High Level Assembler Plugin

Project specification

Michal Bali, Marcel Hruška, Peter Polák,
Adam Šmelko, Lucia Tódová

Supervisor: Miroslav Kratochvíl

Contents

1. Background and goals

The IBM High Level Assembler Language (HLASM) is still actively used commercially, even though it is a relatively old language. Its roots go back to the 1970s, when IBM made their first mainframes. Since then, the IBM assembler has been revised several times — the last version (which is the concern of this project) was released in 1992. Although it is hard to believe, a lot of the software that has been written in the language over the years is still actively used and maintained, mainly because of the conservative mainframe users and IBM's vendor lock-in.

Today, HLASM developers are forced to code in archaic terminals directly on the mainframe. Therefore, they spend a lot of time navigating around the code and the environment. For example, solely due to the fact that the user needs to navigate through plenty of terminal screens it takes around a minute just to get to a screen where it is possible to make a change in a file and recompile. For developers, it would be extremely useful to have an IDE plugin that would minimize contact with the mainframe terminal, could analyze the HLASM program, check its validity and make the code clearer by syntax highlighting.

The aim of this project is to improve HLASM programming experience, so that it can be compared to coding in modern programming languages, by providing instant code validity checks, advanced highlighting, code analysis, and all the functionality that a programmer currently takes for granted when writing code.

1.1 Related HLASM software

Naturally, a HLASM compiler[1] already exists, specifically the one from IBM. It is shipped as part of mainframe operating systems. Our team will be granted access to this compiler. During the implementation of language features, we will use this compiler as a reference and will try to mimic its error recognition capabilities.

Furthermore, Visual Studio Code Marketplace already has a plugin for HLASM language called ibm-assembler[2]. However, it provides only basic syntax highlighting implemented with textmate grammar, which has its limitations. We aim for writing a parser that would understand every aspect of the language and provide much more functionality.

[1] `https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.cbcux01/bpxa5as.htm`

[2] `https://marketplace.visualstudio.com/items?itemName=kelosky.ibm-assembler`

2. HLASM overview

Ordinary assembly languages consist solely of ordinary machine instructions. High-level assemblers generally extend them with features commonly found in high-level programming languages, such as control statements similar to *if, while, for* as well as custom callable macros.

IBM High Level Assembler (HLASM) satisfies this definition and adds other features which will be described in this section.

2.1 Syntax

HLASM syntax is similar to a common assembler, but due to historical reasons it has limitations, like line length limited to 80 characters (as that was the length of a punched card line).

2.1.1 Statement

HLASM program consists of a sequence of *statements*. A statement consists of four fields separated by spaces that can be split into more lines using continuations (see section 2.1.2). They are used to produce both compile-time code and run-time code (see section 2.2). The fields are:

- **Name field** — Serves as a place for named constants that are to be used in the code. This field is optional, but, when present, it must start at the begin column of a line.
- **Instruction field** — The only mandatory field represents the instruction that is executed. It must not begin in the first column, as it would be interpreted as a name field.
- **Operands field** — Field for instruction operands, located immediately after instruction field. Individual operands must be separated by a comma, and, depending on the specific instruction, can be either blank, in a form of an apostrophe separated string, or represented by a sequence of characters.
- **Remark field** — Optional, serves as inline commentary. Located either after the operands field, or, in case the operands are omitted, after the instruction field.

Listing 1 shows an example of a basic statement containing all fields.

Listing 1 An example statement.

```
name    instruction     operands              remark
.NOMOV       AGO     (&WH).L1,.L2,.L3     SEQUENTIAL BRANCH
```

Listing 2 Example program that uses the continuation for overflowing the line.

```
    OP1                                 REG12,REG07,REG04,REG00,REG01,REG11,Rx
                EG02
```

2.1.2 Continuations

Individual statements sometimes contain more than 80 characters, which does not fit to the historic line length limitations. Therefore, a special feature called *continuation* was introduced.

For this purpose the language specification defines four special columns:

- *Begin column* (default position: 1)
- *End column* (default position: 71)
- *Continuation column* (default position: 72)
- *Continue column* (default position: 16)

The begin column defines where the statements can be started.

The end column determines the position of the end of the line. Anything written to the right of it does not count as content of the statement, and is rather used as a line sequence number (see fig. 2.1).

The continuation column is used to indicate that the statement continues on the next line. For proper indication, an arbitrary character other than space must be written in this column. The remainder of the statement must then start on the continue column.

An example of an instruction where its last operand exceeded column 72 of the line can be seen in listing 2.

Some instructions also support the *extended format* of the operands. That allows the presence of a continuation character even when the contents of a line have not reached the continuation column (see listing 3).

