by EDSAC and the address of all preceding instructions would be incorrect. This is a quirk of the implementation, but one which ensures the control codes are included in the three-op code output while maintaining a correct mapping from three op code to the program inside EDSAC.

-f / --functionmappings

This option functions similarly to "***-l***" and "***-s***", outputting the mappings and offsetting for all functions declared within the program. This is useful for building functions or subroutines as independent programs, as well as for diagnosing errors relating to the mapping of functions, the control flow jumps into and out of functions or subroutines. Note that this one argument covers both Subroutines and Functions, however arithmetic functions are not included here.

-r / --regex

As addressed previously, the compiler interprets statements in FORTRAN II by first identifying their type from one of an initialised list of statement types. Each statement type has attached to it a Regular Expression (regex), which defines the structure, allowed characters, and format of the statement expected for that token type. This is the barrier to entry for parsing of statements, and each Statement should only ever be identified by a single regular expression.

The effect of this option is to output, at the start of tokenization, the regular expressions used by every statement interpreted by the compiler. This effectively acts as a syntax map. Note that the complete list of regular expressions interpreted by the compiler is included at the foot of this document.

-b / --baseoffset <Integer>

This argument allows the specification of a custom base memory offset. The implication of this is effectively the starting address of the program. By default, this value is 56. The benefit of this value is that a program can be moved to a later point in memory, depending on the size of the initial orders, or preceding code. Alternatively, Subroutines and functions may be generated with the compiler, before being appended to the end of another program with their base memory offset set such that they function normally.

FORTRAN Implementation

Introduction

The implementation of the FORTRAN language is not universally consistent with original FORTRAN implementations, and some modifications have been made to make the format of EDSAC compatible with FORTRAN II. The specifics of these modifications are outlined, however it is prudent to first examine the specification used, before comparing in detail the modifications made to it.

Hence, this section will briefly examine the syntax and semantics of each FORTRAN statement supported by the compiler, as well as indicating the Statements not supported by the compiler. Additionally, this section will outline the state of implementation of each FORTRAN statement, as some statements are parsed by the compiler, however not implemented in EDSAC.

Arithmetic Statements

One major expansion on the FORTAN syntax made on some specific language elements it the availability of Arithmetic Statements. Syntactically, these are elements where arithmetic operations are valid, and will be passed into the main Arithmetic Parser before being processed. This means that the full range of arithmetic operations is valid within them, including function calls. However, care must be exercises to avoid nesting function calls.

An example of this is demonstrated with the PRINT and IF statements. The Arithmetic element of the syntax is indicated by <Arith>.

```
PRINT <Arith>
IF ( <ARITH> ) <Integer> <Integer> <Integer>
```

Hence, the following statement is interpretable by the compiler.

```
      IF (X – K * 2) 10, 15, 20
```

More details on the Arithmetic Compiler are included later.

Notation

The following sections detailing statement implementations in detail use a notation summarised as a readable alternative to Regular Expressions, where constant values such as <Variable Name> and <Integer> are used in place of their regular expression counterparts. Note that this is an indicative measure designed to make the syntax more approachable. The full matching regular expressions (and hence a complete map of the syntax) is included at the conclusion of this document. Additionally, elements such as [ ] are used to indicate a list, and ? is used to indicate optionality. For example `, [ <Integer>, (<Integer>,)? ]` is used to represent a list of integers of infinite length, where at least one integer must be specified. ( ) is used to indicate grouping, however where ( ) is a syntax element, they are indicated as escaped by `\(`

For the purposes of this entire section, all whitespace is irrelevant in syntax definitions.

Program Structure

All programs are formatted as follows:

```
<5 character line label> <Continuation flag value><Statement>
```

Line Label

Each line has an optional line label, allowing the line to be referenced by GOTO, IF, DO, etc statements elsewhere in the program. This line label must be an integer, and has a maximum length of five characters. Any characters unused in the line label must be paddded with whitespace, ensuring that the line label is exactly five characters long. For example:

```
100   DO 100I = 1,10
```

Note the two additional spaces added to ‘100’ to ensure that the length of the line label as a whole is five charaters long. These line labels are essential for referencing other parts of the program from control statement.