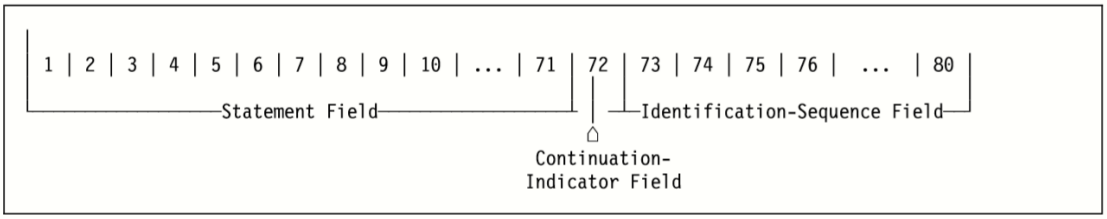

Figure 2.1: Description of line columns (source: HLASM Language Reference).

Listing 3 Extended instruction format.

```
          AIF   ('&VAR' FIND '~').A,     REMARK1                        x
                ('&VAR'  EQ  'L').B,     REMARK2                        x
                (T'&VAR  EQ  'U').C      REMARK3
```

2.2 Assembling

Having briefly described the syntax, now we describe the assembly process hidden behind HLASM.

We distinguish two types of processing: *Conditional Assembly (CA) processing* and *Ordinary processing*.

The ordinary processing works with *machine instructions* and *assembler instructions* (see section 2.2.1.2, section 2.2.1.3). The main purpose of CA processing (see section 2.2.2) is to generate statements for ordinary assembly.

2.2.1 Ordinary assembly

With help of machine and assembler instructions, the ordinary assembly processing is responsible for runtime behavior of the program. Ordinary assembly allows to produce code from traditional machine instructions, special-purpose assembler instructions and save values in *ordinary symbols*.

2.2.1.1 Ordinary symbols

In HLASM, an *ordinary symbol* is a named constant that can represent either a simple integer or an address. It can be defined by writing its name into the name field of a statement. Each ordinary symbol can only be defined once, and its value is constant. There are two kinds of ordinary symbols:

- An *absolute symbol* that simply has an integral value. It can be defined using a special assembler instruction.
- A *relocatable symbol* that represents an address in the resulting object code. It can be defined by writing the ordinary symbol name into the name field of a statement with a machine instruction and denotes the address of the instruction.

2.2.1.2 Machine instructions

Machine instructions represent the actual processor instructions executed during runtime. The assembler translates them into corresponding opcodes and processes their operands. That does not differ from traditional assemblers. In contrast to that, HLASM allows expressions to be passed as operands of the instructions. These expressions may use ordinary symbols and support integer and address arithmetic.

2.2.1.3 Assembler instructions

In addition to machine instructions, the HLASM assembler provides the *assembler instructions* (in other systems commonly termed as *directives*). They instruct the assembler

Listing 4 A sample program that shows that symbols can be used prior to their definition.

```
[01]              DS    CL(LEN)
[02]   ADDR       DS    CL(SIZE)
[03]
[04]   HERE       DS    0H
[05]   LEN        EQU   HERE-ADDR
[06]   SIZE       EQU   1
```

to make specific actions rather than to assemble opcodes. For example, they generate run-time data constants, create ordinary symbols, organize the resulting object code and generally affect how the assembler operates.

The assembler instructions include the following examples:

- **ICTL** — Changes values of the previously described line columns (i.e. begin column may begin at column 2 etc.).
- **DC**, **DS** — Reserves space in object code for data described in operands field and assembles them in place (i.e. assembles float, double, character array, address etc.).
- **EQU** — Defines ordinary symbols.
- **COPY** — Copies text from a specified file[1] and pastes it in place of the instruction. It is very similar to the C preprocessor `#include` directive.
- **CSECT** — Creates an executable control section. Serves as the beginning of a machine instruction sequence and as the start of relative addressing.

2.2.1.4 Ordinary symbols resolution

All the assembler instructions and ordinary symbols must be resolved before the assembler writes the final object file. The HLASM language supports forward declaration of ordinary symbols, so the assembly may be quite complicated. Consider an example in listing 4. When the instruction in line 1 is seen for the first time, it is impossible to determine its length, because the symbol `LEN` is not defined yet [2]. The same applies to the length of the instruction in the second line. Furthermore, it is also impossible to determine the exact value of relocatable symbols `ADDR` and `HERE`, because of the unknown length of the preceding instructions.