Continuation bit

The above example is correct, however a sharp eyed reader may have noticed that three spaces are added to the ‘100’. The final one of these is known as the continuation bit. Placing **any character** in the location at index six of the statement will mark the statement as a continuation, and append that statement to the line before.

This practice allows for very long statements, with the limit on continuuations being set by the FORTRAN language at 20 consecutive continuations. Commonly, the symbol & is used to mark continuations, however this is purely a convention. Any character placed in location six is valid.

Note that first statement of a continuation does not require it’s continuation bit to be set. This first statement is also the only statement in the continuation which is allowed a line label.

For example, the following may form the statement, ‘`DO 100 I = 1, 10`’.

```
100   DO
     *10
     *0
     *I =
     *1
     *,
     *1
     *0
```

Continuations are expanded before parsing, and there are no restrictions on which language elements can be spread across continuations. The singular restriction is that a continuation cannot be blank.

Tab Characters

It is important to note that the breakdown of this section of the statement by character index means that the compiler **does not support tab characters** in this position. Setting the length of a tab character in an IDE before attempting to compile is extremely unlikely to work. Best practice is to set the value of ‘Tab’ in your IDE to a value of ‘**six spaces**’.

Statement Breakdown:

ARITHMETIC FUNCTIONS

Arithmetic functions are inline function definitions, allowing simple arithmetic operations to be aliased to a function call. Functions are only callable within arithmetic expressions, hence are treated almost exactly the same as Arithmetic Functions. The syntax for defining Arithmetic functions is as follows:

```
<Function Name> \( [<Variable Name> ( ,<Variable Name>)? ] \) = <Arith>
```

Hence, an example of this would be:

```
SUM(X,Y) = X + Y
```

Arithmetic Functions are implemented in a manner identical to conventional functions, hence they have their own scope. To this effect, only variables declared as arguments may be used within the scope of an arithmetic function. Constant values may be used however. For example, the following is a valid arithmetic function:

```
SERIES(X) = X * 2 + 1
```

However, the following would be an invalid arithmetic function, and would throw an error relating to the unavailability of Y.

```
SUM(X) = X + Y + 1
```

Arithmetic functions can be called inside any other Arithmetic Statement block, via `<Function Name>(<Arguments>)`. Arithmetic functions are identical in their access to declared multi-line functions.

ASSIGN Statements

Assign statements are a simpler form of the VARIABLE DECLARATION Statement, allowing only constant variables to be declared. Semantically, ASSIGN statements are handled identically to variable declaration statements. The syntax as such is:

```
ASSIGN <VARIABLE NAME> TO <CONSTANT>
```

Hence, a valid ASSIGN Statement would be:

```
       ASSIGN X TO 1500
```

This is functionally the same as a variable declaration of the same nature:

```
       X = 1500
```

ASSIGNED GOTO

Assigned GOTO Statements are an effective as a form or comparison operator, allowing a variable to be compared against a series of integer constants representing line labels in the program. The syntax of this statement is:

```
GOTO <VARIABLE> \([<INTEGER> (,<INTEGER>)] \)
```

Hence, an example of a valid ASSIGNED GOTO statement:

```
GOTO X (10, 15, 20)
```

The semantics of this statement are as follows; each bracketed constant must be a line label declared elsewhere in the program. EDSAC will iterate through each bracketed line label in turn, comparing it's value to X. If X contains the value of that line label, then control flow will jump to that line. Therefore, in pseudocode, the previous example has the semantics of:

```

If (X == 10)
       GOTO 10
ELSE IF (X == 15)
       GOTO 15
       ELSE IF (X == 20)
              GOTO 20
       ELSE
              CONTINUE
```

It should be noted that the non-constant variable, i.e. X in the above example, is not an Arithmetic field. A statement such as `GOTO X + 1 (10, 15,20)` would be invalid.

CALL

The call statement is relatively simple and provides the only method of accessing a SUBROUTINE. Note that unlike Functions and Arithmetic Functions, SUBROUTINE's cannot be accessed inside an Arithmetic Statement. Instead, they are accessed through the 'CALL' keyword. The syntax is as follows:

```
CALL <FUNCTION NAME>\( [<ARITH> (,<ARITH>)?] \)
```

As such, an example CALL statement would be (presuming a single argument SUBROUTINE has been declared as SQUARE(X)).

```
       CALL SQUARE(5)
```

Alternatively, an equally valid statement demonstrating the use of arithmetic inside arguments would be:

```
CALL SQUARE(5^2)
```

Again, it should be noted that SUBROUTINES cannot be nested, attempting to call a subroutine inside another will produce the following error.

COMMON

Common blocks are not implemented programmatically, however their syntax is parsed and handled. Common blocks follow the syntax of named common blocks from later versions of FORTRAN, hence:

```
COMMON /<Block Name>/ [<VARIABLE> (,<VARIABLE>)]
```

Hence, an example of a valid syntax would be:

```
COMMON /TEST/ X,Y,Z
```

Again, common blocks are not implemented in this version of the Compiler. However, they are a priority for future work, and the ability to parse them has been left in the compiler alongside a warning that they are not programmatically implemented.

COMPUTED GOTO

Computed GOTO works similarly to a ASSIGNED GOTO, however with a slightly different syntax. It should be noted that on EDASC, Computed GOTO's are extremely expensive, requiring upwards of 10 memory addresses for each variable declared. Hence, they should be avoided if possible.

Computed GOTO's allow the user to validate a variable against indexes from 1 -> N. The syntax of a computed GOTO is as follows:

```
GOTO \([<INTEGER> (,<INTEGER>)?]\), <VARIABLE>
```

Hence, an example of a Computed GOTO statement would be:

```
GOTO ( 10, 15, 20), X
```

Semantically, the bracketed array can be of any size, and each value must be a Line Label from elsewhere in the program. The address in location 1 of the bracketed list will be jumped to if X = 1, and the value in location 2 of the bracketed list will be jumped to if X = 2, and so on. Any number of variables can be specified. The control variable specified to the right of the bracketed variables is not arithmetic, and should be assigned prior to the statements execution.

An example program using Computed GOTO is included below. The expected output from EDSAC is **10**.

```
      X = 1
      GOTO (5, 10, 15), X
      PRINT 100
      STOP
5     PRINT 5
      STOP
10    PRINT 10
      STOP
15    PRINT 15
      STOP
      END
```

CONTINUE

Continue is perhaps the simplest FORTRAN statement, with no arguments or semantics to speak of. The meaning of CONTINUE is effectively, 'Do nothing'. However, Continue is extremely useful as a placeholder statement, as it can be used as a line label. The syntax of Continue is extremely simple, at just:

```
CONTINUE
```

DIMENSION

As with COMMON statements, Dimension statements are not supported. Unlike common, they are identified but not parsed by the compiler. In the attempt that a Dimension statement is included in a program, the compiler will display an error message.

DO

DO Loops are fully implemented in the compiler, with a syntax as follows:

```
       DO <INTEGER> <VARIABLE> = <INTEGER>,<INTEGER>
```

Hence, an example DO loop would be formatted as follows:

```
       DO 10 I = 1, 10
```

The semantics of this are reasonably simple, the program will loop upto and **including** the line label specified as the first integer (10 in the example statement), executing everything between the loop initiation and the labelled instruction. The variable provided acts as a loop index, with the two integers provided acting as the initialisation and final values for the loop. Each loop, the provided variable is iterated by 1, until it equals the second provided integer. In this case, the loop will start with I = 1, and ultimately exit (out of the bottom of the loop) when I = 10.

For example, given the following program:

```
      DO 10LOOPINDEX = 1,10
10    PRINT LOOPINDEX
      END
```

The expected output would be a loop printing integer from 1 -> 10. Note that throughout the loop, the control variable (in the above example, this is LOOPINDEX) is accessible. The control variable remains accessible after the loop has terminated, however with one caveat – it's value will be one larger than the termination value of the loop. This is a side effect of the (albeit more efficient) implementation of DO loops, which are top heavy. Once the loop reaches the bottom, it always jumps back to the top, regardless of the value of the control variable. Here, the control variable is iterated, and if it is larger than the final value, the loop is escaped. This implementation is a quirk of the lack of a comparison operator in FORTRAN.