In the next step, `LEN` is defined, but it cannot be evaluated, because the subtraction of addresses `ADDR` and `HERE` is dependent on the unknown length of instruction on second line and therefore on the symbol `SIZE`. The whole program is resolved only when the assembly reaches the last line, which defines the length of instruction `02`. Afterwards, it is possible to resolve `LEN` and finally the length of instruction `01`.

The dependency graph created by these principles can be arbitrarily deep and complicated, however it must not contain cycles (a symbol must not be transitively dependent on itself).

[1] Path to the folder of the file is passed to assembler before the start of assembly

[2] Character L with an expression in parentheses in DS operand of type C specifies how many bytes should be reserved in the program.

2.2.2 Conditional assembly

On top of machine and assembler instructions, the HLASM assembler offers the conditional assembly. The user may think of it as a macro-language built above traditional assembler. It alters textual representation of the source code and selects which lines will be processed next. Conditional assembly instructions may define *variable symbols* which can be used in any statement to alter its meaning. Moreover, it is possible to define *macros* — reusable pieces of code with parameters.

2.2.2.1 Variable symbols

Variable symbols serve as points of substitution or information holders.

When they occur in a statement, they are substituted by their value to create a statement processable by ordinary assembly. For example, in this manner, a user can write a variable symbol in an operation field of a statement and generate any instruction that can be a result of a substitution.

Variable symbols also have notion of their type — they can be defined either as an integer, a boolean or a string. CA instructions gather this information for different sorts of conditional branching.

2.2.2.2 CA instructions

CA instructions are not assembled into object code. Rather, they select which instructions will be processed by assembler next.

There are instructions capable of conditional and unconditional branching. HLASM provides a variety of built-in binary or unary operations on variable symbols, which can create complex conditional expressions. This is important in HLASM, as the user can alter flow of instructions that will be assembled into executable program.

Another subset of CA instructions operates on variable symbols. With them, the user can define variable symbols locally or globally, assign or update their values.

2.2.2.3 Macros

A *macro* is a structure consisting of a *name*, *input parameters* and a *body*, which is a sequence of statements. When a macro is called in a HLASM program, each statement in its body is executed. Both nested and recursive calls of macros are allowed. Macro body can contain CA instructions or even a sequence of instructions generating another macro definition. With the help of variable symbols, HLASM macros have the power to create custom, task specific macros.

An example of simple HLASM program with the description of its statements is shown in listing 5.

In lines `01-04`, we see a *macro definition*. It is defined with a name `GEN_LABEL`, variable `NAME` and contains one instruction in its body, which assigns the current address to the label in `NAME`.

In line `06`, the *copy instruction* is used, which includes the contents of the `REGS` file.

Line `08` establishes a start of an executable section `TEST`.

In line `09`, an integer value is assigned to a variable symbol `VAR`. The value is the length attribute of previously non-defined constant `DOUBLE`. The assembler looks for the definition of the constant to properly evaluate the conditional assembly expression. In the

Listing 5 An example of an artificial HLASM program.

```
name        operation   operands

[01]                    MACRO
[02]      &NAME         GEN_LABEL
[03]      &NAME         EQU           *
[04]                    MEND
[05]
[06]                    COPY          REGS
[07]
[08]      TEST          CSECT
[09]      &VAR          SETA          L'DOUBLE
[10]                    AIF           (&VAR EQ 4).END
[11]      LBL1          GEN_LABEL
[12]                    LR            3,2
[13]                    L             8
[14]      LBL2          GEN_LABEL
[15]      LEN           EQU           LBL2-LBL1
[16]                    DC            (LEN)C'HELLO'
[17]      DOUBLE        DC            H'-3.729'
[18]      .END          ANOP
[19]                    END
```

next line, there is CA branching instruction `AIF`. If value of `VAR` equals to 4, next lines are skipped and assembling continues on line `18`, where branching symbol `.END` is located. In that case, all the text between the `AIF` and `.END` is completely skipped — it does not even have to be valid HLASM code.

Lines `12-13` show examples of machine instructions that are directly assembled into object code. Lines `11` and `14` contain examples of macro call.

In line `15`, the constant `LEN` is assigned the difference of two addresses, which results in absolute ordinary symbol. This value is next used to generate character data.

Instruction `DC` in line `17` creates value of type double and assigns its address to ordinary symbol `DOUBLE`. This constant also holds information about length, type and other attributes of the data.

`ANOP` is an empty assembler action which defines the `.END` symbol and line `19` ends the assembling of the program.

Although CA processing may act like a text preprocessing, it is still interlinked with ordinary processing. CA has mechanics that allows the assembler to gather information about statements that are printed during the processing. It can access values created in ordinary assembly and use them in conditional branching. CA is able to lookup constants that are not yet defined prior to the currently processed statement. Moreover, during ordinary assembly, names of these instructions can be aliased.

To sum up, CA processing has variables to store state of compilation and CA instructions for conditional branching. Hence, it is Turing-complete while still evaluated during compile-time.

3. Project scope

This chapter reviews the specific goals of our project. All the features below have been discussed with professional HLASM developers and have been agreed on by the company.

This project aims to produce a VS Code extension, downloadable from the Market Place. The extension contains all executables/libraries that are needed for the project to work correctly on the most popular platforms. No other prerequisites should be required.

Modularity of the software is another important requirement.

The project implementation provides 2 kinds of API: a complex one that mirrors the Language Server Protocol[1] (LSP) specification and a simple one that accepts a text along with a dependency-resolver object and returns diagnostics.

The language server implements the LSP standard, hence it can be easily reintegrated within other IDEs such as Eclipse Che.

3.1 Supported language features

This part provides a brief overview of the parts of HLASM that should be included in the final software.

Syntax Parser The parser recognizes the syntax of HLASM and processes it into predefined structures.

High-level interpretation The library interprets high-level parts of the assembler. Following are the conditional assembly instructions for code generation and macro expansion:

- AIF
- AGO
- MACRO, MEND, MEXIT
- ACTR
- SETA, SETB, SETC
- ANOP
- LCLA, LCLB, LCLC
- GBLA, GBLB, GBLC
- AEJECT

[1] `https://microsoft.github.io/language-server-protocol/`

- ASPACE

and assembler instructions for code layout determination:

- EQU
- DC
- DS
- DSECT, CSECT, RSECT
- LOCTR

Code Validation The back-end semantically and syntactically checks all instructions (including machine instructions) for correct operand format usage. However, it does not analyze the run-time register and memory values, as there is no machine instruction interpretation.

Dependency Search Usage of external files in HLASM is a highly common phenomenon. On mainframes, HLASM programs are built according to its JCL[2] file, which contains a list of libraries. When the compiler encounters an undefined instruction or a `COPY` instruction, it does a top-down search through the whole list. Both ways of invoking a dependency search are demonstrated in listing 6. As a large portion of the programs use the same libraries, defining these JCL files gets repetitive. Therefore, the build and source management system Endevor creates an abstraction above the libraries and groups them into `processor groups`. As a result, JCL offers an option to identify the libraries to be included by their processor group's ID instead of listing them all manually.

We adapt this system to our needs and define 2 configuration files. The first one mirrors the behavior of Endevor and defines the processor groups. The second one matches the source codes to these processor groups.

Continuation Handling Fixed-size lines are another aspect of HLASM that needs to be handled. They make the parsing more difficult, as the position of the continuation character may vary.

We also add a continuation handling option to the IDE, which mostly consists of non-movable continuation characters, i.e. if the user types in front of the continuation character, it stays in place.

Macro Tracer To trace the code generation, the user can step through the code while watching the contents of variables and the call stack using `Macro tracer`. We implement `Debug Adapter Protocol` so that the process resembles standard debugging. This tool is extremely useful as tracing the macro expansions manually gets tedious quickly.

[2] Job Control Language, instructs the system how to run a specific task

Listing 6 An example of both ways the HLASM program may invoke dependency search.

```
        MAC1        1,1
        COPY        COPY1
```

3.2 Supported LSP features

This section demonstrates the possible uses of the extension on the client side. LSP provides a list of well-defined features. The project implements the following:

- `Go to definition` command for all symbols, macro definitions and copy members[3]
- `Find all references` command for all symbols, macro definitions and copy members
- Completion for instructions, defined symbols and macros
- Mouse-over tooltip (hover) for symbol attributes, their locations, contents and other useful information depending on the symbol type
- Diagnostics for syntax and semantic errors and warnings
- Server-side Highlighting for all symbols which is our custom extension of LSP

The highlighting is not a standard part of the LSP, nonetheless it is a needed addition. Due to the complexity of HLASM, a typical syntax highlighting is not sufficient. Consider following examples:

- The language server recognizes operand formats of different instructions. The most simple example is the `SAM31` instruction, which does not have any operands, while the `LR` instruction takes two. So the identifier right after `SAM31` is colored blue as a remark.

```
1       SAM31  REMARK
2       LR     1,1    REMARK
```

- The code skipped by the conditional assembly is not colored and stays white.

```
1       AGO  .HERE
2       J    SYMBOL
3  .HERE  AIF
```

[3] Copy, along with macro expansion, is a mean to include another external file, invoked by `COPY` instruction. Comparing to macro, copy does not neither need to start nor end with any specific instruction and the invoking `COPY` instruction is simply replaced by the COPY file's contents.