Custom Iteration Amounts

It is possible to specify a slightly different DO loop syntax, namely as follows:

```
      DO <INTEGER> <VARIABLE> = <INTEGER>,<INTEGER>, <INTEGER>
```

In this case, the third integer provided represents the iteration amount, i.e. the amount that the starting variable (LOOP INDEX as above) is iterated before being compared to the final variable. Hence, an example statement in this form would be as follows:

```
      DO 10 I = 1, 20, 2
```

While nested DO loops are allowed, overlapping nested DO loops are not. Each DO loop must terminate before its parent DO loop terminates. Termination on the same line is valid, and the inner loop will complete its iterations up to the specified line before iterating the outer loop.

END

The END Statement takes no arguments and is required at the end of a Program block. The END statement can only be used once in a program and must be the final terminating statement of the program. An END statement cannot be included anywhere else in the program, nor can it be used to terminate SUBROUTINE or FUNCTION blocks.

FORMAT

Due to the complexity of FORMAT statements, and complexity of serious output from EDSAC's single output instruction, FORMAT statements were omitted from the compiler in favour of a simpler implementation of PRINT statements.

FUNCTION

Function statements are essentially longer, multi-line definitions of arithmetic functions. They are declared in the same way, with the addition of a return statement and removal of the single line constraints of Arithmetic functions. They are initialised with a FUNCTION statement and terminated with a RETURN. The syntax of the FUNCTION statement is identical to that of SUBROUTINE, with the obvious modification of the keyword.

```
<Function Name> \( [<Variable Name> ( ,<Variable Name>)? ] \)
```

Functions can accept any number of arguments, although like Arithmetic functions, their names must be capitalised. Functions are implemented for EDSAC as true subroutines, jumping into and out of the same definitions.

Hence, an example Function statement would be as follows:

```
FUNCTION SQUARE(X)
```

In FORTRAN II, the initial specification outlines that the value of a computed function call should be left as a value in the accumulator for further compilation. However, with no direct access to the accumulator, and EDSAC's lack of an unconditional GOTO making leaving values in the accumulator difficult (especially when multiple arguments are being used), this is not the case for programs compiled on the EDSAC compiler. The substitute implementation for this involves the RETURN statement. Typically, the RETURN statement is only used to indicate the end of a FUNCTION or SUBROUTINE and takes no arguments. However, the implementation used for the EDSAC compiler takes on argument, exclusively at the end of a Function. Attempting to use a parameterised return at the end of a Subroutine will throw an error. This argument is arithmetic and indicates the return value of the function. Hence, The following would be valid:

```
FUNCTION SQUARE(X)
X = X^2
RETURN X
```

In this case, calling a function inside an arithmetic statement would result in the return value of X. It is important to recognise that this is within the function scope, so returning variables declared outside of the function scope is not valid. For example, the following is invalid, as Y is not referenced within the function scope.

```
ASSIGN Y TO 10
FUNCTION SQUARE(X)
X = X^2
RETURN Y
```

However, the following is valid, as a separate instance of Y is declared within the functions scope. This will be explored in more detail later.

```
ASSIGN Y TO 10
FUNCTION SQUARE(X)
Y = X^2
RETURN X
```

GOTO

In comparison to its computed and assigned GOTO counterparts, the unconditional GOTO is relatively simple. GOTO takes a single argument, an integer line label, and directs program flow to that line. Note that this is **inclusive** of the line attached to the label. The syntax therefore, is:

```
GOTO <INTEGER>
```

GOTO's may be used as forward declarations for control flow, that is to say that referencing a line label inside a GOTO statement prior to that line being processed is acceptable. It is also valid to jump into and out of functions, however care must be taken to ensure that a return is made for the termination of the function. This will be explored in more detail later.