4. Architecture

The architecture is based on the way modern code editors and IDEs are extended to support additional languages. We chose to implement Language Server Protocol [1] (LSP), which is supported by a majority of contemporary editors.

In LSP, the two parties that communicate are called a *client* and a *language server*. A simple example is displayed in fig. 4.1 The client runs as a part of an editor. The language server may be a standalone application that is connected to the client by a pipe or TCP. All language-specific user actions (for example Go to definition command) are transformed into standard LSP messages and sent to the language server. The language server then analyzes the source code and sends back a response, which is then interpreted and presented to the user in editor-specific way. This architecture makes possible to only have one LSP client implementation for each code editor, which may be reused by all programming languages. And vice versa, every language server may be easily used by any editor that has an implementation of the LSP client.

To add support for HLASM, we have to implement the LSP language server and write a thin extension to an editor, which will use an already existing implementation of the LSP client. To implement source code highlighting, we need to extend the protocol with a new notification. This notification will be used for transferring information from language server to VS Code client, which is extended to highlight code in editor based on the incoming custom notifications.

Here, we further decompose the project into smaller components and describe their relations. The two main components are the parser library and the language server — an executable application that uses the parser library. The overall architecture is pictured in fig. 4.2.

4.1 Language server description

The responsibility of the language server component is to maintain the LSP session, convert incoming JSON messages and use the parser library to execute them. The functionality includes:

- reading LSP messages from standard input or TCP and writing responses
- parsing JSON RPC to C++ structures, so they can be further used
- serializing C++ structures into JSON, so it can be sent back to the client
- implementing asynchronous request handling: e.g. when user makes several consecutive changes to a source code, it is not needed to parse on every change

[1] `https://microsoft.github.io/language-server-protocol/`

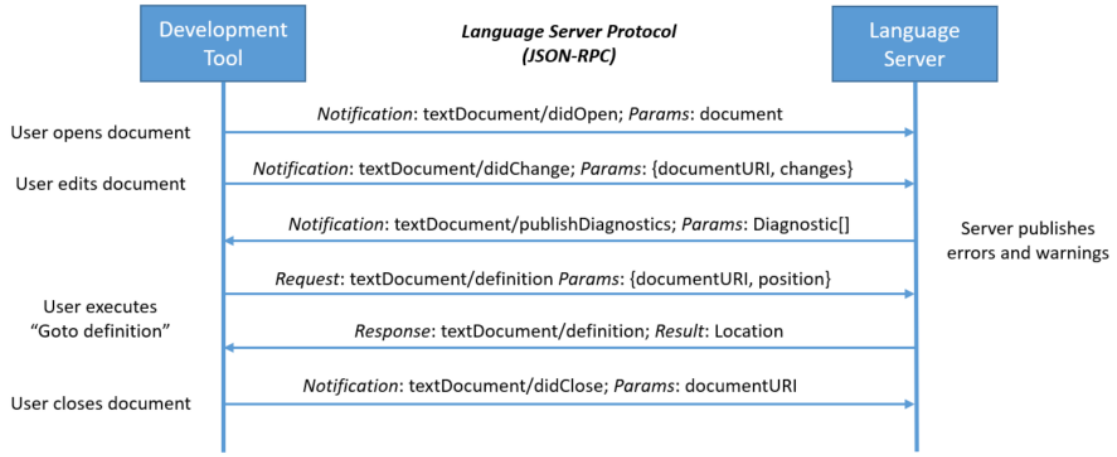

Figure 4.1: LSP session example. (source: `https://microsoft.github.io/language-server-protocol/overview`)

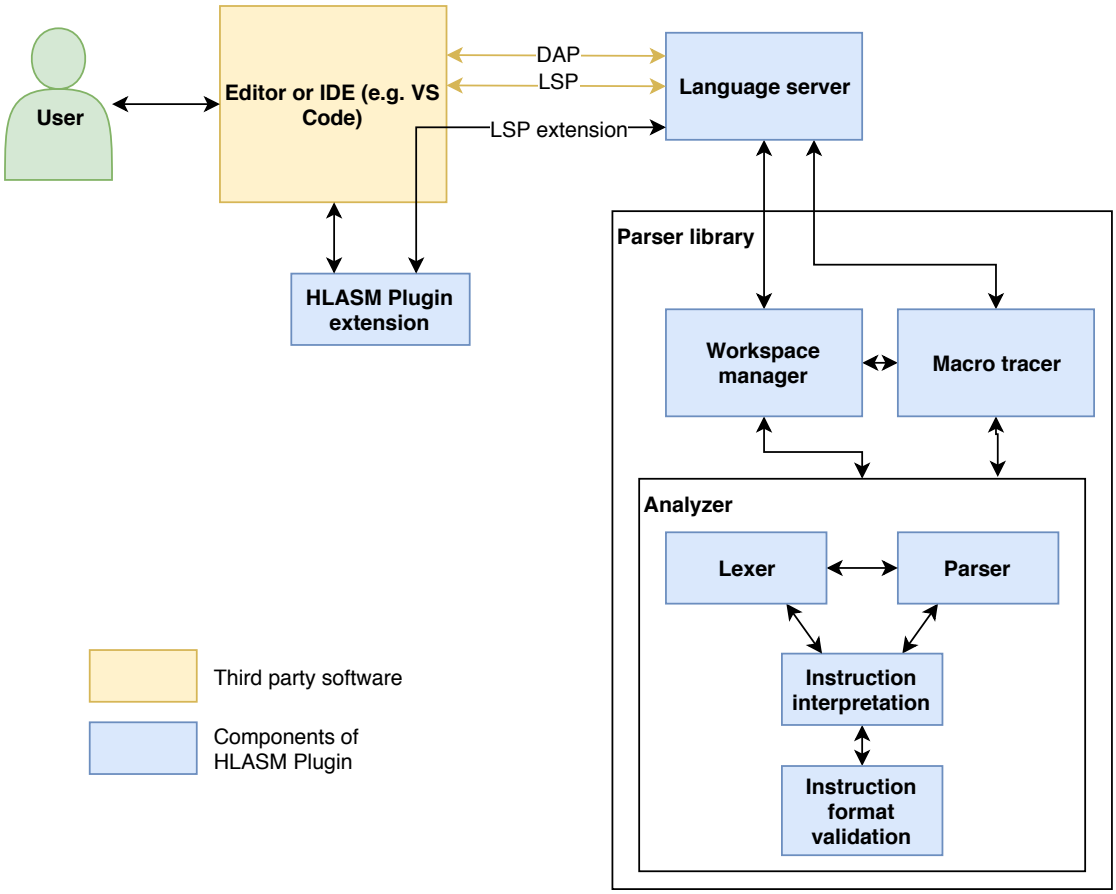

Figure 4.2: The architecture of HLASM Plugin

4.2 Parser library description

Parser library is the core of the project — it encapsulates the analyzer, which provides all parsing capabilities, and workspace manager, which keeps track of open files in the editor and manages their dependencies. It has to keep the representation of workspaces and files in the parser library exactly the same as the user sees in the editor. It also starts the analyzer when needed, manages workspace configuration and provides external macro and copy libraries to analyzer.

4.2.1 Parser library API

The parser library API is based on LSP — every relevant request and notification has a corresponding method in the parsing library.

Firstly, the API has to implement the LSP notifications that ensure the editor state synchronization. Apart from working with individual files, the LSP also supports workspaces. A workspace is basically just a folder which contains related source codes. The LSP also supports working with multiple workspaces at the same time. We use it when searching for dependencies of HLASM source codes (macros, and copy files).

The parser library needs to have the exact contents of all files in opened workspaces. To achieve that, there is a file watcher running in the LSP client that notifies the server when any of the HLASM source files is changed outside of editor. For example, when a user deletes an external macro file, the parser library should react by reporting that it cannot find the macro.

The list of necessary editor state synchronization notifications follows:

- Text synchronization notifications (didOpen, didChange, didClose) which inform the library about files that are currently open in the editor and their exact contents.
- DidChangeWorkspaceFolders notification which informs the library when a workspace has been opened or closed.
- DidChangeWatchedFiles notification

Secondly, the API has to implement the requests and notifications that provide the parsing results:

- publishDiagnostics notification. A diagnostic is used to indicate a problem with source files, such as a compiler error or a warning. The parser library provides a callback to let the language server know that diagnostics have changed.
- Callback for highlighting information provision.
- Language feature requests (definition, references, hover, completion), which provide information needed for proper reaction of the editor on user actions.

4.2.2 Analyzer

The analyzer is able to process a single HLASM file. The processing includes:

- recognition of statements and their parts (lexing and parsing)
- interpretation of instructions that should be executed in compile time

- a check whether the HLASM source code is well-formed
- reporting of problems with the source by producing LSP diagnostics
- providing highlighting and LSP information

A HLASM file may have dependencies — other files that define macros or files brought in by the COPY instruction. The dependencies are only discovered during the processing of files, so it is not possible to provide the files beforehand. The analyzer gets a callback that would find a file with specified name, parse its contents and return it as list of parsed statements.