The concept of inclusivity of line labels is best illustrated with an example. The following program will loop infinitely.

```
1000  GOTO 1000
      END
```

This statement is sometimes referenced as an unconditional GOTO, as if executed, it will always jump. Th

IF

IF statements are particularly useful, both as a conditional jump and comparison operator. An IF statement takes a control argument, and three parameterised integers. This control variable is arithmetic. Hence, the syntax for an IF Statement is as follows:

```
IF \( <ARITH> \) <INTEGER>, <INTEGER>, <INTEGER>
```

An example IF statement would be as follows, where 10, 15 and 20 are line labels:

```
IF (X + 1) 10, 15, 20
```

Note how the parameterised argument is valid for arithmetic, meaning that statements such as the following are also valid:

```
IF (((((X))))) 10, 15, 20
```

The semantics of an IF Statement are simple, summarised as following (where Z is the computed value of the provided arithmetic, and x_1, x_2, x_3 are the provided line labels):

```
IF (Z < 0)
       JUMP TO x₁
ELSE IF (Z > 0)
```

```
       JUMP TO $x_3$
ELSE IF (Z == 0)
       JUMP TO $x_2$
```

This makes an IF Statement especially useful as a comparison operator. Statements such as the following can be used to determine equality between two variables. Included in the working examples is the use of an IF statement for conditionals and comparing two integers.

PRINT

Print statements are a departure from FORTRAN II, offering the only method of output on EDSAC. Print statements take only one argument, an arithmetic statement, the value of which is ultimately printed as a positive Integer. Print statements in the EDSAC compiler take advantage of P6 for printing, and hence printing negative numbers is not supported.

The syntax of PRINT is simply:

```
PRINT <ARITH>
```

Therefore, any manner of arithmetic may be done inside the PRINT statements argument. For example, this is a valid statement.

```
PRINT (10 + 15 + 25)^2 * 4
```

The expected output in this case would be **‘10000**’.

READ

Read statements are not implemented in the Compiler, however they are parsed. This functionality is left in for a view for future expansion.

RETURN

Return statements take two forms, depending on whether an argument is included. Return statements act as a bookend for FUNCTION and SUBROUTINE blocks, with each version of the RETURN statement being applicable to one version. Subroutines exit with a plan RETURN statement, included purely to indicate the end of the Subroutine. This takes no argument, as Subroutines have no return values.

Function definitions take a variable argument, representing the return value of the function. This value must be a variable declared in the Function scope.

The syntax is hence as follows, note the optionality on the parameterised VARIABLE.

```
RETURN <VARIABLE>?
```

In the event that a parameterised RETURN statement is used to cap a Subroutine, an error will be thrown. This is also true of then reverse. Included below are examples of simple Function and Subroutine definitions, the former with a parameterised return and the latter without.

Function Return

```
      FUNCTION SQUAREX(X)
      Y = X^2
      RETURN Y
      PRINT SQUAREX(5) + 5
      END
```

Subroutine Return

```
      SUBROUTINE SQUARE(X)
      PRINT X^2
      RETURN
      CALL SQUARE(5)
      END
```

Note how Subroutine return features no argument, whereas a Function return does. This parameterised return can be used for any manner of arithmetic.

It should be noted that Functions and returns do not need statements, indeed the following is a valid program:

```
      FUNCTION POWERS(X,N)
      RETURN X
      PRINT POWERS(3,5)
      END
```

REWIND

Rewind has no support within EDSAC and is hence not implemented in the Compiler. The statement will parse, and is accepted as a valid token, however this is more with an eye of providing informative 'Not implemented' error messaging, as opposed to any meaningful functionality.

STOP

There are two types of STOP statement implemented for EDSAC. The first takes no arguments, and simply stops program flow via EDSAC's 'STOP' instruction. This features no additions and is effectively an alias to the instruction of the same name on EDSAC.

The parameterised STOP is slightly different, taking an Arithmetic value, which is left in the accumulator prior to the stop instruction being called. Hence, the syntax is as follows:

```
STOP <ARITH>?
```

With the arithmetic argument optional, the simplest form of the STOP statement is written purely as `STOP`. This is reasonably self-explanatory. The parameterised STOP statement, however, is more useful. An example of a parameterised stop statement might be:

```
STOP X + 1
```

In this case, prior to the program STOP, the value of X + 1 will be computed and loaded into the accumulator. Hence, if X = 5, the value in the accumulator at program stop would be 6.