To sum up, the analyzer has a pretty simple API: it takes the contents of a source file by common string and a callback that can parse external files with specified name. It provides a list of diagnostics linked to the file, highlighting, list of symbol definitions, etc.

The analyzer is further decomposed into 4 components.

4.2.2.1 Lexer

Lexer's task is to read source string and break it into tokens — small pieces of text with special meaning. The most important properties of the lexer:

- each token has location in the source text
- has the ability to check whether all characters are valid in the HLASM source
- has the ability to jump in the source file backward and forward if necessary (for implementation of instructions like AGO and AIF). Because of this, it is not possible to use any standard lexing tool and the lexer has to be implemented from scratch.

4.2.2.2 Parser

Parser component takes the stream of tokens the lexer produces and recognizes HLASM statements according to the syntax. To accomplish this, a parser generator tool Antlr 4 [2] is used.

The input to Antlr is a grammar (written in antlr-specific language) that specifies the syntax of HLASM language and generates source code (in C++) for a recognizer, which is able to tell whether input source code is valid or not. Moreover, it is possible to assign a piece of code that executes every time a grammar rule is matched by the recognizer to further process the matched piece of code.

4.2.2.3 Instruction interpretation

Results of the parser component are further analyzed in the processing component. Its most important capabilities are:

- Interpretation of CA instructions, which results in modifying the lexer state (moving back and forth in the input file).
- Substitution of variable symbols. After the substitution, the statement must be reparsed in the lexer and the parser, since the substitution may completely change its meaning.

[2] `https://www.antlr.org`

- Interpretation of assembler instructions
- Ordinary symbols resolution
- MACRO and COPY expansion.

4.2.2.4 Instruction format validation

After a statement is fully processed and all operands of each instruction are known, the statement needs to be checked for errors. There are over 2000 machine instructions with variable number of operands and various restrictions on those operands — some of them take only positive numbers, numbers that are in specific range or are limited to addresses only. The core of the component is a great table that describes all the instructions and their operands.

The API of the validation component is simple: it takes an instruction and list of its operands and returns a list of warnings and errors in the form of LSP diagnostics.

4.3 Client-side VS Code extension

The VS Code extension component ensures seamless integration with the editor. Its functions are:

- to start the HLASM language server and the LSP client that comes with VS Code, and create a connection between them
- to implement extension of the LSP protocol to enable server-side highlighting. The extended client parses the information from the server and uses VS Code API to actually color the text in the editor.
- to implement continuation handling — when the user types something in front of the continuation character, it should stay in place.

4.4 Macro tracer

The macro tracer enables the user to trace the compilation of HLASM source code in a way similar to common debugging. This is the reason why we chose to implement the Debug Adapter Protocol [3] (DAP). It is very similar to LSP, so most of the code implementing LSP in the language server component may be reused for both protocols.

The language server component communicates with the macro tracer component in the parser library. Its API mirrors the requests and events of DAP. The most important features to implement are:

- launch, continue, next, stepIn and disconnect requests, which allow the user to control the flow of the compilation
- SetBreakpoints, which transfers the information about breakpoints that the user has placed in the code

[3] `https://microsoft.github.io/debug-adapter-protocol/`

- Threads, StackTrace, Scopes and Variables requests to allow the DAP client to retrieve information about the current processing stack (stack of nested macros and copy instructions), available variable symbols and their values
- stopped, exited and terminated events to let the DAP client know about state of traced source code

The macro tracer communicates with the workspace manager to retrieve the content of the traced files. Afterwards, it starts analyzing the source file in a separate thread and gets callbacks from the analyzer before each statement is processed. In the callback, the tracer puts the thread to sleep and waits for user interaction. During this time, it is possible to retrieve all variable and stack information from the processing to display it to the user.

5. Project execution

5.1 Team

This team consists of five members:

- Michal Bali,
- Marcel Hruška,
- Peter Polák,
- Adam Šmelko,
- Lucia Tódová.

Each team member is responsible for delivering work packages assigned to him or her (see section 5.3).

5.2 Team management

For managing the project and team members we use a visual process management system Kanban. Additionally, the project is developed within the environment of Broadcom Inc., which supplies additional project management and supervision. We are attempting to follow the Agile software development guidelines — our team meets every week with our Broadcom supervisor at stand-ups and discusses the current status of particular tasks with their assignees, reviews progress and plans work for the next week.

Slack is used for dynamic communication between team members.

5.3 Project timeline

The project was split into several milestones and work packages, specified closer in table 5.1. The implementation of the whole project was planned to be accomplished within nine months.