VARIABLE DECLARATIONS

Variable declarations in the EDSAC compiler function identically to ASSIGN statements, however with syntactic advantages. The syntax of an inline variable declaration is as follows:

```
<VARIABLE NAME> = <ARITH>
```

The advantage of variable declarations over ASSIGN statements is their ability to take arithmetic values. This makes them far more versatile, and the backbone of the arithmetic system in the compiler. An example as such would be:

```
X = (Y + 5)^2
```

This is reasonably self-explanatory and has no significant caveats. As with all variable declarations, function scoping should be observed.

Other notable semantics

Symbol Table Scoping

Function and Subroutine scoping within the Compiler is relatively simple, albeit complicated by the inclusion of GOTO jumps. The summary of considerations with Function and Subroutine scoping is this, on the declaration of a function or Subroutine, a new symbol table is generated for the function or subroutine. This symbol table remains the **in-use** symbol table until **the end of the function or subroutine** is observed. However, Statements processed outside of the scope of a Function or Subroutine will use the **Main Program** symbol table.

The implication of this, is that GOTO's can be used to jump in and out of functions or subroutines, however consideration must be given to the scope that the statements jumped to were processed in. It's also generally advisable to ensure that control flow returns to the function before the end of the program.

This is best demonstrated with an example. The output from following the following program is " **20 25 25 10**".

```
      FUNCTION SQUARES(X)
      X = X^2
      PRINT X
      GOTO 5
10    CONTINUE
      RETURN X
      X = 20
      PRINT X
      PRINT SQUARES(5)
      PRINT X
      STOP
5     X = 10
      GOTO 10
      END
```

The program begins execution from X = 20, where X is immediately printed. This is responsible for the first ’**20**’. The program then makes a control flow jump to the function Squares, where the current value of X in the main program scope is copied across to X in the Function scope. The value of X in the function scope of Squares is hence, 5. X then becomes X^2 , hence ‘**25**’ is printed from within the Function Scope. Control Flow then jumps to line 5, which was **processed inside the main program scope**. In this regard, line 5 is referencing the Main Program Scope, so the value of X in the main program scope is changed from 20 to 10. Control flow then jumps back to the Function Squares, which returns the value of X. The value of X inside the Function scope is 25, so `PRINT SQUARES (5)` is responsible for printing ‘**25**’ a second time. The final `PRINT X` statement is then processed, again back in the main program scope (As the function scope terminated upon RETURN), so the value of X set at line 10 is printed, i.e. the final ‘**10**’.

Figure and Letter shift

Output in EDSAC is enabled via a teleprinter attached to the machine. In reality, this teleprinter has featured on it a hardware switch, enabling toggling between ‘Figure’ and ‘Letter’ shift. For each data bit value in EDSAC, there is a ‘Figure’ and a ‘Letter’, i.e. a character and numerical value. However, the EDSAC Simulator does not have this hardware toggle, and instead Figure and Letter shift can be toggled by printing a special character.

By default, the EDSAC simulator is set to ‘Letter Shift’ and requires toggling to ‘Figure shift’ before output from the compiler will work. FORTRAN II had no support for strings or characters as data values, and hence the entire output of the compiler is in ‘Figure Shift’ mode.

Prior to using the compiler, it is important to ensure that ‘Figure Shift’ is set. This can be done by executing the following program.

Figure Shift enable

Executing the following program on the EDSAC simulator will toggle the simulator into Figure Shift mode.

```
T64K
GK
ZF
O4@
O5@
ZF
#F
10F
EZPF
```

Letter Shift enable

Executing the following program on the EDSAC simulator will toggle the simulator into Letter shift mode.

```
T64K
GK
ZF
O4@
O5@
ZF
*F
HF
EZPF
```

Arithmetic Parser – extended

The Arithmetic Parser is a subsection of the main compiler, accessible through a single function call. The arithmetic parser encompasses all arithmetic operations inside the compiler, taking an arithmetic string and building a parse tree before returning the head of the tree. The head of the tree is a standardised object, on which a single function can be called to generate the three-op code required to compute the entire value of the tree. This approach has two major benefits:

1) The arithmetic parser is a separately testable, entirely modular part of the program.
2) Oddities in the FORTRAN language are isolated from arithmetic statements, a pure, pre-processed arithmetic string is passed to the Arithmetic Parser, free of continuations, whitespace, keywords, etc.