The project has been already worked on for four months, which has resulted in completion of the initial milestones up to a large portion of Preview. Therefore, there is now a working prototype of the HLASM plugin, and some of the presented work packages are already finished.

Since most of the architecture, design and planning questions have already been answered during the development of the prototype, we do not expect any serious technical

or architectural issues to surface at this point. Additionally, the company support from Broadcom Inc. prevents financing and motivation issues. We therefore assume that it is very probable that the project will be completed as planned.

The work packages have been assigned to individual team members based on long-term planning. The assignments and corresponding deadlines are listed in table 5.1, and summarized by a diagram in fig. 5.1.

Table 5.1: The milestones and work packages organisation

Ms.	Milestone description	WP	Work package description	Assignees
M1	**Research and analysis** Deadline: month 1	WP1	**HLASM language analysis** Analysis of HLASM specification, available code and discussion with HLASM users.	Adam Marcel
		WP2	**Parser libraries research** Research and comparison of contemporary lexical and parser libraries (Bison, ANTLR, ...).	Peter
		WP3	**IDEs research** Research of available IDEs to which would be the HLASM language support integrated.	Lucia Michal
M2	**HLASM syntax support** At the second milestone, the plugin is able to parse HLASM syntax, which is shown to the user with server-side highlighting. There is a working implementation of LSP on the server side, which communicates to the VS Code LSP client. The LSP is extended to transfer server-side highlighting information. Deadline: month 2	WP4	**Lexer**	Lucia Peter
		WP5	**Parser**	Adam Marcel
		WP6	**LSP implementation**	Michal
		WP7	**Server-side highlighting**	Marcel

M3	**Detailed specification** Deadline: month 3	WP8	**Detailed specification**	all
M4	**Preview** The output of this milestone is a working demo and its presentation in Broadcom. The parser is able to interpret conditional assembly instructions and expand macros defined within the same source file. It is able to present problems with HLASM source code via LSP diagnostics, and LSP features like Hover or Go to definitions are working with variable symbols. Deadline: month 4	WP9	**Macro tracer POC** Proof of concept of the macro tracer by implementation of the Debug Adapter Protocol (see section 4.4).	Michal
		WP10	**Validation of assembler instructions operands**	Lucia
		WP11	**CA instructions** Interpretation and validation of the conditional assembly instructions.	Adam
		WP12	**CA expressions** Evaluation of the conditional assembly expressions.	Peter
		WP13	**Macro expansion** Only macros defined within the same file required.	Adam
		WP14	**Conditional assembly LSP features** Hover, Go to definition and Find all references for variable symbols	Marcel

M5	**Multiple files parsing** The plugin is able to parse macros from separate files and is able to interpret the COPY instruction. The user experience is improved by continuation handling. Deadline: month 5	WP15	**Machine expressions** Evaluation of expressions that are used in assembler and machine instructions.	Michal
		WP16	**Continuation handling** VS Code extension functionality to improve working with continuations — when typing, the continuation at the end of the line should stay in place.	Marcel
		WP17	**External files parsing** Interpretation of COPY instruction, macro expansion from separate files.	Adam
		WP18	**File dependencies** Configuration of processor groups and dependency search (see section 3.1)	Michal
M6	**Ordinary assembly implementation** The plugin is able to interpret the EQU and DC instructions and thus evaluate most of ordinary symbols (see section 2.2.1.4). The acquired values are then used to validate machine instruction operands. Additionally, the user can Go to definition on ordinary symbols and show their values using mouse hover tooltips. Deadline: month 7	WP19	**Validation of assembler instructions operands**	Lucia
		WP20	**DC instruction** parsing, validation and length of data definition operand	Michal Peter
		WP21	**Ordinary symbols** ordinary assembly implementation	Peter Adam
		WP22	**Ordinary LSP features** implementation of support for ordinary symbols	Marcel

M7	**Finalization** We reserve some time to polish user experience, finish all components and implement smaller (but important) HLASM features that were not explicitly planned. Deadline: month 8	WP23	**Finalization**	all
M8	**Feature testing** After the M8 milestone, the software should be well tested, stable and able to run seamlessly on any major platform. Deadline: month 9	WP24	**Testing**	all
		WP25	**Multi-platform deployment** Deployment on Windows, Linux and MacOS	Michal
		WP26	**Code coverage**	Lucia
		WP27	**Benchmark** Creation of a performance measuring tool that measures how much time the parser needs to analyze a file. It also measures number of processed lines.	Marcel
M9	**Documentation** Deadline: month 9	WP28	**Documentation**	all
M10	**Final Presentation** Deadline: month 9	WP29	**Final presentation**	all

Figure 5.1: Work packages overview