Functions, whether arithmetic or conventional, are accessible by the arithmetic compiler, as are variables declared elsewhere in the program. It is this structure which handles operations performed on the output values of functions, in addition to basic variable access and constant value access.

Common Problems / FAQ's

Common Compilation Problems:

No END statements included in source

A relatively easy mistake to make is to forget to include an 'END' statement in a program. The error displayed is outputted at the top of the compilers output, and hence is easily missed. The error message outputted appears as:

```
:: Beginning Tokenization ::

[ERROR] Warning – failed to load a fail program block from file
Tests/Error_Tests/No_End.f
```

```
[ERROR] Did you forget to include an END Statement?
```

Multiple END Statements included in source:

Another relatively simple mistake to make when writing FORTRAN source for the compiler is to include multiple END Statements. END is the indicator for the end of the program and should not be used in the place of a STOP statement.

In the case of multiple END statements, the **last seen END statement is respected**, i.e. All preceding END statements are ignored. Hence, for an example such as:

```
      X = 10
      PRINT 10
      END
      X = X + 1
      PRINT X
      END
```

In this case, the following Segment breakdown is included:

```
+PROGRAM [1,3]
[INFO] Entered main program at line 4
+PROGRAM [1,6]
```

The generated program as such, begins at line 4 and continues until line six. The first program segment is ignored.

Invalid use of Arithmetics:

Careful attention needs to be paid to the location of Arithmetic Statements in the compiler, as using them in incorrect places will throw a Syntax error, and the statement will not be parsed.

Notable examples of this include:

- The variable fields of DO loops. For example:

  ```
        DO X I = 1, K
  ```

  Neither X nor K is valid in this context.

- The line label fields of an IF statement: For example:

  ```
        IF (X) I, I + 1, J*J
  ```

  None of the line label fields are valid in this context (I, I + 1, J*J). Only integer line labels may be specified.

- GOTO Statements – No line label referenced in any form of a GOTO statement is valid.

  For example:

```
      GOTO X+1

      GOTO X, (X+1, K, 2+2)
```

  None of the above is valid. Any line labels specified in GOTO statements must be integer values only.

Invalid Arithmetic Statements

Arithmetic errors are handled within the Arithmetic Parser, which is enclosed within the main compiler. This has its benefits; however, one disadvantage is that if the arithmetic parser fails, the rest of compilation is irrecoverable. The significant benefit is greatly improved error messaging. Hence, if an arithmetic error is included, the entire compiler will exit. For example, given the following program:

```
      PRINT X ++ 5
      END
```

The outputted error is:

```
[1][PRINT] { PRINT X ++ 5 }
[INFO] Variable X not found in Symbol Table.
[INFO] Added X to symbol table UNDECLARED_VAR
[ERROR] Arithmetic Parser Error. Message: syntax error, unexpected PLUS
[ERROR] Error - arithmetic value X++5 failed to parse correctly.
[ERROR] Arithmetic errors are unrecoverable, the compiler will exit.
```

Referencing invalid line labels

Invalid line labels do not produce an error, rather a warning, and as such is one of the harder semantic errors to diagnose. If an invalid line label is used, the compiler will produce a warning similar to:

```
[WARN] Failed to retrieve line mapping for 100
[WARN] This may be indicative of an invalid line label, or a reference to an non-
existent line label.
```

Software Versions

Software Version	MacOS	Linux	Windows
Flex	2.6.4	2.6.4	2.5.4
Bison	3.3.2	2.3.0	3.0.0
C++14 Compiler	Clang 10.0.1	G++ 7.0.3	MSVC 141
Boost	1.6.90	1.6.90	1.6.9

Regular Expression Values

Included in the following section are the regular expression values for each Statement type. These define the syntax of the language, and act as a barrier to entry for parsing of specific values. Note that some syntactic elements of statements may be supported in parsing purely for the purposes of informing the user that a specific element of the syntax is not supported.

Statement Type:	Regular Expression
DO	`DO([0-9]+)(([0-9a-zA-Z\.\(\)\+\*\/\^\-])+)=([0-9]+),([0-9]+)(,([0-9]+))?`
SUBROUTINE	`SUBROUTINE([A-Z]([0-9A-Z]+)?)(\(([A-Z]([0-9A-Z]+)?(((,[A-Z]([0-9A-Z]+)?)?))+)?\))?`
END	`END(\([0-9]+,[0-9]+,[0-9]+,[0-9]+,[0-9]+\))?|(END)(FILE)([0-9a-zA-Z\.\(\)\+\*\/\^\-])+`
CALL	`CALL([A-Z]([0-9A-Z]+)?)\((([0-9A-Z\.\(\)\+\*\/\^\-])+((,[0-9A-Z.\(\)\+\*\/\^\-]+)+)?)?\)`
RETURN	`RETURN|RETURN([0-9a-zA-Z\.\(\)\+\*\/\^\-])+`
ARITHMETIC_FUNCTION	`([A-Z]([0-9A-Z]+)?)(\([A-Z]([0-9A-Z]+)?(((,[A-Z]([0-9A-Z]+)?)?))+)?\)[=]([0-9A-Z\.\(\)\+\*\/\^\-])+((,[0-9A-Z.\(\)\+\*\/\^\-]+)+)?`
STOP	`STOP|STOP([0-9A-Z\.\(\)\+\*\/\^\-])+((,[0-9A-Z.\(\)\+\*\/\^\-]+)+)?`
VAR_DECLR	`[A-Z]([0-9A-Z]+)?=(([0-9a-zA-Z\.\(\)\+\*\/\^\-])+|(([A-Z]([0-9A-Z]+)?)\(([0-9a-zA-Z\.\(\)\+\*\/\^\-])+((,([0-9a-zA-Z\.\(\)\+\*\/\^\-])+)?)+\)))+`
DIMENSION	`DIMENSION(([A-Z]([0-9A-Z]+)?\([0-9]+\),?)+)`
GOTO	`GOTO([0-9]+)`
ASSIGN	`ASSIGN[A-Z]([0-9A-Z]+)?TO-?([0-9]+)(.[0-9]+)?`
IF	`IF(\(([0-9a-zA-Z\.\(\)\+\*\/\^\-])+\)|ACCUMULATOROVERFLOW|QUOTIENTOVERFLOW|DIVIDECHECK|(\(SENSELIGHT[0-9]+\))|(\(SENSESWITCH[0-9]+\)))-?([0-9]+),-?([0-9]+)((,-?([0-9]+))?)+`
PAUSE	`PAUSE|PAUSE([0-9A-Z\.\(\)\+\*\/\^\-])+((,[0-9A-Z.\(\)\+\*\/\^\-]+)+)?`

FUNCTION	`FUNCTION([A-Z]([0-9A-Z]+)?)\((([0-9A-Z\.\(\)\+\*\/\^\-])+((,[0-9A-Z.\(\)\+\*\/\^\-]+)+)?)?\)`
COMPUTED GOTO	`GOTO[A-Z]([0-9A-Z]+)?,\(([0-9])+((,[0-9]+)+)?\)`
ASSIGNED GOTO	`GOTO\(([0-9])+((,[0-9]+)+)?\),[A-Z]([0-9A-Z]+)?`
PRINT	`PRINT([0-9A-Z\.\(\)\+\*\/\^\-])+((,[0-9A-Z.\(\)\+\*\/\^\-]+)+)?`
READ	`READ[A-Z]([0-9A-Z]+)?`
COMMON	`COMMON\/([A-Z]([0-9A-Z]+)?)?\/([A-Z]([0-9A-Z]+)?(((,[A-Z]([0-9A-Z]+)?)?))+)?`
CONTINUE	`CONTINUE`

Arithmetic Parser Grammar

The following section contains the Left-Right grammar for the Arithmetic Parser.

```
start -> expressions

expressions -> expressions top_level_expressions | top_level_expressions

expression -> expression + expression | expression - expression | expression *
expression | expression / expression | VARIABLE ( arguments ) | VARIABLE ( ) |
( expression ) | - INT | - FLOAT | INT | FLOAT | VARIABLE

arguments -> single_argument | expression , arguments

single_argument -> expression
```